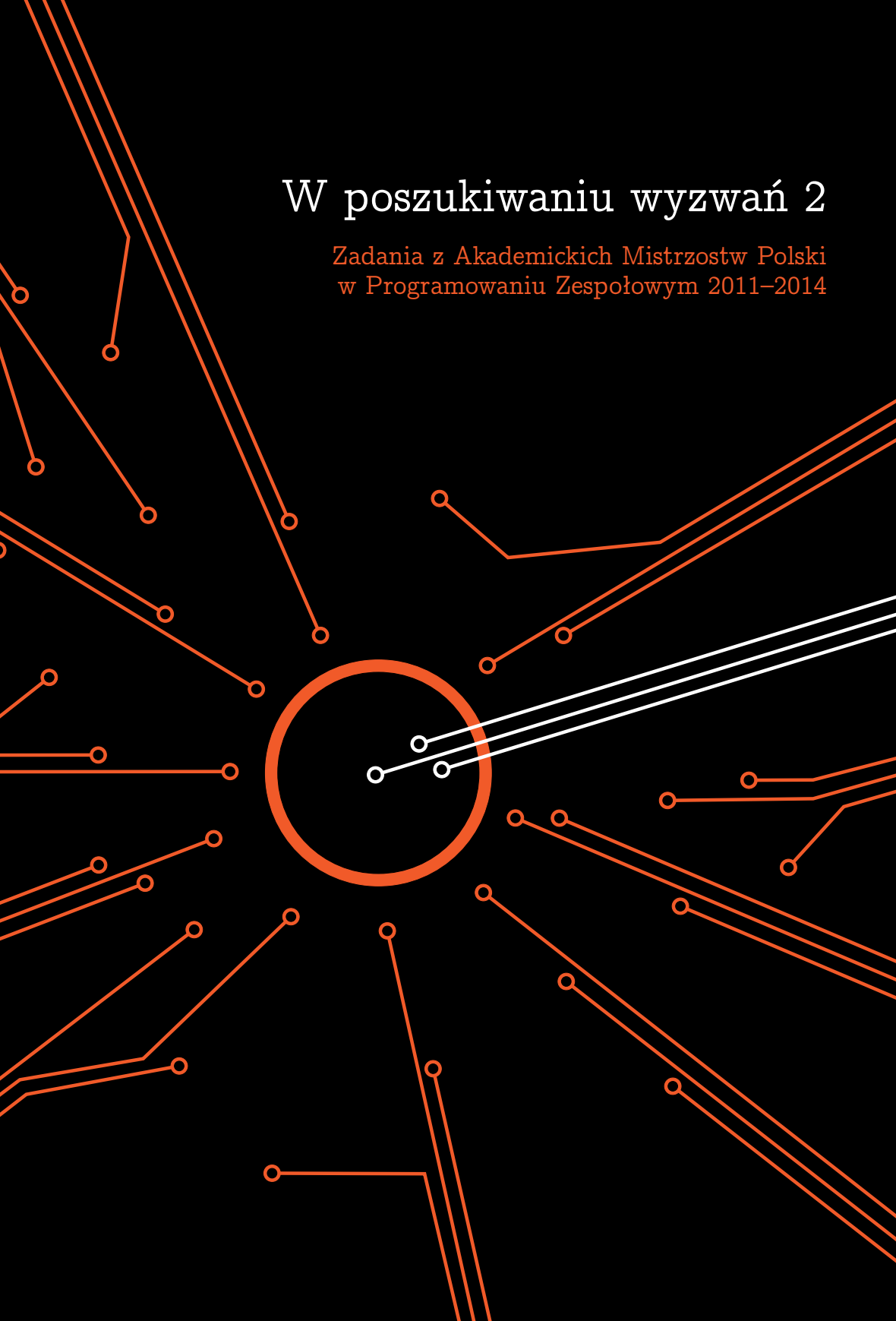
W poszukiwaniu wyzwań 2

Zadania z Akademickich Mistrzostw Polski w Programowaniu Zespołowym 2011–2014



Wydanie drugie (wersja 2.1)

Redakcja i skład
Krzysztof Diks, Tomasz Idziaszek, Jakub Łącki, Jakub Radoszewski

Autorzy opracowań
Szymon Acedański, Tomasz Idziaszek, Adam Karczmarz, Tomasz Kociumaka, Eryk Kopczyński, Jakub Łącki, Jakub Radoszewski

Autorzy zadań i programów wzorcowych
Szymon Acedański, Marcin Andrychowicz, Karol Cwalina, Marek Cygan, Dawid Dąbrowski, Krzysztof Diks, Paweł Gawrychowski, Tomasz Idziaszek, Adam Karczmarz, Tomasz Kociumaka, Eryk Kopczyński, Karol Kurach, Jakub Łącki, Jakub Pachocki, Jakub Pawlewicz, Jakub Radoszewski, Wojciech Rytter, Bartosz Szreder, Wojciech Śmietanka, Jacek Tomasiewicz, Wojciech Tyczyński, Tomasz Waleń

Wstęp

Oddajemy w ręce Czytelników opisy rozwiązań zadań z Akademickich Mistrzostw Polski w Programowaniu Zespołowym w latach 2011–2014. Te cztery edycje Akademickich Mistrzostw Polski w Programowaniu Zespołowym (w skrócie AMPPZ) zostały zorganizowane przez Wydział Matematyki, Informatyki i Mechaniki oraz Wydział Zarządzania Uniwersytetu Warszawskiego. Sponsorem głównym zawodów był PKO Bank Polski, a Fundacja PKO Banku Polskiego wsparła merytoryczne przygotowanie Mistrzostw oraz fundowała stypendia najlepszym studentom.

Sukces każdych zawodów w programowaniu zespołowym zależy przede wszystkim od odpowiedniego doboru zadań konkursowych oraz sprawnego przeprowadzenia konkursu. Zestaw zadań na zawody powinien być dobrany tak, by dawać szansę rozwiązania co najmniej jednego zadania każdej startującej drużynie, a zarazem pozwolić wyłonić prawdziwych mistrzów. Zadania powinny być oryginalne, zaciekawiać i stymulować do doskonalenia swoich umiejętności wszystkich, którzy sobie z nimi nie poradzili podczas zawodów. Nad poziomem merytorycznym AMPPZ w latach 2011–2014 czuwało trzech młodych naukowców z Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego — Tomasz Idziaszek, Jakub Łącki oraz Jakub Radoszewski — wspieranych przez liczną rzeszę studentów, doktorantów i pracowników Wydziału. Na potrzeby AMPPZ stworzono 44 oryginalne zadania. Przygotowanie każdego z nich wymagało napisania rozwiązań wzorcowych oraz alternatywnych, a także odpowiedniego dobrania zestawu testów sprawdzających jakość rozwiązań zawodniczych.

Zawody w programowaniu zespołowym nie mogłyby się odbyć bez odpowiedniego zaplecza technicznego: komputerów połączonych w wydajną sieć oraz specjalistycznego oprogramowania konkursowego, które na bieżąco oceniało rozwiązania nadsyłane przez zawodników. Twórcami tego oprogramowania, wykorzystywanego także w wielu innych konkursach programistycznych organizowanych przez Uniwersytet Warszawski, są Szymon Acedański oraz jego zespół z Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego.

Niniejsza książka zawiera szczegółowe dyskusje rozwiązań zadań z AMPPZ w latach 2011–2014, napisane przez wybitnych młodych naukowców pracujących w dziedzinie algorytmiki, którzy odnieśli także znaczące sukcesy (jako zawodnicy i organizatorzy) w zawodach w programowaniu zespołowym, na szczeblu krajowym i międzynarodowym. Opisy wzorcowych rozwiązań powstały w ramach projektu Mistrzowie Algorytmiki realizowanego w programie Generacja Przyszłości, którego celem było merytoryczne przygotowywanie studentów z Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego do udziału w zawodach w programowaniu zespołowym.

W roku 2015 wydaliśmy te opisy rozwiązań w postaci drukowanej, a teraz udostępniamy je w postaci elektronicznej, by służyły całej społeczności algorytmicznej w Polsce. Co więcej, przygotowujemy również angielską wersję tej publikacji. Każdy, kto chciałby się zmierzyć z tym zadaniami, rozwiązując je samodzielnie, może skorzystać z portalu `szkopul.edu.pl` (linki do poszczególnych zadań na portalu podane są również w środku książki).

Zadania z AMPPZ to doskonały materiał treningowy dla każdego, kto chciałby doskonalić swoje umiejętności algorytmiczno-programistyczne lub dla własnej satysfakcji zgłębiać arkana algorytmiki. Aby każdy Czytelnik mógł łatwo znaleźć coś dla siebie, za pomocą gwiazdek oznaczyliśmy poziom trudności każdego zadania. Jedna gwiazdka oznacza zadania najprostsze, zaś przy najtrudniejszych zadaniach umieszczone zostały cztery gwiazdki.

Zapraszamy do lektury książki, życząc jednocześnie wspaniałej intelektualnej przygody.

Podziękowania

Akademickie Mistrzostwa Polski w Programowaniu Zespołowym w latach 2011–2014, zorganizowane w Warszawie przez Uniwersytet Warszawski, były wielkimi przedsięwzięciami organizacyjnymi, wymagającymi niestandardowych rozwiązań logistycznych, technicznych i merytorycznych z zakresu informatyki. Wiele instytucji i osób przyczyniło się do sukcesu tych przedsięwzięć. Słowa wdzięczności kieruję do wszystkich, którzy wnieśli swój wkład w powodzenie zawodów. Jednocześnie w sposób szczególny chciałbym podziękować osobom najbardziej zaangażowanym w przygotowanie Mistrzostw.

Gorące słowa podziękowania kieruję do Pana Zbigniewa Jagiełły, Prezesa PKO Banku Polskiego, oraz Pani Urszuli Kontowskiej, Prezesa Fundacji PKO Banku Polskiego, za to, że rozumieją znaczenie tego rodzaju konkursów dla rozwoju polskiej informatyki i jej promocji w świecie. PKO Bank Polski jako sponsor główny zawodów w Warszawie oraz drużyn Uniwersytetu Warszawskiego przyczynił się znacząco do umocnienia rangi samych Mistrzostw oraz sukcesów Polaków w międzynarodowych konkursach programistycznych.

Mistrzostwa zostały przeprowadzone na Uniwersytecie Warszawskim z wykorzystaniem uniwersyteckiej infrastruktury. Dziękuję władzom Uniwersytetu za życzliwość i wsparcie w przeprowadzeniu tak złożonego organizacyjnie przedsięwzięcia. W szczególności słowa podziękowania kieruję do Ich Magnificencji Rektorów Uniwersytetu Warszawskiego pani prof. dr hab. Katarzyny Chałasińskiej-Macukow, Rektora UW w latach 2005–2012, oraz pana dr. hab. Marcina Pałysa,

Rektora UW w latach 2012–2016, jak również do prorektorów, profesorów Tadeusza Tomaszewskiego i Alojzego Nowaka. Duże zrozumienie i pomoc okazali dziekani wydziałów bezpośrednio organizujących zawody: Wydziału Matematyki, Informatyki i Mechaniki — profesorowie Stanisław Betley i Andrzej Tarlecki oraz Wydziału Zarządzania — profesorowie Alojzy Nowak i Jerzy Turyna.

Zawody odbywały się na Wydziale Zarządzania Uniwersytetu Warszawskiego. Każdorazowo sala gimnastyczna Wydziału była przekształcana w wielkie laboratorium komputerowe wyposażone w kilkadziesiąt stanowisk komputerowych połączonych w wydajną sieć, z serwerami i drukarkami na zapleczu oraz możliwością przeprowadzania transmisji telewizyjnych i internetowych. Konkursy wymagały specjalistycznego, dedykowanego oprogramowania, które umożliwiało ich przeprowadzanie w czasie rzeczywistym. Bez żadnej przesady można stwierdzić, że techniczna strona zawodów była zabezpieczona przez światowej klasy fachowców, którzy pracowali pod kierownictwem Marka Mossakowskiego i Jerzego Rolewicza z Wydziału Zarządzania (przygotowanie sali, zabezpieczenie energetyczne, sieć komputerowa) oraz Szymona Acedańskiego z Wydziału Matematyki, Informatyki i Mechaniki (dedykowane oprogramowanie konkursowe). Wszystkim pracującym nad zapleczem technicznym zawodów wyrażam słowa najwyższego uznania.

Podobnie do technicznej, tak i merytoryczna część zawodów stała na najwyższym światowym poziomie. Słowa podziwu i wdzięczności za wykonaną pracę kieruję do Tomasza Idziaszka, Jakuba Łąckiego i Jakuba Radoszewskiego, którzy wraz z liczną rzeszą studentów, doktorantów i pracowników Wydziału Matematyki, Informatyki i Mechaniki włożyli tytaniczny wysiłek w przygotowanie zadań konkursowych i samo przeprowadzenie zawodów.

Na koniec, ale nie mniej gorąco, chciałbym podziękować osobom, które bezpośrednio pomagały mi w koordynowaniu przygotowań i przeprowadzeniu zawodów w latach 2011–2014. Praca z nimi była prawdziwą przyjemnością. Dziękuję panu profesorowi Janowi Madeyowi za to, że wprowadził mnie w świat konkursów programistycznych i szefował razem ze mną zawodom w Warszawie. Dziękuję Rafałowi Sikorskiemu, prawnikowi, który polubił informatykę, za wszelką codzienną pomoc organizacyjną. Jestem także wdzięczny Krzysztofowi Ciebierze, na którego zawsze mogłem liczyć i za jego, często bezcenne, rady dotyczące organizacji zawodów i rozwiązań technicznych.

Krzysztof Diks

O ZAWODACH W PROGRAMOWANIU ZESPOŁOWYM

Konkursy w programowaniu zespołowym sprawdzają wiedzę i umiejętności potrzebne w pracy każdego programisty. Praca algorytmików i programistów, dzięki którym możemy przechowywać monstrualne ilości informacji, tak aby można było bardzo szybko przeszukiwać je i docierać do tych właśnie nas interesujących, pośrednio wpływa na życie każdego z nas. Podobnie jak w przypadku praktycznych problemów informatycznych, każde zadanie konkursowe polega na opracowaniu metody (algorytmu) szybkiego przetwarzania danych w celu obliczenia pożądanych wyników. Trzeba mieć świadomość, że uczestnik konkursu nie tylko musi ułożyć stosowny algorytm, ale także poprawnie zaprogramować go w wybranym przez siebie języku programowania (najczęściej C++, C, Java, Python). Wiadomo, że nawet wytrawni programiści muszą się napracować, by napisać poprawnie działający program. W drodze do tego celu walczą najpierw z błędami syntaktycznymi (kompilacji), następnie przeprowadzają szereg testów pozwalających wyeliminować błędy logiczne, a na koniec badają jeszcze wydajność swoich rozwiązań.

Taką drogę przechodzi każdy uczestnik konkursu algorytmiczno-programistycznego. W małej skali pokonuje on etapy realizacji rzeczywistych projektów programistycznych. Konkursy programowania zespołowego w naturalny sposób uczą także pracy grupowej. Typowe zasady takich zawodów stanowią, że zawodnicy podzieleni są na trzyosobowe drużyny, a każda drużyna ma do dyspozycji tylko jeden komputer. Zawodnicy muszą umiejętnie podzielić zadania do rozwiązania w zależności od swoich kompetencji i wspólnie poszukiwać rozwiązań najtrudniejszych zadań. Żeby z sukcesami startować w zawodach w programowaniu zespołowym, niezbędne są następujące umiejętności:

- zdolność precyzyjnej analizy zadań algorytmicznych (znajomość matematyki i logicznego rozumowania są w tym niezmiernie pomocne),
- sprawność w programowaniu,
- znajomość co najmniej jednego środowiska programistycznego oraz umiejętność kompilowania, śledzenia i wykonywania programów w tym środowisku,
- znajomość technik projektowania algorytmów i struktur danych,
- umiejętność pracy zespołowej.

Najlepsi mają opanowane te umiejętności na poziomie niedoścignionym przez przeciętnego zawodowego informatyka.

Akademickie Mistrzostwa Świata w Programowaniu Zespołowym

Akademickie Mistrzostwa Świata w Programowaniu Zespołowym (oficjalna angielska nazwa to *International Collegiate Programming Contest*, w skrócie ICPC), są najstarszym i najbardziej prestiżowym konkursem informatycznym na świecie. Konkurs jest przeznaczony dla studentów. Każdy zespół uczestniczący w zawodach składa się z trzech osób reprezentujących tę samą uczelnię. Mistrzostwa są dwuetapowe. Etap pierwszy to eliminacje regionalne, rozgrywane na sześciu zamieszkałych kontynentach.

Zarówno zawody eliminacyjne, jak i zawody finałowe są rozgrywane w ten sam sposób. Każda trzyosobowa drużyna ma do dyspozycji jeden komputer i od kilku do kilkunastu zadań algorytmiczno-programistycznych do rozwiązania w ciągu pięciu godzin. Rozwiązaniem każdego zadania jest program komputerowy, który powinien poprawnie i szybko obliczać wyniki dla danych przygotowanych przez organizatorów zawodów. Rozwiązania poszczególnych zadań są sprawdzane w czasie rzeczywistym, a drużyny są informowane o wyniku sprawdzenia zadania: zaakceptowane, błędna odpowiedź, przekroczenie limitu czasu na rozwiązanie, błąd wykonania. Jeśli rozwiązanie nie jest zaakceptowane, zostaje odrzucone, a drużyna może nadsyłać kolejne rozwiązania do tego zadania. Zawody wygrywa drużyna, która ma najwięcej zaakceptowanych (rozwiązanych) zadań. W przypadku takiej samej liczby zaakceptowanych zadań przez więcej niż jedną drużynę, o ich kolejności w rankingu decyduje krótszy łączny czas przeznaczony na rozwiązanie wszystkich zaakceptowanych zadań. Dodatkowo za każde wcześniej odrzucone zgłoszenie zadania, które ostatecznie zostaje zaakceptowane, nalicza się 20 minut kary.

Korzenie konkursu ICPC sięgają roku 1970, kiedy w Teksasie rozegrano po raz pierwszy zawody programistyczne pod patronatem organizacji Upsilon Pi Epsilon Computer Science Honor Society (UPE). Rok 1977 uznaje się za pierwszy rok Akademickich Mistrzostw Świata w Programowaniu Zespołowym. Po raz pierwszy wtedy konkurs był dwustopniowy, a same finały rozegrano w Atlancie w połączeniu z konferencją ACM Computer Science Conference. Do roku akademickiego 1988/1989 w konkursie startowały głównie reprezentacje uczelni z USA i Kanady. W roku 1989/1990 konkurs zyskał zasięg światowy. W roku 1997 głównym sponsorem konkursu została firma IBM i od tego czasu obserwuje się jego gwałtowny rozwój. W roku 1997 w eliminacjach konkursu wystartowało 2520 studentów, podczas gdy w roku akademickim 2018/2019 było ich aż 52 709. Wśród 43 dotychczasowych mistrzów świata znajdziemy zespoły reprezentujące najlepsze uczelnie informatyczne świata z czterech kontynentów, w tym Uniwersytet Warszawski. Wykaz na następnej stronie przedstawia pełną listę wszystkich mistrzów świata.

Tak jak konkurs ICPC jest nierozerwalnie związany z osobą profesora Billa Pouchera z Uniwersytetu w Baylor, który był pomysłodawcą i animatorem konkursu, rozwój konkursów w programowaniu zespołowym w Polsce i sukcesy studentów Uniwersytetu Warszawskiego łączą się z osobą profesora Jana Madeya, który w roku 1994 sformował pierwszą reprezentację i wysłał ją na eliminacje regionalne do Amsterdamu. Zespół w składzie Jacek Chrząszcz, Piotr Krysiuk i Tomasz Śmigielski sprawił olbrzymią niespodziankę, zwyciężając w tych eliminacjach i awansując do finałów w Nashville. W finałach wystartowało wówczas 38 drużyn, a polski niedoświadczony zespół zajął jedenaste miejsce — pierwsze nienagradzane medalem*. Od tego czasu zespoły Uniwersytetu Warszawskiego nieprzerwanie biorą udział w finałach konkursu. Uniwersytet Warszawski jest jedyną uczelnią na świecie, która może pochwalić się taką liczbą nieprzerwanych startów w finałach. Największe sukcesy Uniwersytetu Warszawskiego (i Polski) to zwycięstwa w finałach konkursu w latach 2003 i 2007. W roku 2003 zwyciężyła drużyna w składzie: Tomasz Czajka, Andrzej Gąsienica-Samek i Krzysztof Onak. W roku 2007 mistrzami świata zostali Marek Cygan, Marcin Pilipczuk i Filip Wolski.

*Od kilkunastu lat w finałach przyznawanych jest 12 medali: 4 złote, 4 srebrne i 4 brązowe. Wcześniej przyznawano co najwyżej 10 medali. W bardzo rzadkich przypadkach, gdy osiągnięcia drużyn są zbliżone, możliwe jest przyznanie dodatkowych medali brązowych.

Rok i miejsce zawodów		*Mistrzowie Świata w Programowaniu Zespołowym*
1	1977 Atlanta (USA)	Michigan State University (USA)
2	1978 Detroit (USA)	Massachusetts Institute of Technology (USA)
3	1979 Dayton (USA)	Washington University (USA)
4	1980 Kansas City (USA)	Washington University (USA)
5	1981 St. Louis (USA)	University of Missouri-Rolla (USA)
6	1982 Indianapolis (USA)	Baylor University (USA)
7	1983 Melbourne (USA)	University of Nebraska (USA)
8	1984 Filadelfia (USA)	Johns Hopkins University (USA)
9	1985 Nowy Orlean (USA)	Stanford University (USA)
10	1986 Cincinnati (USA)	California Institute of Technology (USA)
11	1987 St. Louis (USA)	Stanford University (USA)
12	1988 Atlanta (USA)	California Institute of Technology (USA)
13	1989 Louisville (USA)	University of California at Los Angeles (USA)
14	1990 Waszyngton (USA)	University of Otago (Nowa Zelandia)
15	1991 San Antonio (USA)	Stanford University (USA)
16	1992 Kansas City (USA)	University of Melbourne (Australia)
17	1993 Indianapolis (USA)	Harvard University (USA)
18	1994 Phoenix (USA)	University of Waterloo (Kanada)
19	1995 Nashville (USA)	Albert-Ludwigs-Universität Freiburg (Niemcy)
20	1996 Filadelfia (USA)	University of California at Berkeley (USA)
21	1997 San Jose (USA)	Harvey Mudd College (USA)
22	1998 Atlanta (USA)	Uniwersytet Karola w Pradze (Czechy)
23	1999 Eindhoven (Holandia)	University of Waterloo (Kanada)
24	2000 Orlando (USA)	Petersburski Uniwersytet Państwowy (Rosja)
25	2001 Vancouver (Kanada)	Petersburski Uniwersytet Państwowy (Rosja)
26	2002 Honolulu (USA)	Shanghai Jiao Tong University (Chiny)
27	2003 Beverly Hills (USA)	Uniwersytet Warszawski (Polska)
28	2004 Praga (Czechy)	Uniwersytet ITMO w Petersburgu (Rosja)
29	2005 Szanghaj (Chiny)	Shanghai Jiao Tong University (Chiny)
30	2006 San Antonio (USA)	Saratowski Uniwersytet Państwowy (Rosja)
31	2007 Tokio (Japonia)	Uniwersytet Warszawski (Polska)
32	2008 Banff (Kanada)	Uniwersytet ITMO w Petersburgu (Rosja)
33	2009 Sztokholm (Szwecja)	Uniwersytet ITMO w Petersburgu (Rosja)
34	2010 Harbin (Chiny)	Shanghai Jiao Tong University (Chiny)
35	2011 Orlando (USA)	Zhejiang University (Chiny)
36	2012 Warszawa (Polska)	Uniwersytet ITMO w Petersburgu (Rosja)
37	2013 Petersburg (Rosja)	Uniwersytet ITMO w Petersburgu (Rosja)
38	2014 Jekaterynburg (Rosja)	Petersburski Uniwersytet Państwowy (Rosja)
39	2015 Marrakesz (Maroko)	Uniwersytet ITMO w Petersburgu (Rosja)
40	2016 Phuket (Tajlandia)	Petersburski Uniwersytet Państwowy (Rosja)
41	2017 Rapid City (USA)	Uniwersytet ITMO w Petersburgu (Rosja)
42	2018 Pekin (Chiny)	Moskiewski Uniwersytet Państwowy (Rosja)
43	2019 Porto (Portugalia)	Moskiewski Uniwersytet Państwowy (Rosja)

Na poniższym wykazie przedstawiamy pełną listę osiągnięć polskich studentów w historii startów w Akademickich Mistrzostwach Świata w Programowaniu Zespołowym (uzyskane miejsca i medale). Dotychczas do finałów awansowały drużyny z następujących uczelni: Uniwersytet Warszawski (UW), Uniwersytet Jagielloński (UJ), Uniwersytet Wrocławski (UWr), Akademia Górniczo-Hutnicza (AGH) i Politechnika Poznańska (PP). Polskie zespoły zdobyły w sumie sześć medali złotych, osiem srebrnych i siedem brązowych.

ICPC 1995 (38 drużyn)

11.	UW	Jacek Chrząszcz, Piotr Krysiuk, Tomasz Śmigielski

ICPC 1996 (43 drużyny)

17.	UW	Marcin Mucha, Krzysztof Sobusiak, Tomasz Śmigielski

ICPC 1997 (50 drużyn)

11.	UW	Bartosz Klin, Marcin Mendelski-Guz, Marcin Sawicki

ICPC 1998 (54 drużyny)

9. (brąz)	UW	Adam Borowski, Jakub Pawlewicz, Krzysztof Sobusiak

ICPC 1999 (62 drużyny)

11.	UW	Bartosz Klin, Marcin Sawicki, Marcin Stefaniak

ICPC 2000 (60 drużyn)

22.	UW	Łukasz Anforowicz, Marek Futrega, Eryk Kopczyński

ICPC 2001 (64 drużyny)

6. (srebro)	UW	Tomasz Czajka, Andrzej Gąsienica-Samek, Marcin Stefaniak

ICPC 2002 (64 drużyny)

11.	UW	Łukasz Kamiński, Eryk Kopczyński, Tomasz Malesiński

ICPC 2003 (70 drużyn)

1. (złoto)	UW	Tomasz Czajka, Andrzej Gąsienica-Samek, Krzysztof Onak

ICPC 2004 (73 drużyny)

10. (brąz)	UW	Tomasz Malesiński, Krzysztof Onak, Paweł Parys
27.	UJ	Grzegorz Gutowski, Arkadiusz Pawlik, Paweł Walter

ICPC 2005 (78 drużyn)

5. (srebro)	UWr	Paweł Gawrychowski, Jakub Łopuszański, Tomasz Wawrzyniak
17.	UW	Szymon Acedański, Tomasz Idziaszek, Jacek Jurewicz

ICPC 2006 (83 drużyny)

2. (złoto)	UJ	Arkadiusz Pawlik, Bartosz Walczak, Paweł Walter
7. (srebro)	UW	Marcin Michalski, Paweł Parys, Bartłomiej Romański

ICPC 2007 (88 drużyn)

1. (złoto)	UW	Marek Cygan, Marcin Pilipczuk, Filip Wolski
26.	UWr	Paweł Gawrychowski, Jakub Łopuszański, Tomasz Wawrzyniak

ICPC 2008 (100 drużyn)

13.	UW	Marek Cygan, Marcin Pilipczuk, Filip Wolski
31.	AGH	Daniel Czajka, Andrzej Szombierski, Marcin Wielgus
31.	UJ	Rafał Józefowicz, Alan Meller, Bartosz Walczak

ICPC 2009 (100 drużyn)

9. (brąz)	UW	Marcin Andrychowicz, Maciej Klimek, Marcin Kościelnicki
34.	UJ	Kamil Kraszewski, Marek Wróbel, Paweł Zaborski

ICPC 2010 (103 drużyny)

8. (srebro)	UW	Karol Kurach, Krzysztof Pawłowski, Michał Pilipczuk
14.	UWr	Władysław Kwaśnicki, Przemysław Pająk, Przemysław Uznański

ICPC 2011 (103 drużyny)

9. (brąz)	UJ	Robert Obryk, Adam Polak, Maciej Wawro
13.	UW	Tomasz Kulczyński, Jakub Pachocki, Wojciech Śmietanka
42.	UWr	Krzysztof Piecuch, Damian Rusak, Łukasz Zatorski

ICPC 2012 (112 drużyn)

2. (złoto)	UW	Tomasz Kulczyński, Jakub Pachocki, Wojciech Śmietanka
18.	UJ	Robert Obryk, Adam Polak, Maciej Wawro
18.	UWr	Marcin Dublański, Jarosław Gomułka, Karol Pokorski
36.	PP	Konrad Baumgart, Piotr Żurkowski, Tomasz Żurkowski

ICPC 2013 (120 drużyn)

6. (srebro)	UW	Marcin Andrychowicz, Maciej Klimek, Tomasz Kociumaka
9. (brąz)	UJ	Jakub Adamek, Grzegorz Guśpiel, Jonasz Pamuła
27.	UWr	Anna Piekarska, Damian Straszak, Jakub Tarnawski

ICPC 2014 (122 drużyny)

5. (srebro)	UW	Jarosław Błasiok, Tomasz Kociumaka, Jakub Oćwieja
13.	UWr	Anna Piekarska, Tomasz Syposz, Jakub Tarnawski
45.	UJ	Jakub Adamek, Igor Adamski, Piotr Bejda

ICPC 2015 (128 drużyn)

12. (brąz)	UW	Kamil Dębowski, Błażej Magnowski, Marek Sommer
13.	UWr	Bartłomiej Dudek, Maciej Dulęba, Mateusz Gołębiewski
15.	UJ	Piotr Bejda, Grzegorz Guśpiel, Michał Seweryn

ICPC 2016 (128 drużyn)

5. (srebro)	UW	Wojciech Nadara, Marcin Smulewicz, Marek Sokołowski
9. (brąz)	UWr	Barłomiej Dudek, Maciej Dulęba, Mateusz Gołębiewski
14.	UJ	Krzysztof Maziarz, Michał Zając, Szymon Łukasz

ICPC 2017 (133 drużyny)

2. (złoto)	UW	Wojciech Nadara, Marcin Smulewicz, Marek Sokołowski
20.	UWr	Paweł Michalak, Tomasz Syposz, Michał Łowicki
34.	UJ	Vladyslav Hlembotskyi, Krzysztof Maziarz, Michał Zając

ICPC 2018 (140 drużyn)

14.	UW	Kamil Dębowski, Mateusz Radecki, Marek Sommer
31.	UJ	Vladyslav Hlembotskyi, Franciszek Stokowacki, Michał Zieliński

ICPC 2019 (135 drużyn)

4. (złoto)	UW	Jakub Boguta, Konrad Paluszek, Mateusz Radecki
6. (srebro)	UWr	Anadi Agrawal, Michał Górniak, Jarosław Kwiecień

Akademickie Mistrzostwa Europy Środkowej w Programowaniu Zespołowym

Od roku akademickiego 1995/1996 polskie zespoły studenckie walczą o awans do finałów w konkursie regionalnym — Akademickich Mistrzostwach Europy Środkowej w Programowaniu Zespołowym (ang. *Central European Regional Contest*, w skrócie CERC). W zawodach CERC startują zespoły studenckie z wydziałów informatycznych czołowych uczelni z następujących krajów Europy Środkowej: Austrii, Chorwacji, Czech, Polski, Słowacji, Słowenii i Węgier. Region CERC jest bardzo silny i często awans do finałów światowych z tego regionu jest gwarancją sukcesu, o czym świadczą chociażby trzy tytuły mistrzowskie dla zespołów z tego regionu — dwa razy wygrywał Uniwersytet Warszawski, zaś raz Uniwersytet Karola w Pradze. Poniżej przedstawiamy listę zwycięzców Akademickich Mistrzostw Europy Środkowej w Programowaniu Zespołowym.

Rok i miejsce zawodów			*Mistrzowie Europy Środkowej w Programowaniu*
1	1995	Bratysława	Uniwersytet Warszawski
2	1996	Bratysława	Uniwersytet Komeniusza w Bratysławie
3	1997	Bratysława	Uniwersytet Komeniusza w Bratysławie
4	1998	Praga	Uniwersytet Komeniusza w Bratysławie
5	1999	Praga	Uniwersytet Karola w Pradze
6	2000	Praga	Uniwersytet Warszawski
7	2001	Warszawa	Uniwersytet Warszawski
8	2002	Warszawa	Uniwersytet Warszawski
9	2003	Warszawa	Uniwersytet Warszawski
10	2004	Budapeszt	Uniwersytet Warszawski
11	2005	Budapeszt	Uniwersytet Warszawski
12	2006	Budapeszt	Uniwersytet Warszawski
13	2007	Praga	Uniwersytet Warszawski
14	2008	Wrocław	Uniwersytet Warszawski
15	2009	Wrocław	Uniwersytet Warszawski
16	2010	Wrocław	Uniwersytet Warszawski
17	2011	Praga	Uniwersytet Warszawski
18	2012	Kraków	Uniwersytet Komeniusza w Bratysławie
19	2013	Kraków	Uniwersytet Komeniusza w Bratysławie
20	2014	Kraków	Uniwersytet w Zagrzebiu
21	2015	Zagrzeb	Uniwersytet Warszawski
22	2016	Zagrzeb	Uniwersytet Warszawski
23	2017	Zagrzeb	Uniwersytet Warszawski
24	2018	Praga	Uniwersytet Warszawski

Akademickie Mistrzostwa Polski w Programowaniu Zespołowym

Choć pierwsze starty polskich zespołów w zawodach w programowaniu zespołowym miały charakter przypominający pospolite ruszenie, to od roku akademickiego 1996/1997 przygotowania do udziału w mistrzostwach i wyłanianie drużyn reprezentacyjnych odbywają się w sposób systemowy. Zazwyczaj drużyny reprezentacyjne z poszczególnych uczelni są formowane w lokalnych eliminacjach uczelnianych. Najlepsze drużyny rywalizują w Akademickich Mistrzostwach Polski w Programowaniu Zespołowym, które odbywają się najczęściej pod koniec października. Pierwsze zawody AMPPZ odbyły się w 1996 roku na Politechnice Poznańskiej. Pomysłodawcami zawodów byli profesorowie Jerzy Nawrocki z Politechniki Poznańskiej i Jan Madey z Uniwersytetu Warszawskiego. Do roku 2018 rozegrano 23 edycje AMPPZ, w których za każdym razem zwyciężał zespół z Uniwersytetu Warszawskiego.

Rok i miejsce zawodów			*Mistrzowie Polski w Programowaniu Zespołowym*
1	1996	Poznań (PP)	Marcin Mucha, Jakub Pawlewicz, Krzysztof Sobusiak
2	1997	Wrocław (PWr)	Bartosz Klin, Marcin Mendelski-Guz, Marcin Sawicki
3	1998	Warszawa (UW)	Bartosz Klin, Marcin Sawicki, Marcin Stefaniak
4	1999	Warszawa (UW)	Jakub Pawlewicz, Marcin Stefaniak, Tomasz Waleń
5	2000	Warszawa (UW)	Tomasz Czajka, Andrzej Gąsienica-Samek, Marcin Stefaniak
6	2001	Wrocław (UWr)	Łukasz Kamiński, Eryk Kopczyński, Tomasz Malesiński
7	2002	Wrocław (UWr)	Łukasz Kamiński, Tomasz Malesiński, Paweł Parys
8	2003	Wrocław (UWr)	Tomasz Malesiński, Krzysztof Onak, Paweł Parys
9	2004	Kraków (UJ)	Marcin Michalski, Paweł Parys, Bartłomiej Romański
10	2005	Kraków (UJ)	Marek Cygan, Marcin Pilipczuk, Piotr Stańczyk
11	2006	Kraków (UJ)	Marek Cygan, Marcin Pilipczuk, Filip Wolski
12	2007	Poznań (PP)	Marek Cygan, Marcin Pilipczuk, Filip Wolski
13	2008	Poznań (PP)	Tomasz Kulczyński, Jakub Łącki, Piotr Mikulski
14	2009	Poznań (UAM)	Jakub Łącki, Piotr Niedźwiedź, Wojciech Śmietanka
15	2010	Poznań (UAM)	Tomasz Kulczyński, Jakub Pachocki, Wojciech Śmietanka
16	2011	Warszawa (UW)	Tomasz Kulczyński, Jakub Pachocki, Wojciech Śmietanka
17	2012	Warszawa (UW)	Jarosław Błasiok, Mirosław Michalski, Jakub Oćwieja
18	2013	Warszawa (UW)	Jarosław Błasiok, Tomasz Kociumaka, Jakub Oćwieja
19	2014	Warszawa (UW)	Wojciech Nadara, Grzegorz Prusak, Marcin Smulewicz
20	2015	Wrocław (UWr)	Kamil Dębowski, Błażej Magnowski, Marek Sommer
21	2016	Wrocław (UWr)	Wojciech Nadara, Marcin Smulewicz, Marek Sokołowski
22	2017	Wrocław (UWr)	Kamil Dębowski, Mateusz Radecki, Marek Sommer
23	2018	Wrocław (UWr)	Jakub Boguta, Konrad Paluszek, Mateusz Radecki

2011

XVI Akademickie Mistrzostwa Polski
w Programowaniu Zespołowym
Warszawa, 28–30 października 2011

A

Prostokąt arytmetyczny

Autor zadania: Jakub Radoszewski
Opis rozwiązania: Jakub Radoszewski
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2011/ary`

Na każdym polu kartki w kratkę, składającej się z $n \times m$ kwadratów jednostkowych, zapisano jedną liczbę całkowitą. W tym zadaniu interesują nas *prostokąty arytmetyczne* położone na tej kartce, czyli takie prostokąty złożone z kwadratów jednostkowych, że liczby w każdym wierszu i w każdej kolumnie tworzą ciągi arytmetyczne. Przypomnijmy, że ciąg arytmetyczny to ciąg liczbowy, w którym każde dwa kolejne wyrazy różnią się o tę samą liczbę.

Na danej kartce w kratkę poszukujemy największego prostokąta arytmetycznego, tj. obejmującego najwięcej kwadratów jednostkowych. Przykładowo, największy prostokąt arytmetyczny na poniższej kartce składa się z dziewięciu kwadratów jednostkowych:

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

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita t ($1 \leqslant t \leqslant 10\,000$) oznaczająca liczbę zestawów testowych opisanych w dalszej części wejścia. Opis każdego zestawu rozpoczyna się od wiersza z dwiema liczbami całkowitymi n i m ($1 \leqslant n, m \leqslant 3000$). W każdym z kolejnych n wierszy znajduje się m liczb całkowitych z zakresu od 0 do 10^9. Są to liczby wpisane w poszczególne kwadraty jednostkowe kartki w kratkę. Rozmiar każdego pliku wejściowego będzie nie większy niż 20 MB.

Wyjście

Należy wypisać t wierszy z odpowiedziami dla kolejnych zestawów testowych. Odpowiedzią dla jednego zestawu jest jedna liczba całkowita równa liczbie kwadratów jednostkowych zawartych w największym prostokącie arytmetycznym na kartce opisanej w danym zestawie.

Przykład

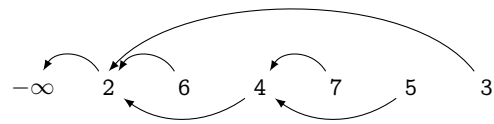
Dla danych wejściowych:	poprawnym wynikiem jest:
2 4 4 5 3 5 7 2 4 4 4 3 5 3 1 6 3 2 4 2 3 0 1 2 1 2 3	9 6

Rozwiązanie

W omówieniu zadania *Prostokąt arytmetyczny* posłużymy się dwoma problemami pomocniczymi, których rozwiązania są mniej lub bardziej znane. W rozwiązaniu każdy kolejny problem będziemy sprowadzać do poprzedniego.

Katastrofy lotnicze

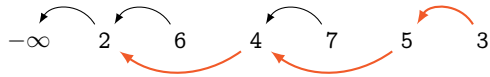
W problemie tym dany jest n-elementowy ciąg liczb całkowitych $a_1, \ldots, a_n$. Chcielibyśmy dla każdego elementu ciągu wyznaczyć najbliższy mniejszy element położony na lewo od niego. Formalnie, dla każdego $i \in \{1, \ldots, n\}$ szukamy największego takiego j, $j < i$, że $a_j < a_i$. Aby wartość ta była zawsze dobrze zdefiniowana, dokładamy do ciągu sztuczny element $a_0 = -\infty$ (patrz rysunek 1). Oryginalny problem katastrof lotniczych ma dosyć drastyczne sformułowanie. Mamy dane liczby ofiar w katastrofach lotniczych, które miały miejsce w kolejnych latach. Dla każdej katastrofy chcielibyśmy stwierdzić, od ilu lat nie było równie tragicznej katastrofy. W tej wersji szukalibyśmy najbliższego *większego* elementu położonego na lewo. Wróćmy jednak do naszego początkowego sformułowania problemu.



Rysunek 1. Rozwiązanie problemu katastrof lotniczych dla ciągu $2, 6, 4, 7, 5, 3$.

Problem katastrof lotniczych można prosto rozwiązać w złożoności liniowej, stosując metodę *pójścia za strzałkami*. Idea jest jasna: przetwarzamy ciąg od lewej do prawej i przypisujemy kolejnym elementom strzałki prowadzące do elementów najbliższych mniejszych. Dla danego i zaczynamy od sprawdzenia, czy $a_{i-1} < a_i$. Jeśli tak, to wiemy, że strzałka z a_i prowadzi do a_{i-1}. A jeśli nie, to idziemy do pierwszego elementu mniejszego niż a_{i-1}, czyli dokładnie wzdłuż strzałki z a_{i-1}.

Kontynuujemy to postępowanie aż do momentu, gdy znajdziemy element mniejszy niż a_i. Przykładowo, rysunek 2 przedstawia wyznaczanie strzałki wychodzącej z elementu $a_6 = 3$.



Rysunek 2. Obliczanie wyniku dla $a_6 = 3$.

Aby uzasadnić, że algorytm działa w czasie liniowym, wystarczy wykazać, że wzdłuż każdej strzałki przejdziemy co najwyżej raz. Załóżmy więc, że podczas wyznaczania strzałki dla elementu a_i przechodzimy wzdłuż strzałki wychodzącej z pewnego a_j $(j < i)$. To oznacza, że $a_j \geqslant a_i$, gdyż w przeciwnym razie samo a_j byłoby kandydatem na wynik dla a_i. Oznaczmy przez a_k element wynikowy dla elementu a_i. Dalej w algorytmie będziemy rozważać elementy $a_{i+1}, a_{i+2}, \ldots$ Wykażemy, że dla żadnego z nich, idąc po strzałkach, nie napotkamy elementu a_j. Aby osiągnąć element a_j, musielibyśmy w pewnym momencie za pomocą strzałek przeskoczyć element a_i (gdybyśmy napotkali sam element a_i, to na pewno nie znaleźlibyśmy się w tej rundzie w a_j). Jednak strzałka biegnąca nad a_i trafia w element mniejszy od a_i, a najwcześniejszym takim elementem jest a_k. Tak więc taka strzałka nie może trafić w a_j, który jest położony na prawo od a_k.

Działka

Zajmijmy się teraz drugim problemem pomocniczym. Mamy daną planszę (tablicę A) złożoną z n wierszy i m kolumn, wypełnioną zerami i jedynkami. Należy znaleźć prostokątny fragment tej planszy wypełniony samymi jedynkami, o jak największym polu (patrz rysunek 3). Tytuł sekcji wziął się stąd, że problem ten pojawił się na IX Olimpiadzie Informatycznej jako zadanie *Działka*. Dla prostoty języka nasz problem będziemy zatem określać jako poszukiwanie *optymalnej działki*.

1	0	0	1	1
0	1	1	1	1
1	1	1	1	0
1	1	1	1	0
0	1	1	0	1

Rysunek 3. Optymalna działka w tym przypadku ma pole 9.

Zadanie *Działka* można rozwiązać w złożoności liniowej, czyli $O(nm)$. Przedstawimy tu jedno takie rozwiązanie, nieco inne od rozwiązania firmowego z IX Olimpiady.

Dla ustalenia uwagi przez (i, j) będziemy oznaczać pole planszy znajdujące się w i-tym wierszu od góry i j-tej kolumnie od lewej. Na początku dla każdego pola (i, j) wyznaczymy pomocniczą wartość $D[i, j]$ oznaczającą liczbę kolejnych pól wypełnionych jedynkami położonych w dół od tego pola, wliczając samo pole (i, j). Takie wartości łatwo obliczyć w czasie $O(nm)$, idąc od dołu do góry planszy. Korzystamy z faktu, że jeśli $A[i, j] = 1$, to $D[i, j] = D[i + 1, j] + 1$, a w przeciwnym razie oczywiście $D[i, j] = 0$ (patrz rysunek 4).

1	0	0	4	2
0	4	4	3	1
2	3	3	2	0
1	2	2	1	0
0	1	1	0	1

Rysunek 4. Tablica pomocnicza D.

Teraz czas na kluczowe spostrzeżenie. Otóż optymalną działkę możemy skonstruować tak: bierzemy jakieś pole (i, j) (w rozwiązaniu sprawdzimy wszystkie możliwości wyboru pola (i, j)) i wyznaczamy prostokąt zawierający to pole w górnym wierszu, o wysokości $D[i, j]$ i sięgający w prawo i w lewo tak daleko, jak tylko się da — czyli aż do napotkania brzegu planszy lub pola z zerem. Uzasadnijmy, że dla pewnego pola (i, j) faktycznie otrzymamy optymalną działkę. Wynikowa działka musi być maksymalna w tym sensie, że każdy jej bok albo pokrywa się z bokiem planszy, albo też sąsiaduje bezpośrednio z polem zawierającym zero. Dotyczy to w szczególności dolnego boku działki. Niech (i', j) oznacza pole znajdujące się w dolnym wierszu działki sąsiadujące bezpośrednio z polem z zerem (lub też dowolne pole, jeśli dolny brzeg działki styka się z brzegiem planszy). Niech i oznacza numer górnego wiersza działki. Wtedy działka ma wysokość $D[i, j]$, zawiera pole (i, j) w swoim górnym wierszu i nie jest rozszerzalna ani w lewo, ani w prawo, co jest zgodne ze spostrzeżeniem. Przykładowo, na rysunkach 3 i 4 pole (i, j) optymalnej działki o polu powierzchni 9 to jej prawy górny róg.

Poczynione spostrzeżenie możemy sformułować inaczej: dla każdego pola (i, j) o niezerowym $D[i, j]$ szukamy najbliższych pól w tym samym wierszu, dla których wartości D są mniejsze niż $D[i, j]$, czyli takich indeksów j' i j'', że

$$j' < j < j'', \quad D[i, j'], D[i, j''] < D[i, j], \quad j' \text{ jest maksymalne}, \ j'' \text{ jest minimalne}.$$

Indeksy j' i j'' to numery kolumn, które znajdują się tuż za bokami naszej działki. Przyjmujemy dla uproszczenia, że $D[i, 0] = D[i, m + 1] = 0$. Wówczas wynikiem jest maksimum z iloczynów postaci $(j'' - j' - 1) \cdot D[i, j]$ dla wszystkich pól (i, j). Zauważmy, że wyznaczenie j' i j'' to dokładnie zastosowanie problemu katastrof lotniczych do i-tego wiersza tablicy D, tyle że raz od lewej do prawej (wyznaczanie j'), a raz od prawej do lewej (wyznaczanie j''). Stosując poprzedni algorytm wiersz po wierszu, otrzymujemy algorytm działający w czasie $O(nm)$.

Prostokąt arytmetyczny

Przypomnijmy, że w problemie z zadania mamy daną planszę rozmiaru $n \times m$ zawierającą nieujemne liczby całkowite. Szukamy w niej prostokąta arytmetycznego o maksymalnym polu, przy czym prostokąt arytmetyczny to prostokąt, w którym liczby w każdym wierszu oraz w każdej kolumnie tworzą ciągi arytmetyczne. Przykład takiego prostokąta znajduje się na rysunku 5.

1	3	7	11	15
2	4	6	8	10
5	5	5	5	5
8	6	4	2	0
6	3	0	4	8

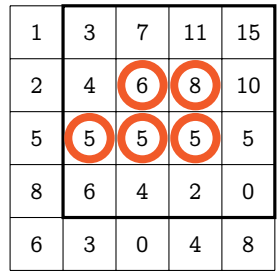
Rysunek 5. Maksymalny prostokąt arytmetyczny ma pole 16.

Na początek zajmijmy się prostokątami o szerokości 1 (tzn. zawartymi w jednym wierszu). Widzimy, że każdy wiersz planszy możemy podzielić na maksymalne ciągi arytmetyczne, z których każdy ma długość co najmniej dwa i każde dwa kolejne ciągi mają dokładnie jeden element wspólny. Przykładowo, dolny wiersz tablicy z rysunku 5 dzielimy na ciągi $(6, 3, 0)$ i $(0, 4, 8)$, a górny na ciągi $(1, 3)$ i $(3, 7, 11, 15)$. Korzystając z takiego przedstawienia, w czasie $O(nm)$ łatwo znajdziemy najdłuższy prostokąt arytmetyczny o szerokości 1. Podobnie rozpatrujemy prostokąty o długości 1, szerokości 2 (jak?) i długości 2.

Odtąd interesować nas będą tylko prostokąty, których każdy bok ma długość co najmniej 3. Znajdowanie takich prostokątów sprowadzimy do zadania *Działka*.

Zaznaczmy mianowicie kółkiem każdy taki element planszy, że kwadrat o boku 3 zawierający ten element w środku jest prostokątem arytmetycznym (patrz rysunek 6). Okazuje się, że prostokąt o obu wymiarach nie mniejszych niż 3 jest arytmetyczny wtedy i tylko wtedy, gdy wszystkie zawarte w nim elementy planszy poza, ewentualnie, jego wewnętrzną obwódką o szerokości 1 zostały zaznaczone. Rzeczywiście, jeśli prostokąt jest arytmetyczny, to każdy jego podprostokąt jest arytmetyczny, więc w szczególności wszystkie kwadraty o boku 3 zawarte w tym prostokącie są arytmetyczne. W drugą stronę, jeśli w danym prostokącie planszy dwa sąsiednie elementy są zaznaczone kółkami, to ciągi arytmetyczne w otaczających je kwadratach 3×3 sklejają się w dłuższe ciągi arytmetyczne dokładnie tak, jak trzeba. Kontynuując to rozumowanie, możemy wykazać, że każdy podprostokąt o maksymalnej długości i szerokości 3 lub o maksymalnej szerokości i długości 3 jest arytmetyczny. Stąd mamy, że każdy wiersz i każda kolumna prostokąta tworzą ciągi arytmetyczne, czyli cały prostokąt jest prostokątem arytmetycznym.

Podsumowując: wąskie prostokąty arytmetyczne rozpatrzyliśmy osobno, a problem szukania odpowiednio grubego prostokąta arytmetycznego o maksymalnym polu sprowadziliśmy do poszukiwania maksymalnych prostokątów złożonych wyłącznie z wyróżnionych pól. Ten ostatni problem odpowiada właściwie zadaniu *Działka*, choć jest tu pewien haczyk. Otóż w tamtym problemie poszukiwaliśmy



Rysunek 6. Zaznaczone elementy planszy znajdują się w środkach kwadratów arytmetycznych 3×3.

dokładnie jednej działki — tej o maksymalnym polu. Po wykonaniu naszej transformacji nie mamy jednak gwarancji, że taka działka, poszerzona o obwódkę o szerokości 1, daje prostokąt arytmetyczny o największym możliwym polu. Może się przecież tak zdarzyć, że dla pewnych liczb całkowitych dodatnich a, b, c, d mamy $ab > cd$, ale $(a+2)(b+2) < (c+2)(d+2)$ (czy potrafisz podać taki przykład?). Na szczęście dosyć łatwo wybrnąć z tej sytuacji. Być może optymalny prostokąt arytmetyczny nie odpowiada optymalnej działce, ale na pewno odpowiada jakiejś działce *maksymalnej*, czyli nierozszerzalnej w żadnym kierunku. Wystarczy teraz przypomnieć sobie, że nasze rozwiązanie zadania *Działka* tak naprawdę przeglądało wszystkie maksymalne działki (w poszukiwaniu tej o największym polu), więc możemy je rzeczywiście zastosować do znalezienia optymalnego prostokąta arytmetycznego. Ostatecznie otrzymujemy rozwiązanie działające w optymalnym czasie $O(nm)$.

B

Bajtocki Bieg Uliczny

★★★★

Autor zadania: Jakub Łącki
Opis rozwiązania: Jakub Łącki
Dostępna pamięć: 64 MB
https://oi.edu.pl/pl/archive/amppz/2011/baj

W centrum Bajtogrodu ma się jutro odbyć Bajtocki Bieg Uliczny. Ulice w Bajtogrodzie tworzą równomierną kratkę: wszystkie prowadzą z południa na północ bądź z zachodu na wschód. Uczestnikom biegu udostępniono jedynie pewne ich fragmenty.

Bajtazar ma się zająć rozstawieniem reklam sponsorów imprezy na niektórych skrzyżowaniach i w tym celu musi przyjrzeć się mapie trasy biegu. Mapa przedstawia fragmenty ulic, które udostępniono biegaczom. Zaznaczono na niej n skrzyżowań oraz m pionowych i poziomych odcinków ulic. Każdy odcinek zaczyna się i kończy na jakimś skrzyżowaniu i nie zawiera żadnych innych skrzyżowań. Odcinki ulic nie przecinają się poza skrzyżowaniami.

Skrzyżowania są ponumerowane od 1 do n. Bieg ma rozpocząć się na skrzyżowaniu numer 1 i zakończyć na skrzyżowaniu numer n. Biegacze mogą wybrać swoją trasę biegu, przy czym zobowiązani są biec tylko na południe lub na wschód i jedynie po odcinkach zaznaczonych na mapie. Odcinki ulic na mapie są dobrane tak, że biegnąc zgodnie z zasadami, z każdego miejsca da się dotrzeć do mety i każde miejsce jest osiągalne ze skrzyżowania startowego.

Bajtazar chciałby porozwieszać reklamy tak, żeby mieć pewność, że żaden biegacz nie zobaczy dwukrotnie reklamy tego samego sponsora. Wobec tego, dla niektórych par skrzyżowań Bajtazar musi sprawdzić, czy możliwe jest, by trasa jakiegoś uczestnika przebiegała przez obydwa skrzyżowania. Bieg startuje już jutro, dlatego pilnie potrzebny jest program, który pomoże mu w pracy.

Wejście

Pierwszy wiersz wejścia zawiera trzy liczby całkowite n, m i k ($2 \leqslant n \leqslant 100\,000$, $1 \leqslant m \leqslant 200\,000$, $1 \leqslant k \leqslant 300\,000$). Oznaczają one odpowiednio liczbę skrzyżowań na trasie biegu, liczbę odcinków na mapie oraz liczbę par skrzyżowań do sprawdzenia.

Kolejne n wierszy opisuje położenia skrzyżowań. W i-tym spośród nich znajdują się współrzędne i-tego skrzyżowania w postaci dwóch liczb całkowitych x_i, y_i ($-10^9 \leqslant x_i, y_i \leqslant 10^9$). Dodatkowo, $x_1 \leqslant x_n$ i $y_1 \geqslant y_n$. W danym miejscu może znajdować się co najwyżej jedno skrzyżowanie. Osie układu współrzędnych utożsamiamy w naturalny sposób z kierunkami świata: oś OX prowadzi na wschód, zaś oś OY — na północ.

Każdy z kolejnych m wierszy zawiera opis jednego odcinka na mapie składający się z pary liczb całkowitych a_i, b_i ($1 \leqslant a_i, b_i \leqslant n$, $a_i \neq b_i$) oznaczających numery skrzyżowań połączonych tym odcinkiem. Wszystkie te odcinki są pionowe lub poziome i nie mają punktów wspólnych poza wspólnymi końcami na skrzyżowaniach.

W następnych k wierszach znajdują się opisy par skrzyżowań do sprawdzenia. W i-tym z tych wierszy znajdują się dwie liczby całkowite p_i, q_i ($1 \leqslant p_i, q_i \leqslant n$, $p_i \neq q_i$).

Wyjście

Twój program powinien wypisać k wierszy. W i-tym spośród tych wierszy powinno znaleźć się słowo `TAK`, jeśli trasa pewnego uczestnika biegu może prowadzić przez skrzyżowania p_i i q_i (w dowolnej kolejności). W przeciwnym wypadku należy wypisać `NIE`.

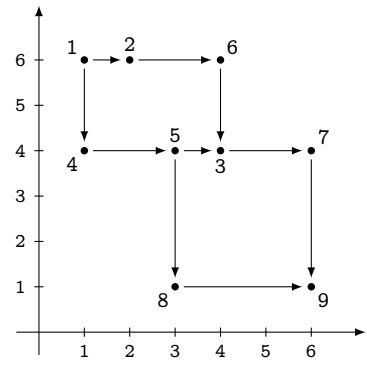
Przykład

Dla danych wejściowych:

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

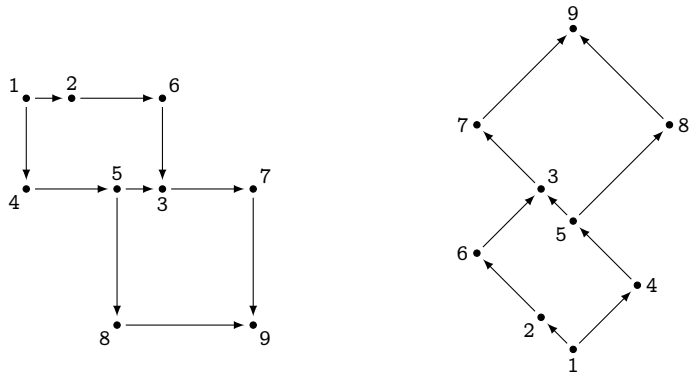
poprawnym wynikiem jest:

```
TAK
NIE
NIE
TAK
```



Rozwiązanie

Na początek obróćmy mapę Bajtogrodu o 135° w lewo. W efekcie skrzyżowanie startowe znajdzie się na samym dole mapy, skrzyżowanie końcowe na górze, a ulice będą wychodziły ze skrzyżowań w dwóch skośnych kierunkach: w prawo i do góry oraz w lewo i do góry (patrz rysunek 1). Obrót w żaden sposób nie zmienia wyniku, a w dalszej części opisu prościej nam będzie operować na tak właśnie zorientowanej mapie.



Rysunek 1. Przykładowy plan Bajtogrodu z treści zadania (po lewej) oraz plan po obróceniu o 135° (po prawej).

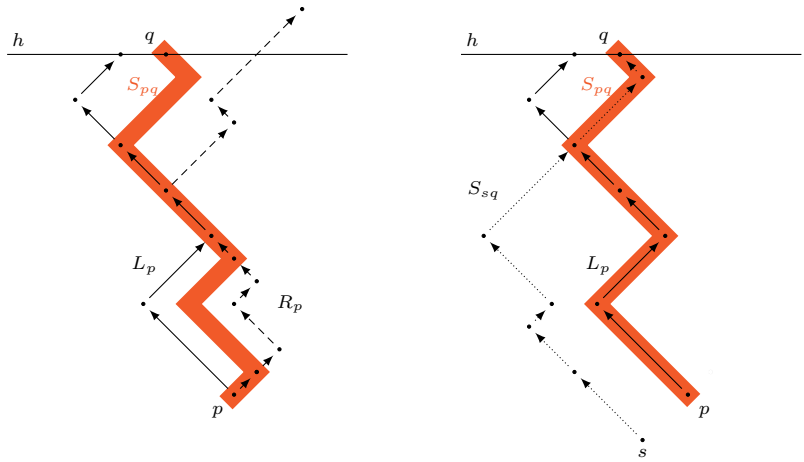
Mapę Bajtogrodu będziemy traktować jako graf skierowany. W tym grafie jest n wierzchołków reprezentujących skrzyżowania oraz m krawędzi odpowiadających odcinkom dróg. Ustalmy pewien wierzchołek v. *Ścieżką lewą* wychodzącą z wierzchołka v nazwiemy ścieżkę, która zaczyna się w v i z każdego kolejnego wierzchołka, jeśli to tylko możliwe, wychodzi w lewo. Jeśli z wierzchołka wychodzi tylko krawędź w prawo, to ścieżka lewa wychodzi tą właśnie krawędzią. Ścieżka lewa kończy się w wierzchołku odpowiadającym mecie. Analogicznie definiujemy *ścieżkę prawą*. Pojęcia ścieżki lewej i prawej będą stanowiły podstawę naszego efektywnego rozwiązania. Oznaczymy przez L_v ścieżkę lewą wychodzącą z wierzchołka v, zaś przez R_v — ścieżkę prawą zaczynającą się w v. W przykładzie z rysunku 1 ścieżka L_4 przebiega kolejno przez wierzchołki $4, 5, 3, 7, 9$, zaś R_4 przez wierzchołki $4, 5, 8, 9$.

Przyjmijmy teraz, że chcemy sprawdzić, czy z wierzchołka $p = (x_p, y_p)$ możemy dostać się do wierzchołka $q = (x_q, y_q)$. Narysujmy poziomą prostą h przechodzącą przez wierzchołek q. Załóżmy, że wierzchołek p leży poniżej prostej h, bo inaczej ścieżka z p do q oczywiście nie istnieje (chyba że $p = q$). Poprowadźmy też z wierzchołka p ścieżki L_p i R_p. Jeśli z wierzchołka p da się dotrzeć do wierzchołka q, to zachodzi następujący warunek:

Warunek 1. Ścieżka L_p przecina prostą h w wierzchołku q lub na lewo od niego, a ścieżka R_p przecina prostą h w wierzchołku q lub na prawo od niego.

Dlaczego tak się dzieje? Wiemy, że istnieje ścieżka z p do q (nazwijmy ją S_{pq}), która prostą h przecina w punkcie q. Ścieżka lewa z p nigdy nie może znaleźć się

na prawo od S_{pq}. Analogicznie, ścieżka prawa zaczynająca się w p na pewno nie będzie prowadzić na lewo od S_{pq}. Przykład znajduje się po lewej stronie rysunku 2.



Rysunek 2. Po lewej: ścieżki z warunku 1 oraz prosta h. Ścieżka S_{pq} musi znajdować się w obszarze ograniczonym przez ścieżki L_p i R_p. Po prawej: ścieżka S_{sq} przecina L_p, dzięki czemu możemy skonstruować ścieżkę z p do q.

Co więcej, zachodzi również odwrotna zależność: jeśli warunek 1 jest spełniony, to istnieje ścieżka z p do q. Aby to uzasadnić, rozważmy ścieżkę S_{sq} z wierzchołka startowego s do wierzchołka q (prawa strona rysunku 2). Taka ścieżka na pewno istnieje, bo z wierzchołka startowego s możemy dostać się do każdego innego wierzchołka. Ponadto każda ścieżka prowadząca do wierzchołka q, która zaczyna się w wierzchołku p lub poniżej, musi przeciąć ścieżkę L_p lub R_p. Dla ustalenia uwagi przyjmijmy, że ścieżka S_{sq} przecina ścieżkę L_p. Wówczas, aby dostać się z p do q, podążamy ścieżką L_p aż do punktu przecięcia jej ze ścieżką S_{sq}, a tam przesiadamy się na ścieżkę S_{sq} i podążamy nią aż do wierzchołka q. Wszystko to oznacza, że aby stwierdzić, czy z p możemy dostać się do q, wystarczy sprawdzić warunek 1.

Struktura ścieżek lewych i prawych

Zauważmy, że ścieżki lewe i prawe mają bardzo silną strukturę. Ścieżkę lewą wychodzącą z wierzchołka v możemy skonstruować następująco. Idziemy pierwszą krawędzią (jeśli się da, to w lewo, a w przeciwnym razie w prawo), docieramy do wierzchołka w i od tego momentu dalsza część ścieżki L_v pokrywa się ze ścieżką L_w.

Skonstruujemy teraz dwuwymiarową tablicę *lewa*, dzięki której będziemy mogli szybko poruszać się po ścieżkach lewych. Dla każdego wierzchołka v i każdego i, takiego że $0 \leqslant i \leqslant \lfloor \log_2 n \rfloor$, w komórce $lewa[v, i]$ chcemy zapisać wierzchołek, do którego dotrzemy z wierzchołka v, jeśli wykonamy 2^i kroków po ścieżce L_v. Analogiczną tablicę *prawa* wyznaczamy dla prawych ścieżek. Dla wygody w wierzchołku z metą dodajemy krawędź, która prowadzi do niego samego. Wówczas wszystkie wartości w tablicach są dobrze określone.

Dla każdego wierzchołka v wartość $lewa[v, 0]$ wyznaczamy bez trudu: wystarczy sprawdzić, dokąd prowadzi pierwsza krawędź ścieżki L_v. Dalej nasz algorytm działa w $\lfloor \log_2 n \rfloor$ fazach. W i-tej fazie zakładamy, że wyznaczyliśmy już wartości $lewa[\cdot, i-1]$, i na tej podstawie chcemy wyznaczyć wartości $lewa[\cdot, i]$. Postępujemy zgodnie z następującym pomysłem: aby sprawdzić, dokąd dojdziemy ścieżką L_v po 2^i krokach, wykonujemy nią najpierw 2^{i-1} kroków. Wierzchołek w, do którego dotrzemy, wyznaczyliśmy w poprzedniej fazie. Teraz z wierzchołka w wykonujemy kolejne 2^{i-1} kroków ścieżką L_w, która pokrywa się z pozostałą częścią ścieżki L_v. W ten sposób dowiemy się, dokąd trafimy z v po 2^i krokach. Innymi słowy, wystarczy wykonać przypisanie

$$lewa[v, i] := lewa[lewa[v, i-1], i-1].$$

Ponieważ tablice *lewa* i *prawa* mają po $O(n \log n)$ komórek, potrafimy je wypełnić w łącznym czasie $O(n \log n)$.

Kiedy już wypełnimy obydwie tablice, możemy ich użyć do sprawdzania warunku 1. W tym celu posłużymy się pewną wersją wyszukiwania binarnego. Przyjmijmy, że interesuje nas, gdzie ścieżka L_p przetnie prostą h.

W tym celu, zaczynając w wierzchołku p, będziemy poruszać się ścieżką L_p, uważając, by nie przekroczyć prostej h. Kolejno dla $i = \lfloor \log_2 n \rfloor, \lfloor \log_2 n \rfloor - 1, \ldots, 0$ sprawdzamy, czy po wykonaniu 2^i kroków po ścieżce lewej z wierzchołka, w którym aktualnie się znajdujemy, znajdziemy się powyżej prostej h. Jeśli tak, to nie robimy nic, a jeśli nie, to wykonujemy owe 2^i kroków.

Po wykonaniu tej procedury znajdziemy najwyższy wierzchołek w na ścieżce L_p leżący nie wyżej niż prosta h. Teraz wystarczy jedynie sprawdzić, z której strony punktu q pierwsza krawędź ścieżki L_w przecina prostą h. Analogicznie postępujemy dla ścieżki R_p, co pozwoli nam sprawdzić warunek 1 i odpowiedzieć na zapytanie.

Otrzymujemy w ten sposób algorytm, który po obliczeniach wstępnych w czasie $O(n \log n)$, na każde zapytanie odpowiada w czasie $O(\log n)$.

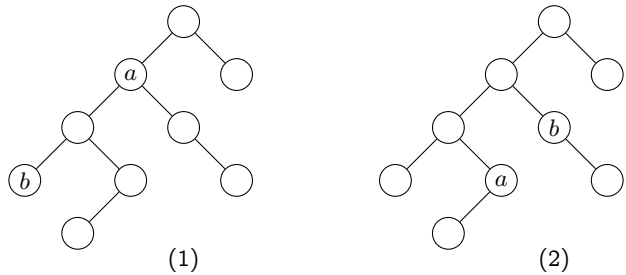
Szybsze rozwiązanie

Mimo że powyższe rozwiązanie jest już dostatecznie szybkie, opiszemy również interesujący algorytm działający w czasie $O(n)$. Na początku zajmiemy się, wydawać by się mogło, nieco innym problemem. Później pokażemy jego związek z naszym zadaniem.

Wzajemne położenie wierzchołków w drzewie

Rozważmy ukorzenione drzewo binarne, w którym każdy wierzchołek ma zero, jednego lub dwóch synów. W przypadku, gdy wierzchołek posiada dwóch synów, zakładamy, że są oni uporządkowani: będziemy mówić o lewym i prawym synu. Wyróżnijmy dwa różne wierzchołki tego drzewa, które nazwiemy a i b. Jakie może być ich wzajemne położenie w drzewie?

Oznaczmy przez S_a i S_b ścieżki od korzenia do, odpowiednio, wierzchołków a i b. Okazuje się, że możliwe są dokładnie cztery przypadki: wierzchołek a może leżeć nad wierzchołkiem b (wtedy a leży na ścieżce S_b), pod wierzchołkiem b (wtedy b leży na ścieżce S_a), na lewo od wierzchołka b (idąc z a do korzenia, na ścieżkę S_b wchodzimy z lewej strony) lub na prawo od wierzchołka b (idąc z a do korzenia, na ścieżkę S_b wchodzimy z prawej strony). Łatwo widzimy, że pierwsze dwa



Rysunek 3. Dwa przypadki wzajemnego położenia wierzchołków w drzewie: (1) wierzchołek a leży nad wierzchołkiem b, (2) wierzchołek a leży na lewo od wierzchołka b. W dwóch pozostałych przypadkach zamieniamy role wierzchołków a i b.

przypadki są symetryczne, podobnie i drugie dwa. Ilustracja przypadków znajduje się na rysunku 3. Pokażemy teraz, w jaki sposób szybko odpowiadać na zapytanie o wzajemne położenie wierzchołków.

Obejdźmy rozważane drzewo w głąb i dla każdego wierzchołka v zapiszmy czas wejścia I_v oraz czas wyjścia O_v, tzn. zapiszmy moment, gdy po raz pierwszy odwiedzamy wierzchołek v, oraz moment zaraz po tym, gdy obejdziemy wszystkie wierzchołki poniżej wierzchołka v. Dzięki temu położenie dwóch różnych wierzchołków a i b wyznaczyć możemy następująco:

- a jest nad b wtedy i tylko wtedy, gdy $I_a < I_b$ oraz $O_b < O_a$,
- a jest pod b wtedy i tylko wtedy, gdy $I_b < I_a$ oraz $O_a < O_b$,
- a jest na lewo od b wtedy i tylko wtedy, gdy $O_a < I_b$,
- a jest na prawo od b wtedy i tylko wtedy, gdy $O_b < I_a$.

Zatem po obliczeniach wstępnych w czasie $O(n)$ wzajemne położenie wierzchołków w drzewie możemy wyznaczać w czasie stałym.

Rozwiązanie w czasie liniowym

Teraz jesteśmy już gotowi, by opisać szybsze rozwiązanie naszego zadania. Skonstruujemy *graf ścieżek lewych* będący sumą ścieżek lewych wychodzących ze wszystkich wierzchołków. Zawiera on te same n wierzchołków co mapa Bajtogrodu, ale z każdego wierzchołka (poza metą) wychodzi w nim tylko jedna krawędź (jest to krawędź w lewo, jeśli istnieje, a w przeciwnym wypadku krawędź w prawo).

Zauważmy, że tak skonstruowany graf jest drzewem binarnym, którego korzeniem jest wierzchołek z metą. To właśnie dlatego mapę Bajtogrodu obróciliśmy tak, by meta była na górze — dzięki temu graf ścieżek lewych wygląda jak każde przyzwoite drzewo w informatyce: ma korzeń u góry i rośnie w dół. Oznaczmy to drzewo przez T_L. Analogicznie definiujemy *graf ścieżek prawych* i oznaczamy go przez T_R.

Tak jak poprzednio chcemy sprawdzić, czy da się z wierzchołka p dotrzeć do wierzchołka q. Jak łatwo zgadnąć, nasz algorytm będzie sprawdzać wzajemne położenie tych wierzchołków w drzewach T_L i T_R. Ponownie będziemy też korzystać z poziomej prostej h przechodzącej przez wierzchołek q.

Rozważmy teraz wzajemne położenie wierzchołków p i q w drzewie T_L. Pierwsze dwa przypadki są proste. Jeśli p leży pod q, to szukana ścieżka istnieje. Wiemy również, że p nie może leżeć nad q w T_L, bo musiałby się znajdować powyżej prostej h.

Przyjmijmy teraz, że p leży na lewo od q w drzewie T_L. Oznacza to, że jeśli spojrzymy na ścieżki lewe L_p i L_q wychodzące z p i q, to do wierzchołka, w którym te ścieżki się spotykają, L_p wchodzi z lewej a L_q z prawej strony. Jeśli więc będziemy cofać się po obydwóch ścieżkach jednocześnie, aż natrafimy na prostą h, to ścieżką L_q trafimy oczywiście do q, a na ścieżce L_p będziemy na prostej h na lewo od q. Nietrudno przekonać się też, że jeśli p leży na prawo od q w drzewie T_L, to ścieżka lewa wychodząca z p przetnie prostą h na prawo od wierzchołka q.

Brzmi znajomo, prawda? To przecież jedna z części warunku 1. Jeśli dołożymy do tego badanie wzajemnego położenia wierzchołków p i q w drzewie T_R, to dostaniemy bardzo efektywny sposób na sprawdzanie warunku 1.

Podsumujmy nasz algorytm. Na początku konstruujemy drzewa T_L i T_R i wykonujemy obliczenia wstępne pozwalające na sprawdzanie wzajemnego położenia wierzchołków w tych drzewach. Oba te kroki wykonujemy w czasie $O(n)$. Następnie, aby odpowiedzieć na pytanie o istnienie ścieżki z p do q, sprawdzamy, czy p nie leży powyżej q, oraz wyznaczamy ich wzajemne położenie wierzchołków w drzewach T_L i T_R. Wszystkie te sprawdzenia wykonujemy w czasie stałym.

Okazuje się więc, że całe zadanie możemy rozwiązać *optymalnie*. Wykonujemy obliczenia wstępne w czasie liniowym, a następnie na każde zapytanie odpowiadamy w czasie stałym. Wszystko to możliwe jest dzięki silnej strukturze naszego grafu: jest on planarny i acykliczny, a ponadto zawiera dokładnie jedno źródło i jedno ujście. W grafach dowolnych problem ten jest znacznie trudniejszy: najlepszy znany algorytm, który odpowiada na zapytania w czasie stałym, wymaga obliczeń wstępnych w czasie $O(n^{2,38})$ i korzysta z algorytmu szybkiego mnożenia macierzy.

C

Czy się zatrzyma?

Autor zadania: Jakub Łącki
Opis rozwiązania: Jakub Łącki
Dostępna pamięć: 64 MB
`https://oi.edu.pl/pl/archive/amppz/2011/czy`

Bajtazar przechadzał się koło Biblioteki Uniwersyteckiej w Warszawie i na jednej z fasad zobaczył fragment programu opatrzony pytaniem „Czy się zatrzyma?". Problem wyglądał intrygująco, dlatego Bajtazar postanowił zająć się nim po powrocie do domu. Niestety, gdy zapisywał kod na kartce, popełnił błąd i zanotował:

```
while n > 1 do
  if n mod 2 = 0 then
    n := n/2
  else
    n := 3 · n + 3
```

Bajtazar próbuje teraz ustalić, dla jakich wartości początkowych zmiennej n zapisany przez niego program zatrzyma się. Zakładamy przy tym, że zmienna n ma nieograniczony rozmiar, tj. może przyjmować dowolnie duże wartości.

Wejście

Pierwszy i jedyny wiersz wejścia zawiera jedną liczbę całkowitą n ($2 \leqslant n \leqslant 10^{14}$), dla której należy sprawdzić, czy podany program zatrzyma się.

Wyjście

W pierwszym i jedynym wierszu wyjścia Twój program powinien wypisać jedno słowo `TAK`, jeśli program zatrzyma się dla podanej wartości n, lub `NIE` w przeciwnym przypadku.

Przykład

Dla danych wejściowych:

```
4
```

poprawnym wynikiem jest:

```
TAK
```

Rozwiązanie

W treści zadania podany jest fragment programu, którego zachowanie powinniśmy przeanalizować. Skoro kod źródłowy jest nam znany, możemy po prostu wstawić go do naszego rozwiązania i wykonać. Jeśli uruchomimy kod dla podanej liczby n, a ten zatrzyma się, natychmiast poznajemy odpowiedź. Nieco trudniej jest w przeciwnym przypadku: jak bowiem mieć pewność, że uruchomiony kod nie zatrzymuje się? Zacznijmy od przyjęcia bardzo odważnego (i niespecjalnie uzasadnionego) założenia. Jeśli kod nie zatrzyma się w ciągu, powiedzmy, 100 000 obrotów pętli, to przyjmiemy, że nie zatrzymuje się w ogóle.

Takie podejście łatwo przełożyć na rozwiązanie, jednak ma ono dwie wady. Po pierwsze, skąd wiemy, że wystarczy ograniczyć się do 100 000 iteracji? Po drugie, co gorsza, jeśli liczbę n reprezentować będziemy jako liczbę 64-bitową ze znakiem (czyli na przykład zadeklarujemy ją jako liczbę typu `long long` w języku C++), to wpadniemy w pułapkę. Okazuje się, że nawet jeśli początkowa wartość n, zgodnie z treścią zadania, nie przekracza 10^{14}, to po pewnej liczbie obrotów pętli możemy dostać liczbę znacznie większą. Na przykład dla początkowej wartości $n = 366\,713\,142\,269$ po około 160 iteracjach n osiąga wartość $10\,010\,331\,589\,553\,303\,736$, czyli ponad 10^{19}, podczas gdy w 64-bitowej liczbie ze znakiem reprezentować możemy liczby nie większe niż $2^{63} - 1 \approx 9{,}2234 \cdot 10^{18}$.

Podejdźmy więc do zadania inaczej. Sprawdźmy, jak zachowuje się rozważany kod dla wszystkich n od 1 do 20. Tu warto wspomóc się komputerem. Bardzo szybko zauważymy wtedy, że kod zatrzymuje się dla n równych 1, 2, 4, 8 i 16, a dla pozostałych wartości zapętla się, gdyż od pewnego momentu n przyjmuje kolejno wartości $3, 12, 6, 3, 12, 6, 3, 12, 6, \ldots$ i tak w nieskończoność. Wygląda więc na to, że kod zatrzymuje się dla potęg dwójki. No tak, wystarczy chwila zastanowienia, by przekonać się, że jeśli n jest potęgą dwójki, to nasz kod będzie zmienną n dzielił tak długo przez 2, aż dojdziemy do $n = 1$. Zatem, jeśli n jest potęgą dwójki, nasz kod z pewnością się zatrzyma. W tej chwili zgadywać możemy, że dla pozostałych wartości n kod będzie działał w nieskończoność. Tym razem nasz strzał okazuje się słuszny, jednak dlaczego?

Jeśli początkowa wartość n jest nieparzysta, to wykonamy przypisanie $n := 3 \cdot n + 3$. Łatwo zauważyć, że nowa wartość n będzie podzielna przez 3. Co więcej, od tego momentu n już zawsze będzie podzielne przez 3. Dlaczego? Jeśli n jest podzielne przez 3, to niezależnie od tego, czy wykonamy przypisanie $n := n/2$, czy też $n := 3 \cdot n + 3$, nowa wartość n będzie dalej podzielna przez 3. A więc w tym przypadku nigdy nie osiągniemy $n = 1$! Pozostaje nam do sprawdzenia przypadek, gdy n jest parzyste, jednak nie jest potęgą dwójki. Wówczas jednak będziemy dzielić n przez 2 tak długo, aż stanie się ono nieparzyste. Wiemy jednak, że będzie to liczba nieparzysta większa od 1 (bo jedynkę osiągnęlibyśmy, jedynie startując od potęgi dwójki). A skoro n stanie się nieparzyste, zgodnie z wcześniejszą analizą, kod nie zatrzyma się.

Całe zadanie sprowadza się więc do sprawdzenia, czy n jest potęgą dwójki. Ten warunek sprawdzić możemy bardzo prosto: tak długo, jak n jest podzielne przez 2, wykonujemy dzielenie. Jeśli po tej procedurze zachodzi $n = 1$, mieliśmy do czynienia z potęgą dwójki. W przeciwnym razie, podane n potęgą dwójki nie było.

Co ciekawe, sprawdzenie, czy n jest potęgą dwójki, można wykonać jeszcze szybciej. Posłużymy się dwoma operatorami bitowymi. Wykonują one operacje na

zapisach binarnych liczb i są dostępne w bodaj wszystkich popularnych językach programowania. Operator bitowy **and** (w języku C++ zapisywany symbolem `&`) działa następująco: wynikiem operacji a **and** b jest liczba, której zapis binarny ma jedynki dokładnie na tych pozycjach, na których jedynki ma zarówno liczba a jak i liczba b. Z kolei operator **xor** (w języku C++ oznaczany symbolem `^`) oblicza liczbę, w której jedynki stoją na tych pozycjach, na których zapisy binarne dwóch liczb się różnią. Aby sprawdzić, czy podana liczba n jest potęgą dwójki, wystarczy sprawdzić, czy

$$(n \textbf{ xor } (n-1)) \textbf{ and } n \text{ jest równe } n.$$

Dlaczego ten trik działa? To już pozostawiamy jako ćwiczenie.

Problem Collatza

Na fasadzie Biblioteki Uniwersyteckiej w Warszawie rzeczywiście znaleźć można kawałek kodu podobny do tego, z którym mamy do czynienia w tym zadaniu. Niestety, jak czytamy w treści zadania, Bajtazar w trakcie zapisywania kodu na kartce popełnił błąd. Kod na bibliotece wygląda następująco:

```
while n > 1 do
  if n mod 2 = 0 then
    n := n/2
  else
    n := 3 · n + 1
```

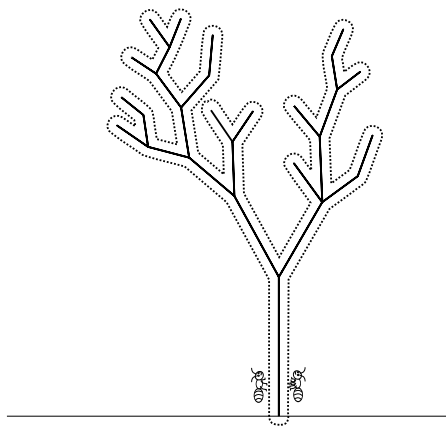
W ostatnim wierszu mamy więc $n := 3 \cdot n + 1$ zamiast $n := 3 \cdot n + 3$. Choć różnica pomiędzy dwoma kodami wygląda na drobną, ma ona spore znaczenie. W naszym zadaniu udało nam się wyznaczyć wszystkie liczby, dla których podany kod zatrzymuje się. Analogiczny problem dla kodu, który znajduje się na ścianie biblioteki, jest znacznie trudniejszy. Już w 1937 roku niemiecki matematyk Lothar Collatz sformułował hipotezę, że rozważany kod zatrzymuje się dla dowolnego dodatniego n. Nad hipotezą Collatza pracowało wielu matematyków, jednak dotychczas nikomu nie udało się jej rozstrzygnąć. Została ona nawet potwierdzona komputerowo dla wszystkich $n < 20 \cdot 2^{58} \approx 5{,}7646 \cdot 10^{18}$ i powszechnie uważa się, że jest ona prawdziwa, jednak nie istnieje na to żaden formalny dowód. Zatem gdybyśmy w naszym zadaniu rozważali oryginalny problem Collatza, na pytanie *Czy się zatrzyma?* nie potrafilibyśmy udzielić jednoznacznej odpowiedzi.

D

Drzewo i mrówki

★★★★

Autor zadania: Szymon Acedański
Opis rozwiązania: Szymon Acedański, Tomasz Idziaszek
Dostępna pamięć: 6 MB
`https://oi.edu.pl/pl/archive/amppz/2011/drz`



Informatycy lubią drzewa. Mrówki także lubią drzewa. Niech więc będzie dane drzewo, po którym chodzą dwie mrówki — Lewa Mrówka i Prawa Mrówka — w sposób pokazany na powyższym rysunku (po ścieżce oznaczonej kropkowaną linią). Rozpoczynają one swoją wędrówkę na początku pnia, po przeciwnych jego stronach. Idąc od korzenia (w górę), Lewa Mrówka pokonuje każdą krawędź drzewa w dwie sekundy, zaś idąc w kierunku korzenia (w dół) — w sekundę. Prawa Mrówka jest od niej dwa razy szybsza. W momencie, gdy mrówki spotykają się, obydwie zawracają. Jeśli któraś z nich zejdzie z drzewa na ziemię, natychmiast zaczyna wchodzić na drugą stronę pnia. Poza tym mrówki są tak malutkie, że nawet mikroskop nie wystarczyłby do ich zobaczenia (na rysunku celowo zostały powiększone). Twoim zadaniem jest napisanie programu, który obliczy, po jakim czasie mrówki zawrócą po raz drugi.

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita t ($1 \leqslant t \leqslant 1000$) oznaczająca liczbę zestawów testowych opisanych w dalszej części wejścia.

Opis każdego zestawu składa się z dwóch wierszy. W pierwszym z nich znajduje się parzysta liczba n ($2 \leqslant n \leqslant 100\,000\,000$) oznaczająca liczbę krawędzi drzewa. W drugim wierszu znajduje się opis drzewa. Jest to napis długości $\frac{n}{2}$ reprezentujący liczbę binarną $2n$-bitową zapisaną w systemie szesnastkowym (przy użyciu cyfr oraz małych liter od a do f). Liczba ta opisuje, jak wygląda obejście całego drzewa przez Lewą Mrówkę przy założeniu, że Prawa Mrówka stoi w miejscu. Kolejne bity tej liczby (od lewej) oznaczają, czy idąc po kolejnych krawędziach na swojej ścieżce, Lewa Mrówka oddala się od korzenia (bit 1), czy też zbliża się do korzenia

(bit 0). Drzewo posiada pień, tzn. dokładnie jedna krawędź wychodzi z korzenia drzewa.

Rozmiar żadnego pliku wejściowego nie przekroczy 50 MB. Uwaga: to jest istotnie więcej niż rozmiar pamięci dostępnej dla Twojego programu.

Wyjście

Twój program powinien wypisać t wierszy z odpowiedziami dla kolejnych zestawów testowych. Odpowiedzią dla zestawu jest czas w sekundach, po którym mrówki zawrócą po raz drugi, wypisany jako nieskracalny ułamek `p/q` (bez spacji wokół `/`), gdzie p i q to liczby całkowite dodatnie. Jeśli odpowiedź jest liczbą całkowitą, to oczywiście $q = 1$.

Przykład

Dla danych wejściowych:	poprawnym wynikiem jest:

`1`
`28`
`fb1da30d1b7230`
	`282/5`

Przykład odpowiada rysunkowi, a ciąg bitów wygląda następująco:

```
1111 1011 0001 1101 1010 0011 0000 1101 0001 1011 0111 0010 0011 0000
```

Rozwiązanie

Ograniczenie na rozmiar wejścia zostało dobrane tak, aby uniemożliwić przechowanie całego drzewa w pamięci. Rozwiązanie wzorcowe wczytuje i na bieżąco przetwarza dane wejściowe, nie zapamiętując ich. Szkic rozwiązania wygląda tak:

- Zaczynamy od wczytania rozmiaru drzewa. Dzięki temu na starcie znamy łączną liczbę odcinków „w górę” i „w dół”.
- W trakcie wczytywania danych symulujemy ruch Lewej Mrówki do momentu pierwszego spotkania. Dla każdej krawędzi łatwo sprawdzamy, czy doszło na niej do spotkania: skoro wiemy, jaką odległość w górę i w dół pokonała Lewa Mrówka, znamy drogę, jaką musiała pokonać Prawa.
- Po spotkaniu mrówki zaczynają iść w kierunku korzenia. Wczytujemy wejście dalej, tym razem symulując marsz Prawej Mrówki. Okazuje się, że jako pierwsza do korzenia dotrze... Lewa Mrówka. Brzmi to zapewne dość zaskakująco, bo przecież to Prawa Mrówka porusza się szybciej. To zjawisko wyjaśnimy jednak nieco później.
- Po tym, jak Lewa Mrówka dotrze do korzenia, dalej symulujemy marsz Prawej Mrówki aż do momentu drugiego spotkania.

Aby przekonać się, że to zadanie rzeczywiście da się tak rozwiązać, przyjrzyjmy się formalnemu opisowi rozwiązania wzorcowego. Punkt na drzewie możemy opisać przez parę liczb rzeczywistych (a, k), gdzie a oznacza, ile krawędzi musiała przejść Lewa Mrówka od korzenia, żeby dojść do tego punktu (być może przechodząc niektóre krawędzie w obie strony; takie krawędzie liczymy podwójnie), zaś k oznacza wysokość punktu nad poziomem gruntu (liczbę krawędzi od korzenia do tego punktu). Jeżeli Lewa Mrówka krawędź do góry pokonuje w czasie t_g, a krawędź w dół w czasie t_d, to do punktu (a, k) dojdzie w czasie

$$[a, k, t_g, t_d] := \frac{a+k}{2}\, t_g + \frac{a-k}{2}\, t_d.$$

Zatem punkt pierwszego spotkania (a_1, k_1) zostanie osiągnięty, gdy

$$[a_1, k_1, 2, 1] = [2n - a_1, k_1, 1, \tfrac{1}{2}]. \tag{1}$$

Aby znaleźć taki punkt, możemy, wczytując drzewo, symulować ruch Lewej Mrówki. W momencie, gdy stoimy w punkcie (a, k), który jest początkiem pewnej krawędzi, potrafimy sprawdzić, czy spotkanie nastąpi na tej właśnie krawędzi. Niech $b = 1$, jeśli jest to krawędź w górę, i $b = -1$, jeśli jest to krawędź w dół. Gdyby do spotkania miało dojść po przejściu kawałka krawędzi o długości ε, to równość (1) byłaby spełniona dla

$$a_1 = a + \varepsilon, \quad k_1 = k + b\varepsilon,$$

zatem

$$\varepsilon = \frac{6n - 9a - k}{9 + b}.$$

Jeśli więc $0 \leqslant \varepsilon < 1$, to znaleźliśmy punkt pierwszego spotkania. Czas potrzebny na dojście do tego punktu to

$$t_1 = [a_1, k_1, 2, 1] = \frac{3a_1 + k_1}{2}.$$

Jeśli pozwolilibyśmy teraz wrócić mrówkom do korzenia, to Lewa Mrówka zrobiłaby to w czasie $[a_1, -k_1, 2, 1] = t_1 - k_1$, natomiast Prawa — w czasie

$$[2n - a_1, -k_1, 1, \tfrac{1}{2}] = t_1 - \frac{k_1}{2}.$$

Widać zatem, że Lewa Mrówka zawsze zdąży wrócić do korzenia przed Prawą.

Możemy teraz szukać punktu drugiego spotkania. Będziemy to robić, symulując ruch Prawej Mrówki. Jeśli przejdzie ona fragment długości ε' krawędzi wychodzącej z punktu (a', k'), to dojdzie do punktu spotkania (a_2, k_2), o ile będzie spełnione

$$a_2 = a' + \varepsilon', \quad k_2 = k' + b'\varepsilon',$$
$$[a_1, -k_1, 2, 1] + [2n - a_2, k_2, 2, 1] = [a_2 - a_1, k_2 - k_1, 1, \tfrac{1}{2}],$$

tak więc

$$\varepsilon' = \frac{12n - 9(a' - a_1) + (k' - k_1)}{9 - b'}.$$

Jeśli zatem $0 \leqslant \varepsilon' < 1$, to znaleźliśmy drugi punkt spotkania. Czas potrzebny na dojście tam to

$$t_2 = [a_2 - a_1, k_2 - k_1, 1, \tfrac{1}{2}] = \frac{3(a_2 - a_1) + (k_2 - k_1)}{4}.$$

Zatem sumaryczny czas, który upłynie do momentu ponownego spotkania obu mrówek, jest równy

$$t_1 + t_2 = \frac{3(a_1 + a_2) + k_1 + k_2}{4}.$$

Zauważmy, że ε to wielokrotność $\frac{1}{9+b}$, natomiast proste obliczenia pokazują, że ε' to wielokrotność $\frac{9-b}{(9+b)(9-b')}$. Wynika z tego, że wartości a_1, a_2, k_1, k_2 są wielokrotnościami ułamka $\frac{1}{800}$. Można zatem wszystkie obliczenia wykonywać w arytmetyce stałopozycyjnej, używając 64-bitowych zmiennych całkowitoliczbowych, w których przechowujemy wartości przemnożone przez 800.

W efekcie otrzymujemy algorytm, który w każdym kroku wczytuje nową krawędź i wykonuje stałą liczbę rachunków. Działa on zatem w czasie $O(n)$ i stałej pamięci.

E

Eksterminacja świstaków

Autor zadania: Jakub Radoszewski
Opis rozwiązania: Jakub Radoszewski
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2011/eks`

Gnębon Bajtopysk postanowił uprzykrzyć życie bajtockim świstakom. Te sympatyczne zwierzątka zamieszkują w norkach, w górnych partiach pasma Bajtogór Wysokich.

Gnębon odnalazł pewien grzbiet górski, wzdłuż którego w jednej linii jest rozmieszczonych n świstaczych norek (dla ułatwienia kolejne norki wzdłuż grzbietu, patrząc z zachodu na wschód, numerujemy od 1 do n). Diabelski pomysł Gnębona polega na gnębieniu świstaków muzyką z gatunku rock and roll. Aby go zrealizować, nasz bohater zakupił m magnetofonów, w każdym umieścił inny album Bajtelsów i rozmieścił je wszystkie w linii, wzdłuż której są wykopane norki. Wiadomo, że po rozkręceniu głośności w danym magnetofonie na cały regulator, wydobywająca się z niego muzyka nie daje spać świstakom znajdującym się w norkach odległych co najwyżej o l metrów od tego magnetofonu.

Zaniepokojone tą sytuacją świstaki poprosiły Cię o sprawdzenie, w których norkach na pewno nie uda im się tej zimy wyspać. Nie wiedziały, że to jeszcze nie koniec złośliwości Gnębona...

Otóż Gnębon postanowił wywołać jeszcze większe zamieszanie i co jakiś czas przestawiać niektóre magnetofony. Świstakom udało się wykraść tajny plan Gnębona i wiedzą dokładnie, że i-tego dnia rankiem weźmie on magnetofon położony p_i metrów od norki numer 1 i przestawi go w punkt odległy o r_i metrów od tejże norki. Pomóż świstakom sprawdzić, w ilu norkach nie uda im się zasnąć po każdej takiej zamianie.

Wejście

W pierwszym wierszu wejścia znajdują się cztery liczby całkowite n, m, d oraz l ($2 \leqslant n, m \leqslant 500\,000$, $1 \leqslant d \leqslant 500\,000$, $1 \leqslant l \leqslant 10^9$) oznaczające odpowiednio liczbę norek świstaków, liczbę magnetofonów Gnębona, liczbę dni „eksperymentu" Gnębona i zasięg rażenia magnetofonu.

Drugi wiersz wejścia zawiera $n-1$ liczb całkowitych $x_2, x_3, \ldots, x_n$ ($0 < x_2 < x_3 < \ldots < x_n \leqslant 10^9$) oznaczających odległości norek o numerach $2, 3, \ldots, n$ od norki numer 1.

Trzeci wiersz zawiera m liczb całkowitych $z_1, z_2, \ldots, z_m$ ($0 \leqslant z_1 < z_2 < \ldots < z_m \leqslant 10^9$) oznaczających odległości kolejnych magnetofonów od norki numer 1. Wszystkie magnetofony położone są na wschód od tej norki.

Dalej na wejściu następuje d wierszy; i-ty z nich zawiera dwie liczby całkowite p_i oraz r_i ($0 \leqslant p_i, r_i \leqslant 10^9$, $p_i \neq r_i$) oznaczające, że na początku i-tego dnia „zabawy" Gnębon zamierza przestawić magnetofon znajdujący się p_i metrów od norki numer 1 w punkt odległy o r_i metrów na wschód od tej norki. Możesz założyć, że przed wykonaniem takiej operacji na pozycji p_i znajduje się jakiś magnetofon oraz że na pozycji r_i nie ma jeszcze żadnego magnetofonu.

Wyjście

Twój program powinien wypisać $d+1$ wierszy. Wiersz numer i (dla $i = 1, 2, \ldots, d$) powinien zawierać jedną liczbę całkowitą, oznaczającą liczbę norek, w których żaden świstak na pewno się nie wyśpi w nocy *przed* wykonaniem i-tej zamiany. W ostatnim wierszu należy wypisać, w ilu norkach świstaki nie będą mogły spać po ostatniej zamianie.

Przykład

Dla danych wejściowych:	poprawnym wynikiem jest:

`5 3 4 1`
`2 5 6 11`
`2 4 8`
`2 1`
`4 10`
`8 6`
`1 8`
	`2`
	`3`
	`3`
	`5`
	`3`

Rozwiązanie

Cała akcja w zadaniu odbywa się na prostej. Położenia norek świstaków są opisane punktami. Z kolei każdy magnetofon Gnębona odpowiada pewnemu przedziałowi. Wszystkie przedziały mają taką samą długość $2l$, więc do opisu każdego przedziału również wystarcza nam znajomość jednego punktu — w treści zadania jest to środek przedziału. Położenia norek nie zmieniają się w czasie, natomiast Gnębon przestawia swoje magnetofony. Naszym zadaniem jest symulowanie posunięć Gnębona i obliczenie, po każdym z d przestawień magnetofonów, ile różnych norek pokrywają łącznie przedziały odpowiadające magnetofonom.

Jak się okazuje, całe zadanie polega na doborze odpowiednich struktur danych, które pozwolą przeprowadzić symulację. Odpowiednich, czyli możliwie najprostszych, a zarazem wystarczająco szybkich: celujemy w złożoność operacji logarytmiczną względem liczby norek i magnetofonów, czyli n oraz m.

Kluczowe założenie jest takie, że chcielibyśmy po każdym posunięciu Gnębona pamiętać liczbę norek pokrywanych przez przedziały magnetofonów. Wypisywanie wyników jest wtedy trywialne. Jak to często bywa w takich zadaniach, przeniesienie magnetofonu możemy podzielić na dwa etapy: usunięcie magnetofonu i wstawienie go na nowej pozycji. Każdy z tych etapów jest w pewnym sensie lokalny: to, które norki zaczynają lub przestają być pokryte, zależy tylko od przedziałów sąsiadujących bezpośrednio z przedziałem wstawianego bądź usuwanego magnetofonu. Dzieje się tak, ponieważ wszystkie przedziały mają tę samą długość. Wykorzystamy tę własność do konstrukcji efektywnego rozwiązania.

Do przechowywania magnetofonów potrzebujemy struktury danych obsługującej operacje: wstawienia i usunięcia elementu oraz znalezienia najbliższego lewego i prawego sąsiada danego elementu w strukturze. Do tego celu świetnie nadaje się dowolna struktura słownikowa zaimplementowana z użyciem zrównoważonych drzew poszukiwań binarnych, np. `set` z biblioteki standardowej języka C++ (treść

zadania gwarantuje, że żadne dwa magnetofony nie będą nigdy umieszczone w tym samym miejscu). Z kolei położenia norek nie zmieniają się, więc możemy je przechowywać po prostu w tablicy posortowanej rosnąco — dokładnie tak, jak są one podane na wejściu. Struktury danych do przechowywania magnetofonów oraz norek oznaczmy odpowiednio przez M oraz N.

Prześledźmy teraz, co dzieje się, gdy wstawiamy jakiś magnetofon. Załóżmy, że zostanie on umieszczony na pozycji x, a jego bezpośredni sąsiedzi znajdują się na pozycjach x' oraz x'' ($x' < x < x''$). Wtedy musimy wstawić x do słownika M, a do wyniku dodać wszystkie norki pokryte przez przedział $[x-l, x+l]$, które nie były pokrywane przez przedziały $[x'-l, x'+l]$ oraz $[x''-l, x''+l]$. Są to dokładnie norki zawarte w przedziale

$$[\max(x-l, x'+l+1),\ \min(x+l, x''-l-1)]. \tag{1}$$

Zauważmy, że liczbę tych norek możemy wyznaczyć za pomocą dwóch wyszukiwań binarnych w tablicy N. Każde z wyszukiwań binarnych można zrealizować za pomocą funkcji `lower_bound` z biblioteki standardowej.

Operacja usunięcia magnetofonu jest analogiczna: tym razem usuwamy odpowiedni element ze słownika M i odejmujemy od wyniku wszystkie norki zawarte w przedziale postaci (1). Aby każdy magnetofon miał dobrze zdefiniowany poprzednik i następnik, do słownika M można dodać stałych strażników umieszczonych na pozycjach $-l-1$ oraz 10^9+l+1 (współrzędne punktów występujących w zadaniu należą do przedziału $[0, 10^9]$).

Każde z d posunięć Gnębona symulujemy zatem w czasie $O(\log m + \log n)$, przy czym pierwszy logarytm pochodzi z operacji na słowniku M, a drugi z wyszukiwania binarnego w tablicy N. Dla zawodników biegłych w używaniu biblioteki standardowej C++ całe rozwiązanie jest całkiem proste w implementacji.

Frania

Autor zadania: Szymon Acedański
Opis rozwiązania: Jakub Radoszewski
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2011/fra`

Kilkoro przyjaciół postanowiło zrobić pranie. Wszyscy oni są zupełnie porządni, wobec czego każdego dnia zużywają dokładnie jedną parę skarpetek i jedną koszulkę. Wrzucili więc wszystkie zużyte skarpetki i koszulki do swojej wysłużonej pralki i zaczęli zastanawiać się nad strategią ich wysuszenia. Żeby wyeliminować zamieszanie, postanowili, że:

- każda skarpetka będzie przypięta do sznurka jedną klamerką,
- każda koszulka będzie przypięta trzema klamerkami,
- wszystkie skarpetki jednej osoby będą przypięte klamerkami tego samego koloru,
- wszystkie koszulki jednej osoby będą przypięte klamerkami tego samego koloru,
- rzeczy należące do dwóch różnych osób nie mogą być przypięte klamerkami tego samego koloru,
- poza tym użyją najmniejszej możliwej liczby kolorów klamerek.

Tak ustaliwszy, wysypali wszystkie posiadane klamerki na podłogę i skrzętnie policzyli, ile mają klamerek każdego z kolorów. Niestety nie potrafili wykombinować, kto powinien użyć których. Napisz program, który im pomoże.

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite n oraz k ($2 \leqslant n, k \leqslant 1\,000\,000$) oznaczające liczbę przyjaciół oraz liczbę dostępnych kolorów klamerek. W drugim wierszu znajduje się n liczb całkowitych $d_1, d_2, \ldots, d_n$ oznaczających, przez ile dni poszczególni przyjaciele gromadzili pranie ($1 \leqslant d_i \leqslant 1\,000\,000$). W trzecim wierszu znajduje się k liczb całkowitych $l_1, l_2, \ldots, l_k$ oznaczających, ile jest klamerek poszczególnych kolorów ($1 \leqslant l_i \leqslant 4\,000\,000$).

Wyjście

Twój program powinien wypisać minimalną liczbę kolorów klamerek potrzebnych do rozwieszenia prania. Jeśli rozwieszenie prania w opisany sposób nie jest możliwe, Twój program powinien wypisać jedno słowo `NIE`.

Przykład

Dla danych wejściowych:

```
2 4
3 4
20 10 8 10
```

poprawnym wynikiem jest:

```
3
```

natomiast dla danych:

```
3 8
5 4 3
14 14 14 14 14 14 14 14
```

poprawnym wynikiem jest:

```
NIE
```

Wyjaśnienie pierwszego przykładu: Pierwsza osoba potrzebuje 6 klamerek na skarpetki i 9 na koszulki. Druga osoba potrzebuje 8 klamerek na skarpetki i 12 na koszulki. Druga osoba powinna użyć klamerek pierwszego koloru zarówno do powieszenia skarpetek, jak i koszulek. Pierwsza osoba może wówczas użyć na przykład klamerek drugiego i czwartego koloru.

Rozwiązanie

Treść zadania opowiada o n osobach robiących pranie. Do powieszenia swojego prania i-ta osoba potrzebuje $5d_i$ klamerek. Mogą to być klamerki tego samego koloru albo też może to być $3d_i$ klamerek jednego koloru (koszulki) i $2d_i$ klamerek innego koloru (skarpetki). Mamy do dyspozycji k kolorów klamerek, przy czym jest dokładnie l_j klamerek koloru j. Kolory klamerek przydzielonych różnym osobom muszą być różne. Chcemy przypisać osobom klamerki tak, aby użyć jak najmniej różnych kolorów klamerek. Łącznie wykorzystamy między n a $2n$ kolorów, a im więcej osób będzie miało przypisany tylko jeden kolor klamerek, tym lepiej. Musimy także umieć stwierdzić, czy rozwiązanie w ogóle istnieje.

Spróbujmy podejść do rozwiązania zachłannie, zapraszając kolejne osoby, aby mogły wybrać sobie klamerki ze zbioru tych niewybranych przez poprzedników. Osoby z większą ilością prania są bardziej wybredne, więc intuicyjnie rzecz biorąc, opłaca się nam zapraszać je wcześniej, by miały jak największy wybór. Wydaje się też sensownym, aby każda osoba, jeśli tylko jest to możliwe, wybierała jeden kolor klamerek. Ponadto, jeśli w którymś momencie jest więcej niż jedna możliwość wyboru koloru klamerek, to oczywiście opłaca się wybrać pasujący kolor mający jak najmniej klamerek.

Powyższe intuicje są podstawą następującego algorytmu. Rozpatrujemy osoby w kolejności nierosnących d_i i dla każdej z nich:

- jeśli istnieje kolor zawierający co najmniej $5d_i$ klamerek, to spośród takich kolorów wybieramy ten o najmniejszej liczbie klamerek,

- w przeciwnym wypadku, gdy istnieją dwa kolory zawierające odpowiednio co najmniej $2d_i$ oraz $3d_i$ klamerek, to spośród nich wybieramy takie o najmniejszych liczbach klamerek,

- jeśli zaś nie zachodzi żaden z powyższych warunków, to kończymy algorytm z odpowiedzią negatywną.

Okazuje się, że powyższy algorytm poprawnie rozwiązuje nasze zadanie.

Uzasadnienie poprawności

Musimy wykazać, że jeśli rozwiązanie istnieje, to nasz algorytm zachłanny je znajdzie, a co więcej, znajdzie rozwiązanie używające najmniejszej możliwej liczby kolorów klamerek.

Załóżmy zatem, że rozwiązanie istnieje, i niech OPT będzie pewnym rozwiązaniem optymalnym, czyli używającym możliwie najmniejszej liczby kolorów klamerek. Przez ALG oznaczmy zaś przyporządkowanie wyznaczone przez algorytm zachłanny. (Teoretycznie może to być przyporządkowanie częściowe, jeśli nie udało się nam przydzielić odpowiedniej liczby klamerek jednej z osób i z tego powodu przerwaliśmy wykonywanie algorytmu.) Wykażemy, że jeśli $\mathrm{OPT} \neq \mathrm{ALG}$, to możemy przekształcić OPT w ALG, zachowując jego poprawność i nie zmieniając liczby wykorzystanych kolorów klamerek. Stąd od razu wywnioskujemy optymalność rozwiązania ALG.

Załóżmy bez straty ogólności, że osoby są uporządkowane w kolejności rozpatrywania przez algorytm zachłanny, czyli w szczególności $d_1 \geqslant d_2 \geqslant \ldots \geqslant d_n$. Nasz dowód będzie przebiegał krokowo; w każdym kroku będziemy modyfikować rozwiązanie OPT, zwiększając liczbę kolejnych osób, które w OPT i ALG są obsługiwane tak samo.

Niech i będzie numerem pierwszej osoby, której OPT i ALG w różny sposób przypisały kolory klamerek. Wykażemy, że możemy zmodyfikować rozwiązanie OPT tak, by wszystkie osoby $1, \ldots, i$ były obsługiwane tak samo jak w ALG.

Skoro OPT przypisało osobie i jakieś kolory klamerek, to nie może być tak, że w ALG nie udało się tej osoby obsłużyć. Faktycznie w chwili jej rozważania rozwiązanie zachłanne mogło wybrać te same kolory klamerek, które zostały jej przyporządkowane w OPT. Analogicznie nie jest możliwe, by OPT przypisało tej osobie tylko jeden kolor klamerek, a ALG dwa.

Wiemy jednak, że oba rozwiązania różnią się na osobie i. Rozważmy najpierw przypadek, w którym oba rozwiązania przypisują jej taką samą liczbę kolorów klamerek, ale są to różne kolory. Przyjmijmy, że jest to w obu rozwiązaniach jeden kolor; dowód dla dwóch kolorów jest analogiczny. Niech w OPT będzie to kolor p, a w ALG kolor a. Skoro ALG wybiera zawsze pasujący kolor o najmniejszej możliwej liczbie klamerek, mamy $l_a \leqslant l_p$. Jeśli rozwiązanie OPT nie wykorzystuje koloru a, to możemy w tym rozwiązaniu zamienić osobie i kolor klamerek z p na a. Jeśli zaś OPT wykorzystuje klamerki koloru a, przydzielając je pewnej osobie j, to mamy $j > i$ (bo dla $j < i$ OPT przydziela takie same kolory jak ALG) i możemy zamienić w OPT kolory klamerek osób i oraz j. Liczba klamerek koloru p wystarcza dla osoby j, bo skoro $j > i$, to $d_j \leqslant d_i$. Z drugiej strony liczba klamerek koloru a wystarcza dla osoby i, bo tak przydziela te klamerki ALG.

Pozostał nam ostatni przypadek, w którym rozwiązanie OPT przydziela osobie i dwa kolory, powiedzmy p oraz q (odpowiednio na koszulki i skarpetki), na-

tomiast w ALG otrzymuje ona tylko jeden kolor, dajmy na to a. Gdyby w OPT kolor a był wolny, to moglibyśmy przydzielić go osobie i zamiast kolorów p i q, tym samym zmniejszając liczbę wykorzystanych kolorów — nie jest to więc możliwe, bo przeczyłoby optymalności OPT. To oznacza, że kolor a w rozwiązaniu OPT musiał być zajęty, powiedzmy, przez osobę j (znowu $j > i$). Podobnie jak poprzednio, chcemy oddać ten kolor osobie i. Mamy różne przypadki w zależności od tego, ile klamerek koloru a ma osoba j. Jeśli jest to $2d_j$ klamerek, to możemy zamiast klamerek koloru a dać jej klamerki koloru q (jako że $d_j \leqslant d_i$). Jeśli $3d_j$, to zamiast koloru a dajemy jej kolor p. Można zauważyć, że w obu tych przypadkach liczba wykorzystanych kolorów zmalała, co przeczy optymalności OPT. Zatem te przypadki nie mogą mieć miejsca (choć i tak potrafimy sobie z nimi poradzić). W jedynej sytuacji, która faktycznie może wystąpić, osoba j ma w OPT $5d_j$ klamerek koloru a. Wtedy przydzielamy jej $3d_j$ klamerek koloru p i $2d_j$ koloru q. W efekcie liczba wykorzystanych kolorów nie wzrasta, a osoba i ma takie samo przypisanie jak w ALG. To kończy dowód.

Implementacja

Na koniec warto zastanowić się chwilę nad tym, jak efektywnie zaimplementować opisane rozwiązanie. Osoby występujące w zadaniu musimy posortować nierosnąco względem liczb d_i; robimy to w czasie $O(n \log n)$. Do reprezentowania klamerek poszczególnych kolorów potrzebna jest struktura danych, za pomocą której możemy wykonywać dwa typy operacji:

- znajdź najmniej liczny kolor zawierający co najmniej x klamerek,
- usuń wszystkie klamerki danego koloru.

Zarówno kontener `set` z biblioteki standardowej języka C++, jak i statyczne drzewo przedziałowe (lub też drzewo licznikowe, lub drzewo Fenwicka) umożliwia wykonywanie każdej z tych operacji w czasie $O(\log k)$. Inicjowanie struktury danych wymaga czasu $O(k)$. W ten sposób otrzymujemy implementację rozwiązania zachłannego w złożoności czasowej $O(k + n(\log k + \log n))$.

G

Generator bitów

★★★★

Autor zadania: Tomasz Idziaszek
Opis rozwiązania: Tomasz Idziaszek
Dostępna pamięć: 64 MB
`https://oi.edu.pl/pl/archive/amppz/2011/gen`

Bajtazar lubi bawić się generatorem bitów losowych (a w zasadzie pseudolosowych), który jest dostarczany przez oprogramowanie w jego komputerze. Zasada działania tego generatora jest bardzo prosta. Przy starcie komputera w magiczny sposób jest wybierana liczba całkowita z zakresu od 0 do $m-1$, którą nazwiemy *ziarnem* generatora i będziemy zapisywać w zmiennej z. Następnie w celu wygenerowania losowego bitu wywoływana jest poniższa funkcja, która oblicza nowe ziarno i na jego podstawie wyznacza zwracany bit:

$z := \lfloor (z \cdot a + c)/k \rfloor \bmod m$
if $z < \lfloor m/2 \rfloor$ **then**
  return 0
else
  return 1

Używane wartości a, c, k są pewnymi stałymi. Bajtazar wywołał powyższą funkcję n razy i uzyskał ciąg kolejnych bitów $b_1, b_2, \ldots, b_n$. Zastanawia się teraz, ile możliwych wartości mogło przyjąć początkowe ziarno generatora.

Wejście

W pierwszym wierszu wejścia znajduje się pięć liczb całkowitych a, c, k, m i n ($0 \leqslant a, c < m$, $1 \leqslant k < m$, $2 \leqslant m \leqslant 1\,000\,000$, $1 \leqslant n \leqslant 100\,000$). W drugim wierszu znajduje się n-literowy napis złożony z cyfr 0 i 1; i-ta cyfra napisu oznacza bit b_i.

Wyjście

Należy wypisać jedną liczbę całkowitą, która oznacza liczbę liczb z zakresu od 0 do $m-1$, które mogły być początkowym ziarnem generatora bitów.

Przykład

Dla danych wejściowych:

```
3 6 2 9 2
10
```

poprawnym wynikiem jest:

```
4
```

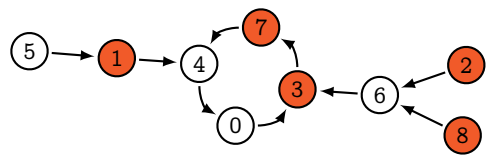
Wyjaśnienie przykładu: Ziarnem generatora bitów mogło być 1, 2, 7 lub 8.

Rozwiązanie

Choć na pierwszy rzut oka może tego nie widać, do rozwiązania zadania przydatne będzie nieco wiedzy z zakresu teorii grafów oraz algorytmów tekstowych. Grafy posłużą nam do przedstawienia przestrzeni stanów generatora, natomiast do poszukiwania początkowego ziarna generatora wykorzystamy słownik podsłów bazowych albo (w szybszym rozwiązaniu) automat wyszukujący wzorzec w tekście.

Stan generatora bitów jest opisywany przez jedną liczbę całkowitą z z zakresu od 0 do $m-1$. Każdorazowe wywołanie funkcji, której kod przedstawiono w treści zadania, powoduje zmianę stanu generatora na $f(z) = \lfloor (z \cdot a + c)/k \rfloor \bmod m$ oraz wygenerowanie bitu $b(z)$. Bit $b(z)$ to 0, jeśli $f(z) < \lfloor m/2 \rfloor$, lub 1 w przeciwnym przypadku. Ponieważ liczby a, c i k opisujące funkcję f są ustalone, zarówno wygenerowany bit $b(z)$ jak i nowy stan generatora $f(z)$ zależą jedynie od aktualnego stanu z.

Generator bitów możemy więc równie dobrze przedstawić jako graf skierowany (patrz rysunek 1). Ma on m wierzchołków $v_0, v_1, \ldots, v_{m-1}$ reprezentujących możliwe stany generatora, a z każdego wierzchołka v_i istnieje dokładnie jedna krawędź skierowana do wierzchołka $v_{f(i)}$. Dodatkowo każdemu wierzchołkowi v_i przypisana jest etykieta $b(i)$.



Rysunek 1. Skierowany graf o 9 wierzchołkach modelujący generator bitów z testu przykładowego. Wyróżnione zostały wierzchołki o etykietach 1.

Oznaczmy przez $p(v_i, n)$ ciąg etykiet na n-wierzchołkowej ścieżce zaczynającej się w wierzchołku v_i. Zauważmy, że ścieżka ta jest wyznaczona jednoznacznie, gdyż z każdego wierzchołka wychodzi dokładnie jedna krawędź. Nasze zadanie możemy sformułować następująco: szukamy liczby takich wierzchołków v_i, że ciąg etykiet $p(v_i, n)$ jest identyczny z ciągiem bitów $b_1, \ldots, b_n$ danym na wejściu.

Algorytm o złożoności $O(nm)$ jest tutaj oczywisty (i nie wymaga nawet powyższego sprowadzenia zadania do postaci grafowej). Dla każdego wierzchołka v_i wystarczy wygenerować ciąg $p(v_i, n)$ i sprawdzić, czy jest on równy $b_1, \ldots, b_n$.

Słownik podsłów bazowych

Szybszy algorytm uzyskamy, korzystając ze struktury danych zwanej *słownikiem podsłów bazowych**. Dla każdej potęgi dwójki 2^j, gdzie $0 \leqslant j \leqslant N$ i $N = \lfloor \log_2 n \rfloor$, będziemy nadawać *nazwy* ciągom etykiet o długości 2^j w taki sposób, by dwa ciągi dostały tę samą nazwę dokładnie wtedy, gdy są równe. Nazwy będą liczbami z przedziału od 0 do $m-1$, dzięki czemu ciągi o długości 2^j będziemy mogli porównywać w czasie stałym. Przez $nazwa[v_i, 2^j]$ oznaczymy nazwę odpowiadającą ciągowi etykiet $p(v_i, 2^j)$.

*Wykorzystanie tej struktury do wyszukiwania powtarzających się podsłów w pojedynczym słowie jest opisane w rozwiązaniu zadania *Powtórzenia* z VII Olimpiady Informatycznej.

Ponieważ podstawą do zbudowania naszej struktury nie jest pojedyncze słowo, ale graf, będziemy potrzebowali efektywnej metody wyznaczania dalszych wierzchołków na ścieżkach. Oznaczmy przez $krok[v_i, l]$ wierzchołek, w którym znajdziemy się po przejściu l krawędzi, zaczynając z wierzchołka v_i. W pierwszej fazie będziemy obliczać wartości $krok[v_i, l]$ dla l będących potęgami dwójki.

Konstrukcja słownika podsłów bazowych jest następująca. Dla $j = 0$ przyjmujemy po prostu $nazwa[v_i, 1] = b(i)$ oraz $krok[v_i, 1] = f(v_i)$. Dla $j = 1, \ldots, N$ nazwy dla ciągów długości 2^j wyznaczamy na podstawie nazw dla ciągów dwa razy krótszych. Na początek dla każdego wierzchołka v_i ciągowi etykiet $p(v_i, 2^j)$ nadajemy tymczasową nazwę

$$\big(nazwa[v_i, 2^{j-1}],\ nazwa[krok[v_i, 2^{j-1}], 2^{j-1}]\big),$$

która jest parą liczb z przedziału od 0 do $m - 1$. Zauważmy, że druga liczba w tej parze to nazwa fragmentu ciągu $p(v_i, 2^j)$, który ma długość 2^{j-1} i zaczyna się dokładnie po 2^{j-1} pierwszych wyrazach tegoż ciągu. W drugim kroku sortujemy wszystkie te pary leksykograficznie w czasie $O(m)$ (możemy zastosować sortowanie pozycyjne) i przechodzimy posortowaną listę par, nadając już właściwe nazwy będące pojedynczymi liczbami — liczy się tylko to, by takim samym parom przypisać taką samą liczbę z zakresu od 0 do $m - 1$. Z kolei wartości $krok[\cdot, 2^j]$ wyznaczamy zgodnie ze wzorem:

$$krok[v_i, 2^j] = krok[krok[v_i, 2^{j-1}], 2^{j-1}].$$

Słownik podsłów bazowych pozwoli nam na szybkie wyznaczanie nazw dla ciągów etykiet długości n. Nazwą dla ciągu $p(v_i, n)$ jest para liczb. Pierwsza liczba w tej parze to nazwa ciągu składającego się z 2^N pierwszych wyrazów $p(v_i, n)$. Z kolei druga liczba to nazwa ciągu składającego się z 2^N ostatnich wyrazów $p(v_i, n)$. Ponieważ $2^N + 2^N > n$, taka para jednoznacznie identyfikuje ciąg. Formalnie nazwą ciągu $p(v_i, n)$ jest $(nazwa[v_i, 2^N], nazwa[krok[v_i, n - 2^N], 2^N])$. W tym przypadku $n - 2^N$ może nie być potęgą dwójki, dlatego wartość $krok[v_i, n - 2^N]$ wyznaczamy w czasie $O(\log n)$: startujemy z wierzchołka $v = v_i$ i dla każdej potęgi dwójki 2^j w rozwinięciu binarnym liczby $n - 2^N$ wykonujemy $v := krok[v, 2^j]$.

Pozostaje pokazać, jak wykorzystać słownik podsłów bazowych do poszukiwania początkowych ziaren generatora bitów. Dla uproszczenia opisu dodajmy do grafu n wierzchołków $u_1, \ldots, u_n$, gdzie wierzchołek u_i ma etykietę b_i oraz krawędź do wierzchołka $u_{\min(i+1,n)}$. Obliczenie nazw dla wszystkich $m + n$ wierzchołków grafu możemy przeprowadzić w czasie $O((m+n) \log n)$. Następnie wystarczy policzyć wierzchołki v_i, dla których nazwy ciągów etykiet $p(v_i, n)$ i $p(u_1, n)$ są równe.

Powyższy algorytm ma złożoność czasową $O((m+n) \log n)$. Niestety taka sama jest też jego złożoność pamięciowa, co przy ograniczeniach z treści zadania powoduje, że program nie mieści się w pamięci. Złożoność pamięciową możemy jednak prosto zmniejszyć do $O(m + n)$. W tym celu podczas budowania słownika podsłów bazowych będziemy pamiętali jedynie dwa ostatnie wiersze (odpowiadające potęgom 2^{j-1} oraz 2^j) oraz na bieżąco obliczali wartości $krok[\cdot, n - 2^N]$.

Automat wyszukujący wzorzec w drzewach

Zadanie można też rozwiązać w złożoności czasowej $O(m+n)$. W tym celu wykorzystamy trochę narzędzi z zakresu algorytmów tekstowych.

Dla ustalonego słowa $x = x_1x_2\ldots x_n$ (wzorca) możemy zbudować *automat wyszukujący wzorzec* w słowie $y = y_1y_2\ldots y_m$ (tekście). Automat ten wczytuje kolejne litery tekstu y i w każdym momencie znajduje się w jednym z $n+1$ stanów $s_0,\ldots,s_n$. A konkretnie: automat po wczytaniu ciągu liter $y_1\ldots y_j$ znajduje się w stanie s_i, jeśli, mówiąc nieformalnie, udało się dopasować dokładnie i liter wzorca do końcówki wczytanego dotąd ciągu. Innymi słowy, najdłuższy sufiks tego ciągu, który jest jednocześnie prefiksem wzorca x, ma długość i (czyli $y_{j-i+1}\ldots y_j = x_1\ldots x_i$ oraz nie istnieje większe i o tej własności). Wszystkie informacje potrzebne do działania automat przechowuje w tabeli przejść δ: automat, który znajduje się w stanie s_i, po przeczytaniu z wejścia litery a przechodzi do stanu $\delta(s_i,a)$. Przykładowo dla $0 \leqslant i < n$ mamy $\delta(s_i,x_{i+1}) = s_{i+1}$.

Teraz, aby wyszukać wzorzec x w tekście y, wystarczy ustawić aktualny stan automatu na s_0, a następnie wczytywać kolejne litery słowa y. Za każdym razem, gdy automat znajdzie się w stanie s_n, wiemy, że znaleźliśmy nowe wystąpienie słowa x (a dokładniej ostatnio wczytana litera jest ostatnią w tym wystąpieniu).

Taki automat można skonstruować w czasie $O(nA)$, gdzie A jest liczbą różnych liter, które mogą wystąpić we wzorcu i w tekście. Nie powinno to sprawić dużo kłopotu tym, którzy znają algorytm Knutha–Morrisa–Pratta. Ten algorytm wyszukiwania wzorca w tekście w trakcie działania utrzymuje stan automatu, choć nie konstruuje w jawny sposób jego tabeli przejść. Z kolei wyszukiwanie wzorca w tekście długości m działa w czasie $O(m)$.

Automat wyszukujący wzorzec możemy też wykorzystać do efektywnego wyszukiwania wzorca w drzewie. Problem jest następujący: mamy ukorzenione drzewo o m wierzchołkach, które są etykietowane literami. Chcemy znaleźć wszystkie ścieżki zaczynające się w dowolnym wierzchołku i idące w dół drzewa, których kolejne wierzchołki mają etykiety tworzące szukane słowo x. Rozwiązanie korzystające z automatu jest proste: przypisujemy wierzchołkom drzewa stany automatu. Jeśli etykietą korzenia jest a, korzeniowi przypisujemy stan $\delta(s_0,a)$. Następnie wierzchołkowi o etykiecie a, którego ojciec ma przypisany stan s_i, przypisujemy stan $\delta(s_i,a)$. Liczba wystąpień wzorca x w drzewie to liczba wierzchołków ze stanem s_n. Całość działa w czasie $O(nA+m)$ i takiej samej złożoności pamięciowej*.

Jesteśmy już gotowi, aby powrócić do naszego zadania. Przypomnijmy, że jest nim znalezienie wszystkich wystąpień słowa długości n w etykietowanym grafie o m wierzchołkach, w którym z każdego wierzchołka wychodzi dokładnie jedna krawędź. Możliwe litery to 0 i 1, więc w tym przypadku $A = 2$.

Zauważmy, że każdą słabo spójną składową grafu możemy rozważyć osobno, a następnie zsumować uzyskane wyniki. Ponieważ z każdego wierzchołka wychodzi

*Kuszącym pomysłem może się wydawać bezpośrednie użycie algorytmu Knutha–Morrisa–Pratta na drzewie. W końcu również pozwala on wyznaczać stany automatu wyszukującego wzorzec i działa w czasie liniowym. Problem w tym, że choć łączny czas działania tego algorytmu jest liniowy, to wyznaczenie *pojedynczego* przejścia po dopasowaniu k liter wzorca może zająć czas $\Theta(k)$. Dla drzew o specyficznym kształcie taki powolny krok może mieć miejsce jednocześnie w bardzo wielu wierzchołkach na tym samym poziomie drzewa. W takiej sytuacji, mimo że na każdej ścieżce idącej w dół drzewa łączny czas działania algorytmu jest liniowy, czas działania algorytmu na całym drzewie może być kwadratowy względem rozmiaru drzewa. Tym samym sam algorytm Knutha–Morrisa–Pratta nie wystarcza i faktycznie potrzebujemy skonstruować automat wyszukujący wzorzec.

dokładnie jedna krawędź, pojedyncza spójna składowa jest cyklem z „podoczepianymi" do niego drzewami. Rozważmy jedną ze składowych i załóżmy, że jej cykl składa się z ℓ wierzchołków, które mają kolejno etykiety $y_1, y_2, \ldots, y_\ell$. Oznaczmy też przez T_i drzewo podczepione do i-tego wierzchołka na cyklu. Zakładamy, że korzeniem drzewa jest wierzchołek z cyklu, zatem w każdym wierzchołku cyklu jest podczepione drzewo (być może jednowierzchołkowe).

Ustalmy na chwilę jedno z drzew T_i i powiedzmy, że chcemy znaleźć wszystkie wierzchołki w tym drzewie, które mogą być początkowym ziarnem generatora bitów. Skoro krawędzie grafu są skierowane „w górę" drzew, aby móc skorzystać z opisanej przed chwilą metody wyszukiwania wzorca w drzewie, będziemy wyszukiwać w nim odwróconego słowa $x = b_n b_{n-1} \ldots b_1$. Metoda ta zadziała dla wszystkich wystąpień wzorca x, które w całości znajdują się w drzewie T_i. Aby znaleźć również te wystąpienia, które częściowo nachodzą na cykl, musimy odpowiednio zainicjować stan automatu w korzeniu drzewa.

W tym celu znajdujemy najdłuższy sufiks ciągu $\ldots y_\ell \ldots y_1 y_\ell \ldots y_1 y_\ell \ldots y_i$ (czyli nieskończonego ciągu etykiet cyklu uciętego po etykiecie y_i), który jest jednocześnie prefiksem wzorca x. Oznaczmy jego długość przez j. Wówczas początkowym stanem automatu w korzeniu drzewa T_i jest s_j.

Pozostaje kwestia szybkiego obliczania stanów początkowych. Zauważmy, że jeśli stanem początkowym dla drzewa T_i jest s_j, to dla drzewa T_{i-1} stan początkowy to $\delta(s_j, y_{i-1})$ (poza szczególnym przypadkiem $i = 1$, gdzie zamiast $i - 1$ mamy ℓ). Wystarczy zatem wyznaczyć stan początkowy dla jednego drzewa.

Opiszemy teraz, jak to zrobić efektywnie dla drzewa T_1, czyli jak znaleźć długość najdłuższego sufiksu nieskończonego ciągu $\ldots y_\ell \ldots y_1 y_\ell \ldots y_1 y_\ell \ldots y_1$, który jest jednocześnie prefiksem wzorca x. Podobnie jak powyżej oznaczmy tę wartość przez j. Wartość j można wyznaczyć, uruchamiając automat wyszukiwania wzorca x na tekście $Y = (y_\ell \ldots y_1)^k$ (cykl rozwinięty k razy) dla dostatecznie dużego k. Jest jednak pewien problem: nawet gdybyśmy znali najmniejszą wartość k, dla której to podejście pozwoliłoby wyznaczyć j, to i tak symulowanie działania automatu mogłoby być zbyt czasochłonne. Obliczenia dla pojedynczego cyklu musimy bowiem wykonać w czasie proporcjonalnym do jego długości.

Z tym problemem poradzimy sobie następująco. Na początek uruchommy automat wyszukiwania wzorca x w tekście $y_\ell \ldots y_1 y_\ell \ldots y_1$ powstałym przez dwukrotne rozwinięcie cyklu i zbadajmy końcowy stan s_i automatu. Jeśli $i \leqslant \ell$, to możemy przyjąć $j = i$.

Natomiast dla $i > \ell$ sprawa jest trudniejsza, bo znaczy to, że słowo x może przechodzić cyklem wiele razy. Wiemy, że prefiks długości ℓ słowa x pokrywa się z cyklem (być może przesuniętym cyklicznie). Niech C będzie największą liczbą, dla której prefiks długości $C\ell$ słowa x pokrywa się z cyklem (czyli x można nawinąć na cykl C razy, ale już nie $C + 1$ razy). To oznacza, że gdybyśmy uruchomili automat wyszukiwania wzorca x w tekście $Y = (y_\ell \ldots y_1)^{C+1}$, to końcowym stanem automatu byłby s_j (czyli wartość, którą chcemy wyznaczyć). Załóżmy zatem, że $C \geqslant 2$. Wiemy już, że po wczytaniu pierwszych 2ℓ liter tekstu Y, automat ten znajdzie się w stanie s_i, zatem z okresowości prefiksu wzorca x, po wczytaniu pierwszych $C\ell$ liter tekstu Y automat znajdzie się w stanie $s_{i+(C-2)\ell}$. Wystarczy zatem zainicjować automat tym stanem i dokończyć symulację, wczytując ostatnie ℓ liter tekstu Y.

Musimy jeszcze umieć efektywnie wyznaczać wartość C. W tym miejscu przyda nam się obliczenie tablicy Pref, w której Pref$[i]$ oznacza długość najdłuższego

prefiksu słowa x, który występuje w tym słowie na pozycji i. Tablicę Pref można wyznaczyć w czasie liniowym od długości słowa*. Mając tę tablicę, wartość C uzyskujemy w czasie stałym, gdyż $C = \lfloor \text{Pref}[\ell + 1]/\ell \rfloor + 1$.

Podsumowując, algorytm działa następująco: najpierw w czasie $O(n)$ budujemy automat wyszukujący wzorzec $x = b_n \dots b_1$ oraz obliczamy tablicę Pref. Następnie dzielimy graf na słabo spójne składowe. W każdej z nich wyszukujemy wzorzec w czasie proporcjonalnym do sumy rozmiarów drzew podczepionych do cyklu oraz długości cyklu, czyli po prostu proporcjonalnie do rozmiaru składowej. Zatem złożoność czasowa całego algorytmu wynosi $O(m + n)$.

*Można o tym poczytać na stronie `was.zaa.mimuw.edu.pl`.

Herbata z mlekiem

Autor zadania: Tomasz Idziaszek
Opis rozwiązania: Jakub Łącki
Dostępna pamięć: 32 MB
`https://oi.edu.pl/pl/archive/amppz/2011/her`

Bajtazar podczas ostatniej wizyty na Wyspach Bitockich bardzo zasmakował w narodowym napoju Bitocjan, którym jest herbata z mlekiem. Sposób przyrządzania i picia tego napoju jest ściśle określony i przebiega według poniższego przepisu. Najpierw napełnia się filiżankę w połowie herbatą, w połowie mlekiem i dokładnie miesza. Następnie ustala się n-literowe *słowo ceremoniału* złożone z liter `H` i `M`. Teraz kolejno dla $i = 1, 2, \ldots, n$ wykonuje się, co następuje: jeśli i-tą literą słowa ceremoniału jest `H`, to należy wypić połowę napoju z filiżanki, a następnie dolać do pełna herbaty i zamieszać. Jeśli zaś i-tą literą tego słowa jest `M`, to wykonuje się to samo, tylko zamiast herbaty dolewa się mleka. Po wykonaniu tych czynności dla każdej litery słowa ceremoniału, pozostały w filiżance napój należy wylać.

Za każdym razem, gdy Bajtazar odprawi powyżej opisaną ceremonię, zastanawia się, czy wypił więcej herbaty, czy też mleka. Pomóż Bajtazarowi w rozwikłaniu tej zagadki.

Wejście

W pierwszym wierszu wejścia znajduje się liczba całkowita n ($1 \leqslant n \leqslant 100\,000$). W drugim wierszu znajduje się n-literowe słowo złożone z liter `H` i `M` — jest to słowo ceremoniału, którego użył Bajtazar.

Wyjście

Twój program powinien wypisać literę `H`, jeśli Bajtazar wypił więcej herbaty niż mleka; literę `M`, jeśli wypił więcej mleka niż herbaty; lub słowo `HM`, jeśli wypił tyle samo herbaty co mleka.

Przykład

Dla danych wejściowych:

```
5
HMHHM
```

poprawnym wynikiem jest:

```
H
```

Wyjaśnienie przykładu: Bajtazar wypił w sumie $1\frac{37}{64}$ filiżanki herbaty i $\frac{59}{64}$ filiżanki mleka.

Rozwiązanie

Na początku odwróćmy postawione pytanie. Zamiast obliczać, ile herbaty i mleka wypija za każdym razem Bajtazar, zastanówmy się, jakiej części dolanych napojów nie wypije. Innymi słowy, skupmy się na obliczeniu końcowej zawartości filiżanki. Ponieważ wiemy, ile filiżanek każdego z napojów dolał Bajtazar w trakcie ceremonii, możemy na tej podstawie udzielić odpowiedzi. Aby uniknąć operowania na ułamkach, załóżmy, że filiżanka ma pojemność 2^{n+1} mililitrów i na początku jest w niej 2^n mililitrów mleka i tyleż samo herbaty.

Skupmy się na sytuacji, w której w pierwszej dolewce Bajtazar dodaje 2^n mililitrów herbaty. Zastanówmy się teraz, jak dużo tej herbaty dotrwa do samego końca ceremonii. W drugiej kolejce Bajtazar wypije połowę herbaty z tej dolewki, czyli zostanie jej 2^{n-1} mililitrów; po trzeciej kolejce zostanie 2^{n-2} i tak dalej. Po n kolejkach, czyli $n-1$ rozcieńczeniach, w filiżance będą więc 2 mililitry pochodzące z pierwszej dolewki.

Druga dolewka będzie rozcieńczana $n-2$ razy, zatem zdecyduje o 4 mililitrach ostatecznej zawartości filiżanki. Ogólnie dokładnie 2^i mililitrów z i-tej dolewki dotrwa w filiżance do końca ceremonii. Oprócz tego w filiżance pozostanie 1 mililitr herbaty oraz 1 mililitr mleka, które trafiły do niej na samym początku ceremonii. Dla przykładowego słowa ceremoniału `HMHHM` na końcu w filiżance będzie $1 + 2^1 + 2^3 + 2^4$ mililitrów herbaty oraz $1 + 2^2 + 2^5$ mililitrów mleka.

W ten sposób możemy końcową zawartość filiżanki wyrazić w postaci sumy potęg dwójki. Wszystkie te potęgi musimy dodać, co w niektórych językach programowania wymaga implementacji arytmetyki dużych liczb. Podobnie możemy obliczyć ilości herbaty i mleka, które Bajtazar wlał do filiżanki. Na koniec wystarczy wykonać dwa odejmowania, by dowiedzieć się, ile każdego napoju wypił Bajtazar.

Tym sposobem dojdziemy do rozwiązania. Okazuje się jednak, że wcale nie musimy wykonywać skomplikowanych obliczeń. Zauważmy, że w zadaniu wystarczy wyznaczyć, którego napoju Bajtazar wypił więcej, a w tym celu niekoniecznie trzeba obliczać dokładną ilość wypitych napojów.

Lepsze rozwiązanie

Na potrzeby opisu załóżmy, że Bajtazar nie wykonuje ostatniej dolewki i po całej ceremonii filiżanka jest w połowie pusta. To oczywiście w żaden sposób nie wpływa na wynik, bo Bajtazar i tak nie pije dolanego wtedy napoju. Usuńmy więc ostatnią literę ze słowa ceremoniału i przyjmijmy odtąd, że ma ono długość $n-1$.

Wprowadźmy kilka oznaczeń. Przez h_w i h_f oznaczmy ilość herbaty wlaną do filiżanki oraz ilość herbaty, która pozostała w filiżance po całej ceremonii. Analogicznie niech m_w i m_f oznaczają te same wielkości, jednak w odniesieniu do mleka. Bajtazar wypił więc $h_w - h_f$ mililitrów herbaty oraz $m_w - m_f$ mililitrów mleka. Ponadto na końcu filiżanka jest w połowie pełna, czyli $h_f + m_f = 2^n$.

Zajmijmy się najpierw przypadkiem, w którym słowo ceremoniału zawiera więcej literek jednego rodzaju niż drugiego. Okazuje się, że wówczas Bajtazar wypija więcej tego napoju, który częściej trafiał do filiżanki. Dla ustalenia uwagi przyjmijmy, że do filiżanki trafiło więcej herbaty, czyli h_w jest co najmniej o pół filiżanki (2^n mililitrów) większe od m_w, tj. $h_w \geqslant m_w + 2^n$. Wprawdzie od ilości dolanych

napojów musimy odjąć jeszcze zawartość filiżanki, czyli h_f i m_f, jednak, co zaraz wykażemy formalnie, wartości te są na tyle małe, że nie mogą zniwelować przewagi ilości herbaty. Filiżanka jest w połowie pusta, a co więcej, jest w niej co najmniej 1 mililitr każdego napoju, pochodzący z jej początkowej zawartości. W szczególności zatem na końcu w filiżance jest co najwyżej $2^n - 1$ mililitrów herbaty ($h_f \leqslant 2^n - 1$). Ostatecznie ilość wypitej herbaty jest ograniczona z dołu następująco:

$$h_w - h_f \geqslant h_w - 2^n + 1 \geqslant m_w + 2^n - 2^n + 1 = m_w + 1.$$

Mamy więc, że Bajtazar wypił więcej herbaty niż wynosi łączna ilość mleka ze wszystkich dolewek. Tym sposobem pokazaliśmy, że jeśli słowo ceremoniału zawiera więcej literek `H` niż `M`, to Bajtazar wypił więcej herbaty. Symetryczny argument pokazuje, że jeśli słowo zawiera więcej literek `M` niż `H`, to Bajtazar wypił więcej mleka.

Do rozważenia pozostał przypadek, gdy słowo zawiera tyle samo literek `H` i `M`, czyli do filiżanki trafia tyle samo każdego z napojów. Dzieje się tak na przykład, gdy $n = 1$. Skoro ignorujemy ostatnią dolewkę, Bajtazar wypija jedynie połowę filiżanki, w której herbata wymieszana jest pół na pół z mlekiem. Zatem każdego napoju wypija po równo.

Załóżmy dalej, że $n > 1$. Jeśli do filiżanki Bajtazar wlał tyle samo mleka co herbaty, to wypija więcej tego napoju, którego *mniej* pozostało na końcu w naczyniu. Wiemy jednak, że połowa końcowej zawartości pochodzi z przedostatniej dolewki (ostatniej, której nie zignorowaliśmy). Przyjmijmy, że Bajtazar dolał wtedy mleka. Dodatkowo w filiżance jest jeszcze 1 mililitr mleka, który został do niej wlany na samym początku. Stąd prosty wniosek, że w filiżance musi być więcej mleka. Czyli Bajtazar wypił więcej herbaty.

Cały algorytm sprowadza się więc do trzech prostych przypadków. Jeśli $n = 1$, to Bajtazar wypija tyle samo każdego z napojów. Jeśli w słowie ceremoniału jednej litery jest więcej niż drugiej (przypominamy, że nie bierzemy tu pod uwagę ostatniej litery podanego na wejściu słowa), to Bajtazar wypija więcej napoju reprezentowanego przez tę literę. W przeciwnym przypadku wiemy, że nasz bohater wypija *mniej* tego napoju, który dolany był jako przedostatni.

I

Iloraz inteligencji

Autor zadania: Marek Cygan
Opis rozwiązania: Eryk Kopczyński
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2011/ilo`

Na Uniwersytecie Bajtockim można studiować jedynie dwa kierunki: matematykę i informatykę. Aktualnie na uniwersytecie uczy się n studentów matematyki oraz m studentów informatyki. Kierunki te są bardzo trudne i nie istnieje student kształcący się w obu z nich jednocześnie.

Rektor Bajtazar zna iloraz inteligencji każdego ze studentów. Na tej podstawie chciałby wyznaczyć zespół studentów, który poradzi sobie z najtrudniejszymi problemami ludzkości. Postanowił więc, że wybierze zespół o możliwie dużej sumie ilorazów inteligencji wszystkich członków.

Iloraz inteligencji to jednak nie wszystko. Dlatego rektor chciałby, aby wszyscy członkowie zespołu się znali. Wiadomo, że wszyscy studenci matematyki się znają. Podobnie, każdy student informatyki zna każdego innego studenta informatyki.

Pomóż rektorowi i napisz program, który wyznaczy zespół studentów o możliwie dużej sumie ilorazów inteligencji, w którym wszyscy się znają.

Wejście

W pierwszym wierszu wejścia znajdują się trzy liczby całkowite n, m i k ($1 \leqslant n, m \leqslant 400$, $0 \leqslant k \leqslant n \cdot m$), które oznaczają odpowiednio liczbę studentów matematyki, liczbę studentów informatyki oraz liczbę par osób z różnych kierunków, które się znają.

Każdy z kolejnych k wierszy zawiera opis jednej pary znajomych: i-ty z tych wierszy zawiera dwie liczby całkowite a_i i b_i ($1 \leqslant a_i \leqslant n$, $1 \leqslant b_i \leqslant m$) oznaczające, odpowiednio, numer studenta matematyki i numer studenta informatyki z i-tej pary. Zarówno studentów matematyki, jak i studentów informatyki numerujemy liczbami całkowitymi, poczynając od 1.

W następnym wierszu znajduje się n liczb całkowitych należących do przedziału od 1 do 10^9, które oznaczają ilorazy inteligencji kolejnych studentów matematyki. Kolejny wiersz zawiera m liczb opisujących ilorazy inteligencji studentów informatyki, w analogicznym formacie.

Wyjście

W pierwszym wierszu wyjścia należy wypisać jedną liczbę całkowitą równą maksymalnej możliwej do uzyskania sumie ilorazów inteligencji w zespole, który spełnia oczekiwania Bajtazara.

Drugi wiersz powinien zawierać jedną liczbę całkowitą, będącą liczbą studentów matematyki, których powinien wybrać Bajtazar. W trzecim wierszu należy wypisać numery tych studentów (w dowolnej kolejności). Jeżeli w zespole nie ma żadnych studentów matematyki, należy wypisać pusty wiersz.

W kolejnych dwóch wierszach należy podać numery studentów informatyki wyznaczonych do zespołu, w analogicznym formacie.

W przypadku, gdy istnieje wiele rozwiązań, Twój program może wypisać dowolne z nich.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
6
1
2
2
1 2
```

Rozwiązanie

Sformułujmy nasz problem w języku teorii grafów. Dany jest graf dwudzielny $G = (V_1, V_2, E)$, gdzie V_1 to studenci matematyki, V_2 to studenci informatyki, a E to relacja znajomości między studentami różnych kierunków. Każdy wierzchołek v ma nieujemną wagę $w(v)$ (iloraz inteligencji studenta). *Kliką dwudzielną* nazywamy graf dwudzielny, w którym każde dwa wierzchołki pochodzące z różnych zbiorów są połączone krawędzią. Problem z zadania polega na znalezieniu w grafie G *najcięższej* kliki dwudzielnej, czyli kliki dwudzielnej o największej możliwej sumie wag wierzchołków. Formalnie, jeśli przez W oznaczymy zbiór wierzchołków tej kliki dwudzielnej, to musi zachodzić $(W \cap V_1) \times (W \cap V_2) \subseteq E$ oraz suma wag wierzchołków ze zbioru W ma być jak największa.

Uproszczona wersja

Zacznijmy od trochę prostszego zadania: przyjmijmy, że wszystkie wagi są równe 1. W tym przypadku maksymalizowanie sumy wag wierzchołków kliki dwudzielnej sprowadza się do maksymalizowania liczby jej wierzchołków. Szukamy zatem *najliczniejszej* kliki dwudzielnej.

Niech $E' = V_1 \times V_2 \setminus E$, czyli E' jest relacją nieznajomości. Pokażemy teraz, że problemy szukania każdego z poniższych są równoważne (patrz także rysunek 1):

- Zbioru wierzchołków najliczniejszej kliki dwudzielnej W.
- Najliczniejszego zbioru niezależnego W' w grafie $G' = (V_1, V_2, E')$. *Zbiór niezależny* to zbiór wierzchołków, między którymi nie ma żadnych krawędzi:

$$((W' \cap V_1) \times (W' \cap V_2)) \cap E' = \emptyset.$$

- Najmniejszego pokrycia wierzchołkowego P w grafie G'. *Pokrycie wierzchołkowe* to taki podzbiór $P \subseteq V_1 \cup V_2$, że dla każdej krawędzi $(v_1, v_2) \in E'$ zachodzi $v_1 \in P$ lub $v_2 \in P$.

- Najliczniejszego skojarzenia w grafie G'. *Skojarzenie* to taki podzbiór krawędzi grafu, że żadne dwie krawędzie nie mają wspólnego końca.

Rysunek 1. Po lewej: graf dwudzielny G z zaznaczoną kolorem najliczniejszą kliką dwudzielną. Po prawej: graf dwudzielny G' (dopełnienie grafu G) z zaznaczonym najliczniejszym skojarzeniem (pogrubione krawędzie), minimalnym pokryciem wierzchołkowym (czarne kółka) i najliczniejszym zbiorem niezależnym (kolorowe kółka).

Równoważność pierwszych dwóch problemów wynika wprost z definicji E'; w tym przypadku mamy po prostu $W = W'$. Równoważność drugiego i trzeciego wynika stąd, że zbiór wierzchołków jest niezależny wtedy i tylko wtedy, gdy jego dopełnienie jest pokryciem wierzchołkowym. Ostatnia równoważność to twierdzenie Königa.

Twierdzenie 1 (König). W grafie dwudzielnym rozmiar najliczniejszego skojarzenia jest równy liczbie wierzchołków w najmniejszym pokryciu wierzchołkowym.

Dowód twierdzenia Königa jest konstruktywny* i za jego pomocą można w czasie liniowym od rozmiaru grafu znaleźć najmniejsze pokrycie wierzchołkowe w G', mając dane najliczniejsze skojarzenie w tym grafie. Z kolei najliczniejsze skojarzenie można wyznaczyć w czasie $O(|V| \cdot |E'|)$ (gdzie V oznacza zbiór wszystkich wierzchołków w grafie) za pomocą standardowego algorytmu przez ścieżki naprzemienne lub też w czasie $O(|E'|\sqrt{|V|})$ za pomocą algorytmu Hopcrofta–Karpa. To oznacza, że szczególny wariant zadania można rozwiązać w czasie $O(|E| + |E'|\sqrt{|V|})$.

Oryginalny problem

Zajmijmy się teraz ogólną wersją zadania, dopuszczającą różne wagi wierzchołków.

Jak moglibyśmy uogólnić poprzednie rozumowanie? Zauważmy, że można to zrobić łatwo, choć bardzo nieefektywnie, w następujący sposób: każdy wierzchołek v zastępujemy jego $w(v)$ kopiami i stosujemy poprzedni algorytm. Można wykazać, że w optymalnym rozwiązaniu bierzemy albo wszystkie kopie danego wierzchołka, albo żadnej, także w ten sposób otrzymamy optymalne rozwiązanie oryginalnego problemu. Doświadczeni zawodnicy łatwo też przywrócą efektywność szukaniu skojarzenia — wiadomo, że problem znajdowania najliczniejszego skojarzenia jest szczególnym przypadkiem problemu znajdowania maksymalnego przepływu.

*Dowód ten można znaleźć np. w artykule *W grafach dwudzielnych jest łatwiej* w numerze 11/2013 miesięcznika *Delta*.

W sieci przepływowej, która wychodzi w naszym przypadku, możemy z powrotem posklejać kopie wierzchołków i zsumować przepustowości sklejanych krawędzi, ponownie uzyskując mały graf.

Okazuje się jednak, że do całego problemu można podejść nieco inaczej, nie przechodząc po drodze przez szczególny przypadek wag jednostkowych. Część uczestników zawodów AMPPZ 2011 zastosowała takie właśnie podejście, dlatego zdecydowaliśmy się opisać je na potrzeby niniejszej książki. Pod względem algorytmicznym otrzymane rozwiązanie jest równoważne powyższemu (w szczególności ukrywa w sobie konstruktywny dowód twierdzenia Königa).

Algorytm

Z wcześniejszej analizy wiemy, że nasz problem jest równoważny szukaniu pokrycia wierzchołkowego o najmniejszej sumie wag wierzchołków (nazwiemy je po prostu *najlżejszym* pokryciem wierzchołkowym). Ten ostatni problem sprowadzimy do szukania maksymalnego przepływu w sieci.

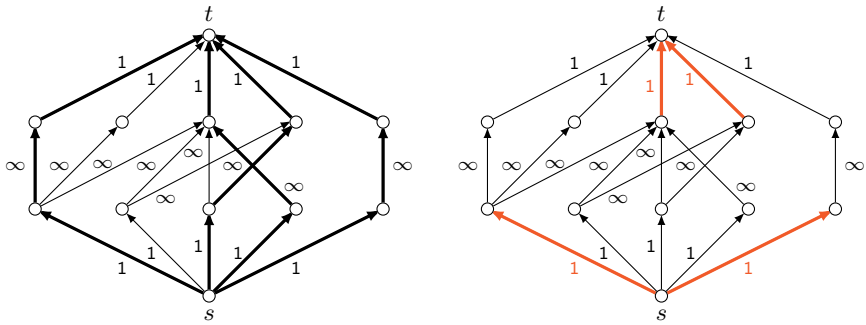
Skonstruujmy mianowicie sieć przepływową $H = (V_H, E_H)$ na podstawie grafu $G' = (V_1, V_2, E')$; patrz lewa strona rysunku 2. Każdej krawędzi w E' nadajemy przepustowość ∞. Następnie dodajemy źródło s i łączymy je z każdym wierzchołkiem $v_1 \in V_1$ krawędzią o przepustowości $w(v_1)$. Dodajemy także ujście t i łączymy każdy wierzchołek $v_2 \in V_2$ z ujściem krawędzią o przepustowości $w(v_2)$. Wykażemy, że poniższe wartości są równe:

- Waga najlżejszego pokrycia wierzchołkowego w grafie G'.
- Wartość minimalnego przekroju w sieci przepływowej H. *Przekrojem* w sieci przepływowej nazywamy zbiór krawędzi, po którego usunięciu w sieci nie istnieje ścieżka z s do t, a *wartością* przekroju nazywamy łączną przepustowość usuniętych krawędzi.
- Wartość maksymalnego przepływu w sieci H.

Równość dwóch ostatnich wynika z podstawowego twierdzenia charakteryzującego przepływy w sieci.

Twierdzenie 2 (O maksymalnym przepływie i minimalnym przekroju). W dowolnej sieci przepływowej minimalny przekrój i maksymalny przepływ mają taką samą wartość.

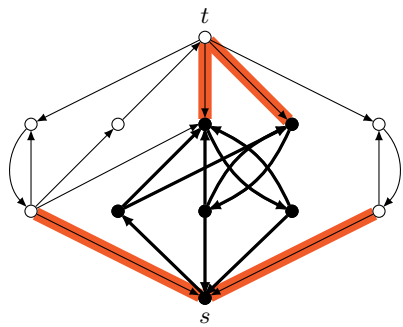
Wykażemy teraz równość pierwszych dwóch wartości. W tym celu wystarczy zauważyć, że istnieje jednoznaczna odpowiedniość między przekrojami w H o wadze mniejszej niż ∞ a pokryciami wierzchołkowymi w G'. Faktycznie, w dowolnym takim przekroju usuwamy z sieci H pewne krawędzie łączące źródło z wierzchołkami z V_1 oraz pewne krawędzie łączące wierzchołki z V_2 z ujściem, tak aby po ich usunięciu nie istniała żadna ścieżka prowadząca ze źródła do ujścia. Równoważnie dla każdej krawędzi $v_1v_2 \in E'$ w przekroju musi się znaleźć krawędź sv_1 lub krawędź v_2t. Usunięcie krawędzi sv_1 lub v_2t zinterpretujemy jako wybór wierzchołka v_1 lub, odpowiednio, v_2 do pokrycia wierzchołkowego. W ten sposób warunek przekroju oznacza dokładnie tyle, że dla każdej krawędzi z E' co najmniej jeden jej koniec musi należeć do pokrycia wierzchołkowego, co stanowi po prostu definicję pokrycia wierzchołkowego (patrz prawa strona rysunku 2). Co więcej, waga przekroju to dokładnie waga odpowiadającego mu pokrycia wierzchołkowego. To pokazuje żądaną równość.



Rysunek 2. Po lewej: sieć przepływowa H z zaznaczonym maksymalnym przepływem, skonstruowana dla grafu dwudzielnego G' z rysunku 1. Ponieważ wagi wierzchołków w tym przykładzie są jednostkowe, maksymalny przepływ odpowiada najliczniejszemu skojarzeniu z rysunku 1. Po prawej: minimalny przekrój w sieci H (o wartości 4), odpowiadający minimalnemu pokryciu wierzchołkowemu z rysunku 1.

Implementacja

Aby rozwiązać zadanie, potrzebna jest nam nie tylko waga najlżejszego pokrycia wierzchołkowego grafu G', lecz także samo to pokrycie. Pokrycie takie uzyskamy natychmiast, znając minimalny przekrój w sieci przepływowej H. Jako ostatni element układanki potrzebna nam jest więc metoda odtwarzania minimalnego przekroju na podstawie struktury maksymalnego przepływu.



Rysunek 3. Sieć rezydualna (bez wag na krawędziach) skonstruowana dla maksymalnego przepływu w sieci z rysunku 2. Zaznaczono zbiór X wierzchołków osiągalnych ze źródła (czarne kółka, pogrubione krawędzie) i zbiór krawędzi, których odpowiedniki w oryginalnej sieci przepływowej stanowią minimalny przekrój (kolorem).

Na szczęście znany jest na to prosty algorytm oparty na sieci rezydualnej. Przypomnijmy, że *sieć rezydualna* to ważony graf skierowany określony na podstawie sieci przepływowej i pewnego przepływu w tej sieci. Jeśli w sieci przepływowej mamy krawędź uv o przepustowości c, przez którą płynie przepływ f, to sieć rezy-

dualna zawiera dwie krawędzie: uv o wadze $c - f$ i vu o wadze f. Pomijamy przy tym krawędzie o wadze zerowej. Niech X będzie zbiorem tych wierzchołków, które w sieci rezydualnej skonstruowanej dla maksymalnego przepływu są połączone ze źródłem krawędziami o dodatniej wadze. Wtedy jednym z minimalnych przekrojów jest zbiór krawędzi pierwotnej sieci przepływowej prowadzących z X do $V_H \setminus X$; patrz rysunek 3.

Na koniec warto zastanowić się nad złożonością czasową naszego rozwiązania. Sieć przepływowa ma $|V_H| = |V| + 2$ wierzchołków i $|E_H| = |E'| + 2 \cdot |V|$ krawędzi. Algorytm Edmondsa-Karpa o złożoności $O(|V_H| \cdot |E_H|^2)$ raczej nie zmieści się w limicie czasu. Lepiej użyć algorytmu Dinica o złożoności $O(|V_H|^2 \cdot |E_H|)$ (jego implementację zawsze warto mieć gotową na zawodach) albo dosyć prostego w implementacji algorytmu skalującego, którego szkic podajemy poniżej.

W algorytmie skalującym zakładamy, że wszystkie skończone przepustowości są z przedziału $[0, 2^B)$. Na początku wszystkie skończone przepustowości ustawiamy na 0. Następnie wykonujemy B faz dla i od $B - 1$ do 0. Przed i-tą fazą przepustowość krawędzi e powiększamy o 2^i wtedy i tylko wtedy, gdy i-ty bit oryginalnej przepustowości tej krawędzi jest równy 1. Przy takim podejściu w i-tej fazie znajdujemy co najwyżej $O(|V_H|)$ ścieżek powiększających, każdą w czasie $O(|V_H|^2)$, zatem wszystko działa w czasie $O(B \cdot |V_H|^3)$.

J

Jaskinia

Autor zadania: Jakub Łącki
Opis rozwiązania: Jakub Łącki
Dostępna pamięć: 256 MB
`https://oi.edu.pl/pl/archive/amppz/2011/jas`

Bajtazar odkrył jaskinię. Okazało się, że jaskinia ta składa się z n komnat połączonych korytarzami w taki sposób, że między dowolnymi dwiema komnatami można przejść na dokładnie jeden sposób.

Jaskinię trzeba teraz staranniej zbadać, dlatego Bajtazar poprosił swoich kolegów o pomoc. Wszyscy przybyli na miejsce i chcą podzielić się na grupy. Każdej grupie przypadnie do zbadania tyle samo komnat, a każda komnata zostanie przydzielona dokładnie jednej grupie. Dodatkowo, żeby ekipy nie wchodziły sobie w drogę, każda z nich powinna być w stanie poruszać się pomiędzy przydzielonymi sobie komnatami bez przechodzenia przez komnaty przydzielone innym grupom.

Na ile grup mogą podzielić się badacze jaskini?

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($2 \leqslant n \leqslant 3\,000\,000$), oznaczającą liczbę komnat w jaskini. Komnaty są ponumerowane od 1 do n.

Kolejne $n-1$ wierszy opisuje połączenia między komnatami. W i-tym spośród nich znajduje się liczba a_i ($1 \leqslant a_i \leqslant i$), która reprezentuje korytarz łączący komnaty o numerach $i+1$ oraz a_i.

Wyjście

Wypisz jeden wiersz zawierający wszystkie takie liczby k, że komnaty w jaskini można podzielić na k rozłącznych zbiorów równej wielkości, a pomiędzy dowolnymi dwiema komnatami w każdym zbiorze można przejść, korzystając jedynie z komnat z tego zbioru. Liczby należy wypisać w kolejności rosnącej, pooddzielane pojedynczymi odstępami.

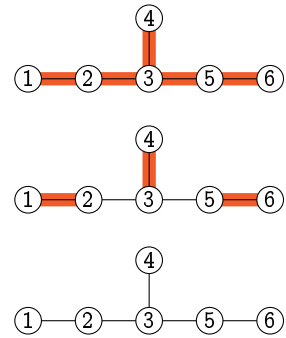
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
1 3 6
```



Rozwiązanie

Przejdźmy na język teorii grafów. Opisana w treści zadania jaskinia to spójny graf bez cykli, czyli drzewo. Zacznijmy od zupełnie oczywistego spostrzeżenia: aby drzewo dało się podzielić na kawałki rozmiaru k (czyli zawierające po k wierzchołków), to liczba k musi być dzielnikiem liczby n. Co więcej, skoro dzielimy drzewo na kawałki rozmiaru k, to otrzymamy dokładnie $\frac{n}{k}$ kawałków. Zauważmy wreszcie, że aby z drzewa o n wierzchołkach stworzyć $\frac{n}{k}$ drzew o rozmiarze k, należy usunąć dokładnie $\frac{n}{k} - 1$ krawędzi. Te proste spostrzeżenia od razu prowadzą nas do pierwszego rozwiązania. Jak to się zwykle zdarza, pierwszy algorytm nie będzie dość dobry, jednak z tym poradzimy sobie później.

Załóżmy, że k jest dzielnikiem n i chcemy sprawdzić, czy drzewo możemy podzielić na kawałki rozmiaru k. Choć na pierwszy rzut oka może to nie być oczywiste, taki podział jest wyznaczony jednoznacznie (o ile tylko istnieje). Szkic uzasadnienia tego faktu mógłby wyglądać tak: rozważmy dowolny podział drzewa na kawałki rozmiaru k, gdzie $k < n$. W takim podziale istnieje pewien kawałek P, który z resztą drzewa połączony jest dokładnie jedną krawędzią e. W każdym podziale drzewa dla tej wartości parametru k krawędź e musi łączyć dwa kawałki podziału, gdyż w przeciwnym razie w podziale zamiast P otrzymalibyśmy kawałek o rozmiarze mniejszym lub większym od k. Teraz kawałek P możemy odciąć od drzewa i podobnym rozumowaniem wykazać, że całą resztę drzewa również podzielić można na dokładnie jeden sposób.

W naszym pierwszym rozwiązaniu stosujemy podejście rekurencyjne. Przeszukujemy wejściowe drzewo w głąb i wracając z wywołań rekurencyjnych, wyznaczamy rozmiary poddrzew zaczepionych w wierzchołkach. Gdy znajdziemy wierzchołek, w którym zaczepione jest poddrzewo o rozmiarze k, to usuwamy krawędź łączącą to poddrzewo z resztą drzewa i usuniętych wierzchołków nie bierzemy już pod uwagę przy obliczaniu rozmiarów kolejnych poddrzew. Powinno być jasne, że jeżeli całe drzewo daje się podzielić na kawałki rozmiaru k, to dokładnie $\frac{n}{k} - 1$ razy stwierdzimy, że należy usunąć krawędź prowadzącą do rozważanego poddrzewa, a na końcu zostanie nam poddrzewo rozmiaru k zawierające korzeń przeszukiwania. Prawdziwa jest również przeciwna implikacja. Jeśli $\frac{n}{k} - 1$ razy odetniemy poddrzewa o rozmiarze k, to znajdziemy podział całego drzewa na $\frac{n}{k}$ kawałków rozmiaru k.

Tę procedurę rekurencyjną możemy zaimplementować tak, by dla ustalonego k działała w czasie $O(n)$. Cały algorytm ma więc złożoność $O(n \cdot d(n))$, gdzie $d(n)$ to liczba dzielników liczby n. Zauważmy, że na początku algorytmu możemy przejrzeć wszystkie liczby od 1 do n i sprawdzić, które z nich dzielą n, bo zajmuje to jedynie czas $O(n)$.

Pozostaje pytanie, jak duża może być wartość $d(n)$. Wśród liczb od 1 do $3 \cdot 10^6$ najwięcej dzielników, bo aż 336, ma liczba 2 882 880. Zatem w pesymistycznym przypadku (który oczywiście wystąpił w testach do zadania) to rozwiązanie będzie za wolne.

Szybsze rozwiązanie

Aby przyspieszyć nasze rozwiązanie, wszystkie możliwe wartości k rozważymy podczas jednego przeszukania drzewa w głąb. Ustalmy krawędź e i odpowiedzmy na następujące pytanie: jak stwierdzić, czy krawędź e jest usuwana podczas podziału

drzewa na kawałki rozmiaru k? Z całą pewnością e musi dzielić drzewo na kawałki o rozmiarach podzielnych przez k. W ogólności zachodzi następujący fakt.

Fakt 1. Drzewo możemy podzielić na kawałki rozmiaru k wtedy i tylko wtedy, gdy istnieje w nim dokładnie $\frac{n}{k} - 1$ krawędzi, które łączą poddrzewa o rozmiarach będących wielokrotnościami k.

Dlaczego? Jeśli żądany podział istnieje, to usuwamy dokładnie $\frac{n}{k} - 1$ krawędzi, a po każdej stronie każdej usuwanej krawędzi znajduje się pewna liczba kawałków o rozmiarach k. Zatem liczba wierzchołków po obydwóch stronach każdej usuwanej krawędzi jest podzielna przez k. W drugą stronę, załóżmy, że istnieje $\frac{n}{k} - 1$ krawędzi spełniających podany warunek. Łatwo pokazać, że gdy usuwamy z drzewa rzeczone krawędzie, każde poddrzewo uzyskane po drodze ma rozmiar podzielny przez k. Ostatecznie dostaniemy więc $\frac{n}{k}$ niepustych kawałków, których rozmiary są podzielne przez k. Ich łączny rozmiar to n, skąd wnioskujemy, że każdy kawałek musi składać się z k wierzchołków.

Fakt 1 pozwala nam znacznie prościej myśleć o rozwiązaniu. Wystarczy jedynie policzyć, ile krawędzi spełnia odpowiednie własności. Konkretniej, przejrzymy wszystkie krawędzie drzewa i dla każdego k obliczymy wartość $kr[k]$, będącą liczbą krawędzi, które łączą poddrzewa o rozmiarach będących wielokrotnościami k.

Rozważmy krawędź, która łączy poddrzewa o rozmiarach a i $n - a$. Dla jakich k powinniśmy zwiększyć wartość $kr[k]$? Jak nietrudno zauważyć, powinniśmy zrobić to dla wszystkich liczb k, które dzielą zarówno a jak i $n - a$. Innymi słowy, zwiększamy $kr[k]$ dla wszystkich liczb k, które są dzielnikami $\mathrm{NWD}(a, n - a)$. Ten algorytm wygodnie nam będzie zrealizować „leniwie". Zamiast od razu aktualizować odpowiednie komórki w tablicy kr, zapiszemy na boku, że chcemy zwiększyć wartości w komórkach, których indeksy są dzielnikami $\mathrm{NWD}(a, n-a)$, a ostateczne wartości w tablicy kr wyznaczymy później.

W tym celu użyjemy jeszcze jednej tablicy, którą nazwiemy t. Aby zanotować, że chcemy zwiększyć o 1 wartość $kr[k]$ dla wszystkich liczb k, które są dzielnikami i, dodamy 1 do $t[i]$. Przy rozpatrywaniu krawędzi łączącej poddrzewa o rozmiarach a i $n - a$ będziemy więc zwiększać o jeden wartość $t[\mathrm{NWD}(a, n - a)]$. Po przejrzeniu wszystkich krawędzi, możemy wyznaczyć wartości $kr[k]$ przez zrealizowanie zmian zapisanych w tablicy t. Takie leniwe podejście pozwoli nam lepiej oszacować czas działania, gdyż dla każdego indeksu i tablicy t, tylko raz przeglądać będziemy wszystkie dzielniki i. Możemy też wyznaczyć końcową zawartość tablicy kr nieco prościej. Wystarczy zauważyć, że wartość $kr[k]$ możemy uzyskać jako sumę $t[i]$ po wszystkich i, które są wielokrotnościami k.

W jakim czasie potrafimy wyznaczyć tablicę kr na podstawie tablicy t? Obliczając $kr[k]$, rozpatrujemy wszystkie wielokrotności k, co wymaga $\lfloor \frac{n}{k} \rfloor$ kroków. Pamiętajmy, że potrzebujemy wartości w tablicy kr jedynie dla dzielników n. Czas działania można więc wyrazić przez $\sum_{k|n} \frac{n}{k}$. Zauważmy, że sumujemy tu wszystkie dzielniki n, bo jeśli k przebiega dzielniki od *najmniejszego*, to $\frac{n}{k}$ również przebiega dzielniki n, tyle że od *największego*. Zatem ten krok wymaga czasu $O(D(n))$, gdzie $D(n)$ to suma dzielników n. Na szczęście $D(n)$ jest funkcją, która rośnie wolno, zachodzi bowiem $D(n) = O(n \log \log n)$.

Najbardziej kosztowną operacją naszego algorytmu jest więc obliczanie największego wspólnego dzielnika liczb z zakresu od 1 do n. Korzystając z algorytmu Euklidesa, jedno takie obliczenie możemy wykonać w czasie $O(\log n)$. Ponieważ wykonujemy je n razy, więc łączny czas działania algorytmu to $O(n \log n)$.

Jeszcze szybsze rozwiązanie

To już wystarcza do rozwiązania zadania. Można jednak wskazać odrobinę efektywniejszą metodę. Okazuje się, że wszystkie wartości postaci $\mathrm{NWD}(i, n-i)$ dla $i = 1, \ldots, n$ możemy obliczyć zawczasu w złożoności $O(n \log\log n)$. Zauważmy, że jeśli liczba d dzieli zarówno i jak i $n-i$, to dzieli również n. Aby obliczyć $\mathrm{NWD}(i, n-i)$, wystarczy zatem znaleźć największy dzielnik n, który dzieli i oraz $n-i$.

Będziemy wypełniać tablicę nwd, tak aby $nwd[i]$ na końcu obliczeń było równe $\mathrm{NWD}(i, n-i)$. Przeglądamy wszystkie dzielniki n w kolejności rosnącej i dla każdego dzielnika d rozpatrujemy wszystkie jego wielokrotności $j \cdot d$. Dla każdej takiej wielokrotności wartość $nwd[j \cdot d]$ poprawiamy na d. Algorytm przegląda więc wszystkie wielokrotności wszystkich dzielników n, podobnie jak algorytm wyznaczający tablicę kr na podstawie tablicy t. Korzystając z poprzedniej analizy, widzimy, że tablicę nwd możemy wyznaczyć w czasie $O(n \log\log n)$. W ten sposób udało nam się zmniejszyć czas działania całego rozwiązania do $O(n \log\log n)$. Warto podkreślić, że w praktyce zysk czasowy z tego przyspieszenia będzie stosunkowo niewielki.

Ile dzielników może mieć liczba?

Na koniec wróćmy jeszcze do funkcji $d(n)$, która zlicza dzielniki n. Podaliśmy ograniczenie na wartości tej funkcji dla $n \leqslant 3 \cdot 10^6$, jednak nie wspomnieliśmy nic o ograniczeniu asymptotycznym. Łatwo udowodnić, że $d(n) < 2\sqrt{n}$, bo każda liczba może mieć co najwyżej $\sqrt{n}$ dzielników nie większych od $\sqrt{n}$, a każdy dzielnik większy od $\sqrt{n}$ jest sparowany z dzielnikiem mniejszym niż $\sqrt{n}$. Zachodzi jednak mocniejsze ograniczenie, a mianowicie $d(n) = n^{O(1/\log\log n)}$.

Na pierwszy rzut oka nie bardzo wiadomo, ile to właściwie jest. Zauważmy, że $\log\log n$ to funkcja bardzo wolno rosnąca. Jeśli przyjmiemy, że log to logarytm dwójkowy, to $\log\log(2^{32}) = 5$, a $\log\log(2^{64}) = 6$. Jednak na potrzeby obliczania czasu działania rozwiązań zadań algorytmicznych, w których mamy zazwyczaj do czynienia z liczbami nie większymi niż 2^{64}, wygodnie przyjąć, że w pesymistycznym przypadku wartość $d(n)$ to około $\sqrt[3]{n}$. Choć z matematycznego punktu widzenia jest to herezja, takie przybliżenie dobrze sprawdza się w praktyce.

Krzyżak

Autor zadania: Szymon Acedański
Opis rozwiązania: Tomasz Idziaszek
Dostępna pamięć: 32 MB
`https://oi.edu.pl/pl/archive/amppz/2011/krz`

Bajtocka odmiana pająka krzyżaka (gatunek *Araneida baitoida*) ma niesamowitą, nawet jak na bajtockie pająki, umiejętność. Potrafi mianowicie w ciągu ułamka sekundy rozpiąć dowolnie dużą pajęczynę, pod warunkiem, że wszystkie włókna składające się na taką sieć będą leżeć w jednej płaszczyźnie. To pozwala krzyżakowi stosować dość oryginalną technikę łowiecką. Nie musi czekać, aż mucha wpadnie w uprzednio zastawioną sieć — wystarczy, że pająk zna aktualną pozycję ofiary, i może natychmiast zbudować pajęczynę tak, by mucha przykleiła się do jednego z jej włókien.

Jeden z takich pająków wypatrzył właśnie w ogrodzie Bajtazara n much. Każda z nich lata nieruchomo w ustalonym punkcie przestrzeni. Pająk zastanawia się, czy będzie w stanie złapać wszystkie muchy w jedną sieć. Napisz program, który rozwieje wątpliwości pająka.

Wejście

W pierwszym wierszu wejścia znajduje się liczba całkowita n ($1 \leqslant n \leqslant 100\,000$). W kolejnych n wierszach znajduje się opis pozycji much w przestrzeni: i-ty z tych wierszy zawiera trzy liczby całkowite x_i, y_i, z_i ($-1\,000\,000 \leqslant x_i, y_i, z_i \leqslant 1\,000\,000$), będące współrzędnymi punktu (w trójwymiarowej przestrzeni euklidesowej), w którym znajduje się i-ta mucha. W dowolnym punkcie może znajdować się co najwyżej jedna mucha.

Wyjście

Twój program powinien wypisać słowo `TAK`, jeśli pająk, rozpinając jedną pajęczynę, jest w stanie złapać wszystkie muchy. W przeciwnym razie Twój program powinien wypisać słowo `NIE`.

Przykład

Dla danych wejściowych:

```
4
0 0 0
-1 0 -100
100 0 231
5 0 15
```

poprawnym wynikiem jest:

```
TAK
```

natomiast dla danych:

```
4
0 1 0
-1 0 -100
100 0 231
5 0 15
```

poprawnym wynikiem jest:

```
NIE
```

Rozwiązanie

Celem zadania *Krzyżak* było sprawdzenie, czy zawodnicy znają podstawowe narzędzia geometrii obliczeniowej. Dla danych n punktów w przestrzeni trójwymiarowej należało stwierdzić, czy wszystkie z nich leżą w jednej płaszczyźnie. Dla ustalenia notacji i-ty punkt będziemy oznaczać przez $p_i = (x_i, y_i, z_i)$.

Dowolne trzy niewspółliniowe punkty (tzn. takie, które nie leżą na jednej prostej) jednoznacznie wyznaczają płaszczyznę w przestrzeni. Rozwiązanie zadania podzielimy zatem na dwie części: najpierw znajdziemy trzy niewspółliniowe punkty (o ile istnieją), a następnie sprawdzimy, czy pozostałe punkty leżą w płaszczyźnie przez nie wyznaczonej. Zauważmy też, że dla $n \leqslant 3$ odpowiedź jest oczywiście twierdząca, więc w dalszej części będziemy zakładać, że mamy co najmniej cztery punkty.

Nie możemy po prostu wziąć trzech dowolnych punktów (np. p_1, p_2 i p_3), gdyż może się okazać, że leżą one na jednej prostej. Ale jeśli nie jest tak, że wszystkie n punktów leży na jednej prostej, to możemy wziąć dwa z nich (np. p_1 i p_2) i poszukać wśród pozostałych punktów jakiegoś niewspółliniowego z nimi. Aby sprawdzić, czy punkt p_i (dla $i = 3, \ldots, n$) leży na prostej wyznaczonej przez punkty p_1 i p_2, obliczamy iloczyn wektorowy $t_i := (p_2 - p_1) \times (p_i - p_1)$. Przypomnijmy, że iloczyn wektorowy dla wektorów $v = (x_v, y_v, z_v)$ i $w = (x_w, y_w, z_w)$ jest wektorem zdefiniowanym następująco:

$$v \times w := (y_v \cdot z_w - y_w \cdot z_v,\ z_v \cdot x_w - z_w \cdot x_v,\ x_v \cdot y_w - x_w \cdot y_v).$$

Punkt p_i leży na prostej zawierającej punkty p_1 i p_2 wtedy i tylko wtedy, gdy wszystkie współrzędne wektora t_i są równe 0.

Jeśli wszystkie wektory t_i odpowiadające punktom $p_3, \ldots, p_n$ są zerowe, to wszystkie punkty leżą na jednej prostej (a zatem w jednej płaszczyźnie) i odpowiedź jest pozytywna. Jeśli natomiast dla któregoś i dostaniemy niezerowy wektor t_i, to trójka punktów p_1, p_2 i p_i jednoznacznie wyznacza nam płaszczyznę. W tej sytuacji iloczyn wektorowy t_i to wektor prostopadły do płaszczyzny zawierającej punkty p_1, p_2 i p_i. Aby sprawdzić, czy jakiś inny punkt p_j leży w tej płaszczyźnie, wystarczy sprawdzić, czy wektor $p_j - p_1$ jest prostopadły do t_i. To sprawdzenie wykonujemy, obliczając iloczyn skalarny $t_i \cdot (p_j - p_1)$. Przypomnijmy, że iloczyn skalarny dwóch wektorów $v = (x_v, y_v, z_v)$ i $w = (x_w, y_w, z_w)$ to liczba zdefiniowana jako

$$v \cdot w := x_v \cdot x_w + y_v \cdot y_w + z_v \cdot z_w.$$

Dwa niezerowe wektory są prostopadłe wtedy i tylko wtedy, gdy ich iloczyn skalarny jest równy 0. Tak więc punkt p_j leży na płaszczyźnie wyznaczonej przez p_1, p_2 i p_i, jeśli tylko iloczyn skalarny t_i oraz $p_j - p_1$ jest równy 0.

Powyższe rozwiązanie można zaimplementować tak, by przejrzeć wszystkie punkty tylko raz. Złożoność czasowa całego rozwiązania to $O(n)$.

Uwaga na ograniczenia

Jedną z decyzji, które musimy podjąć przy implementacji powyższych obliczeń, jest wybór odpowiedniego typu danych. Aby uniknąć subtelnych kłopotów związanych z operacjami na liczbach zmiennoprzecinkowych, warto korzystać z typów całkowitoliczbowych (na szczęście jest to możliwe, gdyż dane wejściowe są całkowite i wykonujemy na nich jedynie operacje dodawania, odejmowania i mnożenia).

Jeśli rozpiszemy iloczyn skalarny z naszego rozwiązania, to okaże się, że jest on sumą sześciu składników. Każdy składnik jest iloczynem trzech wartości będących różnicami współrzędnych punktów. Wartość bezwzględna każdej ze współrzędnych jest ograniczona przez 10^6, zatem wartość bezwzględna każdego ze składników nie przekracza $(2 \cdot 10^6)^3 = 8 \cdot 10^{18}$. To akurat tyle, żeby każdy ze składników zmieścił się w 64-bitowej zmiennej ze znakiem.

Niestety oszacowanie $48 \cdot 10^{18}$ na sumę tych składników przekracza zakres 64-bitowej liczby całkowitej. Można sobie z tym kłopotem poradzić na kilka sposobów. Najprościej jest użyć zmiennych 128-bitowych, o ile nasz kompilator je wspiera (przykładowo, używany na zawodach kompilator GCC udostępnia rozszerzenie `__int128`). Nietrudno też samemu napisać obsługę takich zmiennych, zwłaszcza że wystarczy zaimplementować jedynie operacje dodawania i porównywania z zerem. Trzeci sposób to obliczenie tej sumy dwukrotnie: raz modulo m_1 i raz modulo m_2, gdzie m_1 i m_2 są względnie pierwszymi liczbami, których iloczyn przekracza $48 \cdot 10^{18}$. Z chińskiego twierdzenia o resztach wynika, że oba wyniki będą zerami wtedy i tylko wtedy, gdy rozważana suma jest równa 0.

2012

XVII Akademickie Mistrzostwa Polski
w Programowaniu Zespołowym
Warszawa, 26–28 października 2012

A

Automat

Autor zadania: Jakub Pachocki
Opis rozwiązania: Eryk Kopczyński
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2012/aut`

W budynku uczelni, na której studiuje Bajtazar, stoi automat sprzedający batony. W automacie dostępnych jest n rodzajów batonów, ponumerowanych od 1 do n. Batony poszczególnych rodzajów różnią się rozmiarem oraz smakiem, więc ich ceny mogą być różne.

Niedawno Bajtazar odkrył, że automat jest popsuty. Kiedy kupi się jednego batona rodzaju i, z automatu wypada także po jednym batonie każdego z rodzajów $1, 2, \ldots, i-1$. Oczywiście tylko wtedy, gdy batony tych rodzajów są dostępne — jeśli batonów któregoś rodzaju między 1 a $i-1$ nie ma w automacie, baton tego rodzaju nie wypada. Trzeba dodać, że ta sprytna sztuczka jest możliwa do wykonania tylko wtedy, gdy w automacie faktycznie znajduje się co najmniej jeden baton i-tego rodzaju.

Bajtazar postanowił zrobić użytek z wykrytej usterki. Dysponując pewną kwotą pieniędzy, chciałby stwierdzić, jaką największą wartość (tj. sumę cen) łakoci może wydobyć z automatu za tę kwotę. Bajtazar nie musi zużyć całej dostępnej kwoty.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite n oraz k ($1 \leqslant n \leqslant 40$, $1 \leqslant k \leqslant 64\,000$), oznaczające liczbę rodzajów batonów w automacie oraz kwotę, jaką dysponuje Bajtazar. Drugi wiersz zawiera n liczb całkowitych $c_1, \ldots, c_n$ ($1 \leqslant c_i \leqslant 40$), oznaczających ceny batonów poszczególnych rodzajów. Trzeci wiersz zawiera n liczb całkowitych $l_1, \ldots, l_n$ ($0 \leqslant l_i \leqslant 40$), oznaczających liczby batonów poszczególnych rodzajów dostępnych w automacie.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą — sumę cen łakoci, jakie Bajtazar może nabyć w automacie, dysponując kwotą k.

Przykład

Dla danych wejściowych:	poprawnym wynikiem jest:
6 8 7 2 3 5 7 2 1 3 0 3 2 1	30

Wyjaśnienie przykładu: Kupujemy batona rodzaju 6, a z automatu wypada nam także po jednym batonie rodzajów 1, 2, 4 i 5. Kupujemy batona rodzaju 4, z automatu oprócz niego wypada baton rodzaju 2.

Rozwiązanie

Zastanówmy się najpierw nad tym, w jakiej kolejności opłaca nam się kupować batony. Powiedzmy, że chcemy kupić batony rodzajów i oraz j, gdzie $i < j$. Łatwo zauważyć, że uzyskana wartość łakoci, które wypadną z automatu, nie zależy od kolejności, w jakiej kupimy te dwa batony, ale jeśli w automacie jest tylko jeden baton rodzaju i, to nie możemy zacząć zakupów od batona rodzaju j. Zatem możemy założyć, że wybrane batony kupujemy zawsze w kolejności niemalejących numerów, czyli „od lewej do prawej".

Zadanie moglibyśmy rozwiązać poprzez rekurencyjne sprawdzenie wszystkich możliwych strategii kupowania batonów, w których zaczynamy od mniejszych numerów. Oczywiście takie rozwiązanie nie będzie działało efektywnie.

Żeby to przezwyciężyć, będziemy spamiętywać nasze wyniki. W tym zadaniu jest to dosyć trikowe, bo wymaga następującego pomysłu: choć batony kupujemy od lewej do prawej, to strategię zakupów (czyli wybór, które batony kupić) będziemy ustalać od prawej do lewej. Planując zakup batona rodzaju j, wiemy, że na pewno dostaniemy również batony wszystkich rodzajów $i < j$ (jeśli są), choć być może nie dostaniemy ich razem z batonem rodzaju j, tylko z jakimś innym batonem, który kupimy wcześniej. Sztuczkę wygodnie jest przedstawić na rysunku. Dla każdego rodzaju batona i stwórzmy wieżę z liczb c_i o wysokości l_i. Dla danych z przykładu otrzymujemy następujący rysunek:

3		2		5		
2		[2]		[5]	7	
1	(7)	(2)		(5)	[7]	[2]
	1	2	3	4	5	6

Wybierzmy po jednym batonie rodzajów 4 i 6, tak jak w wyjaśnieniu z treści zadania. Jeśli zrobimy to w kolejności od lewej do prawej, kupimy najpierw batona rodzaju 4, co spowoduje wypadnięcie batonów o łącznej wartości $7+2+5$, a potem batona rodzaju 6, uzyskując dodatkowo $2+5+7+2$. Zauważmy, że w momencie kupowania drugiego batona batony rodzaju 1 już się skończyły, także nie dostaniemy takiego batona. Możemy jednak równoważnie najpierw zaplanować zakup batona rodzaju 6 (wtedy wiemy, że na pewno dostaniemy batony o wartościach $7+2+5+7+2$), a potem zaplanować zakup batona rodzaju 4, pamiętając, że skoro później i tak kupimy baton o większym numerze, to batona rodzaju 1 (który jest tylko jeden) nie będziemy drugi raz liczyć do naszego zysku. Zilustrowaliśmy to na rysunku poniżej. Zauważmy, że zakup batona rodzaju i możemy zaplanować nawet wtedy, gdy już wcześniej zaplanowaliśmy zakup więcej niż l_i batonów o numerach wyższych. Nie możemy jedynie bezpośrednio kupić więcej niż l_i batonów rodzaju i.

3		2		5		
2		[2]		[5]	7	
1	(7)	(2)		(5)	(7)	(2)
	1	2	3	4	5	6

W ten sposób w każdym wierszu wybierzemy tylko jedno pole. Aby zrobić to wygodniej, nasza strategia liczy sumy prefiksów w kolejnych wierszach w tablicy. Oznaczmy sumę prefiksową dla pola z i-tej kolumny leżącego w wierszu y

przez $t[i, y]$. Dla danych z przykładu mamy zatem $t[6, 1] = 7 + 2 + 5 + 7 + 2 = 23$ oraz $t[4, 2] = 2 + 5 = 7$. Tabelka po zastąpieniu każdej wartości sumą prefiksu wygląda następująco:

3	0	2	2	7	7	7
2	0	2	2	7	14	14
1	7	9	9	14	21	23
	1	2	3	4	5	6

Tak więc w naszej strategii z każdego wiersza i, idąc od dołu, wybieramy pewne pole (i, y), pamiętając przy tym, że:

- pole wybrane z kolejnego wiersza nie może znajdować się na prawo od pola wybranego w wierszu poniżej,
- pole w kolumnie i-tej możemy wybrać co najwyżej l_i razy,
- suma liczb c_i dla wybranych pól nie może przekraczać naszego budżetu k,
- suma wartości $t[i, y]$ dla wybranych pól ma być jak największa.

Pola będziemy wybierać od prawej do lewej. Przypuśćmy, że rozważyliśmy właśnie i-tą kolumnę i wybraliśmy w niej pewną liczbę pól, między 0 a $l[i]$. Z powyższego rozumowania widać, że jedyne potrzebne nam na przyszłość informacje o polach wybranych dla kolumn o numerach $j \geqslant i$ to łączna liczba wybranych pól oraz pozostały budżet (oznaczmy te wartości przez y i b). Zatem możemy zastosować rekurencję ze spamiętywaniem albo programowanie dynamiczne: dla każdej trójki liczb (i, y, b) optymalną strategię wyboru pól wystarczy zaplanować tylko raz. Tworzymy tablicę trójwymiarową $w[i, y, b]$, którą wypełniamy zgodnie z poniższą rekurencją (wartość s oznacza liczbę pól, które wybieramy w i-tej kolumnie):

$$w[i, y, b] = \max_{0 \leqslant s \leqslant l[i]} \Big\{ w[i+1,\ y-s,\ b + s \cdot c[i]] \ + \ \sum_{h=1}^{s} t[i, y-h+1] \Big\}. \qquad (1)$$

Oszacujmy złożoność naszego algorytmu. Niech C będzie największą ceną batona, a L największą liczbą batonów tego samego rodzaju. Współrzędna i tabelki przebiega u nas wartości od 0 do n, współrzędna y przebiega wartości od 0 do L (widać z rysunku, że nie ma sensu łącznie kupować więcej niż L batonów), a współrzędna b przebiega wartości od 0 do k. Zatem nasza tablica ma $O(nLk)$ pól i każde z nich obliczamy w czasie $O(L)$, musimy bowiem sprawdzić wszystkie możliwe liczby zakupionych batonów rodzaju i. Z tej analizy wynika, że złożoność czasowa naszego algorytmu to $O(nL^2k)$.

Przy ograniczeniach z treści zadania daje to orientacyjnie $40^3 \cdot 64\,000 \approx 4 \cdot 10^9$ operacji, czyli trochę za dużo. Ale wystarczy zauważyć, że tak duże ograniczenie na k nie ma sensu — jeśli wykupimy L batonów o największych numerach, to dostaniemy wszystkie batony w automacie, a zapłacimy za to maksymalnie LC, przeznaczając resztę na cele charytatywne. Zatem możemy przyjąć, że Bajtazar ma $\min(k, LC)$ pieniędzy, ograniczając złożoność naszego algorytmu do $O(nL^3C)$. Dla maksymalnych danych wejściowych wykonamy już tylko rzędu $40^5 \approx 10^8$ operacji. Złożoność pamięciowa to $O(nL^2C)$, choć można ją łatwo zredukować do $O(L^2C)$ poprzez pamiętanie tylko dwóch warstw naszej tabelki, dla dwóch kolejnych wartości i.

Szybsze rozwiązanie

Powyższe rozwiązanie zostałoby zaakceptowane na zawodach (nawet bez redukowania złożoności pamięciowej), ale istnieje rozwiązanie szybsze. Poniżej przedstawiamy szkic pomysłu, szczegóły zostawiając Czytelnikowi.

Korzystając ze wzoru (1), wiele razy liczymy różne maksima. Czy można te maksima liczyć bardziej efektywnie? Okazuje się, że tak. Dla danej wartości i będziemy przechodzili po wszystkich możliwych wartościach b i liczyli wartości $w[i, y, b - y \cdot c[i]]$ dla każdego y po kolei.

Zauważmy, że jeśli zapomnimy na chwilę o składniku $\sum_{h=1}^{s} t[i, y-h+1]$, to dwie kolejne liczone wartości, $w[i, y, b-y \cdot c[i]]$ i $w[i, y+1, b-(y+1)c[i]]$, są maksimami prawie tego samego podzbioru tablicy w — pierwszy podzbiór jest odcinkiem długości $l[i]$ w jednowymiarowym, „skośnym" fragmencie tablicy, a drugi jest takim samym odcinkiem, w którym dodaliśmy jeden element na końcu i zabraliśmy jeden z drugiej strony. Także nasz problem redukuje się do następującego: dana jest tablica $M[1..N]$ i długość przedziału m; chcemy szybko policzyć maksimum każdego przedziału o długości m, czyli dla każdego y od 1 do $N - m$ chcemy policzyć maksimum $M[y..y + m]$.

Uwzględnienie zapomnianego składnika też nie jest trudne — możemy patrzeć na to tak, że przy przesunięciu przedziału (i liczeniu kolejnego maksimum) każda wartość w tablicy M rośnie o tę samą wartość $t[i, y]$ dla pewnego y. Wystarczy zamiast wartości $M[y] = w[i+1, y, b-y \cdot c[i]+s \cdot c[i]]+\sum_{h=1}^{s} t[i, y-h+1]$ pamiętać w każdym momencie przesunięcie δ, w naszej tablicy trzymać tablicę wartości $M'[y] = M[y] - \delta$ i pamiętać, że rzeczywiste wartości $M[y]$, których maksimum nas interesuje, są równe $M'[y] + \delta$.

Zatem, przy każdym ruchu w prawo będziemy dokładać do naszej tablicy nowe $M'[y]$ zgodnie ze wzorem $M'[y] = w[i + 1, y, b - y \cdot c[i]]$, uaktualniać wartość δ o $t[i, y]$ i nasze $w[i, y, b - y \cdot c[i]]$ będzie równe maksimum odpowiedniego odcinka tablicy M' powiększonemu o δ.

Liczenie kolejnych maksimów

Jak teraz efektywnie liczyć maksima kolejnych odcinków długości m w tablicy $M[1..N]$? Można to zrobić w czasie $O(N)$ w sposób następujący. Przesuwamy okno długości m od lewej do prawej, w każdym momencie pamiętając zbiór S tych pozycji, które są lub potencjalnie w przyszłości mogą zostać maksimami — a zatem takich, że w naszym oknie na prawo od nich nie ma liczby większej lub równej. Łatwo zauważyć, że $S = \{x_1, \ldots, x_k\}$, gdzie ciąg x_i jest rosnący, a ciąg $M[x_i]$ musi być malejący. Maksimum okna jest równe wartości $M[x_1]$.

Po przesunięciu okna o jedno pole wykonujemy dwie operacje: jeśli maksimum wypadło z lewej strony okna, to wyrzucamy x_1 ze zbioru S; następnie uwzględniamy prawy koniec okna, czyli dodajemy go do zbioru S, przy okazji wyrzucając z niego te spośród największych elementów, dla których $M[x_i]$ jest nie większe od nowej wartości. Każdy element jest raz dodawany i raz usuwany z naszej struktury, także łącznie wykonujemy $O(N)$ operacji. Zbiór S możemy trzymać na kolejce dwustronnej, którą łatwo zaimplementować w tablicy tak, aby każda operacja była wykonywana w czasie stałym.

Tak więc całe zadanie można rozwiązać w czasie $O(nLk)$. Jak już wiemy, zamiast k wystarczy wziąć $\min(k, LC)$, co redukuje złożoność do $O(nL^2C)$.

B

Biuro podróży

Autor zadania: Jakub Radoszewski
Opis rozwiązania: Jakub Łącki
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2012/biu`

Bajtazar jest nauczycielem w szkole podstawowej w Bajtołach Dolnych. Korzystając z tego, że pogoda całkiem dopisuje, Bajtazar chciałby zabrać swoją klasę na wycieczkę autokarową do stolicy kraju, Bajtogrodu. Do pomocy w organizacji wycieczki Bajtazar postanowił zatrudnić biuro podróży BajTour.

Ulice w centrum Bajtogrodu tworzą regularną siatkę: każda ulica biegnie albo z zachodu na wschód, albo z południa na północ, a odległości między dwiema sąsiednimi równoległymi ulicami są równe i wynoszą jeden kilometr. Przy niektórych skrzyżowaniach znajdują się atrakcje turystyczne. Bajtoccy przewodnicy każdej atrakcji przypisali pewien *współczynnik ciekawości*: im wyższy współczynnik, tym dana atrakcja jest ciekawsza dla zwiedzających. Bajtazar wie, że jego podopieczni szybko się nudzą, dlatego chciałby, aby kolejno zwiedzane atrakcje miały coraz większe współczynniki ciekawości.

Biuro BajTour zgodziło się spełnić wymagania Bajtazara, ale przy tym chciałoby na wycieczce jak najwięcej zarobić. Biuro pobiera stałą stawkę jednego bajtalara za każdy kilometr trasy autokaru. Przejeżdżając między dwiema kolejnymi atrakcjami na trasie zwiedzania, autokar porusza się zawsze najkrótszą trasą biegnącą wzdłuż ulic Bajtogrodu. Ponadto BajTour zarabia w jeszcze inny sposób: zarządcy niektórych atrakcji płacą biuru za przyprowadzanie wycieczek.

Celem BajTouru jest zaproponować trasę zgodną z warunkami postawionymi przez Bajtazara, która zagwarantuje BajTourowi możliwie największy zysk. Czy pomógłbyś w wyznaczeniu odpowiedniej trasy? Uwaga: przejechanie obok atrakcji turystycznej bez wysiadania nie liczy się jako jej zwiedzenie!

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite n oraz m ($2 \leqslant n, m \leqslant 1000$), oznaczające liczbę ulic biegnących z zachodu na wschód oraz liczbę ulic biegnących z południa na północ.

Dalej następuje n wierszy zawierających opis atrakcji turystycznych. W i-tym wierszu znajduje się m liczb całkowitych $w_{i,j}$ ($0 \leqslant w_{i,j} \leqslant 10^6$) oznaczających współczynniki ciekawości atrakcji turystycznych rozmieszczonych na skrzyżowaniach i-tej ulicy biegnącej ze wschodu na zachód z kolejnymi ulicami biegnącymi z południa na północ. Współczynnik 0 oznacza brak atrakcji turystycznej, natomiast dodatnie współczynniki opisują poszczególne atrakcje. Wiadomo, że w Bajtogrodzie jest co najmniej jedna atrakcja turystyczna.

Każdy z kolejnych n wierszy zawiera po m liczb całkowitych $c_{i,j}$ ($0 \leqslant c_{i,j} \leqslant 10^9$). Liczba $c_{i,j}$, czyli j-ta liczba w i-tym z tych wierszy, oznacza kwotę (w bajtalarach), jaką biuro otrzymuje za wysłanie wycieczki do atrakcji opisanej współczynnikiem ciekawości $w_{i,j}$. Jeśli przy skrzyżowaniu nie ma atrakcji, odpowiadająca mu liczba $c_{i,j}$ jest równa 0.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą, oznaczającą (wyrażony w bajtalarach) zysk z najbardziej dochodowej dla biura trasy, odwiedzającej pewne atrakcje turystyczne w kolejności ściśle rosnących współczynników ciekawości.

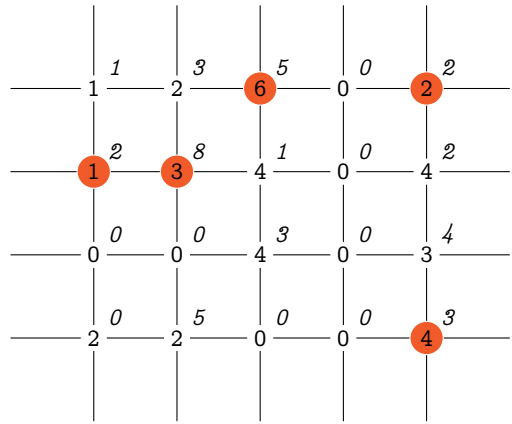
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
39
```



Wyjaśnienie przykładu: Liczby napisane na rysunku krojem prostym oznaczają współczynniki ciekawości atrakcji, a liczby napisane krojem pochyłym — dochody BajTouru za wysłanie wycieczki do poszczególnych atrakcji. Atrakcje odwiedzane na najbardziej dochodowej dla biura trasie zwiedzania są zaznaczone kółkami. Za wysłanie wycieczki do tych atrakcji biuro otrzyma, kolejno, 2, 2, 8, 3 i 5 bajtalarów. Dodatkowo, sumaryczny koszt przejazdu autokaru to 19 bajtalarów.

Rozwiązanie

W treści zadania *Biuro podróży* każdy doświadczony zawodnik odnajdzie graf skierowany. Wierzchołkami grafu są skrzyżowania w Bajtogrodzie, a krawędź między wierzchołkami u i w występuje, jeśli współczynnik ciekawości atrakcji obok skrzyżowania w jest ściśle większy niż analogiczny współczynnik dla skrzyżowania u. Od tego momentu zamiast o *współczynnikach ciekawości* mówić będziemy o *rangach* wierzchołków. W tak zdefiniowanym grafie nie ma żadnych cykli. Po przejściu każdej krawędzi ranga aktualnego wierzchołka staje się coraz większa, zatem jasne jest, że nie da się chodzić w kółko.

Wspomniany graf jest również ważony i to w dwojaki sposób. Po pierwsze każdemu wierzchołkowi przypisujemy wagę równą zyskowi biura za przysłanie wycieczki do atrakcji stojącej przy odpowiadającym skrzyżowaniu. Po drugie krawędź między dwoma wierzchołkami ma wagę równą odległości między skrzyżowaniami. Naszym zadaniem jest znalezienie w tym grafie ścieżki o największej wadze, przy czym wagą ścieżki jest suma wag jej wierzchołków i krawędzi.

Aby rozwiązać problem, moglibyśmy się posłużyć standardowym algorytmem, który znajduje ścieżkę o największej wadze w grafie acyklicznym. Ten algorytm przetwarza wierzchołki grafu w porządku topologicznym. Dla danego wierzchołka w wyznacza maksymalną wagę ścieżki, która kończy się w w. W tym celu przegląda wszystkie krawędzie wchodzące do w. Dla krawędzi uw próbuje utworzyć nową ścieżkę przez połączenie ścieżki o maksymalnej wadze prowadzącej do u oraz krawędzi uw. Algorytm ten działa w czasie liniowym. Problem w tym, że dane wejściowe w naszym zadaniu są tak duże, iż potrzebnego nam grafu nie bylibyśmy w stanie skonstruować w sposób jawny i zmieścić się w limicie czasu. Gdyby każdy wierzchołek miał inną rangę, między każdą parą wierzchołków mielibyśmy krawędź (w jednym z dwóch kierunków). Skoro dla $m = n = 1000$ liczba wierzchołków jest równa $n^2 = 1\,000\,000$, w powstałym grafie byłoby $n^2(n^2-1)/2 \approx 5 \cdot 10^{11}$ krawędzi.

Dla uproszczenia załóżmy teraz, że rangi wierzchołków mieszczą się w przedziale od 1 do r oraz dla każdego i z tego przedziału mamy co najmniej jeden wierzchołek o takiej właśnie randze. Dla każdego $i \in [1, r]$, na szukanej ścieżce może znaleźć się co najwyżej jeden wierzchołek o randze i. Z drugiej strony jeśli ścieżka ma mieć maksymalną wagę, to najbardziej opłaca się odwiedzić dokładnie r wierzchołków: po jednym dla każdej rangi. Zauważmy, że do dowolnej trasy, która nie przechodzi przez wierzchołek o randze i, możemy dodać przejście przez dowolny wierzchołek o randze i. Łatwo przekonać się, że na pewno nie zmniejszy to łącznej wagi ścieżki.

Algorytm będzie działać w r fazach. W fazie i obliczymy maksymalne wagi ścieżek kończących się w wierzchołkach o randze i. Przypadek $i = 1$ jest bardzo prosty: każda ścieżka składa się z tylko jednego wierzchołka. W kolejnych fazach, gdy $i > 1$, używamy wartości wyznaczonych dla wierzchołków o randze $i-1$, by wyznaczyć je dla wierzchołków o randze i. Niech k_i oznacza liczbę wierzchołków o randze i. Wówczas taką fazę chcemy zrealizować w czasie $O(k_{i-1} + k_i)$. W ten sposób wszystkie fazy będą działać w czasie

$$O(k_1) + O(k_1 + k_2) + \ldots + O(k_{r-1} + k_r) = O(k_1 + k_2 + \ldots + k_r) = O(mn),$$

czyli w czasie liniowym od liczby wierzchołków.

Jedna faza

Zajmijmy się teraz realizacją jednej fazy. Zauważmy, że pomiędzy dowolnymi dwoma wierzchołkami przemieszczamy się po jednym z czterech *skośnych kierunków*: trasę między wierzchołkami możemy przebyć, poruszając się tylko w prawo lub do góry, albo tylko w prawo i w dół, albo tylko w lewo i w dół, albo wreszcie tylko w lewo i do góry (jeśli wierzchołki znajdują się na tej samej prostej poziomej lub pionowej, mamy dowolność w wyborze tego, do którego kierunku je zakwalifikować). Rozpatrzmy każdy z tych kierunków osobno.

Zajmijmy się ustalonym kierunkiem, powiedzmy *w prawo i do góry*. Chcielibyśmy obliczyć wyniki dla wszystkich wierzchołków o randze i na podstawie wyników dla wierzchołków o randze $i-1$, przy założeniu że przemieszczamy się tylko

w prawo i do góry. Dla wierzchołka w oznaczmy przez $wynik(w)$ maksymalną wagę ścieżki kończącej się w w. Wartości $wynik(w)$ mamy już obliczone dla wierzchołków o randze $i-1$.

Ujmijmy rzecz bardziej formalnie. Przy rozpatrywaniu kierunku w prawo i do góry dla każdego wierzchołka w wyznaczymy wartość, która jest

- (własność $\succeq$) *nie mniejsza* niż maksymalna waga ścieżki do w, której ostatni fragment przechodzimy w prawo i do góry, oraz
- (własność $\preceq$) *nie większa* niż maksymalna waga ścieżki do w.

W przypadku gdy ostatni odcinek ścieżki do w o maksymalnej wadze prowadzi w prawo i do góry, ograniczenia z powyższych własności spotykają się, zatem wyznaczymy wartość równą maksymalnej wadze ścieżki do w. Wartość $wynik(w)$ otrzymamy, biorąc maksymalną z wartości otrzymanych przy rozpatrywaniu czterech kierunków. Własności nazywamy $\succeq$ i $\preceq$, aby móc się do nich odnieść w dalszej części opisu.

Zacznijmy od wykazania pomocniczego faktu.

Fakt 1. Odległość między punktem (x_u, y_u) oraz punktem (x_w, y_w) jest nie mniejsza niż $x_w - x_u + y_w - y_u$.

Dowód: Odległość między punktami to dokładnie $|x_w - x_u| + |y_w - y_u|$. Ponieważ dla każdej liczby rzeczywistej r mamy $|r| \geqslant r$, natychmiast otrzymujemy tezę. □

Rozważmy teraz wierzchołek w o randze i i współrzędnych (x_w, y_w) oraz wierzchołek u o randze $i-1$ i współrzędnych (x_u, y_u). Przyjmijmy, że z u do w idziemy w prawo i do góry, zatem $x_w \geqslant x_u$ oraz $y_w \geqslant y_u$. Jeśli do w przejdziemy po ścieżce przez u, to otrzymamy ścieżkę o wadze

$$wynik(u)+|x_w-x_u|+|y_w-y_u|+waga(w) = wynik(u)+x_w-x_u+y_w-y_u+waga(w).$$

Wartość bezwzględną możemy pominąć, gdyż $x_w \geqslant x_u$ oraz $y_w \geqslant y_u$. Zatem aby zbudować ścieżkę do w o maksymalnej wadze, należy wybrać wierzchołek u, który maksymalizuje wartość

$$przydatnosc(u) := wynik(u) - x_u - y_u. \tag{1}$$

Dzięki temu ostatecznie skonstruowana ścieżka będzie miała wagę

$$waga(w) + x_w + y_w + przydatnosc(u), \tag{2}$$

a zatem im większa wartość $przydatnosc(u)$, tym lepiej.

Jesteśmy już gotowi do opisania algorytmu. Budujemy tablicę zawierającą wierzchołki o rangach i oraz $i-1$. Następnie sortujemy wierzchołki w tablicy niemalejąco względem *sumy* ich współrzędnych. W efekcie, jeśli z wierzchołka u do wierzchołka w faktycznie idziemy w prawo i do góry, to w kolejności posortowanej w znajdzie się później niż u.

Zacznijmy przeglądać kolejne wierzchołki, idąc w kierunku rosnącej sumy współrzędnych. Będziemy cały czas pamiętać najbardziej przydatny dotąd przetworzony wierzchołek p o randze $i-1$, czyli taki o największej wartości $przydatnosc(p)$. Oznacza to, że właśnie z niego najbardziej opłaca się konstruować ścieżkę do każdego napotkanego wierzchołka o randze i. A zatem w trakcie przeglądania wierzchołków,

gdy napotykamy wierzchołek o randze $i-1$, sprawdzamy, czy nie zastępuje on dotąd najbardziej przydatnego (wzór (1)). Jeśli zaś trafimy na wierzchołek o randze i, konstruujemy nową ścieżkę do tego wierzchołka, korzystając z aktualnie najbardziej przydatnego wierzchołka o randze $i-1$ (wzór (2)). Następnie sprawdzamy, czy skonstruowana ścieżka pozwala nam poprawić obliczony wynik dla przetwarzanego wierzchołka o randze i.

Dlaczego ten algorytm jest poprawny? Rozważmy jeszcze raz wierzchołek w o randze i i współrzędnych (x_w, y_w). Skonstruowaliśmy ścieżkę do tego wierzchołka, której przedostatnim wierzchołkiem jest u (o randze $i-1$ i współrzędnych (x_u, y_u)). Jeśli w leży w prawo i do góry od u, to na pewno skonstruowaliśmy ścieżkę do w o maksymalnej wadze, której ostatni fragment prowadzi w prawo i do góry, bo wybraliśmy u tak, by zmaksymalizować $waga(w) + x_w + y_w + przydatnosc(u)$. Jednak w niekoniecznie musi leżeć w prawo i do góry od u. Zapewniliśmy jedynie, że u znalazł się wcześniej od w w kolejności posortowanej, czyli $x_u + y_u \leqslant x_w + y_w$, a to niekoniecznie oznacza, że z u do w przechodzi się w prawo i do góry (czyli $x_u \leqslant x_w$ i $y_u \leqslant y_w$).

Załóżmy zatem, że z u do w idziemy w innym kierunku. Wówczas zachodzą dwa fakty, które odpowiadają wcześniej wspomnianym własnościom $\succeq$ i $\preceq$. Po pierwsze, nie istnieje wierzchołek u', z którego do w idzie się w prawo i do góry i który dałby ścieżkę do w o wadze większej niż wyznaczona przez nas $waga(w) + x_w + y_w + przydatnosc(u)$, bo to oznaczałoby, że $przydatnosc(u') > przydatnosc(u)$, a przecież u to wierzchołek o maksymalnej przydatności. Po drugie, wartość $waga(w) + x_w + y_w + przydatnosc(u)$ jest *nie większa* niż maksymalna waga ścieżki, która do w prowadzi przez u. To wynika z faktu, że przy obliczaniu wagi ścieżki do w za odległość między u a w przyjęliśmy $x_w - x_u + y_w - y_u$. Ten wzór działa, jeśli z u do w idziemy w prawo i do góry. W ogólności jednak, zgodnie z faktem 1, odległość między u a w jest *nie mniejsza* niż $x_w - x_u + y_w - y_u$. Tym samym wykazaliśmy dwie zapowiedziane własności, których potrzebowaliśmy do udowodnienia poprawności algorytmu.

Efektywne sortowanie

Pozostaje jeszcze wspomnieć o jednym szczególe. W trakcie każdej fazy powinniśmy posortować wierzchołki, które biorą w niej udział, na cztery różne sposoby. Możemy ograniczyć się do dwóch sposobów sortowania, jeśli zauważymy, że kolejność „w lewo i w dół" to po prostu kolejność odwrotna do „w prawo i do góry". Aby sortować efektywnie, skorzystamy z faktu, że suma współrzędnych nie przekracza $m + n$. Dzięki temu możemy zastosować sortowanie przez zliczanie, które dla k liczb z zakresu $[1, W]$ działa w czasie $O(k + W)$. Na samym początku posortujemy więc *wszystkie* wierzchołki względem sumy współrzędnych i osobno względem różnicy współrzędnych. Następnie posegregujemy je względem ich rangi, to znaczy dla każdej możliwej rangi utworzymy dwie tablice posortowane na dwa różne sposoby. Aby przeprowadzić jedną fazę wystarczy teraz złączyć ze sobą dwie posortowane tablice, co da się zrobić w czasie liniowym.

Taka optymalizacja nie była konieczna by rozwiązać zadanie na zawodach. Akceptowane były nawet rozwiązania, które w trakcie fazy przetwarzającej t wierzchołków wykonywały cztery sortowania w czasie $O(t \log t)$. Jednak dzięki szybszemu sortowaniu otrzymujemy algorytm optymalny, który całe zadanie rozwiązuje w czasie $O(mn)$, czyli takim jak rozmiar danych wejściowych.

C

Ciąg

Autor zadania: Jakub Radoszewski
Opis rozwiązania: Jakub Radoszewski
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2012/cia`

Powiemy, że ciąg liczb całkowitych $a_1, a_2, \ldots, a_n$ jest *k-parzysty*, jeśli każdy jego k-elementowy spójny fragment ma parzystą sumę.

Dla danego ciągu liczb całkowitych chcielibyśmy stwierdzić, ile minimalnie wyrazów tego ciągu musimy zmienić, aby stał się on k-parzysty.

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite n oraz k ($1 \leqslant k \leqslant n \leqslant 1\,000\,000$), oznaczające długość ciągu i parametr parzystości ciągu. Drugi wiersz zawiera ciąg n liczb całkowitych $a_1, a_2, \ldots, a_n$. Każda z liczb a_i spełnia $0 \leqslant a_i \leqslant 1\,000\,000\,000$.

Wyjście

W jedynym wierszu wyjścia należy wypisać jedną liczbę całkowitą, oznaczającą najmniejszą liczbę wyrazów podanego ciągu, które trzeba zmienić, żeby ciąg stał się k-parzysty.

Przykład

Dla danych wejściowych:	poprawnym wynikiem jest:
`8 3` `1 2 3 4 5 6 7 8`	`3`
natomiast dla danych:	poprawnym wynikiem jest:
`8 3` `2 4 2 4 2 4 2 4`	`0`

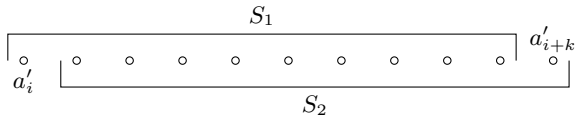
Rozwiązanie

Naszym zadaniem jest zmodyfikowanie danego ciągu liczbowego tak, by stał się on k-parzysty. Zauważmy na wstępie, że nie interesują nas dokładne wartości wyrazów tego ciągu, a jedynie ich parzystość. Możemy więc każdy z wyrazów ciągu zamienić na resztę z dzielenia go przez 2 i odtąd zajmować się wyłącznie ciągami binarnymi.

Łatwo stwierdzić, które konkretnie fragmenty ciągu długości k mają nieparzystą sumę. Wiemy, że w każdym takim fragmencie musimy zmienić parzystość co najmniej jednego wyrazu, jednak nie jest jasne, który konkretnie wyraz najlepiej

wybrać. Ta droga prowadzi do różnych rozwiązań heurystycznych i zapewne trudno w ten sposób otrzymać poprawne rozwiązanie.

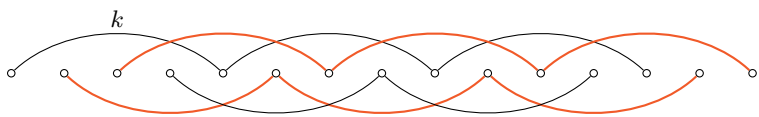
W przypadku tego zadania dużo lepszą metodą jest skoncentrować się na celu i w ten sposób odkryć prawdziwą strukturę problemu. Niech $a'_1, \ldots, a'_n$ oznacza wynikowy ciąg, do którego dążymy. Chcemy, aby każdy fragment tego ciągu długości k miał parzystą sumę. Przyjrzyjmy się dwóm kolejnym takim fragmentom.



Rysunek 1. Dwa kolejne fragmenty długości k w ciągu wynikowym.

Załóżmy, że są to fragmenty $a'_i, \ldots, a'_{i+k-1}$ oraz $a'_{i+1}, \ldots, a'_{i+k}$ o sumach, odpowiednio, S_1 i S_2 (patrz rysunek 1). Mamy $S_2 = S_1 + a'_{i+k} - a'_i$. Ponieważ w ciągu wynikowym S_1 i S_2 są parzyste, więc wyrazy a'_i oraz a'_{i+k} muszą być tej samej parzystości. Ale przecież wszystkie wyrazy ciągu są pojedynczymi bitami! W wynikowym ciągu musi więc po prostu zachodzić $a'_i = a'_{i+k}$, i to dla dowolnie wybranego indeksu i z zakresu od 1 do $n-k$. To oznacza, że ciąg wynikowy jest *okresowy* z okresem k.

Zajmijmy się więc następującym problemem: dany ciąg binarny $a_1, \ldots, a_n$ chcemy przekształcić w ciąg okresowy o okresie k, zmieniając możliwie najmniej wyrazów. Nasz ciąg rozpada się na k łańcuchów wyrazów położonych co k indeksów w ciągu. Teraz wystarczy tak zmienić wyrazy ciągu, by wyrazy w każdym łańcuchu stały się równe (patrz rysunek 2).



Rysunek 2. Łańcuchy składające się z co czwartego wyrazu ciągu.

Zliczmy w każdym łańcuchu zera i jedynki. Jeśli zer jest więcej niż jedynek, to najbardziej opłaca nam się zamienić w tym łańcuchu wszystkie jedynki w zera. I odwrotnie, w przypadku przewagi jedynek wszystkie zera w łańcuchu zamieniamy w jedynki.

Mamy więc bardzo prosty algorytm na to, by za pomocą najmniejszej liczby zmian przekształcić początkowy ciąg binarny w ciąg o okresie k. Czy jest to już ciąg wynikowy? Otóż być może jeszcze nie. Warto jeszcze raz zastanowić się nad tym, co ustaliliśmy w trakcie początkowych rozważań. Wiemy, że aby ciąg binarny był k-parzysty, musi on być okresowy z okresem k. Jednak sama okresowość nie wystarcza: oznacza ona jedynie, że wszystkie fragmenty długości k mają sumy o tej samej parzystości — czyli wszystkie te sumy są albo parzyste, albo nieparzyste! W tym pierwszym przypadku rzeczywiście mamy w ręku rozwiązanie zadania. Jednak co zrobić, gdy zajdzie drugi przypadek?

Jedyne, co możemy robić, to zamieniać wartości wyrazów w całych łańcuchach. Zauważmy, że zamiana wartości w jednym, dowolnym łańcuchu zmieni parzystość

sumy pierwszego k-wyrazowego fragmentu ciągu, więc także wszystkie pozostałe fragmenty będą miały parzystą sumę. Przy czym opłaca nam się wybrać taki łańcuch, w którym liczba zer i liczba jedynek są do siebie jak najbardziej zbliżone (tzn. wartość bezwzględna różnicy między liczbą zer a liczbą jedynek jest jak najmniejsza). Faktycznie, załóżmy, że optymalny łańcuch do zamiany zawiera z zer i j jedynek oraz że $z \leqslant j$ (przypadek $z \geqslant j$ jest symetryczny). Wówczas w pierwszym ciągu, otrzymanym przez zachłanne obsłużenie łańcuchów, liczba zmian w tym łańcuchu wynosiła z, a po zamianie w tym łańcuchu będziemy *zamiast tego* wykonywać j zmian. Całkowita liczba zamian wzrośnie nam zatem o $j - z$, w ogólności o $|j - z|$. Dobrze zatem, aby $|j - z|$ było możliwie najmniejsze.

Przykład

Rozważmy ciąg

$$a = 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 0$$

z parametrem $k = 4$. Aby lepiej zobaczyć, jak w tym przypadku wygląda struktura łańcuchów, zapiszmy ciąg w kolejnych wierszach tabelki tak, aby wyrazy o indeksach różniących się o 4 znajdowały się jeden pod drugim. Innymi słowy, każda z kolumn zawiera jeden łańcuch:

1	0	1	0
0	0	0	0
1	0	1	0
0	1	1	0
0	0		

Optymalne zrównanie wszystkich wyrazów w każdym łańcuchu prowadzi do następującego ciągu (wyróżnione zostały zmienione wyrazy):

0	0	1	0
0	0	1	0
0	0	1	0
0	0	1	0
0	0		

Nie jest to jeszcze ciąg wynikowy, bowiem każdy fragment długości 4 ma w nim nieparzystą sumę. Musimy jedną z kolumn zmienić na odwrót. Najbardziej opłaca się to zrobić w pierwszej kolumnie, gdzie wykonamy dodatkowo jedną zmianę. Oto przykładowy ciąg wynikowy, zapisany w formie tabelki. Do jego otrzymania potrzebowaliśmy pięciu zmian zaznaczonych w tabelce:

1	0	1	0
1	0	1	0
1	0	1	0
1	0	1	0
1	0		

D

DNA

Autor zadania: Jakub Łącki
Opis rozwiązania: Jakub Łącki
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2012/dna`

Szalony naukowiec Bajtazar chciałby stworzyć nowy gatunek zwierząt. W tym celu postanowił zmodyfikować kod DNA myszy bajtockiej.

Kod DNA to ciąg znaków składający się z liter `A`, `C`, `G` oraz `T`. Plan Bajtazara jest następujący. Weźmie on DNA myszy i na jego podstawie stworzy nowy kod o tej samej długości, który będzie *jak najmniej* podobny do kodu myszy. Podobieństwo dwóch kodów DNA to długość ich najdłuższego wspólnego podciągu. Najdłuższy wspólny podciąg dwóch słów x i y to najdłuższe słowo, które można uzyskać z każdego ze słów x, y przez usuwanie liter. (Zwróć uwagę, że dwa słowa mogą mieć wiele najdłuższych wspólnych podciągów, na przykład najdłuższe wspólne podciągi słów `CACCA` i `CAAC` to `CAA` oraz `CAC`.) Napisz program, który wyznaczy szukany kod DNA.

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 10\,000$) oznaczającą długość kodu DNA myszy bajtockiej. W drugim wierszu znajduje się kod DNA myszy w postaci ciągu n wielkich liter należących do zbioru $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$.

Wyjście

Pierwszy wiersz wyjścia powinien zawierać jedną liczbę całkowitą — podobieństwo kodu myszy bajtockiej oraz kodu znalezionego przez Twój program. W drugim wierszu należy wypisać ciąg składający się z n liter `A`, `C`, `G` lub `T`. Powinien być to kod DNA, który jest jak najmniej podobny do kodu podanego na wejściu. Jeśli istnieje wiele poprawnych odpowiedzi, Twój program może wypisać dowolną z nich.

Przykład

Dla danych wejściowych:

```
4
GACT
```

jednym z poprawnych wyników jest:

```
1
TCAG
```

Rozwiązanie

Kiedy opisujemy rozwiązania zadań, zazwyczaj staramy się prezentować kolejne kroki myślowe, jakie należy wykonać, aby w końcu otrzymać poszukiwany algorytm. Tym razem będzie inaczej, bo w zadaniu DNA na rozwiązanie trzeba po prostu wpaść. Warto pewnie rozwiązać kilka przykładów na kartce, zastanowić się lub zdać na intuicję. Ostatecznie jednak algorytm musimy zgadnąć.

Rozwiązanie daje się opisać w jednym zdaniu. Wystarczy znaleźć literę, która w podanym kodzie DNA myszy bajtockiej występuje najrzadziej, i wypisać ją n razy. W szczególności, jeśli pewna z liter `A`, `C`, `G` lub `T` nie występuje w podanym kodzie, tę właśnie literę wypiszemy n razy i wówczas najdłuższy wspólny podciąg znalezionego kodu i kodu DNA myszy będzie pusty, czyli tak krótki jak to tylko możliwe.

Pozostaje nam uzasadnić poprawność tego rozwiązania. Niech $\mathrm{NWP}(x, y)$ oznacza długość najdłuższego wspólnego podciągu słów x i y. Oznaczmy też przez w podany kod myszy bajtockiej i przyjmijmy, że najrzadziej występującą literą w słowie w jest `A`, która występuje a razy. Niech y to słowo składające się z n liter `A`. Wtedy to $\mathrm{NWP}(w, y) = a$, bo w słowie y są same litery `A`, a w słowie w jest dokładnie a liter `A`.

Aby dowieść poprawności naszego rozwiązania wystarczy wykazać, że „lepiej się nie da". Uzasadnimy więc, że dla dowolnego słowa x długości n zachodzi $\mathrm{NWP}(w, x) \geqslant a$.

Rozważmy dwa przypadki. Pierwszy jest bardzo prosty: jeśli słowo x zawiera co najmniej a wystąpień litery `A`, dostajemy natychmiast $\mathrm{NWP}(w, x) \geqslant a$, co kończy dowód w tym przypadku. Pozostaje rozważyć słowo x, w którym litera `A` występuje mniej niż a razy, czyli mniej razy niż w słowie w. Wówczas jedna z liter `C`, `G` lub `T` występuje w słowie x co najmniej tyle razy, co w słowie w. Dlaczego? Gdyby każda z liter `A`, `C`, `G` i `T` miała w słowie x ściśle mniej wystąpień niż w słowie w, to słowo x składałoby się z mniejszej liczby liter niż słowo w. To oczywiście stanowi sprzeczność, gdyż oba słowa mają długość n.

Zatem pewna litera, powiedzmy `C`, która w słowie w występuje c razy, występuje co najmniej c razy w słowie x. To oznacza, że ciąg składający się z c wystąpień litery `C` to podciąg zarówno słowa w jak i słowa x. Co więcej, $c \geqslant a$ (litera `A` występuje najrzadziej), czyli $\mathrm{NWP}(w, x) \geqslant c \geqslant a$. Tym samym dowiedliśmy poprawności naszego rozwiązania.

Posłowie

Kiedy przygotowywaliśmy Mistrzostwa Polski w 2012 roku, zadanie DNA było naszym kandydatem na najprostsze zadanie na zawodach. Wszak całe rozwiązanie streszcza się w zaledwie kilku słowach. Jak się okazało, pomyliliśmy się dosyć znacznie, bo nie wzięliśmy pod uwagę co najmniej trzech faktów przemawiających za trudnością zadania.

Po pierwsze, rozwiązanie znaleźć można stosunkowo łatwo, jednak dopiero gdy zacznie się szukać wśród prostych pomysłów. Innymi słowy, to zadanie jest proste, jeśli tylko wie się, że jest proste. Niestety absolutnie nie widać tego w treści.

Po drugie, test przykładowy został skonstruowany podchwytliwie: dla podanego ciągu `GACT` przykładową odpowiedzią jest `TCAG`, czyli słowo z wejścia przeczytane wspak (choć równie dobrą odpowiedzią jest na przykład `AAAA`).

Wreszcie po trzecie, ograniczenie na długość DNA myszy to zaledwie 10 000 — znacznie mniej niż w większości zadań, w których rozwiązanie działa w czasie liniowym. Czy to kolejna pułapka celowo zastawiona przez organizatorów? Otóż nie — tak małe ograniczenie było konieczne ze względów technicznych. W końcu, aby sprawdzić poprawność odpowiedzi, trzeba uruchomić kwadratowy algorytm szukania najdłuższego wspólnego podciągu. Zatem sam program sprawdzający działał w czasie $\Theta(n^2)$ i wymuszał takie ograniczenie rozmiaru danych testowych, by programy nadsyłane w trakcie zawodów mogły zostać ocenione w ciągu kilku sekund.

Ewaluacja

Autor zadania: Jakub Radoszewski
Opis rozwiązania: Jakub Radoszewski
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2012/ewa`

Dane jest wyrażenie matematyczne E, w którym występują: stałe od 0 do 9, zmienne od a do z, a także operacje: dodawania, mnożenia i potęgowania o stałym wykładniku. Co ciekawe, każda ze zmiennych $a, b, \ldots, z$ występuje w wyrażeniu E *co najwyżej raz*. Zastanawiamy się, dla danej liczby pierwszej p, ile pierwiastków modulo p ma wielomian wyznaczony przez to wyrażenie. Innymi słowy, pytamy, ile jest podstawień liczb od 0 do $p-1$ pod zmienne występujące w E, dla których wartość wyrażenia E jest podzielna przez p. Ponieważ szukana liczba pierwiastków może być duża, wystarczy nam reszta z jej dzielenia przez 30 011.

Przykładowo, wielomian reprezentowany przez wyrażenie

$$E = ((a + y) \cdot (z + 8))^2$$

ma 15 pierwiastków modulo $p = 3$, m.in. następujące trzy pierwiastki:

$$(a = 0,\ y = 0,\ z = 0), \qquad (a = 1,\ y = 2,\ z = 0), \qquad (a = 2,\ y = 0,\ z = 1).$$

Formalnie, *wyrażenie* definiujemy następująco:

- Każda stała 0, 1, ..., 9 jest wyrażeniem.
- Każda zmienna a, b, ..., z jest wyrażeniem.
- Jeśli `A` i `B` są dowolnymi wyrażeniami, to wyrażeniami są także `(A+B)` i `(A*B)`. Pierwsze z nich oznacza sumę wyrażeń `A` i `B`, zaś drugie — ich iloczyn.
- Jeśli `A` jest dowolnym wyrażeniem, a `B` jest stałą z zakresu 2, 3, ..., 9, to wyrażeniem jest także `(A^B)` (wyrażenie `A` do potęgi `B`).

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę pierwszą p ($2 \leqslant p < 15\,000$). Drugi wiersz zawiera wyrażenie E zgodne z podaną specyfikacją, opisane przez ciąg złożony z co najwyżej 300 znaków 0, 1, ..., 9, a, b, ..., z, +, *, ^, (,). W podanym ciągu nie występują odstępy.

Wyjście

Oznaczmy przez k liczbę pierwiastków modulo p wielomianu E. Twój program powinien wypisać jedną nieujemną liczbę całkowitą: resztę z dzielenia k przez 30 011.

Przykład

Dla danych wejściowych:

```
3
(((a+y)*(z+8))^2)
```

poprawnym wynikiem jest:

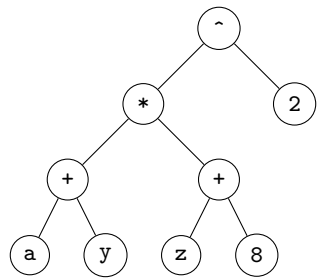
```
15
```

Rozwiązanie

Na pierwszy rzut oka zadanie *Ewaluacja* może wyglądać na problem czysto matematyczny. Okazuje się jednak, że jego rozwiązanie zdecydowanie bardziej opiera się na pomysłach natury algorytmicznej. Pewną wskazówką w tym kierunku jest znajdujące się w treści zadania założenie, że każda zmienna w rozważanym wyrażeniu występuje co najwyżej raz. Tego typu ograniczenia rzadko spotyka się w matematyce. Spróbujmy więc podejść do rozwiązania zadania, od razu stosując metody wykorzystywane w informatyce przy przetwarzaniu wyrażeń.

Parsowanie

W pierwszym kroku przekształcamy podane wyrażenie do nieco wygodniejszej postaci. Budujemy tzw. drzewo wyrażenia, którego liście reprezentują stałe i zmienne, natomiast węzły wewnętrzne odpowiadają działaniom, tj. dodawaniu, mnożeniu i potęgowaniu. Przykładowo, drzewo wyrażenia opisanego w treści zadania wygląda tak jak na rysunku 1.



Rysunek 1. Drzewo wyrażenia $E = ((a + y) \cdot (z + 8))^2$.

Przekształcanie wyrażenia z postaci tekstowej do reprezentacji strukturalnej określa się z angielska mianem *parsowania*. W kręgach zawodniczych słowo to budzi negatywne skojarzenia, gdyż fragment programu realizujący parsowanie opiera się zazwyczaj na analizie wielu niezbyt ciekawych przypadków. Jednak w przypadku naszego zadania parsowanie wykonać możemy wyjątkowo prosto, gdyż dane wejściowe spełniają dwa następujące warunki. Po pierwsze, każda stała i zmienna

jest opisana za pomocą tylko jednego znaku. A po drugie, każde wyrażenie oznaczające działanie jest otoczone parą nawiasów.

W tej sytuacji drzewo wyrażenia można zbudować za pomocą jednej funkcji rekurencyjnej. Przyjmuje ona w argumencie indeks pierwszego znaku podwyrażenia, dla którego chcemy zbudować fragment drzewa, a daje w wyniku indeks końcowego znaku tego podwyrażenia. Decyzja o tym, jak budować fragment drzewa, podejmowana jest na podstawie pierwszego znaku podwyrażenia. Jeśli znak ten jest literą lub cyfrą, to jest on zarazem końcowym znakiem, a reprezentacją podwyrażenia jest po prostu pojedynczy liść. W przeciwnym razie pierwszym znakiem podwyrażenia musi być nawias otwierający. Wówczas funkcja:

- rekurencyjnie wyznacza lewe poddrzewo opisujące podwyrażenie zaczynające się na kolejnym znaku,
- w korzeniu drzewa umieszcza działanie występujące bezpośrednio po zakończeniu opisu tego podwyrażenia,
- następnie rekurencyjnie wyznacza prawe poddrzewo odpowiadające podwyrażeniu zaczynającemu się bezpośrednio po znaku działania,
- daje w wyniku indeks nawiasu zamykającego występującego po opisie tego drugiego podwyrażenia.

Widać wyraźnie, że funkcja działa w czasie liniowym względem długości wyrażenia. Takiego samego rzędu jest też rozmiar otrzymanego drzewa wyrażenia.

Programowanie dynamiczne na drzewie

Mając do dyspozycji drzewo wyrażenia, otrzymujemy naturalny podział całego problemu na podproblemy odpowiadające poszczególnym poddrzewom. Szukaną liczbę pierwiastków wyrażenia modulo p możemy teraz obliczyć za pomocą programowania dynamicznego. Istotnie korzystamy tutaj z założenia, że każda zmienna występuje w wyrażeniu co najwyżej raz, dzięki czemu rozłączne poddrzewa mają rozłączne zbiory zmiennych, i obliczenia możemy wykonywać dla nich niezależnie.

Łatwo przekonać się, że do obliczenia wyniku dla drzewa na podstawie wyników z jego poddrzew nie wystarczy przechowywać w każdym poddrzewie liczby pierwiastków. Dla każdego węzła u potrzebujemy znać liczbę podstawień do wyrażenia reprezentowanego przez u, które dają każdą z możliwych reszt $0, \ldots, p-1$ modulo p. Oznaczmy taką tablicę dla węzła u przez t_u. Zacznijmy od tego, że umiemy wyznaczyć taką tablicę dla liści drzewa:

- jeśli u reprezentuje stałą $c \in \{0, \ldots, 9\}$, to $t_u[c'] = 1$ i $t_u[k] = 0$ dla $k \neq c'$, gdzie $c' = c \bmod p$,
- jeśli u reprezentuje zmienną, to $t_u[k] = 1$ dla wszystkich k.

W przypadku, gdy u jest węzłem wewnętrznym, wszystko zależy oczywiście od tego, jakie działanie on reprezentuje. Jeśli jest to dodawanie lub mnożenie, a dziećmi węzła u są węzły v oraz w, to dla każdego $k \in \{0, \ldots, p-1\}$ mamy

- $t_u[k] = \sum_{i+j \equiv k \pmod p} t_v[i] \cdot t_w[j]$ w przypadku dodawania,
- $t_u[k] = \sum_{i \cdot j \equiv k \pmod p} t_v[i] \cdot t_w[j]$ w przypadku mnożenia.

Wytłumaczmy pierwszy z tych wzorów. Aby reszta z dzielenia przez p sumy dwóch wyrażeń była równa k, reszty z dzielenia wartości tych wyrażeń muszą sumować się do k (modulo p). Przeglądamy zatem wszystkie takie pary reszt (i, j) i dla każdej z nich bierzemy do wyniku wszystkie możliwe podstawienia dające resztę i w pierwszym podwyrażeniu i resztę j w drugim podwyrażeniu. Takich podstawień jest dokładnie $t_v[i] \cdot t_w[j]$, a to właśnie dlatego, że mamy gwarancję, iż żadna zmienna z pierwszego podwyrażenia nie występuje w drugim. Innymi słowy, podstawienia zmiennych w obydwu podwyrażeniach są od siebie niezależne. Uzasadnienie wzoru odpowiadającego mnożeniu wyrażeń jest zupełnie analogiczne.

Wreszcie jeśli u reprezentuje potęgowanie o wykładniku b wyrażenia odpowiadającego węzłowi v, to dla dowolnego k mamy

- $t_u[k] = \sum_{i^b \equiv k \pmod p} t_v[i]$.

Przypomnijmy jeszcze, że ponieważ szukane liczby podstawień mogą być duże, w zadaniu wymaga się od nas jedynie podania reszty z dzielenia wynikowej liczby pierwiastków przez $q = 30\,011$. W obliczeniach możemy zrobić więc to samo i wszystkie działania na elementach tablic t wykonywać modulo q.

Po zakończeniu wszystkich obliczeń w korzeniu r drzewa odczytamy z $t_r[0]$ szukaną liczbę pierwiastków. Powinniśmy się jednak zainteresować tym, ile czasu wymagają wszystkie obliczenia prowadzące do otrzymania tej wartości.

Oznaczmy liczbę węzłów drzewa przez n. Łączny rozmiar wszystkich tablic t to $O(np)$, czyli dostatecznie mało. Jeśli u jest liściem, to wyznaczenie t_u zajmuje oczywiście czas $O(p)$. Tak samo szybko możemy obliczyć wszystkie elementy tablicy t_u w przypadku, gdy węzeł ten odpowiada potęgowaniu. Wystarczy obliczać je zgodnie ze wzorem — nie według k, ale według i: w chwili rozważania $t_v[i]$ zwiększamy $t_u[k]$ dla $k = i^b \bmod p$.

Zdecydowanie bardziej czasochłonne obliczenia wykonujemy dla pozostałych działań (dodawania i mnożenia). W przypadku węzła odpowiadającego dodawaniu dla każdego $k \in \{0, \ldots, p-1\}$ przeglądamy wszystkie i z takiego samego zakresu i wykonujemy pojedyncze obliczenie dla $j = (k - i) \bmod p$ (tak naprawdę będzie to zawsze $j = k - i$ lub $j = k + p - i$). W węźle odpowiadającym mnożeniu robimy to samo, tylko dla $j = (k \cdot i^{-1}) \bmod p$ — korzystamy tu ze znanego faktu teorioliczbowego, że jeśli p jest liczbą pierwszą, to każda niezerowa reszta i ma dokładnie jedną odwrotność modulo p, którą oznaczamy przez i^{-1}. Odwrotność ta spełnia $i \cdot i^{-1} \equiv 1 \pmod p$. W obu przypadkach złożoność czasowa obliczeń to $O(p^2)$, przyjmując optymistyczne założenie, że umiemy szybko liczyć odwrotności modulo. W najgorszym przypadku wszystkie obliczenia zajmują zatem czas $O(np^2)$. Przy limitach z treści zadania to niestety za dużo.

FFT

Tytułowa szybka transformata Fouriera (FFT) pozwala obliczyć współczynniki wielomianu będącego iloczynem dwóch wielomianów

$$A(x) = \sum_{i=0}^{m-1} a_i x^i \quad \text{i} \quad B(x) = \sum_{i=0}^{m-1} b_i x^i$$

stopnia mniejszego niż m w czasie $O(m \log m)$. Wyniki pośrednie w trakcie obliczania FFT mogą nie być liczbami całkowitymi, dlatego w implementacji używamy

typów rzeczywistych. Ponieważ współczynniki wynikowego wielomianu mogą być w pesymistycznym przypadku zbliżone do $m \cdot \max a_i \cdot \max b_i$, musimy użyć typu rzeczywistego, który pozwala dokładnie przechowywać liczby takiego właśnie rzędu. W naszym przypadku m będzie mniejsze niż p, a a_i i b_i nie przekroczą q, więc użyjemy typu rzeczywistego z 63 bitami dokładności (np. `long double` w kompilatorach GCC). Nie będziemy się ponad to wgłębiać w szczegóły działania algorytmu FFT, a raczej potraktujemy go jako *black box* w dalszym opisie rozwiązania. Na zawodach warto mieć ze sobą gotową implementację tego algorytmu.

Przypomnijmy, że wąskim gardłem naszego dotychczasowego algorytmu była obsługa węzłów drzewa wyrażenia odpowiadających dodawaniu i mnożeniu. Zacznijmy od tych pierwszych; mamy dla każdego $k \in \{0, \ldots, p-1\}$ obliczyć wartość wyrażenia

$$t_u[k] = \sum_{i+j \equiv k \pmod p} t_v[i] \cdot t_w[j]. \tag{1}$$

Od razu zauważmy, że obliczenie, które chcemy wykonać, dosyć dokładnie odpowiada właśnie FFT. Faktycznie, wystarczy przemnożyć wielomiany:

$$A(x) = \sum_{i=0}^{p-1} t_v[i]x^i \quad \text{oraz} \quad B(x) = \sum_{i=0}^{p-1} t_w[i]x^i,$$

otrzymując wielomian

$$C(x) = \sum_{i=0}^{2p-2} c[i]x^i.$$

Jeśli dla $i, j \in \{0, \ldots, p-1\}$ mamy $i + j \equiv k \pmod p$, to składnik iloczynu

$$t_v[i]x^i \cdot t_w[j]x^j = t_v[i] \cdot t_w[j] \cdot x^{i+j}$$

wliczy się w wielomianie $C(x)$ do współczynnika stojącego przy x^{i+j} (czyli albo przy x^k, albo przy x^{k+p}). To oznacza, że jako $t_u[k]$ wystarczy przyjąć sumę tych dwóch współczynników:

$$t_u[k] = c[k] + c[k+p].$$

Ostatecznie do obsłużenia węzła z dodawaniem wystarcza tylko jedno wywołanie algorytmu FFT, działające w czasie $O(p \log p)$.

To było rzeczywiście proste. Dużo mniej oczywiste jest, że w bardzo podobny sposób możemy poradzić sobie z węzłami z mnożeniem. Aby to zauważyć, przyda nam się pewien pożyteczny fakt z zakresu teorii liczb.

Lemat 1. Dla każdej liczby pierwszej p istnieje taka dodatnia liczba całkowita g, że każdą dodatnią resztę z dzielenia przez p można jednoznacznie przedstawić w postaci $g^i \bmod p$, dla pewnego $i \in \{0, \ldots, p-2\}$. Liczbę g nazywamy *generatorem modulo p*.

Generator modulo p wykorzystamy do przerobienia sumy obliczanej w węźle z mnożeniem na sumę podobną jak w przypadku węzłów z dodawaniem. Przypomnijmy, że dla każdego $k \in \{0, \ldots, p-1\}$ mieliśmy obliczyć:

$$t_u[k] = \sum_{i \cdot j \equiv k \pmod p} t_v[i] \cdot t_w[j].$$

Załóżmy na początek, że $k > 0$, a zatem także $i, j > 0$. Podstawmy $i \equiv g^{i'}$, $j \equiv g^{j'}$, $k \equiv g^{k'} \pmod{p}$. Nasz wzór przybiera postać:

$$t_u[g^{k'} \bmod p] = \sum_{g^{i'} \cdot g^{j'} \equiv g^{k'} \pmod{p}} t_v[g^{i'} \bmod p] \cdot t_w[g^{j'} \bmod p].$$

Warunek $g^{i'} \cdot g^{j'} \equiv g^{k'} \pmod{p}$, czyli $g^{i'+j'} \equiv g^{k'} \pmod{p}$, na mocy definicji generatora odpowiada po prostu warunkowi $i' + j' \equiv k' \pmod{(p-1)}$. Jeśli zmienimy indeksowanie tablicy t przez $t'_u[x]$ oznaczymy $t_u[g^x \bmod p]$ (i podobnie dla $t'_v[x]$, $t'_w[x]$), to nasz wzór otrzyma ostatecznie postać:

$$t'_u[k'] = \sum_{i'+j' \equiv k' \pmod{(p-1)}} t'_v[i'] \cdot t'_w[j'].$$

Jest to taki sam wzór jak dla węzłów z dodawaniem (1), tylko z $p-1$ zamiast p w operacji modulo. Tak więc wszystkie wartości $t'_u[k']$ możemy obliczyć za pomocą jednego wywołania algorytmu FFT! To daje nam wszystkie wartości postaci $t_u[k]$ poza przypadkiem $k = 0$. Z nim jednak radzimy sobie po prostu w czasie $O(p)$, rozpatrując wszystkie takie pary reszt (i, j), że jedna z nich jest równa zero.

Tak więc, o ile znamy generator g, złożoność czasowa obliczeń dla wszystkich węzłów wyniesie $O(np \log p)$.

Pozostaje nam powiedzieć, w jaki sposób efektywnie znaleźć generator modulo p. Istnieje bardzo szybki algorytm dla tego problemu, wymagający dalszych spostrzeżeń z zakresu teorii liczb, jednak w naszym przypadku możemy sobie z tym problemem poradzić zupełnie siłowo. Wystarczy przeglądać kolejne liczby $2, 3, \ldots$, sprawdzając dla każdej z nich, czy jest ona generatorem modulo p. Robimy to, podnosząc takiego kandydata na generator do kolejnych potęg modulo p i sprawdzając za pomocą prostej tablicy zliczającej, czy żadna z nich nie powtórzy się przed osiągnięciem wykładnika $p-1$. Algorytm ten pesymistycznie działa w czasie $O(p^2)$, ale w praktyce wypada dużo lepiej, jeśli przerywamy go w chwili znalezienia pierwszego generatora.

F

Formuła 1

★★★★

Autor zadania: Tomasz Idziaszek
Opis rozwiązania: Tomasz Idziaszek
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2012/for`

Mały Robert Kubita bardzo lubi oglądać wyścigi Formuły 1, które w Bajtocji odbywają się na torze prowadzącym z Bajtogrodu do Bitowic. Najbardziej ekscytującymi dla Roberta momentami wyścigu są manewry wyprzedzania, dlatego chłopiec chciałby ich widzieć jak najwięcej.

Marzy mu się zobaczyć wyścig, który spełniłby następujące założenia: ścigałoby się w nim n bolidów, a dla każdego i ($1 \leqslant i \leqslant n$) bolid, który startował z i-tej pozycji, wykonałby podczas wyścigu a_i manewrów wyprzedzania. Zakładamy, że w każdej chwili wyścigu odbywa się co najwyżej jeden manewr wyprzedzania, który polega na tym, że pewien bolid przesuwa się przed bolid bezpośrednio poprzedzający go.

Robert zastanawia się, czy taki wyścig jest w ogóle możliwy. Poprosił Cię o pomoc w rozstrzygnięciu tej kwestii.

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita t, oznaczająca liczbę zestawów testowych opisanych w dalszej części wejścia.

Opis każdego zestawu składa się z dwóch wierszy. W pierwszym z nich znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 1\,000\,000$), oznaczająca liczbę bolidów biorących udział w wyścigu. W drugim wierszu znajduje się ciąg n liczb całkowitych $a_1, a_2, \ldots, a_n$ ($0 \leqslant a_i \leqslant 10^9$), który zadaje, ile manewrów wyprzedzania muszą wykonać poszczególne bolidy.

Rozmiar żadnego pliku wejściowego nie przekracza 20 MB.

Wyjście

Twój program powinien wypisać t wierszy z odpowiedziami dla kolejnych zestawów testowych. Odpowiedzią dla zestawu jest słowo `TAK` albo `NIE` w zależności od tego, czy da się zrealizować wyścig zgodnie z wytycznymi Roberta.

Przykład

Dla danych wejściowych:

```
3
2
0 1
3
0 1 4
3
1 1 3
```

poprawnym wynikiem jest:

```
TAK
NIE
TAK
```

Rozwiązanie

Ustalmy bolid o numerze k i zastanówmy się, jaka jest maksymalna liczba manewrów wyprzedzania, które może wykonać ten bolid, pod warunkiem, że pozostałe bolidy wykonają liczbę manewrów zadaną przez ciąg dany na wejściu. Oznaczmy tę wartość przez s_k.

Na początek wyznaczmy ją dla ostatniego bolidu: n-ty bolid może wyprzedzić kolejno wszystkie pozostałe, co umożliwi mu wykonanie $n-1$ manewrów. Ponadto po wyprzedzeniu i-tego bolidu może nastąpić a_i par manewrów wyprzedzania (i-ty wyprzedza n-tego, n-ty wyprzedza i-tego), co skutkuje dodatkowymi a_i manewrami wykonanymi przez n-ty bolid oraz wyzerowaniem liczby manewrów, które musiał wykonać i-ty bolid. Łatwo zauważyć, że jest to najlepsza strategia działania, zatem

$$s_n = \sum_{i=1}^{n-1} (a_i + 1).$$

Bolid o numerze $k \neq n$ może w analogiczny sposób wyprzedzać bolidy startujące przed nim, ale by mógł wyprzedzać bolidy startujące za nim, musi poczekać, aż one wyprzedzą go najpierw, co nie zawsze jest możliwe. Wyznaczymy zatem $s_k = A_k + B_k$, gdzie wartość A_k zależy tylko od wartości a_i dla bolidów przed bolidem k, zaś B_k zależy jedynie od a_i dla $i > k$. Wartości A_k wyliczyć możemy z prostego wzoru:

$$A_k = \sum_{i=1}^{k-1} (a_i + 1).$$

Zajmijmy się teraz B_k. Oznaczmy przez $b_{k,i}$ minimalną liczbę manewrów wyprzedzania, które musi wykonać i-ty bolid ($i > k$), aby znaleźć się tuż za bolidem k-tym, pod warunkiem, że bolidy pomiędzy nimi wyprzedzą bolid k-ty, jeśli tylko mogą. Mamy:

$$b_{k,k+1} = 0, \qquad b_{k,i+1} = b_{k,i} + [a_i - b_{k,i} \leqslant 0] \quad \text{dla } i > k.$$

W powyższym wzorze $[e]$ oznacza nawias Iversona, tzn. $[e] = 1$, jeśli warunek e jest spełniony, a w przeciwnym wypadku $[e] = 0$. Wyjaśnienie wzoru jest następujące. Bolid o numerze $k+1$ od razu znajduje się tuż za bolidem k-tym. Natomiast dla $i > k$ bolid o numerze $i+1$ musi wykonać tyle samo manewrów co bolid i-ty plus dodatkowo musi wyprzedzić bolid i-ty, jeśli ten ma za małe a_i, żeby wyprzedzić bolid k-ty.

Każdy z bolidów, któremu uda się znaleźć tuż za bolidem k-tym, będzie mógł zużyć pozostałe mu manewry na naprzemienne wyprzedzanie się z bolidem k-tym, tak więc:

$$B_k = \sum_{i=k+1}^{n} \max(0, a_i - b_{k,i}).$$

Powyższe wzory pozwalają nam wyznaczyć jedną wartość s_k w czasie $O(n)$.

Algorytm

Oczywiste jest, że aby rozwiązanie istniało, liczba manewrów wyprzedzania, które jest zobowiązany wykonać bolid, musi być nie większa niż maksymalna liczba manewrów, które jest w stanie wykonać w najlepszym przypadku. Innymi słowy, musi być spełniony warunek

$$s_k \geqslant a_k \qquad \text{dla każdego } 1 \leqslant k \leqslant n. \tag{1}$$

Dużo mniej jasne (a na pewno dużo trudniejsze do udowodnienia) jest, że powyższy warunek jest wystarczający. Stosowny dowód przedstawimy w dalszej części tekstu. Na razie zastanówmy się, jak szybko możemy sprawdzić prawdziwość tego warunku.

Bezpośrednie użycie podanych powyżej wzorów daje nam algorytm działający w czasie $O(n^2)$. Nie jest trudno przyspieszyć go tak, aby obliczał wszystkie wartości A_k w sumarycznym czasie $O(n)$. Uważna analiza wzorów pozwala również skonstruować dość skomplikowane drzewo przedziałowe, dzięki któremu można wyznaczyć wszystkie wartości B_k w sumarycznym czasie $O(n \log n)$. Nie będziemy jednak opisywać tego rozwiązania, gdyż istnieje znacznie prostsze.

Dzięki obliczeniu w czasie $O(n)$ wszystkich wartości A_k, łatwo możemy sprawdzić, czy może przypadkiem dla niektórych bolidów jest spełnione $A_k \geqslant a_k$, z czego wynikałoby spełnienie warunku $s_k \geqslant a_k$ bez konieczności obliczania B_k. Problematyczne są więc jedynie bolidy, dla których $A_k < a_k$. Ponieważ jednak takie bolidy muszą mieć wartość a_k większą od sumy wartości wszystkich poprzedzających je bolidów, wartości a_k dla problematycznych bolidów rosną wykładniczo. Innymi słowy, problematycznych bolidów (czyli tych, dla których musimy wyznaczać wartości B_k) jest raptem $O(\log(\max_i a_i))$. Ta obserwacja prowadzi nas do nietrudnego rozwiązania liniowo-logarytmicznego.

Możemy jednak łatwo otrzymać jeszcze szybszy algorytm. Rozważmy ostatni bolid, dla którego $a_m \geqslant A_m$. Taki bolid albo „zużywa" wszystkie wyprzedzenia bolidów startujących przed nim, albo musi go wyprzedzić jakiś bolid startujący za nim. Nazwiemy ten bolid *krytycznym*. Okazuje się, że warunek (1) może być niespełniony jedynie dla bolidu krytycznego. W efekcie cały warunek (1) jest równoważny jednej nierówności

$$s_m \geqslant a_m. \tag{2}$$

Pokażemy teraz, że dla wszystkich $k \neq m$ mamy spełnione $s_k \geqslant a_k$. Jeśli $k > m$, to z definicji bolidu krytycznego mamy $a_k < A_k$, zatem

$$s_k = A_k + B_k \geqslant A_k > a_k.$$

Jeśli zaś $k < m$, to $a_m \geqslant A_m \geqslant a_k + m - 1$, a ponieważ zawsze $b_{k,m} \leqslant m - 2$, dostajemy

$$s_k \geqslant B_k \geqslant \max(0, a_m - b_{k,m}) \geqslant \max(0, (a_k + m - 1) - (m - 2)) > a_k. \tag{3}$$

Wyznaczenie wartości m oraz sprawdzenie warunku (2) jesteśmy w stanie łatwo zrobić w czasie $O(n)$ i taka też jest złożoność czasowa całego algorytmu.

Dowód poprawności algorytmu

Wykażemy teraz, że spełnienie warunku (1) jest wystarczające do istnienia wyścigu określonego warunkami zadania. Dowód będzie konstruktywny, tzn. będzie

polegał na wskazaniu kolejności, w jakiej bolidy powinny wykonywać manewry wyprzedzania.

Zasada konstrukcji wyścigu będzie bardzo prosta: za każdym razem wyprzedzamy pierwszym bolidem, który jeszcze może wyprzedzać i który znajduje się za bolidem krytycznym, chyba, że takiego bolidu nie ma — wtedy wyprzedzamy bolidem krytycznym. Zauważmy przy tym, że to, który bolid jest krytyczny, może zmieniać się w czasie wyścigu.

Dla ustalonego ciągu $a_1, \ldots, a_n$ spełniającego warunek (1) udowodnimy, że wykonanie manewru zgodnie z powyższą zasadą będzie prowadzić do nowego ciągu $a'_1, \ldots, a'_n$, również spełniającego ten warunek. Maksymalne liczby manewrów dla nowego ciągu oznaczymy przez s'_k.

Przypadek 1

Na początek rozważmy przypadek, gdy żaden bolid startujący za bolidem krytycznym nie może już wyprzedzać (włączając w to sytuację, gdy bolid krytyczny jest ostatni), czyli $a_i = 0$ dla wszystkich $i > m$, a zatem $B_m = 0$. Ponadto dla bolidu krytycznego mamy $a_m \geqslant A_m$, więc

$$A_m = s_m - B_m = s_m \geqslant a_m \geqslant A_m.$$

Z tego wynika, że $a_m = A_m$, zatem bolid krytyczny może zastosować strategię wyprzedzania opisaną na początku tekstu (z i-tym bolidem wykonując ciąg a_i naprzemiennych manewrów wyprzedzania) i w ten sposób uda się zrealizować cały wyścig zgodnie z wymaganiami. W szczególności wykonanie przez niego pierwszego manewru prowadzi do sytuacji, w której istnieje sposób przeprowadzenia dalszego wyścigu, co oznacza, że w nowej sytuacji warunek (1), będący warunkiem koniecznym istnienia takiego wyścigu, jest spełniony.

Przypadek 2

Załóżmy teraz, że za bolidem krytycznym startują bolidy, które mogą wyprzedzać, i niech pierwszy z nich ma numer j (innymi słowy $j > m$ jest najmniejszym indeksem, takim że $a_j > 0$). Wykażemy, że poprawnym manewrem jest wyprzedzenie bolidem j-tym, zatem po wykonaniu tego manewru warunek (1) jest nadal spełniony dla nowego ciągu, tzn. $s'_k \geqslant a'_k$ dla wszystkich k. Warto podkreślić, że nie możemy ograniczyć się jedynie do sprawdzenia warunku (2), gdyż nie wiemy, który z bolidów będzie krytyczny po wykonaniu manewru.

Po wykonaniu wyprzedzania mamy $a'_i = a_i$ dla wszystkich $i \notin \{j-1, j\}$ oraz $a'_{j-1} = a_j - 1$, $a'_j = a_{j-1}$. Ustalmy $i > j$. Wtedy $A'_i = A_i - 1$, a ponieważ z definicji bolidu krytycznego mamy $A_i > a_i$, otrzymujemy

$$s'_i = A'_i + B'_i \geqslant A'_i = A_i - 1 \geqslant a_i = a'_i. \tag{4}$$

Stąd po wykonaniu manewru warunek (1) jest nadal spełniony dla wszystkich bolidów startujących za bolidem j-tym.

Rozważmy teraz dwa przypadki w zależności od tego, czy wykonując manewr bolidem j-tym, wyprzedzamy bolid krytyczny. Ta część dowodu będzie nieco techniczna.

Przypadek 2A

Jeśli bolid j-ty nie wyprzedza bolidu krytycznego, to sytuacja na torze wygląda tak:

$$a_1 \ \ldots \ a_m \ \underbrace{0\,0\,0}_{j-m-1} \ a_j \ a_{j+1} \ \ldots \ a_n \quad \to \quad a_1 \ \ldots \ a_m \ \underbrace{0\,0}_{j-m-2} \ (a_j - 1) \ 0 \ a_{j+1} \ \ldots \ a_n$$

Ponieważ z definicji bolidu krytycznego mamy $A_j > a_j$, po wykonaniu manewru warunek (1) jest spełniony dla indeksu $j-1$:

$$s'_{j-1} = A'_{j-1} + B'_{j-1} \geqslant A'_{j-1} = A_{j-1} = A_j - 1 > a_j - 1 = a'_{j-1}.$$

Ponadto z prawdziwości (4) oraz tego, że ten sam warunek jest w oczywisty sposób spełniony dla wszystkich $i > m$, dla których $a_i = 0$, dostajemy, że jest spełniony dla wszystkich $i > m$.

Przyglądając się definicji B_i, widzimy, że dla $i \leqslant m$ mamy $B'_i = B_i$ (bolid j-ty i tak musi wyprzedzić wszystkie bolidy pomiędzy nim a bolidem krytycznym, a dla bolidów startujących za nim ten manewr nic nie zmienia). Oczywiście $A'_i = A_i$, zatem warunek (1) jest spełniony również dla $i \leqslant m$:

$$s'_i = A'_i + B'_i = A_i + B_i = s_i \geqslant a_i = a'_i.$$

Przypadek 2B

Jeśli bolid j-ty wyprzedza bolid krytyczny, to mamy $j = m+1$, a sytuacja na torze przestawia się następująco:

$$a_1 \ \ldots \ a_{m-1} \ a_m \ a_{m+1} \ a_{m+2} \ \ldots \ a_n \quad \to \quad a_1 \ \ldots \ a_{m-1} \ (a_{m+1}-1) \ a_m \ a_{m+2} \ \ldots \ a_n$$

Pokazując nierówność (4), udowodniliśmy, że warunek (1) jest spełniony dla $i > m+1$. Dla $i < m$ mamy

$$a'_{m+1} = a_m \geqslant A_m \geqslant a_i + m - 1 = a'_i + m - 1,$$

więc, korzystając z pomysłu jak w nierówności (3), dostajemy $s'_i \geqslant a'_i$.

Z definicji bolidu krytycznego otrzymujemy, że $a_{m+1} < A_{m+1} = A_m + (a_m + 1)$, zatem dla indeksu m dostajemy

$$s'_m \geqslant A'_m + a'_{m+1} = A_m + a_m \geqslant a_{m+1} > a_{m+1} - 1 = a'_m.$$

Ponieważ dla $i > m+1$ zachodzi $a'_i = a_i$, mamy $B'_{m+1} = B_{m+1}$. Ponadto skoro $b_{m,m+1} = 0$ i $a_{m+1} > 0$, używając definicji otrzymujemy $b_{m,m+2} = 0 = b_{m+1,m+2}$. Zatem $B_m = a_{m+1} + B_{m+1}$. Stąd dla indeksu $m+1$ zachodzi

$$s'_{m+1} = A'_m + (a'_m + 1) + B'_{m+1} = A_m + a_{m+1} + B_{m+1} = A_m + B_m = s_m \geqslant a_m = a'_{m+1},$$

co kończy dowód.

Posłowie

Zadanie *Formuła 1*, pomimo bardzo prostej treści, sprawiło dużo kłopotu zespołom biorącym udział w zawodach, co po lekturze powyższego rozwiązania nie powinno nikogo dziwić. Zauważmy jednak, że aby napisać działający program, nie trzeba było przeprowadzać żmudnego dowodu — wystarczyła pewna doza algorytmicznej intuicji, aby najpierw sformułować konieczny warunek (1), a następnie uwierzyć, że jest on wystarczający.

Taka algorytmiczna intuicja pojawia się wraz z doświadczeniem, które zdobywamy, rozwiązując różnorodne zadania. W szczególności w rozwiązaniu zadania *Formuła 1* można odnaleźć podobieństwa do twierdzenia Erdősa–Gallaiego, które podaje konieczny i wystarczający warunek na istnienie grafu o zadanym ciągu stopni wierzchołków. O tym twierdzeniu można przeczytać w artykule *Odtwarzanie grafu* w numerze 11/2011 miesięcznika *Delta*.

Generał kontra gawiedź

Autor zadania: Tomasz Idziaszek
Opis rozwiązania: Tomasz Idziaszek
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2012/gen`

W Bajtogrodzie niesamowite poruszenie! Bajtoccy archeolodzy odkryli w pobliżu miasta szczątki dinozaurów. Gdy tylko gawiedź się o tym zwiedziała, ten i ów jął wybierać się na wykopalisko po to, by zwędzić mniejszą lub większą kość. Proceder ten stał się tak nagminny, że postanowiono do ochrony terenu wykopalisk zatrudnić wojsko.

Generał Bajtazar rozlokował n żołnierzy w *strategicznych punktach* na terenie wykopalisk. (Żołnierze nie mogą stanąć gdzie bądź, aby nie przeszkadzać archeologom. Poza tym muszą mieć też dobrą widoczność, aby chronić teren wykopalisk.) Powiemy, że dany punkt terenu jest *chroniony*, jeśli ruszając się z niego w jakimkolwiek kierunku, zbliżymy się do któregoś z żołnierzy (tj. nasza odległość od tego żołnierza zmaleje).

Bajtazarowi przydzielono właśnie nowego rekruta. Generał może go umieścić w jednym z m jeszcze nieobsadzonych punktów strategicznych. Dla każdego z tych punktów interesuje go, jaka będzie powierzchnia chronionego terenu, gdy nowy żołnierz stanie w tym punkcie.

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite n i m ($3 \leqslant n \leqslant 100\,000$, $1 \leqslant m \leqslant 100\,000$) oznaczające liczbę rozstawionych żołnierzy i liczbę nieobsadzonych punktów strategicznych. Następne n wierszy opisuje żołnierzy: w i-tym z tych wierszy znajdują się liczby całkowite x_i, y_i ($-10^8 \leqslant x_i, y_i \leqslant 10^8$), które oznaczają współrzędne punktu (w prostokątnym układzie współrzędnych), w którym stoi i-ty żołnierz. W kolejnych m wierszach zapisane są w tym samym formacie kolejne nieobsadzone punkty strategiczne. Punkty podane na wejściu nie powtarzają się.

Można założyć, że powierzchnia terenu chronionego przez już rozstawionych żołnierzy jest dodatnia.

Wyjście

Na wyjście należy wypisać dokładnie m wierszy. W i-tym wierszu należy wypisać całkowitą powierzchnię chronionego terenu, jeśli nowy rekrut stanie w i-tym nieobsadzonym punkcie strategicznym. Liczby należy wypisywać z dokładnie jedną cyfrą po kropce dziesiętnej.

Przykład

Dla danych wejściowych:

```
3 2
0 0
2 -1
1 2
3 1
1 0
```

poprawnym wynikiem jest:

```
5.0
2.5
```

Rozwiązanie

Na początek warto zastanowić się, jak dla danego rozstawienia żołnierzy wygląda chroniony przez nich obszar. Jeśli rozważymy dowolnych trzech żołnierzy stojących w punktach p_i, p_j i p_k, to każdy punkt wewnątrz trójkąta $p_ip_jp_k$ należy do chronionego obszaru (ruszając się z dowolnego punktu leżącego wewnątrz trójkąta, zbliżamy się do co najmniej jednego z jego wierzchołków). A zatem każdy punkt należący do wnętrza otoczki wypukłej punktów, w których stoją żołnierze, jest chroniony. Co więcej, są to wszystkie chronione punkty: dowolny punkt leżący na zewnątrz otoczki można oddzielić od otoczki prostą. Ruszając się z tego punktu tak, by oddalić się od prostej, jednocześnie oddalamy się od wszystkich punktów otoczki (a więc również od każdego punktu obsadzonego żołnierzem). Dla pełności dodajmy, że punkty na brzegu otoczki nie są chronione, choć to akurat ma niewielkie znaczenie, bo brzeg otoczki ma zerowe pole.

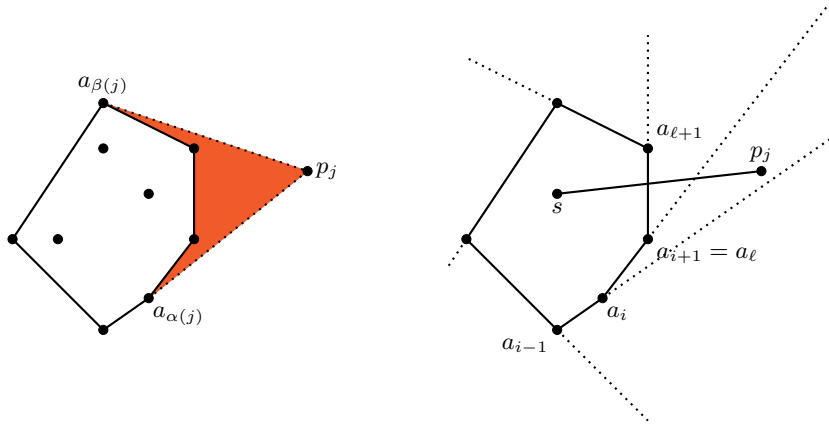
Tak więc chroniony obszar jest wnętrzem otoczki wypukłej punktów, w których stoją żołnierze. Zadanie możemy teraz sformułować następująco: dla każdego z m nieobsadzonych punktów $p_i = (x_i, y_i)$ należy wyznaczyć pole otoczki wypukłej n obsadzonych punktów i punktu p_i. Ponieważ otoczka wypukła jest wielokątem, do wyznaczenia jej pola możemy wykorzystać jeden z algorytmów liczenia pola wielokąta.

Łatwo podać algorytm rozwiązujący to zadanie, rozpatrujący każdy z nieobsadzonych punktów niezależnie. W tym celu m razy wykonujemy algorytm wyznaczania otoczki wypukłej w czasie $O(n \log n)$ oraz liczenia pola wielokąta w czasie $O(n)$. Takie rozwiązanie działa w niezbyt porywającej złożoności czasowej $O(mn \log n)$.

Dodajmy na marginesie, że możemy nieznacznie przyspieszyć je do złożoności $O(n \log n + mn)$, jeśli zawczasu posortujemy wszystkie obsadzone punkty, gdyż otoczkę wypukłą przy posortowanych danych możemy wyznaczyć w czasie $O(n)$.

Szybsze rozwiązanie

Na początek obliczmy chroniony obszar, czyli otoczkę wypukłą n obsadzonych punktów. Zajmie nam to czas $O(n \log n)$. Oznaczmy powstały wielokąt przez W, a jego k wierzchołków przez $a_0, a_1, \ldots, a_{k-1}$. Przyjmujemy przy tym, że $a_k = a_0$



Rysunek 1. Po lewej: wielokąt W oraz punkty krytyczne dla nieobsadzonego punktu p_j. Po prawej: punkt p_j znajduje się w obszarze A_i.

oraz $a_{-1} = a_{k-1}$. Zakładamy, że wierzchołki wielokąta W są ponumerowane w kierunku przeciwnym do ruchu wskazówek zegara. Aby obliczyć dwukrotność pola tego wielokąta, możemy skorzystać ze wzoru

$$\sum_{i=0}^{k-1} a_i \times a_{i+1}, \tag{1}$$

czyli zsumować k iloczynów wektorowych, z których i-ty jest wyznaczony na podstawie końców i-tego boku wielokąta. Przypomnijmy, że jeśli $a_i = (x_i, y_i)$ oraz $a_j = (x_j, y_j)$ to iloczyn wektorowy $a_i \times a_j$ jest równy $x_i y_j - y_i x_j$.

Ustalmy teraz pewien nieobsadzony punkt p_j i rozważmy najmniejszy kąt o wierzchołku w tym punkcie, który zawiera chroniony obszar (patrz rysunek 1). Rozważmy dwa punkty obszaru zawarte w ramionach tego kąta (jeśli ramię kąta zawiera więcej niż jeden punkt obszaru, to wybieramy ten najbliższy punktowi p_j). Te punkty muszą być wierzchołkami wielokąta W; oznaczmy ich indeksy przez $\alpha(j)$ i $\beta(j)$.

Obszar chroniony po ustawieniu nowego żołnierza w punkcie p_j jest wnętrzem wielokąta (nazwijmy go W_j), którego wierzchołki to punkty p_j, $a_{\alpha(j)}$ i $a_{\beta(j)}$ oraz wierzchołki na brzegu wielokąta W leżące pomiędzy $a_{\alpha(j)}$ i $a_{\beta(j)}$. Oczywiście wybieramy tu tę część brzegu W, która leży dalej od p_j. Innymi słowy, wielokąt W_j zawiera boki $a_{\alpha(j)}p_j$, $p_j a_{\beta(j)}$ oraz spójny fragment ciągu boków wielokąta W. Widać zatem, że będziemy chcieli efektywnie obliczać wkład do pola wielokąta W_j pochodzący z pewnej liczby kolejnych boków wielokąta W. W tym celu przyda nam się obliczenie sum częściowych dla wzoru (1):

$$d[0] = 0, \qquad d[i+1] = d[i] + a_i \times a_{i+1} \quad \text{dla } i = 0, \ldots, k-1.$$

Przy tych oznaczeniach dwukrotność pola wielokąta W_j możemy obliczyć w czasie stałym ze wzoru

$$a_{\alpha(j)} \times p_j \;+\; p_j \times a_{\beta(j)} \;+\; \begin{cases} d[k] - d[\beta(j)] + d[\alpha(j)] & \text{dla } \alpha(j) < \beta(j), \\ d[\alpha(j)] - d[\beta(j)] & \text{dla } \alpha(j) > \beta(j). \end{cases}$$

Zamiatanie

Pozostaje jedynie pokazać, jak szybko wyznaczyć indeksy $\alpha(j)$ i $\beta(j)$. Przedłużmy każdy bok $a_i a_{i+1}$ wielokąta W do półprostej o początku w a_i. Półproste te dzielą nam zewnętrze wielokąta W na rozłączne obszary, będące kątami o wierzchołkach w punktach a_i (patrz rysunek 1). Przez półprostą o wierzchołku a_i będziemy rozumieli półprostą zawartą w półprostej $a_{i-1}a_i$, ale zaczynającą się w punkcie a_i.

Oznaczmy przez A_i obszar składający się z półprostej o wierzchołku a_i oraz *wnętrza* kąta, którego ramiona to półprosta o wierzchołku a_i oraz półprosta $a_i a_{i+1}$. Zauważmy, że przy tych oznaczeniach punkt p_j należy do obszaru $A_{\alpha(j)}$.

Skorzystamy z tej obserwacji, aby szybko wyznaczyć indeksy $\alpha(j)$ oraz $\beta(j)$. Zrobimy to jednocześnie dla wszystkich nieobsadzonych punktów. W tym celu wszystkie punkty wielokąta W i punkty nieobsadzone zamiatamy kątowo względem dowolnego punktu s wewnątrz wielokąta W. Czyli półprostą zaczepioną w punkcie s (zwaną *miotłą*) obracamy przeciwnie do ruchu wskazówek zegara, zatrzymując ją w momentach (zwanych *zdarzeniami*), gdy znajduje się na niej wierzchołek wielokąta W lub nieobsadzony punkt. W każdym momencie utrzymujemy bok wielokąta $a_\ell a_{\ell+1}$ aktualnie przecinany przez miotłę. W sytuacji, gdy kilka zdarzeń ma miejsce jednocześnie, tj. punkt s, wierzchołek wielokąta W oraz pewna liczba nieobsadzonych punktów leżą na tej samej prostej, wygodnie nam będzie najpierw rozpatrzeć zdarzenie odpowiadające wierzchołkowi wielokąta, a następnie pozostałe zdarzenia (w dowolnej kolejności).

Rozpatrując zdarzenie odpowiadające wierzchołkowi a_i wielokąta, uaktualniamy przecinany bok, przyjmując $\ell = i$. Z kolei podczas zdarzenia odpowiadającego nieobsadzonemu punktowi p_j będziemy wyznaczać wartość $\alpha(j)$ jak następuje. Niech $a_\ell a_{\ell+1}$ będzie bokiem wielokąta aktualnie przecinanym przez miotłę. Zatem bok ten przecina również wnętrze odcinka sp_j. Tak więc wiemy, że półproste o wierzchołkach $a_{\ell+1}, a_{\ell+2}, \ldots, a_{\alpha(j)}$ nie przecinają wnętrza odcinka sp_j, natomiast pozostałe półproste o wierzchołkach $a_{\alpha(j)+1}, \ldots, a_\ell$ przecinają je (zapis $a_x, \ldots, a_y$ dla $x > y$ należy rozumieć jako $a_x, \ldots, a_{k-1}, a_0, \ldots, a_y$). Tak więc wartość $\alpha(j)$ możemy wyznaczyć wyszukiwaniem binarnym — wystarczy jedynie umieć sprawdzać, czy półprosta o ustalonym wierzchołku przecina wnętrze odcinka sp_j.

Całe zamiatanie działa w czasie $O((n+m)\log(n+m))$: na początek sortujemy kątowo $n+m$ punktów, a następnie dla każdego z m nieobsadzonych punktów wykonujemy jeszcze $O(\log n)$ operacji. Symetrycznym algorytmem wyznaczamy wszystkie indeksy $\beta(j)$.

Ostatecznie uzyskujemy algorytm o złożoności czasowej $O((n+m)\log(n+m))$.

Hydra

Autor zadania: Tomasz Idziaszek
Opis rozwiązania: Tomasz Idziaszek
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2012/hyd`

Bituś dostał na urodziny grę komputerową o nazwie *Niesamowite przygody Rycerza Bajtazara*. Zabawa polega na kierowaniu postacią tytułowego rycerza, który przemierza królestwo Bajtocji i pomaga uciśnionym przez poczwary, poborców podatkowych i gradobicia. Bituś przeszedł już prawie całą grę, ale utknął na ostatnim poziomie, w którym Bajtazar walczy z wielkim morskim wężem — Bajtocką Hydrą.

Do walki z potworem Bajtazar używa swojego miecza. W grze dostępne są dwa rodzaje ciosów: rycerz może albo uciąć głowę węża, albo (co oczywiście wymaga więcej wysiłku) zmasakrować tę głowę. Jednakowoż ucięcie głowy węża, choć prostsze, powoduje, że w miejscu odcięcia z szyi węża odrastają nowe głowy. Wodny potwór zostanie pokonany dopiero wtedy, gdy Bajtazar pozbawi go wszystkich głów i żadna głowa nie będzie już mogła odrosnąć.

Bajtocka Hydra może mieć n rodzajów głów, które będziemy oznaczali liczbami od 1 do n. Na samym początku wąż ma jedną głowę rodzaju 1. Głowa rodzaju i (dla $1 \leqslant i \leqslant n$) charakteryzuje się następującymi cechami: liczbą machnięć miecza u_i, które musi wykonać Bajtazar, aby uciąć tę głowę, liczbą machnięć miecza z_i, która jest wymagana do zmasakrowania tej głowy, oraz listą r_i rodzajów głów $g_{i,1}, \ldots, g_{i,r_i}$, które odrastają na miejsce głowy rodzaju i po jej ucięciu.

Podpowiedz Bitusiowi, ile minimalnie machnięć mieczem należy wykonać, aby pokonać Hydrę.

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 200\,000$), oznaczająca liczbę rodzajów głów Hydry. Kolejne n wierszy opisuje poszczególne rodzaje głów; w i-tym z tych wierszy opisana jest głowa rodzaju i. Wiersz ten zaczyna się trzema liczbami całkowitymi u_i, z_i, r_i ($1 \leqslant u_i < z_i \leqslant 10^9$, $1 \leqslant r_i$), po których następują liczby całkowite $g_{i,1}, \ldots, g_{i,r_i}$ ($1 \leqslant g_{i,j} \leqslant n$). Suma liczb r_i nie przekracza 1 000 000.

Wyjście

W jedynym wierszu wyjścia należy wypisać minimalną liczbę machnięć mieczem, która jest potrzebna, by wygrać grę.

Przykład

Dla danych wejściowych:

```
4
4 27 3 2 3 2
3 5 1 2
1 13 2 4 2
5 6 1 2
```

poprawnym wynikiem jest:

```
26
```

Rozwiązanie

W rozwiązaniu zadania obliczymy trochę więcej, niż nas o to proszą, a mianowicie dla każdego rodzaju głowy i wyznaczymy minimalną liczbę machnięć mieczem potrzebnych do uśmiercenia hydry, której jedyna głowa ma rodzaj i. Oznaczmy tę wartość przez $koszt[i]$. Jeśli chcemy unicestwić hydrę z głową rodzaju i, mamy dwa wyjścia: albo możemy tę głowę zmasakrować za pomocą z_i machnięć mieczem, albo możemy ją uciąć u_i machnięciami, a następnie niezależnie uśmiercić wszystkie głowy, które wyrosną na jej miejscu (oznaczmy zbiór tych głów przez G_i). To prowadzi nas do następującej rekurencji:

$$koszt[i] = \min(z_i,\ u_i + \sum_{j \in G_i} koszt[j]). \tag{1}$$

Rozwiązaniem zadania jest oczywiście $koszt[1]$. Pozostaje pytanie: w jakiej kolejności obliczać wartości $koszt[i]$? Wszystkie zbiory G_i są niepuste, więc prosty algorytm rekurencyjny działający zgodnie z wzorem (1) z pewnością się zapętli.

Istnieje jednak taki rodzaj głowy i, dla którego wyznaczenie $koszt[i]$ nie wymaga skorzystania ze wzoru (1) — jest to ten o najmniejszej wartości z_i. Istotnie: uśmiercenie tego rodzaju głowy wymaga zmasakrowania co najmniej jednej z głów (samo ucinanie nigdy nie wystarcza), a skoro $z_j \geqslant z_i$ dla dowolnego rodzaju głowy j, to każda metoda postępowania kosztuje co najmniej z_i machnięć mieczem. Najprościej zatem od razu zmasakrować głowę rodzaju i, co pozwala nam przyjąć $koszt[i] = z_i$.

Skonstruujemy algorytm będący uogólnieniem powyższej obserwacji. Algorytm będzie składał się z kolejnych faz. Przez C oznaczmy zbiór tych rodzajów głów i, dla których obliczyliśmy już $koszt[i]$ (na początku zbiór ten jest pusty). Przed pierwszą fazą i po każdej fazie utrzymujemy niezmiennik

$$\text{jeśli } G_i \subseteq C,\ \text{to } i \in C,$$

który mówi, że jeśli pewna wartość $koszt[i]$ nie jest wyznaczona ($i \notin C$), to tylko dlatego, że nie znamy wartości $koszt[j]$ dla co najmniej jednego rodzaju głowy $j \in G_i$ (istnieje takie $j \in G_i$, że $j \notin C$).

Na początku fazy znajdujemy rodzaj głowy $i \notin C$ o najmniejszej wartości z_i. Wykażemy, że podobnie jak poprzednio najbardziej opłaca nam się od razu zmasakrować tę głowę. Faktycznie, zauważmy, że jeśli zdecydujemy się uciąć głowę i,

to wśród głów, które wyrosną na jej miejscu, będzie pewna głowa $j_1 \notin C$ (co wynika z tego, że spełniony jest niezmiennik). Analogicznie, jeśli utniemy głowę j_1, to wśród nowych głów będzie pewna głowa $j_2 \notin C$ itd. W ten sposób otrzymujemy nieskończony ciąg rodzajów głów $i, j_1, j_2, \ldots$, z których żaden nie należy do C. Aby uśmiercić głowę rodzaju i, musimy w pewnym momencie zmasakrować jedną z głów w tym ciągu, co kosztuje nas co najmniej z_i, ponieważ jest to minimalna liczba machnięć mieczem do zmasakrowania głowy spoza zbioru C. Zatem rzeczywiście najtaniej jest od razu zmasakrować głowę i. W tej sytuacji przyjmujemy $koszt[i] = z_i$ i dodajemy i do zbioru C.

W drugiej części fazy przywracamy niezmiennik, tzn. dopóki możemy znaleźć taki rodzaj głowy i, że $G_i \subseteq C$ oraz $i \notin C$, to obliczamy $koszt[i]$ ze wzoru (1) i dodajemy i do C.

Aby szybko znajdować rodzaj głowy i w pierwszej części fazy, możemy na początku algorytmu posortować rodzaje głów niemalejąco po wartościach z_i w czasie $O(n \log n)$, a następnie przeglądać je w takiej kolejności, pomijając rodzaje głów, dla których w międzyczasie została wyznaczona wartość $koszt[i]$.

Efektywną implementację drugiej części uzyskamy, konstruując na początku programu graf skierowany o n wierzchołkach (odpowiadających rodzajom głów), w którym krawędź (j, i) istnieje wtedy i tylko wtedy, gdy $j \in G_i$. Krawędź wchodząca do wierzchołka i oznacza, że do wyznaczenia wartości $koszt[i]$ nie możemy użyć wzoru (1), gdyż nie znamy wartości $koszt[j]$ dla pewnego $j \in G_i$. Z tego grafu będziemy usuwać wierzchołki odpowiadające rodzajom głów dodawanych do zbioru C oraz wychodzące z nich krawędzie. W miarę usuwania krawędzi utrzymujemy stopień wejściowy każdego wierzchołka, czyli liczbę krawędzi do niego wchodzących. Dla każdego wierzchołka i, którego stopień wejściowy spadł do 0, wykonujemy aktualizację wartości $koszt[i]$ (czyli dodajemy i do C), a następnie rekurencyjnie usuwamy go z grafu*.

Ostatecznie złożoność czasowa rozwiązania to $O(n \log n + \sum_i r_i)$, a pamięciowa $O(n + \sum_i r_i)$, gdzie r_i to rozmiar zbioru G_i.

*Opisany algorytm w swej naturze przypomina jedną z metod sortowania topologicznego acyklicznego grafu skierowanego (mimo że rozważany graf bynajmniej nie jest acykliczny).

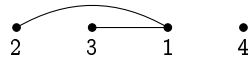
I

Inwersje

Autor zadania: Krzysztof Diks
Opis rozwiązania: Jakub Łącki
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2012/inw`

Bajtazar odkrył nową rodzinę grafów nieskierowanych, które można reprezentować za pomocą inwersji. Niech $V = \{1, 2, \ldots, n\}$ będzie zbiorem wierzchołków grafu, natomiast $a_1, a_2, \ldots, a_n$ — pewnym ciągiem parami różnych liczb ze zbioru V. Wierzchołki a_i oraz a_j są połączone krawędzią w grafie, jeśli para (i, j) jest *inwersją* w tym ciągu, to znaczy $i < j$ oraz $a_i > a_j$.

Dla przykładu rozważmy $n = 4$ i ciąg $2, 3, 1, 4$. Wtedy uzyskujemy graf jak na rysunku:



Bajtazar chce pokazać, że wymyślona przez niego reprezentacja jest użyteczna. Postanowił napisać program, który wyznacza *spójne składowe* grafu. Przypomnijmy, że dwa wierzchołki $u, v \in V$ znajdują się w tej samej spójnej składowej grafu, jeśli istnieje taki ciąg wierzchołków, którego pierwszym wyrazem jest u, ostatnim — v, a każde dwa kolejne wierzchołki są połączone krawędzią grafu. W naszym przykładzie mamy dwie spójne składowe: $\{1, 2, 3\}$ oraz $\{4\}$.

Pomóż Bajtazarowi!

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 1\,000\,000$) oznaczająca liczbę wierzchołków grafu. W drugim wierszu znajduje się ciąg n liczb całkowitych $a_1, a_2, \ldots, a_n$.

Wyjście

W pierwszym wierszu wyjścia należy wypisać liczbę spójnych składowych grafu; oznaczmy tę liczbę przez m. W każdym z kolejnych m wierszy należy podać opis jednej spójnej składowej grafu. Na początku wiersza wypisać należy liczbę k oznaczającą rozmiar składowej, a następnie *rosnący* ciąg k numerów wierzchołków tej składowej. Składowe należy wypisać w takiej kolejności, by pierwsze numery wierzchołków z każdego wiersza tworzyły ciąg rosnący. Innymi słowy, jeśli S i S' są dwiema składowymi, $u \in S$, $v \in S'$ są ich najmniejszymi wierzchołkami oraz $u < v$, to składową S należy wypisać przed składową S'.

Przykład

Dla danych wejściowych:

```
4
2 3 1 4
```

poprawnym wynikiem jest:

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

Rozwiązanie

Zadanie *Inwersje* było jednym z najprostszych zadań na Mistrzostwach Polski w 2012 roku. Choć większość drużyn uporało się z nim bez trudu, pewnie tylko nieliczni zawodnicy zorientowali się, że zadanie ma niezwykle proste rozwiązanie, które daje się zaimplementować w zaledwie kilku wierszach kodu. Inni wybrali trochę bardziej skomplikowane, standardowe podejście, o którym nie będziemy tu wspominać.

Naszym zadaniem jest wyznaczenie spójnych składowych grafu, który zakodowany jest za pomocą permutacji liczb od 1 do n. Sam graf może mieć na tyle dużo krawędzi, że jego jawna reprezentacja nie zmieściłaby się w limicie pamięci. Musimy zatem operować bezpośrednio na reprezentacji grafu podanej w zadaniu.

Zastanówmy się, jak wygląda spójna składowa, do której należy a_1. Jeśli $a_1 = 1$, to wierzchołek 1 tworzy jednoelementową spójną składową, bo na prawo od a_1 nie ma wyrazu ciągu o mniejszej wartości, czyli z 1 nie wychodzi żadna krawędź. Co jeśli $a_1 > 1$? Rozważmy przykładową reprezentację grafu dla $n = 10$:

$$\boxed{4}\quad 1\quad 8\quad 2\quad 7\quad 3\quad 5\quad 6\quad 10\quad 9$$

Prostokątem zaznaczamy wyrazy ciągu, o których wiemy, że należą do spójnej składowej wierzchołka a_1. Skoro $a_1 = 4$, to wierzchołki $1, 2, 3$ są połączone krawędziami z 4 (bo mają mniejsze numery i leżą na prawo od 4), zatem należą do tej samej spójnej składowej:

$$\boxed{4\quad 1}\quad 8\quad \boxed{2}\quad 7\quad \boxed{3}\quad 5\quad 6\quad 10\quad 9$$

Teraz w naszej spójnej składowej mamy wierzchołki $1, 2, 3, 4$, czyli wszystkie wierzchołki o numerach od 1 do a_1. Zauważmy jednak, że nie są to cztery pierwsze wyrazy ciągu, bo między 1 a 2 mamy wierzchołek 8, a między 2 a 3 jest też wierzchołek 7. Te wierzchołki również należą do spójnej składowej wierzchołka a_1: ich numery są większe od numerów wierzchołków wcześniej dodanych do składowej i na prawo od nich znajduje się co najmniej jeden wierzchołek należący do składowej:

$$\boxed{4\quad 1\quad 8\quad 2\quad 7\quad 3}\quad 5\quad 6\quad 10\quad 9$$

Kiedy dodamy wierzchołki 7 i 8, w składowej mamy wierzchołek 8, zatem wszystkie wierzchołki leżące na prawo, których numery są mniejsze niż 8, również do naszej spójnej składowej należą. W naszym przypadku są to 5 i 6:

$$\boxed{4\quad 1\quad 8\quad 2\quad 7\quad 3\quad 5\quad 6}\quad 10\quad 9$$

Po tym kroku w naszej składowej są wszystkie wierzchołki o numerach od 1 do 8 i stanowią one 8 pierwszych wyrazów ciągu $a_1, \ldots, a_{10}$. Łatwo przekonać się, że nie istnieje krawędź od wierzchołka składowej do wierzchołka spoza niej. Taka krawędź musiałaby się kończyć w wierzchołku na prawo od wierzchołków składowej, w wierzchołku, którego numer jest mniejszy niż 8. Ale przecież wszystkie wierzchołki o numerach mniejszych niż 8 już należą do składowej!

W ogólności, aby wyznaczyć składową, do której należy a_1, stosujemy na przemian dwie reguły:

1. Jeśli w składowej jest wierzchołek m, to należą do niej także wierzchołki $1, \ldots, m-1$.

2. Jeśli w składowej jest wierzchołek a_k, to należą do niej także wierzchołki $a_1, \ldots, a_{k-1}$.

Dowód tych dwóch własności proponujemy jako pouczające ćwiczenie. Na początek warto udowodnić poprawność drugiej reguły, po czym założyć, że nasza spójna składowa to k pierwszych wyrazów ciągu, i na tej podstawie udowodnić poprawność pierwszej reguły. Pamiętajmy, że reguły te działają jedynie wtedy, gdy szukamy składowej zawierającej wierzchołek a_1.

W momencie, gdy stosowanie reguł nie będzie już powiększało naszej spójnej składowej, poznajemy jej ostateczną zawartość. Skoro reguły nie pozwalają powiększyć składowej, zawiera ona pewną liczbę początkowych wyrazów ciągu (reguła 2), a ponadto te wyrazy to wszystkie liczby od 1 do m (reguła 1). A zatem składowa zawiera dokładnie m pierwszych wyrazów ciągu $a_1, \ldots, a_n$.

Algorytm

Jak możemy wykorzystać ten fakt do skonstruowania efektywnego rozwiązania? Wiemy, że m pierwszych wyrazów ciągu może utworzyć spójną składową, tylko gdy zbiór tych wyrazów to $\{1, \ldots, m\}$. Aby sprawdzić ten warunek, skorzystamy z bardzo prostej własności. Rozważmy dowolny ciąg m parami różnych dodatnich liczb całkowitych. Otóż wyrazy tego ciągu tworzą zbiór $\{1, \ldots, m\}$ wtedy i tylko wtedy, gdy największa z tych liczb jest równa m.

Ten fakt pozwala nam wykryć, kiedy pierwsze m wyrazów ciągu to $\{1, \ldots, m\}$. Co więcej, gdy tylko znajdziemy najmniejsze takie m, łatwo przekonać się, że faktycznie znaleźliśmy spójną składową (zauważmy, że nie wychodzą z niej żadne krawędzie).

Algorytm jest więc wyjątkowo prosty: wystarczy utrzymywać liczbę przejrzanych wyrazów ciągu oraz wartość największego z nich. Kiedy tylko te dwie wartości staną się równe, znaleźliśmy składową zawierającą a_1. W tym momencie od naszego ciągu możemy odciąć m pierwszych wyrazów i zająć się szukaniem spójnych składowych w pozostałej części ciągu. A do tego użyjemy takiego samego algorytmu. Wystarczy tylko pamiętać, że odtąd nasz ciąg zawiera $n-m$ różnych liczb z zakresu od $m+1$ do n. Możemy teraz od każdego z pozostałych wyrazów odjąć m i powtórzyć opisany algorytm. Oczywiście nie warto odejmować w sposób jawny: lepiej gdzieś na boku zapisać, że od każdego wyrazu, który pozostał w ciągu, chcemy odjąć m. W ten sposób otrzymujemy algorytm działający w czasie liniowym od rozmiaru danych wejściowych.

J

Jutro

Autor zadania: Tomasz Idziaszek
Opis rozwiązania: Jakub Radoszewski
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2012/jut`

prokrastynacja (łac. *procrastinatio* z *pro cras* — na jutro)
— tendencja do nieustannego przekładania pewnych czynności na później

Bajtazar ma w zwyczaju odkładać wszystko na ostatnią chwilę. Właściwie można powiedzieć, że prokrastynacja to jego drugie imię. Niemniej jednak, jeśli się do czegoś zobowiąże, to można na niego liczyć. Bajtazar wstał dzisiaj rano i wypisał sobie listę n zadań, które musi wykonać w najbliższym czasie. Wykonanie i-tego zadania z listy zajmie mu dokładnie d_i kolejnych dni, zaś musi ono zostać ukończone przed upływem t_i dni, licząc od dziś. Bajtazar chciałby wiedzieć, jak długo może zwlekać, zanim będzie musiał wziąć się w końcu do roboty. Pomóż mu i napisz program, który to obliczy. Bajtazar mógłby to zrobić sam, ale byłoby to wbrew jego naturze.

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 1\,000\,000$), oznaczająca liczbę zadań, które musi zrealizować Bajtazar. W kolejnych n wierszach znajdują się opisy zadań: i-ty z tych wierszy zawiera dwie liczby całkowite d_i i t_i ($1 \leqslant d_i, t_i \leqslant 10^9$). Zakładamy, że Bajtazar będzie w stanie zrealizować wszystkie zaplanowane zadania.

Wyjście

Na wyjście należy wypisać jedną liczbę całkowitą k, oznaczającą liczbę dni, przez które Bajtazar może unikać pracy. Innymi słowy, najpóźniej w dniu $k + 1$ musi zacząć wykonywać jakieś zadanie, aby był w stanie zrealizować swój plan.

Przykład

Dla danych wejściowych:	poprawnym wynikiem jest:
3 2 8 1 13 3 10	5

Wyjaśnienie przykładu: Bajtazar przez pięć dni odpoczywa, przez następne pięć dni wykonuje zadanie pierwsze oraz trzecie (w tej kolejności), a następnie jeden z trzech kolejnych dni poświęca na wykonanie zadania drugiego.

Rozwiązanie

Nasz bohater ma do wykonania n zadań, z których i-te ma znany czas wykonywania d_i oraz termin ukończenia t_i. Wiadomo, że na pewno zdąży wykonać wszystkie zadania w terminie, a zależy mu tylko na tym, by jak najpóźniej zacząć pracę. Ponadto każde zadanie należy wykonywać przez d_i *kolejnych* dni. Zauważmy jednak, że nie ma to większego znaczenia: gdyby nasz bohater zdecydował się wykonywać jakieś zadanie w kilku fragmentach czasu, to zamiast tego mógłby je zrealizować za jednym zamachem, a wszystko inne, co robił w międzyczasie, wykonać odpowiednio wcześniej.

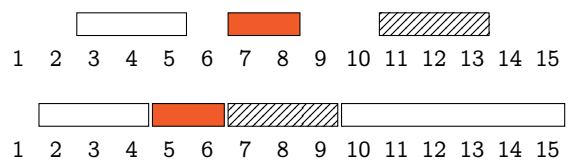
W problemach tego typu główną trudnością jest zazwyczaj określenie najlepszej możliwej kolejności obsługiwania zadań. W naszym przypadku jest to jednak wyjątkowo proste i intuicyjne: zadania należy przetwarzać według terminów ich ukończenia. Formalnie można by to uzasadnić tak: jeśli zadania nie byłyby wykonywane zgodnie z kolejnością terminów ukończenia, to jakieś zadanie o późniejszym terminie ukończenia zostało zrealizowane bezpośrednio przed innym zadaniem o wcześniejszym terminie. Wówczas jednak moglibyśmy zamienić w czasie wykonania tych dwóch zadań i otrzymalibyśmy inne, równie dobre rozwiązanie.

Zadania wykonywać będziemy w kolejności niemalejących terminów ukończenia. Dla uproszczenia załóżmy, że $t_1 \leqslant t_2 \leqslant \ldots \leqslant t_n$. Pozostaje nam zaplanować, kiedy rozpocząć wykonywanie każdego z zadań. Można to robić zasadniczo na dwa sposoby.

Od początku do końca

W pierwszej metodzie rozpatrujemy zadania po kolei, starając się je wykonywać możliwie jak najpóźniej. Załóżmy, że przejrzeliśmy już $i-1$ pierwszych zadań. Wiadomo, że wszystkie z nich musieliśmy ukończyć w trakcie t_{i-1} pierwszych jednostek czasu. W trakcie przetwarzania zadań pamiętamy, przez ile jednostek czasu nie wykonywaliśmy żadnych zadań. Tę wartość nazwiemy *zapasem czasu*. Ponieważ szczególnie interesuje nas zapas czasu przed rozpoczęciem jakiejkolwiek pracy (tę wartość chcemy zmaksymalizować), więc cały zapas czasu podzielimy na dwie części: początkowy zapas (zmienna *pocz_zapas*) i pozostały zapas, znajdujący się gdzieś między wykonaniami poszczególnych zadań (zmienna *dod_zapas*).

Pokażemy, jak aktualizować te dwie zmienne przy przetwarzaniu kolejnego, i-tego zadania. Mamy wtedy do dyspozycji $t_i - t_{i-1}$ dodatkowych jednostek czasu. Jeśli ten czas wystarczy na realizację zadania, to wykonujemy to zadanie najpóźniej



Rysunek 1. Ostatni krok obliczania rozwiązania dla ciągu czterech zadań: $(t_1 = 5, d_1 = 3)$, $(t_2 = 8, d_2 = 2)$, $(t_3 = 13, d_3 = 3)$, $(t_4 = 15, d_4 = 6)$. Dla poprawy czytelności zadania oznaczyliśmy różnymi kolorami.

jak się da, a (dodatkowy) zapas czasu zwiększamy o $t_i - t_{i-1} - d_i$. W przeciwnym razie musimy „pożyczyć" trochę czasu z wcześniejszego zapasu. W praktyce osiągamy to tak, że wykonania wcześniejszych zadań przesuwamy wstecz, począwszy od ostatniego, tak aby zwolnić d_i jednostek czasu — przypomina to przesuwanie się wagoników po torze, gdy skrajnie prawy z nich pchamy w lewo (patrz rysunek 1). Zauważmy, że w pierwszej kolejności zmniejszamy zmienną *dod_zapas*, a w ostateczności zmienną *pocz_zapas*.

Oto pseudokod takiego rozwiązania.

```
pocz_zapas := t_1 − d_1
dod_zapas := 0
for i := 2 to n do
  dod_zapas := dod_zapas + t_i − t_{i−1} − d_i
  if dod_zapas < 0 then
    pocz_zapas := pocz_zapas + dod_zapas
    dod_zapas := 0
return pocz_zapas
```

Spostrzeżenie związane z przesuwaniem zadań wstecz („wagoniki") można wykorzystać do dalszego uproszczenia całego rozwiązania. Załóżmy, że rozważamy zadanie j i w efekcie zmniejszamy początkowy zapas czasu. Wobec tego w naszym częściowym rozwiązaniu uwzględniającym zadania o numerach od 1 do j pracujemy bez przerwy od początku aż do momentu t_j, a początkowy zapas czasu to dokładnie:

$$t_j - d_1 - d_2 - \ldots - d_j.$$

Zatem dla pewnego zadania j, w trakcie jego przetwarzania po raz ostatni modyfikujemy początkowy zapas czasu i ustawiamy wartość *pocz_zapas* zgodnie z powyższym wzorem. Gdybyśmy znali ową wartość j, od razu moglibyśmy obliczyć odpowiedź. Jednak co zrobić, gdy jej nie znamy? Wystarczy wziąć minimum z wyrażeń tej postaci dla wszystkich $i = 1, \ldots, n$. Dlaczego? Otóż któreś z tych wyrażeń będzie rzeczywiście szukanym wyrażeniem wynikowym, a przecież od razu widzimy, że wynik w zadaniu nie może być większy od żadnego z nich!

Od końca do początku

W tej metodzie zadania będziemy przetwarzać od końca, również starając się wykonywać je jak najpóźniej. Można też na to spojrzeć inaczej. Odwróćmy mianowicie oś czasu. Dostajemy następujący, równoważny problem: mamy pewną liczbę zadań do wykonania, z których każde ma określony czas wykonywania d_i oraz moment $t'_i = -t_i$, kiedy najwcześniej możemy zacząć je wykonywać. Naszym zadaniem jest jak najszybciej zakończyć wykonywanie wszystkich zadań.

Tak postawiony problem już bardzo łatwo rozwiązać. Zadania wykonujemy w kolejności $t'_1 \leqslant t'_2 \leqslant \ldots \leqslant t'_n$. Wystarczy pamiętać tylko jedną zmienną: chwilę, w której skończyliśmy przetwarzać ostatnie zadanie (oznaczoną poniżej jako *koniec*).

```
koniec := −∞
for i := 1 to n do
  koniec := max(koniec, t'_i) + d_i
return koniec
```

Rozwiązaniem oryginalnego problemu będzie oczywiście liczba przeciwna do liczby *koniec*.

Każda z zaprezentowanych metod działa w czasie liniowym, jeśli tylko zadania są posortowane względem terminu wykonania. Złożoność czasowa całego rozwiązania to w obu przypadkach $O(n \log n)$.

Króliki

Autor zadania: Tomasz Idziaszek
Opis rozwiązania: Jakub Radoszewski
Dostępna pamięć: 256 MB
https://oi.edu.pl/pl/archive/amppz/2012/kro

Bajtazar postanowił zacząć żyć ekologicznie, i został hodowcą sałaty. Jak wiadomo, bajtockie króliki uwielbiają sałatę, nic więc dziwnego, że wkrótce zawitały one do ogródka Bajtazara.

Wokół domu Bajtazara rozmieszczone jest n grządek sałaty, które będziemy numerować od 1 do n. Każde dwie kolejne grządki sąsiadują ze sobą, tzn. dla każdego $i = 1, 2, \ldots, n-1$ grządki o numerach i oraz $i+1$ sąsiadują ze sobą. Dodatkowo grządka o numerze n sąsiaduje z grządką o numerze 1. Na grządce o numerze i siedzi a_i królików i pałaszuje sałatę Bajtazara.

Bajtazar chciałby przegonić z ogródka jak najwięcej królików. Ma do dyspozycji swoją starą, wierną strzelbę, w której jest k nabojów. Króliki są bardzo płochliwe, więc gdy tylko Bajtazar wystrzeli w kierunku grządki o numerze i, to wszystkie króliki z tej grządki uciekają w siną dal. Co więcej, przestraszone króliki z obu sąsiednich grządek przeskakują na dalsze grządki, tzn. kicają na grządkę sąsiadującą z tą, na której stoją — oczywiście nie tą, w kierunku której strzelał Bajtazar.

Pomóż Bajtazarowi i wyznacz maksymalną liczbę królików, które może on wygonić z ogródka za pomocą co najwyżej k strzałów.

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite n i k ($5 \leqslant n \leqslant 2000$, $1 \leqslant k \leqslant n$), oznaczające liczbę grządek w ogródku i liczbę nabojów w strzelbie Bajtazara. W drugim wierszu znajduje się ciąg liczb całkowitych $a_1, a_2, \ldots, a_n$ ($0 \leqslant a_i \leqslant 1\,000\,000$), które oznaczają liczby królików na kolejnych grządkach.

Wyjście

Na wyjście należy wypisać jedną liczbę całkowitą — maksymalną liczbę królików, które Bajtazar jest w stanie przegonić z ogródka.

Przykład

Dla danych wejściowych:	poprawnym wynikiem jest:
5 2 6 1 5 3 4	13

Wyjaśnienie przykładu: Bajtazar przegania 6 królików z grządki numer 1 (wskutek czego króliki z grządki numer 5 przeskakują na grządkę numer 4, zaś króliki z grządki numer 2 przeskakują na grządkę numer 3), a następnie przegania 7 królików z grządki numer 4.

Rozwiązanie

Dla uproszczenia opisu zamiast o grządkach będziemy mówić o polach. Zacznijmy od tego, ile strzałów na pewno wystarczy, by przegonić wszystkie króliki. Jeśli n jest parzyste, to możemy strzelić raz w kierunku każdego kolejnego pola o numerze nieparzystym. Wtedy króliki z kolejnych pól o numerach parzystych będą przeskakiwać na kolejne pola o numerach nieparzystych, skąd będą przeganiane wskutek kolejnych strzałów. Podobnie jeśli n jest nieparzyste, to wystarczy $\lceil \frac{n}{2} \rceil$ strzałów.

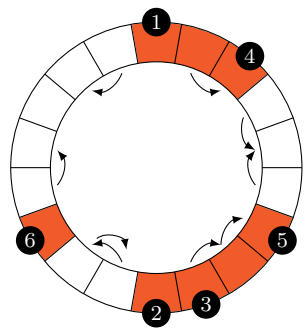
Rozpatrzenie tego przypadku pozwala wysunąć hipotezę, że dobrym pomysłem jest strzelanie w kierunku co drugiego pola. Postaramy się sformalizować tę intuicję.

Analiza przypadku ogólnego

Spróbujmy wyobrazić sobie, jak wygląda optymalne rozwiązanie, w którym strzelamy k razy. Powiemy, że pole jest *czyste*, jeśli po oddaniu rzeczonych k strzałów, króliki, które początkowo znajdowały się na tym polu, zostały skutecznie przegonione. Nie chodzi więc tylko o te pola, w których kierunku oddaliśmy strzały, gdy były tam króliki, lecz także o takie, z których króliki najpierw przeskoczyły na inne pola, a dopiero później zostały z nich przegonione. Podzielmy pola na okręgu na maksymalne przedziały złożone z pól czystych (patrz rysunek 1).

Rozważaliśmy już przypadek, gdy udaje nam się przegonić z ogródka wszystkie króliki — mamy wtedy tylko jeden przedział pokrywający wszystkie pola. Wcześniejsza analiza pokazuje, że w takim przypadku wystarczy oddać $\lceil \frac{n}{2} \rceil$ strzałów. Tyle jest konieczne, gdyż gdybyśmy oddali mniej strzałów, to z prostego wyliczenia wynika, że między którymiś dwoma kolejnymi polami, w których kierunku oddaliśmy strzały, musiałaby być przerwa długości co najmniej dwóch pól. Jednak łatwo widzimy, że wtedy nie bylibyśmy w stanie przegonić królików z żadnego z pól należących do tej przerwy.

Będziemy więc odtąd zakładać, że przedziały nie pokrywają wszystkich pól. Zastanówmy się, jak mogą wyglądać te przedziały. Przykładowo, czy mogą istnieć dwa przedziały, które sąsiadują z jednym polem przerwy? Jak się okazuje, nie. Załóżmy



Rysunek 1. Przykładowy okrąg zawierający $n = 18$ pól, w który oddajemy $k = 6$ strzałów (kolejne strzały są oznaczone ponumerowanymi czarnymi kółkami). Strzałki pokazują kierunek przeskakiwania królików. Czyste pola są zaznaczone kolorem; mamy 3 maksymalne przedziały.

bowiem, że mamy dwa takie przedziały. Żeby króliki położone na skrajnie prawym polu lewego przedziału zostały przegonione, konieczny był strzał w kierunku tego właśnie pola. Podobnie widzimy, że musieliśmy także oddać strzał w kierunku skrajnie lewego pola prawego przedziału. Rozważmy tylko te dwa strzały. W momencie oddawania pierwszego z nich króliki z pola pomiędzy przedziałami przeskoczyły na pole, w którego kierunku oddaliśmy drugi spośród tych strzałów. A zatem te króliki zostały przegonione drugim strzałem, co stanowi sprzeczność z założeniem. Ten sam argument pokazuje, że nie jest możliwe, by pola czyste tworzyły jeden przedział o długości $n-1$.

Odstępy między maksymalnymi przedziałami złożonymi z pól czystych są więc na tyle duże, by strzały oddane w jednym z nich nie miały wpływu na to, co dzieje się w pozostałych przedziałach. Możemy więc każdy z tych przedziałów rozpatrywać osobno. Weźmy pod uwagę jeden taki przedział złożony z m pól i załóżmy, że m jest nieparzyste. Zastanówmy się, ilu strzałów potrzeba, by przegonić wszystkie króliki z tego przedziału. Wcześniejsza analiza pokazuje, że w takim przedziale odstępy między kolejnymi polami, w których kierunku oddaliśmy strzały, nie mogą składać się z więcej niż jednego pola. To oznacza, że w tym przypadku potrzeba i wystarczy dokładnie $\lceil \frac{m}{2} \rceil$ strzałów, oddawanych w co drugie pole. Co ciekawe, strzały możemy wtedy oddawać w dowolnej kolejności (uzasadnienie tego faktu pomijamy).

A co jeśli m jest parzyste? W takim przypadku potrzebne byłoby $\frac{m}{2}+1$ strzałów. Tego przypadku jednak w ogóle nie musimy rozważać, gdyż zamiast takiego przedziału moglibyśmy taką samą liczbą strzałów „wyczyścić" dłuższy przedział, złożony z $m+1$ pól.

Spróbujmy podsumować całą analizę. Jeśli $k \geqslant \lceil \frac{n}{2} \rceil$, to jesteśmy w stanie przegonić wszystkie króliki. W przeciwnym przypadku przeganiamy króliki z pewnej liczby parami rozłącznych przedziałów nieparzystej długości, przy czym w każdym z nich strzelamy dokładnie w kierunku co drugiego pola. To pozwala nam ograniczyć się do szukania rozwiązania, które (w przypadku $k < \lceil \frac{n}{2} \rceil$) spełnia dwie własności. Po pierwsze, każde dwa strzały oddajemy w kierunku pól oddalonych od siebie o co najmniej dwa pola. Po drugie, oddawane strzały nigdy nie powodują, że króliki z pewnego pola skaczą dalej niż na sąsiednie pole. Zatem w każdej chwili liczba królików na danym polu zależy jedynie od strzałów oddanych w odległości co najwyżej dwóch pól od rozważanego pola. W dalszej części podamy rozwiązanie, które opiera się na tych dwóch własnościach.

Rozwiązanie wzorcowe

Nasze dotychczasowe obserwacje pozwalają skonstruować efektywne rozwiązanie oparte na programowaniu dynamicznym. Będziemy rozważać kolejne pola i dla każdego z nich sprawdzać, co stanie się, jeśli zdecydujemy się oddać strzał w jego kierunku. Wcześniej jednak musimy zmierzyć się z drobną trudnością polegającą na tym, że cała sytuacja ma miejsce na okręgu, więc strzały oddane na początku ciągu pól mogą wpływać na to, co dzieje się z królikami z pól końcowych. Na szczęście możemy to łatwo kontrolować. Wystarczy zapamiętać, czy oddaliśmy strzał w kierunku któregoś spośród dwóch pierwszych pól.

Możemy więc odtąd skoncentrować się na rozwiązaniu problemu na odcinku, czyli dla ciągu pól, w którym pole 1 nie sąsiaduje z polem n. Stany w programowaniu dynamicznym będziemy indeksować numerem pola $i \in \{1, \ldots, n\}$, liczbą

oddanych już strzałów $j \in \{0, \ldots, k\}$ oraz informacją o strzałach oddanych na ostatnich dwóch polach. Dla każdego z $3n(k+1)$ stanów pamiętamy maksymalną liczbę królików, jakie mogło się udać dotychczas przegonić. Ponumerujmy typy stanów:

(1) ostatni strzał był w kierunku pola i;

(2) ostatni strzał był w kierunku pola $i-1$;

(3) ostatni strzał był w kierunku pola o numerze mniejszym niż $i-1$ lub też nie oddaliśmy jeszcze żadnego strzału;

i przez $t_a[i,j]$ dla $a \in \{1,2,3\}$ oznaczmy wyniki dla stanów poszczególnych typów. Wówczas dla $i > 1$ mamy następujące wzory:

$$\begin{aligned} t_1[i,j] &= \max(t_2[i-1,j-1] + a_{i-1} + a_i,\ t_3[i-1,j-1] + a_i), \\ t_2[i,j] &= t_1[i-1,j], \\ t_3[i,j] &= \max(t_2[i-1,j],\ t_3[i-1,j]). \end{aligned}$$

Dla przykładu wyjaśnijmy pierwszy z nich. Opisuje on sytuację, gdy oddajemy strzał w kierunku pola i, i rozważa dwie możliwości. Jeśli ostatni strzał oddaliśmy w kierunku pola $i-2$ (stan $t_2[i-1,j-1]$), to króliki z pola $i-1$ znalazły się na polu i, zatem natychmiast przeganiamy $a_{i-1} + a_i$ królików. Jeśli zaś ostatni strzał wykonaliśmy w kierunku pola o numerze mniejszym niż $i-2$ lub nie oddaliśmy strzału (stan $t_3[i-1,j-1]$), to na polu i jest a_i królików i tylko te przeganiamy oddawanym strzałem. Z poprzednich rozważań wynika, że nie musimy rozważać sytuacji, w której poprzedni strzał oddaliśmy w kierunku pola $i-1$.

Aby otrzymać rozwiązanie działające na okręgu, należy rozszerzyć tę tablicę o jeszcze jeden wymiar, przechowujący informację, czy oddany został strzał w kierunku któregoś z pól 1 albo 2, i brać ten wymiar pod uwagę przy obliczaniu wartości dla $i \in \{n-1, n\}$. Wynik dla każdego stanu obliczamy w czasie stałym, więc całe rozwiązanie działa w czasie $O(nk)$.

2013

XVIII Akademickie Mistrzostwa Polski
w Programowaniu Zespołowym
Warszawa, 25–27 października 2013

A

Autostrada

Autor zadania: Jakub Radoszewski
Opis rozwiązania: Jakub Łącki
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2013/aut`

Spółka AutoBajt zajmuje się budową jednej z autostrad w Bajtocji. Dotychczas firma pobierała opłaty przy wjeździe na autostradę, jednak nowy prezes, Bajtazar, zauważył, że w takiej sytuacji opłata nie zależała od liczby przejechanych bajtomil. W związku z tym spółka planuje teraz wybudować kasy na samej autostradzie.

Bajtazar przejechał autostradą i, korzystając z zamontowanego w samochodzie licznika przejechanych bajtomil, zanotował pozycje wszystkich n wjazdów (pozycja wjazdu to jego odległość od początku autostrady). Firma postanowiła rozmieścić $n+1$ kas równomiernie na długości autostrady, to znaczy tak, by odległość między każdymi dwiema kolejnymi kasami była taka sama. Jednocześnie między każdymi dwiema kasami powinien być wjazd i między każdymi dwoma wjazdami powinna być kasa. Szczęśliwie okazało się, że przy istniejącym układzie wjazdów takie rozmieszczenie kas jest możliwe.

Twoim zadaniem jest obliczenie najmniejszej i największej możliwej odległości między kasami. Mówiąc formalnie, szukamy najmniejszej i największej wartości l, takiej że dla pewnej pozycji b_0 pierwszej kasy, kolejne kasy mogą zostać umieszczone na pozycjach $b_0 + l, b_0 + 2l, \ldots, b_0 + nl$. Dopuszczamy sytuację, w której wyznaczona w ten sposób pozycja pewnej kasy jest równa pozycji pewnego wjazdu (wówczas kasa zostanie wybudowana tuż przed lub tuż za wjazdem). Innymi słowy, pozycja j-tego wjazdu powinna zawierać się w przedziale $[b_0 + (j-1)l, b_0 + jl]$.

Wejście

W pierwszym wierszu wejścia znajduje się liczba całkowita n ($3 \leqslant n \leqslant 1\,000\,000$), oznaczająca liczbę wjazdów na autostradę. Drugi wiersz wejścia zawiera rosnący ciąg n liczb całkowitych $a_1, a_2, \ldots, a_n$ ($0 \leqslant a_i \leqslant 10^9$). Kolejne wyrazy ciągu opisują pozycje kolejnych wjazdów na autostradę w bajtomilach.

Wyjście

Twój program powinien wypisać dwie liczby rzeczywiste, oznaczające najmniejszą i największą możliwą odległość między dwiema kolejnymi kasami, w bajtomilach. Możesz założyć, że różnica między tymi liczbami jest nie mniejsza niż 10^{-9}.

Twój wynik będzie uznany za poprawny, jeżeli znajduje się w przedziale $[x(1-\varepsilon)-\varepsilon, x(1+\varepsilon)+\varepsilon]$, gdzie x jest prawidłową odpowiedzią, a $\varepsilon = 10^{-8}$. Tak więc tolerowany będzie zarówno błąd względny, jak i błąd bezwzględny równy ε.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
0.833333333333 1.250000000000
```

Rozwiązanie

W opisie rozwiązania skorzystamy z interpretacji geometrycznej. Z tego powodu zmienimy nieco oznaczenia z treści zadania. Rozstawienie kas opisane jest przez dwie liczby rzeczywiste. Niech x oznacza odległość między dwiema kolejnymi kasami, a y — pozycję początkowej kasy. Wówczas $n+1$ kas rozstawiamy na pozycjach

$$y,\ y+x,\ y+2x,\ \ldots,\ y+n\cdot x.$$

Dla ułatwienia kasy będziemy numerować od zera, tzn. przyjmiemy, że kasy mają numery $0,\ldots,n$. Pisząc początkowa kasa, będziemy mieć na myśli kasę o numerze 0.

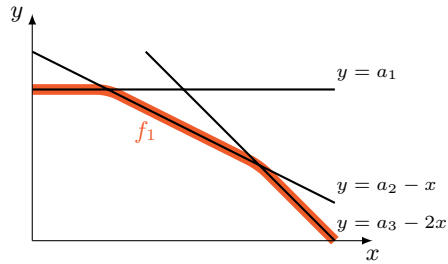
Rozstawienie kas musi spełniać dwa warunki: między każdymi dwiema kasami powinien być wjazd i między każdymi dwoma wjazdami powinna znajdować się kasa. Te warunki można równoważnie wyrazić następująco. Pozycja i-tej kasy to $y+i\cdot x$ (dla $0\leqslant i\leqslant n$), zaś pozycja i-tego wjazdu to a_i (dla $1\leqslant i\leqslant n$). Dla $i=0,\ldots,n-1$, i-ta kasa powinna znajdować się przed $(i+1)$-szym wjazdem, zaś $(i+1)$-sza kasa powinna znajdować się za $(i+1)$-szym wjazdem. Pierwszy warunek możemy zapisać jako $y+i\cdot x\leqslant a_{i+1}$, zaś drugi jako $y+(i+1)x\geqslant a_{i+1}$. Po przeniesieniu $i\cdot x$ oraz $(i+1)x$ na prawe strony, dla każdego $i=0,\ldots,n-1$ otrzymujemy

$$y\leqslant a_{i+1}-i\cdot x, \tag{1}$$
$$y\geqslant a_{i+1}-(i+1)x. \tag{2}$$

Każde rozwiązanie zadania spełnia wszystkie nierówności postaci (1) i (2), a każde rozwiązanie tych nierówności jest też rozwiązaniem zadania.

Teraz czas na interpretację geometryczną. Rozstawienie kas, w którym odległość między kasami to x, a początkowa kasa stoi na pozycji y, utożsamimy z punktem (x,y). Przyjrzyjmy się teraz nieco bliżej naszym nierównościom. Nierówności (1) mówią, że rozwiązania znajdują się na lub poniżej prostej danej równaniem $y=a_{i+1}-i\cdot x$, czyli w półpłaszczyźnie ograniczonej od góry tą prostą. Jeśli chcemy znaleźć zbiór punktów spełniających wszystkie nierówności (1), powinniśmy po prostu wyznaczyć przecięcie odpowiednich półpłaszczyzn. Przecięcie tych półpłaszczyzn jest ograniczone od góry pewną funkcją $f_1:\mathbb{R}\to\mathbb{R}$, która każdej współrzędnej x przypisuje najsilniejsze ograniczenie górne na współrzędną y wynikające z nierówności (1) (patrz rysunek 1):

$$f_1(x)=\min_{i=0,\ldots,n-1} a_{i+1}-i\cdot x. \tag{3}$$



Rysunek 1. Trzy funkcje ograniczające z góry półpłaszczyzny wyznaczone przez nierówności oraz funkcja f_1 ograniczająca z góry przecięcie tych półpłaszczyzn.

Z kolei nierówności (2) wyznaczają przecięcie półpłaszczyzn, które jest ograniczone od dołu pewną funkcją f_2. Zauważmy teraz, że f_1 jest funkcją wklęsłą, zaś f_2 to funkcja wypukła. Najprościej przekonać się o tym, patrząc na rysunek 1.

Zbiór punktów, które odpowiadają rozwiązaniom zadania, to wszystkie punkty leżące pod wykresem f_1 (lub na nim) i nad wykresem f_2 (lub na nim). Oznaczmy przez S obszar leżący pomiędzy wykresami funkcji f_1 i f_2. Aby rozwiązać zadanie, chcemy znaleźć rozwiązania o minimalnej i maksymalnej odległości między kasami, czyli punkty w obszarze S o minimalnej i maksymalnej pierwszej współrzędnej.

Na początku znajdźmy dowolny punkt w S. Wiemy, że S jest niepusty, więc dla pewnego $x \in \mathbb{R}$ musi zachodzić $f_1(x) \geqslant f_2(x)$, czyli $f_1(x) - f_2(x) \geqslant 0$. Ponieważ $f_1(x)$ to funkcja wklęsła, a $f_2(x)$ to funkcja wypukła, również $F(x) := f_1(x) - f_2(x)$ jest funkcją wklęsłą. Ponadto dla dowolnego x wartość $F(x)$ możemy wyznaczyć w czasie $O(n)$ za pomocą równania (3) i analogicznego wzoru dla f_2.

Teraz czas wykonać kluczowy krok: skorzystamy z wyszukiwania ternarnego, aby znaleźć współrzędną x_m, która *maksymalizuje* wartość $F(x_m)$ (patrz rysunek 2)*. Dla tej wartości x_m zachodzi $F(x_m) = f_1(x_m) - f_2(x_m) \geqslant 0$. Wyszukiwanie ternarne rozpoczynamy od dowolnego przedziału, w którym leży x_m, na przykład od przedziału $[0, a_n]$.

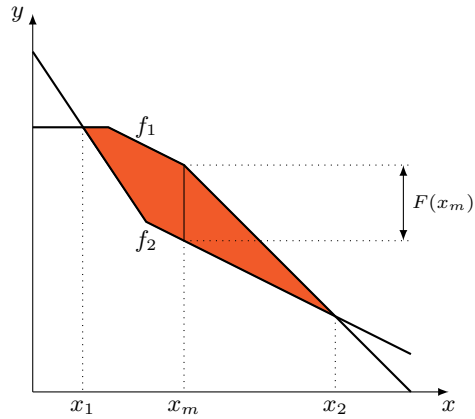
Wiemy teraz, że funkcja $F(x)$ jest niemalejąca na przedziale $[-\infty, x_m]$ i nierosnąca na przedziale $[x_m, \infty]$. Interesuje nas najmniejsza wartość x_1, dla której $F_1(x_1) \geqslant 0$, oraz największa wartość x_2 spełniająca taką samą nierówność. Zajmijmy się pierwszą z nich. Aby ją obliczyć, wystarczy w przedziale $[0, x_m]$ znaleźć wartość x_1, dla której $F(x_1) = 0$. W tym celu możemy posłużyć się wyszukiwaniem binarnym.

Czas działania takiego rozwiązania to $O(n \log M)$, gdzie M jest ograniczeniem na zakres danych i żądaną dokładność. Czynnik $\log M$ pochodzi więc z użycia wyszukiwania binarnego i ternarnego.

Krótsze rozwiązanie

Zadanie *Autostrada* możemy również rozwiązać za pomocą prostszego, choć nieco mniej naturalnego algorytmu. Do obliczenia każdej z poszukiwanych liczb będziemy chcieli użyć jednego wyszukiwania binarnego. Ustalmy pewną odległość między

*Mówiąc ściślej, możemy znaleźć wartość x_m^*, która leży dowolnie blisko x_m. Im więcej iteracji przeprowadzimy, tym bardziej zbliżymy się do x_m.



Rysunek 2. Obszar S (zaznaczony kolorem) między wykresami funkcji f_1 i f_2. Zaznaczono dwa skrajne punkty odpowiadające poszukiwanym rozwiązaniom oraz punkt x_m maksymalizujący wartość $F(x_m)$.

kasami x. Chcemy stwierdzić, która z trzech możliwości zachodzi: czy taka odległość jest dopuszczalna (tj. da się rozstawić $n+1$ kas w odległości x), za mała, czy też za duża. Jeśli na to pytanie będziemy potrafili odpowiadać w czasie $O(n)$, całe zadanie rozwiążemy w czasie $O(n \log M)$ za pomocą wyszukiwania binarnego.

Algorytm działa następująco. Ustalmy x. Rozstawiamy kolejne kasy na autostradzie i utrzymujemy przedział dopuszczalnych pozycji początkowej kasy, czyli wartości y. Dostawienie każdej kasy może skrócić nam ten przedział z każdej strony, bo uwzględnić musimy dwie dodatkowe nierówności (1) i (2). Jeśli po uwzględnieniu wszystkich kas przedział dopuszczalnych wartości y jest niepusty, wnioskujemy, że początkowa wartość x była dobra. W przeciwnym razie, przy stawianiu pewnej kasy przedział musiał się zdegenerować, tj. jego prawy koniec przesunęliśmy na lewo od lewego końca lub odwrotnie. Przyjmijmy, że stało się to podczas stawiania kasy numer i, tj. kasy oddalonej o $i \cdot x$ od początkowej kasy. Aby kasa zmieściła się przed wjazdem numer $i+1$, musi zachodzić $y \leqslant a_{i+1} - i \cdot x$. Twierdzimy, że jeśli nierówność ta nie zachodzi dla najmniejszej aktualnie dopuszczalnej wartości y (oznaczmy ją przez y_1), to przyjęta odległość między kasami jest zbyt duża.

Aby udowodnić to stwierdzenie, rozważmy dopuszczalne rozstawienie pierwszych $i-1$ kas. Ustaliliśmy już, że takie rozstawienie istnieje. Przesuńmy je teraz maksymalnie w lewo. Wówczas początkowa kasa znajdzie się na pozycji y_1, a pewna z kas „oprze się" z lewej strony o pewien wjazd na pozycji a_j (gdyby żadna kasa nie oparła się o wjazd, to dopuszczalna byłaby jeszcze mniejsza wartość y_1, wbrew definicji). Skoro nawet przy takim ustawieniu kasa numer i wypada za wjazdem a_{i+1}, to zwiększanie odległości między kasami na pewno nie pomoże, bo kasy opartej o wjazd a_j nie możemy już dalej przesunąć w lewo. Przyjęta wartość x jest więc za duża. W analogiczny sposób uzasadniamy, że jeśli dla maksymalnej dopuszczalnej wartości y nie zachodzi $y \geqslant a_i - i \cdot x$, to wartość x jest zbyt mała.

Zatem w czasie $O(n)$ potrafimy stwierdzić, czy nasza odgadnięta wartość x jest za mała, za duża, czy też dopuszczalna. Stąd od razu dostajemy rozwiązanie zadania w czasie $O(n \log M)$, które wykonuje jedynie dwa wyszukiwania binarne.

Szybsze rozwiązanie

Co ciekawe, całe zadanie daje się również rozwiązać w czasie liniowym. Pierwszym krokiem jest efektywne wyznaczenie obszarów poniżej wykresu funkcji f_1 i powyżej wykresu f_2. Dla ustalenia uwagi zajmijmy się wykresem f_1. Naszym zadaniem jest wyznaczenie f_1 na podstawie nierówności (1). Możemy to zrobić za pomocą algorytmu podobnego do algorytmu obliczania otoczki wypukłej zbioru punktów. Taki algorytm działa w czasie $O(n \log n)$, jednak czynnik logarytmiczny pochodzi jedynie z sortowania względem współczynników kierunkowych. W naszym przypadku współczynniki kierunkowe to kolejne liczby całkowite, zatem sortowanie możemy bez trudu wykonać w czasie liniowym. Gdy już wyznaczymy obszary ograniczone przez wykresy funkcji f_1 i f_2, możemy efektywnie wyznaczyć obszar S, a następnie znaleźć w nim dwa skrajne punkty.

Pomimo lepszego czasu działania nie polecamy tego rozwiązania. Jest ono kłopotliwe z powodu trudnych do obsłużenia problemów z niedokładnością reprezentacji liczb rzeczywistych, dlatego też lepiej traktować je jako ciekawostkę teoretyczną.

B

Bajthattan

Autor zadania: Jakub Łącki
Opis rozwiązania: Jakub Łącki
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2013/baj`

Bajthattan to jedna z wysp w stolicy Bajtocji. Na wyspie organizowane są liczne parady, defilady i festyny. Jest ich tak dużo, że często powodują zamknięcia ulic i poważne utrudnienia w ruchu. Bajtazar pracuje w ratuszu i jego zadaniem jest monitorowanie ruchu drogowego w mieście.

Ulice na Bajthattanie tworzą regularną siatkę $n \times n$. Spójrzmy na mapę Bajthattanu jak na układ współrzędnych: dla każdej pary liczb całkowitych x, y, takiej że $1 \leqslant x, y \leqslant n$, w punkcie o współrzędnych (x, y) znajduje się skrzyżowanie. Każde dwa skrzyżowania oddalone od siebie o 1 są połączone ulicą o długości 1.

Do Bajtazara napływają informacje o zamknięciach ulic. Każda taka informacja oznacza, że od tej pory ulica będzie nieprzejezdna. Po każdej informacji Bajtazar powinien stwierdzić, czy po zamknięciu ulicy nadal będzie się dało przejechać między skrzyżowaniami położonymi na jej końcach, poruszając się jedynie po ulicach, które dotychczas nie zostały zamknięte. Pomóż mu i napisz program, który wspomoże go w pracy.

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite n i k ($2 \leqslant n \leqslant 1500$, $1 \leqslant k \leqslant 2n(n-1)$). Oznaczają one odpowiednio liczbę ulic na Bajthattanie oraz liczbę informacji o zamknięciach ulic. W każdym z kolejnych k wierszy znajduje się informacja o zamknięciu jednej z ulic; informacje te są podane w porządku chronologicznym. Każdy z tych wierszy składa się z *dwóch* następujących po sobie opisów ulic. W rzeczywistości dokładnie jedna z tych dwóch ulic zostaje zamknięta*. Jeśli po zamknięciu ulicy określonej w *poprzednim* wierszu nadal daje się przejechać między skrzyżowaniami na jej końcach, zamknięta zostaje pierwsza z dwóch wymienionych ulic. W przeciwnym razie, zamknięcie dotyczy drugiej z nich. Pierwsze spośród k zamknięć opisanych na wejściu dotyczy pierwszej z dwóch wymienionych ulic. Każda ulica może zostać zamknięta co najwyżej jednokrotnie.

Opis jednej ulicy to para liczb całkowitych a_i, b_i ($1 \leqslant a_i, b_i \leqslant n$), po której następuje litera c_i ($c_i \in \{\mathtt{N}, \mathtt{E}\}$). Taka trójka wyznacza ulicę, której jeden koniec znajduje się na skrzyżowaniu o współrzędnych (a_i, b_i). Jeśli $c_i = \mathtt{N}$, drugi koniec tej ulicy leży na skrzyżowaniu o współrzędnych $(a_i, b_i + 1)$. Jeśli zaś $c_i = \mathtt{E}$, drugi koniec ulicy to skrzyżowanie o współrzędnych $(a_i + 1, b_i)$. Gdy $c_i = \mathtt{N}$, to $b_i < n$; podobnie, gdy $c_i = \mathtt{E}$, to $a_i < n$.

*Intencją sędziów jest, by taki nietypowy format wejścia stwarzał konieczność przetworzenia każdego zamknięcia ulicy przed rozpoczęciem przetwarzania kolejnych.

Wyjście

Na wyjście należy wypisać dokładnie k wierszy. Jeśli po i-tym zamknięciu z wejścia nadal da się przejechać między skrzyżowaniami na końcach zamykanej ulicy, to w i-tym wierszu wyjścia należy wypisać słowo `TAK`. W przeciwnym razie i-ty wiersz wyjścia powinien zawierać słowo `NIE`.

Przykład

Dla danych wejściowych:

```
3 4
2 1 E 1 2 N
2 1 N 1 1 N
3 1 N 2 1 N
2 2 N 1 1 N
```

poprawnym wynikiem jest:

```
TAK
TAK
NIE
NIE
```

Rozwiązanie

W zadaniu *Bajthattan* mamy do czynienia z grafem nieskierowanym, z którego usuwane są krawędzie. Po każdym usunięciu krawędzi musimy stwierdzić, czy dwa wierzchołki na jej końcach znajdują się w tej samej spójnej składowej grafu. Równoważnie, chcemy sprawdzić, czy usunięcie spowodowało wzrost liczby spójnych składowych.

Problem utrzymywania liczby spójnych składowych można nietrudno rozwiązać w przypadku, gdy krawędzie są *dodawane* do grafu. Wówczas wystarczy użyć struktury danych dla zbiorów rozłącznych, zwanej również *find-union*. Choć nasz problem wydaje się odwrotny, struktura find-union faktycznie przyda nam się w rozwiązaniu.

Wyjaśnijmy jeszcze, skąd wziął się tak nietypowy format danych wejściowych. Czy nie prościej byłoby po prostu podać listę usuwanych krawędzi? Faktycznie, ale to uprościłoby nie tylko wejście, ale i całe zadanie. Moglibyśmy bowiem na samym początku wczytać wszystkie usunięcia krawędzi i symulować cały proces od końca. Wówczas krawędzie byłyby do grafu dodawane i dla każdej dodawanej krawędzi musielibyśmy stwierdzić, czy łączy ona dwie różne spójne składowe. Z pomocą struktury find-union dostalibyśmy rozwiązanie działające w łącznym czasie $O(n^2 + k\alpha(n))$, gdzie $\alpha(n)$ to odwrotność funkcji Ackermanna. Choć w tym przypadku nietypowy format danych wejściowych nie pozwala nam użyć tej sztuczki, to warto o niej pamiętać, gdyż bywa bardzo pomocna.

Kluczem do rozwiązania jest specyficzna postać naszego grafu, a dokładniej fakt, że jest on *planarny*. Innymi słowy, da się go narysować na płaszczyźnie tak, by żadne krawędzie się ze sobą nie przecinały. Wyobraźmy sobie rysunek naszego

grafu (narysowany zgodnie z treścią zadania). Na rysunku mamy $(n-1) \times (n-1)$ obszarów rozmiaru 1×1 i jeden nieograniczony obszar. Te obszary nazywamy *ścianami*.

Skorzystamy z bardzo ważnego faktu dotyczącego grafów planarnych. Rozważmy dowolny rysunek pewnego grafu planarnego i oznaczmy przez v liczbę wierzchołków, przez e liczbę krawędzi, przez f liczbę ścian, zaś przez c liczbę spójnych składowych. Wówczas zachodzi równość znana jako *wzór Eulera*:

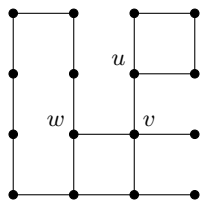
$$v - e + f = c + 1.$$

Wzór ten będzie podstawą do rozwiązania zadania. Zgodnie z tym wzorem, aby wyznaczyć liczbę spójnych składowych w grafie planarnym, wystarczy, że znamy liczby wierzchołków, krawędzi i ścian tego grafu. W naszym zadaniu liczba wierzchołków się nie zmienia i wynosi n^2, zaś liczba krawędzi wynosi na początku $2n(n-1)$ i spada o 1 z każdym usunięciem krawędzi. Nieco więcej trudności sprawia utrzymywanie liczby ścian.

Na początku mamy $(n-1)^2+1$ ścian. Za każdym razem, gdy usuwamy krawędź, powinniśmy sprawdzić, czy po obu stronach tej krawędzi znajdują się różne ściany. Jeśli tak, to po usunięciu krawędzi te dwie ściany połączą się i staną się jedną ścianą.

Zauważmy, że każda ściana w grafie powstałym po usunięciu pewnej liczby krawędzi składa się po prostu z pewnej liczby ścian początkowego grafu. Co więcej, w miarę jak usuwamy krawędzie, ściany mogą się jedynie łączyć ze sobą. Wobec tego do reprezentacji ścian w grafie możemy użyć struktury find-union! Tym razem zbiory reprezentowane przez tę strukturę to po prostu zbiory ścian oryginalnego grafu. Struktura find-union pozwala nam sprawdzać, czy po dwóch stronach krawędzi leży ta sama ściana, i łączyć ze sobą różne ściany. Każde usunięcie krawędzi wymaga co najwyżej dwóch operacji na strukturze find-union, a zatem czas działania takiego rozwiązania to $O(n^2 + k\alpha(n))$.

Umiemy zatem utrzymywać liczby wierzchołków, krawędzi i ścian. Ostatecznie ze wzoru Eulera wynika, że jeśli wskutek usunięcia krawędzi liczba ścian nie wzrosła, to usunięcie tej krawędzi zwiększa liczbę spójnych składowych. W przeciwnym razie, po usunięciu krawędzi liczba spójnych składowych pozostaje bez zmian (patrz rysunek 1).



Rysunek 1. Po obydwóch stronach krawędzi uv znajduje się ta sama ściana, więc usunięcie tej krawędzi zwiększy liczbę spójnych składowych. Z kolei krawędź vw rozdziela dwie różne ściany, więc jej usunięcie nie zwiększy liczby spójnych składowych.

Co jeszcze możemy zrobić?

W naszym zadaniu odpowiadamy na pewne zapytania dotyczące spójnych składowych w zmieniającym się grafie planarnym. Czy jednak moglibyśmy odpowiadać na bardziej ogólne pytania, na przykład o istnienie ścieżki łączącej dwa dowolnie wskazane wierzchołki? Jak się okazuje, ten problem możemy rozwiązać bardzo podobnym algorytmem.

Utrzymujemy tablicę *skl*, która wierzchołkowi v przypisuje jedną liczbę $skl[v]$, będącą identyfikatorem spójnej składowej, do której należy wierzchołek v. Wartości w tablicy mogą być dowolne, byleby tylko wierzchołki w ramach jednej spójnej składowej miały przypisaną taką samą liczbę, a liczby przypisane wierzchołkom z różnych składowych były różne.

Gdy, po usunięciu krawędzi uw, sprawdzimy, że liczba spójnych składowych w grafie nie wzrosła, nie musimy nic robić. W przeciwnym razie powstają dwie osobne spójne składowe C_u i C_w zawierające, odpowiednio, wierzchołki u i w. Uruchamiamy wówczas *równolegle* dwa przeszukiwania w głąb rozpoczynające się w wierzchołkach u i w i gdy tylko zakończy się pierwsze z nich, drugie przerywamy. W ten sposób wykryjemy *mniejszą* z dwóch powstałych spójnych składowych. Bez straty ogólności przyjmijmy, że $|C_u| \leqslant |C_w|$. Składowa C_u jest więc co najmniej dwukrotnie mniejsza od składowej $C_u \cup C_w$, która rozpadła się w wyniku usunięcia krawędzi uw. Aby zaktualizować tablicę *skl*, wystarczy dla każdego $x \in C_u$ zmienić $skl[x]$ na nową, unikatową wartość.

Aktualizacja tablicy zajmie więc czas $O(|C_u|)$. Co więcej, czas działania przeszukiwań to również $O(|C_u|)$, gdyż oba działały równolegle i zakończyły się w tym samym momencie. Korzystamy tu z faktu, że jeśli spójna składowa grafu planarnego ma v wierzchołków i e krawędzi, to $e \leqslant 3v - 6$. Zatem łączny czas działania całego algorytmu jest proporcjonalny do łącznej liczby zmian w tablicy *skl*. Jednocześnie, gdy zmieniamy wartość $skl[x]$, rozmiar spójnej składowej zawierającej wierzchołek x zmniejsza się co najmniej dwukrotnie. Zatem każda wartość może się zmienić co najwyżej $O(\log n)$ razy, dzięki czemu wykonamy w sumie tylko $O(n^2 \log n)$ zmian. Utrzymywanie tablicy *skl* działa więc w czasie $O(n^2 \log n)$.

Znając wartości tablicy *skl*, na zapytania odpowiadać możemy w czasie stałym. Zatem w niewiele większym czasie udało nam się rozwiązać znacznie ogólniejszy problem niż ten przedstawiony w zadaniu.

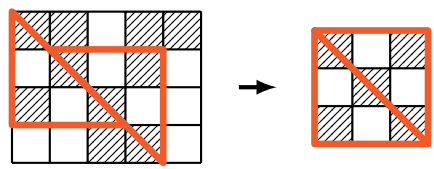
C

Cieśla

Autor zadania: Tomasz Idziaszek
Opis rozwiązania: Tomasz Idziaszek
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2013/cie`

Bajtazar chętnie zagrałby w warcaby, ale jego szachownica gdzieś się zawieruszyła. Udało mu się tylko znaleźć drewnianą deskę rozmiaru $n \times m$ podzieloną na nm jednakowych kwadratowych pól. Każde pole jest pomalowane na biało lub czarno, ale rozkład kolorów na desce niekoniecznie odpowiada planszy do warcabów. Bajtazar postanowił zatem wykorzystać swoje ciesielskie umiejętności i za pomocą piły wyciąć szachownicę, czyli kwadrat składający się z pewnej liczby pól, w którym każde dwa pola o wspólnym boku mają różne kolory.

Nie jest jasne, czy Bajtazarowi uda się znaleźć na desce kwadrat o odpowiedniej wielkości. Dlatego stwierdził on, że wytnie z deski dwa trójkątne kawałki i sklei je razem w taki sposób, by powstała szachownica. (Kawałki muszą być rozłączne, ale można je po wycięciu w dowolny sposób obracać.) Pomóż Bajtazarowi i oblicz, jaki jest największy rozmiar szachownicy, którą może on w ten sposób wyprodukować. Poniższy obrazek przedstawia deskę rozmiaru 4×5 i dwa trójkąty, które można skleić razem w szachownicę rozmiaru 3×3:



Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite n i m ($1 \leqslant n, m \leqslant 1000$), które oznaczają rozmiar deski. Kolejne n wierszy zawiera po m liczb całkowitych: j-ta liczba z i-tego wiersza ($1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant m$) oznacza kolor pola leżącego na przecięciu j-tej kolumny i i-tego wiersza deski. Liczba 0 oznacza białe pole, a liczba 1 — pole czarne.

Wyjście

W pierwszym i jedynym wierszu wyjścia należy wypisać jedną liczbę całkowitą, oznaczającą największy rozmiar szachownicy, którą można uzyskać przez wycięcie z deski dwóch trójkątnych kawałków i sklejenie ich.

Przykład

Dla danych wejściowych:

```
4 5
1 1 0 1 1
0 1 0 1 0
1 0 1 0 0
0 0 1 1 0
```

poprawnym wynikiem jest:

```
3
```

natomiast dla danych:

```
3 3
1 1 1
1 1 0
0 1 0
```

poprawnym wynikiem jest:

```
2
```

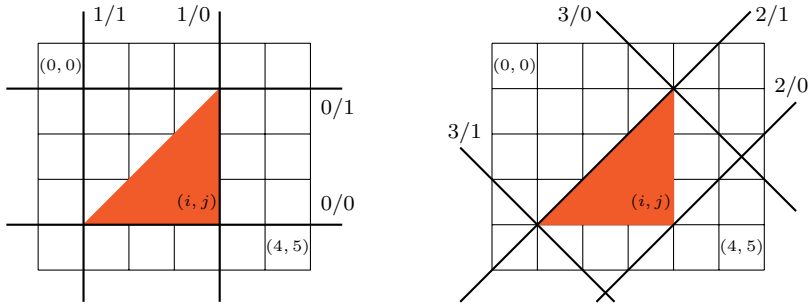
Rozwiązanie

W zadaniu mamy dany prostokąt rozmiaru $n \times m$ podzielony na nm pól; każde pole jest czarne lub białe. Mamy wyciąć z niego dwa trójkąty i złożyć z nich jak największą kwadratową szachownicę.

Jedyny sposób, w jaki można podzielić kwadrat o boku k na dwa trójkąty, to rozciąć go wzdłuż przekątnej. W efekcie otrzymujemy dwa trójkąty prostokątne, których obydwie przyprostokątne mają długość k. Co więcej, są one pokolorowane w szachownicę i mają takie same kolory pól przy kątach prostych. Łatwo przekonać się, że powinniśmy wobec tego wyciąć trójkąty, których przyprostokątne są równoległe do boków całego prostokąta.

Mając dane dwa takie trójkąty na płaszczyźnie, możemy je rozdzielić linią prostą — poziomą, pionową lub w jednym z dwóch kierunków skośnych. Zatem dla każdej takiej prostej i każdego z dwóch kolorów chcemy wiedzieć, jaki jest największy rozmiar trójkąta (pokolorowanego w szachownicę i o polu przy kącie prostym w ustalonym kolorze) po każdej stronie tej prostej. Przyjmujemy, że rozmiar trójkąta to długość jego przyprostokątnych. Mamy $n + 1$ prostych poziomych, $m + 1$ pionowych i po $n+m+1$ prostych w kierunkach skośnych, czyli w sumie $3n+3m+4$ proste (dla uproszczenia opisu uwzględniamy też proste zawierające się w bokach prostokąta oraz przechodzące przez jego rogi). Zatem jeśli znamy maksymalne rozmiary trójkątów, to odpowiedź bez problemu wyznaczymy w czasie $O(n + m)$.

Każdy trójkąt jest w jednej z czterech orientacji (w zależności od tego, po której stronie od wierzchołka przy kącie prostym znajduje się jego przeciwprostokątna). Algorytm opisany w dalszej części tekstu będzie działał dla ustalonej orientacji trójkątów. Należy go zatem wykonać czterokrotnie, za każdym razem obracając cały prostokąt o $90°$.



Rysunek 1. Prostokąt rozmiaru 5×6 oraz trójkąt rozmiaru 3 zaczepiony w polu $(i, j) = (3, 3)$. Na prawym rysunku zaznaczono osiem prostych dotykających trójkąta, które rozważamy w algorytmie. Każda z ośmiu prostych jest podpisana kierunkiem i stroną.

Trójkąty maksymalne

Na początek rozważmy trójkąt, którego pole przy kącie prostym ma ustalone współrzędne (i, j); będziemy mówić, że trójkąt jest *zaczepiony* w tym polu. Orientację trójkąta ustalamy tak, że pole to ma największe współrzędne spośród wszystkich pól trójkąta. Trójkąt taki nazwiemy *maksymalnym*, jeśli ma największy rozmiar spośród trójkątów pokolorowanych w szachownicę (tzn. nie da się go powiększyć, przesuwając jego przeciwprostokątną).

Rozmiary trójkątów maksymalnych dla wszystkich pól prostokąta możemy wyznaczyć w czasie $O(nm)$, korzystając z prostego programowania dynamicznego. Oczywiście każdy taki trójkąt ma rozmiar co najmniej 1. Natomiast trójkąt maksymalny zaczepiony w polu (i, j) ma rozmiar $k \geqslant 2$, jeśli pola $(i-1, j)$ i $(i, j-1)$ mają inny kolor niż pole (i, j) oraz mniejszy z trójkątów maksymalnych zaczepionych w tych polach ma rozmiar $k-1$.

Dla każdej z półpłaszczyzn wyznaczonych przez ustaloną prostą chcemy znaleźć największy rozmiar trójkąta *maksymalnego* znajdującego się w tej półpłaszczyźnie i dotykającego tej prostej. Rozważymy tu dwa przypadki w zależności od tego, czy dotyka on prostej wierzchołkiem przy kącie prostym. Na tej podstawie dla każdej z półpłaszczyzn obliczymy największe rozmiary *dowolnych* trójkątów dotykających prostych. Ten krok jest konieczny, gdyż największy trójkąt dotykający pewnej prostej może nie być trójkątem maksymalnym (gdy trójkąt ten możemy powiększyć, ale spowoduje to, że kawałek trójkąta znajdzie się po drugiej stronie rozważanej prostej). W trzecim kroku obliczymy największe rozmiary trójkątów po każdej stronie każdej z prostych.

Trójkąty maksymalne dotykające prostych

Każdy trójkąt maksymalny, który znaleźliśmy, dotyka dokładnie ośmiu prostych, przy czym trzech z nich wierzchołkiem przy kącie prostym (patrz rysunek 1). Każda prosta jest identyfikowana przez swój typ (kierunek i stronę, po której znajduje się trójkąt) oraz numer.

Proste mające ten sam typ ponumerujemy następująco. Dla prostych pozio-

mych (kierunek 0) i pionowych (kierunek 1) numerem prostej będzie jej odległość od równoległego boku prostokąta, który leży po tej samej stronie prostej co trójkąt. Dla prostych skośnych (kierunki 2 i 3) definiujemy numer prostej jako pomnożoną przez $\sqrt{2}$ odległość od najdalszego wierzchołka prostokąta, znajdującego się po tej samej stronie prostej co trójkąt. Dzięki takiej definicji numeracja prostych zmieni się bardzo nieznacznie po obróceniu prostokąta o 90°. Jedyna zmiana będzie taka, że proste o kierunku 0 zamienią się z prostymi o kierunku 1 o takich samych numerach i podobnie dla prostych o kierunkach 2 i 3. To pozwoli nam prościej połączyć wyniki uzyskane przy rozpatrywaniu czterech możliwych obrotów prostokąta.

Rozważmy trójkąt maksymalny rozmiaru k zaczepiony w polu (i, j). Poniższa tabelka przedstawia kierunki i numery ośmiu prostych, których dotyka ten trójkąt, jak również informacje, po której stronie prostej się on znajduje (kolumna *strona*). W późniejszej części algorytmu przyda nam się również informacja, czy trójkąt dotyka tej prostej wierzchołkiem przy kącie prostym (kolumna *dotyka*).

kierunek	*numer*	*strona*	*dotyka*
0	$i+1$	0	tak
0	$n-i+k-1$	1	nie
1	$j+1$	0	tak
1	$m-j+k-1$	1	nie
2	$i+j+2$	0	tak
2	$n+m-i-j+k-2$	1	nie
3	$n-i+j+k$	0	nie
3	$m-j+i+k$	1	nie

Naszym celem jest teraz wyznaczenie największego rozmiaru trójkąta maksymalnego dla każdej kombinacji powyższych czterech parametrów. Każdy z trójkątów maksymalnych rozpatrujemy w czasie stałym, zatem sumaryczny czas tej fazy algorytmu to $O(nm)$.

Pozostałe trójkąty

Mając powyższe dane, tzn. największe rozmiary trójkątów maksymalnych dotykających prostych danego typu, jednym zamiataniem możemy obliczyć największe rozmiary dowolnych trójkątów dotykających prostych. Korzystamy tu z obserwacji, że przesuwając prostą dotykającą trójkąta rozmiaru k w stronę trójkąta, dostajemy prostą dotykającą trójkąta rozmiaru $k-1$. Istotna jest tu informacja, czy prosta dotyka trójkąta w wierzchołku przy kącie prostym, bo wtedy mniejszy trójkąt ma przeciwny kolor. Zamiatanie realizujemy znowu programowaniem dynamicznym, przeglądając proste ustalonego typu w kolejności malejących numerów.

Następnie jednym zamiataniem możemy obliczyć maksymalne rozmiary dowolnych trójkątów leżących w półpłaszczyznach, niekoniecznie dotykających prostych. Korzystamy tu z oczywistej obserwacji, że oddalając prostą od trójkąta rozmiaru k, dostajemy brzeg półpłaszczyzny, w której leży trójkąt rozmiaru k. Zatem teraz przeglądamy proste danego typu w kolejności rosnących numerów.

Oba zamiatania wykonujemy w czasie $O(nm)$.

Podsumowanie

Ostatecznie uzyskujemy algorytm o złożoności czasowej i pamięciowej $O(nm)$. Rozwiązanie koncepcyjnie nie jest trudne i wymaga jedynie dobrego obycia z programowaniem dynamicznym. Jednak od strony implementacyjnej jest dość dużym wyzwaniem: należy uważnie rozważyć wszystkie przypadki, biegle operować wielowymiarową przestrzenią stanów (dla maksymalnego trójkąta dotykającego prostej musimy pamiętać aż pięć wartości: kierunek, stronę i numer prostej, kolor pola przy kącie prostym i informację, czy prosta dotyka tego pola) oraz przyjąć taką metodę reprezentowania stanów, w której łatwo uwzględnić różne przypadki (numeracja prostych).

D

Demonstracje

Autor zadania: Jakub Łącki
Opis rozwiązania: Jakub Łącki
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2013/dem`

W najbliższą niedzielę w Bajtogrodzie będzie obchodzony doroczny Dzień Bajtu — jedno z najważniejszych bajtockich świąt. Wszystko wskazuje jednak na to, że tegoroczne obchody nie będą tylko sielskim rodzinnym festynem.

Otóż w ostatnim czasie obywatele Bajtogrodu są podzieleni w jednej zasadniczej kwestii. Jedni uważają, że zgodnie z tradycją bajt powinien być zawsze równy ośmiu bitom. Są jednak zwolennicy postępu, którzy chętniej widzieliby dużo pojemniejsze, 16-bitowe bajty. Inni patrzą na całą sprawę znacznie bardziej rygorystycznie i najchętniej ogłosiliby, że bajt powinien mieć zawsze tylko 4 bity. Są wreszcie w Bajtogrodzie mniej znaczące ruchy wywrotowe, których członkowie twierdzą, że liczba bitów w bajcie nie powinna być potęgą dwójki, a nawet że nie musi być w ogóle parzysta! Każde ze stowarzyszeń zaplanowało na niedzielę demonstrację, w trakcie której będzie próbowało przekonać mieszkańców Bajtogrodu do swojej racji.

Wielu obywateli miasta obawia się, że duża liczba demonstracji może zakłócić obchody Dnia Bajtu. Burmistrz Bajtogrodu wyczuł, że może zyskać duże poparcie społeczne, nie wyrażając zgody na przeprowadzenie niektórych demonstracji. Ponieważ taka decyzja zawsze budzi kontrowersje, burmistrz postanowił, że ograniczy się do odwołania co najwyżej dwóch demonstracji. Chciałby przy tym wybrać takie demonstracje, po których odwołaniu łączny czas, w trakcie którego będą odbywały się w mieście jakiekolwiek demonstracje, będzie możliwie najkrótszy. Pomóż burmistrzowi i podpowiedz mu, ile czasu bez demonstracji w mieście jest on w stanie w ten sposób uzyskać.

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n ($2 \leqslant n \leqslant 500\,000$), oznaczająca liczbę zaplanowanych demonstracji. Każdy z kolejnych n wierszy opisuje jedną demonstrację: i-ty z tych wierszy zawiera dwie liczby całkowite a_i oraz b_i ($0 \leqslant a_i < b_i \leqslant 10^9$), które oznaczają, że i-ta demonstracja rozpoczyna się a_i bajtominut po wschodzie słońca i kończy się b_i bajtominut po wschodzie słońca.

Wyjście

Twój program powinien wypisać dokładnie jedną nieujemną liczbę całkowitą, oznaczającą, o ile maksymalnie może skrócić się czas, podczas którego odbywać się będą demonstracje, jeśli burmistrz Bajtogrodu odwoła co najwyżej dwie demonstracje.

Przykład

Dla danych wejściowych:

```
5
0 9
1 4
2 5
7 9
6 7
```

poprawnym wynikiem jest:

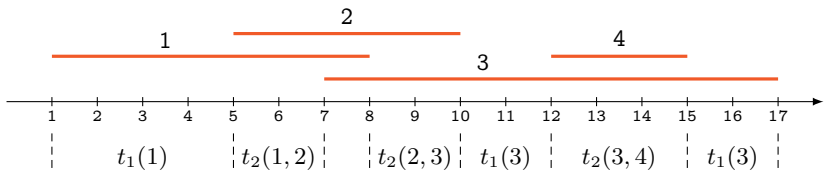
```
4
```

Wyjaśnienie przykładu: Burmistrz Bajtogrodu powinien nie wydać pozwoleń na pierwszą oraz na czwartą demonstrację.

Rozwiązanie

Zacznijmy od prostego spostrzeżenia. Jeśli w danym przedziale czasu odbywają się co najmniej trzy demonstracje, burmistrz nie może nic na to poradzić: nawet po jego decyzji w tym czasie odbędzie się co najmniej jedna z nich. A zatem interesują nas jedynie przedziały czasu, w których zaplanowane są co najwyżej dwie demonstracje, bo tylko w takich przedziałach czasu możemy coś zmienić.

Niech $t_1(i)$ oznacza łączny czas, gdy *jedyną* odbywającą się demonstracją jest demonstracja i. Z kolei przez $t_2(i, j)$ oznaczmy czas, gdy jednocześnie odbywają się demonstracje i oraz j i nie odbywa się żadna inna demonstracja. Przykład znajduje się na rysunku 1. Przy tych oznaczeniach, jeśli burmistrz odwoła demonstracje i oraz j, uzyska dokładnie $t_1(i) + t_1(j) + t_2(i, j)$ czasu bez demonstracji.



Rysunek 1. Rozważmy cztery demonstracje odbywające się w przedziałach czasu $[1, 8]$, $[5, 10]$, $[7, 17]$ i $[12, 15]$. Mamy wówczas $t_1(1) = 4$, $t_1(2) = 0$, $t_1(3) = 4$ i $t_1(4) = 0$. Ponadto $t_2(1, 2) = t_2(2, 3) = 2$ oraz $t_2(3, 4) = 3$. W tym przypadku optymalne rozwiązanie to odwołanie demonstracji 1 i 3, czyli demonstracji o dwóch największych wartościach $t_1(\cdot)$ (zwróć uwagę, że $t_2(1, 3) = 0$).

Pokażemy teraz, jak wyznaczyć wartości $t_1(i)$ oraz $t_2(i, j)$. W tym celu najpierw tworzymy tablicę, która zawiera opisy $2n$ *zdarzeń* postaci „demonstracja i rozpoczyna/kończy się o czasie t”, i sortujemy ją niemalejąco względem czasu. Następnie przeglądamy kolejno posortowane zdarzenia, czyli poruszamy się po osi czasu od lewej do prawej, i utrzymujemy zbiór aktualnie toczących się demonstracji. Gdy pojawia się kolejne zdarzenie, a aktualnie toczy się dokładnie jedna

demonstracja i, zwiększamy $t_1(i)$ o czas, który upłynął od ostatniego zdarzenia. Z kolei gdy toczą się dokładnie dwie demonstracje i i j, zwiększamy wartość $t_2(i,j)$ (oraz symetrycznie $t_2(j,i)$).

Do przechowywania aktualnie toczących się demonstracji wykorzystać możemy chociażby strukturę `set` dostępną w bibliotece STL. Wartości $t_1(i)$ możemy przechowywać w n-elementowej tablicy. Nie możemy tak zrobić z wartościami $t_2(i,j)$, gdyż tablica taka miałaby rozmiar n^2. Zauważmy jednak, że podczas przetwarzania zdarzeń co najwyżej $4n$ razy wykonujemy operację zwiększenia wartości $t_2(\cdot,\cdot)$. Tak więc istnieje nie więcej niż $4n$ par demonstracji i oraz j, takich że $t_2(i,j) > 0$. Do przechowywania tych wartości możemy posłużyć się strukturą `map` z STL.

Ponieważ sortujemy $2n$ zdarzeń, a następnie dla każdego zdarzenia wykonujemy stałą liczbę operacji na strukturze danych (każda w czasie $O(\log n)$), wyznaczenie wartości $t_1(i)$ oraz $t_2(i,j)$ zajmuje nam czas $O(n \log n)$.

Problem grafowy

Nasz problem ma pomocną interpretację grafową. Wyobraźmy sobie nieskierowany graf pełny, czyli taki, w którym każda para wierzchołków jest połączona krawędzią. Wierzchołki tego grafu odpowiadają demonstracjom. Wierzchołki i krawędzie mają przypisane wagi: wierzchołkowi odpowiadającemu demonstracji i przypisujemy wagę $t_1(i)$, zaś krawędzi łączącej wierzchołki i oraz j przypisujemy wagę $t_2(i,j)$. W tym grafie chcemy znaleźć taką parę wierzchołków, że suma ich wag oraz wagi łączącej je krawędzi jest jak największa. Co istotne, że nasz graf ma reprezentację rozmiaru $O(n)$, gdyż jedynie $4n$ krawędzi w grafie ma dodatnią wagę.

Odtąd skoncentrujemy się na rozwiązaniu problemu grafowego. Dla danych dwóch wierzchołków i oraz j oznaczmy sumę ich wag oraz wagi krawędzi je łączącej przez $T(i,j) := t_1(i) + t_1(j) + t_2(i,j)$. Wprawdzie możliwych par wierzchołków i oraz j jest dużo, jednak tylko $4n$ krawędzi ma dodatnią wagę i te pary wyznaczyliśmy w pierwszej fazie algorytmu. Możemy zatem przejrzeć wszystkie te krawędzie i wybrać taką krawędź ij, która maksymalizuje $T(i,j)$. Czy to już koniec? Jeszcze nie — to niestety nie musi być optymalne rozwiązanie, bo może się zdarzyć, że maksymalną wartość $T(i,j)$ uzyskamy dla wierzchołków połączonych krawędzią o wadze 0. Aby rozważyć ten przypadek, znajdujemy dwa wierzchołki i' oraz j' o największych wagach, tzn. takie, które maksymalizują $t_1(i') + t_1(j')$. Ostateczny wynik dostaniemy, wybierając lepsze z dwóch rozważanych rozwiązań.

Warto zwrócić uwagę na pewien szczegół: w drugim przypadku szukamy dwóch największych wag t_1 i może się zdarzyć, że ostatecznie znajdziemy dwa wierzchołki i' oraz j' połączone krawędzią o dodatniej wadze. Czy nie stworzy to problemu? Jak się okazuje, nie. Gdybyśmy ograniczyli się do rozpatrywania tylko par wierzchołków i i j połączonych krawędzią o wadze 0, na pewno uzyskalibyśmy nie lepszy wynik (bo szukalibyśmy maksimum w mniejszym zbiorze). Z drugiej strony, ponieważ i' i j' są połączone krawędzią o dodatniej wadze, to parę wierzchołków i' oraz j' uwzględnimy także w pierwszej fazie algorytmu. Co więcej, zysk pochodzący z tej pary będzie wynosił $t_1(i') + t_1(j') + t_2(i',j')$, czyli będzie większy niż $t_1(i') + t_1(j')$, bo $t_2(i',j') > 0$.

Udowodniliśmy zatem poprawność naszego algorytmu. Prosta analiza pozwala nam ograniczyć jego czas działania przez $O(n \log n)$.

Posłowie

Gdy przygotowywaliśmy to zadanie na zawody, rozważaliśmy możliwość dołożenia do niego drugiego polecenia polegającego na wyborze takich dwóch demonstracji, po których odwołaniu łączny czas odbywania się demonstracji w mieście byłby *jak najdłuższy*. W naszej interpretacji grafowej szukalibyśmy wtedy pary wierzchołków i oraz j *minimalizującej* wartość $T(i,j) = t_1(i) + t_1(j) + t_2(i,j)$.

Na pierwszy rzut oka problem ten może wydawać się symetryczny do tego z zadania, jednak w istocie jest on trochę trudniejszy. Podobnie jak poprzednio rozwiązanie byłoby dwuczęściowe: rozważamy $O(n)$ par wierzchołków połączonych krawędziami o dodatnich wagach (ta część jest bez zmian), a spośród pozostałych par chcemy wybrać taką, która minimalizuje sumę wag t_1 wierzchołków. Jednak tym razem w drugiej części rozwiązania nie możemy po prostu wybrać dwóch wierzchołków o minimalnych wagach, gdyż ewentualna krawędź o dodatniej wadze łącząca te wierzchołki może powodować, że nie dadzą one minimalnego wyniku. Na szczęście można inaczej podejść do tej części rozwiązania, otrzymując algorytm działający w czasie $O(n \log n)$. Dopracowanie tego pomysłu pozostawiamy do przemyślenia czytelnikowi.

Egzamin

Autor zadania: Jakub Łącki
Opis rozwiązania: Tomasz Idziaszek
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2013/egz`

Profesor Bajtoni szykuje egzamin z *Teorii bitów i bajtów*. Przygotował już n pytań. Każdemu z nich profesor przypisał oczekiwany współczynnik trudności będący liczbą naturalną z zakresu od 1 do n. Każde pytanie uzyskało inny współczynnik trudności.

Teraz profesor zastanawia się nad kolejnością pytań na egzaminie. Profesor chciałby sprawdzić, czy jego studenci potrafią samodzielnie oceniać trudność pytań. Dlatego planuje on uszeregować pytania tak, aby współczynniki trudności kolejnych pytań różniły się co najmniej o k. Pomóż profesorowi znaleźć takie uszeregowanie.

Wejście

Pierwszy i jedyny wiersz wejścia zawiera dwie liczby całkowite n i k ($2 \leqslant n \leqslant 1\,000\,000$, $1 \leqslant k \leqslant n$), oznaczające liczbę pytań przygotowanych przez profesora oraz dolne ograniczenie na różnicę trudności kolejnych pytań na egzaminie.

Wyjście

Twój program powinien wypisać jeden wiersz, zawierający szukane uszeregowanie współczynników trudności pytań, czyli ciąg n parami różnych liczb naturalnych z zakresu od 1 do n, w którym każde dwie kolejne liczby różnią się co najmniej o k. Jeśli jest wiele poprawnych odpowiedzi, Twój program powinien wypisać dowolną z nich. Jeśli szukane uszeregowanie pytań nie istnieje, Twój program powinien wypisać tylko jedno słowo `NIE`.

Przykład

Dla danych wejściowych:

```
5 2
```

jednym z poprawnych wyników jest:

```
2 4 1 5 3
```

natomiast dla danych:

```
5 4
```

poprawnym wynikiem jest:

```
NIE
```

Rozwiązanie

W najprostszym zadaniu na zawodach w 2013 roku mieliśmy znaleźć permutację liczb od 1 do n, w której różnica każdych dwóch sąsiednich liczb wynosi co najmniej k (lub stwierdzić, że taka permutacja nie istnieje).

Zauważmy przede wszystkim, że dla ustalonej wartości n nie musimy przygotowywać różnych permutacji dla wszystkich możliwych wartości parametru k. Permutacja spełniająca wymaganą własność dla parametru k spełnia ją również dla wszystkich mniejszych wartości parametru, więc wystarczy nam znaleźć największą wartość k_n, dla której permutacja istnieje, a następnie wypisywać tę permutację dla wszystkich $k \leqslant k_n$. Z kolei dla $k > k_n$ wypisujemy odpowiedź `NIE`.

Na początek warto wykonać parę eksperymentów na kartce, sprawdzając różne permutacje dla małych wartości n. Jest spora szansa, że w wyniku tych eksperymentów zauważymy, że permutacje o dużych różnicach sąsiednich elementów otrzymać można następująco. Dzielimy liczby od 1 do n na dwa ciągi w miarę równej długości, zawierające odpowiednio małe i duże liczby, a następnie przeplatamy te ciągi ze sobą. Dzięki temu każda para sąsiednich liczb będzie zawierać jedną małą i jedną dużą liczbę. W praktyce wygląda to następująco: dzielimy liczby na ciągi $1, 2, \ldots, \lfloor \frac{n+1}{2} \rfloor$ oraz $\lfloor \frac{n+1}{2} \rfloor + 1, \ldots, n$, a następnie przeplatamy je tak, aby $\lfloor \frac{n+1}{2} \rfloor$ było ostatnią liczbą w permutacji. Przykładowo dla $n = 6$ przeplatamy ze sobą ciągi $1, 2, 3$ i $4, 5, 6$, otrzymując:

$$4\ 1\ 5\ 2\ 6\ 3.$$

Z kolei dla $n = 7$ uzyskamy permutację

$$1\ 5\ 2\ 6\ 3\ 7\ 4.$$

W ogólności, dla parzystego n wynikowy ciąg rozpoczynamy od najmniejszej spośród dużych liczb, zaś dla nieparzystych n pierwszą liczbą wypisywanego ciągu jest 1. Nietrudno się przekonać, że różnica każdych dwóch sąsiednich liczb wynosi tu $\lfloor \frac{n}{2} \rfloor$ lub $\lfloor \frac{n}{2} \rfloor + 1$.

Jak się okazuje, lepiej się już nie da i możemy przyjąć $k_n = \lfloor \frac{n}{2} \rfloor$. Aby to wykazać, zauważmy, że musimy w którymś momencie wypisać liczbę $\lfloor \frac{n+1}{2} \rfloor$. Tuż obok niej wypisujemy pewną inną liczbę z zakresu od 1 do n. Różnica między tymi dwiema liczbami wynosi co najwyżej $\lfloor \frac{n}{2} \rfloor$ (w najbardziej optymistycznym przypadku, tj. gdy drugą liczbą jest n, różnica wyniesie dokładnie $\lfloor \frac{n}{2} \rfloor$).

Czas działania algorytmu jest zdominowany przez wypisywanie wyniku, które w oczywisty sposób ma złożoność $O(n)$.

F

Fotoradary

Autor zadania: Tomasz Idziaszek
Opis rozwiązania: Tomasz Idziaszek
Dostępna pamięć: 256 MB
`https://oi.edu.pl/pl/archive/amppz/2013/fot`

Burmistrz Bajtogrodu zamierza postawić w mieście fotoradary. W Bajtogrodzie jest n skrzyżowań, ponumerowanych liczbami od 1 do n, oraz $n-1$ dwukierunkowych odcinków dróg. Każdy z tych odcinków łączy dwa skrzyżowania. Sieć dróg umożliwia dotarcie z każdego skrzyżowania do dowolnego innego.

Fotoradary mają być umieszczane na skrzyżowaniach (na każdym co najwyżej jeden), przy czym burmistrz chciałby postawić ich jak najwięcej. Aby nie denerwować zbytnio bajtogrodzkich kierowców, przyjął on, że na każdej trasie, która przebiega po drogach Bajtogrodu i nie odwiedza wielokrotnie żadnego skrzyżowania, może stać co najwyżej k fotoradarów (włączając skrzyżowania na końcach trasy). Twoim zadaniem jest napisanie programu, który wyznaczy, gdzie należy postawić fotoradary.

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite n i k ($1 \leqslant n, k \leqslant 1\,000\,000$), oznaczające liczbę skrzyżowań w Bajtogrodzie i maksymalną liczbę fotoradarów, które mogą znaleźć się na pojedynczej trasie. Kolejne $n-1$ wierszy opisuje sieć dróg Bajtogrodu: w i-tym z tych wierszy znajdują się dwie liczby całkowite a_i i b_i ($1 \leqslant a_i, b_i \leqslant n$), oznaczające, że istnieje dwukierunkowy odcinek drogi łączący skrzyżowania o numerach a_i oraz b_i.

Wyjście

W pierwszym wierszu wyjścia należy wypisać liczbę m oznaczającą maksymalną liczbę fotoradarów, które można ustawić w Bajtogrodzie. W drugim wierszu należy wypisać ciąg m liczb, będących numerami skrzyżowań, na których należy ustawić fotoradary. Jeśli istnieje wiele rozwiązań, Twój program może wypisać dowolne z nich.

Przykład

Dla danych wejściowych:

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

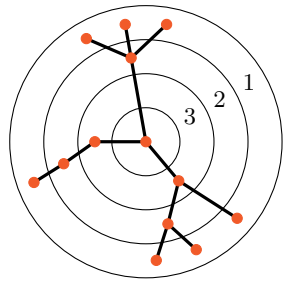
jednym z poprawnych wyników jest:

```
3
1 2 4
```

Rozwiązanie

Jeśli przetłumaczymy treść zadania na język teorii grafów, otrzymamy następujący problem: w danym drzewie o n wierzchołkach mamy zaznaczyć jak najwięcej wierzchołków tak, by na każdej ścieżce było zaznaczonych co najwyżej k wierzchołków. Zauważmy, że w naszych rozważaniach możemy ograniczyć się do ścieżek maksymalnych, tzn. niedających się przedłużyć w żadną stronę. Faktycznie, każda inna ścieżka rozszerza się do jakiejś ścieżki maksymalnej, więc jeśli ta ścieżka maksymalna ma co najwyżej k zaznaczonych wierzchołków, to tym bardziej ścieżka nieprzedłużona. Każda ścieżka maksymalna zaczyna się i kończy w wierzchołku, który jest liściem drzewa.

W przypadku pojedynczej ścieżki nie ma znaczenia, które k wierzchołków ścieżki zaznaczymy, ale już dla dwóch ścieżek współdzielących kilka krawędzi widać, że bardziej opłaca się zaznaczać wierzchołki, które należą tylko do jednej ze ścieżek, niż te, które leżą na obu z nich. Mówiąc nieformalnie, lepiej zaznaczać wierzchołki, które leżą bliżej liści drzewa, bo potencjalnie należą do mniejszej liczby ścieżek niż wierzchołki bliżej „środka" drzewa i dzięki temu będziemy mogli zaznaczyć ich więcej.



Rysunek 1. Podział wierzchołków drzewa na warstwy.

Ta obserwacja prowadzi nas do całkiem prostego rozwiązania zachłannego. Dzielimy wierzchołki na warstwy (rysunek 1). W pierwszej warstwie znajdują się liście drzewa. W drugiej warstwie znajdują się te wierzchołki, które stają się liśćmi po usunięciu z drzewa wszystkich wierzchołków z pierwszej warstwy. Ogólnie, w i-tej warstwie znajdują się wierzchołki, które stają się liśćmi po usunięciu wierzchołków z warstw o numerach $1, \ldots, i-1$.

Jeśli k jest parzyste, zaznaczamy wszystkie wierzchołki należące do $\frac{k}{2}$ warstw o najniższych numerach (oczywiście, jeśli jest mniej niż $\frac{k}{2}$ warstw, to zaznaczamy wszystkie wierzchołki w drzewie). Jeśli k jest nieparzyste, zaznaczamy wierzchołki z $\lfloor \frac{k}{2} \rfloor$ warstw oraz dowolny inny wierzchołek (jeśli jakiś pozostał).

Rozważmy teraz dowolną ścieżkę od liścia do liścia i wypiszmy ciąg numerów warstw, do których należą kolejne wierzchołki na tej ścieżce. Ponieważ każdy wierzchołek może sąsiadować z co najwyżej jednym wierzchołkiem z warstwy o wyższym numerze (i oczywiście nie sąsiaduje z innymi wierzchołkami z tej samej warstwy), więc do pewnego momentu ciąg ten jest ściśle rosnący, a dalej jest ściśle malejący. Wynika z tego, że na dowolnej ścieżce leżą co najwyżej dwa wierzchołki z każdej warstwy. Tak więc dla parzystego k na ścieżce zaznaczonych jest co najwyżej

$2 \cdot \frac{k}{2} = k$ wierzchołków w pierwszych $\frac{k}{2}$ warstwach, a dla k nieparzystego w pierwszych $\lfloor \frac{k}{2} \rfloor$ warstwach mamy zaznaczone $2 \cdot \lfloor \frac{k}{2} \rfloor = k-1$ wierzchołków (zatem możemy pozwolić sobie na zaznaczenie jeszcze jednego). Tym samym powyższy algorytm zaznacza poprawny zbiór wierzchołków.

Nieco później udowodnimy także, że zbiór ten jest najliczniejszy. Na razie zastanówmy się, jak zaimplementować powyższy algorytm. Na początek wyznaczamy stopień każdego wierzchołka w drzewie, a następnie tworzymy kolejkę Q_1, do której wstawiamy wszystkie wierzchołki o stopniu 1 (czyli liście drzewa). Dalej wykonujemy $\lfloor \frac{k}{2} \rfloor$ faz. W i-tej fazie zaznaczamy wierzchołki należące do i-tej warstwy, czyli znajdujące się w kolejce Q_i. Następnie usuwamy je z drzewa, zmniejszając stopień każdego wierzchołka v sąsiadującego z pewnym wierzchołkiem w kolejce Q_i, przy czym jeśli zmniejszyliśmy stopień v do 1, to wstawiamy wierzchołek v do kolejki Q_{i+1}, która zawiera wierzchołki do przetworzenia w kolejnej fazie.

Na końcu jeśli k jest nieparzyste i pozostał jakiś niezaznaczony wierzchołek, to zaznaczamy dowolny z nich. Otrzymujemy algorytm działający w czasie $O(n)$.

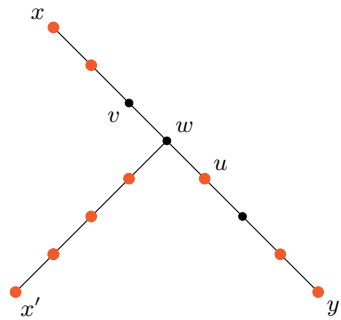
Dowód poprawności

Pozostaje nam pokazać poprawność algorytmu zachłannego. Ustalmy k parzyste i oznaczmy przez K zbiór wierzchołków z pierwszych $\frac{k}{2}$ warstw. Weźmy dowolne rozwiązanie optymalne i załóżmy, że pewien wierzchołek $v \in K$ nie został w tym rozwiązaniu zaznaczony. Jeśli jest więcej takich wierzchołków, to weźmy ten z warstwy o najmniejszym numerze (i oznaczmy numer tej warstwy przez ℓ). Musi też istnieć jakiś zaznaczony wierzchołek w warstwie wyższej niż ℓ, gdyż w przeciwnym razie rozwiązanie miałoby za mało zaznaczonych wierzchołków, aby być optymalnym. Rozważmy więc pewien zaznaczony wierzchołek u, który jest w warstwie wyższej niż ℓ i taki, że pomiędzy v i u nie ma zaznaczonych wierzchołków. Przy tych oznaczeniach wykażemy, że zachodzi następujący fakt.

Fakt 1. Gdy usuniemy ze zbioru zaznaczonych wierzchołków u i dodamy do niego v, dostaniemy poprawne rozwiązanie.

Wprowadźmy pomocnicze oznaczenie. Dla dowolnych dwóch wierzchołków a i b oznaczmy przez P_{ab} ścieżkę między wierzchołkami a i b. Rozważmy dwa dowolne liście x i x'. Pokażemy, że po zamianie (tj. po usunięciu zaznaczenia z u i zaznaczeniu v) ścieżka $P_{xx'}$ zawiera nadal nie więcej niż k zaznaczonych wierzchołków. Jeśli ścieżka $P_{xx'}$ nie przechodzi przez wierzchołek v lub przechodzi przez wierzchołek u, to zamiana nie może spowodować wzrostu liczby zaznaczonych wierzchołków na ścieżce $P_{xx'}$, zatem w tym przypadku dowód natychmiast się kończy. Przyjmijmy zatem, że $P_{xx'}$ przechodzi przez wierzchołek v, ale nie przechodzi przez wierzchołek u (rysunek 2). Rozważmy dowolną ścieżkę P_{xy} od liścia x do liścia y, która zawiera oba wierzchołki u i v. Taka ścieżka nie musi istnieć, ale wtedy istnieje ścieżka z x' do y, która zawiera wierzchołki u i v (dlaczego?), więc możemy wierzchołki x i x' zamienić rolami. Niech w będzie ostatnim wspólnym wierzchołkiem ścieżek $P_{xx'}$ i P_{xy}. Wówczas w leży na ścieżce pomiędzy v a u.

Dalszy dowód przeprowadzimy następująco. Pokażemy, że ścieżka $P_{xx'}$ po zamianie zawiera co najwyżej tyle zaznaczonych wierzchołków, co ścieżka $P_{x'y}$ przed zamianą. Wiemy, że przed zamianą na każdej ścieżce leżało co najwyżej k zaznaczonych wierzchołków, zatem wystarczy to do zakończenia dowodu.



Rysunek 2. Ilustracja dowodu poprawności ($k = 6$); zaznaczone wierzchołki narysowano w kolorze.

Ścieżki $P_{xx'}$ i $P_{x'y}$ mają wspólny kawałek: ścieżkę $P_{x'w}$, na którą zmiana w żaden sposób nie wpływa. Wystarczy więc zająć się pozostałymi kawałkami tych ścieżek i pokazać, że na ścieżce P_{wx} jest co najwyżej tyle zaznaczonych wierzchołków, co na ścieżce P_{wy} przed zamianą.

Ścieżka P_{wy} zawierała przed zamianą co najmniej $\ell - 1$ zaznaczonych wierzchołków na jej końcu (bo założyliśmy, że warstwa o najmniejszym numerze z niezaznaczonym wierzchołkiem to ℓ) plus dodatkowo wierzchołek u. Z kolei ścieżka P_{wx} po zamianie zawiera dokładnie ℓ zaznaczonych wierzchołków (przypomnijmy, że pomiędzy v a u nie ma zaznaczonych wierzchołków). To kończy dowód faktu.

Korzystając wielokrotnie z udowodnionego faktu, pokazujemy, że istnieje rozwiązanie optymalne, które zaznacza wszystkie wierzchołki z K, zatem zbiór wierzchołków generowany przez nasz algorytm jest najliczniejszy. Dowód dla k nieparzystego zostawiamy jako ćwiczenie dla czytelnika.

G

Gra w kulki

★★★★

Autor zadania: Jakub Radoszewski
Opis rozwiązania: Jakub Radoszewski
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2013/gra`

Bajtek i Bitek postanowili zagrać w kulki. W urnie znajduje się parzysta liczba kulek. Na każdej z nich zapisana jest dokładnie jedna cyfra. Zasady gry są bardzo proste: gracze na przemian wyjmują z urny po jednej, losowo wybranej kulce. Gra kończy się, gdy w urnie nie ma już żadnych kulek. Wygrywa gracz, który zgromadził zestaw kulek o większym iloczynie cyfr.

Chłopcy bardzo polubili tę grę. Obaj są bardzo ambitni i naprawdę lubią wygrywać, więc jedyną sytuacją końcową, która nie zadowala żadnego z nich, jest remis. Bajtek i Bitek chcieliby za wszelką cenę uniknąć takich rozstrzygnięć. Napisz program, który sprawdzi, czy dla zadanej początkowej zawartości urny gra może zakończyć się remisem.

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą t ($1 \leqslant t \leqslant 1000$), oznaczającą liczbę przypadków testowych do rozważenia.

Każdy z kolejnych t wierszy zawiera po dziesięć nieujemnych liczb całkowitych $k_0, \ldots, k_9$ ($0 \leqslant k_i \leqslant 10^{15}$), gdzie k_i oznacza liczbę kulek, na których zapisana jest cyfra i. Suma liczb k_i w każdym przypadku testowym jest parzysta i dodatnia.

Wyjście

Twój program powinien wypisać t wierszy z odpowiedziami dla poszczególnych przypadków testowych. Odpowiedzią dla jednego przypadku testowego jest słowo `TAK`, jeśli rozważana gra może zakończyć się remisem, lub `NIE`, w przeciwnym przypadku.

Przykład

Dla danych wejściowych:

```
5
0 1 0 1 1 4 1 0 5 1
0 1 1 0 3 0 0 0 0 3
1 1 0 4 0 0 2 0 0 2
100000 100000 100000 100000 100000 100000 100000 100000 100000 100000
0 99999 99999 100000 100000 100000 100000 100000 100000 100000
```

poprawnym wynikiem jest:

```
TAK
NIE
NIE
TAK
NIE
```

Rozwiązanie

W rozwiązaniu najlepiej zacząć od rozpatrzenia cyfr 0, 5 i 7. Cyfra 0 to, rzecz jasna, przypadek szczególny. Jeśli są co najmniej dwie kulki z zerami, to na pewno możemy odpowiedzieć `TAK`. Jeśli tylko jedna, od razu odpowiadamy `NIE`. W przeciwnym razie po prostu nie ma żadnych zer. Teraz rozpatrujemy piątki i siódemki. Wiadomo, że jeśli kulek odpowiadających którejś z tych cyfr jest nieparzyście wiele, to odpowiedź brzmi `NIE`. W przeciwnym razie, dążąc do konfiguracji remisowej, kulki z tymi cyframi musimy rozdzielić równo pomiędzy graczy i możemy się już nimi dalej nie zajmować.

Wszystkie pozostałe cyfry to wielokrotności 2 i 3. Oznaczmy przez a_i, dla $i \in \{1,2,3,4,6,8,9\}$, liczby kulek poszczególnych typów wziętych przez pierwszego gracza, podobnie oznaczmy przez $b_i = k_i - a_i$ liczby kulek wziętych przez drugiego gracza. Rozwiązanie wzorcowe opiera się na dosyć naturalnej hipotezie, która pozwala nam znacząco ograniczyć obszar, w którym szukamy rozwiązania prowadzącego do remisu.

Hipoteza 1. Jeśli istnieje rozwiązanie prowadzące do remisu, to istnieje takie rozwiązanie, w którym dla każdego i spełniony jest warunek

$$|a_i - b_i| \leqslant 6. \tag{1}$$

Przyjmijmy teraz, że hipoteza ta jest prawdziwa. Wówczas dopóki istnieje więcej niż 6 kulek z pewną cyfrą i, możemy wyrzucać po dwie takie kulki. Ostatecznie na mocy hipotezy 1, otrzymamy równoważny problem, w którym każda cyfra zapisana jest na co najwyżej 6 kulkach. Wystarczy teraz rozważyć wszystkie potencjalne rozwiązania spełniające warunek (1) i dla każdego z nich sprawdzić, czy są spełnione warunki remisu w grze, tj.

$$\begin{cases} a_1 + a_2 + a_3 + a_4 + a_6 + a_8 + a_9 = b_1 + b_2 + b_3 + b_4 + b_6 + b_8 + b_9 \\ a_2 + 2a_4 + a_6 + 3a_8 = b_2 + 2b_4 + b_6 + 3b_8 \\ a_3 + a_6 + 2a_9 = b_3 + b_6 + 2b_9 \end{cases}$$

Pierwszy warunek sprawdza, czy każdy z graczy otrzymał taką samą liczbę kulek, a dwa kolejne odpowiadają za sprawdzenie, czy w iloczynach cyfr na kulkach graczy dwójka i trójka występują w takich samych potęgach. Do przeglądania potencjalnych rozwiązań spełniających warunek (1) używamy przeszukiwania z nawrotami (*backtracking*). Dla każdego i mamy co najwyżej 7 możliwości wyboru wartości a_i oraz b_i, co daje łącznie co najwyżej $7^7 \approx 800\,000$ możliwości. Dodatkowo do rozwiązania wzorcowego dodane jest jeszcze jedno proste usprawnienie, które istotnie ogranicza czas działania: badamy na wstępie, czy łączny wykładnik 2 i 3 we wszystkich kulkach jest parzysty, tj. czy obie liczby $k_2 + 2k_4 + k_6 + 3k_8$ i $k_3 + k_6 + 2k_9$ są parzyste, a jeśli nie, to od razu odpowiadamy `NIE`.

Dowód poprawności

Rozwiązanie wzorcowe oparliśmy na dosyć śmiałej hipotezie. Przyszła pora, aby uzasadnić jej prawdziwość. Postąpimy trochę nietypowo i przeprowadzimy dowód wspomagany komputerowo.

Zaczniemy od zmiany sposobu opisu rozwiązań. Potencjalne rozwiązania, dotychczas zapisywane za pomocą ciągów liczb (a_i) oraz (b_i), będziemy odtąd przedstawiać w postaci ciągu liczb całkowitych (x_i) określonego jako $x_i = a_i - b_i$. Dane w zadaniu $(k_1, k_2, k_3, k_4, k_6, k_8, k_9)$ będziemy nazywać *konfiguracją*, a siódemkę liczb całkowitych $(x_1, x_2, x_3, x_4, x_6, x_8, x_9)$ reprezentującą potencjalne rozwiązanie będziemy nazywać *układem*.

Powiemy, że układ jest *remisowy*, jeśli

$$\begin{cases} x_1 + x_2 + x_3 + x_4 + x_6 + x_8 + x_9 = 0 \\ x_2 + 2x_4 + x_6 + 3x_8 = 0 \\ x_3 + x_6 + 2x_9 = 0 \end{cases}$$

Powiemy, że konfiguracja (k_i) *dopuszcza* układ (x_i), jeśli dla każdego i zachodzi $|x_i| \leqslant k_i$ oraz $x_i \equiv k_i \pmod 2$. Wreszcie układ nazwiemy *6-ograniczonym*, jeśli zgodnie z hipotezą dla każdego i zachodzi $|x_i| \leqslant 6$. Nasze całe uzasadnienie sprowadzi się do dowodu następującego twierdzenia, z którego łatwo wynika prawdziwość hipotezy 1.

Twierdzenie 1. Dla każdej konfiguracji $(k_1, k_2, k_3, k_4, k_6, k_8, k_9)$, która ma rozwiązanie, istnieje 6-ograniczony, remisowy układ $(x_1, x_2, x_3, x_4, x_6, x_8, x_9)$ dopuszczany przez tę konfigurację.

Łatwo sprawdzić, że jeśli powyższe twierdzenie jest spełnione, to dla dowolnej konfiguracji posiadającej rozwiązanie, przyjmując

$$a_i = \frac{k_i + x_i}{2} \quad \text{oraz} \quad b_i = \frac{k_i - x_i}{2},$$

otrzymamy rozwiązanie remisowe spełniające warunek (1). W takiej sytuacji będziemy także mówić, że sam układ (x_i) jest *rozwiązaniem* dla konfiguracji (k_i).

Tak sformułowanego twierdzenia nie da się jeszcze zweryfikować za pomocą programu komputerowego. Aby móc to zrobić, ograniczymy liczbę konfiguracji, które trzeba sprawdzić. Powiemy, że konfiguracja (k_i) jest *generowana* przez konfigurację (k'_i), jeśli (k_i) dopuszcza (k'_i), ponadto dla każdego i zachodzi $k'_i \leqslant 6$ oraz obie konfiguracje są zgodne na elementach nie większych niż 4 (czyli jeśli dla pewnego i zachodzi $k_i \leqslant 4$ lub $k'_i \leqslant 4$, to $k_i = k'_i$). Skąd taka definicja? Rozważmy konfigurację (k_i) i następnie, dopóki istnieje ponad 6 kulek pewnego rodzaju, usuwajmy dwie kulki tego rodzaju. Wtedy z konfiguracji (k_i) dostaniemy właśnie konfigurację (k'_i), która generuje (k_i). W ten sposób uzasadniliśmy, że:

Obserwacja 1. Każda konfiguracja (k_i) jest generowana przez jakąś konfigurację (k'_i).

Oczywiście, jeśli konfiguracja generująca ma rozwiązanie, to musi to być rozwiązanie 6-ograniczone, czyli spełniające warunki twierdzenia 1. Zamiast badać wszystkie konfiguracje, skupimy się teraz na badaniu konfiguracji 6-ograniczonych. Chcielibyśmy powiedzieć, że jeśli jedna konfiguracja generuje drugą, to obie te konfiguracje są równoważne w sensie rozwiązań, a zatem że jeśli konfiguracja generująca ma rozwiązanie, to konfiguracja generowana także je ma, a jeśli konfiguracja

generująca nie ma żadnego rozwiązania, to to samo dotyczy również konfiguracji generowanej. Do sprawy podejdziemy jednak stopniowo.

Na początek zajmijmy się prostym przypadkiem. Powiemy, że konfiguracja (k_i) jest *ewidentnie zła*, jeśli zachodzi co najmniej jeden z poniższych warunków:

(a) suma k_i jest nieparzysta,

(b) łączny wykładnik 2 lub 3 we wszystkich kulkach jest nieparzysty, tj. co najmniej jedna z liczb $k_2 + 2k_4 + k_6 + 3k_8$ lub $k_3 + k_6 + 2k_9$ jest nieparzysta,

(c) k_4 jest nieparzyste i $k_2 = k_6 = k_8 = 0$ (wykładnik 2 pochodzi jedynie z kulek z czwórką, dlatego nie da się go podzielić po równo),

(d) k_9 jest nieparzyste i $k_3 = k_6 = 0$ (wykładnik 3 pochodzi jedynie z kulek z dziewiątką, dlatego nie da się go podzielić po równo),

(e) $k_2 + k_4 + k_6 = 1$ (po rozdzieleniu kulek z ósemką wykładniki 2 u obydwu graczy są podzielne przez 3, zatem nie da się dobrze przydzielić dodatkowej kulki z dwójką, czwórką lub szóstką).

Oczywiście jeśli konfiguracja jest ewidentnie zła, to nie istnieje dla niej rozwiązanie.

Lemat 1. Załóżmy, że konfiguracja (k_i) jest generowana przez konfigurację (k'_i). Jeśli konfiguracja (k'_i) ma rozwiązanie, jest ono również rozwiązaniem dla konfiguracji (k_i). Jeśli natomiast konfiguracja (k'_i) jest ewidentnie zła, to konfiguracja (k_i) również jest ewidentnie zła.

Dowód: Pierwsza implikacja: Załóżmy, że układ (x_i) jest rozwiązaniem dla konfiguracji (k'_i), czyli układem remisowym dopuszczanym przez (k'_i). Ponieważ dla każdego i zachodzi zawsze $k_i \geqslant k'_i$ oraz są to liczby tej samej parzystości, to (x_i) jest również układem remisowym dopuszczanym przez konfigurację (k_i), czyli rozwiązaniem dla (k_i).

Druga implikacja: Wystarczy zauważyć, że jeśli (k'_i) spełnia któryś z warunków (a)-(e) z definicji ewidentnie złej konfiguracji, to (k_i) spełnia ten sam warunek. W przypadku warunków (a) oraz (b) wynika to z faktu, że k_i ma zawsze tę samą parzystość co k'_i. Natomiast w przypadku pozostałych warunków musimy skorzystać także z równości tych konfiguracji na elementach o małych wartościach. □

Potrafimy już rozstrzygnąć wynik gry dla konfiguracji, które mają rozwiązanie lub są ewidentnie złe. Gdyby każda konfiguracja należała do jednej z tych dwóch kategorii, dowód byłby zakończony. Sprawdźmy więc, czy istnieje konfiguracja, która nie ma rozwiązania i nie jest ewidentnie zła. W tym celu napiszemy program komputerowy.

Wygenerujmy najpierw wszystkie 6-ograniczone układy remisowe (x_i). Jak się okazuje, takich układów jest 4497. Następnie dla wszystkich 6-ograniczonych konfiguracji będziemy sprawdzać, czy każda z nich jest albo ewidentnie zła, albo posiada rozwiązanie odpowiadające jednemu spośród wygenerowanych układów. Zauważmy, że w tym drugim kroku będziemy tylko sprawdzać, czy konfiguracja dopuszcza jeden spośród układów, a w tym celu nie będą już nas interesować znaki liczb x_i. Możemy więc wszystkie elementy układów wziąć z dokładnością do wartości bezwzględnej i usunąć powtórzenia. W ten sposób zostaje nam tylko 2216 różnych układów, których będziemy używać, co około dwukrotnie przyspieszy obliczenia.

Po uruchomieniu programu weryfikującego znajdujemy niestety kilkaset *kłopotliwych* konfiguracji, które nie spełniają żadnego z warunków: ani nie są ewidentnie złe, ani też nie znajdujemy dla nich rozwiązania. Wykażemy, że konfiguracje generowane przez te kłopotliwe konfiguracje również nie posiadają rozwiązań. Z pomocą przyjdzie nam algorytm eliminacji Gaussa.

Ustalmy jedną taką kłopotliwą konfigurację (k'_i). Chcemy teraz wykazać, że żadna konfiguracja (k_i) generowana przez (k'_i) nie ma rozwiązania. Rozważmy więc hipotetyczny układ remisowy (x_i) dopuszczany przez (k_i). Zauważmy, że definicja (x_i) wymusza szereg warunków, które ten układ musi spełniać. Jak się za chwilę okaże, warunki te są w większości przypadków sprzeczne ze sobą.

Zacznijmy od zapisania warunków na (x_i) w postaci układu równań. Przede wszystkim, (x_i) musi spełniać równości opisujące układ remisowy. Ponadto dla wszystkich i, takich że $k'_i \leqslant 4$, mamy $x_i \in \{-k'_i, \ldots, k'_i\}$, a parzystość x_i jest taka sama jak parzystość k'_i. Tu mamy pewną dowolność w wyborze x_i, dlatego wygenerujemy wszystkie możliwe tego typu układy równań (odpowiadające wszystkim możliwym kombinacjom wartości x_i). Następnie spróbujemy rozwiązać każdy z tych układów (w liczbach rzeczywistych). Jeśli każdy z układów nie będzie miał rozwiązań albo będzie miał dokładnie jedno rozwiązanie, w którym któraś współrzędna x_i jest niecałkowita lub też całkowita, ale o innej parzystości niż odpowiadające jej k'_i, to będzie jasne, że żadna konfiguracja (k_i) generowana przez (k'_i) nie posiada rozwiązań.

Przedstawmy to na przykładzie. Oto jedna z kłopotliwych konfiguracji, która ani nie posiada rozwiązań, ani nie jest ewidentnie zła:

$$(k'_1 = 0,\ k'_2 = 6,\ k'_3 = 1,\ k'_4 = 0,\ k'_6 = 1,\ k'_8 = 5,\ k'_9 = 3).$$

Tworzymy dla niej układy równań postaci

$$\begin{cases} x_1 + x_2 + x_3 + x_4 + x_6 + x_8 + x_9 = 0 \\ x_2 + 2x_4 + x_6 + 3x_8 = 0 \\ x_3 + x_6 + 2x_9 = 0 \\ x_1 = 0 \\ x_3 = \pm 1 \\ x_4 = 0 \\ x_6 = \pm 1 \\ x_9 = \pm 1 \text{ lub } \pm 3 \end{cases}$$

Łącznie otrzymamy $2 \cdot 2 \cdot 4 = 16$ układów równań. Większość z nich nie posiada rozwiązań. Z trzeciego równania i ograniczeń na zmienne x_3, x_6 i x_9 wynika, że tego typu układ ma rozwiązanie tylko wtedy, gdy $x_3 = x_6 = 1$ i $x_9 = -1$ lub $x_3 = x_6 = -1$ i $x_9 = 1$. W pierwszym z tych przypadków dwa pierwsze równania po podstawieniu znanych wartości przyjmują postać

$$\begin{cases} 0 + x_2 + 1 + 0 + 1 + x_8 - 1 = 0 \\ x_2 + 2 \cdot 0 + 1 + 3x_8 = 0 \end{cases}$$

czyli

$$\begin{cases} x_2 + x_8 = -1 \\ x_2 + 3x_8 = -1 \end{cases}$$

Jedynym rozwiązaniem tego układu jest $x_2 = -1$, $x_8 = 0$. Nie jest to jednak rozwiązanie dopuszczane przez początkową konfigurację (k'_i) (ani też przez żadną

inną konfigurację generowaną przez (k'_i)), gdyż parzystości x_2 i k'_2 nie zgadzają się; podobnie z x_8 i k'_8.

Za pomocą tak skonstruowanego programu komputerowego możemy przejrzeć wszystkie kłopotliwe konfiguracje. Okazuje się, że wśród nich istnieją dokładnie dwie złośliwe konfiguracje, które przechodzą przez całe powyższe sito i metoda układów równań nie jest dla nich skuteczna. Są to:

$$(k'_1 = 5,\ k'_2 = 0,\ k'_3 = 0,\ k'_4 = 5,\ k'_6 = 6,\ k'_8 = 0,\ k'_9 = 6)$$

oraz

$$(k'_1 = 5,\ k'_2 = 0,\ k'_3 = 0,\ k'_4 = 6,\ k'_6 = 6,\ k'_8 = 0,\ k'_9 = 5).$$

Dla każdej z nich otrzymujemy układ równań posiadający nieskończenie wiele rozwiązań. Ostateczne sprawdzenie dla tych dwóch ostatnich konfiguracji wykonamy „ręcznie".

Zajmijmy się pierwszą z nich (rozumowanie dla drugiej jest bardzo podobne). Otóż wśród konfiguracji generowanych przez (k'_i) są tylko takie, w których liczba czwórek jest nieparzysta, a liczba szóstek i dziewiątek parzysta. Załóżmy, że jakaś taka konfiguracja miałaby rozwiązanie $(x_1, x_2, x_3, x_4, x_6, x_8, x_9)$. Wtedy x_9 jest parzyste (gdyż k'_9 jest parzyste). Stąd (ze względu na wykładnik 3) x_6 musi być podzielne przez 4, czyli (ze względu na wykładnik 2) x_4 musi być parzyste. To jednak nie jest możliwe, bo k'_4 jest nieparzyste. Ta sprzeczność kończy cały, wspomagany komputerowo, dowód.

Heros

Autor zadania: Tomasz Idziaszek
Opis rozwiązania: Jakub Łącki
Dostępna pamięć: 256 MB
`https://oi.edu.pl/pl/archive/amppz/2013/her`

Bajtyzeusz, najsłynniejszy bajtocki heros, kolejny raz zwycięsko wyszedł z bitwy. Podczas gdy załoga jego statku ładuje na pokład zdobyte kosztowności, Bajtyzeusz w swojej kajucie planuje drogę powrotną do rodzinnej Bitaki. Nie jest to łatwe zadanie. Wielu bogów zazdrości Bajtyzeuszowi popularności wśród bajtockiego ludu i chętnie utarłoby mu nosa. Na szczęście niektórzy z nich patrzą na niego przychylnie, a zwłaszcza bogini Bajtena. I to ona właśnie zesłała wczoraj w nocy Bajtyzeuszowi sen, w którym ostrzegła go o grożących mu niebezpieczeństwach.

Na Morzu Bajtockim znajduje się n wysp, które wygodnie nam będzie ponumerować liczbami od 1 do n. Aktualnie statek Bajtyzeusza znajduje się na wyspie 1, zaś celem podróży jest Bitaka — wyspa n. Niektóre pary wysp połączone są *jednokierunkowymi* szlakami morskimi, przy czym każda z wysp jest początkiem co najwyżej 10 szlaków. Szlaki numerujemy liczbami od 1 do m; i-ty szlak prowadzi z wyspy a_i na wyspę b_i, a jego pokonanie wymaga dokładnie d_i dni. Jeśli więc statek wyruszy i-tym szlakiem z wyspy początkowej a_i o świcie dnia j, to na wyspę docelową b_i dotrze o świcie dnia $j + d_i$. Na każdej z wysp statek może czekać dowolnie długo przed wyruszeniem w drogę. Jednak zanim dotrze do kolejnej wyspy, nie może zboczyć z raz obranego szlaku ani płynąć dłużej niż wymaga tego szlak. Z wyspy 1 Bajtyzeusz może wyruszyć najwcześniej o świcie pierwszego dnia.

Ostrzeżenie Bajteny było bardzo konkretne. Podała ona Bajtyzeuszowi dokładną listę p pułapek, które przygotowali bogowie. Każda z nich znajduje się na pewnej wyspie i jest aktywna przez pewien przedział czasu. Konkretniej mówiąc, i-ta pułapka znajduje się na wyspie w_i i jest aktywna cały czas od dnia s_i do dnia k_i włącznie. Pułapki są naprawdę niebezpieczne — jeżeli statek Bajtyzeusza znajdzie się na wyspie z aktywną pułapką, to żaden członek załogi nie ujdzie z życiem. Szczęśliwie rodzinna Bitaka jest wolna od pułapek, a na wyspie 1 żadna pułapka nie jest aktywna pierwszego dnia.

Oczywiste jest, że Bajtyzeusz chce tak zaplanować drogę do domu, aby ominąć wszystkie pułapki. Zastanawia się jednak, jak bardzo wydłuży to czas podróży. Pomóż mu i wyznacz minimalną liczbę dni potrzebnych na bezpieczny powrót do Bitaki.

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite n i m ($2 \leqslant n \leqslant 100\,000$, $1 \leqslant m \leqslant 1\,000\,000$), oznaczające liczbę wysp i liczbę szlaków na morzu. Kolejne m wierszy opisuje szlaki: w i-tym z nich znajdują się trzy liczby całkowite a_i, b_i, d_i ($1 \leqslant a_i, b_i \leqslant n$, $a_i \neq b_i$, $1 \leqslant d_i \leqslant 10^9$), oznaczające, że i-ty szlak prowadzi z wyspy a_i na wyspę b_i i pokonanie go trwa d_i dni. Wszystkie szlaki są jednokierunkowe. Na każdej wyspie zaczyna się co najwyżej 10 szlaków.

Następny wiersz zawiera liczbę całkowitą p ($0 \leqslant p \leqslant 100\,000$), oznaczającą liczbę pułapek. Kolejne p wierszy opisuje pułapki: w i-tym z nich znajdują się trzy liczby całkowite w_i, s_i, k_i ($1 \leqslant w_i < n$, $1 \leqslant s_i \leqslant k_i \leqslant 10^9$), oznaczające, że i-ta pułapka znajduje się na wyspie w_i oraz jest aktywna od dnia s_i do dnia k_i włącznie. Jeśli $w_i = 1$, to $s_i > 1$.

Wyjście

Jeśli nie można tak zaplanować drogi, by ominąć wszystkie pułapki, w pierwszym i jedynym wierszu wyjścia należy wypisać słowo `NIE`. W przeciwnym wypadku należy wypisać liczbę całkowitą d, oznaczającą minimalną liczbę dni potrzebnych na odbycie podróży (tzn. statek dociera do Bitaki świtem dnia $d + 1$).

Przykład

Dla danych wejściowych:	poprawnym wynikiem jest:

`5 6`
`1 2 3`
`1 4 13`
`2 3 1`
`2 4 2`
`3 2 2`
`4 5 1`
`5`
`1 2 4`
`1 8 8`
`2 6 7`
`2 10 11`
`4 6 7`
	`10`

Wyjaśnienie przykładu: Bajtyzeusz wyrusza z wyspy 1 o świcie pierwszego dnia i dociera na wyspę 2 czwartego dnia. Tam czeka jeden dzień, a następnie wyrusza na wyspę 3 i po dotarciu na nią szóstego dnia od razu zawraca na wyspę 2, skąd ósmego dnia wypływa w stronę wyspy 4. Przybywa na nią dziesiątego dnia i ostatecznie jedenastego dnia dopływa do Bitaki.

Rozwiązanie

Tym razem zajmujemy się jeszcze jedną odmianą problemu szukania najkrótszej ścieżki w grafie. Nasze rozwiązanie oparte będzie na algorytmie Dijkstry, który w nieujemnie ważonym grafie skierowanym wyznacza długości najkrótszych ścieżek z ustalonego wierzchołka startowego.

Przypomnijmy zatem, jak działa algorytm Dijkstry. Dla każdego wierzchołka v utrzymujemy wartość $d[v]$, oznaczającą długość najkrótszej ścieżki z wierzchołka startowego do wierzchołka v, którą dotychczas udało nam się skonstruować (czyli najlepsze znane nam w danym momencie ograniczenie górne na długość najkrótszej ścieżki prowadzącej do v). Początkowo wartość $d[v]$ dla wierzchołka startowego

inicjujemy na 0, a dla pozostałych wierzchołków na ∞. Wierzchołki dzielimy na przetworzone i nieprzetworzone; na początku wszystkie wierzchołki uznajemy za nieprzetworzone. Następnie w każdym kroku znajdujemy nieprzetworzony wierzchołek u o najmniejszej wartości $d[u]$ i *relaksujemy* wychodzące z niego krawędzie. Relaksacja skierowanej krawędzi uw polega na próbie poprawienia aktualnej odległości $d[w]$ za pomocą sumy $d[u]$ oraz wagi krawędzi uw. Po zrelaksowaniu wszystkich krawędzi wychodzących z u wierzchołek ten uznajemy za przetworzony.

Jak przenieść to na nasze zadanie? Pierwsza myśl to zbudować graf, w którym wierzchołki odpowiadają wyspom, a skierowane krawędzie szlakom morskim. W grafie mielibyśmy wtedy n wierzchołków i m krawędzi. W zadaniu występuje jednak dodatkowa trudność, która sprawia, że na takim grafie nie możemy użyć algorytmu Dijkstry w niezmienionej postaci. Dla każdej wyspy mamy daną listę p przedziałów czasu, w których zabronione jest przebywanie na tej wyspie. W naturalny sposób wyznaczają one *przedziały dopuszczalne*, czyli maksymalne przedziały czasu, podczas których wolno nam przebywać na wyspie. Ponieważ mamy p informacji o zabronionych przedziałach czasu, na wszystkich wyspach mamy łącznie $O(n + p)$ przedziałów dopuszczalnych.

Musimy więc nieco inaczej skonstruować graf. Przyjmijmy, że każdy wierzchołek odpowiada jednemu przedziałowi dopuszczalnemu. Dla każdego takiego przedziału dopuszczalnego v (powiedzmy, że jest to przedział $[a, b]$ na wyspie s) chcemy wyznaczyć wartość $d[v]$, czyli najwcześniejszy moment $t \in [a, b]$, w którym jesteśmy w stanie dotrzeć na wyspę s. Jeśli w ogóle nie możemy znaleźć się na tej wyspie w przedziale czasu $[a, b]$, to przyjmujemy $t = \infty$. Zauważmy, że jeśli na wyspę s jesteśmy w stanie dotrzeć w momencie $t \in [a, b]$, to możemy także znaleźć się tam w dowolnym momencie $t' \in [t, b]$, gdyż możemy dotrzeć na wyspę s w chwili t i zaczekać na niej aż do chwili t'.

Na początku ustawiamy wszystkie wartości $d[v]$ na ∞, z wyjątkiem odległości do przedziału dopuszczalnego, który zawiera chwilę początkową i znajduje się na wyspie 1 — w tym przypadku wartość $d[v]$ ustawiamy na 0. Ponadto wszystkie przedziały dopuszczalne uznajemy za nieprzetworzone.

Teraz w każdym kroku znajdujemy nieprzetworzony przedział dopuszczalny v, któremu przypisana jest najmniejsza wartość $d[v]$. Powiedzmy, że jest to przedział $[a, b]$ na wyspie s. I tu dochodzimy do sedna problemu, bowiem główna trudność leży w relaksacji krawędzi. Ponieważ znajdujemy się na wyspie s, rozpatrujemy wszystkie krawędzie (odpowiadające szlakom morskim) wychodzące z wyspy s. Z każdej takiej krawędzi możemy skorzystać w dowolnym momencie między chwilą $t = d[v]$ a chwilą b. Zatem jeśli długość krawędzi to d, do docelowej wyspy dotrzeć możemy w dowolnym momencie między $t + d$ a $b + d$. Powinniśmy więc zrelaksować wszystkie przedziały dopuszczalne na docelowej wyspie, które mają niepuste przecięcie z przedziałem $[t + d, b + d]$. Jeśli będziemy trzymać takie przedziały posortowane po ich lewych końcach (możemy też sortować po prawych końcach; to bez różnicy, bo przedziały dopuszczalne są rozłączne), to możemy efektywnie przejrzeć jedynie te przedziały, które nas interesują.

Niestety to jeszcze nie wystarcza do otrzymania szybkiego rozwiązania. Może się zdarzyć następująca sytuacja. Na pewnej wyspie mamy p przedziałów dopuszczalnych. Do wyspy tej prowadzi n krawędzi i przy relaksacji każdej z nich przeglądać będziemy wszystkie p przedziałów. Zatem łącznie wykonamy co najmniej pracę proporcjonalną do iloczynu np, co przy ograniczeniach zadania może zająć zbyt wiele czasu.

Do rozwiązania tego problemu wystarczy nam jedna obserwacja. Wróćmy do opisywanej wcześniej sytuacji, w której do danej wyspy docelowej możemy dopłynąć w dowolnym momencie między $t + d$ a $b + d$. Przyjmijmy, że przedział $[t+d, b+d]$ przecina przedziały dopuszczalne $[a_1, b_1], \ldots, [a_k, b_k]$ na docelowej wyspie, takie że $a_1 \leqslant b_1 < a_2 \leqslant b_2 < \ldots < a_k \leqslant b_k$. Zauważmy teraz, że do każdego z przedziałów $[a_2, b_2], [a_3, b_3], \ldots, [a_k, b_k]$ (czyli do każdego poza $[a_1, b_1]$) możemy dotrzeć na samym początku jego istnienia. Innymi słowy, dla każdego $i = 2, \ldots, k$ do przedziału $[a_i, b_i]$ dotrzeć możemy w czasie a_i. Zatem po poprawieniu oszacowania odległości dla każdego takiego przedziału wiemy, że nigdy więcej nie musimy uwzględniać go przy relaksacjach krawędzi.

Na potrzeby analizy czasu działania rozbijmy relaksację na trzy kroki:

(1) znalezienie przedziału $[a_1, b_1]$ (czyli pierwszego przedziału przecinającego się z $[t + d, b + d]$),

(2) zrelaksowanie przedziału $[a_1, b_1]$,

(3) przejście po przedziałach $[a_2, b_2], \ldots, [a_k, b_k]$ i ich ostateczne zrelaksowanie.

Aby zaimplementować relaksację efektywnie, przedziały dopuszczalne każdej wyspy trzymamy np. w strukturze `set` z biblioteki STL, posortowane po lewych końcach. Dzięki temu krok pierwszy wykonać możemy w czasie $O(\log p)$. Czas działania drugiego kroku szacujemy przez $O(\log(n + p))$, bo może on wymagać operacji na kopcu zawierającym $O(n + p)$ elementów. Dla każdego przedziału dopuszczalnego na wyspie s, każdą krawędź wychodzącą z wyspy s zrelaksujemy raz. Szczęśliwie z każdej wyspy wychodzi stała liczba krawędzi (co najwyżej 10), dzięki czemu łącznie wykonamy $O(n + p)$ relaksacji krawędzi. Zatem łączny czas wykonywania kroków 1 i 2 w trakcie wszystkich relaksacji to $O((n + p) \log(n + p))$.

Zajmijmy się teraz trzecim krokiem. Rozpatrzenie jednego przedziału wymaga znalezienia kolejnego przedziału dopuszczalnego oraz (być może) wykonania operacji na kopcu o rozmiarze $O(n+p)$. To zajmuje czas $O(\log(n+p))$. Ponadto każdy z rozważanych przedziałów możemy usunąć ze struktury zawierającej przedziały do relaksacji, co wymaga czasu $O(\log p)$. W efekcie krok 3 wykonamy co najwyżej raz dla każdego przedziału docelowego, a zatem łączny czas wykonywania kroku 3 w trakcie wszystkich relaksacji to $O((n + p) \log(n + p))$.

Poza relaksacjami algorytm Dijkstry wykonuje jeszcze $O(n + p)$ operacji na kopcu, gdy znajduje nieprzetworzony przedział o minimalnym oszacowaniu odległości. Każda z nich zajmuje czas $O(\log(n + p))$. Wstępne wyznaczenie przedziałów dopuszczalnych to koszt $O(n + p \log p)$. Całe rozwiązanie działa więc w czasie $O((n + p) \log(n + p))$.

W tym momencie wystarczy zaimplementować opisany algorytm i rozwiązanie zadania gotowe. Sprawa nie jest jednak tak prosta, jak by się mogło wydawać. Napisanie programu to dobre ćwiczenie praktyki programowania: kod jest stosunkowo trudny i łatwo w nim o pomyłkę. Zapewne dlatego w trakcie zawodów z zadaniem *Heros* uporały się jedynie cztery zespoły.

I

Inżynieria genetyczna

Autor zadania: Tomasz Idziaszek
Opis rozwiązania: Jakub Łącki
Dostępna pamięć: 128 MB
https://oi.edu.pl/pl/archive/amppz/2013/inz

Bajtoccy paleoarcheologowie odkopali niedawno kilka bursztynów, w których zatopione są komary. Po przeanalizowaniu próbek owadów okazało się, że pochodzą one z jury, a więc prawdopodobnie miały styczność z wielkimi gadami, które wówczas dominowały na bajtockich lądach. To podsunęło genetykom oryginalny pomysł: spróbują odzyskać z krwi komarów materiał genetyczny bajtoraptora.

Genom bajtoraptora, tak jak u wszystkich bajtockich organizmów, jest ciągiem składającym się z pewnej liczby bajtokwasów. Rodzaje bajtokwasów oznaczamy dla uproszczenia liczbami naturalnymi. W genomie występuje redundancja — każdy rodzaj bajtokwasu jest k-krotnie powtórzony (w szczególności, długość każdego poprawnego genomu jest wielokrotnością k). Innymi słowy, jeśli podzielimy genom na bloki składające się z k kolejnych bajtokwasów, to każdy blok będzie zawierać bajtokwasy tego samego rodzaju.

Genetykom udało się wydzielić z krwi komara podejrzany łańcuch długości n składający się z bajtokwasów. Niestety łańcuch ten może nie być prawidłowym genomem — naukowcy podejrzewają, że mógł on zostać zanieczyszczony obcymi bajtokwasami. Chcą teraz sprawdzić swoją hipotezę i usunąć z tego ciągu jak najmniej bajtokwasów, tak aby powstał prawidłowy genom. W przypadku wielu równie dobrych możliwości, naukowców interesuje genom, który jest *najwcześniejszy* w porządku leksykograficznym*. Twoim zadaniem jest pomóc im w dokonaniu przełomowego odkrycia!

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite n i k ($1 \leqslant n \leqslant 1\,000\,000$, $2 \leqslant k \leqslant 1\,000\,000$), oznaczające długość wydzielonego łańcucha bajtokwasów i stopień redundancji poprawnego genomu. W drugim wierszu znajduje się ciąg n liczb całkowitych $g_1, \ldots, g_n$ ($1 \leqslant g_i \leqslant 1\,000\,000$), oznaczających rodzaje kolejnych bajtokwasów w łańcuchu.

Wyjście

Na wyjście należy wypisać dwa wiersze. Pierwszy z nich powinien zawierać liczbę m ($0 \leqslant m \leqslant n$) oznaczającą długość najdłuższego poprawnego genomu, który może powstać poprzez usunięcie niektórych bajtokwasów z podanego łańcucha.

W drugim wierszu należy wypisać ciąg m liczb oznaczających rodzaje kolejnych bajtokwasów w poprawnym genomie. Jeśli istnieje wiele rozwiązań, Twój

*Niech l_1 i l_2 to dwa różne ciągi równej długości, składające się z bajtokwasów. Aby stwierdzić, który z nich jest wcześniejszy w porządku leksykograficznym, należy znaleźć pierwszą pozycję, na której te ciągi się różnią. Wcześniejszy w porządku leksykograficznym jest ten ciąg, który na tej pozycji zawiera bajtokwas oznaczony mniejszą liczbą.

program powinien wypisać najmniejsze leksykograficznie. Jeśli $m = 0$ (tzn. genetykom nie udało się wydzielić żadnego niepustego poprawnego genomu), to drugi wiersz wyjścia powinien być pusty.

Przykład

Dla danych wejściowych:

```
16 3
3 2 3 1 3 1 1 2 4 2 1 1 2 2 2 2
```

poprawnym wynikiem jest:

```
9
1 1 1 2 2 2 2 2 2
```

Rozwiązanie

Na potrzeby opisu rozwiązania wygodnie będzie nam zapomnieć o archeologach i genomie bajtoraptora. Dany jest ciąg $g_1, \ldots, g_n$ składający się z n liczb całkowitych z zakresu od 1 do M oraz liczba całkowita k, gdzie $1 \leqslant k \leqslant n$. Ciąg nazywamy *poprawnym*, jeśli jego długość jest podzielna przez k, a po podzieleniu go na bloki składające się z k kolejnych wyrazów, wyrazy w ramach każdego bloku są sobie równe. Naszym celem jest usunięcie z wejściowego ciągu g minimalnej liczby wyrazów, tak aby powstał poprawny ciąg. Innymi słowy, chcemy w ciągu g znaleźć najdłuższy poprawny podciąg.

Samo znalezienie długości poszukiwanego podciągu jest dosyć proste. W tym celu możemy postępować w sposób zachłanny: przeglądamy kolejne wyrazy ciągu g i zliczamy, ile razy wystąpiła w nim każda z liczb. Kiedy tylko znajdziemy k wystąpień pewnej liczby x, to do wynikowego podciągu dopisujemy blok składający się z tych k wystąpień i kontynuujemy przeglądanie ciągu wejściowego (oczywiście wystąpienia poszczególnych liczb zliczamy znów od zera).

Trudność zadania leży jednak w znalezieniu najwcześniejszego leksykograficznie rozwiązania. Zgodnie z definicją porządku leksykograficznego powinniśmy przede wszystkim zapewnić, by pierwsza liczba (zatem i kolejne $k - 1$ liczb) wynikowego podciągu była jak najmniejsza. Następnie minimalizujemy liczby w drugim bloku i tak dalej. Postępujemy więc również, w pewnym sensie, zachłannie. Mimo to musimy się trochę napracować, by pierwotny algorytm zachłanny rozszerzyć o szukanie najwcześniejszego leksykograficznie rozwiązania.

Aby zilustrować tę trudność, rozważmy $k = 3$ i ciąg wejściowy

$$4\ 4\ 1\ 1\ 4\ 1\ 7\ 7\ 2\ 7.$$

W oryginalnym podejściu zachłannym, zaraz po napotkaniu trzeciego wystąpienia liczby 4, do rozwiązania dodamy $4, 4, 4$ i ostatecznie znajdziemy poprawny podciąg $4, 4, 4, 7, 7, 7$. Jednak w ciągu wejściowym znajduje się też podciąg $1, 1, 1, 7, 7, 7$, który również ma długość 6 i jest wcześniejszy leksykograficznie od znalezionego. Problem w tym, że pierwszy blok wynikowego podciągu budujemy od razu po znalezieniu trzeciego wystąpienia liczby 4, nie wiedząc, że już za chwilę w ciągu

wejściowym pojawi się trzecie wystąpienie jedynki. Z drugiej strony, jeśli nie zbudujemy pierwszego bloku rozwiązania najwcześniej, jak to możliwe, to może się zdarzyć, że otrzymamy rozwiązanie krótsze od optymalnego.

Problem wygląda może niełatwo, ale ma zaskakująco proste rozwiązanie. Przyjmijmy, że najdłuższy poprawny podciąg ma długość s, czyli składa się z s/k bloków. Wartość s możemy obliczyć, stosując algorytm zachłanny. Oznaczmy teraz przez d_i długość najdłuższego poprawnego podciągu, który można znaleźć w *sufiksie* ciągu g zaczynającym się na pozycji i, tzn. długość najdłuższego poprawnego podciągu w ciągu $g_i, g_{i+1}, \ldots, g_n$. Łatwo zauważyć, że ciąg wartości $d_1, \ldots, d_n$ jest nierosnący.

Pierwszy blok wynikowego podciągu może zakończyć się liczbą g_j tylko wtedy, gdy z pozostałej części ciągu g (tzn. z $g_{j+1}, \ldots, g_n$) można wybrać poprawny podciąg o długości $s-k$. Musi więc zachodzić $d_{j+1} = s-k$. Gdy znajdziemy największe j, dla którego $d_{j+1} = s-k$, pierwszy blok możemy skonstruować, wybierając k równych wyrazów spośród $g_1, \ldots, g_j$. Co więcej, pierwszy blok na pewno nie może zawierać wyrazu g_{j+1} lub późniejszego, bo wtedy z pewnością nie skonstruowalibyśmy rozwiązania o długości s. Zatem pierwszy blok w najwcześniejszym leksykograficznie ciągu wynikowym konstruujemy przez znalezienie najmniejszej liczby, która występuje co najmniej k razy wśród $g_1, \ldots, g_j$. Następnie możemy w analogiczny sposób skonstruować dalszą część rozwiązania, przy czym przeglądanie ciągu g rozpoczynamy tuż za k-tym wystąpieniem liczby, która utworzyła pierwszy blok.

Opiszemy teraz, jak zrealizować ten algorytm. Po pierwsze, musimy wyznaczyć wartości d_i. Przypomnijmy, że d_i to długość najdłuższego poprawnego podciągu, który znaleźć można w ciągu $g_i, \ldots, g_n$. Zauważmy, że opisany na początku algorytm zachłanny potrafi wyznaczyć długość najdłuższego poprawnego podciągu w każdym prefiksie ciągu g. Jeśli więc uruchomimy go dla odwróconego ciągu, tzn. dla $g_n, g_{n-1}, \ldots, g_1$, to wyznaczymy wartości d_i.

Następnie zaczynamy przetwarzać wejściowy ciąg g od początku i zliczamy wystąpienia poszczególnych liczb. W tym celu posługujemy się tablicą, która w p-tej komórce zlicza wystąpienia liczby p. Gdy dotrzemy do ostatniego miejsca, w którym $d_{i+1} = s-k$, będziemy chcieli znaleźć najmniejszą liczbę, która wystąpiła dotąd co najmniej k razy. To jednak możemy robić już w trakcie przetwarzania ciągu: możemy na bieżąco zliczać wystąpienia poszczególnych liczb i wykrywać, kiedy każda z liczb występuje dokładnie po raz k-ty. A skoro tak, to możemy także na bieżąco utrzymywać najmniejszą spośród tych liczb.

Teraz wiemy już, która liczba tworzy pierwszy blok, i chcemy dalsze poszukiwanie rozpocząć od miejsca tuż za k-tym wystąpieniem wybranej liczby. To również zrobimy bez kłopotu, bo możemy zapamiętać, gdzie się to k-te wystąpienie znajduje. W tym momencie musimy też wyczyścić tablicę, która służy nam do zliczania wystąpień liczb. Najprościej zrobić to po prostu, jeszcze raz przeglądając przetworzony kawałek ciągu — przy każdej napotkanej liczbie odejmujemy jedno jej wystąpienie w tablicy. Po tych wszystkich operacjach możemy kontynuować przeglądanie ciągu g, by znaleźć kolejne bloki w najwcześniejszym leksykograficznie rozwiązaniu.

Zastanówmy się, w jakim czasie działa ten algorytm. Przetworzenie jednego wyrazu ciągu g zajmuje czas stały. Jednak po znalezieniu każdego bloku cofamy się tuż za k-te wystąpienie liczby, która stworzyła kolejny blok.

Szczęśliwie dla nas, nie wpływa to na asymptotyczny czas działania algorytmu. Zauważmy, że jeśli cofamy się po przetworzeniu wyrazu g_j tuż za wyraz g_l, to $d_{j+1} = d_{l+1}$. Co więcej, jeśli kolejne cofnięcie następuje po przetworzeniu wyrazu $g_{j'}$ $(j' > j)$, to $d_{j'+1} < d_{j+1}$. Zatem jeśli zaznaczymy w ciągu $1, \ldots, n$ przedziały indeksów, po których się cofamy, to przedziały te będą rozłączne. Dzięki temu łączna liczba wykonanych kroków jest liniowa ze względu na n. Ponadto używamy tablicy rozmiaru M do zliczania wystąpień liczb w ciągu wejściowym. Nasz algorytm działa więc w czasie $O(n + M)$.

Jak błyskawicznie wyzerować tablicę?

W naszym rozwiązaniu używamy tablicy, która zlicza wystąpienia poszczególnych liczb we fragmentach ciągu $g_1, \ldots, g_n$. Co pewien czas musimy tę tablicę wyzerować, jednak nie chcemy w tym celu przeglądać jej wszystkich komórek (to mogłoby być nieefektywne). Powyżej poradziliśmy sobie z tym problemem, przeglądając wszystkie wyrazy przetworzonego fragmentu ciągu, jednak okazuje się, że możemy to zrobić w prostszy i bardziej ogólny sposób. Możemy mianowicie skonstruować tablicę, którą daje się zerować w czasie stałym!

Nasza ulepszona tablica, którą nazwiemy *tablicą zerowalną*, będzie udostępniać następujące operacje, każdą w czasie stałym:

- odczyt wartości z danej komórki,
- zapisanie wartości w pewnej komórce,
- zerowanie wszystkich komórek.

Przyjmijmy, że chcemy zaimplementować tablicę zerowalną długości n. W tym celu potrzebujemy dwóch zwykłych tablic długości n. W jednej z nich pamiętamy dane, a w drugiej czas, kiedy do danej komórki wykonaliśmy ostatni zapis. Czas mierzymy w *epokach* — nowa epoka rozpoczyna się wtedy, gdy zerujemy naszą tablicę. Zatem, jeśli odczytujemy wartość z komórki, do której ostatnio pisaliśmy przed bieżącą epoką, to komórka ta została wyzerowana od czasu ostatniego zapisu. Numer aktualnej epoki, czyli bieżący czas, pamiętamy w osobnej zmiennej.

Na samym początku zerujemy wszystkie komórki obu tablic i nasz licznik epok. Przy każdym zapisie w tablicy zerowalnej, zmieniamy wartość odpowiedniej komórki w tablicy z danymi i do przypisanej jej komórki w drugiej tablicy zapisujemy aktualny numer epoki. Gdy odczytujemy wartość pewnej komórki, sprawdzamy, czy zapisaliśmy do niej wartość w aktualnej epoce. Jeśli tak, to tę wartość zwracamy. W przeciwnym razie tablica została wyzerowana od czasu zapisu, więc zwracamy wartość 0. Dzięki temu, aby wyzerować tablicę, wystarczy zwiększyć wartość licznika epok o jeden. Sprawi to, że wartości we wszystkich komórkach się zdezaktualizują, czyli (zanim zapiszemy nowe wartości) każdy odczyt z tablicy zwróci 0. W ten sposób zerowanie dowolnie długiej tablicy sprowadza się do zmiany wartości jednej zmiennej.

J

Janosik

Autor zadania: Tomasz Idziaszek
Opis rozwiązania: Tomasz Idziaszek
Dostępna pamięć: 128 MB
`https://oi.edu.pl/pl/archive/amppz/2013/jan`

Jak wiadomo, Janosik zabiera bogatym, by rozdawać biednym. Wraz ze swoją bandą złupił właśnie konwój, który przewoził złoto do zamku murgrabiego. Łupem zbójów padło n szkatułek. Po przetransportowaniu ich do jaskini bandy okazało się, że i-ta szkatułka (dla $i = 1, 2, \ldots, n$) zawiera dokładnie i mieszków ze złotem.

Gdy do Janosika przychodzi biedak, prosząc o kilka złotych dukatów, Janosik stosuje następującą procedurę. Najpierw wybiera niepustą szkatułkę, która zawiera najmniej mieszków ze złotem. Jeśli jest w niej dokładnie jeden mieszek, Janosik wręcza go biedakowi, a ten odchodzi uradowany. W przeciwnym przypadku, jeśli w szkatułce znajduje się nieparzysta liczba mieszków, to Janosik chowa jeden z nich do kieszeni i zaczyna całą procedurę od nowa. Jeśli zaś w szkatułce znajduje się parzyście wiele mieszków, Janosik wyjmuje dokładnie połowę z nich ze szkatułki, przekłada je do jednej z pustych szkatułek (szczęśliwie w jaskini jest ich pod dostatkiem) i zaczyna całą procedurę od nowa. Tak więc jeśli biedak przybędzie do Janosika, gdy ten będzie miał jeszcze jakąś niepustą szkatułkę, to w wyniku (być może wielokrotnego) zastosowania procedury przez Janosika biedak na pewno otrzyma mieszek ze złotem. Biedacy przychodzą do jaskini Janosika tak długo, aż wszystkie szkatułki zostaną opróżnione.

Zbóje z bandy Janosika zastanawiają się, czy ich przywódca swoim postępowaniem nie narusza dobrego imienia zbójów. Chcieliby wiedzieć, ile złupionych mieszków pozostanie w kieszeni Janosika po opróżnieniu wszystkich szkatułek.

Wejście

W pierwszym i jedynym wierszu wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 10^9$), która oznacza liczbę szkatułek zrabowanych przez bandę Janosika.

Wyjście

W pierwszym i jedynym wierszu wyjścia należy wypisać liczbę całkowitą oznaczającą liczbę mieszków ze złotem, które pozostaną w kieszeni Janosika po opróżnieniu wszystkich szkatułek.

Przykład

Dla danych wejściowych:

```
5
```

poprawnym wynikiem jest:

```
2
```

Rozwiązanie

W zadaniu opisany został pewien algorytm, a naszym celem jest obliczyć wynik jego działania dla konkretnych danych wejściowych. Bezpośrednia implementacja tego algorytmu nie powinna nam sprawić trudności — po prostu symulujemy kolejne działania Janosika. Aktualne rozlokowanie mieszków w szkatułkach pamiętamy na liście: dla każdej szkatułki zawierającej i mieszków ze złotem, na liście trzymamy liczbę i. Początkowo na liście znajdują się liczby $1, 2, \ldots, n$. Dodatkowo w zmiennej *ile* pamiętamy liczbę mieszków w kieszeni Janosika.

Krok algorytmu wygląda następująco: na początku przeglądamy wszystkie elementy listy i znajdujemy element najmniejszy; oznaczmy jego wartość przez j. Następnie usuwamy ten element z listy. Jeśli $j = 1$, to w tym kroku już nic więcej nie robimy (mieszek został ofiarowany biedakowi). W przeciwnym wypadku, jeśli j było nieparzyste, to zwiększamy licznik *ile* (mieszek trafił do kieszeni Janosika) oraz dodajemy do listy element $j - 1$. Jeśli zaś j było parzyste, to dodajemy do listy dwa elementy $j/2$. Całość powtarzamy aż do momentu, gdy lista stanie się pusta.

Możemy wprowadzić pewne usprawnienie, które pozwoli nam zrealizować każdy krok algorytmu za pomocą stałej liczby działań. Będziemy mianowicie dbali o to, by elementy na liście były posortowane niemalejąco. Przy takim podejściu znalezienie elementu najmniejszego j jest proste — znajduje się on na początku listy. A ponieważ dodawane do listy elementy $j - 1$ lub $j/2$ są mniejsze od j (więc także mniejsze od wszystkich innych elementów na liście), możemy je wstawiać na początek listy, nie psując jej porządku.

Choć taka bezpośrednia implementacja algorytmu da zawsze poprawny wynik, to niestety nie działa ona wystarczająco szybko dla dużych wartości n. Spróbujmy się o tym przekonać, licząc, ile kroków algorytmu musimy wykonać. W każdym kroku wykonujemy jedną z dwóch operacji: (A) zmniejszamy liczbę mieszków w szkatułkach o 1 albo (B) dzielimy zawartość pewnej szkatułki. Aby opróżnić wszystkie szkatułki, dla każdego z mieszków musimy w pewnym momencie wykonać operację (A). Mieszków mamy w sumie $1+2+\ldots+n = \frac{n(n+1)}{2}$, więc dokładnie tyle razy wykonamy operację (A). Dla $n = 1\,000\,000\,000$ daje to około $5 \cdot 10^{17}$ operacji, przez co nawet gdybyśmy wykonywali te operacje błyskawicznie (powiedzmy, każdą z nich w ciągu paru nanosekund), to nasz program i tak działałby kilkadziesiąt lat. Nie musimy więc nawet liczyć operacji (B), ale dociekliwi czytelnicy mogą udowodnić, że wykonamy ich nie więcej niż $\frac{n(n+1)}{2}$. Zatem złożoność czasowa powyższego rozwiązania to $\Theta(n^2)$.

Aby rozwiązać zadanie, potrzebujemy szybciej obliczać wynik działania algorytmu Janosika.

Nie wykonujemy tej samej pracy dwa razy

Przyda nam się jeszcze jedna obserwacja. Za każdym razem, gdy Janosik decyduje się na otworzenie szkatułki zawierającej i mieszków, wszystkie mieszki z tej szkatułki musi rozdysponować, *zanim* otworzy jakąkolwiek inną szkatułkę zawierającą i lub więcej mieszków. Podczas całego procesu Janosik może wielokrotnie otwierać szkatułki zawierające i mieszków, ale każdym razem rozdziela on mieszki z takiej szkatułki w ten sam sposób. Tak więc liczba mieszków z każdej szkatułki zawie-

rającej i mieszków, które w ostateczności wylądują w jego kieszeni, jest zawsze taka sama.

To pozwala nam wyrazić tę liczbę za pomocą równania rekurencyjnego. Oznaczmy przez $d(i)$ liczbę mieszków pochodzących ze szkatułki o i mieszkach, które pozostaną w kieszeni Janosika. Wartości $d(\cdot)$ można opisać następującym wzorem rekurencyjnym:

$$d(i) = \begin{cases} 0 & \text{dla } i = 1, \\ 1 + d(i-1) & \text{dla } i \text{ nieparzystego i większego niż } 1, \\ 2 \cdot d(i/2) & \text{dla } i \text{ parzystego.} \end{cases} \tag{1}$$

Będziemy obliczać kolejno wartości $d(1), d(2), \ldots, d(n)$ i zapamiętywać je w tablicy. Dzięki temu każdą wartość $d(i)$ będziemy mogli obliczać w czasie stałym (zależy ona bowiem od wartości $d(i-1)$ lub $d(i/2)$, które zostały obliczone wcześniej).

Ponieważ $d(i) \leqslant i$ (Janosik nie może zabrać więcej mieszków, niż jest ich w szkatułce), więc tablica może składać się ze zmiennych 32-bitowych. Musimy jednak pamiętać, aby do obliczenia ostatecznego wyniku $d(1) + d(2) + \ldots + d(n)$ użyć zmiennej 64-bitowej.

Liczba kroków, które musimy wykonać, to n, zatem program działa w złożoności czasowej $\Theta(n)$. Niestety jego złożoność pamięciowa również wynosi $\Theta(n)$. W efekcie dla $n = 1\,000\,000\,000$ tak napisany program wykonuje się przez kilka sekund i w dodatku wykorzystuje sporo pamięci RAM. Każda komórka tablicy zajmuje 4 bajty, zatem cała tablica ma rozmiar około 3,7 GB. Jest to prawie 30 razy więcej niż rozmiar pamięci dostępnej w zadaniu. Potrzebujemy więc sprytniejszego rozwiązania.

Rozwiązujemy rekurencję

Rekurencja (1) nie jest bardzo skomplikowana, więc możemy pokusić się o jej rozwiązanie, czyli wyznaczenie wzoru na wartość $d(i)$, który nie odwołuje się do innych wartości $d(\cdot)$. W takiej sytuacji warto wypróbować następujący pomysł: obliczamy $d(i)$ dla kilku lub kilkunastu małych wartości i (czy to ręcznie na kartce, czy to przy użyciu komputera) i próbujemy zgadnąć poszukiwany wzór. Jeśli zgadniemy prawidłowo, to najczęściej przeprowadzenie dowodu poprawności zgadniętego wzoru nie nastręcza większych trudności. Poniższa tabelka przedstawia wartości $d(i)$ dla $i \leqslant 20$:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$d(i)$	0	0	1	0	1	2	3	0	1	2	3	4	5	6	7	0	1	2	3	4
$i - d(i)$	1	2	2	4	4	4	4	8	8	8	8	8	8	8	8	16	16	16	16	16

Może nam to zasugerować następujący wzór:

$$d(i) = i - 2^{\lfloor \log_2 i \rfloor}, \tag{2}$$

gdzie $2^{\lfloor \log_2 i \rfloor}$ to największa potęga dwójki nie większa niż i. Obliczenie $i - 2^{\lfloor \log_2 i \rfloor}$ polega zatem na zgaszeniu najbardziej znaczącej jedynki w zapisie binarnym liczby i. Pokażemy, że wzór (2) jest rzeczywiście rozwiązaniem rekurencji (1).

Wykorzystamy do tego indukcję matematyczną. Dla $i = 1$ mamy $\log_2 i = 0$, więc $1 - 2^0 = 0 = d(1)$, zatem baza indukcji jest spełniona. Dla $i \geqslant 2$ zakładamy,

że wzór (2) jest poprawny dla wszystkich $j < i$, i zajmujemy się badaniem $d(i)$. Jeśli i jest parzyste, mamy

$$d(i) = 2 \cdot d(i/2) = 2 \cdot (i/2 - 2^{\lfloor \log_2(i/2) \rfloor}) = 2 \cdot (i/2 - 2^{\lfloor \log_2 i \rfloor - 1}) = i - 2^{\lfloor \log_2 i \rfloor}.$$

Z kolei dla i nieparzystego mamy

$$d(i) = 1 + d(i-1) = 1 + (i - 1 - 2^{\lfloor \log_2(i-1) \rfloor}) = 1 + (i - 1 - 2^{\lfloor \log_2 i \rfloor}) = i - 2^{\lfloor \log_2 i \rfloor}.$$

Korzystamy tu z faktu, że jeśli i jest liczbą nieparzystą, to największa potęga dwójki nie większa niż i jest nie większa niż $i - 1$.

Ponieważ każdą wartość $d(i)$ możemy obliczać niezależnie od pozostałych, nie potrzebujemy już tablicy i złożoność pamięciowa naszego programu jest stała. Wygodnie jest obliczać kolejno wartości dla $i = 1, 2, \ldots, n$, pamiętając jednocześnie wartość największej potęgi dwójki nie większej niż aktualne i i zwiększając tę potęgę po 1, 2, 4, 8 itd. krokach. Złożoność czasowa takiego programu wynosi $\Theta(n)$. Jest to jednak nadal trochę zbyt wolno, żeby zmieścić się w limicie czasowym ustalonym przez autorów zadania.

Upraszczamy sumę

Oczywiście gdybyśmy mieli obliczyć wszystkie wartości $d(1), d(2), \ldots, d(n)$, to i tak potrzebowalibyśmy wykonać co najmniej n operacji. Nas jednak interesuje suma tych wartości $d(1) + d(2) + \ldots + d(n)$, którą możemy obliczyć szybciej.

Rozpisując poszukiwaną sumę, dostajemy następujący wzór:

$$\sum_{i=1}^{n} d(i) \;=\; \sum_{i=1}^{n} (i - 2^{\lfloor \log_2 i \rfloor}) \;=\; \frac{n(n+1)}{2} - \sum_{i=1}^{n} 2^{\lfloor \log_2 i \rfloor}.$$

Jak widać z umieszczonej powyżej tabelki, w ostatniej sumie każda potęga dwójki 2^j występuje dokładnie 2^j razy, być może oprócz największej potęgi dwójki $2^{\lfloor \log_2 n \rfloor}$, która występuje dokładnie $n - (2^0 + 2^1 + \ldots + 2^{\lfloor \log_2 n \rfloor - 1}) = n - 2^{\lfloor \log_2 n \rfloor} + 1$ razy. Każdą grupę zawierającą te same potęgi dwójki wystarczy liczyć tylko raz:

$$\sum_{i=1}^{n} d(i) \;=\; \frac{n(n+1)}{2} - \sum_{j=0}^{\lfloor \log_2 n \rfloor} 2^j \cdot \min(n - 2^j + 1,\; 2^j).$$

Skorzystanie z powyższego wzoru wymaga zsumowania $2 + \lfloor \log_2 n \rfloor$ wyrazów. W szczególności dla $n = 1\,000\,000\,000$ jest ich jedynie 31. Nasze ostateczne rozwiązanie ma złożoność czasową $\Theta(\log n)$ i działa w mgnieniu oka.

Kocyki

Autor zadania: Jakub Łącki
Opis rozwiązania: Eryk Kopczyński
Dostępna pamięć: 128 MB
https://oi.edu.pl/pl/archive/amppz/2013/koc

Tego lata mieszkańcy Bajtogrodu tłumnie wylegają na miejską plażę nad Jeziorem Bajtockim, by zaznać rozkoszy kąpieli słonecznej. Każdy obywatel Bajtogrodu przybywa na plażę zaopatrzony w najmodniejszy w tym sezonie kocyk wyprodukowany przez firmę *Bajtazar i Syn*. Wszystkie kocyki mają jednakowe wymiary $a \times b$ (choć różnorodne wzory), a każdy plażowicz zawsze ustawia swój kocyk dłuższym bokiem prostopadle do brzegu jeziora.

Jednym z tegorocznych plażowiczów jest profesor Bajtoni. Po kilku dniach plażowania profesor zauważył, że każdy z przybywających na plażę mieszkańców zawsze ustawia kocyk w tym samym, ulubionym przez siebie miejscu plaży. Mimo że mieszkańcy przybywają na plażę i opuszczają ją o różnych porach, profesor nigdy nie słyszał o tym, żeby któryś z plażowiczów swoim kocykiem zajął ulubione miejsce innego plażowicza. Fakt ten zaciekawił profesora, więc postanowił on zbadać to zjawisko.

W tym celu ustalił na plaży układ współrzędnych i dla każdego z n mieszkańców Bajtogrodu zapisał sobie współrzędne miejsca na plaży, w którym obywatel ten zawsze rozkłada swój kocyk. Układ jest dobrany w ten sposób, że oś OX jest równoległa do boków długości a, a oś OY — do boków długości b wszystkich kocyków. Profesor chciał początkowo obliczyć, dla każdej pary kocyków, ile wynosi pole przecięcia obszarów zajmowanych przez te kocyki. Potem jednak zorientował się, że do dalszych badań wystarczy mu *średnia* z tych wartości. Innymi słowy, interesuje go wartość oczekiwana pola przecięcia obszarów zajmowanych przez kocyki należące do dwóch różnych losowych mieszkańców Bajtogrodu. Korzystając z danych dostarczonych przez profesora, pomóż mu wykonać obliczenia.

Wejście

W pierwszym wierszu wejścia znajdują się trzy liczby całkowite n, a i b ($2 \leqslant n \leqslant 200\,000$, $1 \leqslant a, b \leqslant 1\,000\,000$), oznaczające, odpowiednio, liczbę mieszkańców Bajtogrodu i wymiary kocyków. Każdy z kolejnych n wierszy zawiera dwie liczby całkowite x_i i y_i ($0 \leqslant x_i, y_i \leqslant 1\,000\,000$), oznaczające współrzędne miejsca, w którym i-ty mieszkaniec Bajtogrodu zawsze układa lewy dolny róg swojego kocyka.

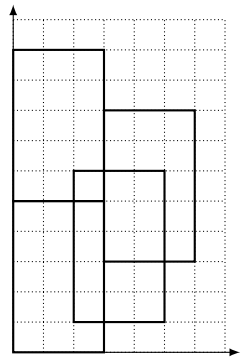
Wyjście

Twój program powinien wypisać jedną liczbę rzeczywistą, oznaczającą średnie pole przecięcia obszarów zajmowanych przez kocyki par mieszkańców Bajtogrodu. Twój wynik będzie uznany za poprawny, jeżeli znajdzie się w przedziale $[x - \varepsilon, x + \varepsilon]$, gdzie x jest prawidłową odpowiedzią, a $\varepsilon = 10^{-2}$.

Przykład

Dla danych wejściowych:	poprawnym wynikiem jest:

`4 3 5`
`0 0`
`2 1`
`3 3`
`0 5`
	`1.833333333`



Wyjaśnienie przykładu: Dokładny wynik to $\frac{4+0+0+1+6+0}{6} = 1\frac{5}{6}$.

Rozwiązanie

Zacznijmy od prostej obserwacji: aby obliczyć średnie pole powierzchni przecięcia dwóch kocyków, wystarczy obliczyć sumę pól powierzchni przecięć. Oznaczmy tę wartość przez S. Innymi słowy, aby wyznaczyć S, możemy rozważyć każdą parę kocyków i i j, gdzie $1 \leqslant i < j \leqslant n$, i dla każdej takiej pary dodać do S pole przecięcia kocyków i i j. Ostateczny wynik uzyskamy, dzieląc S przez $\binom{n}{2}$.

Łatwiej nam będzie jednak podejść do problemu z drugiej strony. Jeśli pewien kwadrat jednostkowy jest pokryty m kocykami, to należy on do przecięć $\binom{m}{2}$ par kocyków. Zatem kwadrat ten wnosi wkład $\binom{m}{2}$ do S. Skonstruujemy algorytm, który sumuje wkłady wszystkich kwadratów jednostkowych.

Doświadczeni w zadaniach geometrycznych zawodnicy zapewne podejrzewają, że w tym celu skorzystamy z techniki zamiatania — i słusznie. Będziemy sumować wkłady kwadratów jednostkowych kolumnami. Przesuwamy miotłę od lewej do prawej, w każdym momencie pamiętając s, czyli sumę wkładów we wszystkich kwadratach jednostkowych w kolumnie, w której obecnie jest miotła. W momencie, gdy miotła przekracza początek lub koniec kocyka, odpowiednio aktualizujemy wartość s. Miotły nie przesuwamy w sposób ciągły, tylko skaczemy do następnego zdarzenia (początku lub końca kocyka) — i przy takim skoku powiększamy wartość S o sx, gdzie x jest odległością, o którą się przesuwamy. Pozostaje pytanie: jak uaktualniać wartość s?

W tym celu utrzymujemy również multizbiór dolnych końców kocyków przecinanych przez miotłę, który oznaczamy przez D. Czyli jeśli miotła znajduje się

w pozycji X, to dla każdego kocyka (x, y) takiego, że $X \in [x, x + a]$, wartość y znajduje się w naszym multizbiorze.

Załóżmy, że w danym momencie mamy multizbiór D, miotła znajduje się na pozycji X i dochodzi nowy kocyk (X, y). Jak uaktualnić wartość s? Łatwo sprawdzić, że dla każdego $y' \in D$ musimy do s dodać wartość $m(y') = \max(0, b - |y - y'|)$. Musimy również wstawić y do naszego multizbioru. W przypadku gdy kocyk $(X - a, y)$ się kończy, postępujemy symetrycznie: usuwamy y z naszego multizbioru i odpowiednio zmniejszamy wartość s.

Jak zrobić to efektywnie? Zapiszmy nasz wzór w następującej postaci:

$$\begin{aligned}
\Delta &= \sum_{y' \in D} m(y') \\
&= \sum_{y' \in D} \max(0, b - |y - y'|) \\
&= \sum_{\substack{y' \in D \\ y - b \leqslant y' \leqslant y + b}} (b - |y - y'|) \\
&= \sum_{\substack{y' \in D \\ y - b \leqslant y' \leqslant y}} (b - y + y') + \sum_{\substack{y' \in D \\ y < y' \leqslant y + b}} (b - y' + y) \\
&= \sum_{\substack{y' \in D \\ y - b \leqslant y' \leqslant y}} y' - \sum_{\substack{y' \in D \\ y < y' \leqslant y + b}} y' + \sum_{\substack{y' \in D \\ y - b \leqslant y' \leqslant y}} (b - y) + \sum_{\substack{y' \in D \\ y < y' \leqslant y + b}} (b + y)
\end{aligned}$$

Zauważmy, że wszystkie składniki trzeciej sumy są równe, więc do jej obliczenia wystarczy nam poznać liczbę wartości y' w D, które należą do przedziału $[y - b, y]$. Podobnie możemy sobie poradzić z czwartą sumą. Z kolei aby obliczyć pierwsze dwie sumy, musimy umieć dla danego przedziału obliczyć sumę elementów D w danym przedziale. Strukturą pozwalającą wykonywać takie obliczenia jest drzewo przedziałowe.

Aby reprezentować multizbiór D, użyjemy dwóch drzew przedziałowych: jednego do zliczania elementów D w danym przedziale, a drugiego do liczenia ich sumy. Oznaczmy przez m górne ograniczenie na wartość wartość bezwzględną współrzędnych i wymiary kocyków. Do D wstawiamy liczby rzędu $O(m)$, dlatego każdą operację na drzewie przedziałowym możemy wykonać w czasie $O(\log m)$. Początki i końce kocyków możemy posortować do zamiatania w czasie $O(m + n)$ lub $O(n \log n)$, więc całe rozwiązanie działa w czasie $O(m + n \log m)$. Tę złożoność możemy poprawić do $O(n \log n)$, jeśli zastosujemy kompresję współrzędnych, to znaczy przekształcimy drugie współrzędne w liczby rzędu $O(n)$. Jest to jednak nieciekawe i żmudne, a przy ograniczeniach z zadania ma niewielki wpływ na czas działania rozwiązania.

Trzeba jeszcze zwrócić uwagę na dokładność: dane są za duże, by S zmieściło się w typie `long long`. Z kolei wykonywanie obliczeń na `double` okazuje się za mało dokładne. Możemy użyć albo typu `long double`, albo dwóch liczb 64-bitowych, by zasymulować działanie liczb 128-bitowych, albo wreszcie typu `__int128` dostępnego w niektórych kompilatorach, w tym w używanym na zawodach GCC. Okazuje się, że każde z tych podejść daje dostatecznie dokładny wynik.

2014

XIX Akademickie Mistrzostwa Polski
w Programowaniu Zespołowym
Warszawa, 24–26 października 2014

Adwokat

Autor zadania: Jakub Łącki
Opis rozwiązania: Jakub Łącki
Dostępna pamięć: 256 MB
`https://oi.edu.pl/pl/archive/amppz/2014/adw`

Adwokat Bajtazar, współwłaściciel kancelarii *Bajtazar i wspólnicy*, jest jednym z najbardziej rozchwytywanych członków bajtockiej palestry. Nic więc dziwnego, że jest wiecznie zajęty. Każdego dnia umawia się na liczne spotkania i dawno już przestał kontrolować to, czy będzie w stanie uczestniczyć w nich wszystkich. Zatrudnił więc sekretarza, który ma mu pomóc w ogarnięciu tego chaosu. Bajtazar postanowił, że każdego dnia uda się tylko na dwa spotkania i będzie w nich uczestniczył od początku do końca. Na pozostałe spotkania zostaną oddelegowani asystenci, których w kancelarii Bajtazara nie brakuje.

Niestety czasem w przepełnionym kalendarzu Bajtazara trudno znaleźć dwa spotkania, których terminy nie nachodzą na siebie. Przyjmujemy, że dwa spotkania nie nachodzą na siebie, jeśli jedno z nich zaczyna się *ściśle* po zakończeniu drugiego. Pomóż sekretarzowi Bajtazara i napisz program, który poradzi sobie z tym problemem.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite n i m ($2 \leqslant n \leqslant 500\,000$, $1 \leqslant m \leqslant 20$), oznaczające liczbę spotkań w terminarzu Bajtazara oraz liczbę dni, które obejmuje terminarz.

W każdym z kolejnych n wierszy opisane jest jedno spotkanie. Opis spotkania składa się z trzech liczb całkowitych a_i, b_i, d_i ($1 \leqslant a_i < b_i \leqslant 80\,000\,000$, $1 \leqslant d_i \leqslant m$), które oznaczają, że dnia d_i Bajtazar ma zaplanowane spotkanie, które rozpocznie się dokładnie a_i milisekund po północy i zakończy się b_i milisekund po północy.

Wyjście

Twój program powinien wypisać na wyjście m wierszy. W i-tym z tych wierszy powinna znaleźć się informacja, czy Bajtazar może uczestniczyć w dwóch spotkaniach i-tego dnia. Jeśli jest to niemożliwe, należy wypisać jedno słowo `NIE`. W przeciwnym razie należy wypisać `TAK`, a następnie numery dwóch spotkań, w których może uczestniczyć Bajtazar. Spotkania są ponumerowane od 1 do n, zgodnie z ich kolejnością na wejściu. Pierwsze z tych dwóch spotkań powinno się rozpoczynać wcześniej. Drugie spotkanie powinno zacząć się co najmniej milisekundę po zakończeniu pierwszego.

Jeśli istnieje wiele poprawnych odpowiedzi, Twój program powinien wypisać dowolną z nich.

Przykład

Dla danych wejściowych:

```
6 3
3 5 1
2 4 2
1 8 1
6 7 3
3 5 2
7 12 1
```

poprawnym wynikiem jest:

```
TAK 1 6
NIE
NIE
```

Rozwiązanie

Rozważmy każdy z m dni osobno. Dla ustalonego dnia nasze zadanie streścić można następująco. Dany jest zbiór przedziałów, wśród których należy znaleźć takie dwa przedziały, że prawy koniec pierwszego z nich jest ściśle mniejszy niż lewy koniec drugiego. Jeśli szukane przedziały istnieją, powinniśmy wypisać `TAK` oraz numery znalezionych przedziałów. W przeciwnym razie należy wypisać `NIE`.

Ten opis możemy natychmiast przełożyć na program, który przegląda wszystkie pary przedziałów i dla każdej pary przedziałów $[a_i, b_i]$ i $[a_j, b_j]$ sprawdza, czy $b_i < a_j$. Gdy znajdziemy pierwszą parę, natychmiast przerywamy dalsze poszukiwania i wypisujemy `TAK` oraz numery znalezionych przedziałów. Jeśli żadna taka para się nie znajdzie, wypisujemy `NIE`.

Problem w tym, że wszystkich par przedziałów może być bardzo dużo. W pesymistycznym przypadku przedziałów jest 500 000 i każde dwa z nich na siebie nachodzą (np. w sytuacji, gdy wszystkie przedziały są takie same). Wówczas przeglądamy $500\,000^2 = 2{,}5 \cdot 10^{11}$ par przedziałów. Zakładając (całkiem optymistycznie), że w ciągu sekundy można sprawdzić pół miliarda par, dla takich danych cały program będzie działać ponad 8 minut.

Aby znaleźć lepsze rozwiązanie, należy się chwilę zastanowić. Jeśli chcemy wybrać dwa przedziały $[a_i, b_i]$ i $[a_j, b_j]$, takie że $b_i < a_j$, to jak znaleźć dobrego kandydata na pierwszy z tych przedziałów? Znajdźmy po prostu przedział p o *najmniejszym* prawym końcu! Z kolei kandydatem na drugi przedział niech będzie przedział q o *największym* lewym końcu. Jeśli prawy koniec p jest mniejszy niż lewy koniec q (czyli $b_p < a_q$), to mamy rozwiązanie zadania. W przeciwnym razie poszukiwana para przedziałów nie istnieje, gdyż każdy przedział zawiera odcinek od punktu a_q do punktu b_p.

Teraz już nie musimy przeglądać wszystkich par przedziałów, bo aby znaleźć przedziały p i q, wystarczy tylko raz przejrzeć listę wszystkich przedziałów. Przedziałów może być co najwyżej 500 000, dzięki czemu program napisany na podstawie tego pomysłu zadziała w ułamku sekundy.

B

Benzyna

Autor zadania: Jakub Łącki
Opis rozwiązania: Adam Karczmarz
Dostępna pamięć: 256 MB
https://oi.edu.pl/pl/archive/amppz/2014/ben

Bajtazar pracuje w dziale transportu bajtockiego giganta paliwowego Bajtoil i planuje dostawy paliwa do stacji benzynowych.

W Bajtocji jest n skrzyżowań (ponumerowanych liczbami od 1 do n) oraz m dwukierunkowych dróg łączących pewne pary skrzyżowań. Przy niektórych skrzyżowaniach stoją stacje benzynowe Bajtoilu.

Flota transportowa firmy składa się z cystern o różnych pojemnościach baków. Każda cysterna spala 1 litr benzyny na kilometr przejechanej drogi. Można więc założyć, że cysterna o pojemności baku b litrów może przejechać maksymalnie b kilometrów bez konieczności uzupełnienia paliwa w baku. Kierowcy cystern nie mogą korzystać z paliwa przewożonego w zbiorniku cysterny, mogą za to za darmo uzupełniać paliwo w baku na stacjach benzynowych Bajtoilu.

Bajtazar w swojej pracy wielokrotnie musi sprawdzać odpowiedzi na pytania: czy cysterna o pojemności baku b litrów może przejechać ze stacji przy skrzyżowaniu x do stacji przy skrzyżowaniu y? Cysterna o pojemności baku b litrów nie może pokonać odcinka dłuższego niż b kilometrów, w trakcie którego nie będzie żadnej stacji benzynowej Bajtoilu. Cysterny zawsze rozpoczynają podróż na skrzyżowaniu, przy którym stoi stacja Bajtoilu, i kończą również na skrzyżowaniu, przy którym znajduje się stacja.

Pomóż Bajtazarowi zautomatyzować odpowiadanie na zapytania.

Wejście

W pierwszym wierszu wejścia znajdują się trzy liczby całkowite n, s i m ($2 \leqslant s \leqslant n \leqslant 200\,000$, $1 \leqslant m \leqslant 200\,000$), oznaczające odpowiednio liczbę skrzyżowań, liczbę stacji benzynowych i liczbę dróg w Bajtocji. W drugim wierszu wejścia znajduje się ciąg s parami różnych liczb całkowitych $c_1, c_2, \ldots, c_s$ ($1 \leqslant c_i \leqslant n$), oznaczających skrzyżowania, przy których stoją stacje Bajtoilu.

W kolejnych m wierszach opisane są drogi w Bajtocji; i-ty z tych wierszy zawiera trzy liczby całkowite u_i, v_i i d_i ($1 \leqslant u_i, v_i \leqslant n$, $u_i \neq v_i$, $1 \leqslant d_i \leqslant 10\,000$), oznaczające, że i-ta z dróg ma długość d_i kilometrów i łączy skrzyżowania u_i i v_i. Pomiędzy każdą parą skrzyżowań istnieje co najwyżej jedna droga.

W następnym wierszu znajduje się jedna liczba całkowita q ($1 \leqslant q \leqslant 200\,000$), oznaczająca liczbę zapytań. W kolejnych q wierszach znajdują się opisy zapytań; i-ty z tych wierszy zawiera trzy liczby całkowite x_i, y_i i b_i ($1 \leqslant x_i, y_i \leqslant n$, $x_i \neq y_i$, $1 \leqslant b_i \leqslant 2 \cdot 10^9$), oznaczające zapytanie o możliwość przejazdu cysterną o pojemności baku b_i litrów, ze stacji przy skrzyżowaniu x_i do stacji przy skrzyżowaniu y_i. Można założyć, że przy obu skrzyżowaniach x_i, y_i stoją stacje Bajtoilu.

Wyjście

Twój program powinien wypisać na wyjście dokładnie q wierszy. W i-tym z tych wierszy powinno znaleźć się jedno słowo `TAK` lub `NIE`, w zależności od tego, czy cysterna o pojemności baku b_i jest w stanie przejechać ze skrzyżowania x_i do skrzyżowania y_i.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
TAK
TAK
TAK
NIE
```

Rozwiązanie

Na sieć dróg z zadania można spojrzeć jak na graf $G = (V, E)$, w którym zbiór wierzchołków V odpowiada skrzyżowaniom, a zbiór krawędzi E — drogom. Dla krawędzi $(u, v) \in E$, oznaczmy przez $c(u, v)$ długość drogi od skrzyżowania u do skrzyżowania v wyrażoną w kilometrach. W zadaniu mamy także wyróżniony podzbiór S wierzchołków odpowiadających skrzyżowaniom, przy których stoją stacje benzynowe.

Naszym zadaniem jest wielokrotne odpowiadanie na zapytania następującej postaci. Dane są dwa wierzchołki $x, y \in S$ i liczba b. Czy istnieje taka ścieżka P z x do y, że jeśli przetniemy ją w każdym punkcie należącym do S, to otrzymamy kawałki długości nie większej niż b? Ścieżkę P nazwiemy *dobrą*, a zapytanie oznaczymy przez (x, y, b).

Załóżmy najpierw, że mamy odpowiedzieć na tylko jedno zapytanie (x, y, b). Oznaczmy przez $d_S(v)$ odległość z wierzchołka v do najbliższego wierzchołka ze zbioru S (dla $v \in S$ mamy $d_S(v) = 0$). Wprowadźmy też dla krawędzi $(u, v) \in E$ oznaczenie

$$c^*(u, v) := d_S(u) + c(u, v) + d_S(v).$$

Pokażemy teraz, że przy ustalonym b, dobra ścieżka z x do y istnieje wtedy i tylko wtedy, gdy pomiędzy x i y istnieje ścieżka złożona z krawędzi (u, v), dla których $c^*(u, v) \leqslant b$.

Załóżmy, że dla pewnej krawędzi (u, v) mamy $c^*(u, v) > b$. Wtedy żadna dobra ścieżka pomiędzy pewnymi wierzchołkami S nie może przebiegać krawędzią (u, v). Jeżeli bowiem trasa miałaby korzystać z tej krawędzi, musiałaby mieć fragment złożony z trzech części:

(1) odcinka od pewnego $s_1 \in S$ do u o długości co najmniej $d_S(u)$,

(2) krawędzi (u, v), o długości $c(u, v)$,

(3) odcinka od v do pewnego $s_2 \in S$, o długości nie mniejszej niż $d_S(v)$.

Tym samym ścieżka musiałaby mieć fragment długości co najmniej

$$d_S(u) + c(u, v) + d_S(v) = c^*(u, v) > b,$$

na którym nie ma żadnego wierzchołka z S. Na dobrej ścieżce taka sytuacja nie jest jednak dozwolona.

Z drugiej strony pokażemy, że jeśli między x i y istnieje w G ścieżka złożona jedynie z krawędzi (u, v), takich że $c^*(u, v) \leqslant b$, to istnieje też dobra ścieżka z x do y. Oznaczmy przez $l_S(v)$ dowolny wierzchołek z S położony najbliżej wierzchołka v, tzn. taki, że odległość z v do $l_S(v)$ jest równa dokładnie $d_S(v)$. Niech zatem $x = v_1, v_2, \ldots, v_p = y$ będzie taką ścieżką, że dla każdego $i = 1, 2, \ldots, p-1$ mamy $c^*(v_i, v_{i+1}) \leqslant b$. Ponieważ $x, y \in S$, mamy $l_S(x) = x$ i $l_S(y) = y$. Rozważmy teraz ścieżkę P:

$$l_S(v_1) \rightsquigarrow v_1 - v_2 \rightsquigarrow l_S(v_2) \rightsquigarrow v_2 - v_3 \rightsquigarrow l_S(v_3) \rightsquigarrow v_3 - \ldots \\ \ldots - v_{p-1} \rightsquigarrow l_S(v_{p-1}) \rightsquigarrow v_{p-1} - v_p \rightsquigarrow l_S(v_p),$$

gdzie przez $u \rightsquigarrow v$ oznaczamy najkrótszą ścieżkę z u do v, a $v_i - v_j$ oznacza przejście krawędzią (v_i, v_j). Ścieżka P prowadzi z x do y i składa się z $p-1$ odcinków postaci $l_S(v_i) \rightsquigarrow v_i - v_{i+1} \rightsquigarrow l_S(v_{i+1})$, dla każdego $i = 1, \ldots, p-1$. Każdy taki odcinek zaczyna się i kończy w wierzchołku należącym do S. Długość i-tego odcinka to, z definicji l_S, $d_S(v_i) + c(v_i, v_{i+1}) + d_S(v_{i+1}) = c^*(v_i, v_{i+1})$, czyli na mocy naszego założenia nie więcej niż b. Zatem rozważana ścieżka P jest dobra.

Podsumowując powyższe rozważania, przy ustalonym b, dobra ścieżka z x do y istnieje wtedy i tylko wtedy, gdy x i y są w tej samej spójnej składowej grafu G_b, który zawiera wszystkie wierzchołki grafu G, ale tylko te krawędzie (u, v), dla których $c^*(u, v) \leqslant b$. Innymi słowy

$$G_b = (V,\ \{(u, v) \in E : c^*(u, v) \leqslant b\}).$$

Algorytm

Pierwszym krokiem algorytmu rozwiązującego nasz problem będzie obliczenie dla każdej krawędzi $(u, v) \in E$ wartości $c^*(u, v)$. W tym celu musimy znaleźć wartość $d_S(v)$ dla każdego $v \in V$. Przypomnijmy, że $d_S(v)$ to długość najkrótszej ścieżki z v do najbliższego wierzchołka z S. Zbudujmy graf G' przez dodanie do grafu G superwierzchołka z połączonego z każdym wierzchołkiem ze zbioru S krawędzią o długości 0. Wówczas $d_S(v)$ jest równe odległości z z do v w grafie G'.

Ponieważ długości krawędzi są nieujemne, wartości $d_S(\cdot)$ możemy znaleźć w czasie $O((n+m)\log n)$, uruchamiając algorytm Dijkstry z wierzchołka z w grafie G'. Mając obliczone wartości $d_S(\cdot)$, obliczamy wartości $c^*(u,v)$ dla wszystkich krawędzi $(u,v) \in E$.

Zauważmy, że nie musimy udzielać odpowiedzi na wcześniejsze zapytania przed wczytaniem kolejnych zapytań. Ten fakt wykorzystujemy w rozwiązaniu: wczytujemy wszystkie zapytania zawczasu, sortujemy je niemalejąco względem wartości b_i i właśnie w tej kolejności wyznaczamy odpowiedzi. Przyjmijmy więc, że mamy dany ciąg zapytań $(x_1, y_1, b_1), \ldots, (x_q, y_q, b_q)$, gdzie $b_1 \leqslant b_2 \leqslant \ldots \leqslant b_q$.

Załóżmy, że dysponujemy strukturą danych reprezentującą nieskierowany graf H na zbiorze wierzchołków V. Na samym początku graf H nie posiada żadnych krawędzi. Struktura danych pozwala efektywnie wykonywać następujące operacje:

(1) Dodaj do H krawędź (u,v), gdzie $u, v \in V$.

(2) Sprawdź dla danych $u, v \in V$, czy w H istnieje ścieżka z u do v, to znaczy, czy u i v należą do tej samej spójnej składowej.

Aby obliczyć odpowiedzi na zapytania, najpierw sortujemy wszystkie krawędzie niemalejąco względem wartości $c^*(u,v)$. Następnie dodajemy kolejno posortowane krawędzie do wyżej wspomnianej struktury danych. Dla każdego $i = 1, \ldots, q$, struktura danych będzie w pewnym momencie reprezentować graf $H = G_{b_i}$. Możemy więc wykorzystać ten fakt i użyć struktury danych do wyznaczenia odpowiedzi na zapytanie (x_i, y_i, b_i). Przypomnijmy, że odpowiedź na zapytanie (x_i, y_i, b_i) jest twierdząca wtedy i tylko wtedy, gdy w G_{b_i} mamy ścieżkę z x_i do y_i. To jednak możemy sprawdzić, wykonując jedno zapytanie na strukturze danych.

Łącznie wykonamy na strukturze danych m operacji typu (1) i q operacji typu (2). Pozostaje wskazać, jak zaimplementować taką strukturę danych. Można w tym celu wykorzystać klasyczną *strukturę danych dla zbiorów rozłącznych*, zwaną też strukturą *find-union*. Pozwala ona wykonać $m+q$ operacji typu (1) lub (2) w czasie $O((m+q)\log^* n)$, który jest mniejszy niż $O((m+q)\log n)$. Podsumowując, wszystkie kroki algorytmu w sumie zajmą czas $O((n+m+q)\log(n+q))$.

Obsługa zapytań on-line

Jak już wspomnieliśmy, aby rozwiązać zadanie na zawodach, wystarczyło zaimplementować algorytm odpowiadający na zapytania w dowolnie wybranej kolejności. W naszym przypadku wygodnie było obsługiwać zapytania w kolejności rosnących wartości b_i. Możliwość odpowiadania na wszystkie zapytania naraz wydaje się jednak bardzo nierealistycznym założeniem — w sytuacji, gdy zapasy benzyny na pewnej stacji nagle się skończą, dostawę należy zaplanować natychmiast. W rzeczywistości nie możemy też zakładać, że wraz z upływem czasu flota transportowa koncernu paliwowego będzie używała cystern o coraz większych pojemnościach baków.

Pokażemy, że można zmodyfikować powyższe rozwiązanie tak, aby odpowiedź na dowolne zapytanie zajmowała czas $O(\log n)$. Przypomnijmy, że zapytanie (x, y, b) jest równoważne sprawdzeniu, czy x i y są w tej samej składowej grafu G_b. Zaproponujemy strukturę danych, która pozwoli nam szybko sprawdzać ten warunek.

Naszym celem jest zbudowanie lasu F na wierzchołkach V składającego się z ważonych, ukorzenionych drzew. Las F będzie spełniać dwie własności:

(1) Dla każdego b i dowolnych $v, w \in V$, wierzchołki v i w są w grafie G_b w tej samej składowej wtedy i tylko wtedy, gdy w lesie F istnieje między nimi ścieżka i żadna z wag na tej ścieżce nie przekracza b.

(2) Wysokość żadnego z drzew w lesie F nie przekracza $\log_2 n$.

Na pierwszy rzut oka może się wydawać zaskakujące, że taki las F istnieje. Gdyby interesował nas tylko warunek (1), moglibyśmy wykorzystać następujący fakt.

Fakt 1. Rozważmy ważony graf nieskierowany G, jego dowolne dwa wierzchołki v i w i dowolną liczbę b. Wówczas w grafie G istnieje ścieżka łącząca v i w składająca z krawędzi o wadze nie większej niż b wtedy i tylko wtedy, gdy taka ścieżka istnieje w minimalnym lesie rozpinającym grafu $G^\star$.

Zatem, gdybyśmy zajmowali się jedynie warunkiem (1), minimalny las rozpinający grafu G (z wagami $c^*(u,v)$) byłby dobrym kandydatem na F. My potrzebujemy jednak lasu spełniającego obydwa warunki, dlatego zaproponujemy nieco inną konstrukcję.

Zanim przejdziemy do opisu owej konstrukcji, wyobraźmy sobie, że dysponujemy lasem F o żądanych własnościach. Ponieważ drzewa są ukorzenione, dla każdego $v \in V$ niebędącego korzeniem jednego z drzew, istnieje jednoznacznie określony ojciec $p(v)$. Rozważmy następujący proces: zaczynamy w wierzchołku v i następnie w pętli przesuwamy się do ojca aktualnego wierzchołka dopóty, dopóki nie znajdujemy się w korzeniu i waga krawędzi do ojca nie przekracza b. Oznaczmy przez $r_b(v)$ wierzchołek, w którym opisany proces się kończy. Zauważmy teraz, że dwa wierzchołki v i w są w lesie F połączone ścieżką składającą się z krawędzi o wagach nie większych niż b wtedy i tylko wtedy, gdy $r_b(v) = r_b(w)$.

Teraz czas skorzystać z własności lasu F. Własność (1) gwarantuje, że wierzchołki v i w są w tej samej składowej G_b wtedy i tylko wtedy, gdy $r_b(v) = r_b(w)$. Z kolei własność (2) pozwala obliczać $r_b(v)$ i $r_b(w)$ w czasie proporcjonalnym do wysokości drzewa, a więc w czasie $O(\log n)$. Pozostaje tylko pokazać, jak skonstruować las F.

Konstrukcja lasu

Zauważmy, że jeśli z lasu F spełniającego własności (1) i (2) usuniemy krawędzie o wagach większych niż pewne B, to otrzymamy las, w którym własność (1) jest nadal prawdziwa dla każdego $b \leqslant B$. Oznaczmy tak utworzony las przez F_B. Będziemy konstruować lasy F_B dla coraz większych wartości B, dodając krawędzie o coraz większych wagach $c^*(u,v)$. Ostatecznie otrzymamy $F = F_C$, gdzie C jest największą wartością $c^*(u,v)$.

Rozpoczynamy od pustego lasu F_0, w którym nie ma żadnych krawędzi, przez co wartości $p(v)$ są niezdefiniowane dla wszystkich v. Dodatkowo dla każdego wierzchołka v utrzymujemy liczbę $h(v)$ określającą wysokość poddrzewa ukorzenionego w tym wierzchołku. Początkowo $h(v) = 0$ dla każdego $v \in V$. Las F_0 spełnia w sposób trywialny własność (1) dla $b \leqslant 0$.

Załóżmy dla uproszczenia, że wagi $c^*(u,v)$ są parami różne, i niech $B = c^*(u,v)$ będzie wagą pewnej krawędzi. Pokażemy, jak użyć lasu F_{B-1}, by otrzymać las F_B. Obliczamy najpierw $x = r_{B-1}(u)$ i $y = r_{B-1}(v)$ w czasie $O(h(x) + h(y))$, wędrując

⋆Minimalny las rozpinający grafu otrzymujemy, obliczając minimalne drzewo rozpinające w każdej spójnej składowej tego grafu.

krawędziami lasu F_{B-1}. Zauważmy, że x i y to korzenie drzew w F_{B-1}, bo w F_{B-1} wszystkie krawędzie mają wagi nie większe niż $B-1$. Załóżmy bez straty ogólności, że $h(x) \geqslant h(y)$.

Jeśli teraz $x = y$, to nic nie musimy robić, bo $F_B = F_{B-1}$. W przeciwnym razie łączymy drzewa o korzeniach w x i y. Chcemy drzewo o korzeniu w y uczynić poddrzewem x, zatem ustawiamy $p(y) = x$ i aktualizujemy wartości $h(\cdot)$: jeśli $h(x) = h(y)$, zwiększamy $h(x)$ o 1. Prosta analiza tej konstrukcji pozwala nam przekonać się, że w tak otrzymanym lesie F_B własność (1) jest spełniona dla dowolnego $b \leqslant B$.

Aby oszacować czas konstrukcji F i jednocześnie pokazać, że własność (2) jest spełniona, pokażemy ograniczenie na wartości $h(v)$. Konkretnie, pokażemy, że w każdym momencie działania algorytmu poddrzewo zaczepione w v jest rozmiaru co najmniej $2^{h(v)}$. Prześledźmy zmiany wartości $h(v)$ (formalnie rzecz biorąc, dowód przebiega indukcyjnie ze względu na wartość $h(v)$). Na początku, gdy $h(v) = 0$, wierzchołek v jest izolowany, więc teza zachodzi. Podczas konstrukcji wysokości nigdy nie zmniejszają się. Wartość $h(v)$ zwiększamy tylko wtedy, gdy do v podłączamy poddrzewo zaczepione w pewnym $w \neq v$ takim, że $h(w) = h(v)$. Czyli przed podłączeniem zarówno poddrzewo ukorzenione w v, jak i to ukorzenione w w, miały rozmiary co najmniej $2^{h(v)}$. Zatem po podłączeniu rozmiar poddrzewa v jest nie mniejszy niż $2^{h(v)+1}$, czego należało dowieść. Ponieważ F ma n wierzchołków, $2^{h(v)} \leqslant n$, czyli $h(v) \leqslant \log_2 n$.

Ostatecznie las F zajmuje $O(n)$ pamięci i konstruujemy go w sumarycznym czasie $O(n+m \log n)$. Na każde zapytanie odpowiadamy niezależnie w czasie $O(\log n)$.

C

Ceny

Autor zadania: Jakub Radoszewski
Opis rozwiązania: Jakub Radoszewski
Dostępna pamięć: 256 MB
`https://oi.edu.pl/pl/archive/amppz/2014/cen`

Bajtazar pracuje jako zaopatrzeniowiec w pewnej bajtockiej restauracji. Każdego dnia wieczorem otrzymuje on od kierownika listę produktów spożywczych, które musi zakupić kolejnego dnia rano. Bajtazar powinien kupić dokładnie jedną sztukę każdego produktu z listy. Kierownikowi zależy na tym, by wykonanie zakupów kosztowało jak najmniej.

Bajtazar siada wieczorem do komputera i sprawdza ceny potrzebnych produktów w lokalnych hurtowniach spożywczych. Zna on także koszt dojazdu z restauracji do każdej z hurtowni i z powrotem. Teraz Bajtazar musi zdecydować, które produkty kupić w każdej z hurtowni.

Dla każdej hurtowni, w której Bajtazar postanowił kupić jakieś produkty, zrobi on co następuje. Pojedzie do tej hurtowni, zrobi w niej zakupy i od razu kupione produkty zawiezie do restauracji. Szczęśliwie, bagażnik jego samochodu jest na tyle duży, że nie musi on do żadnej hurtowni jeździć wielokrotnie. Produkty spożywcze szybko się psują, dlatego w trakcie jednego kursu Bajtazar może odwiedzić tylko jedną hurtownię.

Napisz program, który pomoże Bajtazarowi obliczać najtańszy sposób wykonania wszystkich zakupów.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite n oraz m ($1 \leqslant n \leqslant 100$, $1 \leqslant m \leqslant 16$) oznaczające liczbę hurtowni oraz liczbę produktów, które Bajtazar powinien kupić. Kolejne n wierszy zawiera opisy cen w poszczególnych hurtowniach.

Pierwsza liczba w i-tym z tych wierszy, d_i ($1 \leqslant d_i \leqslant 1\,000\,000$), oznacza koszt dojazdu z restauracji do i-tej hurtowni (wliczając powrót). Po niej następuje ciąg m liczb całkowitych $c_{i,1}, c_{i,2}, \ldots, c_{i,m}$ ($1 \leqslant c_{i,j} \leqslant 1\,000\,000$): liczba $c_{i,j}$ oznacza cenę j-tego produktu w i-tej hurtowni.

Wyjście

Twój program powinien wypisać na wyjście jeden wiersz zawierający jedną liczbę całkowitą, oznaczającą sumę cen produktów i kosztów dojazdu do wybranych przez Bajtazara hurtowni w najtańszym możliwym planie zakupów.

Przykład

Dla danych wejściowych:	poprawnym wynikiem jest:
3 4 5 7 3 7 9 2 1 20 3 2 8 1 20 1 1	16

Wyjaśnienie przykładu: W pierwszej hurtowni Bajtazar kupuje produkt numer 2, a w drugiej wszystkie pozostałe produkty. Tak więc nie musi on odwiedzać trzeciej hurtowni.

Rozwiązanie

W tym zadaniu naszym celem jest stworzenie planu zakupu m produktów spożywczych. Mamy do dyspozycji n hurtowni, z których każda ma w swoim asortymencie każdy rodzaj produktu. Znając cenę każdego produktu w każdej hurtowni, jak również koszt dojazdu do każdej hurtowni, chcemy przygotować plan zakupów o jak najniższym łącznym koszcie. Nie musimy przy tym odwiedzać wszystkich hurtowni. Wiemy, że łącznie mamy do kupienia co najwyżej 16 produktów, a hurtowni jest nie więcej niż 100.

Pierwsze rozwiązanie

Niech $N = \{1, \ldots, n\}$ i $M = \{1, \ldots, m\}$. Ograniczenie $m \leqslant 16$ sugeruje, iż nasze rozwiązanie może mieć nawet złożoność wykładniczą względem m. Jedną ze standardowych metod prowadzących do rozwiązania o takiej złożoności czasowej jest programowanie dynamiczne, w którym stany odpowiadają podzbiorom zbioru wszystkich produktów. Dla każdego takiego podzbioru $A \subseteq M$ chcemy wyznaczyć najmniejszy koszt zakupienia produktów ze zbioru A. Oznaczmy taką wartość przez $koszt[A]$. Wartości $koszt[A]$ będziemy wyznaczać dla coraz liczniejszych podzbiorów zbioru M, zapamiętując wszystkie dotychczas obliczone wartości.

Jak obliczyć wartość $koszt[A]$? Jeśli $A \neq \emptyset$, wiemy, że zakup wszystkich produktów ze zbioru A wymaga pewnej liczby wizyt w hurtowniach. Rozważmy więc hurtownię, którą odwiedziliśmy jako ostatnią, oraz kupione w niej produkty. Oznaczmy, zgodnie z treścią zadania, przez $c_{i,j}$ cenę j-tego produktu w i-tej hurtowni, a przez d_i koszt dojazdu do i-tej hurtowni. Ponadto niech

$$zakup[i, B] = d_i + \sum_{j \in B} c_{i,j} \tag{1}$$

oznacza łączny koszt zakupu wszystkich produktów ze zbioru B w hurtowni numer i, wliczając w to koszt dojazdu do tej hurtowni. Wówczas wartości $koszt[\cdot]$ możemy obliczać za pomocą następującego wzoru rekurencyjnego:

$$koszt[A] = \min\{zakup[i, B] + koszt[A \setminus B] \,:\, i \in N,\, B \subseteq A,\, B \neq \emptyset\}. \tag{2}$$

W tym wzorze i reprezentuje numer ostatniej odwiedzonej hurtowni, a B oznacza zbiór produktów kupowanych w tej hurtowni. Warto zauważyć, że powyższy

wzór rozważa także plany zakupów, w których pewną hurtownię odwiedzamy wielokrotnie (tj. hurtownia numer i mogła wystąpić przy obliczaniu wyniku dla zbioru produktów $A \setminus B$), jednak jest to z pewnością nieopłacalne.

W ten sposób udało nam się uzyskać pierwsze, dosyć abstrakcyjne rozwiązanie. Choć nie zastanawialiśmy się jeszcze nad szczegółami jego implementacji, spróbujmy zgrubnie oszacować jego złożoność obliczeniową. Przyjrzyjmy się wzorowi (2): możliwych zbiorów A jest 2^m, dla każdego z nich rozważamy n hurtowni i $2^{|A|} \leqslant 2^m$ podzbiorów produktów, co można prosto ograniczyć przez $O(4^m n)$. Takie ograniczenie nie jest jednak zadowalające. Jak się okaże, na podstawie wzoru (2) można skonstruować rozwiązanie działające znacznie szybciej.

Reprezentacja podzbiorów

Podzbiory zbioru M będziemy reprezentować jako liczby całkowite z zakresu od 0 do $2^m - 1$. Ustalmy podzbiór $A \subseteq M$. Zauważmy, że liczbę z zakresu od 0 do $2^m - 1$ możemy zapisać, używając dokładnie m bitów. Każdy z tych bitów wykorzystujemy do zapisania, czy pewien element M należy do A: i-ty bit identyfikatora zbioru A (czyli liczby reprezentującej ten zbiór) jest zapalony wtedy i tylko wtedy, gdy $i \in A$. Przy tej reprezentacji operacje bitowe na identyfikatorach (liczbach) odpowiadają podstawowym operacjom na zbiorach. Na przykład, aby obliczyć przecięcie dwóch zbiorów, wystarczy wykonać operację bitową **and** na ich reprezentacjach. Z kolei różnica zbiorów, z których jeden jest podzbiorem drugiego, odpowiada zwykłemu odejmowaniu identyfikatorów tych zbiorów. Taka reprezentacja pozwoli nam także efektywnie indeksować tablice w rozwiązaniu, w szczególności tablicę *koszt*.

Podczas wypełniania tablicy *koszt* powinniśmy zadbać o to, aby w chwili rozpatrywania zbioru A mieć już do dyspozycji wyniki dla wszystkich podzbiorów tego zbioru. W tym celu moglibyśmy oczywiście przetwarzać zbiory np. w kolejności rosnącej liczby elementów. Jednak przy obranej reprezentacji możemy to wykonać istotnie prościej. Zauważmy, że jeśli $B \subseteq A$, to identyfikator zbioru B jest nie większy od identyfikatora zbioru A. Tak więc wystarczy przeglądać zbiory w kolejności rosnących identyfikatorów, od 0 do $2^m - 1$.

W dalszej części opisu będziemy dla uproszczenia utożsamiać zbiory z ich identyfikatorami.

Podzbiory podzbiorów

W analizie wcześniej opisanego rozwiązania oszacowaliśmy z góry złożoność czasową wyznaczania pojedynczej wartości $koszt[A]$ przez $O(2^m n)$, choć w rzeczywistości jest to „zaledwie" $O(2^{|A|} n)$. Czy ma to jednak jakieś znaczenie? Okazuje się, że ma.

W naszym rozwiązaniu przeglądamy wszystkie podzbiory A zbioru M, a w drugiej kolejności dla każdego z nich jego wszystkie (niepuste) podzbiory B. Spójrzmy na ten sam proces nieco inaczej: przeglądamy wszystkie pary zbiorów (A, B), takie że $B \subseteq A \subseteq M$. Spróbujemy teraz dokładnie wyznaczyć liczbę takich par. Każdą parę możemy zakodować jako pokolorowanie wszystkich elementów zbioru $M = \{1, \ldots, m\}$ trzema kolorami: kolorem *beżowym* oznaczamy wszystkie elementy należące do zbioru B, kolorem *amarantowym* — wszystkie elementy należące do zbioru A, ale nienależące do zbioru B, wreszcie kolorem *żółtym* — elementy nienależące do zbioru A (a więc także nienależące do zbioru B). Jako że

$B \cup (A \setminus B) \cup (M \setminus A) = M$ i każde dwa spośród sumowanych zbiorów są rozłączne, to rzeczywiście każdy element zbioru M otrzymuje dokładnie jeden z trzech dostępnych kolorów.

Nietrudno zauważyć, że każda para zbiorów (A, B) wyznacza inne pokolorowanie elementów zbioru M, a każde pokolorowanie odpowiada innej parze zbiorów. Wszystkich pokolorowań elementów zbioru M trzema kolorami jest 3^m, więc taka jest też liczba rozważanych par zbiorów. Pozwala to istotnie lepiej oszacować złożoność czasową naszego rozwiązania.

Czym innym jest jednak teoria, a czym innym to, w jaki sposób można to tak efektywnie zaimplementować. Aby złożoność naszego rozwiązania rzeczywiście zależała od 3^m a nie od 4^m (co ma znaczenie: 4^{16} to ponad 4 miliardy, gdy tymczasem 3^{16} to zaledwie 40 milionów), musimy rzeczywiście przeglądać w nim tylko takie pary zbiorów (A, B), że $B \subseteq A \subseteq M$. Jest na to pewien sposób — prosty, acz niebywale sprytny.

Dla danego zbioru A, jego podzbiory będziemy przeglądać w kolejności malejących identyfikatorów. Zaczniemy więc od zbioru $B = A$. W każdym kolejnym kroku będziemy zmniejszali identyfikator zbioru o jeden. Jeśli w ten sposób uzyskamy podzbiór zbioru A, to niewątpliwie będzie to bezpośredni poprzednik rozważanego podzbioru. Jeśli nie, to jako aktualny podzbiór przyjmiemy *część wspólną* uzyskanego w ten sposób zbioru oraz zbioru A. Tę ideę ilustruje poniższy pseudokod:

$B := A$
while $B > 0$ **do**
  $B := (B - 1)$ **and** A

Przykładowo, wszystkie podzbiory zbioru $A = \{1, 2, 4, 6\}$ wygenerowane za pomocą powyższego algorytmu, zapisane w notacji dwójkowej, wyglądają tak (należy czytać kolumnami):

```
101011   100011   001011   000011
101010   100010   001010   000010
101001   100001   001001   000001
101000   100000   001000   000000
```

Spróbujmy zrozumieć, dlaczego ten algorytm działa. Jasne jest, że wszystkie uzyskiwane po drodze zbiory są podzbiorami zbioru A. Co więcej, pętla **while** na pewno nie będzie obracać się w nieskończoność, gdyż w każdym kroku liczba B maleje co najmniej o 1 (a operacja **and** może ją tylko zmniejszyć). Trzeba jeszcze tylko wykazać, że po drodze nie pominiemy żadnego podzbioru zbioru A.

Intuicyjnie można to uzasadnić tak. Gdyby zbiór A składał się z iluś kolejnych liczb, począwszy od 1, to operacja **and** w ogóle nie byłaby potrzebna i wówczas niewątpliwie powyższy algorytm wygenerowałby wszystkie podzbiory zbioru A. Jeśli natomiast w zbiorze A występują „luki" — czyli istnieją takie liczby $x \notin A$, że pewna większa liczba należy do A — to po każdym obrocie pętli **while** bity odpowiadające tym lukom w B będą wyzerowane. To oznacza, że nie będą one wpływały na to, co dzieje się na bitach odpowiadających elementom ze zbioru A w chwili odejmowania 1, więc możemy po prostu o nich wcale nie myśleć.

Formalnie dowód można przeprowadzić za pomocą następującego lematu.

Lemat 1. Załóżmy, że $B \subseteq A$ $(B \neq \emptyset)$, i niech $B' = (B - 1)$ **and** A. Wówczas B' jest podzbiorem zbioru A o największym identyfikatorze mniejszym od B.

Dowód: Niech $A = \{a_1, \ldots, a_k\}$ i $a_1 < a_2 < \ldots < a_k$. Załóżmy najpierw, że $a_1 \in B$. Wówczas podzbiorem zbioru A będącym poprzednikiem zbioru B jest po prostu $B \setminus \{a_1\}$. W przeciwnym razie, niech a_i $(i > 1)$ będzie najmniejszym elementem zbioru B. Wówczas poprzednikiem zbioru B jest zbiór $(B \cup \{a_1, \ldots, a_{i-1}\}) \setminus \{a_i\}$. W obydwu przypadkach zbiór B' okazuje się właśnie szukanym poprzednikiem. □

Ostatecznie zastosowanie efektywnego algorytmu generowania wszystkich podzbiorów danego zbioru prowadzi do rozwiązania o złożoności czasowej $O(3^m n)$.

Niewielkie usprawnienie i rozwiązanie wzorcowe

Możemy w prosty sposób jeszcze trochę poprawić efektywność naszego rozwiązania. Spróbujmy mianowicie wyeliminować ze złożoności czasowej czynnik n.

Wróćmy jeszcze raz do wzoru (2). Przypomnijmy sobie, że rozważamy w nim wszystkie możliwe hurtownie i oraz wszystkie zbiory produktów B, które możemy w takiej hurtowni kupić. Tymczasem dla każdego zbioru B wystarczyłoby rozważać tylko jedną hurtownię — tę, w której ten zbiór produktów możemy nabyć najtaniej! Oznaczmy ten najtańszy sposób zakupienia produktów ze zbioru B przez

$$\mathit{min\_zakup}[B] = \min\{\mathit{zakup}[i, B] \,:\, i \in N\}. \tag{3}$$

Wówczas wzór (2) możemy przepisać tak:

$$\mathit{koszt}[A] = \min\{\mathit{min\_zakup}[B] + \mathit{koszt}[A \setminus B] \,:\, B \subseteq A,\ B \neq \emptyset\}. \tag{4}$$

Złożoność czasowa obliczenia w ten sposób wszystkich wartości $\mathit{koszt}[A]$ wynosi już tylko $O(3^m)$.

Wszystkie wartości $\mathit{min\_zakup}[B]$ możemy spamiętać na samym początku, bezpośrednio ze wzorów (3) i (1), w czasie $O(2^m mn)$. Można by to też zrobić nieco szybciej, w czasie $O(2^m n)$, wyznaczając $\mathit{zakup}[i, A]$ na podstawie wcześniej obliczonych wartości — jednak nie wpływa to na ostateczną efektywność rozwiązania. Tego typu rozwiązania były już akceptowane na zawodach.

Rozwiązanie alternatywne

Okazuje się, że istnieje jeszcze lepsze rozwiązanie, w którym czynnik wykładniczy w złożoności jest równy tylko 2^m. To rozwiązanie również opiera się na programowaniu dynamicznym po podzbiorach zbioru produktów. Tym razem będziemy przechowywać więcej stanów, co w zamian umożliwi nam rozpatrywanie hurtowni i produktów pojedynczo.

Niech $\mathit{koszt}[i, A]$ oznacza najmniejszy koszt zakupienia produktów ze zbioru $A \subseteq M$ przy ograniczeniu, że możemy korzystać tylko z hurtowni o numerach $1, \ldots, i$. Niech dalej $\mathit{koszt\_z}[i, A]$ oznacza taki sam koszt przy dodatkowym założeniu, że na pewno robimy zakupy w hurtowni numer i. Wiemy, jak wyglądają wartości $\mathit{koszt}[0, \cdot]$:

$$\mathit{koszt}[0, \emptyset] = 0, \quad \mathit{koszt}[0, A] = \infty \ \text{ dla } A \neq \emptyset,$$

a naszym celem jest wyznaczenie wartości $\mathit{koszt}[n, M]$. Dla każdej kolejnej hurtowni $i = 1, \ldots, n$, będziemy najpierw obliczać wartości $\mathit{koszt\_z}[i, \cdot]$, korzystając z wartości $\mathit{koszt}[i-1, \cdot]$, a następnie $\mathit{koszt}[i, \cdot]$ wyznaczymy z prostego wzoru:

$$\mathit{koszt}[i, A] = \min(\mathit{koszt}[i-1, A],\ \mathit{koszt\_z}[i, A]).$$

Mówi on, że jeśli produkty ze zbioru A chcemy kupić w hurtowniach $1, \ldots, i$, to albo nie udajemy się do hurtowni i i wówczas płacimy $koszt[i-1, A]$, albo też do tej hurtowni jedziemy i wtedy całe zakupy kosztują nas $koszt\_z[i, A]$.

Pozostaje ustalić, jak obliczać wartości $koszt\_z[i, \cdot]$. Przypomnijmy, że wartość $koszt\_z[i, A]$ to optymalny koszt zakupów, podczas których udajemy się do hurtowni i (i być może innych). Załóżmy, że w hurtowni i kupujemy pewien produkt $j \in A$. Dla tego produktu (w ogólności dla każdego produktu j kupionego w hurtowni i) prawdziwy jest wzór:

$$koszt\_z[i, A] = koszt\_z[i, A \setminus \{j\}] + c_{i,j}.$$

W rozwiązaniu nie wiemy jednak, które produkty kupujemy w hurtowni i, dlatego wybieramy najlepszą możliwość:

$$koszt\_z[i, A] := \min\{koszt\_z[i, A \setminus \{j\}] + c_{i,j} \ : \ j \in A\}.$$

Ten wzór rekurencyjny musimy uzupełnić jeszcze o warunek brzegowy. Odpowiada on sytuacji, w której udajemy się do hurtowni i, ale nic w niej nie kupujemy. W takiej sytuacji mamy:

$$koszt\_z[i, A] = koszt[i-1, A] + d_i.$$

Podsumujmy te rozważania pseudokodem rozwiązania. Przypomnijmy, że podzbiory zbioru M utożsamiamy z ich identyfikatorami $0, \ldots, 2^m - 1$. W szczególności, aby sprawdzić, czy $j \in A$, za pomocą operacji **and** sprawdzamy, czy przecięcie zbiorów $\{j\}$ i A jest puste. Ponadto, jeśli $j \in A$, operację $A \setminus \{j\}$ realizujemy przez odjęcie liczby 2^j od identyfikatora zbioru A.

```
koszt[0, 0] := 0
for A := 1 to 2^m − 1 do
  koszt[0, A] := ∞
for i := 1 to n do
  for A := 0 to 2^m − 1 do
    koszt_z[i, A] := koszt[i − 1, A] + d_i
    for j := 1 to m do
      if (A and 2^j) ≠ 0 then
        koszt_z[i, A] := min(koszt_z[i, A], koszt_z[i, A − 2^j] + c_{i,j})
  for A := 0 to 2^m − 1 do
    koszt[i, A] := min(koszt[i − 1, A], koszt_z[i, A])
return koszt[n, 2^m − 1]
```

Gołym okiem widać, że złożoność czasowa tego rozwiązania to $O(2^m mn)$. Złożoność pamięciowa wzrosła do $O(2^m n)$, jednak bardzo łatwo sprowadzić ją z powrotem do $O(2^m)$. Wystarczy zauważyć, że w danym kroku korzystamy tylko z wartości postaci $koszt[i, \cdot]$ dla dwóch kolejnych indeksów i (i tylko z jednego indeksu i w przypadku tablicy $koszt\_z$). Wystarczy nam do tego tablica złożona z dwóch wierszy, a wszystkie odwołania typu $koszt[i, A]$ możemy zastąpić odwołaniami do $koszt[i \bmod 2, A]$. Możemy też zaoszczędzić jeszcze trochę pamięci i nie wprowadzać tablicy $koszt\_z$. Pozostawiamy Czytelnikowi sprawdzenie, że jeśli wszystkie odniesienia do tej tablicy w powyższym pseudokodzie zamieni się po prostu na odniesienia do tablicy $koszt$, algorytm pozostanie poprawny.

D

Dzielniki

Autor zadania: Jakub Łącki
Opis rozwiązania: Jakub Łącki
Dostępna pamięć: 256 MB
https://oi.edu.pl/pl/archive/amppz/2014/dzi

Dany jest ciąg n liczb całkowitych $a_1, a_2, \ldots, a_n$. Należy wyznaczyć liczbę takich par uporządkowanych (i, j), że $i, j \in \{1, \ldots, n\}$, $i \neq j$ oraz a_i jest dzielnikiem a_j.

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 2\,000\,000$). W drugim wierszu znajduje się ciąg n liczb całkowitych $a_1, a_2, \ldots, a_n$ ($1 \leqslant a_i \leqslant 2\,000\,000$).

Wyjście

W pierwszym i jedynym wierszu wyjścia należy wypisać jedną liczbą całkowitą, oznaczającą szukaną liczbę par.

Przykład

Dla danych wejściowych:

```
5
2 4 5 2 6
```

poprawnym wynikiem jest:

```
6
```

Wyjaśnienie przykładu: Istnieje 6 par o podanych własnościach: $(1, 2)$, $(1, 4)$, $(1, 5)$, $(4, 1)$, $(4, 2)$, $(4, 5)$.

Rozwiązanie

Rozważmy nasze zadanie dla ciągu $1, 2, \ldots, M$. Zastanawiamy się zatem, ile jest par liczb (i, j), takich że $1 \leqslant i, j \leqslant M$, i jest dzielnikiem j oraz $i \neq j$. W szczególności zatem $i < j$. Przykładowo, dla $M = 9$ mamy 14 takich par:

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

Aby przejrzeć wszystkie te pary, dla każdej liczby i od 1 do M przeglądamy wielokrotności i, które są większe od i, ale nie przekraczają M. Jeśli ustalimy $1 \leqslant i \leqslant M$, to takich wielokrotności jest dokładnie $\lfloor \frac{M}{i} \rfloor - 1$. Jedynkę odejmujemy

dlatego, że interesują nas jedynie wielokrotności *większe* od i, także samą liczbę i musimy wykluczyć. Liczbę przejrzanych par możemy ograniczyć z góry przez

$$\sum_{i=1}^{M} \left(\left\lfloor \frac{M}{i} \right\rfloor - 1 \right) \leqslant \sum_{i=1}^{M} \frac{M}{i} = M \sum_{i=1}^{M} \frac{1}{i} = O(M \log M).$$

W ostatnim kroku korzystamy z ważnego faktu: suma wartości $\frac{1}{i}$ po wszystkich liczbach i od 1 do M jest bardzo bliska $\log M$ (log oznacza tutaj logarytm naturalny). Co oznacza „bardzo bliska"? Dla wszystkich wartości M suma ta różni się od $\log M$ o nie więcej niż 1.

Główny morał płynący z powyższych rozważań jest następujący. Niech S oznacza zbiór par (i, j), takich że $1 \leqslant i < j \leqslant M$ oraz i dzieli j. Pokazaliśmy, że rozmiar zbioru S jest stosunkowo niewielki, bo ograniczyliśmy go z góry przez $O(M \log M)$. Co więcej, wszystkie elementy zbioru S możemy efektywnie przejrzeć, gdyż wystarczy dla każdej kolejnej liczby i po prostu generować jej wielokrotności nieprzekraczające M.

Stąd już niedaleko do rozwiązania zadania dla ogólnego ciągu $a_1, \ldots, a_n$. Zauważmy, że każda para wyrazów ciągu, która wlicza się do wyniku, odpowiada pewnej parze ze zbioru S. Zatem nasz problem możemy odwrócić: wystarczy przeglądać wszystkie pary (i, j) w zbiorze S i dla każdej z nich obliczyć, ile par wyrazów ciągu $a_1, \ldots, a_n$ jest równych parze (i, j). Dla przykładu, jeśli rozpatrujemy parę $(3, 12) \in S$, a w ciągu $a_1, \ldots, a_n$ liczba 3 występuje 5 razy, zaś liczba 12 występuje 7 razy, to z ciągu $a_1, \ldots, a_n$ możemy wybrać dokładnie $5 \cdot 7 = 35$ par wyrazów, tak by uzyskać parę $(3, 12)$. Zatem dla pary $(3, 12)$ do wyniku dodamy 35. Aby ten algorytm zaimplementować efektywnie, potrzebujemy najpierw dla każdej liczby w zakresie od 1 do M obliczyć, ile razy występuje ona w ciągu $a_1, \ldots, a_n$.

Opiszmy teraz cały algorytm. Od teraz niech M będzie równe największej spośród liczb $a_1, \ldots, a_n$. Na początku wyznaczmy tablicę *ile*, przy czym *ile*$[i]$ oznacza, ile wyrazów ciągu $a_1, \ldots, a_n$ ma wartość i. W kolejnym kroku przeglądamy kolejne komórki tablicy *ile*. Gdy natrafimy na wartość *ile*$[i]$ różną od 0, przeglądamy wielokrotności liczby i większe od i i nie większe niż M. Dla każdej takiej wielokrotności k wiemy, że mamy *ile*$[k]$ wyrazów o wartości k. Co więcej, i jest dzielnikiem k. Zatem do wyniku dodajemy iloczyn *ile*$[i]$ oraz *ile*$[k]$.

Trzeba jeszcze uwzględnić pary tych elementów ciągu $a_1, \ldots, a_n$, które są sobie równe. Skoro mamy *ile*$[i]$ elementów o wartości i, tworzą one *ile*$[i] \cdot (ile[i] - 1)/2$ dodatkowych par.

Aby oszacować czas działania, wystarczy zauważyć, że główna część rozwiązania działa w czasie proporcjonalnym do rozmiaru zbioru S. Do tego na początku zliczamy wystąpienia poszczególnych liczb w ciągu $a_1, \ldots, a_n$. Zaproponowany algorytm działa zatem w czasie $O(n + M \log M)$: tablicę *ile* wypełniamy w czasie $O(n+M)$, a następnie potrzebujemy jeszcze czasu $O(M \log M)$ na zliczenie wszystkich par.

Posłowie

Kluczem do rozwiązania zadania *Dzielniki* była własność

$$\sum_{i=1}^{n} \frac{1}{i} = O(\log n).$$

Warto o niej pamiętać, bo przydaje się ona w wielu sytuacjach, nawet gdy bezpośrednio nie zajmujemy się własnościami dzielników czy szeroko pojętą teorią liczb. Jednym z przykładów jest tak zwany *problem kolekcjonera kuponów*. Kolekcjoner ten zbiera naklejki z rysunkami superbohaterów (w oryginalnej wersji: zbiera kupony), które znajduje w paczkach płatków śniadaniowych. W każdej paczce znajduje się dokładnie jedna naklejka losowo wybrana spośród n możliwych rodzajów naklejek. Ile paczek płatków powinien (w średnim przypadku) otworzyć kolekcjoner, by w swojej kolekcji mieć naklejki wszystkich n rodzajów?

Pierwszą naklejkę kolekcjoner zdobędzie przy otwarciu pierwszej paczki. Z kolei, gdy będzie już miał $n-1$ rodzajów naklejek, będzie musiał średnio otworzyć jeszcze n paczek, by trafić na naklejkę ostatniego brakującego rodzaju. W ogólności można zauważyć, że jeśli kolekcjoner ma już $i-1$ różnych rodzajów naklejek, to zdobycie naklejki nowego rodzaju wymagać będzie średnio otwarcia $\frac{n}{n-i+1}$ kolejnych paczek. Jeśli zsumujemy te liczby dla wszystkich i, otrzymamy

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

Zatem kolekcjoner będzie musiał kupić średnio $O(n \log n)$ paczek płatków, zanim zdobędzie wszystkie n naklejek.

Euklidesowy Nim

Autor zadania: Tomasz Idziaszek
Opis rozwiązania: Adam Karczmarz
Dostępna pamięć: 256 MB
https://oi.edu.pl/pl/archive/amppz/2014/euk

Euklides i Pitagoras to pseudonimy dwóch Bajtocczan słynących z zamiłowania do matematycznych zagadek. Ostatnio wieczory spędzają, grając w następującą grę. Na stole leży stos n kamieni. Przyjaciele wykonują na przemian ruchy. Ruch Euklidesa polega na zabraniu ze stosu dowolnej dodatniej wielokrotności p kamieni (jeśli na stosie jest co najmniej p kamieni) albo dołożeniu do stosu dokładnie p kamieni — dołożyć kamienie można jednak tylko wtedy, gdy na stosie jest ich mniej niż p. Ruch Pitagorasa jest analogiczny, z tym że albo zabiera on wielokrotność q kamieni, albo dokłada dokładnie q kamieni. Wygrywa ten z graczy, który opróżni stos. Grę zaczyna Euklides.

Przyjaciele zastanawiają się, czy rozgryźli tę grę do końca. Pomóż im i napisz program, który będzie stwierdzał, jaki powinien być wynik rozgrywki w przypadku optymalnej gry obu graczy.

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą t ($1 \leqslant t \leqslant 1000$) oznaczającą liczbę zestawów testowych opisanych w dalszej części wejścia. Opis zestawu testowego składa się z jednego wiersza zawierającego trzy liczby całkowite p, q i n ($1 \leqslant p, q, n \leqslant 10^9$).

Wyjście

Na wyjściu powinno znaleźć się dokładnie t wierszy zawierających odpowiedzi do kolejnych zestawów testowych z wejścia. Odpowiedź powinna być jedną literą `E` (jeśli Euklides może doprowadzić do swojego zwycięstwa, niezależnie od ruchów Pitagorasa), `P` (jeśli Pitagoras może doprowadzić do swojego zwycięstwa, niezależnie od ruchów Euklidesa) lub `R` (jeśli gra będzie się toczyć w nieskończoność).

Przykład

Dla danych wejściowych:	poprawnym wynikiem jest:
4 3 2 1 2 3 1 3 4 5 2 4 3	P P E R

Wyjaśnienie przykładu: W rozgrywce z pierwszego zestawu testowego Euklides w swoim ruchu musi dołożyć na stos 3 kamienie. Dzięki temu Pitagoras może zabrać w swoim ruchu wszystkie 4 kamienie i tym samym wygrać.

Rozwiązanie

Oznaczmy przez d największy wspólny dzielnik liczb p i q. Zauważmy najpierw, że każdy z ruchów, które mogą wykonać gracze (zarówno dołożenie jak i zabranie kamieni), zmienia liczbę kamieni na stosie o wielokrotność d. Zatem liczba $r = n \bmod d$ (reszta z dzielenia liczby kamieni na stosie przez d) nie zmienia się przez cały czas trwania rozgrywki. Jeśli więc początkowo $r \neq 0$, to żaden z graczy nie będzie w stanie opróżnić stosu i gra toczyć się będzie w nieskończoność. Jak się potem okaże, jest to jedyna sytuacja, w której mamy remis.

Załóżmy zatem, że $r = 0$. Wtedy w trakcie rozgrywki liczba kamieni na stosie zawsze będzie podzielna przez d. Możemy więc podzielić każdą z liczb n, p i q przez d, co nie zmieni wyniku rozgrywki, ale spowoduje, że liczby p i q staną się względnie pierwsze.

Następnie, jeżeli $p = q$, to sytuacja także jest prosta. Ponieważ p i q są względnie pierwsze, to obie muszą być równe 1. Grę wygrywa więc w oczywisty sposób gracz, który zaczyna, zabierając w pierwszym ruchu wszystkie kamienie.

Aby rozwiązać zadanie, musimy uporać się z sytuacją, gdy $\mathrm{NWD}(p, q) = 1$ i $p \neq q$. Dla ułatwienia analizy zapomnimy o imionach graczy i o tym, który z nich zaczyna. Zamiast tego będziemy nazywać gracza, który może zdejmować ze stosu kamienie w liczbie sztuk podzielnej przez p, graczem P, a drugiego gracza — graczem Q. Rozważymy również niezależnie przypadek, gdy zaczyna gracz P, i przypadek, gdy zaczyna gracz Q; dzięki temu możemy bez straty ogólności przyjąć, że $p < q$.

Rozważamy przypadki

Rozwiązanie zadania polegać będzie na przeanalizowaniu kilku przypadków, z których każdy kolejny będziemy sprowadzać do poprzednich.

(A) Grę rozpoczyna gracz Q i $n < q$. Pokażemy, że w tym przypadku gracz P jest w stanie wygrać, zmuszając gracza Q za każdym razem do dokładania q kamieni.

W pierwszym ruchu gracz Q musi dołożyć do stosu q kamieni. Następnie gracz P może zdjąć największą dozwoloną liczbę kamieni (ponieważ $n + q > p$) tak, aby na stosie pozostało $(n + q) \bmod p < p < q$ kamieni. Jeśli $(n + q) \bmod p = 0$, to gracz P wygrywa, a w przeciwnym razie gracz Q znów jest zmuszony dołożyć q kamieni. Kontynuujmy to rozumowanie: w kolejnych ruchach gracz Q będzie zawsze dokładał q kamieni (wobec tego na pewno nie opróżni stosu i nie wygra), a po k-tym ruchu gracza P na stosie będzie $(n + kq) \bmod p$ kamieni (oczywiście, o ile gra nie skończy się wcześniej).

Aby dokończyć dowód, wystarczy zatem pokazać, że rozgrywka się zakończy, czyli że istnieje takie $k \geqslant 1$, że $(n + kq) \bmod p = 0$. Wykorzystamy tutaj rozszerzony algorytm Euklidesa, który przy okazji znajdowania największego wspólnego dzielnika względnie pierwszych liczb p i q znajdzie nam takie liczby całkowite l' i k', że $l'p + k'q = 1$, czyli $k'q \equiv 1 \pmod{p}$. Przyjmując $k = (-nk') \bmod p + p$, mamy $n + kq \equiv n - nk'q + pq \equiv 0 \pmod{p}$ i $k \geqslant p \geqslant 1$.

(B) Grę rozpoczyna gracz P i $n \geqslant p$. Wtedy gracz P może wygrać, bo w pierwszym ruchu może albo opróżnić stos (jeżeli p dzieli n), albo zostawić na stosie $n \bmod p < q$ kamieni i doprowadzić do przypadku (A).

(C) Grę rozpoczyna gracz P i $n < p$. Pokażemy, że w tym wypadku gracz P może wygrać wtedy i tylko wtedy, gdy n jest niepodzielne przez $q - p$.

W pierwszym ruchu gracz P musi dołożyć do stosu p kamieni. Jeśli $n+p < q$, to ruch ten doprowadza do przypadku (A) i gracz P wygrywa. W przeciwnym razie $q \leqslant n + p < 2q$, więc gracz Q jest zmuszony do zabrania q kamieni ze stosu. Po tych dwóch ruchach na stosie jest $n+p-q$ kamieni. Jeśli $n+p-q = 0$, to gra się kończy i wygrywa gracz Q. Podobnie, jeśli trzeci ruch doprowadzi do przypadku (A), to wygrywa gracz P, a jeśli nie, to po czwartym ruchu na stosie pozostaje $n - 2(q - p)$ kamieni. Kontynuujemy to rozumowanie. Jeśli przyjmiemy, że gracz P wygrywa, gdy po jego ruchu zostaje mniej niż q kamieni (przypadek (A)), to wszystkie kolejne ruchy obu graczy są jednoznacznie określone: gracz P zawsze dodaje p kamieni, a gracz Q zabiera q kamieni. Zauważmy, że po $2k$-tym ruchu mamy na stosie $n - k(q - p)$ kamieni, a po $(2k + 1)$-szym ruchu mamy $n - k(q - p) + p$ kamieni (oczywiście, o ile gra nie skończy się wcześniej).

Niech $l = \lfloor \frac{n}{q-p} \rfloor$. Po pierwsze, gracz Q nie wygra przed $2l$-tym ruchem, bo dla każdego całkowitego $l' < l$ mamy $n-l'(q-p) > n-l(q-p) \geqslant 0$. Podobnie gracz P nie wygra przed $(2l + 1)$-szym ruchem, bo dla każdego całkowitego $l' < l$ mamy $n - l'(q - p) + p \geqslant n - (l - 1)(q - p) + p = n - l(q - p) + q \geqslant q$.

Załóżmy, że $q-p$ dzieli n. Wtedy gracz Q opróżni stos po $2l$-tym ruchu w grze i wygra, bo wiemy, że gracz P nie może wygrać przed $(2l + 1)$-szym ruchem.

Jeśli z kolei $q-p$ nie dzieli n, to $n = l(q-p)+r$, dla $0 < r < q-p$ i po $(2l+1)$-szym ruchu na stosie zostanie $n - l(q - p) + p = l(q - p) + r - l(q - p) + p = r+p < (q-p)+p < q$ kamieni i gracz P wygra po tym ruchu (przypadek (A)). Wcześniejsze opróżnienie stosu przez gracza Q jest niemożliwe, bo n nie dzieli się przez $q - p$.

(D) Grę rozpoczyna gracz Q i $n \geqslant q$. W tym przypadku gracz P przegra wtedy i tylko wtedy, gdy liczba $r = n \bmod q$ spełnia jednocześnie warunki $r < p$ i r jest podzielne przez $q - p$.

Niech $n = kq + r$, gdzie $0 \leqslant r < q$. Jeśli gracz Q zdejmie mniej niż kq kamieni, to na stosie pozostanie co najmniej $q + r > p$ kamieni, więc ruch taki doprowadzi do przypadku (B) i gracz P wygra. Gracz Q musi zatem zostawić na stosie dokładnie r kamieni. Jeśli $r \geqslant p$, to mamy przypadek (B) i gracz P wygrywa. W przeciwnym razie mamy przypadek (C) i gracz P wygrywa wtedy i tylko wtedy, gdy $q - p$ nie dzieli r.

Podsumowując, rozwiązanie polega na wykonaniu kilku wstępnych testów, a następnie rozważeniu powyższych czterech przypadków. Dla każdego zestawu testowego z wejścia, po obliczeniu $d = \mathrm{NWD}(p, q)$ w czasie $O(\log p)$ wszystkie sprawdzenia zajmują czas stały.

F

Filary

★★★★

Autorzy zadania: Jakub Radoszewski, Tomasz Idziaszek
Opis rozwiązania: Tomasz Idziaszek
Dostępna pamięć: 256 MB
`https://oi.edu.pl/pl/archive/amppz/2014/fil`

Bajtazar jest administratorem dużej hali magazynowej. Przewidując ciężką zimę, postanowił zamontować w hali ogrzewanie podłogowe.

Plan hali jest prostokątem o parzystych wymiarach $n \times m$ podzielonym na kwadraty jednostkowe. Większość kwadratów jednostkowych stanowi powierzchnię magazynową, jednak niektóre z nich są zajmowane przez masywne filary, które stanowią dodatkowe zabezpieczenie konstrukcyjne hali. Każdy filar na planie hali zajmuje kwadrat 2×2, złożony z kwadratów jednostkowych. Filary nie są rozmieszczone zbyt gęsto — wiadomo, że środki każdych dwóch z nich są oddalone co najmniej o 6 jednostek (w metryce euklidesowej). Ponadto środek każdego filara jest oddalony od każdej ściany zewnętrznej hali również co najmniej o 3 jednostki.

Ogrzewanie zostanie zrealizowane za pomocą jednej rury grzewczej zainstalowanej pod podłogą hali. Rura ma przebiegać przez środki wszystkich kwadratów jednostkowych z pominięciem kwadratów jednostkowych zajmowanych przez filary. Rura musi na każdym odcinku biec równolegle do którejś ze ścian hali i może zakręcać tylko w środkach kwadratów jednostkowych. Rura musi zaczynać się i kończyć w tym samym miejscu. W tym miejscu wychłodzona woda z rury będzie wyprowadzana na zewnątrz, a wprowadzana będzie woda gorąca.

Bajtazar poprosił Cię o zaplanowanie przebiegu rury pod halą. Aby Ci pomóc, na planie hali wprowadził prostokątny układ współrzędnych, w którym odcięte należą do przedziału $[0, n]$, a rzędne do przedziału $[0, m]$. Współrzędne środków wszystkich kwadratów jednostkowych na hali są liczbami postaci $k + \frac{1}{2}$ dla $k \in \mathbb{N}$.

Wejście

Pierwszy wiersz wejścia zawiera trzy liczby całkowite n, m oraz f ($1 \leqslant n, m \leqslant 1000$ oraz n i m są parzyste) oznaczające wymiary hali oraz liczbę filarów. Każdy z kolejnych f wierszy zawiera dwie liczby całkowite x_i i y_i ($0 \leqslant x_i \leqslant n$, $0 \leqslant y_i \leqslant m$) oznaczające współrzędne środka i-tego filara.

Wyjście

W pierwszym wierszu wyjścia Twój program powinien wypisać jedno słowo `TAK` lub `NIE` w zależności od tego, czy jest możliwa realizacja ogrzewania podłogowego hali zgodnie z wymaganiami Bajtazara, czy też nie. Jeśli odpowiedzią jest `TAK`, w drugim wierszu powinien znaleźć się opis przykładowego planu przebiegu rury w postaci ciągu $nm - 4f$ liter. Zakładamy umownie, że początek rury znajduje się w punkcie o współrzędnych $(\frac{1}{2}, \frac{1}{2})$. Kolejne fragmenty rury są oznaczane następująco: przejście o wektor $[0, 1]$ oznaczamy literą `G`, o wektor $[0, -1]$ — literą `D`, o wektor $[1, 0]$ — literą `P`, a o wektor $[-1, 0]$ — literą `L`. Jeśli istnieje wiele poprawnych odpowiedzi, Twój program powinien wypisać dowolną z nich.

Przykład

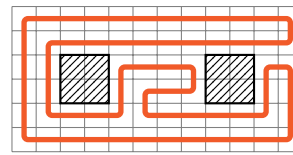
Dla danych wejściowych:

```
12 6 2
3 3
9 3
```

poprawnym wynikiem jest:

```
TAK
PPPPPPPPPPPGGGLDDLLLLLGPPGLLLDDLLLGGGPPPPPPPPPPGLLLLLLLLLLLDDDDD
```

Podane w przykładzie wyjście odpowiada poniższemu rysunkowi:



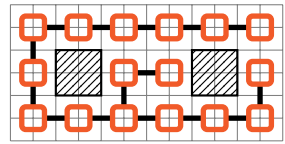
Rozwiązanie

Problem zawarty w zadaniu *Filary* można wyrazić w języku teorii grafów. Rozważmy graf, w którym wierzchołkami są kwadraty jednostkowe niezajęte przez filary, a krawędzie łączą kwadraty jednostkowe mające wspólny bok. Zadanie polega na znalezieniu w tym grafie cyklu Hamiltona (czyli takiego, który przechodzi przez każdy wierzchołek dokładnie raz). W pełnej ogólności (tzn. jeśli chcemy znaleźć algorytm dający rozwiązanie dla dowolnego grafu) jest to problem NP-zupełny, zatem należy on do klasy problemów, dla których nie jest znane efektywne rozwiązanie. Skorzystamy więc z tego, że graf z naszego zadania ma specjalną postać*.

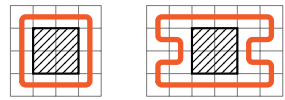
Na początek rozważmy prostszy wariant zadania, w którym środek każdego filaru jest odległy od każdego z brzegów prostokąta o nieparzystą liczbę. Innymi słowy, jeśli posklejamy wszystkie kwadraty jednostkowe w kawałki o rozmiarach 2×2, to każdy z takich kawałków będzie albo w całości pokryty przez filar, albo też nie będzie zawierał żadnego pola filaru. W każdym kawałku drugiego typu umieszczamy cykl o długości 4 (patrz rysunek 1). Następnie tworzymy graf, którego wierzchołkami są kawałki (połączone krawędzią, jeśli mają wspólny bok) i znajdujemy w nim dowolne drzewo rozpinające. Porównując rysunek 1 z rysunkiem z treści zadania, nietrudno zauważyć, jak za pomocą tego drzewa połączyć małe cykle, by dały rozwiązanie. Złożoność czasowa rozwiązania to $O(nm + f)$.

Rozwiązując zadanie bez uproszczeń, będziemy musieli dopuścić podział prostokąta na trochę mniej regularne kawałki. Filary odległe od każdego z brzegów prostokąta o parzystą liczbę umieścimy w kawałkach rozmiaru 4×4, natomiast filary odległe od brzegów o liczby o różnej parzystości umieścimy w kawałkach 4×6 lub 6×4 (patrz rysunek 2). Założenia o odległościach pomiędzy środkami filarów wystarczają do tego, by nasze kawałki nie nachodziły na siebie. Dalsza część rozwiązania z budowaniem drzewa rozpinającego i łączeniem cykli jest analogiczna.

*W szczególności jest planarny, ale to nam nie wystarczy, gdyż znajdowanie cyklu Hamiltona w dowolnym grafie planarnym jest również problemem NP-zupełnym.



Rysunek 1. W każdym z 16 kawałków niezawierających filarów umieszczamy cykl. Drzewo rozpinające grafu kawałków zaznaczono pogrubionymi czarnymi krawędziami.

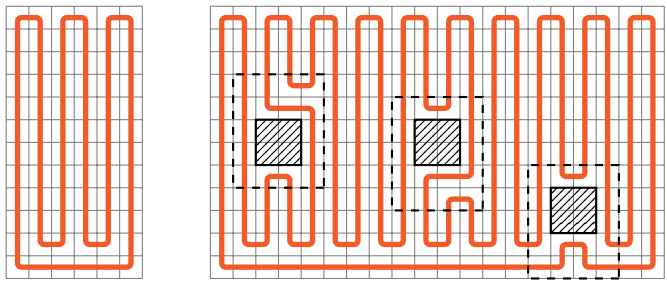


Rysunek 2. Otoczenie filaru cyklem w zależności od rozmiaru kawałka.

Zadanie ma też prostsze rozwiązanie. Na początek wypełniamy pełny prostokąt „kaloryferem" jak na rysunku 3 (cykl przechodzi przez kwadraty jednostkowe położone wzdłuż dolnego brzegu, a następnie kolejnymi kolumnami, zmieniając kierunek).

Następnie umieszczamy kolejne filary, za każdym razem poprawiając cykl lokalnie. Na rysunku 3 są przedstawione trzy przypadki, które musimy w tym celu rozważyć. W pierwszym z nich (po lewej) odległość środka filaru od lewego brzegu prostokąta jest nieparzysta, w pozostałych przypadkach jest parzysta (ostatni przypadek występuje, gdy odległość od dolnego brzegu jest równa 3).

Co ciekawe, jeśli zwiększymy do 4 minimalną odległość środków filarów od brzegów prostokąta, to rozwiązanie stanie się jeszcze prostsze i nie tylko dlatego, że odpadnie nam trzeci przypadek, ale też z uwagi na alternatywną implementację. Umieszczamy wszystkie filary, a następnie, zaczynając od prawego dolnego rogu, budujemy cykl zachłannie, zawsze wykonując pierwszy możliwy ruch z następującej listy: lewo, góra, dół, prawo. Łatwo sprawdzić, że dla prostokąta z rysunku 3 bez dolnego filaru algorytm lokalnie poprawiający cykl i algorytm zachłanny dają taki sam wynik.



Rysunek 3. Po lewej: początkowe wypełnienie pełnego prostokąta w drugim rozwiązaniu. Po prawej: trzy przypadki lokalnych poprawek „kaloryfera".

G

Globalne ocieplenie

★★★

Autorzy zadania: Jacek Tomasiewicz, Tomasz Idziaszek
Opis rozwiązania: Tomasz Idziaszek
Dostępna pamięć: 256 MB
`https://oi.edu.pl/pl/archive/amppz/2014/glo`

Profesor Bajtoni przygotowuje raport dla Międzybajtockiego Zespołu ds. Zmian Klimatu, z którego ma jednoznacznie wynikać, jaki jest wpływ mieszkańców Bajtocji na zmiany klimatyczne w regionie. Wprawdzie profesor ma sporo danych empirycznych, jednak aby przeniknąć do mediów głównego nurtu, nie wystarczą merytoryczne argumenty, ale równie ważne jest zaprezentowanie ich w dobitny sposób. W tym celu chce z rozmysłem wybrać dane, które przedstawi na głównym wykresie w raporcie.

Kluczowy wykres będzie zawierał informacje o średniej temperaturze powietrza na przestrzeni lat. Profesor dysponuje danymi dotyczącymi średniej rocznej temperatury dla ostatnich n lat. Chce okrasić ten wykres komentarzem w stylu „w roku r_{min} temperatura była najniższa, a w roku r_{max} była najwyższa, wobec tego jasno widać, że...". Niestety obawia się, że taka sama minimalna lub maksymalna temperatura mogła wystąpić kilkukrotnie, co osłabiłoby dobitność tego zdania.

Profesor postanowił zatem zaprezentować na wykresie jedynie część danych. Chce teraz wybrać taki przedział lat, że *w tym przedziale* jest dokładnie jeden rok z minimalną temperaturą w tym przedziale oraz dokładnie jeden rok z maksymalną temperaturą w tym przedziale. Wybrany przedział może nie zawierać roku z maksymalną lub minimalną średnią temperaturą w ciągu ostatnich n lat (lub żadnego z nich). Oczywiście profesor chciałby umieścić na wykresie jak najwięcej danych, więc interesuje go jak najdłuższy przedział lat.

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 500\,000$), oznaczająca liczbę lat, dla których znamy średnie temperatury. W drugim wierszu znajduje się ciąg n liczb całkowitych $t_1, t_2, \ldots, t_n$ ($-10^9 \leqslant t_i \leqslant 10^9$). Liczba t_i oznacza średnią temperaturę w i-tym roku.

Wyjście

Na wyjście należy wypisać dwie liczby całkowite l i k. Oznaczają one, że najdłuższy przedział spełniający warunki profesora ma długość l lat, a najwcześniejszy rok, w którym taki przedział się zaczyna, to k.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
6 4
```

Wyjaśnienie przykładu: Na wykresie zostaną przedstawione temperatury 5, 2, 3, 4, 6, 3. W tym przedziale jest dokładnie jeden rok z minimalną temperaturą 2 i jeden rok z maksymalną temperaturą 6.

Rozwiązanie

Dany jest ciąg liczb $t_1, \ldots, t_n$, w którym chcemy znaleźć najdłuższy spójny podciąg o następującej własności: w tym podciągu zarówno najmniejsza, jak i największa liczba występują dokładnie raz. Ponieważ spójny podciąg jest wyznaczony przez pewien przedział indeksów ciągu, w dalszej części opisu skupimy się właśnie na szukaniu przedziału wyznaczającego poszukiwany podciąg.

Ustalmy pozycję i i zastanówmy się, jak wygląda najdłuższy przedział T_i o powyższej własności, który zawiera liczbę t_i jako swoje jedyne minimum. Rozważmy zbiór pozycji, na których znajdują się liczby nie większe niż t_i. Oczywiście żadna pozycja z tego zbioru (poza pozycją i) nie może należeć do przedziału T_i. Oznaczmy zatem przez $l_{i,1}$ największą pozycję z tego zbioru znajdującą się na lewo od i, a przez $r_{i,1}$ najmniejszą pozycję na prawo od i. Aby te pozycje zawsze istniały, przyjmijmy, że $t_0 = t_{n+1} = -\infty$. Przedział T_i musi zawierać się w otwartym przedziale $(l_{i,1}, r_{i,1})$.

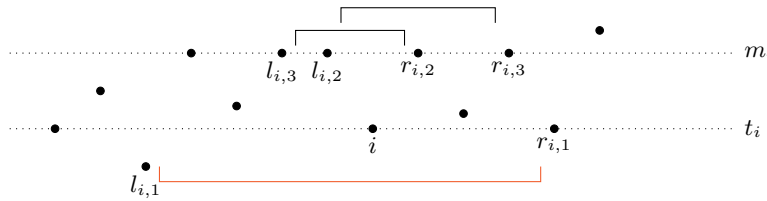
Jeśli przedział $(l_{i,1}, r_{i,1})$ zawiera jedynie liczbę i, to przedział T_i jest oczywiście również jednoelementowy. Dlatego w dalszej części opisu zakładamy, że przedział $(l_{i,1}, r_{i,1})$ jest co najmniej dwuelementowy, i wówczas także T_i składa się z co najmniej dwóch elementów. Oznaczmy przez m_i największą liczbę z pozycji w przedziale $(l_{i,1}, r_{i,1})$. Łatwo przekonać się, że przedział T_i musi zawierać dokładnie jedno wystąpienie liczby m_i — wystarczy zauważyć, że każdy poprawny (tj. zawierający dokładnie jedno minimum na pozycji i i jedno maksimum) przedział, który nie ma takiej własności, można przedłużyć tak, by nadal był poprawny i zawierał jedno wystąpienie m_i.

Wymienione dotychczas ograniczenia na przedział T_i są wystarczające, by zapewnić, że w tym przedziale znajduje się dokładnie jedno minimum i jedno maksimum. A zatem T_i to najdłuższy przedział, który spełnia następujące warunki:

- zawiera pozycję i,
- jest zawarty w przedziale otwartym $(l_{i,1}, r_{i,1})$,
- zawiera dokładnie jedno wystąpienie m_i.

Aby skonstruować przedział T_i, sprawdzamy dwie możliwości: wystąpienie m_i należące do T_i może się znajdować na lewo lub na prawo od pozycji i. Dla każdej z tych opcji rozważamy przedział, którego końcami są wystąpienie m_i oraz pozycja i, i następnie rozszerzamy ów przedział maksymalnie w obie strony. Rozszerzanie skończyć musimy albo gdy natrafimy na koniec przedziału $(l_{i,1}, r_{i,1})$, albo też przed kolejnym wystąpieniem m_i. W ten sposób otrzymamy dwóch kandydatów, spośród których wybierzemy przedział T_i.

Czas na bardziej formalny opis. Niech $l_{i,2}$ oraz $l_{i,3}$ ($l_{i,3} < l_{i,2}$) będą skrajnie prawymi pozycjami w przedziale $(l_{i,1}, i)$, na których występuje m_i (patrz rysunek 1).



Rysunek 1. Pozycja i, sześć interesujących pozycji ograniczających umiejscowienie przedziału T_i oraz dwa przedziały będące kandydatami na przedział T_i. Wyrazy ciągu $t_1, \ldots, t_n$ oznaczone są czarnymi kropkami. Wysokość, na której umieszczona została kropka reprezentująca t_i, odpowiada wartości wyrazu t_i.

Może się zdarzyć, że na pozycjach z przedziału $(l_{i,1}, i)$ znajduje się tylko jedna liczba m_i lub nie ma tam żadnej takiej liczby. Wtedy przyjmujemy, że pozycja $l_{i,3}$ (lub odpowiednio obie pozycje $l_{i,2}$ i $l_{i,3}$) są równe $l_{i,1}$. Symetrycznie definiujemy pozycje $r_{i,2}$ oraz $r_{i,3}$ ($r_{i,2} \leqslant r_{i,3}$).

Przedział T_i musi zawierać pozycję i oraz dokładnie jedno wystąpienie m_i. Zatem jedynymi kandydatami na przedział T_i są przedziały $(l_{i,3}, r_{i,2})$ oraz $(l_{i,2}, r_{i,3})$.

Implementacja

Pozostaje kwestia obliczenia zdefiniowanych powyżej pozycji. Wszystkie pozycje $l_{i,1}$ możemy wyznaczyć w sumarycznym czasie $O(n)$, korzystając ze stosu, na którym trzymamy kandydatów na te pozycje*. Powiemy, że pozycja j jest *widoczna* na lewo z pozycji i, jeśli $j < i$, $t_j \leqslant t_i$ oraz wszystkie liczby znajdujące się pomiędzy tymi pozycjami są nie mniejsze niż t_j. Przy takiej definicji łatwo widać, że $l_{i,1}$ jest największą pozycją widoczną na lewo z pozycji i.

Przeglądamy kolejno wyrazy ciągu od lewej do prawej, utrzymując niezmiennik, że po przetworzeniu wyrazów $t_0, \ldots, t_i$ na stosie znajduje się pozycja i oraz wszystkie pozycje widoczne na lewo z i. Pozycje ułożone są na stosie kolejno. Największa z nich znajduje się na czubku stosu. Gdy analizujemy wyraz t_i, usuwamy ze stosu wszystkie pozycje wyrazów większych od t_i. Oczywiście nie mogą być one pozycją $l_{i,1}$, ale nie mogą też być $l_{j,1}$ dla żadnej pozycji $j > i$, bo pozycja i jest w tym wypadku lepszym kandydatem. Element, który po tych usunięciach znajdzie się na szczycie stosu, to pozycja $l_{i,1}$. Następnie wrzucamy pozycję i na stos.

Pozycję $r_{i,1}$ wyznaczamy analogicznie, przeglądając ciąg od prawej do lewej. Z kolei pozycje $l_{i,2}$ i $l_{i,3}$ możemy wyznaczać, korzystając z drzewa przedziałowego, które udostępnia operacje znajdowania skrajnie prawego maksimum w danym przedziale. Symetryczne drzewo przedziałowe pozwoli nam wyznaczać pozycje $r_{i,2}$ i $r_{i,3}$. Dzięki temu uzyskujemy algorytm o złożoności czasowej $O(n \log n)$.

Istnieje jednak prostsze rozwiązanie, które pozwoli nam wyznaczać te pozycje w sumarycznym czasie $O(n)$. Wymaga ono nieznacznej modyfikacji naszego stosu. Dotychczas na stosie trzymaliśmy elementy $a_0 = 0,\ a_1,\ \ldots,\ a_s = i$, będące pozycjami widocznymi na lewo z pozycji i. Teraz dodatkowo wraz z pozycją a_j trzy-

*Inny zapis algorytmu rozwiązującego prawie identyczny problem można znaleźć na początku opisu rozwiązania zadania *Prostokąt arytmetyczny* w książce.

mamy zbiór A_j, zawierający pozycje największych wyrazów ciągu leżących pomiędzy pozycjami a_{j-1} oraz a_j. Łatwo zauważyć, że $l_{i,2}$ i $l_{i,3}$ są dwiema największymi liczbami ze zbioru A_s (o ile tylko A_s ma co najmniej dwa elementy). Utrzymywanie zbiorów A_j przebiega następująco: jeśli usuwamy pozycję a_j ze stosu, to musimy uaktualnić zbiór A_{j-1} przez rozważenie pozycji a_j oraz wszystkich pozycji ze zbioru A_j.

Ponieważ zbiory A_j są nam potrzebne do wyznaczania dwóch pozycji, więc wystarczy pamiętać jedynie dwie największe liczby z tych zbiorów. Tym samym pojedyncze uaktualnienie zbioru wykonujemy w czasie stałym, co powoduje, że ta faza algorytmu działa w sumarycznym czasie $O(n)$.

Na koniec musimy wprowadzić drobną poprawkę: ponieważ pozycje $l_{i,2}$, $l_{i,3}$ oraz $r_{i,2}$, $r_{i,3}$ wyznaczaliśmy niezależnie, może się zdarzyć, że tylko jedna para odpowiada maksymalnej wartości m_i. W takim przypadku pozycje drugiej pary należy zamienić na odpowiednio $l_{i,1}$ lub $r_{i,1}$.

Ostatecznie algorytm wygląda następująco. Najpierw obliczamy pozycje $l_{i,j}$, $r_{i,j}$ dla wszystkich pozycji i oraz $j \in \{1, 2, 3\}$. Następnie dla każdej pozycji i wyznaczamy dwóch kandydatów na przedział T_i. Na koniec spośród wszystkich tych kandydatów wybieramy najdłuższy przedział, a w przypadku remisu, ten zaczynający się najwcześniej. Złożoność czasowa algorytmu to $O(n)$.

Hit sezonu

Autorzy: Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń
Opis rozwiązania: Tomasz Kociumaka
Dostępna pamięć: 256 MB
`https://oi.edu.pl/pl/archive/amppz/2014/hit`

Bajtocki Zakład Poligraficzny (BZP) otrzymał duże zlecenie na produkcję prążkowanych tapet, stanowiących hit sezonu w projektowaniu wnętrz. Każda tapeta składa się z n jednakowej szerokości pionowych pasków w kolorach: czerwonym, zielonym i niebieskim. BZP ma zająć się zaprojektowaniem oraz wydrukowaniem tapet. Zleceniodawca określił barwy niektórych pasków na tapecie, natomiast w przypadku pozostałych pasków pozostawił BZP pełną dowolność.

Do wydruku tapet w BZP używa się matryc drukujących pewną liczbę kolejnych pasków na tapecie. Matryca ma określone barwy każdego z drukowanych pasków i może być krótsza niż cała tapeta. Przy każdym przyłożeniu matrycy jej paski muszą pokrywać się z paskami tapety; drukują się wówczas wszystkie paski matrycy. W ten sposób jeden pasek tapety może zostać zadrukowany więcej niż raz. W przypadku, gdy dany pasek zostanie zadrukowany różnymi barwami, jego ostateczny kolor będzie stanowił mieszankę tych barw. Matryca działa w tylko jednej orientacji i nie wolno jej w żaden sposób obracać.

Pracownicy BZP, niezależnie od posiadanego wyczucia estetyki, chcieliby przede wszystkim zaprojektować możliwie najkrótszą matrycę, która pozwoli wydrukować całą tapetę. Muszą oni pamiętać o tym, że w przypadku pasków określonych przez zleceniodawcę muszą użyć czystej barwy, bez domieszki innych barw. Innymi słowy, przy każdym przyłożeniu matrycy pokrywającym taki pasek, barwa paska na matrycy musi być dokładnie taka, jak określona przez zleceniodawcę. Żaden pasek tapety nie może pozostać bezbarwny.

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita t ($1 \leqslant t \leqslant 10$) oznaczająca liczbę zestawów testowych. Każdy z kolejnych t wierszy opisuje jeden zestaw testowy i zawiera napis złożony z wielkich liter `R`, `G`, `B` oraz gwiazdek (`*`), określający oczekiwany wygląd tapety. Poszczególne litery oznaczają barwy pasków, natomiast gwiazdki oznaczają paski, których barwa nie została określona przez zleceniodawcę. Napis jest niepusty, składa się z co najwyżej 3000 znaków i zawiera co najwyżej 19 gwiazdek.

Wyjście

Twój program powinien wypisać dla każdego zestawu testowego jeden wiersz zawierający jeden napis złożony z liter `R`, `G`, `B`: matrycę o minimalnej długości, która pozwala wydrukować żądaną tapetę. Jeśli dla danego zestawu testowego istnieje wiele poprawnych rozwiązań, Twój program powinien wypisać dowolne z nich.

Przykład

Dla danych wejściowych:

```
1
RRG*R*BRR**B
```

poprawnym wynikiem jest:

```
RRGB
```

Rozwiązanie

Zanim przejdziemy do opisu rozwiązania zadania, wprowadźmy kilka pojęć. Podstawowym obiektem naszych rozważań będą *słowa częściowe*, w których poza zwykłymi literami (w naszym przypadku `R`, `G` i `B`) może występować symbol uniwersalny (tzw. *blank*), oznaczany tutaj przez `*`. Mówimy, że dwa znaki (ze zbioru $\{\texttt{R}, \texttt{G}, \texttt{B}, *\}$) *pasują* do siebie, jeśli są równe lub gdy jeden z nich jest blankiem. Relację tę oznaczamy przez $\approx$. Można ją rozszerzyć na słowa częściowe tej samej długości, wymagając, by pasowały do siebie odpowiadające sobie znaki tych słów. Przykładowo, $\texttt{R*B} \approx \texttt{RG*}$ oraz $\texttt{R*B} \approx \texttt{*RB}$, jednakże $\texttt{RG*} \not\approx \texttt{*RB}$, gdyż na drugiej pozycji mamy różne litery `G` oraz `R`. Relacja $\approx$ nie jest więc przechodnia. Prawdą jest natomiast, że słowa częściowe pasują do siebie wtedy i tylko wtedy, gdy oba pasują do pewnego *pełnego* słowa (niezawierającego blanków) nazywanego ich *wspólnym wypełnieniem*. Na przykład wspólnym wypełnieniem słów `R*B` i `RG*` jest `RGB`.

Za pomocą relacji $\approx$ łatwo zdefiniować na słowach częściowych problem wyszukiwania wzorca w tekście. Problem ten polega na znalezieniu fragmentów tekstu pasujących do wzorca. Celem naszego zadania jest jednak skonstruowanie najkrótszego *szablonu* (matrycy), czyli wzorca, którego wystąpienia pokrywają cały tekst (projekt tapety). Wymagamy przy tym, by szablon był pełnym słowem (w przeciwnym przypadku trywialnym rozwiązaniem byłby jednoznakowy szablon `*`). Problem ten w wersji dla zwykłych słów był niegdyś tematem zadania na Olimpiadzie Informatycznej (*Szablon*, II etap XII OI). Warto jednak zauważyć, że biorąc po prostu najkrótszy spośród szablonów wszystkich pełnych słów pasujących do tekstu, niekoniecznie otrzymamy najkrótszy szablon tekstu, co widać już nawet w przykładzie z treści zadania.

Weryfikacja szablonu

W opisie rozwiązania będziemy zakładać, że litery słów są ponumerowane od zera. Na początek zastanówmy się, jak można sprawdzić, czy dany wzorzec S długości m jest szablonem słowa częściowego T długości n. Aby tak było, S musi pokrywać początkową i ostatnią pozycję T, a więc pasować do prefiksu i do sufiksu słowa T. Ponadto każde dwa kolejne wystąpienia S muszą na siebie nachodzić lub występować bezpośrednio po sobie, gdyż w przeciwnym przypadku pozycje znajdujące się pomiędzy nimi nie byłyby pokryte. Aby sformalizować te warunki, zdefiniujmy zbiór $\mathrm{Occ}(S, T)$ pozycji, na których zaczynają się wystąpienia S w T, oraz funkcję MaxGap, która zbiorowi $A = \{a_1, \ldots, a_k\}$, gdzie $a_1 < \ldots < a_k$, przypisuje wartość $\max\{a_{i+1} - a_i : i = 1, \ldots, k-1\}$.

Obserwacja 1. Wzorzec S jest szablonem słowa częściowego T wtedy i tylko wtedy, gdy spełnione są jednocześnie następujące warunki:

(a) $0 \in \mathrm{Occ}(S, T)$,

(b) $\mathrm{MaxGap}(\mathrm{Occ}(S, T) \cup \{|T|\}) \leqslant |S|$.

Aby skorzystać z powyższej obserwacji, wystarczy wyznaczyć zbiór $\mathrm{Occ}(S, T)$, co łatwo uczynić w czasie $O(nm)^\star$, a następnie sprawdzić kryterium sformułowane powyżej.

Rozwiązanie naiwne

W zadaniu nie mamy danego kandydata na szablon, lecz musimy wyznaczyć najkrótszy szablon spośród wszystkich słów. Warunek (a) Obserwacji 1 ogranicza zbiór kandydatów do słów pasujących do pewnego prefiksu tekstu T. Takich słów jest $O(3^k n)$, gdyż każde jest jednoznacznie określone przez swoją długość oraz wypełnienie tekstu T, tj. przez wybór liter na k pozycjach, na których w T występuje $*$. Korzystając z opisanego wyżej algorytmu weryfikacji, otrzymujemy naiwne rozwiązanie działające w czasie $O(3^k n^3)$.

Wydajna implementacja

Nietrudno dostrzec, że w zbiorze kandydatów rozważamy wiele dość podobnych słów. Intuicja podpowiada więc, że zamiast niezależnie wyszukiwać je jako kolejne wzorce, można uwspólnić część obliczeń. Jak się wkrótce przekonamy, wszystkie szablony można wyznaczyć zasadniczo w czasie proporcjonalnym do liczby kandydatów.

Pierwszy krok

Zacznijmy od prostej obserwacji: gdy ustalimy jakieś wypełnienie T, czyli pełne słowo $\bar{T}$ pasujące do T, to kandydaci odpowiadający temu wypełnieniu stanowią po prostu zbiór prefiksów $\bar{T}$. Prefiksy możemy łatwo przetworzyć w kolejności rosnących długości, utrzymując zbiór wystąpień Occ. Zbiór Occ będzie się w każdym kroku zmniejszał z $\mathrm{Occ}_{m-1} = \mathrm{Occ}(\bar{T}[0..m-1], T)$ do $\mathrm{Occ}_m = \mathrm{Occ}(\bar{T}[0..m], T)$ (przez $\bar{T}[0..m]$ oznaczamy tu fragment słowa $\bar{T}$ składający się z liter z pozycji $0, \ldots, m$). Wystarczy zatem sprawdzić, które wystąpienia w Occ_{m-1} przedłużają się o literkę $\bar{T}[m]$, co oczywiście można wykonać w czasie proporcjonalnym do wielkości zbioru Occ_{m-1}. Weryfikacja, czy $\bar{T}[0..m]$ jest szablonem, wymaga przejrzenia zbioru Occ_m, a więc jeden krok implementujemy w czasie $O(n)$. Daje to algorytm o łącznej złożoności $O(3^k n^2)$.

Tablica Pref

Wspólne elementy obliczeń dla różnych wypełnień dają pole do kolejnej optymalizacji. Jeśli przedłużamy wzorzec $\bar{T}[0..m-1]$ do wzorca $\bar{T}[0..m]$, to, o ile $T[m] \neq *$,

*Można też użyć algorytmu o złożoności $O(n \log n)$, który, jak się później okaże, w naszym zadaniu nie jest potrzebny. Czytelników, którzy go nie znają, zachęcamy jednak, by w ramach ciekawego ćwiczenia spróbowali go wymyślić. Najlepiej skoncentrować się na początku na alfabecie binarnym.

po prostu dokładamy do wzorca literę $T[m]$. Wystąpienie na ustalonej pozycji $i \in \mathrm{Occ}_{m-1}$ przedłuża się o $T[m]$ wtedy i tylko wtedy, gdy $T[i+m] \approx T[m]$. Warunek ten nie zależy od tego, które wypełnienie $\bar{T}$ badamy, choć od wypełnienia może zależeć, czy w ogóle $i \in \mathrm{Occ}_{m-1}$.

Wiemy zatem, że $T[i+p] \not\approx T[p]$ gwarantuje $i \notin \mathrm{Occ}_m$ dla $m \geqslant p$. Naturalnym pomysłem jest więc wyznaczenie tablicy Pref, w której

$$\mathrm{Pref}[i] = \min(\{n-i\} \cup \{p : T[i+p] \not\approx T[p]\})$$

oznacza długość najdłuższego prefiksu T, który występuje w T na pozycji i. Tablicę Pref można wyznaczyć w czasie $O(nk)$, lecz nam wystarczy naiwny algorytm kwadratowy. Oprócz tego będziemy potrzebować tablicy odwrotnej Pref^{-1}, która dla danego p przechowuje wszystkie pozycje i spełniające $\mathrm{Pref}[i] = p$.

Jeśli $T[m] \neq *$, to mamy $\mathrm{Occ}_m = \mathrm{Occ}_{m-1} \setminus \mathrm{Pref}^{-1}[m-1]$, co pozwala bardzo prosto zaktualizować zbiór Occ. Jednakże naszym zadaniem jest także sprawdzenie, czy $\bar{T}[0..m]$ jest szablonem, za pomocą kryterium podanego w Obserwacji 1. Oczywiście musimy tylko zweryfikować warunek (b), do czego przydatna jest znajomość wartości $\mathrm{MaxGap}(\mathrm{Occ} \cup \{|T|\})$. Okazuje się, że jej utrzymywanie nie jest trudne. Musimy umieć uaktualniać tę wartość, gdy z $\mathrm{Occ} \cup \{|T|\}$ usuwamy pewną pozycję. Nigdy nie usuwamy 0 ani $|T|$, więc MaxGap może tylko wzrosnąć. Przy usuwaniu wystarczy zlokalizować poprzedni i następny element w zbiorze $\mathrm{Occ} \cup \{|T|\}$, a następnie rozważyć różnicę tych liczb jako potencjalną nową wartość MaxGap. Aby efektywnie zaimplementować tę operację, zbiór $\mathrm{Occ} \cup \{|T|\}$ możemy utrzymywać jako listę dwukierunkową wzbogaconą o tablicę, która dla danej pozycji i przechowuje wskaźnik do odpowiedniego elementu listy (lub null, jeśli i nie należy do zbioru).

Dla ustalonego wypełnienia każdy element usuniemy ze zbioru Occ tylko raz, więc koszt operacji na Occ to $O(n)$. W łącznym czasie liniowym działa też wyznaczenie pozycji do usunięcia na podstawie Pref^{-1}. W przypadku $T[m] = *$ nie możemy skorzystać z tablicy Pref^{-1}, więc naiwnie sprawdzamy, dla których pozycji i zachodzi $\bar{T}[m] \approx T[i+m]$, co zajmuje łącznie czas $O(kn)$. Sumując po wszystkich wypełnieniach i uwzględniając konstrukcję tablicy Pref, otrzymujemy całkowity czas działania algorytmu równy $O(n^2 + 3^k kn)$.

Algorytm rozgałęziający się

Kolejna optymalizacja opiera się na fakcie, że prefiksy dwóch różnych wypełnień pokrywają się aż do pierwszego blanku wypełnionego różnymi literami. Zanim wartość m dojdzie do tej pozycji, obliczenia dla takich wypełnień można prowadzić wspólnie. Innymi słowy, stworzymy algorytm, który prefiks $\bar{T}[0..m-1]$ rozszerzy o $T[m]$, jeśli jest to litera, a w przypadku $T[m] = *$ rozgałęzi się na trzy instancje, z których każda przyjmie inną wartość $\bar{T}[m]$. Operacja taka wymaga skopiowania stanu algorytmu (czyli reprezentacji zbioru $\mathrm{Occ} \cup \{|T|\}$), co zajmuje czas $O(n)$. Poza tym szczegółem, program działa identycznie jak poprzednie rozwiązanie. Wiemy jednak, że pomiędzy kolejnymi rozgałęzieniami tylko raz musimy naiwnie aktualizować Occ (tuż po rozgałęzieniu), skąd czas działania pojedynczej instancji algorytmu to, nie licząc wywołań rekurencyjnych, $O(n)$.

Łączna złożoność czasowa instancji wywołanej po wypełnieniu j-tego blanku

wynosi zatem:

$$C(n,j) = \begin{cases} O(n) & \text{jeśli } j = k \\ O(n) + 3C(n,j+1) & \text{w przeciwnym przypadku,} \end{cases}$$

co rozwiązuje się do $C(n,0) = O(3^k n)$. Jeśli dodamy do tego czas potrzebny na obliczenia wstępne, otrzymamy złożoność czasową całego algorytmu $O(n^2 + 3^k n)$.

Rozwiązanie wzorcowe

Dominującym składnikiem czasu działania naszego aktualnego rozwiązania jest $O(3^k n)$, czyli liczba kandydatów, których sprawdzamy. Musimy zatem ograniczyć rozmiar tego zbioru. Zauważmy, że gdybyśmy wiedzieli, iż szukany szablon ma długość co najwyżej m, kandydatów do rozważenia byłoby $O(3^{k_m} m)$, gdzie k_m oznacza liczbę blanków wśród pierwszych m znaków tekstu. Co więcej, nasz algorytm działałby w czasie $O(n^2 + 3^{k_m} n)$. Dzięki symetrii rozważanego problemu ze względu na odwrócenie tekstu i wzorca, moglibyśmy także jako k_m przyjąć liczbę blanków wśród ostatnich m symboli tekstu. Dla $m = \lfloor \frac{n}{2} \rfloor$ mamy zatem $k_m \leqslant \lfloor \frac{k}{2} \rfloor$, gdyż któraś połowa tekstu zawiera nie więcej niż $\lfloor \frac{k}{2} \rfloor$ blanków. W ten sposób otrzymujemy algorytm działający dla takich szablonów w czasie $O(n^2 + 3^{k/2} n)$, a więc na nasze potrzeby wystarczająco szybko.

Może się jednak zdarzyć, że otrzymany tekst T nie ma szablonu długości co najwyżej $\frac{n}{2}$. Szczęśliwie w takim przypadku bardzo łatwo rozstrzygnąć, czy pewien wzorzec długości $m \geqslant \frac{n}{2}$ jest szablonem. Ponieważ prefiks i sufiks długości m pokrywają razem wszystkie pozycje tekstu, wystarczy sprawdzić, czy wzorzec pasuje do obydwu. Innymi słowy, szablon długości $m \geqslant \frac{n}{2}$ istnieje wtedy i tylko wtedy, gdy prefiks i sufiks długości m mają wspólne wypełnienie. Jak już zauważyliśmy na początku naszych rozważań, możemy równoważnie sprawdzić, czy $T[0..m-1] \approx T[n-m..n-1]$, a więc czy $\mathrm{Pref}[n-m] = m$. W przypadku, gdy warunek ten jest spełniony, znalezienie wspólnego wypełnienia jest trywialne. Zatem najkrótszy szablon długości $m \geqslant \frac{n}{2}$ można skonstruować w czasie liniowym przy użyciu tablicy Pref, co w połączeniu z wcześniej opisanym rozwiązaniem dla krótszych szablonów daje czas $O(n^2 + 3^{k/2} n)$. Złożoność pamięciowa to $O(kn)$, lecz staranniejsza implementacja rozgałęzień pozwoliłaby zmniejszyć tę wartość do $O(n)$.

Posłowie

Okazuje się, że przedstawiony powyżej algorytm, choć wystarczający na potrzeby zadania, dla małych k jest daleki od optymalnego. Istnieje bowiem rozwiązanie działające w czasie $O(nk^4 + 2^{O(\sqrt{k}\log k)}) = O(nk^4 + k^{O(\sqrt{k})})$, który to czas jest osiągany nawet dla alfabetu wielkości liniowej od n. Zainteresowanych Czytelników odsyłamy do pracy *Covering Problems for Partial Words and for Indeterminate Strings* autorstwa M. Crochemore'a, C.S. Iliopoulosa i twórców tego zadania, dostępnej pod adresem `http://arxiv.org/pdf/1412.3696.pdf`.

I

Inscenizacja

Autor zadania: Adam Karczmarz
Opis rozwiązania: Adam Karczmarz
Dostępna pamięć: 256 MB
`https://oi.edu.pl/pl/archive/amppz/2014/ins`

Stefan Beitberg jest reżyserem kina akcji. W ostatnim czasie pracuje nad nowym filmem, którego tematem są porachunki bajtockich mafii. Beitberg zastanawia się, jaki powinien być przebieg kulminacyjnej sceny filmu, w której odbędzie się widowiskowa strzelanina.

W scenie uczestniczy n gangsterów, ponumerowanych dla uproszczenia kolejnymi liczbami od 1 do n. Gdy napięcie sięga zenitu, każdy z gangsterów wyciąga swoją broń i wymierza ją w kierunku pewnego innego gangstera. Żadnych dwóch uczestników sceny nie mierzy do tego samego gangstera. Gangsterzy są biedni, lecz dobrze wyszkoleni — każdy z nich może oddać co najwyżej jeden, ale zawsze celny i śmiertelny, strzał.

W pewnym momencie któryś z bandziorów nie wytrzymuje napięcia i rozpoczyna się strzelanina.

Reżyser ustalił pewną początkową kolejność, w jakiej gangsterzy mają pociągać za spust. Mianowicie, gangster i strzela w kierunku gangstera p_i dokładnie w momencie t_i, chyba że do tego czasu gangster i został już zabity. Przyjmujemy, że gangster ginie dokładnie w chwili, gdy ktoś oddaje strzał w jego kierunku.

Reżyser chciałby wiedzieć, ilu gangsterów zostanie przy życiu pod koniec sceny. Beitberg nie jest jeszcze całkowicie pewien co do kolejności, w jakiej gangsterzy mają strzelać. Co pewien czas wydaje polecenie, aby zmienić jedną z wartości t_i. Po każdej takiej zmianie chciałby znać liczbę gangsterów, którzy przeżyją, dla nowej kolejności oddawania strzałów (przy uwzględnieniu wszystkich dotychczas wykonanych zmian).

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n ($2 \leqslant n \leqslant 200\,000$), oznaczająca liczbę gangsterów biorących udział w scenie. W drugim wierszu znajduje się n liczb całkowitych $p_1, p_2, \ldots, p_n$ ($1 \leqslant p_i \leqslant n$, $p_i \neq i$, $p_i \neq p_j$ dla $i \neq j$), określających, do kogo zamierzają strzelać kolejni gangsterzy.

W trzecim wierszu znajduje się n liczb całkowitych $u_1, u_2, \ldots, u_n$ ($1 \leqslant u_i \leqslant 10^9$), opisujących początkową kolejność oddawania strzałów przez gangsterów: początkowa wartość t_i jest równa u_i.

W czwartym wierszu znajduje się jedna liczba całkowita q ($0 \leqslant q \leqslant 200\,000$), oznaczająca liczbę zmian wartości $t_1, \ldots, t_n$ planowanych przez Beitberga. Kolejne q wierszy to opis tych zmian. W i-tym z nich znajdują się dwie liczby całkowite k_i i v_i ($1 \leqslant k_i \leqslant n$, $1 \leqslant v_i \leqslant 10^9$), oznaczające, że i-ta zmiana polega na ustawieniu wartości t_{k_i} na v_i. Liczby $u_1, u_2, \ldots, u_n, v_1, v_2, \ldots, v_q$ są parami różne.

Wyjście

Twój program powinien wypisać na wyjście dokładnie $q + 1$ wierszy. W pierwszym wierszu powinna znaleźć się liczba gangsterów, którzy przeżyją strzelaninę, zakładając początkową kolejność strzelania. W i-tym z q kolejnych wierszy należy wypisać liczbę gangsterów pozostałych przy życiu, przy założeniu, że kolejność strzelania określa ciąg $t_1, \ldots, t_n$ po dokonaniu wszystkich zmian od pierwszej do i-tej.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2
2
1
1
```

Rozwiązanie

Opis rozwiązania rozpoczniemy od analizy początkowego ustawienia gangsterów. Skoro żadnych dwóch gangsterów nie mierzy do tego samego gangstera, a każdy gangster mierzy do dokładnie jednego przeciwnika, wnioskujemy, że do każdego gangstera mierzy dokładnie jeden inny gangster. Bohaterowie zadania są zatem ustawieni w pewną liczbę kółek, tzn. dla dowolnego $i \in \{1, \ldots, n\}$ istnieje $m \geqslant 2$ różnych gangsterów $g_1, \ldots, g_m$, takich że:

- $g_1 = i$,
- dla każdego $j \in \{1, \ldots, m-1\}$, gangster g_j mierzy do gangstera g_{j+1},
- gangster g_m mierzy do gangstera g_1.

Ponieważ strzały nie padają pomiędzy dwoma różnymi kółkami, każda ze zmian kolejności strzelania może wpłynąć na wynik w tylko jednym z kółek. Możemy zatem rozpatrywać każde z kółek osobno — szukana liczba ocalałych to łączna liczba ocalałych we wszystkich kółkach. Podział na kółka można wykonać w czasie $O(n)$.

Jedno kółko, ustalona kolejność strzelania

Zajmijmy się więc sytuacją, gdy $m \geqslant 2$ gangsterów ustawionych jest w jednym kółku. Dla uproszczenia przenumerujmy ich tak, aby gangster 1 mierzył do gangstera 2, gangster 2 — do gangstera 3 itd., a gangster m do gangstera 1. Dla wygody będziemy czasami pisać gangster 0, mając na myśli gangstera m. Niech t_i oznacza moment, w którym strzela gangster i.

Zamiast zliczać ocalałych, łatwiej będzie nam liczyć oddane strzały. Każdy oddany strzał dosięga swojego celu, dlatego jeśli oddanych zostanie s strzałów, przeżyje dokładnie $m - s$ gangsterów. Gangstera i nazwiemy *pewniakiem*, jeśli $t_i < t_{i-1}$, tzn. gangster i strzela wcześniej niż gangster $i - 1$. Zauważmy, że jeśli gangster i jest pewniakiem, to odda strzał niezależnie od tego, czy ostatecznie zginie.

Ponieważ liczby t_i są parami różne, istnieje co najmniej jeden pewniak (gangster strzelający najwcześniej). Rozważmy zatem pewnego pewniaka i. Zaczynamy od gangstera i i przesuwamy się po kółku w kierunku oddawanych strzałów, aż dotrzemy do kolejnego pewniaka. Oznaczmy jego numer przez j. Jeśli jest tylko jeden pewniak, zachodzi $j = i$. Niech $i = w_1, \ldots, w_p$ będą kolejnymi gangsterami, których napotykamy, poruszając się po kółku od i do j, wyłączywszy j. Gangsterzy $w_2, \ldots, w_p$ nie są pewniakami, więc $t_{w_1} < t_{w_2} < \ldots < t_{w_p}$. Zauważmy, że gangster w_2 zginie, zanim pociągnie za spust, i w konsekwencji gangster w_3 przeżyje i odda strzał. W efekcie gangster w_4 także zginie, zanim wystrzeli, a to pozwoli gangsterowi w_5 oddać strzał i tak dalej. Kontynuując to rozumowanie, dojdziemy do wniosku, że oddadzą strzały dokładnie ci gangsterzy w_l, których indeks l jest nieparzysty — będzie ich dokładnie $d_i = \lfloor \frac{p+1}{2} \rfloor$. Ponieważ podzieliliśmy kółko na rozłączne fragmenty zaczynające się od każdego z pewniaków, wystarczy dodać wartości d_i obliczone dla wszystkich pewniaków, aby otrzymać sumaryczną liczbę oddanych strzałów.

Algorytm i implementacja

Z powyższej analizy wynika już algorytm dla ogólnej wersji zadania, w której musimy obsługiwać zmiany wartości t_i. Będziemy utrzymywać zbiór pewniaków w taki sposób, aby można było efektywnie obliczać kolejnego i poprzedniego pewniaka w kółku, a także szybko aktualizować zbiór pewniaków. Po zmianie wartości t_i jedynie gangsterzy i oraz $i + 1$ mogą zyskać lub stracić status pewniaka. Dodatkowo może się zmienić wartość d_s dla pewniaka s, który poprzedza i. Jest jasne, że potrzeba zaktualizować tylko stałą liczbę pewniaków i wartości d_j dla istniejących już pewniaków.

Zbiór pewniaków najwygodniej utrzymywać w zrównoważonym drzewie binarnym. Można użyć na przykład struktury `set` z biblioteki STL. Wtedy dodanie i usunięcie elementu, jak również dostęp do poprzedniego bądź następnego elementu zbioru zajmują czas $O(\log n)$. W ten sposób otrzymujemy rozwiązanie działające w czasie $O(n + q \log n)$, gdzie q to liczba zmian wartości t_i.

J

Jaskinia

Autor zadania: Tomasz Idziaszek
Opis rozwiązania: Jakub Łącki
Dostępna pamięć: 256 MB
`https://oi.edu.pl/pl/archive/amppz/2014/jas`

Grupa speleologów planuje zbadać odkrytą niedawno jaskinię. Jaskinia składa się z n komnat ponumerowanych od 1 do n. Komnaty są połączone za pomocą $n-1$ korytarzy w taki sposób, że z dowolnej komnaty można przejść do dowolnej innej. Każdy korytarz łączy dokładnie dwie komnaty.

Badanie jaskini przeprowadzi grupa m speleologów, których dla uproszczenia ponumerujemy od 1 do m. Każdy speleolog przedstawił wymagania dotyczące obszaru jaskini, który będzie badać. Speleolog i chciałby rozpocząć badanie w komnacie a_i, zakończyć je w komnacie b_i, a po drodze przemierzyć co najwyżej d_i korytarzy (każde przebycie tego samego korytarza liczymy osobno). Bajtazar, kierownik wyprawy, chciałby, by w pewnym momencie wszyscy badacze mogli spotkać się i wymienić swoimi spostrzeżeniami. Z tego powodu zastanawia się, czy może wybrać jedną z komnat jaskini i tak wytyczyć trasy speleologów, by wszystkie prowadziły przez wybraną komnatę. Oczywiście wytyczone trasy muszą spełniać wymagania postawione przez badaczy.

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita t ($1 \leqslant t \leqslant 1000$) określająca liczbę zestawów testowych. Dalej następują opisy poszczególnych zestawów. Opis jednego zestawu rozpoczyna się od wiersza z dwiema liczbami całkowitymi n i m ($2 \leqslant n, m \leqslant 300\,000$), które opisują, odpowiednio, liczbę komnat w jaskini oraz liczbę speleologów. W kolejnych $n-1$ wierszach opisane są korytarze jaskini. Każdy z nich zawiera dwie liczby całkowite u_i, w_i ($1 \leqslant u_i, w_i \leqslant n$), które oznaczają, że komnaty u_i oraz w_i są bezpośrednio połączone korytarzem.

Następne m wierszy opisuje wymagania speleologów. W i-tym z tych wierszy znajdują się trzy liczby całkowite a_i, b_i, d_i ($1 \leqslant a_i, b_i \leqslant n$, $1 \leqslant d_i \leqslant 600\,000$). Oznaczają one, że speleolog i rozpocznie badanie w komnacie a_i, zakończy je w komnacie b_i, a po drodze co najwyżej d_i razy przejdzie korytarzem. Możesz założyć, że da się przejść z komnaty a_i do komnaty b_i, przemierzając nie więcej niż d_i korytarzy. Zarówno suma wartości n po wszystkich zestawach testowych, jak i suma wartości m nie przekraczają 300 000.

Wyjście

Twój program powinien wypisać na wyjście dokładnie t wierszy. W i-tym wierszu powinna znaleźć się odpowiedź dla i-tego zestawu testowego z wejścia. Jeśli da się tak poprowadzić trasy speleologów, by wszystkie przebiegały przez jedną komnatę, należy wypisać `TAK`, a następnie numer komnaty, w której może dojść do spotkania. W przeciwnym razie należy wypisać jedynie słowo `NIE`. Jeśli istnieje wiele poprawnych odpowiedzi, Twój program powinien wypisać dowolną z nich.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
TAK 2
NIE
```

Rozwiązanie

Na początek wprowadźmy kilka oznaczeń. Przez $d(u, w)$ oznaczymy odległość między komnatami u i w, czyli liczbę korytarzy na najkrótszej trasie pomiędzy nimi. Każdy speleolog badający jaskinię opisany jest przez trójkę liczb (a_i, b_i, d_i). Oznacza ona, że speleolog ten porusza się z komnaty a_i do b_i i po drodze przechodzi co najwyżej d_i korytarzy. Zatem może on odwiedzić komnatę x wtedy i tylko wtedy, gdy

$$d(a_i, x) + d(x, b_i) \leqslant d_i. \tag{1}$$

Zbiór komnat, które może odwiedzić speleolog i, będziemy nazywać *obszarem* numer i.

Przejdźmy teraz na język teorii grafów. W zadaniu podane jest drzewo o n wierzchołkach (plan jaskini) oraz opis m obszarów. Naszym celem jest sprawdzenie, czy istnieje wierzchołek należący do wszystkich obszarów i, w przypadku odpowiedzi twierdzącej, podanie przykładu takiego wierzchołka.

Łatwo możemy opisać rozwiązanie działające w czasie $O(n(n + m))$. Na początku w czasie $O(n^2)$ obliczamy odległości między wszystkimi parami wierzchołków w drzewie. To pozwala nam dla każdego z m obszarów wygenerować w czasie $O(n)$ wszystkie wierzchołki, które wchodzą w jego skład — po prostu sprawdzamy kolejno dla każdego wierzchołka, czy należy on do tego obszaru, korzystając ze wzoru (1). Na koniec obliczamy przecięcie wszystkich obszarów (w tym celu wystarczy dla każdego wierzchołka zliczać, do ilu obszarów on należy). Rozwiązanie to jest stosunkowo proste i, rzecz jasna, niewystarczająco szybkie.

Dziel i zwyciężaj

Pierwsze z lepszych rozwiązań oprzemy na technice dziel i zwyciężaj. Standardowo stosujemy ją do tablic i w typowym przypadku na początku wykonujemy pracę proporcjonalną do rozmiaru tablicy, a następnie dzielimy tablicę na dwie połówki i dalej w każdej z tych połówek uruchamiamy algorytm rekurencyjnie. Jednak jak podzielić drzewo na dwa, przynajmniej w miarę równe, kawałki? W tym celu stosujemy następującą własność.

Lemat 1. Niech T będzie drzewem o n wierzchołkach. Wówczas w czasie $O(n)$ możemy znaleźć w tym drzewie wierzchołek v, którego usunięcie spowoduje rozpad drzewa T na poddrzewa o co najwyżej $\frac{n}{2}$ wierzchołkach. Taki wierzchołek v nazywamy *centroidem*.

Dowód: Aby znaleźć centroid, ukorzeniamy drzewo T w dowolnym wierzchołku, a następnie dla każdego wierzchołka w obliczamy rozmiar poddrzewa zaczepionego w w. Ten krok wykonać można oczywiście w czasie $O(n)$.

Teraz przechodzimy po drzewie T w następujący sposób. Rozpoczynamy w korzeniu i dopóki jedno z poddrzew zaczepionych w dzieciach aktualnie rozważanego wierzchołka ma rozmiar większy niż $\frac{n}{2}$, przechodzimy do tego właśnie wierzchołka. Ponieważ poruszamy się jedynie w dół drzewa, w pewnym momencie nasza wędrówka zakończy się w pewnym wierzchołku v. Jak się zaraz okaże, wierzchołek v jest centroidem drzewa T.

Poddrzewo zaczepione w wierzchołku v ma z całą pewnością rozmiar większy niż $\frac{n}{2}$, bo zaczęliśmy w korzeniu i zawsze przechodziliśmy do poddrzewa o takiej własności. Jeśli wierzchołek v nie jest korzeniem drzewa T, po usunięciu v powstanie drzewo zawierające ojca v. Jednak to poddrzewo ma rozmiar mniejszy niż $\frac{n}{2}$, bo poddrzewo v jest większe niż $\frac{n}{2}$. Z drugiej strony poddrzewa zaczepione w dzieciach wierzchołka v mają rozmiar nie większy niż $\frac{n}{2}$, bo inaczej przejścia drzewa nie skończylibyśmy w v. Zatem v jest centroidem drzewa. □

Ogólna idea rozwiązania dziel i zwyciężaj wygląda następująco. Na początku znajdujemy w drzewie T centroid v, po czym sprawdzamy, czy v należy do przecięcia wszystkich obszarów. Jeśli tak, to od razu dostajemy rozwiązanie. W przeciwnym razie, jeśli przecięcie wszystkich obszarów jest niepuste, to musi się znajdować w całości w jednym z drzew powstałych z drzewa T po usunięciu wierzchołka v. Znajdujemy to drzewo i w nim rekurencyjnie wykonujemy nasz algorytm.

Przejdźmy do bardziej szczegółowego opisu. Po wyznaczeniu centroidu v obliczamy odległości od v do wszystkich wierzchołków w drzewie T. To możemy zrobić w czasie $O(n)$. Obliczone odległości wystarczają, by, korzystając ze wzoru (1), w czasie $O(m)$ sprawdzić, do których obszarów należy wierzchołek v. Jeśli v należy do wszystkich obszarów, to od razu przerywamy dalsze poszukiwania i zgłaszamy znaleziony wierzchołek.

Oznaczmy przez T_v zbiór drzew powstałych z drzewa T przez usunięcie centroidu v. Aby wykonać krok rekurencyjny, chcemy znaleźć w zbiorze T_v drzewo, które zawiera wszystkie wierzchołki z przecięcia wszystkich obszarów. Istnieje co najmniej jeden obszar i opisany przez trójkę (a_i, b_i, d_i), do którego nie należy centroid v (bo przeciwny przypadek przed chwilą wykluczyliśmy). Wierzchołki a_i oraz b_i wyznaczające ten obszar należą do tego samego drzewa $F \in T_v$. Jest tak, ponieważ wszystkie wierzchołki na ścieżce łączącej a_i oraz b_i należą do obszaru i, a zatem nie mogą leżeć w różnych drzewach ze zbioru T_v, bo wówczas łącząca je

ścieżka zawierałaby v. Skoro wierzchołki a_i oraz b_i leżą w drzewie F i wyznaczają obszar niezawierający v, to cały obszar i zawarty jest w F. Stąd natychmiast wnioskujemy, że jeśli przecięcie obszarów jest niepuste, to musi ono leżeć w drzewie F. Możemy więc wykonać algorytm rekurencyjnie dla drzewa F.

Jeden poziom rekurencji możemy zaimplementować tak, by działał w czasie $O(n+m)$, gdzie n to rozmiar aktualnie rozważanego drzewa. Z definicji centroidu, po wykonaniu każdego kroku rozmiar rozważanego drzewa maleje co najmniej dwukrotnie. Dzięki temu rekurencja ma $O(\log n)$ poziomów, a czas działania algorytmu możemy oszacować z góry przez $O((n+m)\log n)$. Jesteśmy jednak w stanie podać nieco lepsze oszacowanie. Na każdym poziomie wykonujemy pracę proporcjonalną do rozmiaru aktualnie rozważanego fragmentu drzewa, przy czym rozmiar ten maleje co najmniej dwukrotnie z każdym poziomem. Ponieważ $n + \frac{n}{2} + \frac{n}{4} + \ldots \leqslant 2n$, więc *łączny* czas przetwarzania drzew możemy ograniczyć przez $O(n)$. Nasz algorytm działa zatem w czasie $O(n + m \log n)$.

Zauważmy, że w naszym przypadku po znalezieniu centroidu nie możemy po prostu wykonać algorytmu rekurencyjnie we wszystkich drzewach należących do zbioru T_v. Wprawdzie wejściowe drzewo dzielimy przy takim wywołaniu rekurencyjnym na rozłączne kawałki, jednak każdy obszar może należeć do wielu drzew ze zbioru T_v. Takie obszary musielibyśmy rozważać w wielu wywołaniach rekurencyjnych na tym samym poziomie rekurencji, co znacznie zwiększyłoby czas działania.

Szybsze rozwiązanie

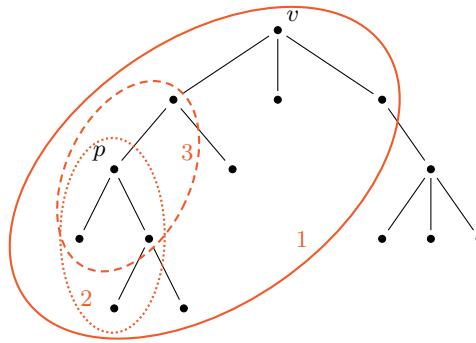
Zadanie możemy też rozwiązać szybszym i prostszym algorytmem. Nieco więcej zastanowienia wymaga jedynie udowodnienie jego poprawności.

Ustalmy dowolny wierzchołek v. *Odległością* obszaru i opisanego przez trójkę (a_i, b_i, d_i) od wierzchołka v nazwiemy odległość pomiędzy wierzchołkiem v a takim wierzchołkiem z obszaru i, który leży najbliżej v. Zauważmy, że odległość obszaru i od wierzchołka v wyraża się następującym wzorem (którego uzasadnienie pozostawiamy jako proste ćwiczenie):

$$\max\left(0, \left\lceil \frac{d(a_i, v) + d(v, b_i) - d_i}{2} \right\rceil\right). \tag{2}$$

Ideę rozwiązania przedstawić można w zaledwie paru zdaniach. Znajdujemy obszar i, który leży najdalej od wierzchołka v, a następnie wierzchołek p z obszaru i, który leży najbliżej v (patrz rysunek 1). Przy tych oznaczeniach prawdziwy jest następujący fakt: jeśli przecięcie wszystkich obszarów jest niepuste, to należy do niego wierzchołek p. Co ciekawe, fakt ten jest prawdziwy niezależnie od wyboru początkowego wierzchołka v (choć oczywiście możemy w efekcie otrzymać różne wierzchołki p).

Aby udowodnić ten kluczowy fakt, ukorzeńmy całe drzewo w wierzchołku startowym v. Jasne jest, że przecięcie wszystkich obszarów musi znajdować się w poddrzewie zaczepionym w p, bo tylko w tym poddrzewie znajdują się wierzchołki obszaru i (ze wszystkich wierzchołków tego obszaru wierzchołek p leży najbliżej korzenia). Z drugiej strony, jeśli pewien obszar j zawiera wierzchołek z poddrzewa zaczepionego w p, to musi zawierać również wierzchołek p (bo inaczej obszar j leżałby dalej od wierzchołka v niż obszar i). Zatem p należy do przecięcia wszystkich obszarów, jeśli tylko jest ono niepuste.



Rysunek 1. Drzewo ukorzenione w wierzchołku v oraz trzy obszary. Najdalej od v leży obszar 2. Wierzchołek p leży najbliżej v spośród wierzchołków należących do obszaru 2.

Rozwiązanie to możemy prosto zrealizować w czasie $O(n+m)$. Aby znaleźć obszar i, który leży najdalej od wierzchołka v, obliczamy odległości od v do wszystkich wierzchołków drzewa i posługujemy się wzorem (2), aby w czasie stałym obliczyć odległość od v do każdego z m obszarów. Po wyznaczeniu obszaru i, obliczamy odległości od definiujących go wierzchołków a_i oraz b_i do wszystkich innych. Dzięki temu dla każdego wierzchołka możemy w czasie stałym stwierdzić, czy należy on do obszaru i. Teraz przeglądamy wszystkie wierzchołki i dla każdego należącego do obszaru i sprawdzamy jego odległość od v. W ten sposób w czasie $O(n)$ znajdujemy wierzchołek p. W ostatnim kroku obliczamy odległości od p do wszystkich wierzchołków i sprawdzamy, czy p należy do wszystkich obszarów.

Kapitan

Autor zadania: Jakub Łącki
Opis rozwiązania: Jakub Łącki
Dostępna pamięć: 256 MB
`https://oi.edu.pl/pl/archive/amppz/2014/kap`

Kapitan Bajtazar przemierza wody Morza Bajtockiego wraz ze swoim niezastąpionym pierwszym oficerem Bajtkiem. Na morzu znajduje się n wysp, które numerujemy liczbami od 1 do n. Przy wyspie numer 1 przycumował statek kapitana. W ramach wyprawy kapitan planuje popłynąć na wyspę numer n.

W trakcie rejsu statek zawsze porusza się w jednym z czterech kierunków świata: na północ, południe, wschód lub zachód. W każdym momencie przy sterach stoi albo kapitan, albo pierwszy oficer. Za każdym razem, gdy statek wykona skręt o 90°, zmieniają się oni przy sterach.

Po drodze statek może zatrzymywać się przy innych wyspach. Po każdym postoju kapitan może zdecydować, czy obejmuje stery jako pierwszy. Innymi słowy, na każdym fragmencie trasy prowadzącym z wyspy do wyspy jeden z marynarzy obejmuje stery, gdy statek płynie na północ lub południe, a drugi z nich steruje podczas rejsu na wschód lub zachód. W szczególności, jeśli pewien fragment trasy prowadzi dokładnie w jednym z czterech kierunków świata, na tym fragmencie steruje tylko jeden z marynarzy.

Kapitan zastanawia się teraz, jak zaplanować trasę najbliższego rejsu i podział pracy, by spędzić jak najmniej czasu przy sterze. Jednocześnie kapitan nie przejmuje się, jak długa będzie wyznaczona trasa. Przyjmujemy, że statek płynie ze stałą prędkością jednej jednostki na godzinę.

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($2 \leqslant n \leqslant 200\,000$) oznaczającą liczbę wysp na morzu. Dla uproszczenia na Morze Bajtockie nanosimy układ współrzędnych, którego osie są równoległe do kierunków świata. Każdą z wysp reprezentujemy jako pojedynczy punkt. Kolejne n wierszy zawiera opisy wysp: i-ty z nich zawiera dwie liczby całkowite x_i, y_i ($0 \leqslant x_i, y_i \leqslant 1\,000\,000\,000$) oznaczające współrzędne i-tej wyspy na morzu. Każda wyspa ma inne współrzędne.

Wyjście

Twój program powinien wypisać na wyjście jedną liczbę całkowitą, oznaczającą najmniejszą liczbę godzin, przez które kapitan będzie musiał sterować statkiem na trasie z wyspy numer 1 do wyspy numer n.

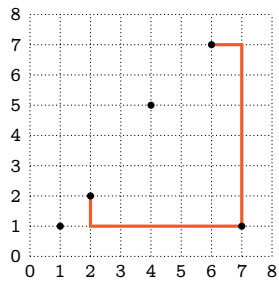
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2
```



Wyjaśnienie przykładu: Kapitan może wyznaczyć trasę, którą zaznaczono na obrazku. W trakcie rejsu z wyspy 1 (współrzędne $(2, 2)$) na wyspę 4 (współrzędne $(7, 1)$) kapitan steruje tylko przez godzinę, gdy statek płynie na południe. W trakcie drugiego fragmentu podróży kapitan steruje jedynie wtedy, gdy statek porusza się na wschód.

Rozwiązanie

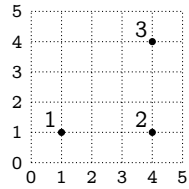
W tym zadaniu przemieszczamy się po płaszczyźnie. Zostało na niej zaznaczonych n punktów $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, a naszym celem jest dotarcie z punktu (x_1, y_1) do punktu (x_n, y_n). W jednym kroku możemy przejść między dowolnymi dwoma punktami, przy czym przejście z punktu (x', y') do punktu (x'', y'') kosztuje nas

$$\min(|x'' - x'|, |y'' - y'|).$$

Chcemy znaleźć trasę z (x_1, y_1) do (x_n, y_n) o jak najmniejszym łącznym koszcie.

Szukamy zatem najkrótszej trasy między dwoma punktami, przy czym „odległość" między punktami mierzymy w dosyć nietypowy sposób. No właśnie, użyliśmy cudzysłowu, gdyż nasz sposób pomiaru kosztu ma mało wspólnego z tym, jak zazwyczaj rozumiemy pojęcie odległości. Mówiąc bardziej formalnie, nie definiuje on metryki na zaznaczonych punktach. Dzieje się tak, ponieważ nie jest spełniony warunek trójkąta.

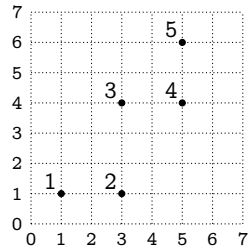
Wyjaśnimy to na przykładzie. Spójrzmy na rysunek 1. Koszt bezpośredniego przejścia z punktu 1 do punktu 3 wynosi $\min(|4 - 1|, |4 - 1|) = 3$. Jednocześnie, jeśli pójdziemy z 1 do 2, a potem do 3, to łączny koszt wyniesie 0. To właśnie oznacza brak nierówności trójkąta: pójście przez punkt pośredni może być tańsze niż wykonanie bezpośredniego kroku do celu.



Rysunek 1. Przejście bezpośrednio z punktu 1 do punktu 3 kosztuje więcej niż przejście najpierw do punktu 2, a dopiero potem do punktu 3.

Skoro koszt mierzymy w tak nietypowy sposób, musimy odłożyć na bok pewne intuicje dotyczące szukania krótkich tras na płaszczyźnie. Mimo to, ostateczne rozwiązanie zadania wcale nie będzie trudne.

Zacznijmy od bardzo prostego pomysłu: może zawsze wystarczy przejść przez co najwyżej jeden punkt pośredni? Niestety nie jest to prawda, o czym przekonać się możemy, patrząc na rysunek 2.



Rysunek 2. Istnieje trasa o koszcie 0 z punktu 1 do punktu 5, jednak wymaga ona przejścia przez trzy punkty pośrednie.

Między dwoma punktami, które leżą na tej samej pionowej lub poziomej prostej, możemy przemieszczać się za darmo. Ogólnie, mało kosztuje nas chodzenie między punktami, których pierwsze lub drugie współrzędne są do siebie zbliżone. Spróbujemy sformalizować tę własność.

Załóżmy, że przemieszczamy się z punktu (x_a, y_a) do punktu (x_c, y_c), gdzie $x_a \leqslant x_c$, i kosztuje nas to $|x_c - x_a|$ (czyli $|x_c - x_a| \leqslant |y_c - y_a|$). Ponadto istnieje punkt (x_b, y_b), taki że $x_a \leqslant x_b \leqslant x_c$. Wówczas możemy z (x_a, y_a) iść do (x_c, y_c) przez (x_b, y_b) i nie zwiększy to kosztu. Dlaczego? Przejście z (x_a, y_a) do (x_b, y_b) będzie nas kosztować nie więcej niż $|x_b - x_a| = x_b - x_a$. Z kolei drugi krok kosztować będzie co najwyżej $|x_c - x_b| = x_c - x_b$. A zatem łączny koszt możemy ograniczyć przez $(x_b - x_a) + (x_c - x_b) = x_c - x_a$.

Z tych rozważań płynie następujący wniosek: jeśli przemieszczamy się z (x_a, y_a) do (x_c, y_c) i kosztuje nas to $|x_c - x_a|$, to możemy po drodze odwiedzić kolejno wszystkie punkty, których pierwsza współrzędna leży między x_a a x_c (o ile tylko idziemy w kolejności niemalejących pierwszych współrzędnych). Taka trasa będzie na pewno nie droższa niż wykonanie bezpośredniego kroku. Podobne rozumowanie możemy zastosować również do drugich współrzędnych.

Przyjmijmy na chwilę, że żadne dwa punkty nie leżą na tej samej pionowej lub poziomej prostej, i rozważmy trasę o najmniejszym koszcie między pewnymi dwoma punktami. W każdym kroku przechodzimy z pewnego punktu (x_i, y_i) do punktu (x_j, y_j) i kosztuje nas to $|x_j - x_i|$ lub $|y_j - y_i|$. Każdy taki krok możemy zastąpić albo przejściem przez wszystkie punkty o pierwszych współrzędnych między x_i a x_j, albo przejściem przez wszystkie punkty o drugich współrzędnych między y_i a y_j. Płynie stąd wniosek kluczowy do efektywnego rozwiązania zadania. Wprowadźmy tablice $t_x[1..n]$ oraz $t_y[1..n]$, które zawierają wszystkie punkty. W tablicy t_x są one posortowane niemalejąco względem pierwszej współrzędnej, zaś w tablicy t_y — względem drugiej. Nasze obserwacje oznaczają, że między dowolnymi dwoma punktami istnieje optymalna trasa, w której w każdym kroku idziemy z pewnego punktu p do jednego z maksymalnie czterech punktów: albo do punktu, który w tablicy t_x leży obok p (tj. tuż przed lub tuż za), albo do punktu, który w tablicy t_y leży obok p.

Aby rozwiązać nasze zadanie efektywnie, konstruujemy graf G, którego wierzchołkami są zaznaczone na płaszczyźnie punkty. Budujemy tablice t_x i t_y i łączymy krawędziami wierzchołki (punkty), które leżą w sąsiednich komórkach tablic. W efekcie otrzymujemy graf nieskierowany z n wierzchołkami i $2n-2$ krawędziami. Każdej krawędzi przypisujemy wagę równą kosztowi przejścia pomiędzy punktami, którym odpowiadają wierzchołki na jej końcach. Zauważmy teraz, że ścieżka o minimalnej wadze między dwoma wierzchołkami w G odpowiada najtańszej trasie między punktami na płaszczyźnie.

Co w przypadku gdy dwa punkty leżą na jednej prostej poziomej lub pionowej? Nasz algorytm nadal będzie działać, gdyż przy konstrukcji grafu zostaną one połączone ścieżką składającą się z krawędzi o wadze 0.

Odległości z wierzchołka numer 1 w grafie G możemy znaleźć za pomocą algorytmu Dijkstry. Zarówno konstrukcję grafu, jak i sam algorytm Dijkstry można zaimplementować tak, by działały w czasie $O(n \log n)$, a to już wystarcza do rozwiązania zadania.

Literatura

L. Banachowski, K. Diks, W. Rytter, *Algorytmy i struktury danych*. WN PWN, Warszawa, 2019

T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Wprowadzenie do algorytmów*. WN PWN, Warszawa, 2012

R.L. Graham, D.E. Knuth, O. Patashnik, *Matematyka konkretna*. WN PWN, Warszawa, 2011

P. Stańczyk, *Algorytmika praktyczna. Nie tylko dla mistrzów*. WN PWN, Warszawa, 2009

W poszukiwaniu wyzwań. Wybór zadań z konkursów programistycznych Uniwersytetu Warszawskiego. K. Diks, T. Idziaszek, J. Łącki, J. Radoszewski (red.). WN PWN, Warszawa, 2018

Przygody Bajtazara. 25 lat Olimpiady Informatycznej – wybór zadań. K. Diks, T. Idziaszek, J. Łącki, J. Radoszewski (red.). WN PWN, Warszawa, 2018

Netografia

`deltami.edu.pl` — strona internetowa miesięcznika *Delta*

`oi.edu.pl` — niebieskie książeczki Olimpiady Informatycznej edycji I–XXIII

`was.zaa.mimuw.edu.pl` — Wykłady z Algorytmiki Stosowanej

Opracowania zadań *Prostokąt arytmetyczny*, *Fotoradary* i *Filary* powstały na podstawie artykułów w miesięczniku *Delta* (odpowiednio z numerów: 3/2012, 1/2015 i 3/2015). Opracowania zadań *Drzewo i mrówki* oraz *Herbata i mleko* zostały zaczerpnięte z książki *W poszukiwaniu wyzwań*.

Spis treści

2011

2012

2013

2014