J. Tomasiewicz
A. Jaskółka Ł. Jocz R. Kozikowski

I Liceum Ogólnokształcące im. Adama Mickiewicza
w Białymstoku
Podlaskie Stowarzyszenie na Rzecz Uzdolnionych

Obóz Naukowy PROSERWY 2010
Część algorytmiczna

Kadra informatyczna:
Ireneusz Bujnowski
Adrian Jaskółka
Joachim Jelisiejew
Łukasz Jocz
Robert Kozikowski
Jacek Tomasiewicz (koordynator)

Główni organizatorzy:
Iwona Bujnowska
Ireneusz Bujnowski
Joachim Jelisiejew
Jacek Tomasiewicz

Serwy, 19 – 25 września 2010
(wydanie drugie)

ISBN 978-83-930856-3-7

Niniejsze wydanie książki ukazuje się dzięki:
inicjatywie i działaniom Olimpiady Informatycznej,
środkom finansowym firmy Google,
wsparciu Ośrodka Edukacji Informatycznej i Zastosowań Komputerów.

Spis treści

1 Wstęp **7**

I Grupa początkująca **9**

2 Zadania **11**
2.1 Dzień próbny . . . 11
Dodawanie . . . 11
2.2 Dzień pierwszy . . . 12
Koszykarz . . . 12
Odchudzanie . . . 13
Wężyk . . . 14
Znak działania . . . 15
2.3 Dzień drugi . . . 16
Monety . . . 16
Permutacje . . . 17
Taśma . . . 18
Pole . . . 19
2.4 Dzień trzeci . . . 20
Wykreślanka . . . 20
Piłeczka . . . 21
Bałwanek . . . 22
Manipulacja rankingu . . . 23
2.5 Dzień czwarty . . . 24
Pinezki . . . 24
Szachy . . . 25
Kratki . . . 26
Patyki . . . 27
2.6 Dzień piąty . . . 28
Grusze i jabłonie . . . 28
Farby . . . 29
Siłownia . . . 30
Dziewczynki . . . 31

3 Rozwiązania **32**
3.1 Dzień próbny . . . 32
3.2 Dzień pierwszy . . . 33
3.3 Dzień drugi . . . 37
3.4 Dzień trzeci . . . 42
3.5 Dzień czwarty . . . 46

3.6 Dzień piąty 50

4 Wykłady **54**
4.1 Złożoność czasowa 54
4.1.1 Wstęp 54
4.1.2 Czym jest złożoność 54
4.1.3 Przykłady różnych złożoności 55
4.1.4 Czy program ma wystraczającą złożoność 57
4.1.5 Podsumowanie 58
4.2 Praktyczne zastosowanie STL, część I 59
4.2.1 Wstęp 59
4.2.2 Czym jest STL 59
4.2.3 Wady STL 59
4.2.4 Para 60
4.2.5 Vector, podstawy 61
4.2.6 Kolejka 63
4.2.7 Kolejka priorytetowa 64
4.2.8 Podstawowe algorytmy i ich wykorzystanie 65

II Grupa średnio zaawansowana **67**

5 Zadania **69**
5.1 Dzień próbny 69
Kostki domina 69
5.2 Dzień pierwszy 70
Sznurki 70
Wieża 71
Tablica 72
5.3 Dzień drugi 73
Obwody 73
Wieża 2 74
Akwarium 75
5.4 Dzień trzeci 76
Tarasy 76
Rolki papieru 77
Karty 78
5.5 Dzień czwarty 79
Las 79
Suma 81
5.6 Dzień piąty 82
Wykres 82
Turniej 83
Przyjęcie 84

6 Rozwiązania **85**
6.1 Dzień próbny 85
6.2 Dzień pierwszy 87
6.3 Dzień drugi 90
6.4 Dzień trzeci 93
6.5 Dzień czwarty 97

6.6 Dzień piąty . . . 102

7 Wykłady **106**
7.1 Teoria liczb I . . . 106
7.1.1 Wstęp . . . 106
7.1.2 Sprawdzenie czy liczba jest pierwsza. . . . 106
7.1.3 Sito Eratostenesa . . . 107
7.1.4 Własności rozkładu liczby na czynniki pierwsze . . . 108
7.1.5 Algorytm Euklidesa . . . 109
7.1.6 Faktoryzacja $n!$ . . . 109
7.2 Algorytmy równoległe . . . 111
7.2.1 Wstęp . . . 111
7.2.2 Model obliczeń . . . 111
7.2.3 Typy maszyny PRAM . . . 111
7.2.4 Przykłady zadań . . . 112
7.2.5 Przeskakiwanie . . . 113
7.2.6 Jeszcze kilka przykładów . . . 114
7.2.7 Zadania . . . 115
7.3 Praktyczne zastosowanie STL, część II . . . 116
7.3.1 Wstęp do wskaźników . . . 116
7.3.2 Iterator . . . 117
7.3.3 Algorytmy wykorzystujące iteratory i wskaźniki . . . 118
7.3.4 Lista . . . 119
7.3.5 Zbiór i multizbiór . . . 120
7.3.6 Mapa . . . 121
7.3.7 Przeładowanie operatora . . . 121
7.3.8 Stabilne sortowanie . . . 123

III Grupa olimpijska **125**

8 Zadania **127**
8.1 Dzień próbny . . . 127
Pociąg . . . 127
8.2 Dzień pierwszy . . . 129
Zegary . . . 129
Cykl . . . 130
Śmieszny konkurs informatyczny . . . 131
8.3 Dzień drugi . . . 133
Najdłuższe rosnące podciągi . . . 133
ABC . . . 134
Ciężkie klocki . . . 135
8.4 Dzień trzeci . . . 136
Guziki . . . 136
Pokoje . . . 137
Przyjazne punkty . . . 138
8.5 Dzień czwarty . . . 139
Dziwna planeta . . . 139
Kupiec . . . 140
Materiały wybuchowe . . . 141

8.6 Dzień piąty 142
Najdłuższy, wspólny, rosnący 142
Bubu 143
Myszka 144

9 Rozwiązania **145**
9.1 Dzień próbny 145
9.2 Dzień pierwszy 147
9.3 Dzień drugi 150
9.4 Dzień trzeci 157
9.5 Dzień czwarty 162
9.6 Dzień piąty 169

10 Wykłady **173**
10.1 Teoria liczb II 173
10.1.1 Rozszerzony algorytm Euklidesa 173
10.1.2 Rozwiązywanie liniowego równania diofantycznego 174
10.1.3 Zliczanie punktów kratowych 175
10.1.4 Zadanie Trzęsienie Ziemi 177

Rozdział 1

Wstęp

Drogi Czytelniku!

Oddajemy Ci do rąk książkę która jest zbiorem zadań, rozwiązań i wykładów z tegorocznego obozu organizowanego przez I Liceum Ogólnokształcące w Białymstoku oraz Podlaskie Stowarzyszenie na Rzecz Uzdolnionych. Treści zadań z tej książki znajdują się również w serwisie http://main.edu.pl/, na tej stronie można też sprawdzić poprawność swojego rozwiązania w warunkach zbliżonych do obozowych. Na wstępie chcielibyśmy podziękować za nieocenioną pomoc następującym osobom:

- Pani mgr Iwonie Bujnowskiej i Panu mgr Ireneuszowi Bujnowskiemu za organizację obozu i wszechstronne wsparcie,
- Pani Joannie Bujnowskiej za autorstwo części zadań,
- Panu prof. dr hab. Krzysztofowi Diksowi za wsparcie merytoryczne,
- Panu Adamowi Iwaniukowi za opracowanie części zadań,
- Panu Joachimowi Jelisiejewowi za redakcję, korektę, obsługę skryptów w LaTeX-u, obsługę sprawdzarki obozowej oraz autorstwo części zadań,
- Panu Tadeuszowi Kuranowi za umożliwienie kolejnego wydania tej książki.
- Pani mgr Annie Opalińskiej, polonistce w I Liceum Ogólnokształcącym, za korektę błędów językowych i stylistycznych,
- Panu mgr Jakubowi Radoszewskiemu za udostępnienie serwisu http://main.edu.pl/ i pomocne rady.

Pierwszy obóz został zorganizowany we wrześniu 2008 z inicjatywy Ireneusza Bujnowskiego i Iwony Bujnowskiej. Od początku przybrał on formę tygodniowego wyjazdu, na którym uczestnicy rozwiązują zadania matematyczne i informatyczne wzorowane na zadaniach z licealnych olimpiad przedmiotowych. Tegoroczny obóz, organizowany w dniach 19 - 25 września w ośrodku wypoczynkowym PROserwy w Serwach koło Augustowa, zgromadził ponad stu uczestników.

Obóz ma profil typowo naukowy – rankami uczestnicy mierzą się z zadaniami, popołudniu, po krótkiej przerwie na obiad i zajęcia sportowe, przeprowadzane są omówienia zadań oraz liczne wykłady. Rozkład dnia jest dość napięty, czas wolny obozowiczów to jedynie późny wieczór. Zadania, warsztaty i wykłady mają bardzo zróżnicowany poziom – najłatwiejsze dostosowane

są do osób dopiero zaczynających naukę pozaszkolną, najtrudniejsze nastawione są na osoby uczestniczące w olimpiadach: obecnych oraz przyszłych finalistów i laureatów.

Tradycyjnie kadrę naukową obozu stanowią byli olimpijczycy – absolwenci I LO w Białymstoku, studenci Uniwersytetu Warszawskiego. Obóz nie jest zamknięty dla osób z zewnątrz – uczestnikami są również uczniowie wielu liceów z całej Polski.

We wrześniu 2011 planujemy zorganizować kolejny, czwarty już, obóz, na który serdecznie zapraszamy. Informacje na temat tego i innych obozów można znaleźć na stronie http://proserwy.ilo.pl.

Dowolne uwagi prosimy przesyłać na adres mailowy: jacek_tomas@o2.pl.

Autorzy.

Część I

Grupa początkująca

Rozdział 2

Zadania

2.1 Dzień próbny

Zadanie: **Dodawanie** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 0, 19.09.2010*

Dodawanie

Łukasz nie umie dodawać. Napisz program, który dodaje dwie liczby całkowite a i b.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczb całkowite a, b ($1 \leqslant a, b \leqslant 1000$).

Wyjście

W pierwszym i jedynym wierszu wyjścia powinna się znajdować jedna liczba całkowita, równa sumie dwóch liczb a i b.

Przykład

dla danych wejściowych:

```
1 2
```

poprawnym wynikiem jest:

```
3
```

2.2 Dzień pierwszy

Zadanie: **Koszykarz** Autor zadania: **Joanna Bujnowska**

Dostępna pamięć: **32 MB** *Dzień 1, 20.09.2010*

Koszykarz

Kozik pragnie zostać koszykarzem. Po rozmowie z trenerem okazało się, że jest za niski. Kozik jest jednak tak zdeterminowany, że chce spełnić wymagania trenera, nawet jeśli okazałoby się to oszustwem. Wpadł więc na genialny pomysł robienia sobie guzów na głowie, aż osiągnie wymagany wzrost. Zauważył, że przy każdym uderzeniu guz się powiększa o m cm. Kozik zastanawia się, ile minimalnie razy będzie musiał się uderzyć.

Wejście

W pierwszej linii wejścia znajdują się 3 liczby całkowite: k, w, m ($1 \leqslant k \leqslant 200, 1 \leqslant w, m \leqslant 10^9$), oznaczające odpowiednio wysokość Kozika, wymaganą przez trenera wysokość oraz wartość powiększania się guza po każdym uderzeniu.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą równą minimalnej liczbie uderzeń, które musi wykonać Kozik.

Przykład

dla danych wejściowych:

```
180 202 10
```

poprawnym wynikiem jest:

```
3
```

Zadanie: **Odchudzanie** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 1, 20.09.2010*

Odchudzanie

Kozik postanowił się odchudzić. Jest po n dniach diety i intensywnego treningu, jednak waga nie spadała mu równomiernie, a czasem nawet (ku zdziwieniu Kozika) zwiększała się.

Kozik codziennie zapisywał swoją wagę i teraz chce się pochwalić kolegom, więc wybierze taki fragment swojego dzienniczka, w którym schudł najbardziej. Znajdź ten fragment i policz, ile w nim schudł (czyli oblicz maksymalny spadek wagi Kozika)

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 10^6$). W drugim wierszu wejścia znajduje się n liczb całkowitych w_k oznaczających wagę Kozika w k - tym dniu diety ($1 \leqslant w_k \leqslant 10^9$).

Wyjście

W pierwszym i jedynym wierszu wyjścia powinna znaleźć się jedna liczba całkowita, oznaczająca maksymalny spadek wagi Kozika.

Przykład

dla danych wejściowych:

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

poprawnym wynikiem jest:

```
5
```

Zadanie: **Wężyk** Autor zadania: **Błażej Osiński**

Dostępna pamięć: **32 MB** *Dzień 1, 20.09.2010*

Wężyk

Ulubioną zabawą Stasia jest gra w "wężyka". Polega ona na tym, że najpierw wybiera sobie jedną liczbę całkowitą n, a następnie zaczyna na kartce zapisywać kolejne liczby naturalne. Chłopiec zapisuje łącznie n wierszy, w każdym z nich po n liczb.

W pierwszym wierszu Staś wypisuje liczby od 1 do n od lewej do prawej. Drugi wiersz wypełnia jednak od prawej do lewej liczbami od $n+1$ do $2n$. I tak dalej: wiersze o numerach nieparzystych wypełnia od lewej do prawej, a te o numerach parzystych – od prawej do lewej.

Twoim zadaniem jest napisanie programu, który będzie automatycznie grał w „wężyka".

Wejście

W pierwszym i jedynym wierszu wejścia znajduje się jedna liczba naturalna n ($1 \leqslant n \leqslant 100$).

Wyjście

Na wyjście należy wypisać n wierszy, w każdym z nich po n liczb naturalnych. Razem ma zostać wypisanych n^2 liczb: wszystkie liczby naturalne od 1 do n^2 . W każdym wierszu wypisane liczby powinny być pooddzielane pojedynczymi odstępami.

Przykład

dla danych wejściowych:

```
6
```

poprawnym wynikiem jest:

```
1 2 3 4 5 6
12 11 10 9 8 7
13 14 15 16 17 18
24 23 22 21 20 19
25 26 27 28 29 30
36 35 34 33 32 31
```

Zadanie: **Znak działania** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 1, 20.09.2010*

Znak działania

Napisz program, który wczyta dwie liczby całkowite a i b, a następnie wstawi znak działania '+', '-' albo '*' między nimi, w taki sposób, aby wynik działania był jak największy. Jeżeli można uzyskać największy wynik za pomocą więcej niż jednego działania, wypisujemy słowo 'NIE'.

Wejście

W pierwszym i jedynym wierszu wejścia znajdują się dwie liczby całkowite a i b ($-10^4 \leqslant a, b \leqslant 10^4$).

Wyjście

W pierwszym i jedynym wierszu wyjścia zapis działania zgodnie z przykładem lub słowo 'NIE' (wszystkie liczby ujemne należy zapisywać w nawiasie).

Przykład

dla danych wejściowych:

```
6 -5
```

poprawnym wynikiem jest:

```
6-(-5)=11
```

2.3 Dzień drugi

Zadanie: **Monety** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 2, 21.09.2010*

Monety

Bajtek posiada n monet. Każda z monet jest o nominale 1 Bajtalara i posiada dwie strony: awers i rewers. Bajtek ma rozłożone monety na stole i zastanawia się ile minimalnie monet musi przewrócić na druga stronę, aby wszystkie monety leżały na awersie lub wszystkie leżały na rewersie.

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 10^6$), oznaczająca liczbę monet Bajtka. Kolejny wiersz zawiera ciąg n liczb całkowitych $a_1, a_2, ..., a_n$, gdzie a_i oznacza opis i-tej monety, 0 – jeśli moneta leży na awersie, 1 – jeśli moneta leży na rewersie.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą równą minimalnej liczbie monet, jakie Bajtek powinien obrócić na drugą stronę.

Przykład

dla danych wejściowych:

```
6
1 0 1 1 1 0
```

poprawnym wynikiem jest:

```
2
```

Zadanie: **Permutacje** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 2, 21.09.2010*

Permutacje

Bajtek napisał na kartce n liczb. Zastanawia się teraz, czy są one permutacją liczb od 1 do n, czyli czy każda z liczb $1, 2, 3..., n-1, n$, występuje dokładnie jeden raz w tym ciągu.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 10^6$), oznaczającą ilość liczb jakie wypisał Bajtek. Kolejny wiersz zawiera ciąg n liczb całkowitych $a_1, a_2, ...a_n$ ($1 \leqslant a_i \leqslant 10^9$), gdzie a_i oznacza i-tą liczbę w ciągu Bajtka.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać słowo 'TAK', jeśli ciąg Bajtka jest permutacją liczb od 1 do n, lub słowo 'NIE', jeśli ciąg Bajtka nie jest permutacją liczb od 1 do n.

Przykład

dla danych wejściowych:

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

poprawnym wynikiem jest:

```
TAK
```

Zadanie: **Taśma** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 2, 21.09.2010*

Taśma

Jaś znalazł w domu długą taśmę. Bez chwili namysłu napisał na taśmie pewien ciąg liczb całkowitych. Teraz chciałby przeciąć taśmę w pewnym miejscu tak, aby różnica pomiędzy sumą liczb na jednym kawałku a sumą liczb na drugim kawałku była jak najbliższa zeru. Chcielibyśmy znać wartość bezwzględną z owej różnicy.

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($2 \leqslant n \leqslant 10^6$), oznaczającą ilość liczb wypisanych na taśmie. Drugi wiersz zawiera n liczb całkowitych a_i ($-10^3 \leqslant a_i \leqslant 10^3$), oznaczających kolejne liczby wypisane na taśmie.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą równą minimalnej wartości bezwzględnej różnicy pomiędzy dwoma kawałkami.

Przykład

dla danych wejściowych:

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

poprawnym wynikiem jest:

```
1
```

Zadanie: **Pole** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 2, 21.09.2010*

Pole

Napisz program, który mając dane współrzędne dwóch prostokątów, których boki są równoległe do osi współrzędnych, policzy pole części wspólnej tych prostokątów.

Wejście

W pierwszym wierszu wejścia znajdują się 4 liczby całkowite: x_1, y_1, x_2, y_2, oznaczające odpowiednio współrzędną x – ową i y – ową lewego górnego rogu i współrzędną x – ową i y – ową prawego dolnego rogu pierwszego prostokąta.

W drugim wierszu wejścia znajdują się 4 liczby całkowite: x_3, y_3, x_4, y_4, oznaczające odpowiednio współrzędną x – ową i y – ową lewego górnego rogu i współrzędną x – ową i y – ową prawego dolnego rogu drugiego prostokąta.

Wszystkie współrzędne są nie mniejsze niż 0 i nie większe niż 1000000.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczę całkowitą równą wartości pola części wspólnej dwóch prostokątów.

Przykład

dla danych wejściowych:

```
0 3 4 0
2 4 6 1
```

poprawnym wynikiem jest:

```
4
```

2.4 Dzień trzeci

Zadanie: **Wykreślanka** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Wykreślanka

Jasio wypisał ciąg n liczb całkowitych. Ciągiem *dobrym* nazwiemy ciąg kolejnych liczb naturalnych $1, 2, 3, ...$ itd. Jasio chciałby wykreślić jak najmniej liczb w taki sposób, aby pozostałe liczby utworzyły *dobry* ciąg. Jeśli Jasio nie może utworzyć żadnego *dobrego* ciągu, to powinien wykreślić wszystkie liczby.

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 10^6$), oznaczająca ilość liczb w ciągu Jasia. Kolejny wiersz zawiera ciąg n liczb całkowitych $a_1, a_2, ..., a_n$ ($1 \leqslant a_i \leqslant 10^9$), oznaczających kolejne wartości liczb w ciągu Jasia.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą równą minimalnej ilości liczb, które powinien wykreślić Jaś.

Przykład

dla danych wejściowych:

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

poprawnym wynikiem jest:

```
3
```

Zadanie: **Piłeczka** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Piłeczka

Asia dostała na urodziny magiczną piłeczkę. Piłeczka ta, spadając z pewnej wysokości, odbija się na wysokość dwa razy większą. Asia zrzuciła piłeczkę z balkonu z pewnej wysokości x. Zastanawia się teraz, po ilu odbiciach piłeczka znajdzie się na wysokości w.

Wejście

W pierwszej linii wejścia znajduje się jedna liczba całkowita z ($1 \leqslant z \leqslant 10^6$), oznaczająca liczbę zestawów danych. Dla każdego zestawu danych w nowym wierszu znajdują się dwie liczby całkowite x i w ($1 \leqslant x \leqslant 10^9, 0 \leqslant w \leqslant 10^9$).

Wyjście

Dla każdego zestawu danych w z kolejnych wierszach powinna znaleźć się jedna liczba całkowita równa liczbie odbić, po których piłeczka znajdzie się na wysokości w.

Przykład

dla danych wejściowych:

```
2
3 4
2 6
```

poprawnym wynikiem jest:

```
1
2
```

Zadanie: **Bałwanek** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Bałwanek

Pewnej mroźnej zimy Kasia i Asia pojechały do parku. Jak to w zimie, park jest pełen śniegu, a dokładniej znajduje się w nim x litrów śniegu. Gdy Asia rozmawiała przez telefon, Kasia postanowiła ulepić bałwana. Zanim Asia skończyła rozmawiać Kasia ulepiła już jedną kulę z k litrów śniegu.

Asia chce, aby bałwan był zbudowany z 3 kul śniegowych i dodatkowo kula leżąca na innej kuli musi być 2 razy mniejsza. Asia nie może już powiększać ani pomniejszać kuli zrobionej przez Kasię i musi ją wykorzystać, gdyż Kasia mogłaby się pogniewać.

Dziewczynki chcę zbudować jak największego bałwana. Teraz zastanawiają się, z ilu mililitrów śniegu może być zbudowany ich bałwan.

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite x, k ($1 \leqslant x \leqslant 10^6, 1 \leqslant k \leqslant 10^5, k < x$), oznaczające odpowiednio ilość litrów śniegu w parku oraz z ilu litrów śniegu zbudowana jest jedna z kul śnieżnych bałwana.

Wyjście

W pierwszym i jedynym wierszu wyjścia powinna znajdować się jedna liczb całkowita, równa maksymalnej objętości bałwana (wynik podaj w mililitrach).

Przykład

dla danych wejściowych:

```
10 2
```

poprawnym wynikiem jest:

```
7000
```

Zadanie: **Manipulacja rankingu** Autor zadania: **Joachim Jelisiejew**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Manipulacja rankingu

Dymówka włamał się do obozowej sprawdzarki i może teraz manipulować rankingiem. Konkretniej mówiąc, może on ustawić wagę każdego zadania. Sumaryczna liczba punktów jest obliczana jako:

$$\sum_{\textit{dla każdego zadania}} \textit{waga zadania} * \textit{liczba punktów za zadanie}$$

Zadania na obozie są oceniane binarnie: rozwiązanie otrzymuje 0 lub 100 punktów. Waga zadania może być ustawiona na dowolną liczbę całkowitą z przedziału [0, 2000]. Dymówka niestety nie umie wyliczyć, na ile opłaca mu się zmiana wag. Pomóż mu w tym.

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite n, m ($1 \leqslant n, m \leqslant 1000$), oznaczające odpowiednio liczbę uczestników obozu oraz liczbę zadań na obozie. W następnych n linijkach wyniki kolejnych uczestników: w każdej linijce m liczb (każda równa 0 lub 100), oznaczające wyniki uczestnika w kolejnych zadaniach. Wynik Dymówki jest podany jako pierwszy.

Wyjście

W jedynej linijce wypisz dwie liczby: pierwsza oznaczająca najlepszą pozycję w rankingu, jaką może zdobyć manipulacją Dymówka, oraz druga, oznaczająca minimalną liczbę osób, z którymi będzie on ex aequo na tej najlepszej pozycji. Pozycje w rankingu liczone są od 1.

Przykład

dla danych wejściowych:

```
3 1
100
0
0
```

poprawnym wynikiem jest:

```
1 1
```

2.5 Dzień czwarty

Zadanie: **Pinezki** Autor zadania: **Joanna Bujnowska**

Dostępna pamięć: **32 MB** *Dzień 4, 23.09.2010*

Pinezki

Jaś wraca do domu, jednak na moście ktoś rozłożył w linii prostej pinezki. Most jest tak wąski, że nie można przejść obok – pinezki trzeba przeskoczyć. Pinezki nie są rozłożone wszystkie obok siebie, czasami znajdują się miejsca, na których można stanąć.

Most złożony jest z n desek i na każdej z nich może znajdować się pinezka. Na deskach bez pinezek Jaś może stawać, a na tych z pinezkami już nie. Chcielibyśmy wiedzieć, jak duży skok musi posiadać Jaś, aby mógł przejść na drugą stronę mostu. Długość skoku to liczb desek, które może przeskoczyć Jaś.

Wejście

W pierwszej linii wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 10^6$), oznaczająca liczbę desek, z których zbudowany jest most. W kolejnym wierszu znajduje się n liczb całkowitych $a_1, a_2, ..., a_n$, gdzie a_i oznacza opis i-tej deski: 0 – jeśli na desce nie ma pinezki, 1 – jeśli na desce jest pinezka.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą równą długości skoku Jasia, jaki musi posiadać, aby przedostać się na drugą stronę mostu.

Przykład

dla danych wejściowych:

```
7
0 1 0 1 1 1 1
```

poprawnym wynikiem jest:

```
4
```

Zadanie: **Szachy** Autor zadania: **Bartosz Nowierski (Wiosenny Turniej Programistyczny)**

Dostępna pamięć: **32 MB** *Dzień 4, 23.09.2010*

Szachy

Tatuś małego Pawełka jest znanym na świecie arcymistrzem szachowym i bardzo chciałby, żeby jego synek podążył jego śladami. Dlatego zaczął go już uczyć grać w szachy, mimo młodego wieku chłopca. Niestety Pawełkowi kiepsko idzie nauka, najwyraźniej nie ma do tego smykałki.

Przez pierwsze 3 dni uczył się ruchów pionka, następne 8 – ruchów skoczka, a teraz uczy się ruchów wieży, z którą radzi sobie dużo lepiej i szacuje się, że po drugim dniu opanuje w pełni jej możliwości. W każdym razie po pierwszym dniu Pawełek umie już poruszać się nią w poziomie.

Chłopiec jeszcze nie wie, jaka lekcja go czeka następnego dnia, więc jest święcie przekonany, że wieżę można przesunąć na dowolne pole w tym samym wierszu i na żadne inne.

Pawełek wymyślił własną zabawę – narysował szachownicę (niekoniecznie o wymiarach 8x8, ale kwadratową) i powypisywał na jej polach różne liczby, po jednej na każdym polu. Następnie zaczął stawiać na polach wieże tak, aby suma liczb na polach zajętych przez nie była możliwie jak największa. Chłopiec może postawić dowolną liczbę wież. Gdy nie postawi ani jednej, przyjmuje się, że suma jest równa O. Żeby zabawa nie była zbyt prosta, trzyma się reguły, że żadne dwie wieże nie mogą stać na jednym polu, ani wzajemnie siebie atakować (według jego aktualnego stanu wiedzy o szachach).

Mając daną wielkość szachownicy oraz liczby jakie Pawełek powpisywał w pola szachownicy, podaj jaką największą sumę może uzyskać, stawiając wieże zgodnie z podanymi regułami.

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 200$), mówiąca jaka jest wysokość i szerokość szachownicy. W kolejnych n wierszach opisu przedstawiona jest zawartość pól w kolejnych wierszach szachownicy. Tak więc w wierszu i-tym (spośród tych n) znajduje się n liczb całkowitych (z przedziału od -1000000 do 1000000), będących wartościami zapisanymi w kolejnych polach i-tego wiersza szachownicy.

Wyjście

Należy wypisać jedną liczbę całkowitą – maksymalną sumę, jaką chłopiec może uzyskać.

Przykład

dla danych wejściowych:

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

poprawnym wynikiem jest:

```
9
```

Zadanie: **Kratki** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 4, 23.09.2010*

Kratki

Jaś ma wiele czystych białych kartek papieru. Kartki mają różne wielkości. Jaś postanowił przerobić pewną białą kartkę na kartkę w kratkę – czyli chciałby dorysować pewną liczbę pionowych kresek i pewną liczbę poziomych kresek (równoległych do boków kartki), które podzielą kartkę na małe prostokąty.

Jaś musi jednak zachować odstęp d milimetrów pomiędzy kolejnymi liniami, zaś odległość linii od dowolnego boku musi wynosić co najmniej 1 milimetr. Jaś chciałby wybrać taką kartkę, którą będzie mógł podzielić na jak największą liczbę prostokątów (prostokąty utworzone pomiędzy pierwszą linią a bokiem kartki również liczymy). Ze wszystkich kartek, które będzie mógł podzielić na taką samą liczbę prostokątów, chciałby wybrać kartkę o największej powierzchni. Znajdź tą powierzchnię.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite n, d ($1 \leqslant n \leqslant 10^6, 1 \leqslant d \leqslant 100$), oznaczające odpowiednio liczbę białych kartek Jasia oraz minimalny odstęp pomiędzy liniami. Następnych n wierszy zawiera opisy kolejnych kartek papieru. Każdy z wierszy składa się z dwóch liczb całkowitych a_i, b_i ($2 \leqslant a_i, b_i \leqslant 10^4$), oznaczających odpowiednio długość i szerokość i-tej kartki.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę oznaczającą powierzchnię kartki, którą powinien wybrać Jaś.

Przykład

dla danych wejściowych:

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

poprawnym wynikiem jest:

```
18
```

Zadanie: **Patyki** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 4, 23.09.2010*

Patyki

Bajtuś znalazł w lesie 3 patyki. Teraz chciałby wiedzieć, czy może z nich zbudować trójkąt prostokątny lub równoboczny. Bajtuś nie może łamać patyków, może je wykorzystać tylko w całości.

Wejście

Pierwszy wiersz wejścia zawiera trzy liczby całkowite a, b, c ($1 \leqslant a, b, c \leqslant 1000$), oznaczające odpowiednio długość pierwszego, drugiego i trzeciego patyka.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę: 0 – jeśli Bajtuś nie może zbudować ani trójkąta prostokątnego, ani równobocznego, 1 – jeśli Bajtuś może zbudować tylko trójkąt prostokątny, 2 – jeśli Bajtuś może zbudować tylko trójkąt równoboczny.

Przykład

dla danych wejściowych:

```
3 4 5
```

poprawnym wynikiem jest:

```
1
```

2.6 Dzień piąty

Zadanie: **Grusze i jabłonie** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 5, 24.09.2010*

Grusze i jabłonie

W pewnym sadzie rosną grusze i jabłonie, wszystkie w linii prostej, oddalone co 1 metr od siebie. Właściciel sadu, Pan Wiktor, chciałby znaleźć gruszę najbardziej oddaloną od jabłoni. Pomóż mu znaleźć te drzewa i podaj odległość pomiędzy nimi.

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($2 \leqslant n \leqslant 10^6$), oznaczającą liczbę drzew w sadzie. Kolejny wiersz zawiera ciąg n liczb całkowitych $a_1, a_2, ...a_n$, gdzie a_i oznacza rodzaj i-tego drzewa: 0 – oznacza gruszę, 1 – oznacza jabłoń. Można założyć, że w sadzie rośnie co najmniej jedna grusza i co najmniej jedna jabłoń.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą, równą maksymalnej odległości pomiędzy gruszą a jabłonią.

Przykład

dla danych wejściowych:

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

poprawnym wynikiem jest:

```
3
```

Zadanie: **Farby** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 5, 24.09.2010*

Farby

W ILO zaczęły się generalne porządki. Pani dyrektor postanowiła, że pan konserwator Jarek powinien odświeżyć sale na parterze i pomalować je na jej ulubione kolory.

Na parterze mamy 6 różnych sal. Pani dyrektor chciałaby, aby pierwsza sala została pomalowana na żółto, druga na zielono, trzecia na niebiesko, czwarta na fioletowo, piąta na czerwono, a ostatnia na pomarańczowo.

Pan Jarek przypomniał sobie, że ze starych zapasów posiada pewną ilość farby czerwonej, żółtej oraz niebieskiej. Ku zadowoleniu pani dyrektor stwierdził, że trzy kolory wystarczą mu na spełnienie prośby dyrekcji, ponieważ ze szkoły podstawowej pamiętał, że aby otrzymać kolor zielony, wystarczy zmieszać w tych samych ilościach farbę żółtą z niebieską, analogicznie fioletowa powstanie z farby niebieskiej i czerwonej, a pomarańczowa z farby żółtej oraz czerwonej.

Pan Jarek chciałby wiedzieć, ile farby każdego rodzaju (czerwonej, żółtej i niebieskiej) musi dokupić, aby móc wymalować wszystkie sale.

Wejście

W pierwszym wierszu podane są trzy liczby całkowite dodatnie c , z oraz n pooddzielane pojedynczymi spacjami: ilość farby czerwonej ($0 < c \leqslant 10^9$), ilość farby żółtej ($0 < z \leqslant 10^9$), oraz ilość farby niebieskiej ($0 < n \leqslant 10^9$), którą posiada Pan Jarek. W kolejnych 6 wierszach podane są liczby całkowite dodatnie $0 < p_i \leqslant 1000$, wyrażające ilość litrów farby potrzebnych na wymalowanie poszczególnych sal.

Wyjście

Twój program powinien wypisać trzy liczby, oznaczające odpowiednio ilość farby czerwonej, żółtej, oraz niebieskiej, którą trzeba dokupić, aby móc wymalować wszystkie sale.

Przykład

dla danych wejściowych:

```
20 10 8
10
4
5
3
3
8
```

poprawnym wynikiem jest:

```
0 6 0.5
```

Zadanie: **Siłownia** Autor zadania: **Robert Kozikowski**

Dostępna pamięć: **32 MB** *Dzień 5, 24.09.2010*

Siłownia

Adrian stwierdził, że ma wątłą klatę i okrągły brzuszek. Z tego powodu postanowił poćwiczyć klatę. Wyznaczył sobie, że do końca roku uda mu się wycisnąć $n, (1 \leqslant n \leqslant 10^9)$ mikrogramów. Adrian pracuje w dwóch różnych browarach, w których produkowane są piwa (oczywiście bezalkoholowe). W jednym produkowane są piwa o wadze a mikrogramów, a w drugim piwa o wadze b mikrogramów, $(1 \leqslant a, b \leqslant n)$. Adrian postanowił wyciskać piwa na klatę. Adrian zaczyna od najmniejszego możliwego ciężaru i idzie stopniowo w górę, idąc zawsze do pierwszego większego ciężaru. Zastanawia się teraz, ile różnych ciężarów uda mu się wycisnąć zanim osiągnie ciężar maksymalny. Adrian nie może przenosić piw z jednego browaru do drugiego. W każdym browarze wystarczy mu piw na osiągnięcie celu.

Wejście

W pierwszej i jedynej linii wejścia znajdują się trzy liczby całkowite: a, b, n, będące odpowiednio wagą piw w pierwszym browarze, wagą piw w drugim browarze oraz ciężaru docelowego Adriana.

Wyjście

W pierwszej linii wyjścia powinna znaleźć się jedna liczba całkowita, będąca liczbą różnych ciężarów, które wyciśnie Adrian.

Przykład

dla danych wejściowych:

```
5 7 15
```

poprawnym wynikiem jest:

```
5
```

Zadanie: **Dziewczynki** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 5, 24.09.2010*

Dziewczynki

Chłopcy i dziewczynki ustawili się w szereg, osoba tuż obok osoby. Zastanawiamy się teraz, ilu minimalnie chłopców musi usunąć się z szeregu, aby pod rząd stało k dziewczynek, jedna tuż obok drugiej i pomiędzy nimi wszystkimi nie stał żaden chłopiec.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n, k ($1 \leqslant k \leqslant n \leqslant 10^6$), oznaczające odpowiednio liczbę osób ustawionych w szeregu oraz liczbę dziewczynek, jakie chcemy, aby stały pod rząd . Kolejny wiersz wejścia zawiera n liczb całkowitych 0 lub 1, oznaczających kolejne osoby ustawione w szeregu: 0 – oznacza dziewczynkę, 1 – chłopca.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, oznaczającą minimalną liczbę chłopców, którzy powinni usunąć się z szeregu, lub jedno słowo 'NIE', gdy nie da się usunąć chłopców tak, aby k dziewczynek stało pod rząd.

Przykład

dla danych wejściowych:

```
8 3
0 1 1 0 1 0 1 0
```

poprawnym wynikiem jest:

```
2
```

Rozdział 3

Rozwiązania

3.1 Dzień próbny

Opracowanie: **Dodawanie** Autor opracowania: **Adrian Jaskółka**

Dostępna pamięć: **32 MB** *Dzień 0, 19.09.2010*

Dodawanie

Rozwiązanie wzorcowe

Trikiem było w tym zadaniu użycie operatora dodawania, które dawało rozwiązanie o złożoności $O(1)$. Trzeba było być ostrożnym, aby nie użyć 8-bitowych liczb całkowitych, gdyż wtedy rozwiązanie padało dla dużych wartości a i b.

3.2 Dzień pierwszy

Opracowanie: **Koszykarz** Autor opracowania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 1, 20.09.2010*

Koszykarz

Rozwiązanie wzorcowe

Na początku obliczmy, ile Kozik musi urosnąć, odejmując od wymaganego wzrostu wzrost Kozika. Jeśli wartość ta nie jest dodatnia, to Kozik nie musi ani razu uderzać się w głowę. W przeciwnym wypadku wystarczy podzielić wartość, jaką Kozik musi jeszcze urosnąć, przez wartość, o jaką rośnie guz Kozika z każdym uderzeniem (należy pamiętać, że jeśli reszta z dzielenia nie wynosi zero, to należy do wyniku dodać 1).

```
1 wczytaj(k, w, m)
2 if w > k then
3     wypisz((w - k + m - 1) / m)
4 else
5     wypisz(0)
```

Opracowanie: **Odchudzanie** Autor opracowania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 1, 20.09.2010*

Odchudzanie

Rozwiązanie wolne

Najprostszym rozwiązaniem jest sprawdzenie każdej pary fragmentu, w której Kozik może rozpocząć i zakończyć swój trening.

```
1 wczytaj(n, t[])
2 wynik := 0
3 for i := 1 to n do
4     for j := i to n do
5         if t[j] - t[i] > wynik then
6             wynik := t[j] - t[i]
7 wypisz(wynik)
```

Takie rozwiązanie jest niestety zbyt wolne, gdyż działa w złożoności czasowej $O(n^2)$. Złożoność pamięciowa wynosi $O(n)$.

Rozwiązanie wzorcowe

Zauważmy, że dla każdego możliwego końca fragmentu najlepszym początkiem będzie maksymalna waga Kozika, która występowała od początku treningu do aktualnego dnia (wtedy spadek wagi będzie właśnie największy). Taką maksymalną wagę możemy łatwo obliczać, aktualizując nasze maksimum z każdym nowym dniem.

```
1 maksimum := 0
2 wynik := 0
3 for k := 1 to n do
4     wczytaj(x)
5     if x > maksimum then
6         maksimum := x
7     if maksimum - x > wynik then
8         wynik := maksimum - x
9 wypisz(wynik)
```

Rozwiązanie działa w złożoności czasowej $O(n)$. Złożoność pamięciowa wynosi $O(1)$.

Rozwiązanie błędne

Rozwiązanie, które znajduje maksimum i minimum w całym ciągu i liczy ich różnice, jest błędne. Prosty przykład: Kozik pierwszego dnia ważył 1, a drugiego 10 – maksymalny spadek Kozika powinien wynosić 0.

Opracowanie: **Wężyk** Autor opracowania: **Adam Iwaniuk**

Dostępna pamięć: **32 MB** *Dzień 1, 20.09.2010*

Wężyk

Najprostsze najlepsze

Zauważmy, że zawsze musimy wypisać n wierszy, a więc najprościej będzie zaprojektować jedną pętlę, która wykonuje n przejść, w których coś robi, oraz wypisuje znak nowej linii.

Teraz zastanówmy się, co ma się wykonywać wewnątrz tej pętli. Musimy wypisać n liczb oddzielonych spacjami, a więc najlepiej zrobić to za pomocą pętli, która wykona n przejść i wypisze po jednej liczbie. Zauważmy, że ciąg liczb jest albo ciągiem rosnących liczb o jeden lub malejących o jeden, a więc aby wypisać odpowiednie liczby, potrzebujemy wiedzieć dwie rzeczy:

1. Jaka jest liczba startowa?
2. Czy jest to ciąg rosnący czy malejący?

Drugi punkt łatwo możemy rozwiązać sprawdzając, czy numer wiersza (tzn. licznik naszej głównej pętli) jest liczbą parzystą.

Natomiast co do pierwszego punktu możemy wykonać proste obserwacje. Rozbijmy to także na dwa przypadki: wiersze parzyste i nieparzyste. Niech x będzie numerem wiersza (numerowanie rozpoczynamy od 1).

Łatwo wymyślić wzór na pierwszą liczbę w wierszach nieparzystych:

```
1 -> 1
3 -> 2 * n + 1
x -> (x - 1) * n + 1
```

natomiast dla wierszy parzystych:

```
2 -> 2 * n
4 -> 4 * n
x -> x * n
```

A więc w zależności, czy wiersz jest parzysty, czy nieparzysty (wystarczy sprawdzić resztę z dzielenia przez 2 na liczniku głównej pętli) wypiszemy ciąg rosnący bądź malejący n liczb, zaczynając od liczby wyliczonej z wzorów, które znaleźliśmy powyżej.

```
1  wczytaj(n)
2  for i := 1 to n do
3      if i mod 2 == 1 then
4          w := (i - 1) * n + 1
5          for j := 1 to n do
6              wypisz(w)
7              w := w + 1
8      else
9          w := i * n
10         for j := 1 to n do
11             wypisz(w)
12             w := w - 1
13     wypisz(nowa_linia)
```

Opracowanie: **Znak działania** Autor opracowania: **Robert Kozikowski, Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 1, 20.09.2010*

Znak działania

Rozwiązanie wzorcowe

W zadaniu należy obliczyć wartość maksymalną z trzech wartości: $a + b$, $a - b$, $a * b$ oraz sprawdzić, czy wynik da się uzyskać więcej niż jednym sposobem. Możemy to zrobić na wiele sposobów. W naszym opracowaniu najpierw policzymy maksymalną wartość, a następnie sprawdzimy iloma sposobami możemy ją osiągnąć.

```
1  wczytaj(a,b)
2  maksimum := a + b
3  znak := '+'
4  if a - b > maksimum then
5     znak := '-'
6     maksimum := a - b
7  if a * b > maksimum then
8     znak := '*'
9     maksimum := a * b
10 sposoby := 0
11 if a + b == maksimum then sposoby := sposoby + 1
12 if a * b == maksimum then sposoby := sposoby + 1
13 if a - b == maksimum then sposoby := sposoby + 1
14 if sposoby > 1 then
15    wypisz(NIE)
16 else
17    if a < 0 then wypisz((a)) else wypisz(a)
18    wypisz(znak)
19    if b < 0 then wypisz((b)) else wypisz(b)
20    wypisz('=')
21    if maksimum < 0 then wypisz((maksimum)) else wypisz(maksimum)
```

Na co należy zwrócić uwagę

Typowe błędy:

- wypisywanie 'NIE', jeśli dwa maksima są równe, ale jeszcze nie obliczyliśmy wyniku (może się okazać, że $a + b == a - b$, ale $a * b > a + b$),
- początkowe ustalenie maksimum na 0 – może się okazać, że maksimum jest ujemne.

3.3 Dzień drugi

Opracowanie: **Monety** Autor opracowania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 2, 21.09.2010*

Monety

Najprostsze najlepsze

Rozwiązanie zadania jest stosunkowo proste. Należy policzyć liczbę monet leżących na jednej stronie i liczbę monet leżących na drugiej stronie, a następnie wybrać minimum z tych wartości.

```
1  wczytaj(n)
2  awers := 0
3  rewers := 0
4  for k := 1 to n do
5      wczytaj(x)
6      if x == 0 then
7          awers := awers + 1
8      else
9          rewers := rewers + 1
10 wypisz(min(awers, rewers))
```

Złożoność czasowa powyższego rozwiązania wynosi $O(n)$.

Opracowanie: **Permutacje** Autor opracowania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 2, 21.09.2010*

Permutacje

Rozwiązanie wolne

Najprostszym rozwiązaniem jest sprawdzanie kolejnych liczb $1, 2, ...n-1, n$ czy występują we wczytanym ciągu.

```
1  wczytaj(n, t[])
2  for i := 1 to n do
3     znalazlem := false
4     for k := 1 to n do
5          if t[k] == i then
6               znalazlem := true
7     if znalazlem == false then
8          wypisz(NIE)
9          zakoncz_program()
10 wypisz(TAK)
```

Powyższe rozwiązanie jest zdecydowanie za wolne – złożoność czasowa wynosi $O(n^2)$, a pamięciowa $O(n)$.

Rozwiązanie wzorcowe

Stwórzmy tablicą n elementową i każdą liczbę 'włóżmy' do odpowiedniej komórki. Na koniec przejdźmy po całej tablicy sprawdzając, czy w każdej komórce jest liczba. Należy uważać, że jeśli liczba jest większa od wielkości tablicy to nic z nią nie robimy (aby nie odwoływać się do nieistniejącej komórki w tablicy).

```
1  wczytaj(n)
2  tab[] := 0
3  for k := 1 to n do
4     wczytaj(x)
5     if x <= n then
6        tab[x] := 1
7  for k := 1 to n do
8     if tab[k] == 0 then
9        wypisz(NIE)
10       zakoncz_program()
11 wypisz(TAK)
```

Rozwiązanie działa w czasie $O(n)$. Złożoność pamięciowa wynosi $O(n)$.

Opracowanie: **Taśma** Autor opracowania: **Adam Iwaniuk**

Dostępna pamięć: **32 MB** *Dzień 2, 21.09.2010*

Taśma

Rozwiązanie brutalne

Najprostszym rozwiązaniem wydaje się dla każdego miejsca na taśmie sprawdzić, ile wynosi szukana wartość. Więc dla każdej liczby, z wyjątkiem ostatniej, sprawdzamy, ile wynosi suma liczb do tej liczby z nią włącznie, oraz suma liczb za tą liczbą (nie robimy tego dla ostatniej liczby, ponieważ odpowiadałoby to sytuacji takiej, gdy wykonujemy cięcie na samym końcu taśmy, co nie sprawi, że podzielimy ją na dwa kawałki). Następnie wyliczamy wartość bezwzględną z różnicy tych sum i poprawiamy wynik.

```
1  wczytaj(n, tab[])
2  wynik := 1000000000
3  for i := 1 to n-1 do
4      sumal := sumap := 0
5      for j := 1 to i do
6          sumal := sumal + tab[i]
7      for j := i + 1 to n do
8          sumap := sumap + tab[i]
9      wartosc := abs(sumal - sumap)
10     wynik := min(wartosc, wynik)
11 wypisz(wynik)
```

Niestety przy takim podejściu przy milionie liczb program będzie wykonywał się bardzo długo, ponieważ dla każdego miejsca cięcia (10^6 takich miejsc) będziemy liczyć sumę 10^6 liczb, co da nam w sumie $10^6 * 10^6 = 10^{12}$ operacji.

Rozwiązanie wzorcowe

Zauważmy, że mając sumę wszystkich liczb (można to policzyć zaraz po ich wczytaniu do tablicy) oraz znając sumę liczb po lewej stronie, możemy policzyć sumę liczb po stronie prawej, odejmując sumę liczb z lewej strony od całkowitej sumy. Jeśli mamy policzoną sumę liczb z lewej strony i chcemy wykonać cięcie o jedną liczbę dalej, to suma liczb będzie różnić się jedynie wartością równą tej liczbie, którą przeskakujemy.

```
1  wczytaj(n, tab[])
2  sumac := sumal := 0
3  for i := 1 to n do
4      sumac := sumac + tab[i]
5  wynik := 1000000000
6  for i := 1 to n-1 do
7      sumal := sumal + tab[i]
8      sumap := sumac - sumal
9      wartosc := abs(sumal - sumap)
10     wynik := min(wartosc, wynik)
11 wypisz(wynik)
```

Złożoność czasowa powyższego rozwiązania wynosi $O(n)$, co jest wystarczająco szybkie.

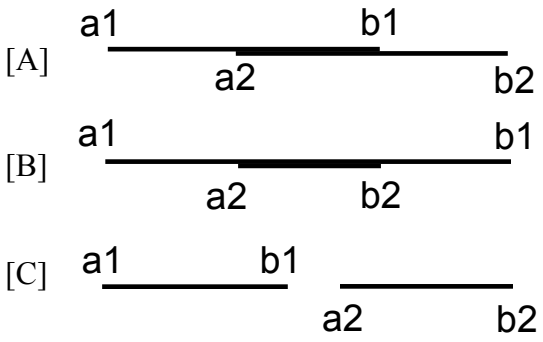
Opracowanie: **Pole** Autor opracowania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 2, 21.09.2010*

Pole

Jeden wymiar

Zastanówmy się, jak rozwiązać zadanie, w którym musielibyśmy znaleźć część wspólną dwóch odcinków. Możemy wyróżnić 3 przypadki:



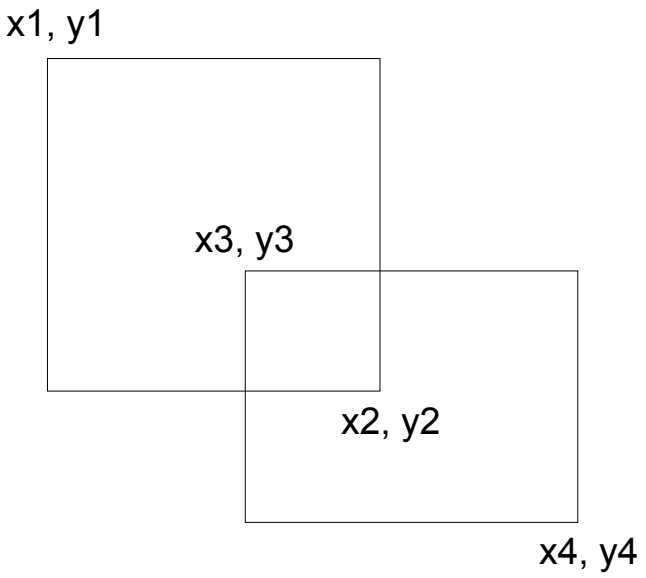
Przypadki ułożenia odcinków.

Zauważmy, że jeśli część wspólna istnieje, to jej początek znajduje się w miejscu rozpoczęcia się odcinka o większej współrzędnej początkowej (czyli maksimum z początków odcinków), natomiast koniec części wspólnej znajduje się w miejscu zakończenia odcinka o mniejszej współrzędnej końcowej (czyli minimum z końców odcinków). Jak sprawdzić, czy część wspólna istnieje? Wystarczy sprawdzić, czy początek części wspólnej jest mniejszy od jego końca.

```
1 function policz_przeciecie(a1, b1, a2, b2)
2 begin
3     poczatek = max(a1, a2)
4     koniec = min(b1, b2)
5     return max(0, koniec - poczatek)
6 end
```

Dwa wymiary

Przejdźmy do rozwiązania zadania, czyli znalezienia części wspólnej dwóch prostokątów.



Przykładowe ułożenie dwóch prostokątów.

Aby obliczyć długości boków prostokąta, będącego częścią wspólną, wystarczy znaleźć części wspólne odpowiednich odcinków danych prostokątów. Pierwszy bok obliczymy licząc część wspólną boków prostokątów ograniczonych współrzędnymi x – owymi, a drugi licząc część wspólną boków ograniczonych współrzędnymi y – owymi.

Pseudokod mógłby wyglądać następująco:

```
1 dlugosc1 = policz_przeciecie(x1, x2, x3, x4)
2 dlugosc2 = policz_przeciecie(y4, y3, y2, y1)
3 wypisz(dlugosc1 * dlugosc2)
```

Zauważmy, że złożoność powyższego rozwiązania wynosi $O(1)$, ponieważ nie wykonujemy żadnej pętli, a każda instrukcja wykonuje się w czasie stałym,. W rozwiązaniu należy pamiętać, aby używać zmiennych 64 – bitowych, ponieważ końcowy wynik może przekraczać dopuszczalne wartości zmiennych 32 – bitowych (długości boków części wspólnej prostokątów mogą wynosić 10^6, więc ich iloczyn może wynosić 10^{12}).

3.4 Dzień trzeci

Opracowanie: **Wykreślanka** Autor opracowania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Wykreślanka

Rozwiązanie wzorcowe

Jaś chce stworzyć jak najdłuższy ciąg liczb $1, 2, 3, ...$ W tym celu musi znaleźć pierwszą jedynkę występującą w ciągu – nawet jeśli występują inne jedynki, to najbardziej opłaca się wziąć najwcześniejszą, gdyż wzięcie innej zmniejsza nam przedział szukania kolejnej liczby. Następnie szukamy pierwszej dwójki występującej za pierwszą jedynką, potem trójki występującej za dwójką itd.

```
1 wczytaj(n)
2 szukany := 1
3 for k := 1 to n do
4     wczytaj(x)
5     if x == szukany then
6         szukany := szukany + 1
7 wypisz(n - szukany + 1)
```

Złożoność czasowa rozwiązania wynosi $O(n)$, a pamięciowa $O(1)$.

Opracowanie: **Piłeczka** Autor opracowania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Piłeczka

Rozwiązanie wzorcowe

Zauważmy, że piłeczka bardzo szybko zwiększa wysokość, do której dolatuje – z każdym odbiciem, wysokość zwiększa się dwukrotnie, więc nawet jeśli piłeczka zostałaby zrzucona z najniższej wysokości 1, to już po 30 odbiciach doleciałaby do wysokości ponad 10^9. Należy zwrócić uwagę, że piłeczka zrzucona z większej wysokości niż wysokość, do której chcemy, aby doleciała, nie musi wykonywać żadnego odbicia.

```
1 wczytaj(z)
2 for k := 1 to z do
3     wczytaj(x, w)
4     odbicia := 0
5     while (x < w) do
6         x := x * 2
7         odbicia := odbicia + 1
8     wypisz(odbicia)
```

Złożoność czasowa powyższego rozwiązania wynosi $O(z \log w)$.

Opracowanie: **Bałwanek** Autor opracowania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Bałwanek

Rozwiązanie wzorcowe

W rozwiązaniu należy rozważyć 4 przypadki:

1. Jeśli kula, którą zbudowała Kasia jest najmniejsza w bałwanie – należy sprawdzić czy z pozostałej ilości śniegu można zbudować 2 większe kule
2. Jeśli kula, którą zbudowała Kasia jest średnia w bałwanie
3. Jeśli kula, którą zbudowała Kasia, jest największa w bałwanie
4. Jeśli kula, którą zbudowała Kasia, jest za duża, aby z pozostałej ilości śniegu wybudować 2 mniejsze kule.

```
1 wczytaj(x, k)
2 if x >= 7*k then wypisz(7000*k)
3 else if 2*x >= 7*k then wypisz(3500*k)
4 else if 4*x >= 7*k then wypisz(1750*k)
5 else wypisz(0)
```

Złożoność czasowa rozwiązania wynosi $O(1)$.

Opracowanie: **Manipulacja rankingu** Autor opracowania: **Joachim Jelisiejew**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Manipulacja rankingu

Rozwiązanie wzorcowe

Powiemy, że zawodnik A jest *obiektywnie* lepszy od Dymówki, jeżeli A zrobił wszystkie zadania, które zrobił Dymówka. Zauważmy, że przy pomocy manipulacji Dymówka nie może wyprzedzić w rankingu zawodnika *obiektywnie* lepszego.

Stwierdzamy, że zawsze istnieje manipulacja, dzięki której Dymówka ostro wyprzedza w rankingu wszystkich zawodników, którzy nie są obiektywnie lepsi od niego i jest ex aequo ze wszystkimi obiektywnie lepszymi (na 1 pozycji). Taka manipulacja jest optymalna z punktu widzenia Dymówki.

Manipulacja polega na ustawieniu wag 1 wszystkim zadaniom, które Dymówka rozwiązał, oraz wag 0 wszystkim zadaniom, których nie rozwiązał. Pozostawiamy czytelnikowi weryfikację, że ta manipulacja faktycznie spełnia założenia (ważne jest tutaj, że są tylko 2 możliwe oceny za zadanie).

Zatem wystarczy policzyć, ile jest zawodników obiektywnie lepszych od Dymówki. Możemy to bez trudu zrobić liniowo względem ilości danych wejściowych:

```
1  wczytaj(n, m, wynik_dymowki[])
2  ile_lepszych := 1
3  for i := 2 to n do
4      lepszy := true
5      for j := 1 to m do
6          wczytaj(wynik)
7          if wynik != 100 and wynik_dymowki[j] == 100 then
8              lepszy := false
9      if lepszy == true then
10         ile_lepszych := ile_lepszych + 1
11 wypisz(1, ile_lepszych)
```

Złożoność czasowa powyższego rozwiązania wynosi $O(nm)$.

3.5 Dzień czwarty

Opracowanie: **Pinezki** Autor opracowania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 4, 23.09.2010*

Pinezki

Rozwiązanie wzorcowe

Rozwiązaniem zadania będzie maksymalna liczba pinezek leżących obok siebie. Z każdą kolejno napotkaną pinezką zwiększamy sumę o jeden, natomiast przy napotkaniu pustej deski zerujemy naszą sumę.

```
1  wczytaj(n)
2  suma := 0
3  wynik := 0
4  for k := 1 to n do
5      wczytaj(pinezka)
6      if pinezka == 1 then
7          suma := suma + 1
8      else
9          suma := 0
10     wynik := max(wynik, suma)
11 wypisz(wynik)
```

Złożoność czasowa rozwiązania wynosi $O(n)$.

Opracowanie: **Szachy** Autor opracowania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 4, 23.09.2010*

Szachy

Rozwiązanie wzorcowe

Na początku zauważmy, że nie zawsze opłaca się stawiać wieżę w każdym wierszu szachownicy. Opłaca nam się to tylko w przypadku, gdy wszystkie liczby w danym wierszu są nieujemne.

Tak więc naszym rozwiązaniem będzie znalezienie w każdym wierszu maksimum z wartości $\{0, w_1, ...w_n\}$, gdzie w_i oznacza liczbę stojącą w i-tej kolumnie rozpatrywanego wiersza.

```
1 wczytaj(n)
2 wynik := 0
3 for wiersz := 1 to n do
4     akt := 0
5     for kolumna := 1 to n do
6         wczytaj(x)
7         akt := max(akt, x)
8     wynik := wynik + akt
9 wypisz(wynik)
```

Złożoność czasowa takiego rozwiązania wynosi $O(n^2)$.

Opracowanie: **Kratki** Autor opracowania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 4, 23.09.2010*

Kratki

Rozwiązanie błędne

Popularnym błędem może być wypisanie maksymalnej powierzchni kartki. Nie jest prawdą, że kartkę o maksymalnej powierzchni da się podzielić na największą liczbę prostokątów.

Rozwiązanie wzorcowe

Maksymalna liczba prostokątów, na jakie można podzielić kartkę, wynosi $(2 + (a - 2)/d) * (2+(b-2)/d)$, gdzie znak '/' jest wzięciem części całkowitej z dzielenia bez reszty. Pierwszą linię rysujemy w odległości 1 milimetra od brzegu, a każdą następną (tyle ile się zmieści) rysujemy co d milimetrów. Dla każdej kartki należy obliczyć liczbę prostokątów na jakie możemy ją podzielić i wybrać maksimum.

```
1  wczytaj(n, d)
2  maksimum := 0
3  for k := 1 to n do
4      wczytaj(a, b)
5      liczba_kratek := (2 + (a - 2) / d) * (2 + (a - 2) / d)
6      if liczba_kratek == maksimum then
7          powierzchnia := max(powierzchnia, a * b)
8      if liczba_kratek > maksimum then
9          maksimum := liczba_kratek
10         powierzchnia := a * b
11 wypisz(powierzchnia)
```

Opracowanie: **Patyki** Autor opracowania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 4, 23.09.2010*

Patyki

Rozwiązanie wzorcowe

Zadanie należy rozwiązać rozważając kilka przypadków. Aby można było zbudować trójkąt prostokątny, to długość najdłuższego boku podniesiona do kwadratu musi być równa sumie kwadratów długości boków krótszych. Aby trójkąt był równoboczny to wszystkie patyki muszą być równej długości.

```
1 wczytaj(a, b, c)
2 if (a == b) and (b == c) then
3     wypisz(2) else
4 if (a*a == b*b + c*c) or (b*b == a*a + c*c) or (c*c == a*a + b*b) then
5     wypisz(1)
6 else
7     wypisz(0)
```

3.6 Dzień piąty

Opracowanie: **Grusze i jabłonie** Autor opracowania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 5, 24.09.2010*

Grusze i jabłonie

Rozwiązanie wolne

Najprostszym rozwiązaniem zadania jest sprawdzenie każdej pary drzewek i wybranie maksymalnej odległości pomiędzy różnymi drzewkami.

```
1  wynik := 0
2  wczytaj(n, drzewko[])
3  for k := 1 to n - 1 do
4      for i := k + 1 to n do
5          if drzewko[k] <> drzewko[i] then
6              wynik := max(wynik, i - k)
7  wypisz(wynik)
```

Powyższe rozwiązanie działa niestety zbyt wolno – złożoność czasowa wynosi $O(n^2)$.

Rozwiązanie wzorcowe

Najbardziej oddalonymi drzewkami będzie pierwsza (występująca w ciągu) grusza z ostatnią jabłonią lub pierwsza jabłoń z ostatnią gruszą. Gdyby tak nie było, to moglibyśmy wziąć odpowiednio wcześniejsze (późniejsze) drzewko i mielibyśmy większą odległość pomiędzy nimi. Drzewka te możemy znaleźć w czasie liniowym, stąd złożoność czasowa rozwiązania wynosi $O(n)$.

```
1  wczytaj(n)
2  pie_grusza := 0
3  pie_jablon := 0
4  ost_grusza := 0
5  ost_jablon := 0
6  for k := 1 to n do
7      wczytaj(drzewko)
8      if drzewko = 0 and pie_grusza = 0 then pie_grusza := k
9      if drzewko = 1 and pie_jablon = 0 then pie_jablon := k
10     if drzewko = 0 then ost_grusza := k
11     if drzewko = 1 then ost_jablon := k
12 wypisz(max(ost_grusza - pie_jablon, ost_jablon - pie_grusza))
```

Opracowanie: **Farby** Autor opracowania: **Robert Kozikowski**

Dostępna pamięć: **32 MB** *Dzień 5, 24.09.2010*

Farby

Rozwiązanie

W rozwiązaniu najpierw wczytujemy obecną ilość farby: c, z oraz n, a następnie wczytujemy ilość farby wymaganą na pokolorowanie każdej sali i odejmujemy tę ilość od objętości odpowiedniej puszki.

Aby ułatwić sobie obliczenia i móc operować tylko na liczbach całkowitych, dobrym pomysłem jest prowadzić obliczenia na liczbach dwa razy większych i dopiero przy wypisywaniu dzielić wynik przez dwa.

Należy pamiętać, że jeśli w którejś puszce będzie nieujemna ilość farby, wtedy nie musimy niczego dokupić. W przeciwnym wypadku musimy dokupić wartość bezwzględną z ilości farby w puszce.

```
1  wczytaj(puszka_cze, puszka_zol, puszka_nie)
2  wczytaj(zol, zie, nie, fio, cze, pom)
3  ile_cze := max(0, fio + pom + 2 * cze - 2 * puszka_cze)
4  ile_zol := max(0, zie + pom + 2 * zol - 2 * puszka_zol)
5  ile_nie := max(0, zie + fio + 2 * nie - 2 * puszka_nie)
6  if (ile_cze mod 2 == 0) then
7      wypisz(ile_cze/2)
8  else
9      wypisz(ile_cze/2 + ".5")
10 if (ile_zol mod 2 == 0) then
11     wypisz(ile_zol/2)
12 else
13     wypisz(ile_zol/2 + ".5")
14 if (ile_nie mod 2 == 0) then
15     wypisz(ile_nie/2)
16 else
17     wypisz(ile_nie/2 + ".5")
```

Opracowanie: **Siłownia** Autor opracowania: **Robert Kozikowski, Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 5, 24.09.2010*

Siłownia

Rozwiązanie brutalne

Niech $\lfloor x \rfloor$ to część całkowita z liczby x, a to wielkość piw w pierwszym browarze, b to wielkość piw w drugim browarze, a n to docelowy ciężar.

W rozwiązaniu brutalnym tworzymy tablicę t wielkości n zapełnioną samymi 0. Następnie odznaczamy w tablicy wielokrotności a na 1 oraz wielokrotności b na 1. Na koniec liczymy sumę w tablicy t.

```
1 wczytaj(a, b, n)
2 t[] := {0}
3 for i := 1 to n/a do
4     t[a*i] := 1
5 for i := 1 to n/b do
6     t[b*i] := 1
7 for i := 1 to n do
8     wynik := wynik + t[i]
9 wypisz(wynik)
```

Powyższe rozwiązanie działa w czasie $O(n)$, a jego złożoność pamięciowa to $O(n)$. Dla maksymalnych danych, czyli $n = 10^9$ program zdecydowanie przekroczy limit czasu oraz dostępny limit pamięci.

Rozwiązanie wzorcowe

Zauważmy, że w pierwszym browarze Adrian może wycisnąć $\lfloor \frac{n}{a} \rfloor$ różnych ciężarów, będą to odpowiednio $a, 2a, ...$ W drugim browarze Adrian może wycisnąć $\lfloor \frac{n}{b} \rfloor$ różnych ciężarów, odpowiednio $b, 2b, ...$

Zastanówmy się teraz, ile jest takich ciężarów, że Adrian może je wycisnąć w obu siłowniach. Spróbujmy znaleźć najmniejszy taki ciężar. Będzie to najmniejsza liczba podzielna przez a i b, czyli $nww(a, b)$. Zauważmy, że wszystkie kolejne powtórzenia będą wielokrotnościami $nww(a, b)$. Zatem wszystkich takich ciężarów mniejszych równych n będzie $\left\lfloor \frac{n}{nww(a,b)} \right\rfloor$, a wszystkich różnych ciężarów wyciskanych przez Adriana będzie $\lfloor \frac{n}{a} \rfloor + \lfloor \frac{n}{b} \rfloor - \left\lfloor \frac{n}{nww(a,b)} \right\rfloor$, gdyż licząc sumę $\lfloor \frac{n}{a} \rfloor + \lfloor \frac{n}{b} \rfloor$ dwukrotnie policzymy te ciężary, które Adrian może wycisnąć w dwóch browarach. $nww(a, b) = \frac{a*b}{nwd(a,b)}$, a $nwd(a, b)$ możemy łatwo policzyć korzystając z algorytmu Euklidesa (opis algorytmu można znaleźć w wykładzie Teoria Liczb I)

```
1 wczytaj(a, b, n)
2 wynik := n/a + n/b + n/(a*b/nwd(a, b))
3 wypisz(wynik)
```

Rozwiązanie działa w złożoności logarytmicznej ze względu na a i b. Tyle zajmuje policzenie $nwd(a, b)$.

Opracowanie: **Dziewczynki** Autor opracowania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 5, 24.09.2010*

Dziewczynki

Rozwiązanie wzorcowe

Zadania rozwiązujemy wykorzystując sposób "gąsienicy". W dwóch zmiennych trzymamy odpowiednio głowę gąsienicy i jej tył. Niech *koniec* będzie jej tyłem, a *początek* jej głową. Początkowo obie zmienne ustawiamy na pierwszej osobie w szeregu. Następnie *początek* przesuwamy tak długo w prawo, dopóki pomiędzy *początkiem* a *końcem* nie będzie dokładnie k dziewczynek. Następnie *koniec* przesuwamy o jedno miejsce w prawo i ponownie *początek* przesuwamy, dopóki nie będzie k dziewczynek. Przypomina więc to ruch gąsienicy.

Przed każdym przesunięciem *końca* musimy obliczyć liczbę chłopców, którzy muszą usunąć się z szeregu i wybrać z tych wartości minimum. Liczbę chłopców możemy w prosty sposób policzyć licząc różnicę: $poczatek - koniec + 1 - k$, gdyż wiemy, że pomiędzy *początkiem* a *końcem* znajduje się dokładnie k dziewczynek, więc resztę muszą stanowić chłopcy.

```
1  wczytaj(n, k, t[])
2  poczatek := 1
3  koniec := 1
4  wynik := n + 1
5  if t[1] == 0 then
6     dziewczynki := 1
7  else
8     dziewczynki := 0
9  while koniec < n do
10    while dziewczynki < k and poczatek < n do
11       poczatek := poczatek + 1
12       if t[poczatek] == 0 then dziewczynki := dziewczynki + 1
13    if dziewczynki == k then wynik := min(wynik, poczatek - koniec + 1)
14    if t[koniec] == 0 then dziewczynki := dziewczynki - 1
15    koniec := koniec + 1
16 if wynik == n + 1 then
17    wypisz(NIE)
18 else
19    wypisz(wynik - k)
```

Złożoność czasowa rozwiązania wynosi $O(n)$.

Rozdział 4

Wykłady

4.1 Złożoność czasowa

Wykład: **Złożoność czasowa** Autor wykładu: **Jacek Tomasiewicz**

4.1.1 Wstęp

Złożoność czasowa to jeden z najważniejszych parametrów charakteryzujących algorytm. Decyduje on o efektywności całego programu.

4.1.2 Czym jest złożoność

Złożoność czasowa programu jest to maksymalna liczba operacji, jakie może wykonać program w zależności od wczytanych danych.

Badając złożoność zawsze wybieramy operację *dominującą* – czyli taką, która będzie wykonywać się najwięcej razy w zależności od wczytanej wartości n.
Spójrzmy na przykład:

```
1 wynik := 0
2 cin >> n
3 for i := 1 to n do
4     wynik := wynik + i
5 cout << n
```

Znajdźmy operację dominującą – jest to z pewnością operacja w linijce 4.:

```
1 wynik := wynik + i
```

Zauważmy, że wykona się ona n razy. I już mamy policzoną złożoność czasową. Wynosi ona $O(n)$ i nazywamy ją *liniową* złożonością.

Jeśli linijka 3. wyglądałaby następująco:

```
1 for i := 1 to 20*n do
```

to liczba operacji wynosi $20n$, jednak wszystkie stałe pomijamy i złożoność pozostaje liniowa. Powiemy więc, że algorytm, który wykonuje kn operacji, gdzie k jest stałą (np. $\frac{1}{2}$, 20, 100) i algorytm który wykonuje n operacji, mają taką samą złożoność czasową.

Tak więc, analizując program będziemy badać, jaka jest największa liczba operacji, które może on wykonać przy określonym rozmiarze danych. Często będziemy wyszukiwać specjalne, „złośliwe" przykłady danych, na których dany program będzie działał długo. Czemu tak należy robić? Po pierwsze, dzięki temu wiemy, że program będzie działał nie wolniej niż ..., a jak się skończy szybciej, to tylko niespodzianka na plus. Po drugie zaś, okazuje się, że pesymistyczne ograniczenie bardzo często jest osiągane – na przykład większość algorytmów wyszukiwania elementu w zbiorze działa najdłużej, gdy szukanego elementu nie ma, co jest w praktyce bardzo częste.

4.1.3 Przykłady różnych złożoności

Złożoność stała

```
1 cin >> n
2 cout << n * n
```

Program wykona zawsze stałą liczbę operacji, niezależnie od wartości n. Złożoność jest więc *stała* i zapisujemy ją jako $O(1)$.

Złożoność logarytmiczna

```
1 cin >> n
2 wynik := 0
3 while (n > 1) do
4     n := n / 2
5     wynik := wynik + 1
6 cout << wynik
```

W linijce 4. wartość n jest w każdym obrocie pętli zmniejszana o połowę. Liczbę takich operacji oblicza się używając logarytmów. Jeśli $n = 2^x$ to $\log n = x$. Łatwo więc zauważyć, że złożoność tego rozwiązania wynosić będzie $O(\log n)$. Nazywać będziemy ją złożonością *logarytmiczną*.

Złożoność liniowa

```
1 cin >> n
2 for i := 1 to n do
3     cin >> x
4     if x = 0 then
5         break
```

Zauważmy, że jeśli pierwszą wczytaną liczbą x byłoby 0, to program od razu by się zakończył, my jednak szukamy „złośliwych" przypadków – będzie to taki, w którym x równy 0 nigdy nie występuje. Złożoność jest więc *liniowa*, czyli $O(n)$.

Złożoność liniowo-logarytmiczna

```
1 cin >> n
2 for i := 1 to n do
3     cin >> p              {p > 0}
4     while (p < n)
5         p := p * 2
6     cout << p
```

Znów znajdźmy „złośliwy" przypadek. Będzie on wtedy, gdy $p = 1$. Wówczas w każdym obrocie pętli z linijki 2. instrukcja w linijce 5. wykonywać się będzie $\log n$ razy. Stąd cała złożoność wynosi $O(n \log n)$ i nazywamy ją złożonością *liniowo-logarytmiczną*.

Złożoność kwadratowa

```
1 suma_i := suma_j := 0
2 cin >> n
3 for i := 1 to n do
4     for j := i to n do
5         suma_i := suma_i + i
6         suma_j := suma_j + j
7 cout << suma_i + suma_j
```

Zauważmy, że operacja w linijce 5. będzie wykonywana nie rzadziej niż operacja w każdej innej linijce. Policzmy liczbę wykonań tej operacji.

Pętla w linijce 4. będzie się wykonywała n, $n-1$, ... 1 razy. Po zsumowaniu liczba operacji wynosić będzie $\frac{n*(n+1)}{2} = \frac{1}{2} * (n^2 + n) \leqslant n^2$, więc złożoność to $O(n^2)$ i nazywamy ją *kwadratową*.

Złożoność wykładnicza

```
1 cin >> n
2 p := 1
3 for i := 1 to n
4     p := p * 2
5 for i := 1 to p
6     wynik := wynik + i
7 cout << wynik
```

Ile operacji wykona powyższy kod. Pętla w linijce 3. obróci się n razy, jednak pętla w linijce 6. obróci się $p = 2^n$ razy. Wybieramy operację dominującą, więc złożoność wynosić będzie $O(2^n)$, która nazywana jest złożonością *wykładniczą*. Złożonościami wykładniczymi są również $O(3^n)$, $O(10^n)$, a także $O(n!)$.

Wiele zmiennych

Nie zawsze złożoność musi zależeć od jednej zmiennej:

```
1 cin >> n >> m
2 for i := 1 to n do
3     cout << i
4 for j := 1 to m do
5     cout << j
```

Zauważmy, że czas działania programu zależy od obu wartości n i m, w związku z tym złożoność wynosi $O(n + m)$. Mówimy, że złożoność jest *liniowa* względem n i m.

Spójrzmy na inny przykład:

```
1 cin >> n >> m
2 if n > m then
3     for i := 1 to n do
4         cout << i
5 else
6     for j := 1 to m do
7         cout << j
```

Złożoność zależy od wartości większej z liczb n i m. Wynosi więc ona $O(max(n, m))$.

I jeszcze jeden:

```
1 cin >> n
2 cin >> m
3 wynik := 0
4 for i := 1 to n do
5     for j := 1 to m do
6         wynik := wynik + i * j
7 cout << wynik
```

Złożoność zależy od obu wartości n i m. W każdym z n obrotów pętli w linii 4. wykonuje się pętla z linii 5. obracająca m razy. Stąd złożoność wynosi $O(nm)$.

4.1.4 Czy program ma wystraczającą złożoność

Podczas pisania programu zawsze warto zastanowić się, czy istnieje rozwiązanie o lepszej złożoności czasowej. Chyba że limity wyraźnie wskazują, że wymyślone rozwiązanie, choć może nieoptymalne, zostanie w pełni zaakceptowane.

Porównajmy 3 programy obliczające dokładnie to samo:

Program A

```
1 cin >> n
2 for i := 1 to n do
3     for j := 1 to i do
4         wynik := wynik + 1
5 cout << wynik
```

Złożoność: $O(n^2)$

Program B

```
1 cin >> n
2 for i := 1 to n do
3     wynik := wynik + i
4 cout << wynik
```

Złożoność: $O(n)$

Program C

```
1 cin >> n
2 wynik := n * (n + 1) / 2
3 cout << wynik
```

Złożoność: $O(1)$

Chcielibyśmy stwierdzić, który program będzie akceptowalny w czasie zawodów. Obecnie można przyjąć, że aby program był akceptowalny w pełni przez sprawdzarkę, to może wykonywać około 10^8 operacji – tyle operacji jest wykonywane przez mniej więcej 1 sekundę.

Jeśli mamy zadeklarowane, że $1 \leqslant n \leqslant 10^4$, to wszystkie programy A, B, C powinny zostać zaakceptowane.

Jeśli $1 \leqslant n \leqslant 10^6$, to program A wykona około 10^{12} operacji, co niestety wywoła przekroczenie limitu czasu. Natomiast program B i C powinien zostać zaakceptowany.

Jeśli $1 \leqslant n \leqslant 10^{10}$, to tylko program C może zostać zaakceptowany.

Warto zwrócić uwagę, że wczytywanie i wypisywanie danych jest szczególnie wolne. Wczytanie lub wypisanie więcej niż 10^6 liczb może wykonywać się ponad 1 sekundę (w zadaniach raczej nie zdarza się, aby potrzebne było wczytanie istotnie wielu liczb)

4.1.5 Podsumowanie

1. Jeśli $n \leqslant 10^6$, to próbujmy pisać programy o maksymalnej złożoności $O(n)$ lub $O(n \log n)$.
2. Jeśli $n \leqslant 10^4$, to próbujmy pisać programy o maksymalnej złożoności $O(n^2)$
3. Jeśli $n \leqslant 500$, to próbujmy pisać programy o maksymalnej złożoności $O(n^3)$

Oczywiście nie są to ścisłe limity, choć z pewnością przybliżone. Często zależą one od konkretnego zadania. Na pewno jeśli stwierdzimy, że program wykonuje dużo ponad 10^8 operacji to powinniśmy się zastanowić, czy nie istnieje szybsze rozwiązanie naszego problemu.

4.2 Praktyczne zastosowanie STL, część I

Wykład: **Praktyczne zastosowanie STL, cześć I** Autor wykładu: **Robert Kozikowski**

4.2.1 Wstęp

Wykład jest celowany do Czytelnika z bardzo podstawową wiedzą o C++. W części I omówione są najbardziej podstawowe i najprostsze algorytmy i struktury. Wykład w założeniu jest bardziej praktyczny niż teoretyczny. Staramy się używać niefachowego słownictwa, aby był bardziej zrozumiały dla osób z podstawową wiedzą, czasem celowo pomijamy szczegóły, gdyż uważamy, że do praktycznych zastosowań nie są one konieczne. W części II są omówione bardziej zaawansowane struktury, ale obie części są stworzone w celu pokazania STL-a Czytelnikowi, który wcześniej go nie znał.

4.2.2 Czym jest STL

STL to skrót od Standard Template Library, czyli po polsku Standardowa Biblioteka Szablonów. STL to zbiór bibliotek w C++, w których są szablony różnych algorytmów i struktur danych, które działają niezależnie od tego, jaki typ zawierają. Za pomocą STL-a możemy wyszukiwać binarnie, sortować elementy w tablicy, tworzyć kolejki priorytetowe w jednej linii tekstu. Dokumentacja STL jest na stronie:

`http://www.sgi.com/tech/stl/`

Szukając dodatkowych informacji lub przypominając materiał z wykładu dobrze jest korzystać w pierwszej kolejności z dokumentacji STL-a, gdyż jego kopia znajduje się na każdym komputerze podczas zawodów Olimpiady Informatycznej (i na wielu innych konkursach), a ten skrypt Czytelnik posiada tylko w domu.

4.2.3 Wady STL

Mimo wielu praktycznych zastosowań i dużej użyteczności, STL posiada szereg wad:

1. Niektóre rzeczy działają wolniej w STL niż zaimplementowane samemu (np. stos albo generowanie wszystkich permutacji.
2. Posiadając gotowe, zaimplementowane niektóre struktury czasem, zamiast pomyśleć chwilę nad szybszym rozwiązaniem, używamy niepotrzebnie STL-a. Często, gdy rozwiązanie wzorcowe jest liniowe, to rozwiązanie z czynnikiem $\log n$ wykorzystujące STL-a, może dostać nawet o 1/3 punktów mniej.
3. Błędy w wykorzystaniu STL-a często trudno się debuguje, gdyż informacje o błędach w STL-u są zazwyczaj skomplikowane i nie zawsze wynika z nich wprost, gdzie leży błąd.

4.2.4 Para

pair $< T1, T2 >$ to, jak nazwa wskazuje, para dwóch elementów typów *T1*, *T2*, które mogą być dowolnymi typami, w szczególności nawet kolejną parą. Pary często wykorzystuje się w geometrii obliczeniowej jako reprezentację punktu lub w grafie ważonym jako pojedyncza krawędź.

Użycie

```
pair<int,int> p = make_pair(2,3);  // utworzenie pary
p.first;                           // pierwszy element pary
p.second;                          // drugi element pary
```

Uwagi

Jeśli typy się nie zgadzają, to program się nie skompiluje, przykład:

```
pair<int,int> p = make_pair(2,make_pair(2,3)); // źle!
```

Kompilator próbował zrobić z obiektu *pair* $< int, int >$ obiekt typu *int*, czego nie potrafi, więc się nie skompilował.

Uwaga co do deklarowania typów złożonych:
Nie możemy dać dwóch znaków << lub >> bez spacji obok siebie, przykład:

```
pair<int,pair<int,int>> p1; // źle!
pair<int,pair<int,int> >p2; // dobrze
```

Przykład programu

Przypuśćmy, że chcemy w pewien sposób porównać dwa pairy ze sobą. Oczywiście możemy porównać najpierw pierwszy element paira, a później drugi, ale pair dostarcza prostszych sposobów:

```
1 pair<int,int> p = make_pair(2,3);
2 pair<int,int> q = make_pair(2,4);
3 pair<int,int> r = make_pair(2,3);
4 pair<int,int> s = make_pair(3,3);
5 if(p==r && r<q && q<s)
6     cout << "daj kamienia" << endl;
```

Następujący kod wypisze na ekran „daj kamienia", zatem mogliśmy porównać pairy ze sobą. Działa to w następujący sposób:
$p == q$ jest równoważne z $p.first == q.first \&\& p.second == q.second$
$p < q$ jest równoważne z $(p.first < q.first) || (p.first == q.first \&\& p.second < q.second)$

4.2.5 Vector, podstawy

vector $< T >$ w korzystaniu bardzo przypomina standardową tablicę. T może być dowolnym typem, nawet kolejnym vektorem (wykorzystuje się to często w implementacji grafów, gdzie dla każdego wierzchołka mamy wektor krawędzi). Główną różnicą jest to, że *vector* $< T >$ może zmieniać swoją wielkość podczas działania programu.

Użycie

Podstawowe operacje na vectorach:

```
#include <vector>  // umożliwia używanie vectorów
vector<int> V;     // deklaracja pustego vectora
V.push_back(x);    // dodanie nowego elementy o wartości x na koniec vectora V
V.pop_back();      // usunięcie ostatniego elementu z vectora V
V[0];              // odwołanie do pierwszego elemntu w vectorze V
V.size();          // liczba elementów w vectorze V
```

Często przydaje się wypisanie wszystkich elementów vectora:

```
1 for(int i=0; i<V.size(); i++)
2     cout << V[i] << ' ';
```

Kolejną ważną funkcją z vectorów jest funkcja resize.

```
V.resize(x);
```

Powyższy fragment zmienia vector w taki sposób, że jeśli vector ma wielkość większą niż x, to usuwa ostatnie elementy, a jeśli mniejszą niż x, to dodaje na końcu 0 tak, aby nowy vector miał wielkość x. Jeśli chcemy, żeby uzupełnił innymi elementami niż 0, możemy napisać:

```
V.resize(x,1);
```

Kilka przydatnych funkcji:

```
V.empty();   // jest równoważne z (V.size()==0)
V.clear();   // jest równoważne z V.resize(0)
V.back();    // jest równoważne z V[V.size()-1]
V.front();   // jest równoważne z V[0]
```

Przekazywanie vectora do funkcji

W ten sposób możemy przekazać vector do funkcji.

```
1 void wypisz(vector<int> V) {
2     for(int i=0; i<V.size(); i++)
3         cout << V[i] << ' ';
4 }
```

Funkcja tworzy kopię vectora, więc jakakolwiek zmiana vectora nie zajdzie na vectorze przekazanym do funkcji.

Jeśli chcemy w jakiś sposób zmieniać vector, to musimy przed nazwą dodać znak & (przekazać przez referencję).

```
1 void zmien(vector<int>& V) {...}
```

Przykład programu

Weźmy teraz kawałek kodu, który zawiera wszystko, o czym mówiliśmy:

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  void wypisz(vector<int> V) {
6      for(int i=0; i<V.size(); i++)
7          cout << V[i] << ' ';
8      cout << endl;
9  }
10 void zmien(vector<int>& V) {
11     for(int i=0; i<V.size(); i++)
12         V[i]++;
13 }
14 int main() {
15      vector<int> V;
16      V.resize(5, 1);
17      wypisz(V);
18      zmien(V);
19      wypisz(V);
20      V.push_back(5);
21      wypisz(V);
22      V.pop_back();
23      wypisz(V);
24      return 0;
25 }
```

Powyższy kod wypisze na ekran:

```
1 1 1 1 1
2 2 2 2 2
2 2 2 2 2 5
2 2 2 2 2
```

4.2.6 Kolejka

queue $< T >$ to STL-owa implementacja kolejki. W kolejce możemy dokładać elementy na koniec i brać z początku w czasie stałym, czyli, innymi słowy, niezależnie od wielkości kolejki dodanie na koniec/zabranie z początku trwa tyle samo.

Użycie

Podstawowe operacje na kolejce:

```
#include <queue>     // umożliwia korzystanie z kolejki
queue<int> Q;        // deklaracja kolejki
Q.push(x);           // dodanie na koniec kolejki elementu x
Q.front();           // zwraca przód kolejki
if(Q.empty()) {...} // sprawdza czy kolejka jest pusta
Q.pop();             // zabiera element z przodu kolejki
```

Wszystkie operacje wykonują się w czasie stałym.

Uwagi

Jeśli kolejka jest pusta, to nie można zabierać elementu z przodu kolejki – przy próbie wystąpi naruszenie ochrony pamięci.

Przykład programu

Program, który dodaje do kolejki dziesięć kolejnych liczb, a następnie zabiera je z kolejki i wypisuje.

```
1  #include <iostream>
2  #include <queue>
3  using namespace std;
4  queue<int> Q;
5
6  int main() {
7      for(int i=0; i<10; i++)
8          Q.push(i);
9      while(!Q.empty()) {
10         cout << Q.front() << ' ';
11         Q.pop();
12     }
13     cout << endl;
14     return 0;
15 }
```

wypisze nam na ekran:

```
0 1 2 3 4 5 6 7 8 9
```

4.2.7 Kolejka priorytetowa

priority_queue $< T >$ to STL-owa implementacja kopca, czyli struktury, w której możemy dodawać elementy do kopca w czasie $O(\log n)$, zabierać elementy z kopca w czasie $O(\log n)$ i uzyskiwać aktualnie maksymalny element kopca w $O(1)$.

Podstawowe zastosowanie jest prawie identyczne jak zastosowanie kolejki.

Użycie

Podstawowe operacje na kolejce priorytetowej:

```
#include <queue>       // umożliwia używanie kolejki priorytetowej
priority_queue<int> Q; // deklaracja kolejki priorytetowej
Q.top();               // uzyskujemy maksymalny element
Q.push(x);             // dodajemy do kolejki element x
Q.pop();               // zabieramy największy element z kolejki
if(Q.empty())          // sprawdzamy czy kolejka jest pusta
```

Jeśli chcemy, aby na szczycie kolejki były elementy minimalne, musimy zadeklarować kolejkę w następujący sposób (zamiast int możemy wstawić dowolny typ porównywalny za pomocą $<$, np. $pair < int, int >$. Więcej o tym i wyjaśnienie w części II):

```
priority_queue<int, vector<int>, greater<int> > Q;
```

Innym sposobem, aby na szczycie kolejki były maksymalne elementy, jest pomnożenie każdego elementu przez -1 podczas włożenia oraz pomnożenie przez -1 podczas wyjmowania.

Przykład programu

```
1  #include <iostream>
2  #include <queue>
3  using namespace std;
4  queue<int> Q;
5
6  int main() {
7      priority_queue<int> Q1;
8      priority_queue<int, vector<int>, greater<int> > Q2;
9      for(int i=0; i<4; i++) {
10         Q1.push(i);
11         Q2.push(i);
12     }
13     while(!Q1.empty()) { // co jest jednoznaczne z Q.empty() == false
14         cout << Q1.top() << ' ' << Q2.top() << endl;
15         Q1.pop();
16         Q2.pop();
17     }
18     return 0;
19 }
```

Wypisze na ekran:

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

4.2.8 Podstawowe algorytmy i ich wykorzystanie

STL poza szablonami struktur danych dostarcza nam także różnych algorytmów. Większość z nich działa na tablicach/vectorach. Aby korzystać z większości poniższych algorytmów, należy dodać linię na początku programu:

```
#include <algorithm>
```

Minimum, maksimum

Zacznijmy od najprostszych:
min(a,b) - znajdowanie minimum z dwóch liczb
max(a,b) - znajdowanie maksimum z dówch liczb.

```
1. int a=2, b=3;
2. cout << max(a,b) << ' ';
3. cout << min(a,b);
```

Powyższy kod wypisze:

```
3 2
```

Jedna uwaga do używania pairów: dwa obiekty muszą być identycznego typu, nie może być takiej sytuacji, że jeden jest rzutowalny na drugi (jak to było w przypadku pairów). Przykład:

```
cout << max(2,'a');       // źle
cout << max(2,int('a'));  // dobrze
```

Sortowanie

Weźmy teraz najczęściej wykorzystywany algorytm, który dodatkowo nie jest łatwo zastampialny przez własny kod – *sort*. Działa on w czasie $O(n \log n)$. Warto powiedzieć, że własna implementacja funkcji sortującej, która byłaby szybsza bądź porównywalna w szybkości do *sorta* (w szczególności na dłuższych tablicach), jest bardzo pracochłonna i na konkursach prawie zawsze wystarczy szybkość, w jakiej działa *sort* (jeśli nie, to da się posortować dane w czasie $O(n)$). Podstawowe użycie jest bardzo proste:

```
sort(V.begin(), V.end());  // posortowanie vectora v
sort(t, t + n);            // posortowanie tablicy t o liczbie elementów n;
```

Jeśli chcemy posortować pierwsze x elementów tablicy t, to wpisujemy:

```
sort(t, t + x);
```

Sortować możemy tylko te tablice/vectory elementów, dla których elementy są porównywalne za pomocą operatora $<$. Czyli możemy sortować leksykograficznie tablice stringów, tablice intów, tablice charów, tablice pairów (dla których poszczególne elementy są też porównywalne za pomocą $<$). Własnych structów sortować już tak prosto nie możemy (więcej w części II).

Permutacje

Kolejnymi często wykorzystywanymi algorytmami są: next_permutation i prev_permutation.

```
next_permutation(V.begin(), V.end()); // następna permutacja
prev_permutation(t, t + n);           // poprzednia permutacja
```

Powyższy kod generuje kolejną/poprzednią leksykograficznie permutację dla vectora/tablicy. Permutacja to dowolne przestawienie elementów tablicy. Kolejna leksykograficznie permutacja to taka permutacja, że jeśli posortujemy leksykograficznie wszystkie permutacje, to będzie znajdować się ona indeks dalej (lub na początku, jeśli nasza permutacja jest ostatnia). next_permutation zwraca wartość bool (true/false) zależnie od tego, czy otrzymana permutacja jest posortowana (pierwsza leksykograficznie). Pozwala to bardzo łatwo generować wszystkie permutacje dowolnego vectora.

Przykład:

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  using namespace std;
5  int main() {
6      vector<int> v;
7      for(int i=2; i>=0; i--) {
8          v.push_back(i);
9      }
10     v.push_back(2);
11     sort(v.begin(), v.end());
12     do {
13         for(int i=0; i<v.size(); i++)
14             cout << v[i] << ' ';
15         cout << endl;
16     } while(next_permutation(v.begin(), v.end()));
17 }
```

Otrzymamy następujący wynik:

```
0 1 2 2
0 2 1 2
0 2 2 1
1 0 2 2
1 2 0 2
1 2 2 0
2 0 1 2
2 0 2 1
2 1 0 2
2 1 2 0
2 2 0 1
2 2 1 0
```

Losowe permutowanie

Weźmy teraz kolejny algorytm - random_shuffle. Wykorzystanie jest takie same jak sorta, tyle że elementy nie muszą być porównywalne. random_shuffle losowo permutuje elementy vectora. Przykład:

```
random_shuffle(V.begin(), V.end());
```

Część II

Grupa średnio zaawansowana

Rozdział 5

Zadania

5.1 Dzień próbny

Zadanie: **Kostki domina** Autor zadania: **Joanna Bujnowska**

Dostępna pamięć: **32 MB** *Dzień 0, 19.09.2010*

Kostki domina

Adrian układa ciągi z kostek domina. Kostki podzielone są na 2 pola z pewną liczbą oczek. Kostki można obracać. Kostki dokładane są z obydwu stron tak, aby stykające się pola miały taką samą liczbę oczek. Adrian ma n kostek. Wykorzystując wszystkie kostki da się ułożyć poprawny ciąg. Adrian zastanawia się, ile oczek będzie z początku (p) i końca (k) ciągu kostek.

Wejście

W pierwszym wierszu wejścia znajduje się liczba n ($1 \leqslant n \leqslant 10^6$), oznaczająca liczbę kostek domina. W n kolejnych wierszach po 2 liczby całkowite a_i, b_i ($1 \leqslant a_i, b_i \leqslant 10^9$), oznaczające odpowiednio liczbę oczek po lewej stronie oraz po prawej stronie i-tej kostki.

Wyjście

W pierwszym i jedynym wierszu wyjścia liczby p i k, gdy $p < k$, w przeciwnym wypadku k i p. Jeśli wynik jest niejednoznaczny odpowiedzią powinno być słowo 'NIE'.

Przykład

dla danych wejściowych:

```
2
3 2
3 4
```

poprawnym wynikiem jest:

```
2 4
```

5.2 Dzień pierwszy

Zadanie: **Sznurki** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 1, 20.09.2010*

Sznurki

Jaś dostał od dziadka n sznurków, które ułożył w linii prostej jeden obok drugiego. Jaś zauważył, że bez problemu może połączyć dwa sąsiednie sznurki w jeden. Połączony sznurek ma długość równą sumie długości obydwu sznurków. Połączony sznurek Jaś może ponownie połączyć z sąsiednim sznurkiem.

Jaś chciałby mieć jak najwięcej sznurków, jednak takich, aby każdy z nich nie był krótszy od jego wzrostu.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite n, w ($1 \leqslant n, w \leqslant 10^6$), oznaczające odpowiednio liczbę sznurków oraz wzrost Jasia. Kolejny wiersz zawiera ciąg n liczb całkowitych $a_1, a_2, ...a_n$ ($1 \leqslant a_i \leqslant 10^6$), oznaczających kolejne sznurki ułożone od lewej do prawej.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą, równą maksymalnej liczbie sznurków jakie może utworzyć Jasio.

Przykład

dla danych wejściowych:

```
8 4
1 2 3 4 8 1 1 3
```

poprawnym wynikiem jest:

```
4
```

Zadanie: **Wieża** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 1, 20.09.2010*

Wieża

W Bajtocji wybudowano wysoką wieżę. Wejście na wieżę składa się z n schodków, a każdy schodek ma pewną wysokość.

Bajtocką wieżę chce odwiedzić m mieszkańców. Każda z osób posiada pewien wzrost, który pomaga w pokonywaniu kolejnych schodków. Aby mieszkaniec Bajtocji mógł wejść na pewien schodek, to musi być wyższy od wysokości schodka. Jeśli pewien schodek jest nie do przejścia przez mieszkańca, to zatrzymuje się on w danym miejscu na wieży – wyżej nie będzie mógł wejść.

Znając wysokości kolejnych schodków i osób zwiedzających wieżę chcielibyśmy wiedzieć, w którym miejscu zatrzyma się każdy mieszkaniec Bajtocji.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite n, m ($1 \leqslant n, m \leqslant 500000$), oznaczające odpowiednio liczbę schodków prowadzących na wieżę oraz liczbę mieszkańców chcących odwiedzić wieżę. Kolejny wiersz zawiera n liczb całkowitych $a_1, a_2, ..., a_n$ ($1 \leqslant a_i \leqslant 10^9$) , gdzie a_i oznacza wysokość i-tego schodka. Pierwszy schodek znajduje się na samym dole wieży, a każdy kolejny wyżej od poprzednich. Następny wiersz wejścia zawiera m liczb całkowitych $b_1, b_2, ..., b_m$ ($1 \leqslant b_i \leqslant 10^9$), gdzie b_i oznacza wzrost i-tego mieszkańca.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać m liczb całkowitych $w_1, w_2, ..., w_m$, gdzie w_i oznacza maksymalny numer schodka, na który może wejść i-ty mieszkaniec Bajocji.

Przykład

dla danych wejściowych:

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

poprawnym wynikiem jest:

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

Zadanie: **Tablica** Autor zadania: **Łukasz Jocz**

Dostępna pamięć: **64 MB** *Dzień 1, 20.09.2010*

Tablica

Kacper i Adi bardzo polubili system dwójkowy. Każdy z nich napisał na tablicy ciąg zer i jedynek. Kacper chciałby teraz w każdym z tych ciągów skreślić niektóre cyfry tak, żeby pozostałe ciągi były takie same oraz były uporządkowane, tj. po pierwszym wystąpieniu jedynki nie może już wystąpić żadne zero. Jaki najdłuższy ciąg może pozostać na tablicy?

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite n, m ($1 \leqslant n, m \leqslant 10^6$), oznaczające długość ciągów napisanych przez kolejno Kacpra i Adiego. W drugim wierszu znajduje się n cyfr 0 lub 1 – ciąg napisany przez Kacpra. W trzecim wierszu znajduje się m cyfr 0 lub 1 – ciąg napisany przez Adiego.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę – długość najdłuższego ciągu który może pozostać na tablicy.

Przykład

dla danych wejściowych:

```
6 6
0 0 1 1 0 1
0 1 0 0 1 1
```

poprawnym wynikiem jest:

```
4
```

5.3 Dzień drugi

Zadanie: **Obwody** Autor zadania: **Joachim Jelisiejew**

Dostępna pamięć: **32 MB** *Dzień 2, 21.09.2010*

Obwody

Jaś na urodziny dostał komplet magicznych elektrycznych kabelków. Każdy kabelek składa się z drutu oraz bateryjki. i-ty drut może wytrzymać napięcie d_i woltów, a i-ta bateryjka ma napięcie b_i woltów.

Jasio buduje z kabelków obwody: wybiera druty, skręca je razem tworząc grubszy drut i robi z niego kółko. W kółku napięcie jest sumą napięć wszystkich bateryjek, a skręcony drut może wytrzymać napięcie będące sumą napięć, które mogą wytrzymać poszczególne druty.

Jaś buduje obwód tak, aby zrobione kółko nie przepaliło się. Z ilu maksymalnie kabelków może się ono składać?

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 500000$), oznaczająca liczbę drutów. W n następnych linijkach pary liczb d_i, b_i opisujące kolejne druty ($1 \leqslant d_i, b_i \leqslant 10^6$).

Wyjście

W jedynej linijce powinna znaleźć się liczba całkowita, oznaczająca maksymalną liczbę drutów tworzących kółko.

Przykład

dla danych wejściowych:

```
2
2 2
2 2
```

poprawnym wynikiem jest:

```
2
```

Zadanie: **Wieża 2** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 2, 21.09.2010*

Wieża 2

W Bajtocji wybudowano wysoką wieżę. Wejście na wieżę składa się z n schodków, a każdy schodek ma pewną wysokość.

Bajtocką wieżę chce odwiedzić m mieszkańców. Każda z osób posiada pewien wzrost, który pomaga w pokonywaniu kolejnych schodków. Aby mieszkaniec Bajtocji mógł wejść na pewien schodek, to musi być wyższy od wysokości schodka. Jeśli pewien schodek jest nie do przejścia przez mieszkańca, to zatrzymuje się on w danym miejscu na wieży – wyżej nie będzie mógł wejść. Niestety schody na wieżę są bardzo wąskie i jeśli pewna osoba zatrzyma się, to każda kolejna nie będzie mogła już wejść wyżej – będzie musiała zatrzymać się schodek wcześniej.

Znając wysokości kolejnych schodków i wzrost kolejnych osób zwiedzających wieżę, chcielibyśmy wiedzieć, w którym miejscu zatrzyma się każdy mieszkaniec Bajtocji.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite n, m ($1 \leqslant n, m \leqslant 500000$), oznaczające odpowiednio liczbę schodków prowadzących na wieżę oraz liczbę mieszkańców chcących odwiedzić wieżę. Kolejny wiersz zawiera n liczb całkowitych $a_1, a_2, ..., a_n$ ($1 \leqslant a_i \leqslant 10^9$) , gdzie a_i oznacza wysokość i–tego schodka. Pierwszy schodek znajduje się na samym dole wieży, a każdy kolejny wyżej od poprzednich. Następny wiersz wejścia zawiera m liczb całkowitych $b_1, b_2, ..., b_m$ ($1 \leqslant b_i \leqslant 10^9$), gdzie b_i oznacza wzrost i-tego mieszkańca.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać m liczb całkowitych $w_1, w_2, ..., w_m$, gdzie w_i oznacza maksymalny numer schodka, na który może wejść i-ty mieszkaniec Bajocji.

Przykład

dla danych wejściowych:

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

poprawnym wynikiem jest:

```
3 1 0 0
```

Zadanie: **Akwarium** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **64 MB** *Dzień 2, 21.09.2010*

Akwarium

Kozik dostał w prezencie na urodziny akwarium z mieczykami. Obserwowanie mieczyków szybko mu się znudziło, ponieważ według Kozika nigdy nie działo się nic nowego. Dlatego też mieczyki szybko sprzedał, a za zarobione pieniądze kupił w promocji pewne afrykańskie ryby.

Afrykańska ryba ma określoną masę i wiek oraz charakteryzuje się tym, że musi codziennie zaspokoić swój głód. Jeśli nie dostanie nic do jedzenia, to pożera inną, mniejszą rybę. Jeśli kilka ryb jest głodnych, to pierwszeństwo ma zawsze ryba o największej masie, która wybiera swoją ofiarę (w przypadku równych mas pierwszeństwo ma ryba starsza). Ofiarą będzie zawsze najmniejsza ryba znajdująca się w akwarium (jeśli kilka ryb ma taką samą masę, ofiarą z nich będzie najmłodsza ryba). Dodatkowo afrykańska ryba zwiększa swoją masę o połowę masy zjedzonej przez nią innej ryby. Jeśli afrykańska ryba nie zaspokoi swojego głodu w ciągu dnia, zdycha. Kilka ryb może mieć równe masy, natomiast wszystkie ryby są innego wieku.

Kozik zastanawia się, czy pewna ryba r, wybrana przez niego, będzie żyła za x dni.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 10^6$), oznaczającą liczbę ryb w akwarium. W n kolejnych wierszach znajdują się po dwie liczby całkowite m_i, w_i ($1 \leqslant m_i, w_i \leqslant 10^9$), oznaczające odpowiednio masę i wiek i – tej ryby. W kolejnym wierszu znajduje się jedna całkowita z ($1 \leqslant z \leqslant 10^6$), ozaczająca liczbę zapytań Kozika. W z następnych wierszach znajdują się po dwie liczby całkowite r_k, x_k ($1 \leqslant r_k \leqslant n, 0 \leqslant x_k \leqslant 10^9$), oznaczające pytanie: czy ryba r_k będzie żyła za x_k dni?.

Wyjście

Dla każdego zapytania słowo 'TAK', jeśli ryba r_k będzie żyła za x_k dni, w przeciwnym wypadku słowo 'NIE'.

Przykład

dla danych wejściowych:

```
4
2 1
4 2
7 3
8 4
2
2 1
3 1
```

poprawnym wynikiem jest:

```
NIE
TAK
```

5.4 Dzień trzeci

Zadanie: **Tarasy** Autor zadania: **PA 2002**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Tarasy

W bajtockich górach wybudowano tarasy widokowe połączone za pomocą wind. Z tarasu położonego niżej można wjechać na taras sąsiedni, położony wyżej, za tyle kredytek ile wynosi różnica pomiędzy wysokościami tarasów. Z tarasu położonego wyżej na taras położony niżej zjeżdża się za darmo. Tarasy połączone są w łańcuch widokowy, w którym z pierwszego tarasu można dostać się tylko na drugi, z drugiego na pierwszy i trzeci, itd.

Policz, jaka jest największa liczba różnych tarasów, które bezpośrednio (czyli bez zjeżdżania z tarasu na ziemię) może odwiedzić turysta posiadający tylko k kredytek. Za wjazd na taras, od którego zacznie swoją wędrówkę, turysta nic nie płaci.

Wejście

W pierwszym wierszu podane są dwie liczby całkowite n, k ($1 \leqslant n \leqslant 10^5, 0 \leqslant k \leqslant 10^5$), oddzielone pojedynczym odstępem. Liczba tarasów to n, a k to liczba kredytek, którymi dysponuje turysta. W kolejnych n wierszach podane są wysokości kolejnych tarasów: $h_1, h_2, ..., h_n$. Każde h_i spełnia nierówności: $1 \leqslant h_i \leqslant 10^4$.

Wyjście

Program powinien wypisać tylko jedną liczbę, równą największej liczbie tarasów, które może odwiedzić turysta za k kredytek.

Przykład

dla danych wejściowych:

```
5 1
4
2
1
2
4
```

poprawnym wynikiem jest:

```
4
```

Zadanie: **Rolki papieru** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Rolki papieru

Ostatnio Kozik, będąc na wszystkich lekcjach w szkole, był zmuszony do odwiedzenia toalety. Zdziwił się bardzo, gdyż znajdowało się tam n rolek papieru toaletowego. Zdenerwowało go jednak to, że nie wszystkie rolki wyglądały tak samo, gdyż nie były rozwinięte na te same długości. Postanowił więc poprawić wygląd toalety i rozwinąć lub zwinąć rolki tak, aby wszystkie były rozwinięte na tą samą długość.

Jeden ruch polega na zwinięciu lub rozwinięciu 1 cm rolki. Każda rolka ma określoną długość oraz rozwinięcie. Długość rozwinięcia nie może przekraczać długości rolki. Pomóż Kozikowi uporządkować rolki w jak najmniejszej liczbie ruchów.

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 10^6$), oznaczająca liczbę rolek. W n kolejnych wierszach znajduje się opis i – tej rolki w postaci dwóch liczb całkowitych d_i i r_i oznaczających odpowiednio długość i rozwinięcie i – tej rolki ($0 \leqslant r_i \leqslant d_i \leqslant 10^9$).

Wyjście

W jedynym wierszu wyjścia powinna znajdować się jedna liczba całkowita, równa minimalnej liczbie ruchów Kozika.

Przykład

dla danych wejściowych:

```
3
50 10
40 20
30 30
```

poprawnym wynikiem jest:

```
20
```

Zadanie: **Karty** Autor zadania: **Singapurska OI**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Karty

Tasujemy talię kart za pomocą następujących operacji:

- A - przełożenie pierwszej karty z góry na dół stosu
- B - przełożenie drugiej karty z góry na dół stosu

Twoim zadaniem jest obliczyć która karta będzie na górze n-elementowej talii kart po k operacjach. Karty są ponumerowane kolejno od góry liczbami od 1 do n.

Wejście

W pierwszym wierszu znajdują się dwie liczby całkowite n i k ($1 \leqslant n, k \leqslant 2000000$). W drugim wierszu znajduje się k znaków: A lub B, oznaczających rodzaj kolejnych operacji na talii kart.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczę całkowitą – numer karty na górze stosu.

Przykład

dla danych wejściowych:

```
6 6
ABBABA
```

poprawnym wynikiem jest:

```
1
```

5.5 Dzień czwarty

Zadanie: **Las** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **64 MB** *Dzień 4, 23.09.2010*

Las

Pan Jan posiada spory teren lasu obejmujący kwadratowy teren o boku n. Rozmieszczonych jest tam n^2 drzew, po n drzew w każdym wierszu i po n drzew w każdej kolumnie. Każde drzewo ma określony wiek. Pan Jan chce zbudować dom o powierzchni d, jednak w tym celu musi wyciąć pewien fragment swojego lasu (a dokładniej d drzew, ponieważ każde drzewo zajmuje 1 jednostkę powierzchni). Fragment ten musi być oczywiście spójny.

Pan Jan zastanawia się teraz, który fragment wybrać. Chciałby, aby najstarsze drzewo ze wszystkich wyciętych było możliwie najmłodsze.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n i d ($1 \leqslant d \leqslant n \leqslant 1\,000$), oznaczające odpowiednio wielkość terenu oraz powierzchnię domu który chce zbudować pan Jan. Kolejnych n wierszy zawiera po n liczb całkowitych $w(i,k)$ ($1 \leqslant w(i,k) \leqslant 10^9$), oznaczających wiek drzewa stojącego w i – tym wierszu i k – tej kolumnie.

Wyjście

Pierwszy wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, równą minimalnemu wiekowi najstarszego drzewa ze wszystkich wyciętych.

Przykład

dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2
```

Zadanie: **Domino** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 4, 23.09.2010*

Domino

Jaś układa domino. Nie robi tego jednak w tradycyjny sposób, tylko bawi się w przewracanie kolejnych klocków domina. Wiemy, że klocki Jasia są różnych wysokości.

Jaś postawił n klocków domina w ciągu w taki sposób, aby przewrócenie się dowolnego klocka spowodowało przewrócenie się kolejnego klocka w ciągu. Wiadomo, że kolejny klocek przewróci się, jeśli wysokość przewracanego klocka jest większa od odległości pomiędzy nimi.

Jaś chciałby wiedzieć, ile niepotrzebnych klocków może usunąć z ciągu, aby przewrócenie pierwszego klocka w ciągu spowodowało (poprzez przewracanie się pośrednich klocków) przewrócenie się ostatniego klocka w ciągu. Jaś nie może zmieniać położenia klocków.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 10^6$), oznaczającą liczbę klocków Jasia. Drugi wiersz wejścia zawiera ciąg n liczb całkowitych $w_1, w_2, ..., w_n$ ($1 \leqslant w_i \leqslant 10^9$), gdzie w_i oznacza wysokość i – tego klocka w ciągu. Trzeci wiersz wejścia zawiera odległości pomiędzy klockami w ciągu. Składa się z ciągu $n-1$ liczb całkowitych $x_1, x_2, ..., x_{n-1}$ ($1 \leqslant x_i \leqslant 10^9$), gdzie x_i oznacza odległość pomiędzy i – tym a $i+1$ – wszym klockiem w ciągu.

Wyjście

Pierwszy i jedyny wiesz standardowego wyjścia powinien zawierać jedną liczbę całkowitą równą maksymalnej liczbie klocków, które możemy usunąć z ciągu.

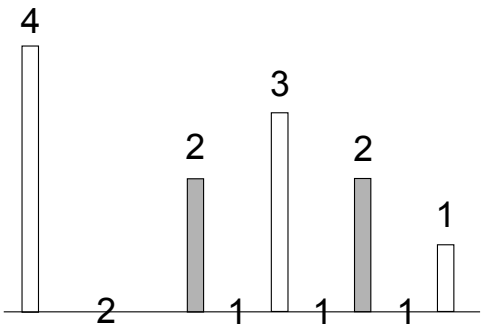
Przykład

dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2
```



Zadanie: **Suma** Autor zadania: **Łukasz Jocz**

Dostępna pamięć: **32 MB** *Dzień 4, 23.09.2010*

Suma

Od kiedy Adi przestał zajmować się ogrodnictwem, to całymi dniami siedzi w swoim pokoju analizując własności liczb w zapisie binarnym.

Od tego czasu nęka go następujący problem: znalezienie sumy wszystkich dodatnich liczb całkowitych, które w zapisie binarnym mają co najwyżej k cyfr. Adi nie ma pomysłu, jak się do tego zabrać, więc o pomoc poprosił Ciebie.

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą k ($1 \leqslant k \leqslant 10^6$).

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę – sumę dodatnich liczb całkowitych, które w zapisie binarnym mają co najwyżej k cyfr. Liczbę tę należy wypisać w systemie dwójkowym.

Przykład

dla danych wejściowych:

```
3
```

poprawnym wynikiem jest:

```
11100
```

5.6 Dzień piąty

Zadanie: **Wykres** Autor zadania: **Łukasz Jocz**

Dostępna pamięć: **32 MB** *Dzień 5, 24.09.2010*

Wykres

Duch Adi bardzo przykłada się do startów w konkursie Ghostcoder. Podczas każdych zawodów są do zrobienia dokładnie 2 zadania. Zatem wynik z danych zawodów można opisać jako liczbę zrobionych zadań należącą do zbioru $0, 1, 2$.

Adi bardzo skrupulatnie notuje wyniki z kolejnych konkursów. Teraz chciałby się pochwalić swoimi postępami przed kolegami, czyli chciałby wybrać wyniki z pewnych zawodów tak, aby ich wykres był rosnący i zarazem jak najdłuższy.

Wejście

Pierwszy i jedyny wiersz wejścia zawiera liczbę całkowitą n ($1 \leqslant n \leqslant 10^6$), oznaczającą liczbę zawodów, w których brał udział Adi. W następnym wierszu znajduje się n liczb, oznaczających wyniki Adiego w kolejnych zawodach.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę, oznaczającą długość najdłuższego rosnącego wykresu jaki może uzyskać Adi.

Przykład

dla danych wejściowych:

```
5
1 0 2 1 2
```

poprawnym wynikiem jest:

```
3
```

Zadanie: **Turniej** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **128 MB** *Dzień 5, 24.09.2010*

Turniej

W Bajtockim Turnieju Programistycznym bierze udział n zawodników. Każdy zawodnik ma pewną siłę i wiadomo, że dwóch różnych zawodników nie posiada takiej samej siły.

Codziennie odbywane są zawody z udziałem zawodników, którzy zakwalifikowali się z dnia poprzedniego. W jednym dniu zawodnicy dzieleni są losowo na pewną liczbę grup po k osób, spośród których odpada zawsze osoba z najmniejszą siłą (pozostałych $k-1$ osób zostaje zwycięzcami w danej grupie). Może się zdarzyć, że jedna grupa nie będzie posiadała k osób. W tym wypadku wszystkie osoby z danej grupy przechodzą automatycznie do zawodów następnego dnia. Turniej się kończy, jeśli nie można już podzielić osób na co najmniej jedną grupę o liczbie osób k. W całym turnieju szukamy więc $k-1$ zwycięzców.

Zastanawiamy się ile różnych osób może zwyciężyć w tym turnieju.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą z ($1 \leqslant z \leqslant 100$), oznaczającą liczbę zestawów danych. Każdy zestaw danych zawiera po dwie liczby całkowite n_i i k_i ($2 \leqslant k_i, n_i \leqslant 10^6$), oznaczające odpowiednio liczbę osób biorących udział w turnieju, oraz liczbę osób, na które dzielone są grupy.

Suma n dla wszystkich zestawów danych nie przekroczy wartości 10^6.

Wyjście

Dla każdego zapytania w osobnym wierszu powinna znaleźć się jedna liczba całkowita oznaczająca liczbę różnych osób, które mogą być zwycięzcami w całym turnieju.

Przykład

dla danych wejściowych:

```
1
2 2
```

poprawnym wynikiem jest:

```
1
```

Zadanie: **Przyjęcie** Autor zadania: **Joanna Bujnowska**

Dostępna pamięć: **32 MB** *Dzień 5, 24.09.2010*

Przyjęcie

Mała Kasia urządza przyjęcie urodzinowe. Na tę okazję kupiła n rodzajów cukierków. Liczba cukierków k – tego rodzaju wynosi x_k. Kasia nie próbowała żadnego rodzaju cukierka, dlatego chciałaby zaprosić tylu gości, aby po równomiernym rozdzieleniu cukierków między gości pozostał jej co najmniej jeden cukierek każdego rodzaju.

Kasia rozdziela cukierki w taki sposób, że każdy gość musi dostać tą samą liczbę cukierków k – tego rodzaju. Dodatkowo jeśli Kasia może wybrać tę liczbę na różne sposoby, to musi wybrać maksymalną z nich (przykładowo Kasia ma 10 cukierków i rozdziela je 4 osobom. Może dać każdej osobie po 0, 1 lub 2 cukierki, więc daje maksymalną liczbę, czyli 2). Cukierków nie można dzielić na mniejsze części.

Kasia nie jest zbyt towarzyska, więc zastanawia się, jaką najmniejszą liczbę osób powinna zaprosić, aby spełnione były warunki zadania.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n, oznaczająca liczbę różnych rodzajów cukierków ($1 \leqslant n \leqslant 10^6$). Drugi wiersz zawiera n liczb całkowitych x_k ($1 \leqslant x_k \leqslant 10^5$) oddzielonych spacją, oznaczających liczbę cukierków k – tego rodzaju.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą, równą minimalnej liczbie osób.

Przykład

dla danych wejściowych:

```
4
2 4 9 10
```

poprawnym wynikiem jest:

```
6
```

Rozdział 6

Rozwiązania

6.1 Dzień próbny

Opracowanie: **Kostki domina** Autor opracowania: **Robert Kozikowski, Jacek Tomasiewicz**

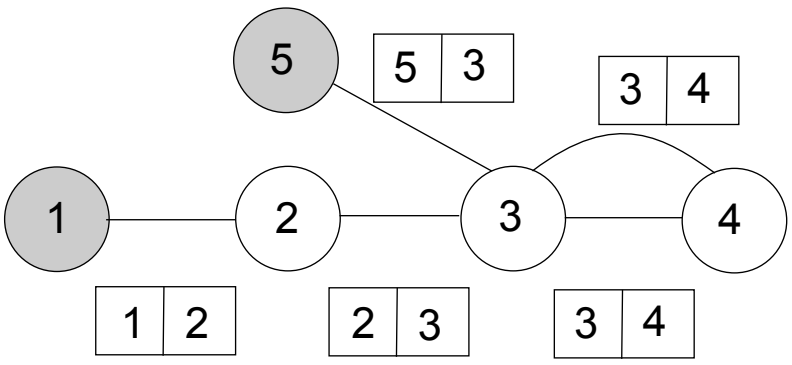
Dostępna pamięć: **32 MB** *Dzień 0, 19.09.2010*

Kostki domina

Uproszczenie zadania

Zadanie dużo łatwiej jest rozwiązać, jeśli zinterpretujemy je w teorii grafów. Wyobraźmy sobie, że mamy jakąś ścieżkę w grafie idącą po wierzchołkach $(v_1, v_2, ..., v_n)$. Zauważmy, że jest ona jednoznaczna z ułożeniem w łańcuch następujących kostek domina:$([v_1, v_2], [v_2, v_3], ..., [v_{n-1}, v_n])$.

Zatem możemy stworzyć sobie graf nieskierowany, który jako zbiór krawędzi zawiera wszystkie kostki domina (jeśli mamy kostkę $[x, y]$ to dodajemy krawędź między wierzchołkami x, y), a jako zbiór wierzchołków wszystkie wierzchołki o stopniu co najmniej 1 (czyli że co najmniej 1 krawędź kończy się w danym wierzchołku).



Rysunek przedstawiający zamianę kostek domina na graf. Przykładowe kostki [1,2], [2,3], [3,4], [3,4], [3,5].

Ciąg ułożony ze wszystkich kostek będzie jednoznaczny ze *Ścieżką Eulera* w takim grafie. *Ścieżka Eulera* to taka ścieżka, która przechodzi po wszystkich krawędziach grafu dokładnie jeden raz. Jednakże musi być możliwa do ułożenia tylko jedna taka ścieżka. Zatem nie może istnieć w danym grafie *Cykl Eulera* (czyli że istnieje cykl przechodzący po wszystkich krawędziach), bo wtedy może być wiele różnych odpowiedzi (każda para v_x, v_x).

W zadaniu zależy nam więc, aby znaleźć wierzchołek początkowy i końcowy *Ścieżki Eulera*.

Rozwiązanie wzorcowe

W naszym rozwiązaniu przydatne będzie następujące twierdzenie:

Twierdzenie.1. ***Twierdzenie o cyklu Eulera.*** *Niepusty graf spójny ma cykl Eulera wtedy i tylko wtedy, gdy stopień każdego wierzchołka jest liczbą parzystą.*

Twierdzenie.2. ***Twierdzenie o ścieżce Eulera.*** *Niepusty graf spójny ma ścieżkę Eulera, ale nie ma cyklu Eulera wtedy i tylko wtedy, gdy stopień każdego, oprócz dwóch wierzchołków, jest liczbą parzystą. Wtedy ta ścieżka zaczyna się w jednym z tych wierzchołków, a kończy w drugim.*

W zadaniu zależy nam, aby znaleźć wierzchołek początkowy i końcowy *ścieżki Eulera*. Na podstawie powyższego twierdzenia widać, że muszą być to dwa różne wierzchołki o stopniu nieparzystym. Poza tym wszystkie pozostałe wierzchołki muszą mieć stopień parzysty oraz graf musi być spójny.

Wiemy, że ścieżka zawsze istnieje, więc będą to wierzchołki o stopniu nieparzystym. Możemy to zrobić w taki sposób, że dodajemy do wektora pary $[a, b]$ oraz $[b, a]$, następnie sortujemy wektor i dla danego x liczymy, ile jest par $[x, a]$ (będą one obok siebie w posortowanym wektorze).

Należy sprawdzić jeszcze dwa szczególne przypadki, gdy graf ma 1 bądź 2 wierzchołki (pamiętajmy, że może być bardzo dużo klocków). Wtedy nawet jeśli istnieje *cykl Eulera*, to istnieje jednoznaczny sposób stworzenia ścieżki.

6.2 Dzień pierwszy

Opracowanie: **Sznurki** Autor opracowania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 1, 20.09.2010*

Sznurki

Rozwiązanie wzorcowe

Zadanie rozwiązujemy zachłannie. Poczynając od pierwszego sznurka łączymy go z kolejnymi, dopóki nie utworzą łącznej długości nie mniejszej od wzrostu Jasia. Następnie bierzemy kolejny sznurek i łączymy z następnymi.

Zauważmy, że takie postępowanie jest optymalne, gdyż pierwszy sznurek możemy połączyć tylko ze sznurkiem drugim, a jeśli jego długość nie jest sama w sobie wystarczająca, to opłaca się go wykorzystać zwiększając inny sznurek (a mamy tylko jedną możliwość). Po połączeniu drugi sznurek staje się pierwszym i postępujemy analogicznie.

```
1  wczytaj(n, wzrost)
2  suma := 0
3  wynik := 0
4  for k := 1 to n do
5      wczytaj(x)
6      suma := suma + x
7      if suma >= wzrost then
8          wynik := wynik + 1
9          suma := 0
10 wypisz(wynik)
```

Złożoność czasowa tego rozwiązania wynosi $O(n)$, a pamięciowa $O(1)$.

Opracowanie: **Wieża** Autor opracowania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 1, 20.09.2010*

Wieża

Rozwiązanie wzorcowe

Zadanie rozwiążemy za pomocą wyszukiwania binarnego. Dla każdego schodka należy znaleźć wysokość schodka o największej wysokości, leżącego na drodze od samego dołu wieży do miejsca danego schodka (np. dla kolejnych wysokości schodków 1, 5, 3, 6, 2, 1, 7 stworzymy ciąg 1, 5, 5, 6, 6, 6, 7). Otrzymany ciąg będzie niemalejący, dzięki temu będziemy mogli wykonać na nim wyszukiwanie binarne. Dla każdego mieszkańca możemy w czasie $O(\log n)$ znaleźć miejsce, w którym się zatrzyma, będzie to pierwszy wyższy bądź równy klocek od jego wzrostu.

```
1 wczytaj(n, m, schodek[], mieszkaniec[])
2 ciag[] := 0
3 for k := 1 to n do
4     ciag[k] := max(ciag[k-1], schodek[k])
5 for k := 1 to m do
6     miejsce := wyszukaj_binarnie(mieszkaniec[k])
7     wypisz(miejsce)
```

Ponieważ mamy m mieszkańców, to złożoność całego rozwiązania wyniesie $O(m \log n)$, co nas całkowicie satysfakcjonuje.

Opracowanie: **Tablica** Autor opracowania: **Łukasz Jocz**

Dostępna pamięć: **64 MB** *Dzień 1, 20.09.2010*

Tablica

Rozwiązanie wzorcowe

Dla każdego z ciągów a i b policzmy tablice $pozA[]$ i $pozB[]$ – oznaczające dla indeksu i numer pozycji w ciągu, na której wystąpi i-te zero, oraz tablice $sufA[]$ i $sufB[]$ – oznaczające dla indeksu i, ile jest jedynek w podciągu $[i...n]$ danego ciągu.

Załóżmy, że w optymalnym rozwiązaniu występuje i zer, wówczas ostatnie zero wystąpi w ciągu a na pozycji $pozA[i]$. Ponieważ ostatnie zero wystąpiło na pozycji $pozA[i]$ i więcej zer już nie wystąpi (bo założyliśmy, że bierzemy dokładnie i zer), możemy do wynikowego ciągu dodać wszystkie jedynki z pozycji $[pozA[i]+1...n]$, a liczba takich jedynek to $sufA[pozA[i]+1]$. Jednak w ciągu a i b musimy wziąć taką samą liczbę jedynek. Liczba tych jedynek będzie więc wynosiła $min(sufA[pozA[i]+1], sufB[pozB[i]+1])$.

Obliczamy więc dla każdego i wartość $i + min(sufA[pozA[i]+1], sufB[pozB[i]+1])$. Wynikiem będzie maksymalna z tych wartości.

```
1  wczytaj(n, m, a[], b[])
2  Aile0 := 0
3  Bile0 := 0
4  wynik := 0
5  for i := 1 to n do
6     if a[i] = 0 then
7        Aile0 := Aile0 + 1
8        pozA[Aile0] := i
9  for i := 1 to m do
10    if b[i] = 0 then
11       Bile0 := Bile0 + 1
12       pozB[Bile0] := i
13 for i := n downto 1 do
14    sufA[i] := sufA[i+1]
15    if a[i] = 1 then
16       sufA[i] := sufA[i] + 1
17 for i := m downto 1 do
18    sufB[i] := sufB[i+1]
19    if b[i] = 1 then
20       sufB[i] := sufB[i] + 1
21 for i := 1 to min(Aile0, Bile0) do
22    wynik := max(wynik, i + min(sufA[pozA[i]+1],sufB[pozB[i]+1]) )
23 wypisz(wynik)
```

6.3 Dzień drugi

Opracowanie: **Obwody** Autor opracowania: **Joachim Jelisiejew**

Dostępna pamięć: **32 MB** *Dzień 2, 21.09.2010*

Obwody

Rozwiązanie wzorcowe

Mamy znaleźć największą liczbę drutów, taką że $b_i \leqslant d_i$, innymi słowy $0 \leqslant d_i - b_i$.

Dla każdego drutu obliczamy $d_i - b_i$, po czym sortujemy te liczby i liniowo (od największej) sumujemy wartości $d_i - b_i$ dopóki suma jest nieujemna.

```
1  wczytaj(n)
2  for i := 1 to n do
3      wczytaj_kolejny_drut(d, b)
4      t[i] := d - b
5  posortuj(t[])
6  suma := 0
7  wynik := 0
8  for i := n downto 1 do
9      suma := suma + t[i]
10     if suma >= 0 then
11         wynik := wynik + 1
12     else
13         break
14 wypisz(wynik)
```

Ze względu na sortowanie rozwiązanie będzie więc miało złożoność $O(n \log n)$. Zadanie można jednak rozwiązać liniowo, czyli w czasie $O(n)$, wykorzystując sortowanie przez zliczanie.

Opracowanie: **Wieża 2** Autor opracowania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 2, 21.09.2010*

Wieża 2

Rozwiązanie wzorcowe

Na początku dla każdego schodka należy znaleźć wysokość schodka o największej wysokości leżącego na drodze od samego dołu wieży do miejsca danego schodka (np. dla kolejnych wysokości schodków 1, 5, 3, 6, 2, 1, 7 stworzymy ciąg 1, 5, 5, 6, 6, 6, 7). Otrzymany ciąg będzie niemalejący.

Następnie dla pierwszego mieszkańca w kolejce znajdujemy miejsce jego zatrzymania się. Poruszamy się od wierzchołka wieży w dół. Wiemy, ze kolejna osoba z kolejki zatrzyma się najwyżej jedno miejsce niżej od poprzedniego mieszkańca. Kolejne wyszukiwanie zaczynamy więc od miejsca zatrzymania się poprzedniego mieszkańca. W ten sposób przejrzymy każdy schodek dokładnie jeden raz, dzięki czemu złożoność całego rozwiązania wynosić będzie $O(n+m)$.

```
1  wczytaj(n, m, schodek[], mieszkaniec[])
2  ciag[] := 0
3  for k := 1 to n do
4     ciag[k] := max(ciag[k-1], schodek[k])
5  miejsce := n
6  for k := 1 to m do
7     while ciag[miejsce] >= mieszkaniec[k] do
8        miejsce := miejsce - 1
9     wypisz(miejsce)
10    if miejsce > 0 then
11       miejsce := miejsce - 1
```

Opracowanie: **Akwarium** Autor opracowania: **Robert Kozikowski**

Dostępna pamięć: **64 MB** *Dzień 2, 21.09.2010*

Akwarium

Spostrzeżenia

Najważniejszym spostrzeżeniem jest to, że w każdym dniu umiera co najmniej połowa żyjących ryb – jeśli liczba ryb jest parzysta, to połowa najcięższych zjada najlżejszą połowę, jeśli jest nieparzysta, to dodatkowo środkowa ryba umiera, gdyż nie ma kogo zjeść. Dzięki temu możemy zauważyć, że ryby mogą żyć co najwyżej $\log n$ dni, czyli dla maksymalnej wartości n ryby mogą żyć co najwyżej 20 dni. Z powodu tak małej liczby dni możemy łatwo przeprowadzić symulację.

Rozwiązanie wzorcowe

W rozwiązaniu najpierw znajdujemy dla każdej ryby dzień jej śmierci, a następnie w czasie stałym odpowiadamy na każde z pytań. Aby znaleźć dzień śmierci każdej z ryb, postępujemy według następujących kroków:

1. Ustal, że dzień to 1.
2. Niech V to vektor wszystkich ryb.
3. Posortuj V tak, aby najpierw były ryby najlżejsze, a w przypadku takiej samej wagi najmłodsze.
4. Niech a to zerowy indeks wektora, b to indeks ostatniego elementu.
5. Dopóki $a < b$ powtarzaj krok 6.
6. Ustal, że ryba spod b zjada rybę spod a, dodaj najedzoną już rybę (ze zwiększoną wagą) spod b do wektora V_2 i ustal, że ryba spod a umiera dzisiaj. Następnie powiększ a o 1 i pomniejsz b o 1.
7. Jeśli $a == b$ to ustaw, że ryba a umiera dzisiaj (gdyż nie może nikogo zjeść)
8. Teraz niech $V = V_2$ i jeśli V_2 nie jest pusty wyczyść V_2, powiększ dzień o 1 oraz wróć do punktu 3.

Po przejściu takiego algorytmu możemy w czasie stałym odpowiadać, kiedy każda z ryb umarła. Powyższy algorytm działa w czasie $O(n(\log n)^2)$. W C++ można go bardzo łatwo zrealizować z użyciem STL (dodatkowe informacje na temat vectorów i sortowania w wykładzie *Praktyczne zastosowanie STL*).

6.4 Dzień trzeci

Opracowanie: **Tarasy** Autor opracowania: **Robert Kozikowski**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Tarasy

Rozwiązanie wzorcowe

Niech $r_1, r_2, ..., r_{n-1}$ to koszty kolejnych przejazdów z r_i do r_{i+1}. Warto zauważyć, że żadna z tych wartości nie będzie ujemna. Interesuje nas znalezienie jak najdłuższego spójnego podciągu tego ciągu takiego, że jego suma jest mniejsza równa k. Wtedy największa liczba tarasów, które może odwiedzić turysta za k kredytek, to długość tego podciągu plus jeden. Taki podciąg możemy znaleźć za pomocą tzw. "dwóch wskaźników", czyli:

1. Najpierw niech $a = 0$ oraz b to jak najdalszy indeks taki, że suma $r_0 + r_1 + ... + r_{b-1} + r_b$ jest mniejsza równa k.
2. Ustalamy, że potencjalny wynik to $(b - a + 1)$.
3. Dla każdego a od 1 do $n - 1$ zrób punkt 4.
4. Przesuń b jak najdalej w prawo, tak aby suma $r_a + r_{a+1} + ... + r_{b-1} + r_b$ pozostała mniejsza równa k. Sprawdź, czy nowe $(b-a+1)$ nie jest większe od potencjalnego wyniku. Jeśli tak, to ustal nowy potencjalny wynik na $(b - a + 1)$.
5. Wypisz wynik.

Wiadomo, że jeśli powiększymy a o 1 oraz b pozostanie takie samo to $r_a + r_{a+1} + ... + r_{b-1} + r_b$ zmaleje bądź pozostanie takie same (gdyż r_a nie było ujemne). Zatem możemy spróbować przesunąć b w prawo. Przesunięcie w lewo nie jest opłacalne, gdyż suma zmalała.

Zauważmy, że powyższy program ma złożoność liniową, gdyż oba wskaźniki a i b każdą komórkę tablicy odwiedzą tylko raz (ponieważ tylko wzrastają a nie maleją).

Opracowanie: **Rolki papieru** Autor opracowania: **Adam Iwaniuk**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Rolki papieru

Obserwacja

Pierwszą obserwacją jest to, że końcowa długość wszystkich rolek nie może przekroczyć długości najkrótszej rolki. W związku z tym, długość, do której musimy rozwinąć wszystkie rolki, jest z przedziału $[0, dmin = min(d_1, d_2, .., d_n)]$.

Rozwiązanie brutalne

Dla każdego rozwinięcia $[0, dmin]$ sprawdzamy, o ile centymetrów będziemy musieli zmienić rozwinięcie danej rolki i wybieramy najlepszą wartość rozwinięcia.

```
1  wczytaj(n, d[], r[])
2  dmin := 10^9
3  for i := 1 to n do
4      if d[i] < dmin then
5      dmin := d[i]
6  wynik := 10^16
7  for j := 0 to dmin do
8      tmp:=0
9      for i := 1 to n do
10         tmp := tmp + abs(r[i] - j)
11         if tmp < wynik then
12             wynik := tmp
13 wypisz(wynik)
```

Złożoność czasowa takiego rozwiązania to $O(n * dmin)$, co dla maksymalnych danych może być równe $10^6 * 10^9$, czyli stanowczo za dużo.

Rozwiązanie wzorcowe

Nasz problem wygląda w następujący sposób:
Mamy pewien ciąg liczb i chcemy wskazać taką liczbę, aby suma różnic bezwzględnych z wszystkimi liczbami z naszego ciągu była jak najmniejsza. Okazuje się, że takie własności ma mediana, czyli środkowa liczba w posortowanym ciągu (jeśli w ciągu jest parzysta liczba liczb to możemy wybrać dowolną z dwóch środkowych). Pozostaje jeszcze problem, gdy nasza mediana jest wyższa niż wartość *dmin*. Łatwo zauważyć, iż im dalej oddalamy się od mediany, tym bardziej rośnie liczba potrzebnych ruchów, a więc wtedy naszą wartością, do której wyrównujemy, jest *dmin*.

```
1  wczytaj(n, d[], r[])
2  dmin := 10^9
3  for i := 1 to n do
4      if d[i] < dmin then
5      dmin := d[i]
6  posortuj(r)
7  mediana := min(dmin, d[n/2 + 1])
8  wynik := 0
9  for i := 1 to n do
10     wynik := wynik + abs(r[i] - mediana)
11 wypisz(wynik)
```

Jak widać, większość operacji wykonuje się liniowo względem n, jedynie sortowanie będzie nas kosztować $O(n \log n)$ operacji, a więc złożoność całego programu to $O(n \log n)$, co jest wystarczające dla $n = 10^6$

Zauważmy jeszcze, że wynik może być liczbą przekraczającą 2^{32}, a więc do liczenia wyniku należy użyć zmiennych 64-bitowych.

Opracowanie: **Karty** Autor opracowania: **Łukasz Jocz**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Karty

Rozwiązanie wzorcowe

Będziemy symulować kolejność kart w talii, zgodnie z podanymi na wejściu operacjami. Do symulowania talii kart wykorzystamy kolejkę, w której będziemy trzymać karty na pozycjach od 2 do n. Kartę z wierzchu talii będziemy trzymać osobno. Zauważmy, że nigdy nie będziemy ingerować w elementy od 3 do n, jedynie zdejmować elementy z góry i wrzucać na koniec kolejki. W wyniku operacji A:

- element pierwszy staje się ostatnim i zostaje dodany na koniec kolejki,
- element drugi staje się pierwszym, zatem opuszcza kolejkę.

W wyniku operacji B:

- element drugi staje się ostatnim, zatem zostaje zdjęty z początku kolejki i dodany na koniec.

6.5 Dzień czwarty

Opracowanie: **Las** Autor opracowania: **Robert Kozikowski**

Dostępna pamięć: **64 MB** *Dzień 4, 23.09.2010*

Las

Rozwiązanie brutalne

Rozwiązanie brutalne ma złożoność $O(n^4)$. Należy zauważyć, że jedno z drzew na planszy musi być najstarsze. Dla każdego drzewa zakładamy, że jest ono najstarsze i wykonujemy przeszukiwanie grafu wszerz (BFS) po drzewach o takim samym wieku bądź młodszych. Z wszystkich takich drzew, z których da się zbudować dom o wielkości poszukiwanej przez pana Jana, wybieramy najmłodsze.

Rozwiązanie wzorcowe alternatywne

Rozwiązanie alternatywne ma złożoność $O(n^2 \log n^2)$. Rozwiązanie opiera się na wyszukiwaniu binarnym. Zauważmy, że jeżeli można zbudować chatkę taką, że najstarsze drzewo ma wiek co najwyżej x, to można także zbudować taką (lub większą) chatkę dla każdej większej wartości x. Czyli jeśli wiemy, że x jest z przedziału $[a, b]$ $(abs(a - b) > 1)$ i można zbudować chatkę o wielkości y, to x jest z przedziału $[a, y]$. W przeciwnym przypadku wiek tego drzewa będzie z przedziału $[y + 1, b]$.

Zauważmy teraz, że wiek najstarszego drzewa musi być jednym z drzew (co jest oczywiste), więc należy posortować wszystkie możliwe wysokości drzew i na takiej tablicy wyszukiwać binarnie. Jeśli chcemy sprawdzić, czy można zbudować dom tak, że najstarsze drzewo ma wiek co najwyżej x, to idziemy po tablicy i jeśli natrafimy na nieodwiedzone pole o wysokości mniejszej bądź równej x, to puszczamy z niego algorytm BFS po punktach o wielkości mniejszej bądź równej x i jeśli którykolwiek z tych BFSów pokryje większe pole niż dom pana Jana, to dla danej wielkości x da się zbudować dom. Las

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe ma złożoność $O(n^2 \log n^2)$. Rozwiązanie opiera się na zbiorach rozłącznych. W rozwiązaniu idziemy po drzewach zaczynając od najmłodszych i idąc w górę wiekiem (tworzymy wektor par (wysokość drzewa, pozycja drzewa), a następnie go sortujemy).

Dla każdego z nich sprawdzamy kolejno wszystkie pola sąsiednie $(x-1, y), (x+1, y), (x, y-1), (x, y+1)$, czy którekolwiek ma nie większy wiek niż pole (x, y). Jeśli tak, to łączymy odpowiadające im zbiory rozłączne (oraz dodajemy do siebie ich wielkości). Jeśli w którymś momencie wielkość zbioru rozłącznego przekroczy (lub dorówna) rozmiar domu pana Jana, to wypisujemy wiek aktualnego drzewa, gdyż wtedy najstarsze wycięte drzewo będzie miało taki wiek (zawsze dla danego wieku drzewa wiemy jak największy dom możemy zbudować tak, że dane drzewo jest najstarsze).

Opracowanie: **Domino** Autor opracowania: **Łukasz Jocz**

Dostępna pamięć: **32 MB** *Dzień 4, 23.09.2010*

Domino

Ogólne spostrzeżenia

Spójrzmy na zadanie trochę z innej perspektywy. Przewróćmy wszystkie klocki domina i potraktujmy je po przewróceniu jak przedziały na osi liczbowej. Klocek znajdujący się w odległości d od klocka pierwszego mający wysokość h będzie reprezentowany przez przedział $[d, d + h]$.

Naszym zadaniem jest teraz znaleźć największy zbiór przedziałów, po usunięciu których suma pozostałych przedziałów będzie spójnym przedziałem, inaczej mówiąc – kolejne przedziały parami będą miały niepustą część wspólną.

Na początku obliczmy tablicę $D[i]$ - początek i-tego przedziału oznaczający odległość i-tego klocka od klocka pierwszego, oraz $K[i]$ - koniec i-tego przedziału.

```
1 for i := 1 to n do
2     D[i] := D[i-1] + x[i-1]
3     K[i] := D[i] + h[i]
```

Rozwiązanie wykładnicze

Zastanówmy się nad jakimkolwiek poprawnym rozwiązaniem naszego zadania. Najprostszym pomysłem jest przeszukanie przestrzeni wszystkich możliwych rozwiązań. Przeglądamy więc wszystkie podzbiory klocków, zakładamy, że właśnie te klocki zostaną usunięte i sprawdzamy, czy pozostałe klocki spełniają odpowiednie warunki. Spośród wszystkich poprawnych podzbiorów wybieramy ten o największej liczbie elementów. Takie rozwiązanie działa w złożoności $O(n2^n)$ i dostawało 20 punktów.

Programowanie dynamiczne

Spróbujmy podejść do tego zadania za pomocą programowania dynamicznego. Przez „połączenie" między przedziałem x i przedziałem y będziemy rozumieć ciąg przedziałów, które kolejno zachodzą na siebie(tj. ich przecięcie jest niepuste), pierwszym elementem tego ciągu jest x, a ostatnim y.

We wszystkich rozwiązaniach zamiast szukać największego zbioru przedziałów, który mamy usunąć, będziemy szukać najmniejszego zbioru przedziałów, który jest „połączeniem" między przedziałem 1 i n.

Dla każdego przedziału i będziemy obliczać wartość $dp[i]$, która oznacza minimalną długość „połączenia" między przedziałem 1 i przedziałem i. Wartości $dp[i]$ możemy policzyć następująco. $dp[i] = \min_{j<i, K[j]>D[i]} dp[j] + 1$.

Inaczej mówiąc, sprawdzamy wszystkie przedziały, które „zachodzą" na przedział i z lewej strony tj. takie, które rozpoczynają się PRZED początkiem przedziału i-tego, a kończą ZA początkiem przedziału i-tego. Niech j będzie jednym z takich przedziałów. Długość najkrótszego „połączenia" między przedziałami 1 i j to $dp[j]$. Z kolei „połączenie" między 1 a i może być zrealizowane jako połączenie 1 z j, a następnie j z i - w ten sposób „połączenie" ma długość $dp[j] + 1$. $dp[i]$ jest równe minimalnej spośród tych wartości dla wszystkich j.

Rozwiązanie to działa w złożoności $O(n^2)$. Takie rozwiązanie dostawało 40 punktów. Pseudokod tego rozwiązania mógłby wyglądać następująco:

```
1  for i := 1 to n do
2      dp[i] := INF
3      for j := 1 to i-1 do
4          if K[j] > D[i] and dp[j]+1 > dp[i] then
5              dp[i] := dp[j] + 1
6  wypisz(n - dp[n])
```

Zastanówmy się, jak przyspieszyć to rozwiązanie. Zamiast bezpośrednio przeglądać wszystkie poprzednie elementy w celu znalezienia minimum, możemy skorzystać z drzewa przedziałowego.

Kolejne liście drzewa będą odpowiadać klockom rosnąco według końców przedziałów $K[j]$. Teraz, chcąc obliczyć $dp[i]$, szukamy minimum na takim przedziale elementów, dla których $K[j] > D[i]$. Przedział elementów, które spełniają ten warunek, jest spójny, gdyż elementy są posortowane wg wartości $K[j]$.

Takie rozwiązanie działa w złożoności $O(n \log n)$ i dostawało od 80 do 100 punktów. Po więcej informacji o drzewach przedziałowych odsyłamy Czytelnika do wykładu Jakuba Radoszewskiego na http://was.zaa.mimuw.edu.pl.

Rozwiązanie wzorcowe

Załóżmy, że wybieramy po kolei kolejne klocki, które mają upaść. Niech ostatnim klockiem, który do tej pory wybraliśmy, będzie i. Chcemy teraz wybrać następny klocek j. Zastanówmy się, jaki klocek opłaca się wybrać.

A no taki, którego przedział „zachodzi" na przedział i z prawej strony i koniec tego przedziału jest możliwie najdalej. Czyli zachodzą warunki $j > i$ oraz $D[j] < K[i]$ oraz $K[j]$ jest możliwie największe. Dlaczego?

Dowód.

Załóżmy przeciwnie, że istnieje rozwiązanie, w którym otrzymamy lepszy wynik niż w wyżej opisanym. Ponieważ nie wybieraliśmy wg powyższego schematu, więc istnieje przedział j, po którym wybraliśmy przedział k, a najdalej kończącym się przedziałem „zachodzącym" z prawej na j jest l. Zamieńmy teraz w tym rozwiązaniu klocek k na l. Zmienione rozwiązanie jest tak samo dobre jak poprzednie i również jest poprawne. Stąd wniosek, że postępując wg powyższego schematu otrzymujemy najlepsze rozwiązanie. ■

Pseudokod rozwiązania działającego wg powyższego schematu mógłby wyglądać następująco:

```
1  last := 1
2  wynik := 2
3  while K[last] <= D[n] do
4      next := last + 1
5      j := last + 1
6      while K[last] > D[j] do
7          if K[j] > K[next] then
8              next := j
9          j := j + 1
10     last := next
11     wynik := wynik + 1
12 wypisz(n - wynik)
```

Zauważmy, że jeśli dla pewnego *last* przejrzymy wszystkie j, dla których $j > last$, $D[j] < K[last]$ i najdalej kończy się przedział *next*, to w kolejnej iteracji pętli nie ma tak naprawdę potrzeby

przeglądać takich przedziałów, których początek jest w przedziale $[D[next], K[last]]$, ponieważ te przedziały w całości zawierają się w przedziale $next$ i z tego powodu nie mogą nam poprawić rozwiązania. Jednak każde takie j zostanie przejrzane co najwyżej 2 razy, dlatego złożoność powyższego algorytmu jest liniowa. Dlaczego? Niech l, l', l'' będą trzema kolejno wybranymi przez nasz algorytm przedziałami. Czyli $D[l''] > D[l']$ oraz $D[l''] < K[l']$. Zauważmy, że musi zachodzić warunek $D[l''] \geqslant K[l]$. Gdyby tak nie było, to najdalej kończącym się przedziałem, który zachodzi na l, byłoby l'' a nie l'. Zatem wszystkie przedziały, których początek jest między $D[l']$, a $K[l]$, zostaną przejrzane tylko przy obiegu pętli, dla l i l'. Algorytm ten ma złożoność $O(n)$.

Opracowanie: **Suma** Autor opracowania: **Łukasz Jocz**

Dostępna pamięć: **32 MB** *Dzień 4, 23.09.2010*

Suma

Rozwiązanie wzorcowe

Zauważmy, że suma z treści zadania jest równoważna sumie liczb od 1 do $2^k - 1$. Wzór na sumę liczb od 1 do n wynosi $\frac{n*(n+1)}{2}$. Podstawiając do tego wzoru $2^k - 1$ otrzymujemy $\frac{(2^k-1)*(2^k)}{2} = (2^k - 1) * 2^{k-1}$. Liczba $2^k - 1$ binarnie ma postać k jedynek. Pomnożenie liczby przez liczbę 2^{k-1} to w systemie dwójkowym dodanie na końcu liczby $k - 1$ zer. Zatem szukana liczba składa się z k jedynek, a następnie $k - 1$ zer.

```
1 wczytaj(k)
2 for i := 1 to k do
3     wypisz(1)
4 for i := 1 to k - 1 do
5     wypisz(0)
```

6.6 Dzień piąty

Opracowanie: **Wykres** Autor opracowania: **Łukasz Jocz**

Dostępna pamięć: **32 MB** *Dzień 5, 24.09.2010*

Wykres

Rozwiązanie wzorcowe

Problem przedstawiony w zadaniu jest problemem najdłuższego podciągu rosnącego. Iterując po kolejnych wynikach Adiego będziemy zapamiętywać, jakie pociągi już wystąpiły i w miarę możliwości dodawać nowe podciągi. Zauważmy, że wystarczą nam informacje, czy wystąpiły podciągi 0, 1 i 01.

```
1  wczytaj(n, t[])
2  bylo0 := bylo1 := bylo01 := 0
3  wynik := 1
4  for i := 1 to n do
5      if t[i] == 0 then
6          bylo0 := 1
7      if t[i] == 1 then
8          bylo1 := 1
9          if bylo0 == 1 then
10             bylo01 := 1
11             wynik := max(wynik, 2)
12     if t[i] == 2 then
13         if bylo01 == 1 then
14             wynik := max(wynik, 3)
15         if bylo0 == 1 or bylo1 == 1 then
16             wynik := max(wynik, 2)
17 wypisz(wynik)
```

Opracowanie: **Turniej** Autor opracowania: **Robert Kozikowski**

Dostępna pamięć: **128 MB** *Dzień 5, 24.09.2010*

Turniej

Rozwiązanie wzorcowe

Zauważmy, że jeśli liczba zawodników jest mniejsza niż wielkość grupy, to wszyscy automatycznie wygrywają.

Od tej pory zajmować się będziemy tylko przypadkiem, gdy liczba zawodników jest nie mniejsza od wielkości grupy. Wtedy zwycięzcą jakkolwiek przeprowadzonego turnieju może zostać dokładnie $k-1$ najlepszych zawodników (gdzie k to liczba osób, na które dzielone są grupy). Zatem wynikiem dla każdego testu jest $min(n, k-1)$

Wyjaśnienie rozwiązania

Pierwszą obserwacją jest to, że nie może być więcej zwycięzców dowolnie przeprowadzonego turnieju niż $k-1$ zawodników, gdyż wtedy turniej odbywa się dopóki zawodników można podzielić na co najmniej jedną grupę k osobową.

Mniej niż $k-1$ zawodników nie może zostać zwycięzcami dowolnie przeprowadzonego turnieju, gdyż w każdej turze, jeśli mamy n zawodników, może zostać wyeleminowanych co najwyżej $\lfloor \frac{n}{k} \rfloor$ zawodników, zatem od chwili gdy $n < 2 * k$ eliminowany jest w każdej turze jeden zawodnik.

Teraz pokażemy, że dowolnie przeprowadzony turniej zawsze wygrywa $k-1$ najlepszych zawodników. Istotnie przypuśćmy, że któryś z $k-1$ najlepszych zawodników nie należy do zwycięzców. Wtedy, aby zostać wyeleminowanym, musiałaby istnieć grupa składająca się z k zawodników, w której był on najsłabszy. Niestety nie jest to możliwe, gdyż musiałoby istnieć $k-1$ lepszych od niego zawodników. Jest to niemożliwe, gdyż siły wszystkich zawodników są różne, a od danego zawodnika istnieje co najwyżej $k-2$ lepszych zawodników.

Opracowanie: **Przyjęcie** Autor opracowania: **Adam Iwaniuk**

Dostępna pamięć: **32 MB** *Dzień 5, 24.09.2010*

Przyjęcie

Wstęp

Z warunków zadania łatwo możemy zauważyć, że aby Kasia mogła spróbować cukierka jakiegoś rodzaju, to liczba tych cukierków nie może być podzielna przez liczbę gości. A więc, naszym zadaniem jest znalezienie najmniejszej liczby dodatniej, która nie dzieli żadnej spośród n podanych nam liczb.

Rozwiązanie brutalne

Zauważmy, że największa liczba, jaka może być odpowiedzią, to 10^5+1, ponieważ cukierków każdego rodzaju może być co najwyżej 10^5, a przy 10^5+1 nikt z gości nie dostanie żadnego cukierka i wszystkie trafią do Kasi.

Możemy więc wczytać do tablicy wszystkie liczby zawierające ile jest cukierków danego rodzaju i sprawdzić po kolei wszystkie liczby od 1 do 10^5+1, czy żadna z liczb w tablicy nie jest podzielna przez sprawdzaną liczbę.

```
1  wczytaj(n)
2  for i := 0 to n-1 do
3     wczytaj tab[i]
4  for i := 1 to 10^5+1 do
5     podzielna := false
6     for j := 0 to n-1 do
7        if tab[j] mod i == 0 then
8           podzielna := true
9     if podzielna == false then
10       wypisz(i)
11       zakoncz_program()
```

Takie rozwiązanie, w najgorszym przypadku, wymaga $10^5 * 10^6$ operacji, czyli 10^{11}, co niestety jest za wolne.

Rozwiązanie wolne

Zauważmy dwie rzeczy:

1. Jeśli jest tyle samo cukierków dwóch różnych rodzajów, to nie zmieniłaby nam się sytuacja, jeśli jeden z tych rodzajów byśmy całkowicie odrzucili.

2. Wszystkich cukierków może być 10^6, a wszystkich możliwych liczb cukierków w każdym rodzaju jest 10^5, a zatem przy 10^6 cukierków mamy na pewno powtórki, po eliminacji których liczba różnych rodzajów wynosiłaby 10^5.

Stąd, po odrzuceniu duplikatów, w tablicy mielibyśmy maksymalnie $10^5 * 10^5 = 10^{10}$ obliczeń co jednak wciąż jest za wolne.

Rozwiązanie wzorcowe

Z poprzedniego punktu wiemy, że nie musimy trzymać informacji o tym, ile jest cukierków danego rodzaju, a tylko informację o tym, czy istnieje rodzaj, w którym występuje k cukierków. Dane możemy reprezentować jako *boolowska* tablica z indeksami od 1 do 10^5, w której to jest *false* jeśli indeks w tablicy jest liczbą, która nie jest ilością cukierków żadnego rodzaju, a *true* w przeciwnej sytuacji.

Taką tablicę możemy wczytać w następujący sposób.

```
1  tab[] := {false}
2  wczytaj(n)
3  for i := 0 to n-1
4      wczytaj(j)
5      tab[j] := true
```

Następnie, tak jak ostatnio, sprawdzamy wszystkie liczby od 1 do 10^5+1, czy spełniają warunki zadania, z tym że, aby sprawdzić czy dana liczba dzieli jakąś z wczytanych liczb, wystarczy sprawdzić w naszej tablicy, czy w indeksach będących wielokrotnościami danej liczby występuje wartość true.

```
1  for i := 1 to 10^5+1 do
2      dzieli := false
3      c := i
4      while c < 10^5 do
5          if tab[c] = true then
6              dzieli := true
7          c := c + i
8      if dzieli == false then
9          wypisz(i)
10         zakoncz_program()
```

Łatwo zauważyć, że sprawdzając coraz większe liczby, musimy w tablicy dokonać coraz mniej sprawdzeń (dla 1 musimy sprawdzić w 10^5 miejscach, dla 2 w $10^5/2$, ale już dla 10 w 10^4 miejscach, natomiast 1000 wymaga juz tylko 100 sprawdzeń, co w sumie daje nam nieco ponad milion sprawdzeń, co jest już wystarczająco szybkie).

Rozdział 7

Wykłady

7.1 Teoria liczb I

Wykład: **Teoria liczb I** Autor wykładu: **Łukasz Jocz**

7.1.1 Wstęp

Przypomnijmy dwie definicje.

Definicja.1. ***Liczby pierwsze*** *to liczby naturalne, które posiadają dokładnie dwa dzielniki (liczbę 1 i samą siebie).*

Definicja.2. ***Liczby złożone*** *to liczby naturalne, które posiadają więcej niż dwa dzielniki.*

7.1.2 Sprawdzenie czy liczba jest pierwsza.

Aby sprawdzić, czy liczba n większa od 1 jest pierwsza, przeglądamy wszystkie liczby całkowite większe od 1 i mniejsze od n. Jeżeli którakolwiek z tych liczb dzieli n, to n jest **liczbą złożoną**. W przeciwnym wypadku n jest **liczbą pierwszą**. Złożoność tego algorytmu to $O(n)$.

W bardzo prosty sposób możemy przyspieszyć powyższy algorytm. Wystarczy zauważyć, że liczbę n możemy uznać za pierwszą, jeżeli nie posiada dzielników mniejszych równych od $\sqrt{n}$. Dlaczego? Jeżeli liczba n ma dzielnik a, to ma również dzielnik $\frac{n}{a}$. Co najmniej jeden z tych dwóch dzielników jest więc mniejszy lub równy $\sqrt{n}$. Gdyby tak nie było, to n musiałoby być iloczynem dwóch liczb większych od $\sqrt{n}$, co jest oczywiście niemożliwe. Przeglądamy więc wszystkie liczby od 2 do $\sqrt{n}$, jeżeli znajdziemy liczbę, która dzieli n, to n jest złożona. W przeciwnym wypadku n jest pierwsza. Złożoność tego algorytmu to $O(\sqrt{n})$. Spróbujmy to zapisać.

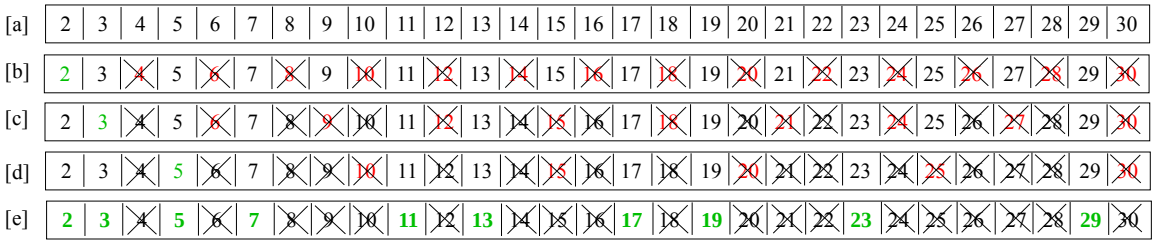
```
1 for(int i=2;i*i<=n;i++) {
2     if (n%i==0) {
3         zlozona=1;
4         break;
5     }
6 }
7 if(zlozona == 1) printf("Liczba n jest zlozona");
8 else printf("Liczba n jest pierwsza");
```

Warunek $i * i <= n$ w tym przypadku jest równoważny $i <= sqrt(n)$, jednak dzięki zastosowanej formie nie występuje konieczność konwersji typów i „wchodzenia" w liczby zmiennoprzecinkowe.

7.1.3 Sito Eratostenesa

Sito Eratostenesa jest znanym algorytmem znajdowania wszystkich liczb pierwszych z przedziału $[2, n]$. Nazwa pochodzi od tego, że wszystkie liczby są po kolei przesiewane i usuwane są wszystkie wielokrotności danej liczby. Żeby wszystko stało się jaśniejsze, przejdźmy do przykładu.

W pierwszym kroku[b] wybieramy najmniejszą liczbę, która nie została wykreślona, czyli 2, i wykreślamy wszystkie jej wielokrotności mniejsze od n. W następnych krokach powtarzamy to samo dla kolejnych niewykreślonych liczb, czyli kolejno 3[c] i 5[d]. Jak zauważyliśmy przy poprzednim algorytmie, każda liczba złożona ma dzielnik mniejszy lub równy $\sqrt{n}$. W szczególność ma też taki dzielnik pierwszy. W związku z tym, wystarczy wykreślić wielokrotności tylko tych liczb pierwszych, które są mniejsze lub równe $\sqrt{n}$. Wówczas każda liczba złożona z przedziału $[2, n]$ zostanie wykreślona.



Implementacja algorytmu może więc wyglądać następująco:

```
1 for(int i=2;i*i<=n;i++)
2     if(!skreslona[i])
3         for(int j=2*i;j<=n;j+=i)
4             skreslona[j] = 1;
```

Algorytm ten możemy jeszcze nieznacznie przyspieszyć zauważając, że nie musimy wykreślać wielokrotności liczby i, które są mniejsze od i^2. Wielokrotności takie mają postać $k * i, k < i$, i zostały już skreślone dla jednego z dzielników pierwszych k.

Po tej poprawce trzecia linijka mogłaby wyglądać tak:

```
1 for(int j=i*i;j<=n;j+=i)
```

Zastanówmy się, jaka jest złożoność sita. Przy obliczaniu liczby operacji nie będziemy uwzględniać ostatniej poprawki, gdyż nie wpływa ona na asymptotyczną złożoność. Dla każdej liczby pierwszej i mniejszej od $\sqrt{n}$ skreślamy $\frac{n}{i}$ liczb. Algorytm wykonuje więc:

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + ... = \sum_{p \in P, p \leqslant \sqrt{n}} \frac{n}{p} = n * \sum_{p \in P, p \leqslant \sqrt{n}} \frac{1}{p} \leqslant n * \sum_{p \in P, p \leqslant n} \frac{1}{p} = O(n\ lnln\ n) \tag{7.1}$$

Dowód na to, że suma odwrotności liczb pierwszych mniejszych od n jest równa asymptotycznie $O(lnln\ n)$, zostawmy matematykom, a po prostu zawierzmy Eulerowi, który to udowodnił.

Dzięki sicie Eratostenesa możemy w prosty sposób znaleźć wszystkie liczby pierwsze mniejsze od n, ale sito może się okazać dużo bardziej przydatne, jeżeli je lekko zmodyfikujemy. Dla każdej wykreślanej liczby będziemy dodatkowo zapisywać informację o dzielniku pierwszym, który ją dzieli.

```
1 for(int i=2;i*i<=n;i++)
2     if(!dzielnikP[i])
3         for(int j=i*i;j<=n;j+=i)
4             dzielnikP[j] = i;
```

Tablica $dzielnikP[i]$ dla $n = 25$.

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0	0	2	0	2	0	2	3	2	0	3	0	2	3	2	0	3	0	2	3	2	0	3	5

Dzięki takiemu podejściu możemy bardzo szybko rozkładać liczby na czynniki pierwsze. Jeżeli wiemy, że liczba x ma dzielnik pierwszy p, to jej rozkładem na czynniki pierwsze jest rozkład liczby $\frac{x}{p}$ oraz dzielnik p.

```
1 int i=x;
2 while(dzielnikP[i]>0) {
3     printf("%d*",dzielnikP[i]);
4     i/=dzielnikP[i];
5 }
6 printf("%d",i);
```

Liczba x nie może mieć więcej niż $O(lg\ x)$ czynników pierwszych. Dlaczego? Przy każdym kroku algorytmu zmniejszamy liczbę i co najmniej dwa razy i dodajemy jeden dzielnik pierwszy. W związku z tym po co najwyżej $lg\ x - 1$ obrotach pętli i zmaleje do liczby pierwszej i algorytm się zakończy. Faktoryzacja liczby x tą metodą ma złożoność czasową $O(lg\ x)$. Warto zauważyć, że kolejne otrzymywane czynniki przy tej metodzie nie są uporządkowane. Przykładowo rozkładając liczbę 18 otrzymujemy $3 * 2 * 3$. Problem ten możemy rozwiązać dwojako. Albo na końcu sortujemy otrzymane czynniki, albo stosujemy sito Eratostenesa bez ulepszeń, które wymusza kolejność czynników.

7.1.4 Własności rozkładu liczby na czynniki pierwsze

Dzięki rozkładowi na czynniki możemy policzyć dużo ciekawych własności liczb.

Największy wspólny dzielnik

Dla dwóch sfaktoryzowanych liczb a i b możemy łatwo policzyć ich największy wspólny dzielnik. Weźmy największy wspólny podzbiór dzielników pierwszych a i b. Podzbiór ten odpowiada największemu wspólnemu dzielnikowi tych liczb. Przykład: $a = 2 * 2 * 2 * 3 * 5, b = 2 * 2 * 3 * 3 * 7, nwd = 2 * 2 * 3$. Inaczej mówiąc, jeżeli dzielnik p występuje w rozkładzie a α razy, a w rozkładzie b β razy, to w rozkładzie $NWD(a, b)$ występuje $min(\alpha, \beta)$ razy.

Największa wspólna wielokrotność

Aby policzyć największą wspólną wielokrotność liczb a i b, sumujemy wszystkie wystąpienia dzielników pierwszych w rozkładzie a oraz b, a następnie odejmujemy te z $NWD(a, b)$. Wynika to stąd, że $NWW(a, b) = \frac{a*b}{NWD(a,b)}$. Przykład: $a = 2 * 2 * 2 * 3 * 5, b = 2 * 2 * 3 * 3 * 7, nwd = 2 * 2 * 3 * 3, nww = \frac{2*2*2*2*3*3*3*5*7}{2*2*3} = 2 * 2 * 2 * 3 * 3 * 5 * 7$. Inaczej mówiąc, jeżeli dzielnik p występuje w rozkładzie a α razy, a w rozkładzie b β razy, to w rozkładzie $NWW(a, b)$ występuje $max(\alpha, \beta)$ razy.

Ilość dzielników liczby

Niech $a = p_1^{a_1} * p_2^{a_2} * ... * p_k^{a_k}$. Wówczas ilość dzielników liczby a to $(a_1+1)*(a_2+1)*...*(a_k+1)$. Dlaczego? Liczba $b = p_1^{b_1} * p_2^{b_2} * ... * p_k^{b_k}$ jest dzielnikiem a wtedy, jeżeli dla każdego p_l, zachodzi $b_l \leqslant a_l$. Dla każdej liczby pierwszej p_l możemy wybrać wykładnik z przedziału $[0, a_l]$ na $a_l + 1$ sposób. Zatem łącznie jest $(a_1+1)*(a_2+1)*...*(a_k+1)$ możliwości wyboru. Przykład: $a = 2^2*3^1$. Dzielniki to $2^0 * 3^0, 2^0 * 3^1, 2^1 * 3^0, 2^1 * 3^1, 2^2 * 3^0, 2^2 * 3^1$. Jest ich istotnie $(2 + 1) * (1 + 1) = 6$.

Suma dzielników

Suma dzielników liczby a to $(p_1^{a_1} + p_1^{a_1-1} + .. + p_1^1 + p_1^0) * (p_2^{a_2} + .. + p_2^1 + p_2^0) * ... * (p_k^{a_k} + .. + p_k^1 + p_k^0)$. Zauważmy, że po wymnożeniu wszystkich nawiasów otrzymamy sumę $(a_1+1)*(a_2+1)*...*(a_k+1)$ składników, które to będą kolejnymi dzielnikami liczby a.

Funkcja Eulera

Funkcja Eulera mówi nam o ilości liczb względnie pierwszych z n, mniejszych od n. Jest dana wzorem:
$\varphi(n) = p_1^{a_1-1} * (p_1 - 1) * p_2^{a_2-1} * (p_2 - 1) * ... * p_k^{a_k-1} * (p_k - 1)$.

7.1.5 Algorytm Euklidesa

Algorytm Euklidesa jest chyba najbardziej znanym algorytmem. Za jego pomocą możemy szybko znaleźć największy wspólny dzielnik dwóch liczb całkowitych a i b. Opiera się on na dwóch faktach. Po pierwsze,

> jeżeli $b|a$, to $NWD(a, b) = b$.

Tego raczej nie trzeba wyjaśniać. Po drugie,

> przedstawmy a jako $a = bt + r$, dla całkowitych t i r. Wówczas $NWD(a, b) = NWD(b, r)$.

Dlaczego? $NWD(a, b) = NWD(b*t+r, b)$. Niech $d = NWD(a, b)$, wtedy $d|b*t+r \wedge d|b \implies d|r$. Stąd $NWD(a, b)|NWD(b, r)$. Pokażę teraz, że $NWD(b, r)|NWD(a, b)$. Niech $c = NWD(b, r)$. $c|b \wedge c|r \implies c|b * t + r$, czyli $c|a$. Stąd $c|NWD(a, b)$. Skoro $NWD(a, b)|NWD(b, r)$ oraz $NWD(b, r)|NWD(a, b)$, to $NWD(a, b) = NWD(b, r)$.

Pokazaliśmy własność rekurencyjną algorytmu Euklidesa, która pozwoli nam szybko policzyć wynik. Będziemy rekurencyjnie wywoływać algorytm, dopóki a nie jest podzielne przez b. Zauważmy, że w każdych dwóch krokach algorytmu co najmniej jedna z liczb zmniejszy się co najmniej o połowę. W związku z tym złożoność algorytmu jest logarytmiczna ze względu na wielkość liczb a i b. Przykładowa implementacja:

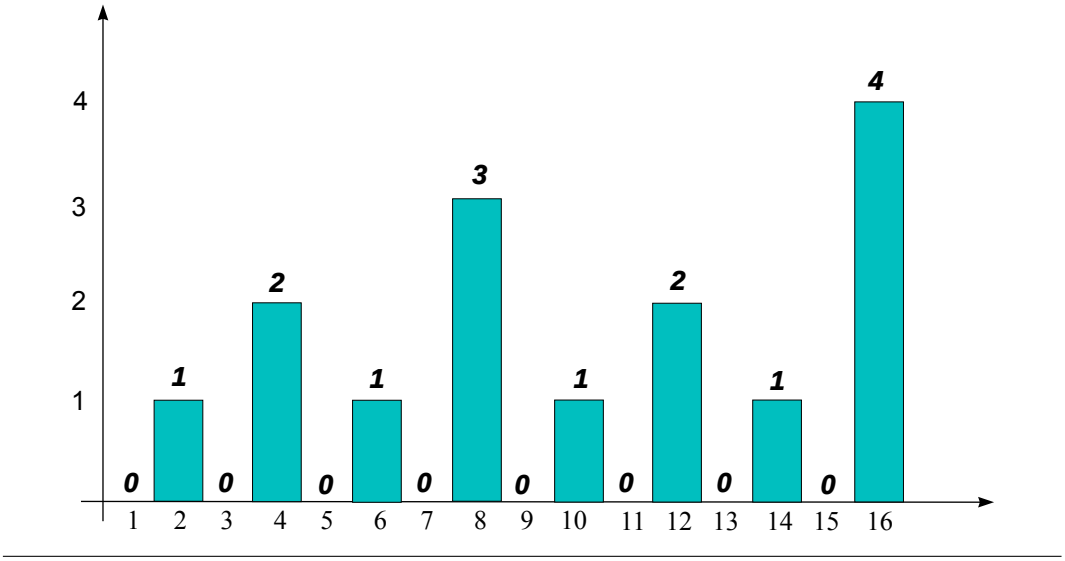
```
1 int NWD(int a,int b) {
2     if(a%b==0) return b;
3     return NWD(b,a%b);
4 }
```

7.1.6 Faktoryzacja $n!$

Rozważanie dużych liczb postaci $n!$ i wykonywanie na nich operacji często jest możliwe jedynie posługując się ich rozkładem na czynniki pierwsze. Takie liczby mogą osiągać rozmiary milionów cyfr, a ich rozkład na czynniki pierwsze pozostanie w akceptowalnym rozmiarze, ponieważ jego

wielkość jest w logarytmicznym stosunku do wielkości liczby. Tak więc, zajmiemy się szybką faktoryzacją liczby postaci $n!$.

Chcemy szybko umieć odpowiedzieć na pytanie, ile razy w rozkładzie na czynniki pierwsze liczby $n!$ wystąpi liczba pierwsza p. Przyjrzyjmy się przykładowi.



Liczba 16! Liczba pierwsza $p = 2$.

Policzenie liczby wszystkich dwójek w rozkładzie na czynniki pierwsze liczby 16! to inaczej suma wysokości wszystkich słupków na powyższym wykresie.

My jednak podejdziemy do tego problemu z nieco innej strony. Sumę wysokości wszystkich słupków możemy również przedstawić jako: liczba słupków o wysokości co najmniej 4 + liczba słupków o wysokości co najmniej 3 + liczba słupków o wysokości co najmniej 2 + liczba słupków o wysokości co najmniej 1. Czyli tak jakby zliczamy wysokości słupków warstwami.

To podejście w tym przypadku jest dobre, gdyż w prosty sposób możemy zliczyć liczbę słupków wyższych od pewnej wysokości. Zauważmy, że liczba słupków o wysokości co najmniej d to inaczej ilość liczb podzielnych przez p^d. A ile jest takich liczb możemy policzyć jako $\lfloor \frac{n}{p^d} \rfloor$, w tym przypadku $\lfloor \frac{16}{2^d} \rfloor$. Suma wysokości wszystkich słupków to zatem:

$$\lfloor \frac{n}{p^1} \rfloor + \lfloor \frac{n}{p^2} \rfloor + ... + \lfloor \frac{n}{p^d} \rfloor \tag{7.2}$$

gdzie d jest największą potęgą, dla której $p^d \leqslant n$. W naszym przykładzie

$$\lfloor \frac{16}{2} \rfloor + \lfloor \frac{16}{4} \rfloor + \lfloor \frac{16}{8} \rfloor + \lfloor \frac{16}{16} \rfloor = 8 + 4 + 2 + 1 = 15 \tag{7.3}$$

i tyle właśnie powinno wyjść.

Dwumiany Newtona

Korzystając z szybkiej faktoryzacji silni, możemy szybko obliczać dwumiany Newtona. Faktoryzujemy liczby $n!$, $k!$ oraz $(n-k)!$, a następnie skracamy powtarzające się czynniki na dole i na górze ilorazu. Wymnażamy pozostałe na górze liczby i otrzymujemy wartość dwumianu.

7.2 Algorytmy równoległe

Wykład: **Algorytmy równoległe** Autor wykładu: **Jacek Tomasiewicz**

7.2.1 Wstęp

Algorytmy równoległe to algorytmy, które pozwalają na wykonywanie w danej chwili więcej niż jednej operacji.

7.2.2 Model obliczeń

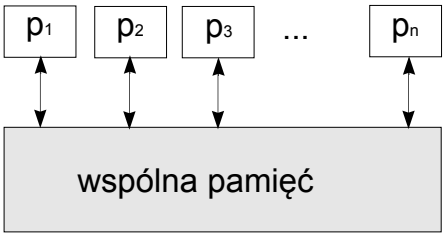
Żeby móc badać algorytmy równoległe, musimy ustalić odpowiedni model obliczeń.

Do tej pory pewnie każdy miał styczność z sekwencyjnym modelem obliczeń – jest to maszyna o dostępie swobodnym (RAM).

My będziemy rozważać równoległy model obliczeń – równoległą maszynę o dostępie swobodnym (PRAM).

Maszyna PRAM:

- składa się z n zwykłych sekwencyjnych procesorów (RAM),
- każdy z nich ma dostęp do wspólnej globalnej pamięci,
- wszystkie procesory mogą jednocześnie odczytywać lub zapisywać do pamięci,
- czas działania mierzony jest liczbą równoległych dostępów do pamięci.



Każdy procesor może w jednej jednostce czasu odwołać się do wspólnej pamięci.

7.2.3 Typy maszyny PRAM

Algorytm dla maszyny PRAM jest algorytmem z:

- jednoczesnym odczytem – **CR** (concurrent-read) – podczas wykonywania algorytmu wiele procesorów może czytać z tej samej komórki,
- wyłącznym odczytem – **ER** (exclusive-read) – nigdy dwa procesory nie mogą czytac z tej samej komórki pamięci,
- jednoczesnym zapisem – **CW** (concurrent-write),
- wyłącznym zapisem – **EW** (exclusive-write).

Rozróżniamy 4 podstawowe typy maszyn:

- EREW PRAM - wyłączny odczyt, wyłączny zapis,
- CREW PRAM - jednoczesny odczyt, wyłączny zapis,
- ERCW PRAM - wyłączny odczyt, jednoczesny zapis,
- CRCW PRAM - jednoczesny odczyt, jednoczesny zapis.

W przypadku CRCW PRAM zakładamy, że jeśli dwa procesory wpisują coś do tej samej komórki, to muszą wpisywać to samo.

Analizując algorytmy równoległe zwracamy uwagę na liczbę użytych procesorów oraz na czas działania.

7.2.4 Przykłady zadań

Warto spróbować rozwiązać kilka problemów.

Występowanie elementu

Sprwdzić, czy w tablicy $A[1..n]$ występuje jakaś jedynka. Maszyna ERCW.

```
1 wynik := false
2 for i := 1 to n in parallel do
3     if A[i] = 1 then wynik := true;
```

Liczba procesorów: $O(n)$, czas działania: $O(1)$.

Na maszynie EREW nie da się tego zrobić w czasie O(1).

Minimalny element

Znaleźć pozycję minimalnego elementu w tablicy $A[1..n]$. Maszyna CRCW.

```
1 for i := 1 to n, j := 1 to n in parallel do
2     if i <> j and A[i] <= A[j] then M[j] := 1
3 for i := 1 to n in parallel do
4     if M[i] = 0 then wynik := i
```

Liczba procesorów: $O(n^2)$, czas działania: $O(1)$.

Pierwsza jedynka

Dana jest tablica zero-jedynkowa $A[1..n]$. Znaleźć pozycję pierwszej jedynki (od lewej). Maszyna CRCW.

```
1 for i := 1 to n, j := i + 1 to n in parallel do
2     if A[i] = 1 and A[j] = 1 then A[j] := 0
3 for i := 1 to n in parallel do
4     if A[i] = 1 then wynik := i
```

Liczba procesorów: $O(n^2)$, czas działania: $O(1)$.

Liczbę procesorów możemy jednak zmniejszyć. Możemy wykonać następujące kroki:

1. Podziel tablicę A na segmenty długości $\sqrt{(n)}$.
2. W każdym segmencie sprawdź, czy jest w nim jedynka.
3. Otrzymujemy ciąg zerojedynkowy C długości $\sqrt{(n)}$ jako wynik kroku 2.
4. Znajdź pierwszą jedynkę w ciągu C.
5. Znajdź pierwszą jedynkę w segmencie odpowiadającym pierwszej jedynce w C.

Liczba procesorów: $O(n)$, czas działania: $O(1)$.

Suma liczb

Obliczenie sumy liczb w tablicy $A[1..n]$. Dla uproszczenia zakładamy, że n jest potęgą 2. Maszyna EREW.

1. Oblicz sumę elementów 1,2, 3,4, 5,6 ... (wykonujemy w czasie stałym za pomocą $O(n)$ procesorów).
2. Otrzymujemy ciąg 2 razy krótszy i wracamy do kroku 1.

```
1 p := n/2
2 while p > 0 do
3     for i := 1 to p in parallel do
4         A[i] := A[i] + A[i+p]
5     p := p/2
6 wynik := A[1]
```

Liczba procesorów: $O(n)$, czas działania: $O(\log n)$.

Ponieważ ciąg za każdym razem zmniejsza się dwukrotnie, to po $\log n$ krokach pozostanie jeden element, który będzie równy sumie wszystkich elementów tablicy A.

7.2.5 Przeskakiwanie

Metoda przeskakiwania będzie pozwalała na tworzenie szybkich algorytmów równoległych dla list (operujemy na wskaźnikach).

Aby równolegle operować na obiektach listy, wygodnie jest związać z każdym obiektem odpowiedzialny za ten obiekt procesor.

Porządek na liście

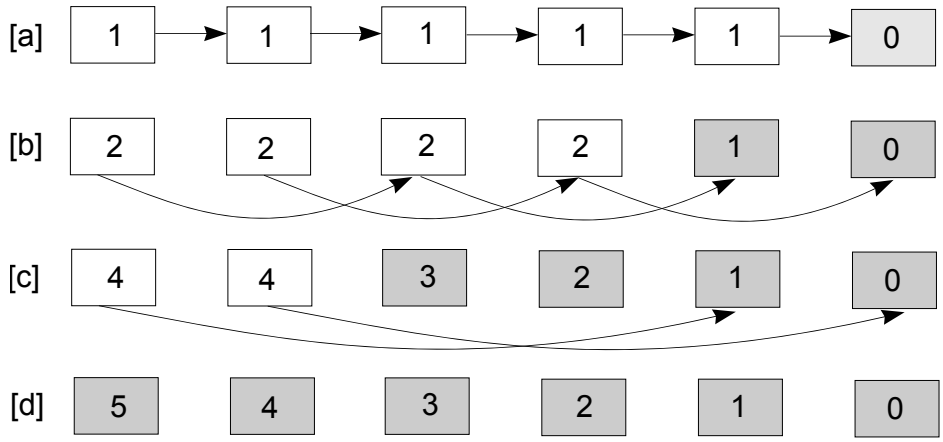
Dane: n elementowa lista L – czyli dla każdego obiektu i znamy wartość $next[i]$, oznaczającą następny obiekt na liście, ostatni element ma $next[i] = NIL$
Szukane: chcemy policzyć wartość $d[i]$, oznaczającą odległość i-tego obiektu od końca listy. Ostatni element powinien mieć $d[i] = 0$, przedostatni $d[i] = 1$ itd.
Maszyna: EREW

Istnieje proste rozwiązanie o złożoności $O(n)$ – n razy przekazujemy odległość w tył poczynając od końca listy.

My jednak zajmiemy się rozwiązaniem o złożoności $O(\log n)$.

```
1 for kazdy procesor i in parallel do
2    if next[i] = NIL then d[i] := 0
3    else next[i] := 1
4 while istnieje obiekt i, taki gdzie next[i] <> NIL do
5    for kazdy procesor i in parallel do
6       if next[i] <> NIL then
7          d[i] := d[i] + d[next[i]
8          next[i] := next[next[i]]
```

Spójrzmy na rysunek przedstawiający rozwiązanie dla przykładowej listy długości 6.



Przebieg algorytmu. Na szaro zaznaczono wierzchołki, które wskazują na NIL.

Dlaczego poprawne?

Dla każdego obiektu i w każdej iteracji pętli z linijki 4. suma wartości d w podliście o początku w i jest równa odległości obiektu i od końca listy L.

Dlaczego czas logarytmiczny?

Zauważmy, że w każdym obiegu pętli w linijce 4. liczba list zwiększa się dwukrotnie (powstaje lista składająca się z obiektów o pozycjach parzystych oraz lista z pozycjami nieparzystymi), natomiast długości tych list zmniejszają się o połowę. W związku z tym po $\log n$ krokach pozostanie n list długości 1.

7.2.6 Jeszcze kilka przykładów

Ciąg sum częściowych

Wykorzystując poprzednie rozwiązanie możemy w prosty sposób obliczać ciąg sum częściowych. Przypomnijmy, że ciąg sum częściowych ciągu $a_1, a_2, ...a_n$ zdefiniowany jest jako ciąg $s_1, s_2, ...s_n$, gdzie $s_i = s_1 + s_2 + ...s_i$.

```
1 for kazdy procesor i in parallel do
2    s[i] := a[i]
3 while istnieje obiekt i, taki gdzie next[i] <> NIL do
4    for kazdy procesor i in parallel do
5       if next[i] <> NIL then
6          s[next[i]] := d[i] + d[next[i]
7          next[i] := next[next[i]]
```

Liczba procesorów: $O(n)$, czas działania: $O(\log n)$.

Korzeń w drzewie

Mamy drzewo o n węzłach. Dla każdego wierzchołka znamy wartość $ojciec[i]$ oznaczającą ojca w drzewie. Należy znaleźć korzeń w drzewie. Maszyna CREW.

```
1 for kazdy procesor i in parallel do
2    if ojciec[i] = NIL then
3       korzen := i
```

Liczba procesorów: $O(n)$, czas działania: $O(1)$.

Na maszynie EREW nie da się tego zrobić w czasie $O(1)$.

7.2.7 Zadania

1. Dla danej n-elementowej listy, której niektóre elementy są białe a niektóre czarne, należy podzielić ją na dwie listy (tylko białe i tylko czarne). Należy zachować kolejność elementów na liście. Maszyna EREW.

2. Sprawdzić, czy ciąg nawiasowy długości n składający się z nawiasów '(' i ')' jest poprawny. Maszyna EREW.

3. Znalezienie korzenia dla każdego wierzchołka w lesie drzew składającym się z n wierzchołków. Maszyna CREW.

7.3 Praktyczne zastosowanie STL, część II

Wykład: **Praktyczne zastosowanie STL, część II** Autor wykładu: **Robert Kozikowski**

7.3.1 Wstęp do wskaźników

Najpierw krótki wstęp do wskaźników.

Każda zmienna, którą mamy w programie, ma pewny adres w pamięci. Wskaźnik jest specjalnym rodzajem zmiennej, która, zamiast trzymać wartości, trzyma adres w pamięci innej zmiennej. Podstawy wskaźników w C++: Deklaracja wskaźnika:

```
1 int *p;
```

Przypisanie adresu do wskaźnika:

```
1 int abc;
2 int *p = &abc;
```

Zmiana wartości pod wskaźnikiem:

```
1 int abc;
2 int *p = &abc;
3 *p = 3;
```

W ten sposób zmienna *abc* otrzymuje wartość 3.
Jeszcze jedną rzecz warto wiedzieć o wskaźnikach:

```
1 pair<int,int> abc = make_pair(0,0);
2 int *p = &abc;
3 p->first = 3;
```

Teraz $abc = (3, 0)$. Jeśli wskaźnik wskazuje na złożoną strukturę to możemy od razu dostać się do któregoś z atrybutów używając operatora $->$.
Oczywiście moglibyśmy napisać:

```
1 pair<int,int> abc = make_pair(0,0);
2 int *p = &abc;
3 (*p).first = 3;
```

Ale przy większych strukturach musielibyśmy dostawiać dużo nawiasów, co może być niewygodne.
Kolejną istotną rzeczą jest to, że nazwa tablicy jest w istocie wskaźnikiem na jej pierwszy element. Poniższy kod powinien rozwiać część wątpliwości:

```
1 int t[10];
2 int main() {
3   int *p = t;
4   int *q = &t[0];
5   if(p == q)
6       cout << "Ale urwal" << endl;
7 }
```

Powyższy kod wypisze nam:

```
Ale urwal
```

Ostatnią interesującą rzeczą jest to, że jeśli dodamy do wskaźnika wartość x (typu *int*), to wskaźnik będzie wskazywał na wartość, który znajduje się x bloków dalej w pamięci. Jest to bardzo użyteczne przy tablicach: Jeśli nazwa tablicy wskazuje na 0-wy element tablicy, to nazwa tablicy+x wskazuje na x-owy element tablicy. Może to być także wykorzystywane do odejmowania dwóch iteratorów od siebie, aby zobaczyć, ile bloków pamięci jest pomiędzy nimi.

7.3.2 Iterator

Większość kontenerów w C++ posiada iterator – typ, który służy na wskazywanie pojedynczych elementów danego kontenera (i chodzenia po sąsiednich elementach). Iterator posiada bardzo wiele analogii do działania wskaźników i w wielu miejscach możemy używać obu zamiennie. Możemy zauważyć analogię do wskaźników, gdzie wskaźnik wskazuje na pojedynczy blok pamięci, iterator wskazuje na pojedynczy element kontenera (oczywiście wskaźnik i iterator mogą wskazywać na to samo).

```
vector<int>::iterator it;  // deklaracja iteratora
```

Czyli ogólniej (*typkontenera*) :: *iterator* Tak stworzony iterator nie wskazuje na nic sensownego, wiemy tylko, że może wskazywać na elementy vectora złożonego z *intów*. Jeśli chcemy, aby wskazywał na konkretny element, możemy to zrobić w następujący sposób:

```
1 vector<int> V;
2 vector<int>::iterator it1, it2;
3 it1 = V.begin();
4 it2 = V.end();
```

Wszystkie kontenery, dla których możemy stworzyć iterator, posiadają przypisane sobie dwa specjalne iteratory – *begin* i *end*. *Begin* wskazuje na pierwszy element kontenera, *end* wskazuje na pierwszy fragment pamięci za końcem kontenera. Taka nietypowa wartość *end* jest spowodowana tym, że gdyby *end* wskazywał na ostatni element, to bardzo niewygodnym byłoby przejście całego kontenera za pomocą iteratorów. Dzięki takim wartościom *begin* i *end* przejście całego vectora może być wykonane bardzo prostą pętlą:

```
1 vector<int> V;
2 for(int i=0; i<5; i++)
3     V.push_back(i);
4 vector<int>::iterator it;
5 for(it = V.begin(); it!=V.end(); it++) {
6     cout << *it << ' ';
7 }
8 cout << endl;
```

Powyższy kod wypisze nam:

```
0 1 2 3 4
```

Powyższej metody można użyć, jeśli chcemy wyrzucić konkretny element z vectora. Jeśli w naszej pętli dodamy linię:

```
1 if(*it == 2) V.erase(it);
```

to otrzymamy wynik:

```
0 1 3 4
```

Uwaga: nie należy używać powyższej operacji do kasowania więcej niż stałej liczby elementów z vectora, gdyż vector realizuje usunięcie w taki sposób, że kasuje dany element i przesuwa wszystkie pozostałe w lewo, więc złożoność tej operacji to $O(n)$.

7.3.3 Algorytmy wykorzystujące iteratory i wskaźniki

W poprzednim wykładzie były tylko algorytmy, które nie wymagają żadnej wiedzy o vectorach i wskaźnikach, ale i tak nieświadomie wykorzystywaliśmy wskaźniki i iteratory. Przypomnijmy sobie, jak korzystaliśmy z funkcji *sort*:

```
1 sort(V.begin(), V.end());
2 sort(t, t+WIELKOSC_T);     \\ t to nazwa tablicy T[WIELKOSC_T];
```

Widzimy teraz, że argumenty, które podaliśmy funkcji *sort*, były to w rzeczywistości iteratory bądź wskaźniki wskazujące początek pamięci, bądź jeden element za końcem pamięci, którą chcieliśmy posortować.
STL dostarcza bardzo dużo algorytmów, ale większość z nich może być zastąpiona jedną prostą pętlą. Można je znaleźć na stronie dokumentacji (punkt 5 - Algorithms):

`http://www.sgi.com/tech/stl/table_of_contents.html`

Na tym wykładzie omówimy tylko te, które są warte pamiętania.
Najpierw weźmy dwie funkcje typowe dla vectorów – *insert* i *erase*. Stosujemy je w następujący sposób:

```
V.insert(it, x);   // wstawia element x przed miejsce, na które wskazuje iterator it
V.erase(it);       // usuwa element, na który wskazuje iterator it
V.erase(it1, it2); // usuwa elementy z przedziału [it1, it2)
```

Oba algorytmy działają w czasie $O(n)$.
Weźmy teraz dwa przydatne algorytmy – *lower_bound* i *upper_bound*. Są to STL-owe implementacje wyszukiwania binarnego – czyli algorytmu, który potrafi w posortowanej tablicy znaleźć dany element w czasie $O(\log n)$ – gdzie n to długość tablicy. Przykład użycia:

```
1 vector<int>::iterator it1 = lower_bound(v.begin(), v.end(), x);
2 vector<int>::iterator it2 = upper_bound(v.begin(), v.end(), y);
```

Lower_bound znajduje nam iterator (bądź wskaźnik) w tablicy pierwszego elementu, który jest większy bądź równy x. *Upper_bound* znajdzie nam indeks pierwszego większego elementu od y.
Więcej będzie widać na przykładzie:
Jeśli $x = 2$, $y = 3$, $V = (1, 2, 2, 3, 3, 4, 5)$ to *lower_bound(x)* wskaże pierwszą 2 (na indeksie 1), a *upper_bound(y)* wskaże 4 (na indeksie 5). Okazuje się, że dzięki temu możemy bardzo łatwo policzyć, ile jest liczb na przedziale $[x, y]$. Będzie to:

```
1 int ile=upper_bound(v.begin(),v.end(),y)-lower_bound(v.begin(),v.end(),x);
```

Odejmujemy tutaj od siebie dwa iteratory, a przy sekcji z iteratorami nie powiedzieliśmy, że jest to możliwe (powiedzieliśmy tylko, że jest to możliwe dla wskaźników). Spowodowane było to tym, że nie wszystkie iteratory można od siebie odejmować. Możemy od siebie odejmować iteratory tylko tych kontenerów, które są zaimplementowane na ciągłej pamięci. Na tym wykładzie jedynym kontenerem spełniającym te warunki jest vector (drugim istotnym jest deque). Przy pomocy powyższej własności można łatwo zapamiętać, jak działają oba algorytmy, wystarczy tak dobrać iteraratory, aby ich różnica dawała ile jest liczb na danym przedziale.

Dosyć użytecznym algorytmem, mimo że łatwo implementowanym samemu, jest unique. Jeśli mamy posortowany vector V, to z użyciem unique możemy usunąć powtórzenia w tym vectorze. Weźmy przykład użycia:

```
vector<int> v;
.....
sort(v.begin(), v.end());
vector<int>::iterator it = unique(v.begin(), v.end());
v.erase(it, v.end());
```

Po powyższym kodzie vector v jest posortowanym vectorem bez powtórzeń. Po użyciu *unique* vector jest zmieniony w taki sposób, że od *v.begin()* do $it - 1$ posiadamy wersję vectora v bez powtórzeń (gdzie it – iterator zwracany przez unique). Chcemy się pozbyć tej ostatniej części vectora, więc usuwamy przedział $[it, v.end()]$. Poniższy kod powinien rozjaśnić powyższe:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void wypisz(vector<int> v) {
    for(int i=0; i<v.size(); i++)
        cout << v[i] << ' ';
    cout << endl;
}
int main() {
    vector<int> v;
    for(int i=0; i<9; i++)
        v.push_back(rand()%8);
    wypisz(v);
    sort(v.begin(), v.end());
    wypisz(v);
    vector<int>::iterator it = unique(v.begin(), v.end());
    wypisz(v);
    v.erase(it, v.end());
    wypisz(v);
    return 0;
}
```

Przykładowy wynik:

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

7.3.4 Lista

List, to STL-owa implementacja listy – w skrócie struktura, w której elementy ze środka możemy usuwać w czasie stałym, ale za to dostęp do dowolnego elementu ze środka listy jest liniowy. Aby korzystać z list, musimy dodać linię:

```
#include <list>
```

Listę możemy przechodzić tylko iteratorem, robimy to tak jak w vectorze, tylko nazwę vector zamieniamy na *list*:

```
list<int> L;
...
list<int>::iterator it;
for(it = L.begin(); it!=L.end(); it++)
    cout << *it << ' ';
```

Lista podstawowych, prostych funkcji listowych:

```
list<int> L
L.size();           // zwraca wielkość listy. UWAGA: czas O(n)
if(L.empty()) {...} // sprawdzamy czy lista jest pusta
L.push_front(x);    // dodanie x na początek listy
L.push_back(x);     // dodanie x na koniec listy
L.pop_front();      // zabranie elementu z początku listy
L.pop_back();       // zabranie elementu z końca listy
                    // UWAGA: gdy lista pusta - błąd kompilacji
L.insert(it,x);     // wstawia element x przed iterator it
L.clear();          // czyści listę
L.sort();           // sortuje liste w czasie O(n log n),
                    // UWAGA: iteratory dalej wskazują na te same elementy.
L.reverse();        // odwrócenie listy
L.remove(x);        // usuwa wszystie wystąpienia x w liście w czasie O(n)
L.erase(it);        // usuwa element na pozycji it.
```

Bardzo często popełnianym błędem przy funkcji *erase* jest to, że zapominamy, iż po jej użyciu iterator, który wskazywał na usunięty element, nie wskazuje teraz na żaden konkretny element listy. Jeśli chcemy skasować jakiś element i mimo to zapamiętać, gdzie jesteśmy (np. przesunąć się do następnego elementu), musimy zrobić to ręcznie, wykorzystując tymczasowy iterator. Przykład:

```
1 list<int> L;
2 list<int>::iterator it = v.begin();
3 list<int>:: iterator it_tmp = it;
4 it++;
5 L.erase(it_tmp);
```

7.3.5 Zbiór i multizbiór

Aby korzystać ze zbioru (*set*) i multizbioru (*multiset*), musimy dodać linię:

```
#include <set>
```

Set to struktura, do której możemy wkładać/wyjmować i wyszukiwać elementy i która trzyma je w każdej chwili posortowane. Dzięki temu większość operacji na *secie* jest w czasie $O(\log n)$. *Set* różni się od *multiseta* tym, że *set* trzyma co najwyżej jeden element danego typu, a *multiset* może trzymać kilka takich samych elementów. Lista podstawowych operacji:

```
set<int> S;
S.insert(x);       // dodanie x do seta
S.erase(x);        // usunięcie wszystkich wystąpień x z seta/multiseta.
S.erase(it);       // usunięcie elementu spod iteratora it
S.clear();         // wyczyszczenie całego seta
if(S.empty()){...} // sprawdzenie czy set jest pusty
S.size();          // liczba elementów seta/multiseta
S.find(x);         // znajduje iterator elementu x, gdy nie ma zwraca S.end()
S.lower_bound(x);  // znajduje iterator pierwszego elementu większego lub równego x
S.upper_bound(x);  // znajduje iterator pierwszego elementu większego od x
```

Set możemy przejść iteratorem tak samo jak listę. Przejdziemy elementy kolejno od najmniejszego do największego. Przy usuwaniu iteratora w *secie* też należy tworzyć tymczasowy iterator, jeśli chcemy zapamiętać miejsce, w jakim jesteśmy.

7.3.6 Mapa

Aby korzystać z mapy (*map*), musimy dodać linię:

```
#include <map>
```

Mapa w pewnym sensie przypomina tablicę, z tą różnicą, że indeksami mogą być dowolne porównywalne obiekty (*string, float, double, int*, itp.). Główną różnicą jest to, że program, zamiast dostać informację jaki jest zbiór indeksów, powiększa zbiór indeksów przy dodawaniu nowych elementów do mapy. Indeks może niektórym kojarzyć się z liczbą naturalną, więc będziemy używać teraz słowa klucz. Zatem mapa jest w istocie zbiorem par (klucz, wartość). Stworzenie mapy:

```
map<string, int> M;
```

Tak stworzona mapa przypisuje różnym stringom wartości *int*. Jedyny warunek, jakie muszą spełniać oba typy, jest taki, że pierwszy z nich musi być porównywalny za pomocą operatora $<$. Dodawanie obiektu do mapy:

```
M["ala"] = 123;
```

Taka operacja przypisuje stringowi 'ala' wartość 123. Jeśli była wcześniej przypisana wartość stringowi 'ala', to zostaje nadpisana przez nową wartość.
Sprawdzanie obiektu w mapie:

```
cout << M["jacekplacek"];
```

Taki sposób wypisze wartość przypisaną stringowi 'jacekplacek'. Jeśli nic nie było przypisane danemu stringowi, to wypisana zostanie wartość zerowa danego typu (czyli taka, jak deklarujemy obiekt danego typu globalnie). Nawet odwołanie się do nieistniejącego obiektu w mapie dodaje ten obiekt do mapy w parze $(x, 0)$ – będzie to istotne, gdy będziemy przeglądać wszystkie indeksy w mapie. Mapę możemy przejść iteratorem:

```
1 map<string, int> M;
2 ...
3 for(map<string,int>::iterator it = M.begin(); it != M.end(); it++) {
4     cout << it->first << ',' << it->second << endl;
5 }
```

Powyższy kod wypisze w kolejnych liniach pary (klucz, wartość). Dla danego iteratora $it->first$ to klucz, a $it->second$ to wartość (dla przypomnienia co znaczy $->$ należy się cofnąć do części ze wskaźnikami). Elementy zostaną wypisane po posortowanym $it->first$. Wszystkie $it->first$ będą różne.

7.3.7 Przeładowanie operatora

W większości poprzednich algorytmów i struktur danych elementy były porównywalne za pomocą znaku $<$ (STL nawet sprawdza, czy elementy są równe, za pomocą znaku $<$, a jest równe b, gdy a nie jest mniejsze od b i b nie jest mniejsze od a). Czyli jeśli dla znanego typu danych lub znanej struktury powiemy jak można je porównać za pomocą znaku $<$, to wszystkie algorytmy i struktury, które wykorzystywały dany operator, będą działać z nowym operatorem (set, map, sort, lower_bound, itd.). Operator możemy przeładować dodając pod deklaracją struktury, a przed wykorzystaniem danego operatora w programie. Możemy to zrobić w następujący sposób:

```
#include <iostream>
#include <algorithm>
using namespace std;
struct butelka {
    string nazwa;
    double procenty;
    double litry;
};
bool operator<(const butelka& a, const butelka& b) {
    return (a.procenty * a.litry) < (b.procenty * b.litry);
}
int main() {
     butelka b1,b2;
     ...
     if(b1<b2) {
     ...
     }
}
```

Powyższy kod „tworzy" operator <, którym możemy porównać dwie butelki. Operator ten ma odpowiedzieć na pytanie: czy obiekt a jest mniejszy od b. Zatem jeśli jest to prawdą, musi zwracać *true*, a jeśli jest to fałszem, musi zwracać *false*. Dla *doubli* już istnieje interesujący nas operator, więc możemy wykorzystać operator porównujący *double* w naszym kodzie. Możemy powyższy kod bardziej uszczegółowić:

```
...
bool operator<(const butelka& a, const butelka& b) {
    if( (a.procenty * a.litry) == (b.procenty * b.litry) ) {
        return a.nazwa < b.nazwa;
    }
    else return (a.procenty * a.litry) < (b.procenty * b.litry);
}
...
}
```

Tak stworzony operator dodaje dodatkowy warunek, że jeżeli dwie butelki są równe względem poprzedniego operatora, porówna je po nazwie. Czasem wygodnie jest dodać porównanie po nieinteresującej nas wartości, żeby dla *seta* obiekty były rozróżnialne.

Czasem chcemy posortować wektor po jakiejś wartości, a następnie posortować ten sam wektor po innej wartości. Oczywiście możemy stworzyć dwie różne struktury, dla każdej stworzyć operator i za każdym razem, jak chcemy posortować po innym kryterium, kopiować cały wektor, ale przy niektórych programach może to być skrajnie niewygodne. Możemy do tego użyć funkcji porównującej:

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  using namespace std;
5  struct butelka {
6     string nazwa;
7     double proceny;
8     double litry;
9  };
10 bool porownaj1(const butelka& a, const butelka& b) {
11    return (a.proceny * a.litry) < (b.procenty * b.litry);
12 }
13 bool porownaj2(const butelka& a, const butelka& b) {
14    if( (a.procenty * a.litry) == (b.procenty * b.litry) ) {
15       return a.nazwa < b.nazwa;
16    }
17    else return (a.procenty * a.litry) < (b.procenty * b.litry);
18 }
19 int main() {
20     butelka b1,b2;
21     vector<butelka> v;
22     sort(v.begin(), v.end(), porownaj1);
23     sort(v.begin(), v.end(), porownaj2);
24 }
```

Wszystkie struktury oraz algorytmy posiadają opcję użycia dowolnej funkcji porównującej. Przy algorytmach zazwyczaj nazwę funkcji porównującej dodaje się po przecinku. Przy niektórych strukturach może być inaczej, po dokładną informację o każdej strukturze należy zajrzeć do dokumentacji.

7.3.8 Stabilne sortowanie

Stabilne sortowanie, czyli *stable_sort*. Zauważyliśmy przed chwilą, że dwa obiekty, które są według STL „równe", nie muszą być w rzeczywistości takie same. *Sort* może dowolnie przestawić obok siebie równe elementy. Jeśli chcemy, aby elementy „równe" były zachowane w takiej samej kolejności, jak były w vectorze przed sortowaniem, należy użyć *stable_sort*.

Część III

Grupa olimpijska

Rozdział 8

Zadania

8.1 Dzień próbny

Zadanie: **Pociąg** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **64 MB** *Dzień 0, 19.09.2010*

Pociąg

Mały Jacuś dostał od mamy pociąg złożony z n wagonów. Po dokładnym obejrzeniu każdego z nich stwierdził, że na odwrocie wagonów znajduje się numer seryjny złożony z małych liter alfabetu angielskiego.

Mały Jacuś może układać wagony w dowolnej kolejności, budując w ten sposób pociąg (zawsze z n wagonów), który po odwróceniu tworzy pewien tekst w. Znudziło mu się to dość szybko, ponieważ chciał zająć się informatyką. W związku z tym wymyślił zadanie.

Poprosił mamusię, aby napisała na kartce pewien tekst t. Tekst w uznał za wzorzec, którego będzie szukał w tekście t. Jacuś chce policzyć sumę wystąpień wszystkich wzorców (zbudowanych z każdej permutacji pociągu) w tekście t.

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n < 10$). W n kolejnych wierszach znajduje się tekst w_i oznaczający numer seryjny i – tego wagonu (długość numeru seryjnego nie przekracza 10^5). W kolejnym wierszu znajduje się tekst t (długość tekstu t nie przekracza 10^6).

Wyjście

W jedynym wierszu wyjścia powinna znaleźć się jedna liczba całkowita, równa sumie wystąpień wszystkich wzorców.

Przykład

dla danych wejściowych:

```
2
ala
ma
alamaalama
```

poprawnym wynikiem jest:

```
3
```

8.2 Dzień pierwszy

Zadanie: **Zegary** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **64 MB** *Dzień 1, 20.09.2010*

Zegary

W Bajtocji mieszka zegarmistrz Gustaw, który od wielu lat naprawia zegary. W jego zakładzie można znaleźć mnóstwo starych zegarów. Interesujące może być to, że wszystkie zegary są wskazówkowe. Każdy z nich posiada 2 wskazówki: bajtogodzinną i bajtominutową. Obie wskazówki poruszają się w prawą stronę. Bajtogodzina w Bajtocji zawsze trwała 100 bajtominut, dlatego też wszystkie zegary posiadają podziałkę na 100 bajtominut. Długość dnia w Bajtocji często się zmieniała, stąd zegary posiadają niekoniecznie taką samą podziałkę na bajtogodziny.

Zegarmistrza chce odwiedzić król, dlatego też Gustaw postanowił zrobić jak najlepsze wrażenie. Stwierdził, że królowi spodoba się, jeśli wszystkie zegary będą wskazywały taki sam czas. W tej chwili każdy z zegarów jest wyłączony. Gustaw nie może przestawiać ręcznie zegarów, gdyż są one bardzo stare i mogłyby ulec zniszczeniu. Może natomiast uruchomić jeden dowolny zegar i poczekać tyle bajtominut, o ile chce przestawić dany zegar. Nie może uruchamiać dwóch zegarów jednocześnie. Pomóż Gustawowi wybrać bajtogodzinę i bajtominutę, którą mają wskazywać wszystkie zegary, aby czas potrzebny na ich przestawienie był jak najkrótszy.

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 10^6$). W n następnych wierszach znajdują się opisy i – tego zegara w postaci trzech liczb całkowitych g_i, m_i, p_i ($0 \leqslant g_i < p_i \leqslant 10^9$), oznaczających odpowiednio liczbę bajtogodzin, wskazywanych aktualnie przez i-ty zegar, liczbę bajtominut oraz wielkość podziałki bajtogodzinnej i-tego zegara.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać dwie liczby całkowite: liczbę bajtogodzin i liczbę bajtominut, które będzie musiał czekać Gustaw.

Przykład

dla danych wejściowych:

```
3
12 20 16
1 10 15
1 10 18
```

poprawnym wynikiem jest:

```
4 90
```

Wyjaśnienie: Wszystkie zegary ustawiamy na 1:10.

Zadanie: **Cykl** Autor zadania: **Adrian Jaskółka**

Dostępna pamięć: **32 MB** *Dzień 1, 20.09.2010*

Cykl

Masz podany skierowany graf z wagami na krawędziach. Twoim zadaniem jest znalezienie cyklu w tym grafie o jak największej średniej wadze krawędzi.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite dodatnie n, m ($2 \leqslant n \leqslant 100, 2 \leqslant m \leqslant 10^4$), oznaczające odpowiednio liczbę wierzchołków oraz krawędzi grafu. W kolejnych m wierszach znajduje się opis kolejnych krawędzi w postaci trzech liczb a, b, c ($1 \leqslant a, b \leqslant n, a <> b, 0 \leqslant c \leqslant 10^6$). Oznacza to, że istnieje krawędź z wierzchołka a do wierzchołka b o wadze krawędzi c. Między każdą parą wierzchołków będzie istniała co najwyżej jedna krawędź w każdą stronę.

Wyjście

W pierwszym i jedynym wierszu wyjścia powinna się znajdować jedna liczba rzeczywista, równa maksymalnej średniej wadze cyklu w podanym grafie. Liczba ta powinna być zapisana z dokładnością do 4 miejsc po przecinku. Możesz założyć, że w grafie zawsze będzie istniał jakiś cykl.

Przykład

dla danych wejściowych:

```
5 6
1 2 6
2 3 2
3 1 3
2 4 1
4 2 5
5 4 100
```

poprawnym wynikiem jest:

```
3.6667
```

Zadanie: **Śmieszny konkurs informatyczny** Autor zadania: **poj.org (tłumaczenie: A. Jaskółka)**

Dostępna pamięć: **32 MB** *Dzień 1, 20.09.2010*

Śmieszny konkurs informatyczny

Olgierd – główny bohater tego zadania, jest tak samo jak Wy uczniem liceum. Stara się rozwiązywać zadania na różnych serwisach, aby zostać finalistą Polskiej Olimpiady Informatycznej.

Kolejnym etapem jego przygotowań jest *ŚKI - Śmieszny Konkurs Informatyczny*. Jak sama nazwa wskazuje, jest to konkurs informatyczny. Na dodatek śmieszny. Konkurs składa się z wielu rund które mogą się na siebie dowolnie nakładać czasowo. Na każdą rundę organizatorzy przygotowali jedno zadanie. Olgierd wprawdzie nie zna jeszcze zadań, ale już ustalił sobie, ile czasu poświęci na każdą rundę, oraz ustalił, że jak raz się zabierze za jakieś zadanie, to będzie je robił bez przerwy tyle czasu ile ustalił. W jednej chwili chłopiec może oddać się jedynie jednemu zadaniu i nie zacznie robić żadnego zadania przed rozpoczęciem danej rundy (przecież nie zna zadań) ani nie ma zamiaru robić zadania po zakończeniu rundy, w której ono było (bo i po co). Jako że zadania na *ŚKI* są bardzo trudne, to czas, który Olgierd poświęci na każdą rundę, jest większy niż połowa czasu trwania całej rundy. Olgierd może robić zadania bez przerwy – nadmiar kawy robi swoje.

Chłopak chciałby wiedzieć, czy może na każde zadanie poświęcić tyle czasu, ile ustalił, ale nie chce marnować swojego cennego czasu na zastanawianie się nad tym, więc powierzył to zadanie konkurencji na *ŚKI* – czyli Wam.

Wejście

W pierwszej linii znajduje się jedna liczba całkowita z, oznaczająca liczbę zestawów danych do rozpatrzenia.

W pierwszej linii każdego zestawu danych znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 2 * 10^5$), oznaczająca liczbę rund *ŚKI*. W kolejnych n liniach opis każdej rundy w postaci trzech liczb całkowitych a_i, b_i, c_i ($0 \leqslant ai < bi \leqslant 10^9, \frac{b_i - a_i}{2} < c_i \leqslant b_i - a_i$), oznaczające, że i-ta runda zaczyna się w a_i-tej jednostce czasu, kończy się w b_i-tej jednostce czasu, oraz Olgierd chce poświęcić na tę rundę c_i jednostek czasu. Suma n we wszystkich zestawach danych nie przekroczy 10^6.

Wyjście

Dla każdego zestawu danych jedno słowo 'TAK', jeżeli Olgierd może zrealizować swój plan, bądź 'NIE' jeżeli jest to niemożliwe.

Przykład

dla danych wejściowych:

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

poprawnym wynikiem jest:

```
TAK
NIE
```

8.3 Dzień drugi

Zadanie: **Najdłuższe rosnące podciągi** Autor zadania: **Łukasz Jocz**

Dostępna pamięć: **32 MB** *Dzień 2, 21.09.2010*

Najdłuższe rosnące podciągi

Twoim zadaniem jest obliczenie liczby najdłuższych rosnących podciągów ciągu A modulo liczba m.

Wejście

W pierwszej linii wejścia znajdują się dwie liczby całkowite n i m ($1 \leqslant n \leqslant 500000$, $1 \leqslant m \leqslant 10^9$), odpowiednio długość ciągu A i liczba m.

W następnym wierszu n liczb całkowitych – kolejne elementy ciągu A ($0 \leqslant a_i \leqslant 10^9$).

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczę całkowitą – liczba najdłuższych rosnących podciągów modulo m.

Przykład

dla danych wejściowych:

```
4 10
3 2 5 4
```

poprawnym wynikiem jest:

```
4
```

Zadanie: **ABC** Autor zadania: **Łukasz Jocz**

Dostępna pamięć: **64 MB** *Dzień 2, 21.09.2010*

ABC

Dane są dwa ciągi znaków X i Y, składające się z liter 'a','b' i 'c'. Należy znaleźć najdłuższy niemalejący wspólny podciąg ciągów X i Y. Inaczej mówiąc, należy znaleźć najdłuższy ciąg, który:

- jest podciągiem ciągów X i Y, czyli da się go otrzymać przez usunięcie pewnych liter z ciągów X i Y;
- jest niemalejący pod względem kolejności liter w alfabecie, czyli przed wystąpieniem litery v nie wystąpi litera o większym kodzie ASCII niż v.

Długości ciągów X i Y nie przekraczają 200000.

Wejście

W pierwszej linii znajdują się dwie liczby całkowite n i m ($1 \leqslant n, m \leqslant 200000$), oznaczające długości ciągów X i Y.

W drugiej linii znajduje się ciąg X, a w następnej Y.

Wyjście

W pierwszym wierszu należy wypisać długość najdłuższego ciągu spełniającego warunki zadania.

Przykład

dla danych wejściowych:

```
5 6
cabbc
bacbcc
```

poprawnym wynikiem jest:

```
3
```

Wyjaśnienie: Najdłuższy ciąg spełniający warunki zadania to „abc”.

Zadanie: **Ciężkie klocki** Autor zadania: **Damian Rusak (informatyka.wroc.pl)**

Dostępna pamięć: **32 MB** *Dzień 2, 21.09.2010*

Ciężkie klocki

Mały Bobbie otrzymał na urodziny zestaw klocków. Jego rodzice wiedzieli dobrze, że nie jest to zbyt oryginalny prezent i gdzieś już słyszeli o dzieciach otrzymujących różne odmiany stymulujących intelektualnie klocków. Postanowili więc tym razem postawić nie na rozmiar, lecz na wagę. Każdy z fascynujących klocków ma swoją wagę, wyrażającą się dodatnią liczbą całkowitą. Żadne dwa klocki nie ważą tyle samo. Na nieszczęście dla Bobbiego, jego rodzice są miłośnikami zagadek logicznych, postanowili więc przerwać jego beztrosko prymitywną zabawę i przygotować dla niego zadanie do wykonania.

Klocki zostały ustawione w rzędzie, jeden obok drugiego. Rodzice wyjaśnili Bobbiemu, że klocki są tak skonstruowane, że jeśli któryś z nich zostanie popchnięty w jedną ze stron, to przewróci wszystkie lżejsze od niego (na zasadzie domina) po tej stronie, aż do pierwszego klocka cięższego niż ten popchnięty bądź do pierwszego miejsca, gdzie jest przewrócony wcześniej klocek. Z uśmiechami na ustach zaproponowali Bobbiemu, że jeśli uda mu się znaleźć sposób na przewrócenie wszystkich klocków w najmniejszej możliwej liczbie "popchnięć", to będzie mógł powrócić do swych poprzednich infantylnych zabaw.

Oczywiście Bobbie ani myśli się tym zajmować, woli oglądać wzorki na dywanie. Ale Ty możesz podjąć wspaniały zamiar i pomóc Bobbiemu poprzez napisanie programu, który odnajdzie żądaną liczbę popchnięć klocków.

Wejście

Pierwsza linia wejścia zawiera liczbę całkowitą n – liczbę klocków. ($1 \leqslant n \leqslant 10^6$). W kolejnej linii znajduje się n różnych liczb całkowitych z przedziału od 1 do 10^9 – wagi kolejnych klocków ustawionych w rzędzie.

Wyjście

Jedyna linia wyjścia powinna zawierać jedną liczbę – najmniejszą liczbę popchnięć klocków potrzebną do przewrócenia ich wszystkich.

Przykład

dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2
```

Zadanie opublikowane dzięki uprzejmości Wrocławskiego Portalu Informatycznego.

8.4 Dzień trzeci

Zadanie: **Guziki** Autor zadania: **Joanna Bujnowska, Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Guziki

Miś Jogi wymyślił sobie nową zabawę. Ułożył na stole kwadrat z n^2 guzików i zastanawia się, ile jest takich prostych przechodzących przez guzik leżący na środku kwadratu, żeby liczba guzików leżących na każdej z tych prostych była równa g. Zakładamy, że guziki są punktami na płaszczyźnie.

Wejście

W pierwszym i jedynym wierszu wejścia znajdują się dwie liczby całkowite n i g ($1 \leqslant g \leqslant n < 2 * 10^6$, n nieparzyste).

Wyjście

W jedynym wierszu wyjścia powinna znaleźć się jedna liczba całkowita równa liczbie prostych z treści zadania. Jeżeli liczba prostych jest większa od 10^{12} wypisz -1.

Przykład

dla danych wejściowych:

```
3 3
```

poprawnym wynikiem jest:

```
4
```

Zadanie: **Pokoje** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Pokoje

W bajtockim hotelu ukrył się tajny szpieg Pituś. Obawiając się, że ktoś zna numer pokoju, w którym się znajduje, poprzekręcał on w nocy niektóre 9 na 6, a niektóre 6 na 9.

W poszukiwaniu szpiega do hotelu przybył agent Dejf. Udało mu się już dowiedzieć, w którym pokoju zakwaterował się Pituś oraz poznać fakt poprzekręcania 6 i 9. Niestety Dejf zna tylko numer pokoju Pitusia przed przekręceniem. Agent zastanawia się teraz, ile pokoi musi sprawdzić, aby mieć pewność, że znajdzie szpiega.

Wiedząc, ile pokoi znajduje się w hotelu oblicz, ile pokoi musi sprawdzić Dejf. Wystarczy, że podasz resztę z dzielenia pokoi przez $10^7 - 3$.

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita h ($1 \leqslant h \leqslant 10^{1000000}$), oznaczająca liczbę pokoi w hotelu. W drugim wierszu wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant h$), oznaczająca numer pokoju, w którym zakwaterował się Pituś.

Wyjście

W pierwszym i jedynym wierszu wyjścia powinna znaleźć się jedna liczba całkowita, równa reszcie z dzielenia przez $10^7 - 3$ pokoi, które musi sprawdzić Dejf.

Przykład

dla danych wejściowych:

```
1000
690
```

poprawnym wynikiem jest:

```
4
```

Zadanie: **Przyjazne punkty** Autor zadania: **acm.sgu.ru (tłumaczenie: Adrian Jaskółka)**

Dostępna pamięć: **64 MB** *Dzień 3, 22.09.2010*

Przyjazne punkty

Na płaszczyźnie mamy n punktów. Każdy punkt ma różną współrzędną x oraz y (nie ma takich dwóch punktów, których współrzędne x-owe są takie same, oraz nie ma takich dwóch punktów, których współrzędne y-owe są takie same).

Mówimy, że dwa punkty są przyjazne sobie nawzajem, gdy prostokąt o bokach równoległych do osi współrzędnych oraz tych dwóch punktach w naprzeciwległych wierzchołkach nie zawiera wewnątrz siebie żadnego innego punktu. Twoim zadaniem jest policzenie liczby par przyjaznych punktów.

Wejście

W pierwszej linii znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 10^5$), równa liczbie punktów. W kolejnych n liniach znajdują się współrzędne kolejnych punktów ($0 \leqslant x, y \leqslant 10^6$).

Wyjście

W pierwszym wierszu wyjścia powinna się znaleźć jedna liczba całkowita, równa liczbie par przyjaznych punktów.

Przykład

dla danych wejściowych:

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

poprawnym wynikiem jest:

```
5
```

8.5 Dzień czwarty

Zadanie: **Dziwna planeta** Autor zadania: **acm.sgu.ru (tłumaczenie: Adrian Jaskółka)**

Dostępna pamięć: **32 MB** *Dzień 4, 23.09.2010*

Dziwna planeta

Pewnego razu gdzieś tam sobie istniała n-wymiarowa przestrzeń. Znajdowała się tam również pewna dziwna planeta. Jedną z jej przedziwnych cech był jej kształt – n-wymiarowy hipersześcian o jednostkowej długości boku. W każdym wierzchołku planety znajdowało się pewne dziwne miasto.

Terytorium tej planety zostało podzielone między trzy wrogie królestwa. Jednak kilka miast ogłosiło swoją niezależność – nazwijmy je neutralnymi. i-te miasto jest niezależne, jeśli $d_1(i) = d_2(i) = d_3(i)$, gdzie $d_j(i)$ oznacza odległość między i-tym miastem oraz stolicą j-tego królestwa. Wszystkie odległości liczone są przy pomocy metryki miejskiej.

Twoim zadaniem jest obliczenie liczby neutralnych miast. Ze względu na to, że wynik może być duży, wystarczy wypisać go modulo $10^9 + 7$.

Wejście

Wejście zawiera trzy linie. W każdej linii znajdują się współrzędne stolicy kolejnego królestwa w postaci liczby binarnej długości n ($1 \leqslant n \leqslant 10^5$).

Wyjście

W pierwszym i jedynym wierszu wyjścia powinna się znaleźć jedna liczba całkowita równa liczbie neutralnych miast modulo $10^9 + 7$.

Przykład

dla danych wejściowych:

```
01
01
10
```

poprawnym wynikiem jest:

```
2
```

Zadanie: **Kupiec** Autor zadania: **Jacek Tomasiewicz**

Dostępna pamięć: **64 MB** *Dzień 4, 23.09.2010*

Kupiec

W Bajtlandii sieć dróg między miastami jest taka, że między dwoma dowolnymi miastami istnieje dokładnie jedna droga (bezpośrednia lub pośrednia).

Pewien kupiec przyjechał do Bajtlandii w celach zarobkowych. Pierwszego dnia chciałby zamieszkać w dowolnym mieście (miasto początkowe), a następnie udać się do wybranego miasta (miasto końcowe). Kupiec na drodze z jednego miasta do drugiego nie może dwa razy odwiedzić tego samego miasta (ze względu na przepisy prawne Bajtalndii). Chciałby jednak wybrać miasto początkowe i końcowe w taki sposób, aby jego zarobek był jak największy.

Kupiec zarabia poruszając się pomiędzy miastami. Dla każdego bezpośredniego połączenia wiemy, ile bajtalarów kupiec zarobi lub straci.

Pomóż wybrać kupcowi dwa miasta tak, aby jego zarobek był jak największy (w szczególności miasto początkowe i końcowe może być tym samym miastem).

Wejście

W pierwszej linii wejścia znajduje się jedna liczba całkowita n ($2 \leqslant n \leqslant 10^6$). W następnych $n-1$ wierszach znajdują się po trzy liczby całkowite a, b, x ($-10^9 \leqslant x \leqslant 10^9$), oznaczające, że istnieje bezpośrednie połączenie między miastami a i b, w którym kupiec zarobi x bajtalarów (ujemne x oznacza stratę kupca).

Wyjście

W pierwszym i jedynym wierszu wyjścia powinna znaleźć się jedna liczba cakłkowita oznaczająca maksymalny zarobek kupca.

Przykład

dla danych wejściowych:

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

poprawnym wynikiem jest:

```
6
```

Zadanie: **Materiały wybuchowe** Autor zadania: **Robert Kozikowski (pomysł z CodeJam)**

Dostępna pamięć: **64 MB** *Dzień 4, 23.09.2010*

Materiały wybuchowe

Dostałeś kontrakt na przewóz materiałów wybuchowych. Dysponujesz n ciężarówkami, każda o pojemności x_i.

Dla każdej ciężarówki musisz zaplanować, jak zapakować ją materiałami wybuchowymi. Niestety każda ciężarówka musi by szczelnie wypełniona materiałami wybuchowymi, gdyż w innym wypadku podczas przewozu materiały mogą ulec uszkodzeniu. Masz do dyspozycji k różnych rodzajów materiałów wybuchowych o różnych wielkościach y_i. Każdego z materiałów możesz zawsze wyprodukować tyle, ile potrzebujesz. Ze względu na szybkość pakowania i rozładowywania ciężarówek zależy ci, aby do zapakowania ciężarówki użyć jak najmniej materiałów.

Policz, ile materiałów użyć dla każdej ciężarówki.

Wejście

W pierwszej linii wejcia znajdują się dwie liczby całkowite n, k ($1 \leqslant n \leqslant 1000, 1 \leqslant k \leqslant 100$), oznaczające odpowiednio ile jest ciężarówek oraz ile jest rodzajów materiałów wybuchowych. W następnych k liniach znajdują się kolejne wielkości kolejnych rodzajów materiałów y_i, ($1 \leqslant y_i < 10^5$). Można założyć że dwa dowolne rodzaje materiałów mają zawsze różne wielkości. W następnych n liniach znajduj się wielkości kolejnych ciężarówek x_i ($10^{10} \leqslant x_i \leqslant 10^{17}$).

Wyjście

Na wyjściu w każdej z kolejnych n linii powinna znaleźć się pojedyncza liczba w_i, mówiąca ile najmniej materiałów wybuchowych należy użyć, aby szczelnie zapakować i-tą ciężarówkę, bądź pojedyncze słowo 'NIE' jeśli jest to niemożliwe.

Przykład

dla danych wejściowych:

```
3 2
10000
10100
10000000000
10000000001
10000000002
```

poprawnym wynikiem jest:

```
990100
NIE
NIE
```

8.6 Dzień piąty

Zadanie: **Najdłuższy, wspólny, rosnący** Autor zadania: **Łukasz Jocz**

Dostępna pamięć: **64 MB** *Dzień 5, 24.09.2010*

Najdłuższy, wspólny, rosnący

Twoim zadaniem jest znalezienie najdłuższego wspólnego rosnącego podciągu danych ciągów A i B.

Wejście

W pierwszej linii wejścia znajdują się dwie liczby całkowite n i m ($1 \leqslant n, m \leqslant 2000$) – długości ciągów A i B.

W następnych dwóch liniach znajdują się ciągi A i B, składające się z dodatnich liczb całkowitych nie większych niż 10^9.

Wyjście

W pierwszym wierszu należy wypisać długość najdłuższego takiego podciągu.

W następnej linii należy wypisać kolejne elementy ciągu oddzielone pojedynczymi spacjami. Jeżeli istnieje wiele poprawnych rozwiązań, należy wypisać dowolne z nich.

Przykład

dla danych wejściowych:

```
6 7
2 2 1 3 2 3
2 1 2 3 1 3 3
```

poprawnym wynikiem jest:

```
4
2 2 3 3
```

Zadanie: **Bubu** Autor zadania: **Joachim Jelisiejew**

Dostępna pamięć: **64 MB** *Dzień 5, 24.09.2010*

Bubu

Bubu wyszło z jaskini szukać pikników, jednak zauważyli je strażnicy parku Jellystone. Strażnicy próbują zagrodzić drogę Bubu, które ucieka do jaskini. Strażnik złapie Bubu, jeśli znajdzie się w tym samym czasie na polanie, na której jest Bubu. Strażnik może odpoczywać na polanach.

Wejście

W pierwszej linii standardwoego wejścia znajdują się trzy liczby całkowite n, m, s, $(1 \leqslant n \leqslant 10^5, 0 \leqslant m \leqslant 2 * 10^5, 0 \leqslant s \leqslant 3 * 10^4)$, oznaczające odpowiednio liczbę polan w lesie, liczbę przesiek między tymi polanami oraz liczbę strażników.

W następnych m liniach znajdują się opisy (dwukierunkowych) przesiek w postaci trzech liczb całkowitych a, b, w, $(1 \leqslant a, b \leqslant n, 1 \leqslant w \leqslant 10^9)$, gdzie a, b oznaczają numery polan, zaś w oznacza czas w sekundach, jaki zajmuje strażnikowi lub Bubu przebycie tej przesieki.

W następnych s liniach podane są numery polan, na których stoją kolejni strażnicy.

W ostatniej linii podany jest numer polany, na której znajduje się Bubu. Jaskinia Bubu jest na polanie 1.

Wyjście

Jeżeli Bubu może dotrzeć do jaskini niezłapany przez strażników, należy wypisać minimalny czas dotarcia Bubu do jaskini, w innym przypadku należy wypisać -1 . Zakładamy, że istnieje ścieżka między jaskinią Bubu a polaną, gdzie się ono znajduje.

Przykład

dla danych wejściowych:

```
2 1 0
1 2 1
2
```

poprawnym wynikiem jest:

```
1
```

Zadanie: **Myszka** Autor zadania: **Adrian Jaskółka**

Dostępna pamięć: **64 MB** *Dzień 5, 24.09.2010*

Myszka

Mała myszka została zamknięta w wielkim labiryncie.

Labirynt ten składa się z n pokoi, między którymi jest m korytarzy. Jednak każdy korytarz ma jakąś szerokość i jeżeli myszka będzie zbyt gruba, to nie będzie w stanie się przez niego przecisnąć.

Zadania nie ułatwia morzący ją głód. W niektórych pokojach znajduje się kawałek przepysznego sera. Jeżeli myszka kiedykolwiek wejdzie do tego pokoju, to nie może się oprzeć i zjada cały kawałek. Zjedzenie kawałka sera może sprawić, że myszce przybędzie na masie i może jej to uniemożliwić korzystania z niektórych korytarzy.

Znając plan labiryntu, szerokość korytarzy, rozmieszczenie kawałków sera, początkowe położenie myszki oraz pokój, który jest wyjściem z labiryntu, odpowiedz na pytanie, jaka może być największa początkowa grubość myszki, tak aby było możliwe wyjście z labiryntu. Zakładamy, że początkowa waga myszki jest większa bądź równa 0.

Wejście

W pierszym wierszu wejścia znajdują się cztery liczby całkowite n, m, s, d ($1 \leqslant n \leqslant 10^6, 1 \leqslant m \leqslant 2 * 10^6, 1 \leqslant s, d \leqslant n, s <> d$), oznaczające kolejno: liczbę pokoi w labiryncie, liczbę korytarzy, początkową pozycję myszki oraz numer pokoju w którym znajduje się wyjście z labiryntu.

W kolejnym wierszu znajduje się n liczb całkowitych $s_1, s_2, ..., s_n$ ($0 \leqslant s_i \leqslant 10^9$), gdzie s_i oznacza, że grubość myszki się zwiększy o s_i, jeżeli zje ona ser z i-tego pokoju (jeżeli $s_i = 0$, to w tym pokoju nie ma sera).

W następnych m wierszach znajduje się opis kolejnych korytarzy. Każdy korytarz opisany jest przez trzy liczby całkowite a, b, c ($1 \leqslant a, b \leqslant n, 1 \leqslant c \leqslant 10^9$), oznaczające, że dany korytarz łączy pokoje a oraz b i myszka może mieć grubość maksymalnie c, aby przecisnąć się przez dany korytarz.

Wyjście

Wyjście powinno zawierać jedną liczbę całkowitą, równą maksymalnej wadze myszki, takiej że myszka może wydostać się z labiryntu, lub -1, jeśli myszka nie może wydostać się z labiryntu.

Przykład

dla danych wejściowych:

```
2 1 1 2
1 0
1 2 5
```

poprawnym wynikiem jest:

```
4
```

Rozdział 9

Rozwiązania

9.1 Dzień próbny

Opracowanie: **Pociąg** Autor opracowania: **Adrian Jaskółka**

Dostępna pamięć: **64 MB** *Dzień 0, 19.09.2010*

Pociąg

Wstęp

Niech n będzie liczbą wagonów, m będzie długością tekstu t oraz s będzie sumą długości numerów seryjnych.

Rozwiązanie brutalne

Zadanie można w dosyć prosty sposób zrobić w złożoności $O(n!sm)$. Przeglądamy wszystkie permutacje wagonów, wyszukując wzorca w tekście brutalnie w złożoności $O(sm)$.

Rozwiązanie wolne

Powyższe rozwiązanie można udoskonalić, wyszukując wzorzec liniowo przy pomocy algorytmu *KMP*, *Rabina-Karpa* bądź *Boyera-Moore'a*. Opisy tych algorytmów można znaleźć w literaturze bądź w Internecie. Rozwiązanie owo działa w złożoności $O(n!(s+m))$.

Rozwiązanie wzorcowe

Przedstawione algorytmy nie korzystają z faktu, że wszystkie wzorce mają takie same długości. Dla rozwiązania wzorcowego jest to bardzo istotne. Można bowiem zmodyfikować algorytm *Rabina-Karpa* tak aby w złożoności $O(w + (t+l)\log t)$ (gdzie t to długość tekstu, w to suma długości wzorców, a l to liczba wzorców) znaleźć wszystkie wystąpienia.

Dalsza część omówienia wymaga znajomości algorytmu *Rabina-Karpa* w postaci podstawowej. Aby rozszerzyć ten algorytm, wpierw znajdujemy hasze wszystkich podsłów tekstu o długości takiej, jaką ma każdy wzorzec, oraz wrzucamy je do mapy M, zliczając ile razy występuje

jaki hasz. Następnie wystarczy zhaszować wszystkie wzorce i dla każdego przy pomocy mapy M stwierdzić, ile razy występuje on w tekście. Jednak zastowowanie tego algorytmu wprost da nam złożoność $O(n!s + (m + n!) \log m)$, czyli gorzej niż drugie rozwiązanie. Widać, że najbardziej czasochłonne jest tutaj liczenie haszy dla wzorców.

Można to zrobić szybciej w złożoności $O(n!n)$ dla wszystkich wzorców. Korzystamy tutaj z faktu że nasze wzorce są popermutowanymi zlepkami tych samych słów. Dokładny sposób, jak to zrobić, zostawię jako ćwiczenie dla Czytelnika. Końcową złożonością naszego rozwiązania jest zatem $O(n!n + (m + n!) \log m)$.

Dodatkowe uwagi

Hasze słów należy zawsze liczyć, trzymać i generalnie robić z nimi wszystko na *long long*-ach. Wiąże się to z tzw. *paradoksem dnia urodzin*, który mówi, że jeżeli losujemy liczby z przedziału od 1 do n, to przeciętna liczba losowań potrzebna do tego, by jakaś się powtórzyła, jest proporcjonalna do $\sqrt{n}$. Wg badań przeprowadzonych przez amerykańskich naukowców przeciętna liczba losowań liczb z zakresu *int*-a, tak aby jakaś się powtórzyła, wynosi około 82000. Tak więc, haszując na *int*-ach, średnio co 82000 różne słowo będzie miało ten sam hasz, co będzie prowadziło do kolizji i błędnej odpowiedzi.

Niestety amerykańscy naukowcy nie dysponowali wystarczająco dobrym sprzętem, aby doprowadzić choćby do jednej kolizji na *long longach*, co jest najlepszym dowodem ich niezawodności.

9.2 Dzień pierwszy

Opracowanie: **Zegary** Autor opracowania: **Robert Kozikowski**

Dostępna pamięć: **64 MB** *Dzień 1, 20.09.2010*

Zegary

Drobne uproszczenie

Pierwszą rzeczą, jaką należy zauważyć jest to, że warto przejść z liczenia czasu z (godziny, minuty) na liczenie czasu w samych minutach. Następnie po policzeniu wyniku przechodzimy ponownie na notację (godziny, minuty). Należy przy tym pamiętać, aby używać zmiennych 64-bitowych.

Rozwiązanie wzorcowe

Minutę wskazywaną przez pewien z zegarów nazwijmy *minutą początkową*. Minutą, którą będą wskazywały wszystkie zegary nazwijmy *minutą końcową*. Pierwszym spostrzeżeniem jest to, że zawsze *minutą końcową* musi być któraś z *minut początkowych*. Dlaczego? Przypuśćmy, że *minuta końcowa* nie jest *minutą początkową*. Wtedy, jeśli byśmy przesunęli *minutę końcową* do najbliższej wcześniejszej *minuty początkowej*, to każdy z zegarów moglibyśmy włączyć na krótszy okres czasu.

Kolejnym spostrzeżeniem jest to, że *minuta końcowa* musi być mniejsza od najmniejszego zegara, gdyż inaczej najmniejszy zegar nie mógłby wskazać danej godziny.

W rozwiązaniu musimy więc sprawdzić, czy *minutą końcową* może być każda z *minut początkowych* mniejszych od wielkości najmniejszego zegara. W tym celu najpierw sortujemy wszystkie *minuty początkowe*. Następnie przypuszczamy, że *minutą końcową* jest najwcześniejsza *minuta początkowa* i liczymy, o ile musimy przesunąć wszystkie zegary. Następnie przechodzimy do kolejnych przypuszczanych wartości *minut końcowych* (wybieramy je według posortowanych wartości *minut początkowych*), dla których możemy łatwo aktualizować wynik.

Przykład: mamy następujące zegary: (5-minutowy, wskazywana minuta 2), (5-minutowy, wskazywana minuta 4), (12-minutowy, wskazywana minuta 5), (7-minutowy, wskazywana minuta 6). Zegary musimy obrócić o kolejno $(0, 3, 9, 3)$, czyli w sumie o 15. Teraz jeśli założymy, że *minutą końcową* jest kolejna (co do wielkości) *minuta początkowa*, to możemy łatwo obliczyć (w czasie stałym) o ile w sumie musimy przesunąć wszystkie zegary. Czas, o jaki musimy przesunąć zegary, które wskazują nową *minutę końcową*, zostanie wyzerowany (gdyż wszystkie zegary dorównujemy do danej wartości, więc zegarów wskazujących daną wartość nie musimy przesuwać), a wszystkie pozostałe zegary będziemy musieli przesunąć o różnicę nowej wartości końcowej i starej wartości końcowej.

Bardziej formalnie – jeśli przechodzimy z minuty końcowej x do minuty końcowej y:
(wynik dla y) = (wynik dla x) + (liczba wszystkich zegarów − liczba zegarów wskazujących y) * (y-x) - (suma wartości, o jakie musieliśmy przesunąć zegary wskazujące y). Wszystko przy założeniu, że $y > x$, oraz że żaden zegar nie wskazuje godziny pomiędzy y i x.

Czyli nowe wartości o jakie musimy przesunąć zegary z przykładu, jeśli założymy, że godziną początkową jest 4, to $(2, 0, 11, 5)$, czyli 18.

Opracowanie: **Cykl** Autor opracowania: **Adrian Jaskółka**

Dostępna pamięć: **32 MB** *Dzień 1, 20.09.2010*

Cykl

Rozwiązanie wzorcowe

Zadanie można rozwiązać wykorzystując wyszukiwanie binarne po wyniku.
Załóżmy, że chcemy stwierdzić, czy istnieje cykl o średniej wadze większej bądź równej S. Jeśli tak, to załóżmy, że ma on k krawędzi o wagach $a_1, a_2, ..., a_k$, których suma wynosi R. Musi on spełniać następującą nierówność, którą nieco przekształcimy:

$R/k >= S$
$R >= S * k$
$a_1 + a_2 + ... + a_k >= S * k$
$a_1 + a_2 + ... + a_k >= S + S + S + ... + S$ (k razy)
$(a_1 - S) + (a_2 - S) + ... + (a_k - S) >= 0$

W związku z tym szukamy takiego cyklu, w którym, po odjęciu od każdej krawędzi wagi S, cykl będzie miał nieujemną sumę wag. Jak stwierdzić, czy istnieje taki cykl? Można od każdej krawędzi grafu odjąć S, a następnie znaleźć w grafie cykl o dodatniej wadze. Można to zrobić na co najmniej 2 sposoby. Używając algorytmu *Floyda-Warshalla* albo algorytmu *Bellmana-Forda*. W zależności od metody otrzymujemy rozwiązanie o złożoności $O(N^3 \log C)$, bądź $O(NM \log C)$.

Opracowanie: **Śmieszny konkurs informatyczny** Autor opracowania: **Adrian Jaskółka**

Dostępna pamięć: **32 MB** *Dzień 1, 20.09.2010*

Śmieszny konkurs informatyczny

Rozwiązanie wzorcowe

W zadaniu było podane pewne założenie: długość czasu, jaki Olgierd ma zamiar poświęcić na każde zadanie, będzie dłuższa niż połowa czasu, jaki trwa runda. Zazwyczaj takie założenia nie są podawane bez powodów (choć jednak nie zawsze) i bez tego założenia zadanie jest niewykonalne. Tak samo jest i tutaj (autorom nie udało się znaleźć rozwiązania szybko działającego dla dowolnych danych). Z warunku $(b_i - a_i)/2 < c_i$ można wywnioskować pewną rzecz: nie ważne, jaki przedział czasu Olgierd wybierze na rozwiązanie zadania, zawsze 'środkowe' jednostki czasu będą musiały zostać zarezerwowane dla tego zadania.

Dla przykładu, jeżeli $a_i = 5, b_i = 17, c_i = 8$, to zawsze przedział czasu $[9, 13]$ będzie musiał być przydzielony dla tego zadania. Skoro dla każdego zadania mamy pewien przedział czasu, który będzie musiał być wykorzystany przez to zadanie, to daje nam to jednoznaczną kolejność, w jakiej Olgierd powinien robić zadania. Teraz wystarczy zachłannie w takiej kolejności przydzielać czas na zadania, starając się je rozpocząć jak najwcześniej, tak aby każde z nich rozpoczęło się po poprzednim zadaniu. Jeżeli nie uda nam się tak przydzielić czasu poświęconego dla tych zadań, aby czas poświęcony na każde zadanie był równy czasowi zaplanowanemu, to powinniśmy wypisać 'NIE'. W przeciwnym przypadku, plany Olgierda są możliwe do zrealizowania i należy wypisać 'TAK'.

9.3 Dzień drugi

Opracowanie: **Najdłuższe rosnące podciągi** Autor opracowania: **Łukasz Jocz**

Dostępna pamięć: **32 MB** *Dzień 2, 21.09.2010*

Najdłuższe rosnące podciągi

Rozwiązanie wzorcowe

Na początku przedstawimy algorytm, obliczający długość najdłuższego rosnącego podciągu ciągu a długości n. Niech $S[i]$ oznacza najmniejszą wartość, jaką kończy się pewien rosnący podciąg długości i. Zauważmy, że tablica $S[i]$ jest rosnąca. Dlaczego? Ponieważ każdy podciąg długości i powstaje przez dodanie elementu do pewnego ciągu długości $i-1$. Ciąg ma być rosnący, więc dodawany element musi być większy niż ostatni element ciągu długości $i-1$. W szczególności, gdy weźmiemy ciąg długości i, w którym ostatni element jest najmniejszy, to będzie istniał ciąg długości $i-1$, w którym ostatni element jest jeszcze mniejszy.

Zastanówmy się, jak zmieni się tablica $S[i]$ w wyniku dodania na koniec ciągu a liczby x. Niech j będzie największym takim indeksem, że $S[j] < b$. Ponieważ tablica $S[i]$ jest rosnąca, więc $S[j+1] \geqslant b$. Wynika stąd, że najdłuższy podciąg, który kończy się elementem b, ma długość $j+1$ (element b można dodać na koniec tylko takiego podciągu długości i, dla którego $S[i] < b$, najdłuższym takim podciągiem jest j – na podstawie tego jak wybieraliśmy j). Trywialnym faktem jest, że w wyniku dodania liczby b nie poprawimy żadnej wartości $S[i]$ dla $i \leqslant j$, ponieważ $S[i] < b$. Zauważmy, że nie poprawimy też żadnej wartości $S[i]$ dla $i \geqslant j+2$. Jest tak dlatego, że najdłuższy ciąg, którego ostatnim elementem jest b, ma długość $j+1$. Natomiast dodając element b na koniec ciągu długości j, otrzymujemy poprawny ciąg długości $j+1$. Wiemy natomiast, że $S[j+1] \geqslant b$. Zatem zawsze poprawimy wartość $S[j+1]$. Aktualizacja tablicy $S[i]$ po dodaniu liczby b polega więc na znalezieniu wyszukiwaniem binarnym indeksu j oraz przypisaniu do $S[j+1]$ wartości b. Będziemy więc kolejno, poczynając od ciągu długości zero, dodawać elementy do ciągu a i aktualizować tablicę $S[i]$. Wynikiem będzie największy indeks tablicy $S[i]$, do którego przypiszemy jakąś wartość.

Rozszerzymy powyższy algorytm tak, aby obliczał jednocześnie liczbę podciągów długości i. Dodając na koniec literę b, tak jak w poprzednim algorytmie, będziemy wyszukiwaniem binarnym znajdować wartość j. Chcemy obliczyć, ile nowych podciągów długości $j+1$, kończących się liczbą b powstało w wyniku dodania liczby b na koniec ciągu a. Jest to równoważne obliczeniu liczby podciągów długości j, kończących się liczbą a taką, że $a < b$. Aby umieć odpowiadać na takie zapytania, musimy rozszerzyć tablicę o dodatkowe informacje. W poprzednim algorytmie nadpisywaliśmy wartości w tablicy $S[i]$. W tym będziemy zapisywać w wektorze „historię nadpisań", czyli w wektorze $S[i]$ będziemy mieć informację, jakimi różnymi elementami kończyły się podciągi długości i i ile jest odpowiednio takich podciągów dla każdego elementu kończącego. Po prostu, zamiast nadpisywać wartości, tak jak w poprzednim algorytmie, będziemy je dodawać na koniec wektora. Elementy w wektorze są nierosnące, co wynika z poprzedniego algorytmu – sytuacja, w której nadpisujemy, zdarza się tylko wtedy, kiedy poprzedni element był nie mniejszy. Zatem, aby znaleźć liczbę podciągów długości j, kończących się liczbą a taką, że $a < b$, musimy znaleźć w wektorze $S[j]$ wszystkie takie a i zsumować liczby podciągów długości j kończących się liczbą a dla wszystkich $a < b$. Zauważmy, że nie jest konieczne przeglądanie całego wektora. Możemy w każdym wektorze policzyć sumy prefiksowe, a następnie, znając najmniejsze takie k, że $S[j][k] < b$, obliczyć liczbę podciągów spełniających powyższe warunki. Indeks k można

znaleźć za pomocą wyszukiwania binarnego. Cały algorytm ma zatem złożoność $O(n \log n)$.

W poniższym pseudokodzie użyliśmy następujących oznaczeń:

- *W.last* - zwraca ostatni element wektora W
- *W.push*(a) - wrzuca na koniec wektora W element a
- $W[i]$ - zwraca i-ty element wektora
- $P = (a, b)$ - para liczb a i b
- *P.l*1 - pierwszy element pary
- *P.l*2 - drugi element pary

```
1  S[0] := (0,1)
2  for i := 1 to n do
3      S[i] := (INF ,0)
4  lic := 0
5  for i := 1 to n do
6      wczytaj(b)
7      j := [ maksymalne j, gdzie S[j].last.l1 < b ] { wyszukujemy binarnie }
8      k := [ maksymalne k, gdzie S[j][k].l1 >= b ]  { wyszukujemy binarnie }
9      ile := ( S[j].last.l2 - S[j][k].l2 ) mod m
10     S[j+1].push( (b, S[j].last.l2 + ile) )
11     if j+1 > lic then
12         lic := j+1
13 wypisz(S[lic].last.l2)
```

Opracowanie: **ABC** Autor opracowania: **Łukasz Jocz**

Dostępna pamięć: **64 MB** *Dzień 2, 21.09.2010*

ABC

Rozwiązanie

Załóżmy, że w naszym rozwiązaniu jest k literek 'a' oraz l literek 'c'. Niech j będzie pozycją, na której wystąpiła k–ta litera 'a' w ciągu X, a $A[j]$ pozycją, na której wystąpiła k–ta litera 'a' w ciągu Y.

Podobnie niech i będzie pozycją, na której wystąpiła l–ta od końca litera 'c' w ciągu X, a $C[i]$ będzie pozycją, na której wystąpiła l–ta od końca litera 'c' w ciągu Y. O ile $i > j$ oraz $C[i] > A[j]$, to w rozwiązaniu może być maksymalnie $min(B_X[j...i], B_Y[A[j]...C[i]])$ liter 'b', gdzie B_X i B_Y jest liczbą liter 'b' w pewnym przedziale ciągu odpowiednio X lub Y. Zatem łącznie najdłuższy wspólny podciąg rosnący będzie miał długość $min(B_X[j...i], B_Y[A[j]...C[i]]) + k + l$. Jeżeli zatem sprawdzimy wszystkie możliwe k i l, obliczymy długość najdłuższego wspólnego rosnącego podciągu, która ma k liter 'a' i l liter 'b', i znajdziemy maksymalną z takich wartości, to otrzymamy rozwiązanie. Zauważmy, że ilość liter 'b' na dowolnym przedziale możemy szybko obliczyć za pomocą sum prefiksowych. Musimy zatem znaleźć wartości k i l, dla których wartość $min(B_X[i] - B_X[j], B_Y[C[i]] - B_Y[A[j]]) + k + l$ jest maksymalna.

Rozbijmy powyższe minimum na dwa przypadki. Niech $M1$ będzie maksimum z wartości:

$$B_X[i] - B_X[j] + k + l \tag{9.1}$$

dla takich k i l, które spełniają $B_X[i] - B_X[j] < B_Y[C[i]] - B_Y[A[j]]$.

Natomiast niech $M2$ będzie maksimum z wartości:

$$B_Y[C[i]] - B_Y[A[j]] + k + l \tag{9.2}$$

dla takich k i l, które spełniają $B_X[i] - B_X[j] \geqslant B_Y[C[i]] - B_Y[A[j]]$. Jeśli obliczymy wartości $M1$ oraz $M2$, to rozwiązaniem zadania będzie $max(M1, M2)$.

Zajmiemy się teraz obliczaniem $M1$. Przekształćmy warunek i wyrażenie (1), tak aby oddzielnie zgrupować wartości, które są zależne od k, a oddzielnie te, które są zależne od l. $M1$ będzie maksimum z wartości:

$$(B_X[i] + l) + (k - B_X[j]) \tag{9.3}$$

dla takich k i l, które spełniają $B_X[i] - B_Y[C[i]] < B_X[j] - B_Y[A[j]]$.

Nasze rozwiązanie będzie polegało na sprawdzaniu każdego l i znajdowaniu dla niego takiego k, które spełnia warunek $B_X[i] - B_Y[C[i]] < B_X[j] - B_Y[A[j]]$ i ma największą spośród wartości $(k - B_X[j])$. W ten sposób zmaksymalizujemy wartość (3), ponieważ dla konkretnego l wartość $(B_X[i] + l)$ jest stała. Niech zatem $(B_X[j] - B_Y[A[j]])$ będzie dla każdego k kluczem w drzewie przedziałowym, a $(k - B_X[j])$ wartością. Zatem nasz problem sprowadza się do znalezienia maksimum na przedziale $((B_X[i] - B_Y[C[i]]), \infty)$. Ostatecznie $M1$ będzie maksymalną z wartości (3) spośród wszystkich l.

Analogicznie przekształcamy warunek i wyrażenie (2). $M2$ będzie maksimum z wartości:

$$(B_Y[C[i]] + l) + (k - B_Y[A[j]]) \tag{9.4}$$

dla takich k i l, które spełniają $B_X[i] - B_Y[C[i]] \geqslant B_X[j] - B_Y[A[j]]$. Podobnie jak przy obliczaniu $M1$, będziemy sprawdzać wszystkie l i wybierać najlepsze k. Tym razem kluczem w drzewie

będzie $B_X[j] - B_Y[A[j]]$, natomiast wartością $(k - B_Y[A[j]])$. Będziemy szukać maksimum na przedziale $(-\infty, B_X[i] - B_Y[C[i]]\rangle$.

Należy też zadbać o to, aby w drzewie przedziałowym były tylko takie k, dla których $j < i$ oraz $A[j] < C[i]$. Będziemy przerabiać algorytm dla l od n do 0. Niech i' będzie $l+1$ od końca literą 'c' w ciągu X. Po każdej iteracji dodamy do drzewa takie k, dla których $i' < j < i$ lub $C[i'] < A[j] < C[i]$. Do tej pory takich wartości k nie było w drzewie, bo nie spełniały warunków $j < i'$ oraz $A[j] < C[i']$ dla $l+1$, natomiast teraz spełniają warunki $j < i$ oraz $A[j] < C[i]$ dla l.

Podsumowując, algorytm wygląda następująco:

1. Obliczamy dla każdego k wartości j i $A[j]$ oraz dla każdego l wartości i oraz $C[i]$. Algorytm obliczania $M1$:
2. Przeglądamy l od n do 0:
   (a) dodajemy do drzewa takie k(kluczem jest $(B_X[j]-B_Y[A[j]])$, a wartością $(k-B_X[j])$), dla których $i' < j < i$ lub $C[i'] < A[j] < C[i]$,
   (b) znajdujemy w drzewie największą wartość na przedziale $((B_X[i]-B_Y[C[i]]), \infty)$ oznaczmy ją c,
   (c) dla danego l najdłuższy ciąg, spełniający warunki zadania, ma długość $c+B_Y[C[i]]+l$. Zatem $M1 := \max(M1, c + B_Y[C[i]] + l)$.
3. Analogicznie obliczamy $M2$.
4. Wypisujemy$(max(M1, M2))$.

Opracowanie: **Ciężkie klocki** Autor opracowania: **Adrian Jaskółka**

Dostępna pamięć: **32 MB** *Dzień 2, 21.09.2010*

Ciężkie klocki

Oznaczenia

Niech $K[i]$ będzie wagą i-tego klocka.
Przez rozbicie będziemy mieli na myśli zepchnięcie wszystkich klocków, o których jest mowa. Przez $DP[a][b]$ będziemy oznaczali wynik dla przedziału klocków $[a, b]$ (przy programowaniu dynamicznym – wyjaśnione dokładniej później)

Rozwiązanie o złożoności $O(n^4)$

Zadanie można bardzo prosto (prosto ideologicznie) rozwiązać w złożoności $O(n^4)$. Do tego celu użyjemy programowania dynamicznego.

Będziemy dla każdego spójnego podciągu klocków obliczać wynik z zadania ($DP[a][b]$ będzie wynikiem dla podciągu zaczynającego się a-tym indeksem i kończącego się b-tym indeksem). Będziemy to robić dla coraz to dłuższych podciągów. Tak więc, wpierw zrobimy to dla n podciągów o długości 1, następnie dla $n-1$ podciągów o długości 2, i tak dalej aż dla 1 podciągu długości n, dla którego wynik jest wynikiem całego zadania.

Jak obliczyć wynik dla jakiegoś podciągu? Wynikiem dla podciągów o długości 1 oraz 2 jest zawsze 1. Gdy chcemy obliczyć wynik dla podciągu dłuższego niż 2, robimy tak: załóżmy, że spójny podciąg zawiera klocki o indeksach $a, a+1, ..., b-1, b$ (wszystkie poniższe operacje będą wykonywane właśnie na tym podciągu – o reszcie zapominamy).

Dla każdego klocka o indeksie i z przedziału $[a, b]$ zakładamy, że właśnie ten klocek jest popychany. Jeżeli popychamy go w lewo, to znajdujemy takie j, że $K[j] > K[i]$, $j < i$, $j >= a$ oraz $i - j$ jest jak najmniejsze (czyli tak naprawdę indeks klocka, na którym i-ty się zatrzyma; jeżeli takie j nie istnieje, to oznacza, że i-ty klocek zepchnięty w lewo popchnie wszystkie na lewo od niego). Tak więc zostaną dwa (bądź też jeden, jeżeli klocek i-ty zepchnie wszystko po jego lewej stronie – czyli nie znajdziemy j) rozłączne spójne przedziały klocków: $a, a+1, ..., j-1, j$ oraz $i+1, i+2, ..., b-1, b$.

Wynik dla tych przedziałów mam już policzony (ponieważ są krótsze od przedziału $[a, b]$, a wynik dla poszczególnych przedziałów liczymy od najkrótszych poczynając). Tak więc, możemy rozbić przedział $[a, b]$ w $DP[a][j] + DP[i+1][b] + 1$ ruchach. Po przejrzeniu wszystkich i wybieramy to najlepsze oraz aktualizujemy wartość $DP[a][b]$. Analogicznie symulujemy popchnięcie w prawą stronę.

Całość ma złożoność $O(n^4)$, ponieważ rozpatrujemy $O(n^2)$ przedziałów, każdy w złożoności $O(n^2)$ (dla każdego i (których jest $O(n)$), musimy znaleźć pierwszy na prawo bądź lewo cięższy klocek od niego – również koszt $O(n)$).

Rozwiązanie o złożoności $O(n^3)$

Przyglądając się rozwiązaniu o złożoności $O(n^4)$ widzimy, że wiele rzeczy robimy tu niepotrzebnie. Przykładowo, szukanie pierwszego większego klocka po lewej i prawej stronie. Ponieważ dla wielu klocków robimy to wielokrotnie, to, aby tego uniknąć, możemy to policzyć na samym początku w złożoności $O(n^2)$. Tym samym złożoność rozwiązania spada do $O(n^3)$.

Rozwiązanie o złożoności $O(n^2 \log n)$ i $O(n^2)$

Powyższe rozwiązania nie wymagały żadnych obserwacji, a tylko znajomości programowania dynamicznego. Aby zrobić to zadanie lepiej, należy zauważyć, że mając pewien przedział $[a, b]$, zawsze opłaca nam się popchnąć najcięższy klocek z tego przedziału. Dowód tego pozostawiamy Czytelnikowi jako proste ćwiczenie.

Mając naszą obserwację, możemy zrobić to zadanie w znacznie lepszej złożoności, korzystając z powyższego dynamika, lecz zamiast rozpatrywać wszystkie możliwe pchnięcia, rozpatrujemy tylko te z najcięższym klockiem.

Korzystając z drzew przedziałowych do wyznaczania maksimum z przedziału, możemy uzyskać złożoność $O(n^2 \log n)$. Zauważmy natomiast, że wyliczając brutalnie w złożoności $O(n^2)$ na początku algorytmu maksimum z przedziału, możemy uzyskać złożoność $O(n^2)$.

Zeszliśmy z $O(n^4)$ do $O(n^2)$ korzystając z preprocesingu oraz prostej obserwacji. Aby zrobić to zadanie dla danych $n \leqslant 10^6$, należy porzucić programowanie dynamiczne i poszukać czegoś lepszego.

Rozwiązanie o złożoności $O(n \log n)$

Korzystając z obserwacji pozwalającej nam uzyskać złożoność $O(n^2)$, możemy skonstruować algorytm, który na pierwszy rzut oka wydaje się być wykładniczy:

```
1 int oblicz_wynik(ciag_klockow T)
2     if rozmiar(T) == 1 then return 1
3     if rozmiar(T) == 0 then return 0
4     x = najwiekszy_klocek(T)
5     int wynik_lewo := oblicz_wynik(popchnij_lewo(x,T))
6     int wynik_prawo := oblicz_wynik(popchnij_prawo(x,T))
7     return min(wynik_lewo, wynik_prawo) + 1
```

gdzie *oblicz_wynik(T)*, liczy wynik dla ciągu klocków T
rozmiar(T) zwraca liczbę klocków w ciągu T
największy_klocek(T) zwraca indeks największego klocka w ciągu T
popchnij_lewo/popchnij_prawo (x,T) zwraca ciąg klocków, jaki powstanie poprzez popchnięcie x-owego klocka w daną stronę ciągu T.

Algorytm wydaje się być wykładniczy, ponieważ głębokość rekurencji może sięgnąć rzędu $O(n)$, a w każdym wywołaniu rekurencyjnym funkcja wywołuje się dwukrotnie, czyli teoretycznie złożoność może sięgnąć rzędu $O(2^n * n)$ ... no właśnie – „teoretycznie".

Ów algorytm, odpowiednio zaimplementowany, może mieć złożoność $O(n)$ lub $O(n \log n)$ (w zależności od sposobu odpowiadania na zapytanie o największy klocek). Na czym polega owa „odpowiedniość"? Na tym, że argumentem wszystkich funkcji nie będzie ciąg klocków w postaci stricte 'ciąg klocków', a w postaci 'przedział, do którego należą klocki'. Drugą rzeczą jest to, że funkcja *największy_klocek*, będzie wykonywana za pomocą drzewa przedziałowego w złożoności $O(n \log n)$ bądź w czasie $O(1)$, z wcześniejszym preprocesingiem o złożoności $O(n \log n)$. By udowodnić, że powyższy algorytm jest szybki, należy pokazać, że łączna liczba wywołań funkcji *oblicz_wynik*, będzie niewielka.

Dowód szybkości powyższego algorytmu:
udowodnimy indukcyjnie, że funkcja oblicz_wynik, wywołana dla ciągu długości n, wywoła siebie samą łącznie (pośrednio lub bezpośrednio) $O(n)$ razy.

- Dla długości ciągu $n = 0$ lub $n = 1$ algorytm działa w czasie stałym, czyli tak naprawdę liniowym dla $n = 0$ i $n = 1$.
- Dla większych n: bierzemy pewien najcięższy klocek i dzielimy zbiór klocków na dwie części o rozmiarach a i b takich że $a + b = n - 1$. Skoro złożoność algorytmu dla $n' < n$ jest liniowa (korzystamy z indukcji), to złożoność dla $n : O(A) + O(B) = O(N)$, czyli jest liniowa.

Tak więc widzimy, że funkcja *oblicz_wynik()* wywoła się maksymalnie $O(n)$ razy, a złożoność każdego wywołania jest uzależniona od złożoności funkcji *najcięższy_klocek()*. Korzystając z wyżej wymienionych metod, otrzymujemy końcową złożoność czasową $O(n \log n)$.

Rozwiązanie wzorcowe o złożoności $O(n \log n)$

Zadanie można zrobić w złożoności $O(n \log n)$, nie korzystając z drzew przedziałowych. Należy zaobserwować, że wynik zadania będzie nie większy niż $\log n$. Aby osiągnąć taki wynik, wystarczy najcięższy klocek spychać na tę stronę, po której jest więcej klocków. Liczba klocków będzie malała co najmniej dwukrotnie, co daje nam górne ograniczenie na wynik równe $\log n$. Oczywiście, postępując zgodnie z tym algorytmem, nie musimy otrzymać optymalnego wyniku.

Teraz, jeżeli wiemy, że wynik nie przekroczy $\log n$, wystarczy w rozwiązaniu wzorcowym ograniczyć głębokość rekurencji do $\log n$ i znajdowanie maksymalnego klocka robić brutalnie. Na każdej głębokości rekurencji przeszukane zostanie w sumie co najwyżej n elementów, tak więc otrzymujemy rozwiązanie o złożoności $O(n \log n)$.

Rozwiązanie o złożoności $O(n)$

Owo zadanie da się rozwiązać w czasie liniowym, lecz wymaga to bardzo skomplikowanego algorytmu pozwalającego liczyć RMQ (maksimum na przedziale) w czasie stałym z liniowym preprocesingiem. Zainteresowanych owym algorytmem odsyłamy do Delty z września 2007 i listopada 2007 (artykuł Jakuba Radoszewskiego).

9.4 Dzień trzeci

Opracowanie: **Guziki** Autor opracowania: **Robert Kozikowski**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Guziki

Oznaczenia

Oznaczenia przyjmujemy takie jak w zadaniu: n - bok kwadratu, który ułożyliśmy, g - liczba guzików, które mają przecinać poszukiwane proste, $\lfloor x \rfloor$ to część całkowita z liczby x.

Spostrzeżenia

1. Łatwiej jest myśleć o zadaniu, jeśli ustalimy, że guzik środkowy ma współrzędną (0,0), a wszystkie pozostałe guziki są punktami kratowymi o współrzędnych z przedziału $[-\lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{2} \rfloor]$, oraz liczba guzików, jakie przecina prosta, jest liczbą punktów kratowych, które przecina prosta. Interesują nas jedynie proste, które przecinają środek układu współrzędnych, czyli takie proste, które mają równanie $y = ax$, oraz prosta pionowa. Oznaczmy, że $n' = \lfloor \frac{n}{2} \rfloor$.

2. Jeśli $g = n$, to wynikiem zawsze jest 4, gdyż są dwie proste $y = x$, $y = -x$, $y = 0$, oraz prosta pionowa, które przecinają n guzików. W dalszej części opracowania możemy założyć, że $g \neq n$.

3. Jeśli jakaś prosta przecina (oprócz punktu (0,0)) x guzików w górnej połowie układu współrzędnych, to przecina także x guzików w dolnej części układu współrzędnych. Niech $q' = \frac{q-1}{2}$ i ograniczmy się do górnej części układu współrzędnych (zauważmy, że zabraliśmy środkowy guzik).

4. Niech $p : y = a_p x$ będzie dowolną prostą przechodzącą przez 1. ćwiartkę układu współrzędnych. Jeśli p przecina x guzików, to także prosta $p' : y = -a_p x$ przecina x guzików, co jest oczywiste. Dzięki temu możemy ograniczyć się tylko do 1. ćwiartki układu współrzędnych, jeśli będziemy pamiętali, aby wynik pomnożyć przez 2.

5. Jeśli liczba guzików $g = -1$, to prostych będzie nieskończenie wiele.

Rozwiązanie wzorcowe

Lemat.1. *Prosta $p : y = ax$ przecina punkt kratowy (s, r), r i s całkowite wtedy i tylko wtedy gdy można zapisać ją w postaci $p : y = \frac{r}{s}x$. Ponadto jeśli r i s są względnie pierwsze, to wszystkie punkty kratowe przecinane przez prostą p możemy zapisać w postaci $(k * r, k * s)$, gdzie k jest liczbą całkowitą.*

Dowód.

Jeśli prosta $p : y = ax$ przecina punkt (s, r), wtedy możemy zapisać równanie $r = a * s$, czyli $a = \frac{r}{s}$. Załóżmy teraz, że r i s są względnie pierwsze. Przypuśćmy, że prosta p przecina punkt kratowy w punkcie niebędącym postaci (kr, ks), niech będzie to (a, b). Wtedy, jak powiedzieliśmy wcześniej, możemy ją zapisać w postaci $y = \frac{b}{a}x$, $\frac{b}{a} \neq \frac{r}{s}$, a zatem prosta postaci $y = \frac{b}{a}x$ nie jest prostą p. ■

Weźmy teraz dowolną prostą $(p : y = \frac{r}{s}x)$, gdzie r i s są względnie pierwsze. Zauważmy, że prosta p przecina guziki we współrzędnych $(s, r), (2s, 2r), (3s, 3r), ...$, co powiedzieliśmy wcześniej. Zauważmy teraz, że liczba przecinanych guzików o współrzędnych mniejszych niż n', to $\left\lfloor \frac{n'}{max(r,s)} \right\rfloor$, gdyż współrzędna x przekroczy n' w punkcie będącym $\left\lfloor \frac{n'}{s} \right\rfloor + 1$ z kolei (bo x skacze co s), a współrzędna y przekroczy n' w punkcie będącym $\left\lfloor \frac{n'}{r} \right\rfloor + 1$ z kolei (bo y skacze co r). Zauważmy teraz, że prosta $(p_1 : y = \frac{r}{s}x)$ przecina tyle samo punktów co prosta $(p_2 : y = \frac{s}{r}x)$, a przypadek gdy $r = s$ wyeliminowaliśmy (zachodzi on wtedy gdy $q = n$), to możemy teraz założyć, że $r > s$, jeśli będziemy pamiętać, aby pomnożyć wynik przez 2.

Chcemy, aby prosta p przecinała guziki w dokładnie q‘ miejscach. Założyliśmy, że $r > s$, zatem interesują nas takie r, dla których $\left\lfloor \frac{n'}{r} \right\rfloor = q'$. Bedą to wszystkie liczby całkowite z przedziału : $(\frac{n'}{q'+1}, \frac{n'}{q'}]$, co łatwo sprawdzić. Dla każdego r interesują nas tylko takie s, że $s < r$ (co założyliśmy wyżej) oraz s i r są względnie pierwsze (gdyż, jeśli nie są, to istnieje wcześniejszy punkt przecinany przez p). Tak naprawdę nie chcemy poznać wszystkich takich s, ale ich liczbę. Istnieje funkcja *phi* (wykład z teorii liczb), która dla danego x daje liczbę liczb mniejszych od x i względnie pierwszych z x. Zatem wynikiem będzie suma $phi(r)$ dla wszystkich r z przedziału $(\frac{n'}{q'+1}, \frac{n'}{q'}]$. Warto powiedzieć, że przy niektórych implementacjach rozkładanie na czynniki każdej liczby może być za wolne i można skorzystać z własności *phi*: Jeśli a i b są względnie pierwsze, to $phi(a * b) = phi(a) * phi(b)$. Możemy wykorzystać to do wyznaczania dla każdej liczby z przedziału $[2, n]$ jej *phi* za pomocą programowania dynamicznego (np. wydzielając (być może w potędze większej niż 1) zawsze jej największy dzielnik pierwszy, który możemy poznać z faktoryzacji).

Opracowanie: **Pokoje** Autor opracowania: **Robert Kozikowski**

Dostępna pamięć: **32 MB** *Dzień 3, 22.09.2010*

Pokoje

Oznaczenia

Oznaczenia z treści zadania: h – liczba pokoi w hotelu, n – numer pokoju Pitusia. Przypomnijmy, że sufiks napisu s to napis złożony z ostatnich znaków s (być może wszystkich).

Rozwiązanie wzorcowe

W zadaniu musimy policzyć, na ile sposobów możemy przestawić 6 i 9 w numerze pokoju, tak aby numer pokoju pozostał mniejszy od numeru maksymalnego pokoju (czyli liczby wszystkich pokoi).

Jeśli długość h jest większa niż długość n, to rozwiązaniem będzie 2^x, gdzie x to liczba 6 i 9 w numerze pokoju – możemy w numerze pokoju zmieniać 6 i 9 na dowolne sposoby.

Jeśli długość h jest mniejsza niż n, to rozwiązaniem będzie 0 – żaden numer pokoju nie jest możliwy.

Pozostaje przypadek, gdy długość h jest równa długości n. Policzmy najpierw (liniowo) wszystkie możliwe potęgi 2^k mod 9999997 dla $k \leqslant$ długość n.
Idźmy znak po znaku w liczbie h (znak z_h, h – hotel) i liczbie n (znak z_p, p – pokój):
Jeśli z_p jest różny od 6 lub 9 wykonaj:

1. Jeśli $z_h = z_p$, idziemy dalej i nic nie zmieniamy – w żaden sposób nie wpływa to na wynik.
2. Jeśli $z_h < z_p$ to wtedy zwracamy 0, gdyż nawet najmniejsza możliwość pokoju z samymi 6 jest niemożliwa.
3. Jeśli $z_h > z_p$, to wtedy wszystkie ustawienia dalej są możliwe, gdyż nawet możliwość pokoju z samymi 9 jest mniejsza od numeru maksymalnego pokoju. Zwracamy 2^x, gdzie x to liczba 6 i 9 w sufiksie zaczynającym się od z_p.

Jeśli z_p jest równe 6 lub 9 wykonaj:

1. Jeśli z_h jest mniejsze od 6, to wtedy żadne ustawienia dalej nie są możliwe, gdyż nawet wersja pokoju z samymi 6 jest niemożliwa, więc zwróć 0.
2. Jeśli z_h jest równe 6, to kontynuuj algorytm, gdyż na danej pozycji w numerze pokoju musi być 6.
3. Jeśli z_h jest z przedziału $[7, 8]$, to zwróć 2^x, gdzie x to liczba 6 i 9 w sufiksie zaczynającym się od z_p, bo wiemy, że na danej pozycji musi stać 6, ale także wiemy, że wszystkie możliwości przestawień 6 i 9 na dalszych pozycjach dadzą wynik mniejszy niż numer maksymalnego pokoju.
4. Jeśli z_h jest równe 9, to do wyniku dodaj 2^x, gdzie x to liczba 6 i 9 w sufiksie zaczynającym się od z_p, czyli przypuść, że na danej pozycji stoi 6, wtedy wszystkie możliwości przestawień 6 i 9 dalej są możliwe) oraz dodatkowo kontynuuj algorytm (postaw na danej pozycji 9).

Powyższy algorytm działa w czasie liniowym, gdyż każdy znak odwiedzimy tylko raz.

Opracowanie: **Przyjazne punkty** Autor opracowania: **Adrian Jaskółka**

Dostępna pamięć: **64 MB** *Dzień 3, 22.09.2010*

Przyjazne punkty

Rozwiązanie wzorcowe

Metodą wymaganą do rozwiązania tego zadania była technika „Dziel i zwyciężaj". Metoda ta polega na rozwiązaniu problemu według schematu:

1. Otrzymujemy pewien problem dla pewnych danych.
2. Jeżeli dane są tak małe, że wynik dla nich jest ustalony lub prosty do obliczenia, to zwracamy ten wynik; w przeciwnym wypadku:
3. Dzielimy podany problem na mniejsze podproblemy dla mniejszych danych (najczęściej na dwie równe części).
4. Liczymy wynik dla mniejszych części.
5. Scalamy otrzymane wyniki, uzyskując wynik dla pierwotnego problemu.

Przykładem algorytmu działającego wg takiego schematu jest algorytm sortowania przez złączanie. Jeżeli chcemy posortować ciąg liczb, to dzielimy go na dwa równe ciągi, które sortujemy rekurencyjnie, a następnie możemy w prosty sposób złączyć dwa posortowane ciągi, uzyskując jeden posortowany. W przypadku gdy ciąg, który chcemy posortować, ma długość 1, zwracamy po prostu ten ciąg – nie da się już go podzielić.

Rozwiązanie naszego zadania będzie działało w następujący sposób:

1. Dostajemy zbiór punktów.
2. Jeżeli do naszego zbioru należy jeden punkt, to zwracamy 0.
3. Dzielimy prostą pionową podany zbiór na dwa równe zbiory.
4. Liczymy liczbę par punktów przyjaznych w każdym z mniejszych zbiorów i dodajemy je do wyniku.
5. Liczymy liczbę par punktów przyjaznych, takich że jeden punkt jest w jednym zbiorze, a drugi w drugim i dodajemy je do wyniku.
6. Zwracamy wynik.

W powyższym rozwiązaniu jedyną niejasną rzeczą do zrobienia może być to, jak szybko policzyć liczbę par punktów przyjaznych, należących do różnych połówek. Dalsza część omówienia będzie temu poświęcona.

Liczenie par przyjaznych punktów

Mamy dwa zbiory punktów. Nazwijmy je lewym i prawym zbiorem, a punkty należące do tych zbiorów lewymi i prawymi punktami. Pokażemy algorytm liczący pary przyjaznych punktów, takich że punkt, należący do lewego zbioru ma większą współrzędną y, od tego należącego do prawego zbioru. Aby obliczyć pary punktów, takie że punkt należący do lewego zbioru ma mniejszą współrzędną y od tego z prawego zbioru, wystarczy odwrócić symetrycznie punkty względem prostej pionowej i wykonać jeszcze raz ten algorytm.

Tak więc, wpierw dla każdego punktu po prawej stronie obliczamy maksymalną współrzędną y, jaką może mieć punkt z nim przyjazny w lewym zbiorze. Wystarczy znaleźć pierwszy nad nim prawy punkt o mniejszej współrzędnej x, co można prosto zrobić korzystając ze stosu. Dokładną metodę pozostawiamy jako ćwiczenie dla Czytelnika.

Aby obliczyć, ile jest punktów po lewej stronie, które tworzą relację przyjaźni z danym punktem po prawej stronie, można skorzystać z takiej obserwacji: jeżeli punkt po prawej stronie, dla którego aktualnie obliczamy liczbę punktów z nim przyjaznych po lewej stronie, ma współrzędne (x, y), oraz po stronie lewej istnieją punkty o współrzędnych (x_1, y_1) i (x_2, y_2), takie że $y < y1, y < y2, y1 < y2, x2 < x1$, to punkt o współrzędnych (x_2, y_2) nie będzie tworzył relacji przyjaźni z punktem (x, y) ani żadnym innym prawym punktem położonym poniżej (x, y) (mówimy wtedy, że punkt (x_1, y_1) przysłania punkt (x_2, y_2). To prowadzi do tego, że jeżeli posortujemy po współrzędnej x punkty lewe przyjazne z pewnym prawym punktem, to będą one również posortowane po współrzędnej y. Pozwala nam to sformułować następujący algorytm:

Będziemy mieli pewną strukturę, na której będziemy trzymali nieprzysłonięte dotychczas punkty z lewego zbioru. Przeglądajmy wszystkie punkty w kolejności malejącej współrzędnych y.

Jeżeli aktualnie rozważany punkt jest po lewej stronie, to oznacza, że być może przysłania on pewne inne punkty po lewej stronie. Tak więc należy usunąć wszystkie te punkty (jako że przeglądamy punkty w porządku malejącej współrzędnej y, to aby pewien punkt był przysłonięty, wystarczy, że ma mniejszą współrzędną x). Po usunięciu przysłoniętych punktów należy dodać aktualny do zbioru, ponieważ nie jest on jeszcze przysłonięty.

Jeżeli przy przeglądaniu punktów natrafimy na punkt z prawego zbioru, to będziemy chcieli policzyć z iloma punktami z lewego zbioru tworzy on relacje przyjaźni. Korzystamy przy tym z naszej struktury do trzymania punktów z lewej strony. Wystarczy sprawdzić, ile punktów w niej ma odpowiednią współrzędną y (policzyliśmy maksymalną współrzędną y, jaką może mieć punkt po lewej, a minimalną współrzędną będzie po prostu współrzędna aktualnego punktu).

W powyższym algorytmie brakuje opisu struktury do lewych punktów. Wszystkie potrzebne operacje w szybkim czasie będą obsługiwane przez stos stworzony na STL-owym vectorze (w c++). Otóż elementy na naszym stosie będą posortowane zarówno po współrzędnej x, jak i y, więc usuwanie oraz dodawanie elementów będzie miało czas zamortyzowany stały, a sprawdzenie, ile punktów ma odpowiednią współrzędną y, można wykonać za pomocą wyszukiwania binarnego.

Tak więc, w naszym algorytmie dziel i zwyciężaj 'scalanie' wyników dla połówek, ma złożoność $O(n \log n)$, co daje całkowity czas działania programu $O(n(\log n)^2)$.

9.5 Dzień czwarty

Opracowanie: **Dziwna planeta** Autor opracowania: **Adrian Jaskółka**

Dostępna pamięć: **32 MB** *Dzień 4, 23.09.2010*

Dziwna planeta

Przedstawienie zadania

Problem zamaskowany w treści zadania jest następujący:
Mamy trzy liczby binarne A, B, C (wszystkie o takiej samej długości n) i naszym zadaniem jest obliczenie, ile jest liczb binarnych X o długości równej n, takich że liczba bitów ustawionych na 1, liczb: $AxorX$, $BxorX$, $CxorX$ jest sobie równa (liczbę bitów ustawionych na 1 dla $XxorY$ będziemy nazywali odległością między X i Y). Operacja *xor* dla dwóch bitów zwraca 1, gdy są różne oraz 0 gdy są takie same. Operacja *xor* dla dwóch liczb binarnych A i B zwraca liczbę binarną C, taką że i-ty bit C jest *xorem* i-tego bitu A oraz i-tego bitu B. Tak więc za wartość i-tego bitu C odpowiadają tylko i-ta wartość A oraz i-ta wartość B. Tak więc liczby A, B oraz C z wejścia można rozpatrywać jako trójki bitów z pozycji od 1 do n. Mamy $2^3 = 8$ możliwych trójek bitów:

```
   A B C
1) 0 0 0
2) 0 0 1
3) 0 1 0
4) 0 1 1
5) 1 0 0
6) 1 0 1
7) 1 1 0
8) 1 1 1
```

Zliczmy teraz liczebność każdej trójki.
Przykładowo:

```
A = 001001
B = 100010
C = 110011
<0,0,0> występuje 1 raz (na 4 pozycji)
<0,0,1> występuje 1 raz (na 2 pozycji)
<0,1,1> występuje 2 razy (na 1 i 5 pozycji)
<1,0,0> występuje 1 raz (na 3 pozycji)
<1,0,1> występuje 1 raz (na 6 pozycji)
```

Rozwiązanie wolne

Możemy teraz opisać algorytm działający w złożoności $O(n^8 \log P)$:
dla każdej z 8 rodzajów trójek określamy na ilu pozycjach z danej trójki w liczbie X umieszczamy 0, a na ilu 1. Powiedzmy, że liczebność każdej trójki wynosi L_i oraz liczba bitów, na których umieszczamy jedynkę, wynosi dla każdej trójki K_i. Otrzymujemy teraz pewien X, dla którego

tak naprawdę nie wiemy dokładnie, na której pozycji jest jaki bit, tylko wiemy dla każdego rodzaju trójek, ile jest jedynek, a ile zer na miejscach zawierających tą trójkę. Do wyniku należy dodać więc $(\binom{L1}{K1} * \binom{L2}{K2} * ... * \binom{L8}{K8})$ (mając L pozycji i chcąc postawić na K jedynki, możemy to zrobić na $\binom{L}{K}$ sposobów oraz przyznawanie pozycji dla każdej trójki jest niezależne). Szybkie obliczanie symbolu Newtona opiszemy później.

Rozwiązanie szybsze

Aby polepszyć powyższe rozwiązanie należy dokonać kilku obserwacji. Jedną z nich jest to, że tak naprawdę trójka <0,0,0> jest taka sama jak <1,1,1>, <0,0,1> taka sama jak <1,1,0>, <0,1,0> taka sama jak <1,0,1>, <0,1,1> taka sama jak <1,0,0>. Można to tak traktować, ponieważ wybranie jedynki w np. <1,0,1> odpowiada wybraniu zera w <0,1,0> oraz na odwrót: wybranie zera w np. <1,0,1> odpowiada wybraniu jedynki w <0,1,0>. Przekształćmy więc <1,1,1> w <0,0,0>, <1,1,0> w <0,0,1>, <1,0,1> w <0,1,0> oraz <0,1,1> w <1,0,0>. Mając teraz 4 rodzaje trójek, możemy zastosować rozwiązanie takie jak dla 8 trójek i uzyskamy rozwiązanie $O(n^4 \log P)$.

Rozwiązanie wzorcowe

Kolejną obserwacją prowadzącą nas do rozwiązania wzorcowego jest to, że niezależnie od tego, jaki bit wybierzemy dla pozycji zawierającej trójki <0,0,0>, to nie zmieni to różnic między odległościami pomiędzy A i X, B i X oraz C i X. Tak więc możemy usunąć tę trójkę i końcowy wynik przemnożyć przez 2^{L_1}, gdzie L_1 jest liczbą trójek <0,0,0>.

Możemy ponownie zastosować początkowy algorytm, uzyskując złożoność $O(N^3 \log P)$. Popatrzmy teraz na te trójki, które nam zostały:

```
<0,0,1>
<0,1,0>
<1,0,0>
```

Załóżmy, że każdy bit X ustawiliśmy na 0, oraz że liczebność powyższych trójek jest kolejno L_1, L_2, L_3.

Oznacza to, że odległość A od X wynosi L_3, B od X wynosi L_2 oraz C od X to L_1. W ogólnym wypadku, jeżeli K_i jest liczbą pozycji i-tej trójki, ustawionych na jeden, to odległości są następujące:

```
odl A od X = L3 - K3 + K2 + K1
odl B od X = L2 - K2 + K3 + K1
odl C od X = L1 - K1 + K3 + K2
```

Dążymy do tego, aby te odległości były sobie równe. Aby tak było, to następujące równanie musi być spełnione:

```
L3 - K3 + K2 + K1 = L2 - K2 + K3 + K1
L3 - K3 + K2 = L2 - K2 + K3
2 * K2 = L2 - L3 + 2 * K3
K2 = (L2 - L3 + 2 * K3) / 2
```

Wiemy, że L_2 oraz L_3 mamy ustalone, ponieważ są to liczebności trójek, natomiast liczby K_3 nie mamy ustalonej. Oznacza to, że jeżeli ustalimy wartość K_3, to aby odległości między A i X oraz B i X były równe, to K_2 musi być równe $\frac{L2-L3+2*K3}{2}$. Podobny wzór można wyznaczyć dla liczebności K_1 względem K_3. Jeżeli okaże się, że K_1 bądź K_2 są niecałkowite, to oznacza

to, że wynikiem jest 0, gdyż bez względu na wartość K_3, K_1 i K_2 będą niecałkowite. Tak więc, wystarczy przejść po każdej możliwej wartości K_3, wg wzorów wyznaczyć wartość K_1 oraz K_2 i dodać do wyniku wartość $(\binom{L1}{K1} * \binom{L2}{K2} * \binom{L3}{K3})$. Otrzymujemy więc algorytm o złożoności $O(N \log P)$.

Pokazuje nam to również, że powyższy algorytm o złożoności $O(N^3 \log P)$ tak naprawdę ma złożoność $O(N^3 + N \log P)$, ponieważ liczenie symbolu Newtona będzie wykonywane co najwyżej liniowo wiele razy.

Liczenie symbolu Newtona

Teraz opiszemy jak szybko policzyć symbol Newtona. Wg wzoru $\binom{N}{K} = \frac{N!}{(N-K)!*K!}$ brutalne liczenie silni byłoby zbyt wolne, więc na początku programu liczymy wszystkie wartości $M!$ dla $M = 0...M$. Mając wyznaczone te wartości, w policzeniu $\binom{N}{K}$ przeszkadza nam dzielenie. Musimy bowiem podzielić jedną liczbę przez inną modulo liczba pierwsza P. Można to zrobić korzystając z tego, że:
$\frac{A}{B} = A * B^{-1}$

Tak więc musimy teraz pomnożyć liczbę A z odwrotnością liczby B. Jak policzyć odwrotność? Można przy pomocy rozszerzonego Euklidesa (opisane w wykładzie z Teorii liczb). Inną metodą jest skorzystanie z małego twierdzenie Fermata, które mówi, że:
$a^{p-1} = 1 \mod p$ dla dowolnego a oraz dla dowolnego p, które jest liczbą pierwszą.
Dzieląc obustronnie przez a, otrzymujemy:
$a^{p-2} = a^{-1} \mod p$
Tak więc, aby otrzymać odwrotność a mod p, wystarczy policzyć a^{p-2}, a to można zrobić korzystając z szybkiego potęgowania modularnego w złożoności $O(\log P)$.

Opracowanie: **Kupiec** Autor opracowania: **Robert Kozikowski**

Dostępna pamięć: **64 MB** *Dzień 4, 23.09.2010*

Kupiec

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe opiera się na programowaniu dynamicznym. W rozwiązaniu ukorzeńmy najpierw drzewo w dowolnym wierzchołku v_k. Niech v_w to dowolny wierzchołek w drzewie. Dla każdego wierzchołka chcemy znaleźć dwie poniższe wartości:

1. Wagę jak najdroższej ścieżki zaczynającej się i kończącej w poddrzewie v_w. Nazwijmy ją wartością *pierwszą* dla v_w.

2. Wagę jak najdroższej ścieżki zaczynającej się w poddrzewie wyznaczanym przez v_w, a kończącą się na v_w. Nazwijmy ją wartością *drugą* dla v_w.

Jeśli v_w jest liściem (wierzchołkiem bez synów) to oczywistym jest, że obie te wartości to 0. Przypuśćmy teraz, że dla wszystkich synów v_w znamy obie te wartości. Przypuśćmy, że v_s to dowolny syn v_w. Dla każdego syna v_s dodajmy do jego wartości *drugiej* wagę krawędzi łączącej v_w, oraz v_s.

Wartość *druga* dla v_w to maksimum z 0 oraz wartości *drugiej* dla każdego syna (już po dodaniu wagi krawędzi), gdyż wartość *druga* to albo pusta ścieżka (o wadze 0), albo ścieżka, która idzie z v_w przez któregoś z jego synów (dla którego mamy już policzoną optymalną wartość *drugą*).

Wartość *pierwsza* dla v_w to max z:

1. 0, bo być może nie istnieje wartość dodatnia.

2. Wszystkich wartości *pierwszych* dla każdego v_s, gdyż być może ścieżka, dla której wagą jest wartość *pierwsza*, nie musi wcale przechodzić przez v_w i być może znalazła się w którymś z jego synów.

3. Suma najdroższej i drugiej najdroższej z kolei wartości *drugiej* dla dwóch różnych synów v_s. Jest to wersja, gdy najdroższa ścieżka, zaczynająca się i kończąca się w poddrzewie v_w, zawiera wierzchołek v_w. Wtedy od wierzchołka v_w muszą odchodzić dwie krawędzie do dwóch różnych synów. Chcemy, aby wartość *pierwsza* dla v_w była jak największa, więc interesują nas dwie największe wartości *drugie* jego synów.

Jeśli znamy obie wartości dla każdego wierzchołka, to wynikiem całego zadania będzie wartość *pierwsza* dla wierzchołka v_k, czyli wierzchołka, w którym ukorzeniliśmy całe drzewo, co widać bez głębszego tłumaczenia.

Rozwiązanie błędne

Rozwiązaniem, które niektóre osoby mogą uznać za prawidłowe, a które jest błędne, jest algorytm, który stosuje się przy szukaniu średnicy drzewa (wersja zadania, gdy wszystkie krawędzie mają nieujemną wagę). Algorytm w skrócie polega na tym, że z dowolnego wierzchołka v_r znajdujemy jak najdalszy wierzchołek v_n (za pomocą przeszukiwania grafu wszerz). Wtedy wynikiem będzie odległość od v_n do jak najbardziej oddalonego od niego wierzchołka.

Kontrprzykład do powyższego algorytmu: weźmy drzewo D takie, że wszystkie krawędzie mają dodatnie wagi. Stwórzmy nowy wierzchołek v_u. Między v_u a dowolnym wierzchołkiem D dodajmy krawędź o bardzo dużej ujemnej wartości (większej niż suma wszystkich krawędzi D). Teraz, jeśli ustalimy, że wierzchołkiem początkowym w powyższym algorytmie będzie v_u, to algorytm da błędną odpowiedź.

Opracowanie: **Materiały wybuchowe** Autor opracowania: **Robert Kozikowski**

Dostępna pamięć: **64 MB** *Dzień 4, 23.09.2010*

Materiały wybuchowe

Wstęp

Przyjmijmy takie oznaczenia jak w zadaniu, czyli n – liczba ciężarówek, k – liczba materiałów wybuchowych, $y_1, ..., y_k$ – kolejne materiały wybuchowe, $x_1, ..., x_n$ – kolejne ciężarówki.

Optymalnym zapakowaniem ciężarówki nazwiemy takie zapakowanie jej materiałami wybuchowymi, że jest szczelnie zapakowana oraz użyto jak najmniej materiałów wybuchowych. Pierwszą rzeczą, jaką należy zrobić, jest posortowanie kolejnych wielkości materiałów wybuchowych oraz usunięcie powtórzeń. Załóżmy teraz, że $(y_i < y_{i+1})$.

Lemat.2. *Należy zauważyć, że skoro ${y_k}^2 < x_i$ (czyli że wielkość każdej ciężarówki jest większa od kwadratu największego materiału wybuchowego), to jeśli każda ciężarówka została optymalnie zapakowana, to znajduje się tam co najmniej jeden największy materiał wybuchowy(czyli y_k).*

Dowód.

(Niekonieczny do zrozumienia rozwiązania). Przypuśćmy, że ciężarówka nie posiada w optymalnym zapakowaniu żadnego największego materiału wybuchowego wielkości y_k. Niech a_i to liczność materiału wybuchowego y_i w optymalnym zapakowaniu. Wiemy, że $a_1 * y_1 + a_2 * y_2 + ... + a_{k-1} * yk - 1 = x_i, x_i > {y_k}^2$. Ponadto wiemy, że $y_i < y_k$ dla $i < k$, zatem suma liczności tych materiałów $z = a_1 + a_2 + ... + a_{k-1} < y_k$. Weźmy teraz ciąg zbiorów materiałów wybuchowych $U_1 \subset U_2 \subset ... \subset U_{z-1} \subset U_z$, gdzie $U_i = U_{i-1} + \{y_j\}$, gdzie y_j to dowolny materiał należący do optymalnego rozwiązania, a nienależący do U_{i-1}. Ponadto U_z to zbiór będący optymalnym zapakowaniem. Czyli innymi słowy $U_1, ..., U_z$ to ciąg kolejnych zawierających się w sobie podzbiorów optymalnego rozwiązania, takich, że zaczynamy od zbioru mocy 1, a kończymy na zbiorze mocy z. Zauważmy teraz, że jeśli weźmiemy dla każdego zbioru sumę wielkości jego materiałów wybuchowych modulo wielkość największego materiału wybuchowego (y_k), to z zasady szufladkowej Dirichleta w ciągu $U_1, ..., U_z$ muszą istnieć dwa takie zbiory U_i oraz $U_j, (i < j)$, że suma ich elementów modulo y_k jest taka sama (bo $z > y_k$). Wtedy zbiór $U_j \backslash U_i$ ma sumę elementów modulo y_k równą 0 oraz jest niepusty. Zauważmy, że zbiór $U_j \backslash U_i$ możemy zastąpić zbiorem P składającym się z samych materiałów wielkości y_k, takim że suma wielkości jego materiałów wybuchowych będzie taka sama jak zbioru $U_j \backslash U_i$, jego liczność będzie mniejsza, zatem dla naszego rozwiązania istnieje rozwiązanie o mniejszej liczności – zatem sprzeczność. ■

Rozwiązanie brutalne

Najprostsze rozwiązanie polega na tym, że dla każdej ciężarówki tworzymy tablicę wielkości $x_i + 1$ i za pomocą przeszukiwania wszerz szukamy jaka jest najtańsza ścieżka idąc od 0 do x_i za pomocą wszystkich materiałów wybuchowych. Zrozumienie i zakodowanie takiego rozwiązanie nie powinno być problemem.

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe jest ulepszeniem rozwiązania brutalnego. W tym rozwiązaniu zakładamy, że ciężarówka zawsze zostanie wypełniona pewną liczbą materiału y_k. W rozwiązaniu najpierw zakładamy, że ciężarówka x_i jest wypełniona $W = x_i/y_k$ materiałami wybuchowymi.

Niestety prawie zawsze zostaje nam jakaś reszta $R = x_i \bmod y_k$. Dla danej reszty możemy spróbować BFSem (tak jak w sposobie brutalnym) wypełnić ją materiałami. Tak wypełnioną tablicę nazwijmy T_1. Następnie możemy dodać na początek tablicy y_k komórek (pozostałe zostawić niezmienione), w nową pierwszą komórkę wstawić wartość -1 (bo o tyle wynik w danej komórce będzie się różnił od W) i próbować z tej komórki puścić BFS itd. (tablicę po jednym odjęciu nazwijmy T_2). Jeśli będziemy kolejno zabierać y_k, wtedy w komórce $y_k + i$ będzie minimum z wartości, jeśli dopełniliśmy i elementów, zabraliśmy y_k i dopełniliśmy $y_k + i$ elementów, zabraliśmy $2 * y_k$ i dopełniliśmy $2 * y_k + i$ itd., ale istnieje lepszy sposób.

Teraz niech T_3 to tablica T_1, do której dodaliśmy na koniec y_k komórek i kontynuowaliśmy BFS, którego użyliśmy przy generowaniu T_1. Zauważmy, że w tablicy T_2 ostatnie R komórek to minimum z i-tej komórki T_1 oraz $i + y_k$-tej komórki T_3 pomniejszonej o 1. Zauważmy teraz, że interesuje nas tak naprawdę tylko R ostatnich komórek tablicy T_2, a w tablicy T_3 posiadamy wystarczającą ilość informacji, aby je obliczyć. Zatem zastąpmy pierwsze R komórek tablicy T_3 przez ostatnie R komórek T_2. Teraz możemy zabrać ostatnie R komórek T_3, gdyż już one nas nie interesują. Zauważmy, że wystarczyłoby, jeśli od razu tablica T_3 miałaby wielkość y_k, gdyż jeśli przekroczylibyśmy komórkę y_k, przypuśćmy, że jesteśmy w komórce $y_k + p$, to wystarczyło, by aktualny wynik pomniejszony o 1 wpisać do komórki p. Zauważmy, że jeżeli w komórce nie zmieniamy wartości (tzn. została wartość sprzed przekroczenia y_k, to nie warto dalej puszczać z niej BFSa). Jeśli dalej będziemy puszczać BFS-a po takiej tablicy w opisany wyżej sposób, to znajdą się tam kolejno wartości, jeśli byśmy dodali na początek T_2 $2{*}y_k$ i wpisali -2 w pierwszą komórkę, $3{*}y_k$ i wpisali -3 w pierwszą komórkę, itd., gdyż będą to wartości, które osiągniemy po kolejnych przekroczeniach komórki y_k. Zawsze możemy odjąć y_k od aktualnej sumy klocków, co udowodniliśmy wcześniej. Interesuje nas R-ta już wygenerowanej tablicy T_3. Warto powiedzieć, że do generowania T_3 lepiej użyć algorytmu Dijkstry, gdyż używając BFSa będziemy dużo częściej poprawiać komórki. Zauważmy, że mimo, iż wyszliśmy z tablicą T_3 dla jednej ciężarówki, to tablica ta może nam posłużyć przy generowaniu wyniku dla kolejnych ciężarówek. Wynik dla i-tej ciężarówki to $x_i/y_k + T_3[x_i \bmod y_k]$.

9.6 Dzień piąty

Opracowanie: **Najdłuższy, wspólny, rosnący** Autor opracowania: **Łukasz Jocz**

Dostępna pamięć: **64 MB** *Dzień 5, 24.09.2010*

Najdłuższy, wspólny, rosnący

Rozwiązanie wzorcowe

Niech A, tak jak w treści, oznacza pierwszy, a B – drugi ciąg liczb podanych na wejściu. Zdefiniujmy $S[i][j]$ jako najdłuższy rosnący podciąg ciągu $A[1..i]$ oraz $B[1..j]$, taki że ostatnim wyrazem tego podciągu jest $B[j]$.

Będziemy obliczać kolejne wiersze tablicy $S[i][j]$, począwszy od $i = 1$ do n. Zauważmy, że chcąc obliczyć i-ty wiersz tabeli, możemy poprawić tylko takie komórki $S[i][j]$, gdzie $A[i] = B[j]$. Wówczas możemy pewien wspólny rosnący podciąg ciągów $A[1..i-1]$ oraz $B[1..j-1]$, kończący się wartością $k, k < A[i]$, wydłużyć, dodając na koniec liczbę $A[i]$. Zatem $S[i][j] = \max_{l \in [1,j-1], B[l] < A[i]} T[i-1][l] + 1$. W przeciwnym wypadku, czyli gdy $A[i] \neq B[j]$ przepisujemy do komórki $S[i][j]$ wartość $S[i-1][j]$. Równolegle zapisujemy dla każdej komórki, z jakiej innej komórki wzięliśmy najlepszą wartość, aby móc na końcu odtworzyć rozwiązanie. Będziemy zatem dla każdego i przebiegać po wartościach j od 1 do m i na bieżąco obliczać maksymalną spośród wartości $S[i-1][l] + 1$ dla takich l, które spełniają $B[l] < A[i]$. Oznaczmy tę wartość jako X. Napotykając na takie j, że $A[i] = B[j]$, będziemy zapisywać w komórce $S[i][j]$ wartość X. Rozwiązaniem jest maksymalna wartość z ostatniego wiersza tablicy, czyli spośród wartości $S[n][j], 1 \leqslant j \leqslant m$.

Poniżej znajduje się pseudokod rozwiązania. W tablicy $P[i][j]$ trzymamy wartości potrzebne do odtworzenia wyniku, czyli z której komórki z poprzedniego wiersza wzięliśmy wynik (inaczej, do której komórki w poprzednim wierszu musimy przejść odtwarzając wynik).

```
1  for i:=1 to n do
2      best := 0
3      for j:=1 to m
4          if A[i] == B[j] then
5              S[i][j] := S[i-1][best] + 1
6              P[i][j] := best
7          else
8              S[i][j] := S[i-1][j]
9          if B[j] < A[i] and S[i-1][j] > S[i-1][best] then
10             best := j
```

Rozwiązanie to ma złożoność czasową i pamięciową $O(nm)$.

Opracowanie: **Bubu** Autor opracowania: **Robert Kozikowski**

Dostępna pamięć: **64 MB** *Dzień 5, 24.09.2010*

Bubu

Ważna obserwacja

Obserwacją, dzięki której rozwiązanie zadania staje się bardzo proste, jest to, że optymalna strategia dla strażników jest taka, że każdy strażnik idzie do Jaskini Bubu i czeka tam, aż do przybycia Bubu. Jest tak, gdyż jeśli strażnicy potrafią w jakiś sposób zablokować drogę Bubu (stanąć na drodze Bubu przed jego przybyciem), to mogą równie dobrze szybciej dojść do Jaskini.

Przypuśćmy, że Bubu idzie najkrótszą ścieżką $v_l, v_{l-1}, ..., v_1$ (gdzie v_1 to jaskinia Bubu) do Jaskini. Wtedy żaden strażnik, jeśli nie może złapać Bubu w jaskini, to nie może złapać Bubu wcześniej. Jeśli któryś ze strażników może złapać Bubu przed Jaskinią (w którymś z wierzchołków $v_i, i \leqslant n$), to wtedy, jeśli będzie dalej szedł tą scieżką to będzie w jaskini wcześniej niz Bubu.

Przypuśćmy, że Bubu nie będzie szedł najkrótszą ścieżką. Jeśli strażnik ma dalej do Jaskini niż Bubu, to Bubu, aby uciec danemu strażnikowi, może pójść najkrótszą ścieżką. Jeśli strażnik ma bliżej do Jaskini, to może złapać Bubu w Jaskini.

Rozwiązanie wzorcowe

W rozwiązaniu wzorcowym obliczamy dla każdego wierzchołka v_i najkrótszą ścieżkę z v_i do wierzchołka v_1 (jaskini Bubu). Wykorzystujemy do tego algorytm *Dijkstry* z wierzchołkiem startowym v_1 (gdyż najkrótsza ścieżka z v_i do v_1 ma taką samą długość, jak ścieżka z v_1 do v_i, bo graf jest nieskierowany). Jeśli któryś strażnik stoi na wierzchołku bliżej Jaskinii niż Bubu, to odpowiedzią jest -1, w przeciwnym wypadku odległość Bubu od Jaskini.

Opracowanie: **Myszka** Autor opracowania: **Adrian Jaskółka**

Dostępna pamięć: **64 MB** *Dzień 5, 24.09.2010*

Myszka

Wstęp

W treści mamy myszkę, labirynt z pokojami i korytarzami, serek w pokojach, szerokości korytarzy oraz pytanie o maksymalną wagę myszki, tak aby udało się jej wydostać z labiryntu. Nie jest jakimś trudnym spostrzeżeniem, że mamy tutaj graf nieskierowany z wagami w wierzchołkach oraz krawędziach. Tak więc przy omawianiu rozwiązania tego zadania będziemy się posługiwali terminami znanymi z teorii grafów. Będziemy również naprzemiennie używali określenia waga myszki oraz grubość myszki.

Rozwiązanie o złożoności $O(m \log m \log c)$

Postawmy nieco inny problem niż w treści zadania: znając początkową masę myszki należy odpowiedzieć na pytanie, czy myszka może dojść do końca labiryntu. Jeżeli potrafilibyśmy odpowiedzieć na to pytanie, to moglibyśmy prosto odpowiedzieć na pytanie z treści, wykorzystując wyszukiwanie binarne.

Dosyć jasne jest, że dla myszki lepiej być chudszą niż grubszą przy wchodzeniu do jakiegoś pokoju. Obliczmy więc dla każdego pokoju, jaką najniższą wagę może mieć myszka przy wchodzeniu do niego. Jasne jest, że dla pokoju startowego będzie to po prostu waga myszki. A jak obliczyć to dla reszty? Można wykorzystać tu algorytm *Dijkstry*. Jeżeli wiemy, że możemy dojść do jakiegoś wierzchołka v mając wagę x, to wiemy, że opuścimy go mając wagę $x + ser[v]$. Należy teraz sprawdzić wszystkie krawędzie wychodzące z wierzchołka v i sprawdzić, czy $x + ser[v]$ nie przekracza jej szerokości. Jeżeli nie, to oznacza że myszka może przejść tą krawędzią i należy zaktualizować wartość dla wierzchołka po drugiej stronie krawędzi. Jeżeli algorytm *Dijkstry* dojdzie kiedyś do wierzchołka, w którym znajduje się wyjście, to oznacza że początkowa waga myszki pozwala jej na dojście do końca. Powyższe rozwiązanie ma złożoność $O(M \log M \log C)$ (złożoność *Dijkstry* pomnożona przez liczbę jej wywołań w wyszukiwaniu binarnym).

Rozwiązanie wzorcowe o złożoności $O(m \log m)$

W powyższym rozwiązaniu szukaliśmy binarnie po wyniku i puszczaliśmy algorytm *Dijkstry*, który miał odpowiedzieć na pytanie o najmniejszą wagę myszki, z jaką może ona wejść do każdego wierzchołka.

Podejdźmy do problemu zupełnie odwrotnie: znajdźmy dla każdego wierzchołka maksymalną wagę myszki, taką, że mając taką wagę i zaczynając z danego wierzchołka, jest w stanie dojść do wyjścia labiryntu. Oznaczmy to przez $W[v]$. Tu też puścimy zmodyfikowany w pewien sposób algorytm *Dijkstry*, jednak na kolejce priorytetowej, która zwraca największy element. Otóż wiemy, że nasza szukana wartość dla wierzchołka końcowego jest równa INF. Dla reszty ustalmy to jako -INF. Wrzućmy wierzchołek końcowy z wartością INF do kolejki priorytetowej, a następnie, jak to w algorytmie *Dijkstry*, róbmy coś dopóki coś jest w kolejce. Będziemy pobierali dotychczas nieprzetworzony wierzchołek o najwiekszej wartości $W[v]$ i teraz musimy zrelaksować wszystkie wierzchołki z nim sąsiadujące. Jeżeli wzięliśmy wierzchołek v i relaksujemy krawędź $v-u$ o szerokości c oraz mamy s sera w wierzchołku u, to relaksujemy następująco:

$W[u] = max(W[u], min(W[v] - s, c - s))$.

Wątpliwość może rodzić $min(W[v] - s, c - s)$. Czemu jest to maksymalna waga myszki zaczynającej w u, która idzie przez krawędź $u - v$ oraz dochodzi do końca? Po pierwsze, myszka musi się przecisnąć przez krawędź $u - v$, czyli musi mieć maksymalnie grubość $c - s$. Po przedostaniu się przez krawędź $u - v$ musi również dojść do końca, a to jej zapewnia fakt, że jest nie grubsza niż $W[v]$ w wierzchołku v, czyli nie grubsza niż $W[v] - s$ w wierzchołku u. Tak więc, po wykonaniu tego algorytmu, wynik zadania będzie się znajdował w $W[start]$.

Rozwiązanie to ma taką samą złożoność co algorytm *Dijkstry*, czyli $O(M \log M)$.

Rozdział 10

Wykłady

10.1 Teoria liczb II

Wykład: **Teoria liczb II** Autor wykładu: **Łukasz Jocz**

10.1.1 Rozszerzony algorytm Euklidesa

Na początku zajmiemy się rozwiązywaniem równań $ax + by = d$, gdzie $d = nwd(a, b)$. Takie równanie ma nieskończenie wiele rozwiązań w liczbach całkowitych x, y. Nasz algorytm będzie opierał się na tym, że jak mamy obliczone liczby x_p, y_p, x_q, y_q, takie że $p = a * x_p + b * y_p$ oraz $q = a * x_q + b * y_q$, to możemy łatwo obliczyć takie liczby x, y, że $(p \bmod q) = a * x + b * y$. W wyznaczeniu liczb x, y bardzo pomocna będzie następująca tożsamość $(p \bmod q) = p - \left\lfloor \frac{p}{q} \right\rfloor q$. Bierze się to stąd, że liczbę $(p \bmod q)$ można obliczyć, odejmując od liczby p liczbę q, dopóki liczba ta jest większa od 0. Liczba $\left\lfloor \frac{p}{q} \right\rfloor$ oznacza, ile razy musimy odjąć q. Podstawiając za liczby p i q odpowiednio $a * x_p + b * y_p$ oraz $a * x_q + b * y_q$, otrzymujemy $(p \bmod q) = p - \left\lfloor \frac{p}{q} \right\rfloor q = x_p a + y_p b - \left\lfloor \frac{p}{q} \right\rfloor x_q a - \left\lfloor \frac{p}{q} \right\rfloor y_q b = a(x_p - \left\lfloor \frac{p}{q} \right\rfloor x_q) + b(y_p - \left\lfloor \frac{p}{q} \right\rfloor y_q)$. Zatem $x = x_p - \left\lfloor \frac{p}{q} \right\rfloor x_q, y = y_p - \left\lfloor \frac{p}{q} \right\rfloor y_q$. Algorytm Euklidesa opiera się na tożsamości $nwd(p, q) = nwd(q, p \bmod q)$. Wielokrotnie wywołujemy algorytm rekurencyjnie, dopóki obie liczby p i q są większe od 0. Wywołując algorytm rekurencyjnie, cały czas zachowane są własności $p = ax_p + by_p$, $q = ax_q + by_q$. Możemy to zachować dzięki zastosowaniu wyżej wyprowadzonych wzorów na x i y dla $p \bmod q$. W pierwszym wywołaniu algorytmu $p = a, q = b$. Zauważmy, że liczby $x_p = 1, y_p = 0$ spełniają równanie $p = a * x_p + b * y_p$. Natomiast liczby $x_q = 0, y_q = 1$ spełniają równanie $q = a * x_q + b * y_q$. Algorytm się kończy, kiedy q jest równe zero, wówczas $p = nwd(a, b)$ i liczby x_p oraz y_p są naszym rozwiązaniem. Podsumujmy, jak będą zmieniały się argumenty przy wywoływaniu rekurencyjnym.

p	q	x_p	y_p	x_q	y_q
$p' = q$	$q' = p \bmod q$	$x'_p = x_q$	$y'_p = y_q$	$x'_q = x_p - \left\lfloor \frac{p}{q} \right\rfloor x_q$	$y'_q = y_p - \left\lfloor \frac{p}{q} \right\rfloor y_q$

Wartość q zostaje „przepisana” jako nowe p, według własności $nwd(p, q) = nwd(q, p \bmod q)$. W związku z tym, przepisane zostają również liczby x_q i y_q.

Zobaczmy na przykładzie jak będzie wyglądał algorytm dla $a = 11, b = 7$.

p	q
$x_p = 1,\ y_p = 0$ $p = 11$	$x_q = 0,\ y_q = 1$ $q = 7$
$x_p = 0,\ y_p = 1$ $p = 7$	$x_q = 1,\ y_q = -1$ $q = 4$
$x_p = 1,\ y_p = -1$ $p = 4$	$x_q = -1,\ y_q = 2$ $q = 3$
$x_p = -1,\ y_p = 2$ $p = 3$	$x_q = 2,\ y_q = -3$ $q = 1$
$x_p = 2,\ y_p = -3$ $p = 1$	$x_q = -7,\ y_q = 11$ $q = 0$

10.1.2 Rozwiązywanie liniowego równania diofantycznego

W tej części zajmiemy się rozwiązywaniem równań postaci $ax + by = c$, dla dowolnego c. Równanie takie ma 0 lub nieskończenie wiele rozwiązań. Zależy nam na znalezieniu dowolnego małego rozwiązania, takiego równania lub stwierdzenia, że takie rozwiązanie nie istnieje. Niech $d = nwd(a, b)$.

Lemat.3. *Równanie $ax + by = c$ ma rozwiązanie wtedy i tylko wtedy, kiedy $d|c$.*

Dlaczego? Jeżeli c nie jest podzielne przez d, to $ax + by$ jest podzielne przez d, a c nie. Równanie takie nie może więc mieć rozwiązań.

Jeżeli $d|c$ to z rozszerzonego algorytmu Euklidesa możemy otrzymać liczby x_1, y_1, takie że $d = ax_1 + by_1$. Mnożąc stronami przez $\frac{c}{d}$, otrzymujemy $c = a\frac{x_1 c}{d} + b\frac{y_1 c}{d}$. Zatem istnieje rozwiązanie równania.

Jednak takie rozwiązanie nas nie satysfakcjonuje, gdyż otrzymane wartości x i y mogą być kwadratowo duże ze względu na liczby a, b, c. Algorytm, który tutaj zastosujemy, będzie nieznaczną modyfikacją wyżej podanego, rozszerzonego algorytmu Euklidesa. Zauważmy, że mając rozwiązać równanie $ax + by = n$ oraz mając dane liczby x_q, y_q, takie że $q = ax_q + by_q$, możemy uprościć równanie $ax + by = n$, odejmując od niego, największą wielokrotność równania $q = ax_q + by_q$. Niech k będzie największą taką liczbą, że $kq \leqslant n$. Zatem $k = \left\lfloor \frac{n}{q} \right\rfloor$. Wówczas równanie, które mamy rozwiązać, upraszcza się do postaci $ax+by = n-kq$. Jeżeli rozwiążemy to równanie i dodamy do niego stronami $kax_q + kby_q = kq$, otrzymamy rozwiązanie początkowego równania. W ten sposób będziemy zmniejszać n, dopóki $n > 0$. Przyjrzyjmy się temu na przykładzie. Niech $a = 11, b = 7, c = 24$. Równanie ma więc postać $11x + 7y = 24$. W pierwszym wywołaniu zachodzi $p = a$, $q = b$, $n = c$.

p	q	n	Największe k, takie, że $kq \leqslant n$
$x_p = 1,\ y_p = 0$ $p = 11$	$x_q = 0,\ y_q = 1$ $q = 7$	$n = 24$	$k = 3, kq = 21, x = kx_q = 0, y = ky_q = 3$
$x_p = 0,\ y_p = 1$ $p = 7$	$x_q = 1,\ y_q = -1$ $q = 4$	$n = 3$	$k = 0, kq = 0, x = kx_q = 0, y = ky_q = 0$
$x_p = 1,\ y_p = -1$ $p = 4$	$x_q = -1,\ y_q = 2$ $q = 3$	$n = 3$	$k = 1, kq = 3, x = kx_q = -1, y = ky_q = 2$
$x_p = -1,\ y_p = 2$ $p = 3$	$x_q = 2,\ y_q = -3$ $q = 1$	$n = 0$	$k = 0, kq = 0, x = kx_q = 0, y = ky_q = 0$

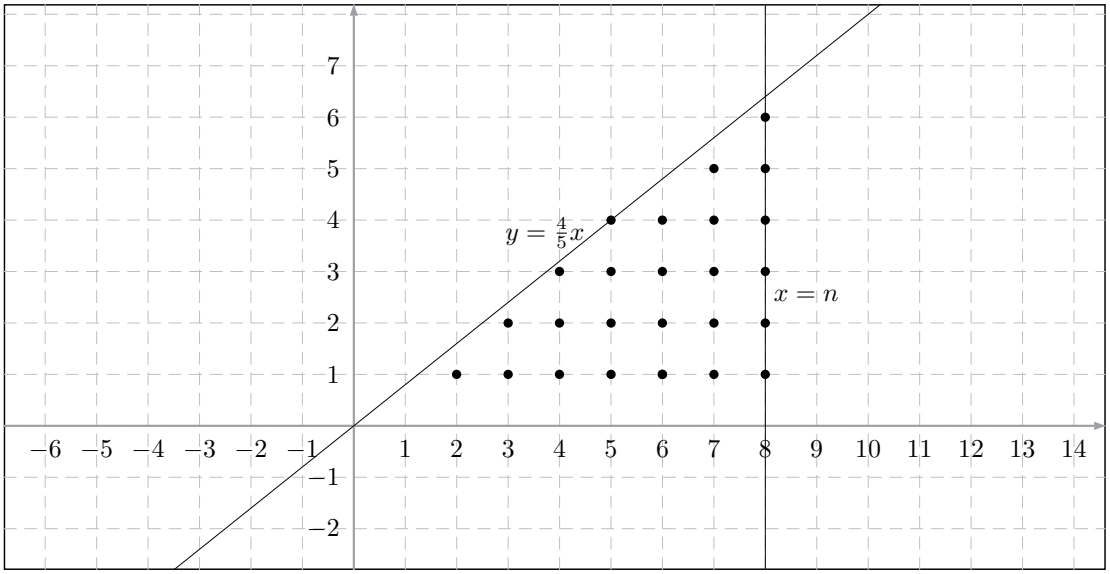
Nowe n jest równe staremu n, pomniejszonemu o kq. Algorytm kończymy, gdy $n = 0$.

Dzięki takiemu postępowaniu, po zsumowaniu wartości x i y z ostatniej kolumny, otrzymamy rozwiązanie $x = -1, y = 5$.

10.1.3 Zliczanie punktów kratowych

Punktem kratowym nazywać będziemy punkt, który ma obie współrzędne całkowite.

W tej części wykładu zajmiemy się zliczaniem punktów kratowych w obszarze ograniczonym prostymi $y = \frac{a}{b}x, x = n$ oraz osią OX. Przy czym nie będziemy zliczać punktów, które leżą na osi OX.



Najprostszym algorytmem jest przebieganie po wszystkich wartościach i z przedziału $[0, n]$ i dla każdego i zliczanie, ile jest punktów kratowych między prostą $y = \frac{a}{b}x$, a osią OX, w których pierwsza współrzędna wynosi i. Zgodnie z tym pomysłem, poszukiwaną liczbę punktów kratowych możemy przedstawić jako:

$$\sum_{i=0}^{n} \left\lfloor \frac{ai}{b} \right\rfloor \tag{10.1}$$

akie rozwiązanie działa w złożoności $O(n)$ i nas nie satysfakcjonuje.

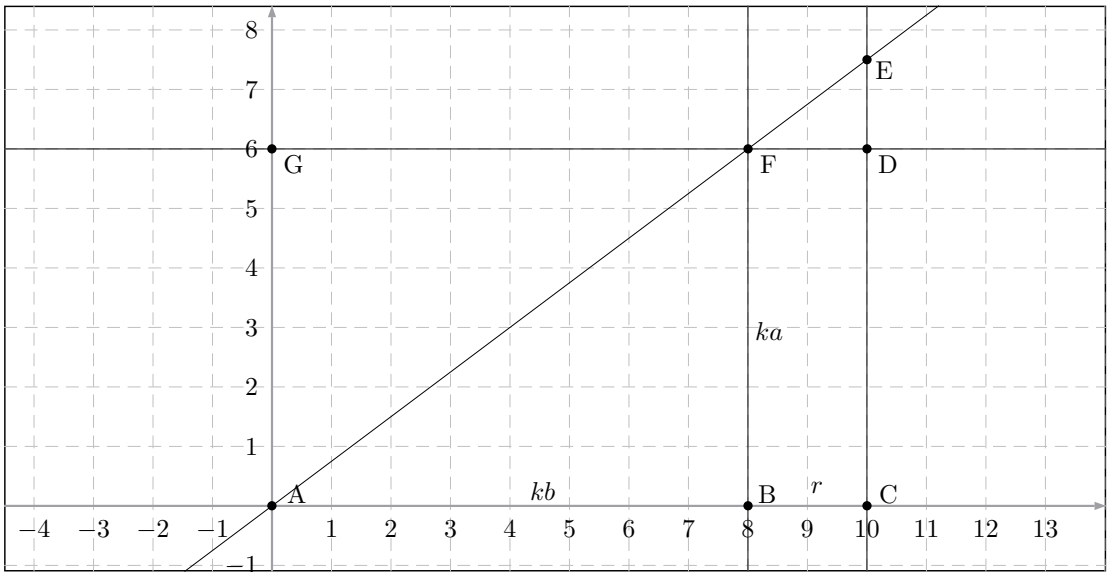
W dalszych rozważaniach będziemy zakładać, że liczby a i b są względnie pierwsze. Jeżeli tak nie jest, to możemy obie podzielić przez ich największy wspólny dzielnik i wtedy już będą względnie pierwsze. Nasz pomysł rozwiązania tego zadania będzie opierał się na rekurencyjnym sprowadzaniu problemu z (a,b,n), do mniejszych podproblemów. Rozpatrzymy 3 przypadki.

Przypadek $n \geqslant b$

W przypadku pierwszym zakładamy, że $n \geqslant b$.

Wówczas, niech k będzie największą taką liczbą, dla której $bk \leqslant n$, a r będzie resztą z dzielenia n przez b. Inaczej mówiąc, niech kb największą wielokrotnością liczby b, mniejszą lub równą n, a r będzie równe $n - kb$. Zauważmy, że punkt (kb, ka) leży na prostej $y = \frac{a}{b}x$ i ma obie współrzędne całkowite. Niech $n = kb + r$. Liczbę punktów kratowych w trójkącie $\triangle ABF$ możemy szybko policzyć w następujący sposób. Wiemy, że w prostokącie $ABFG$ jest $(ka + 1) \cdot (kb + 1)$ punktów kratowych. Na przekątnej leży $k + 1$ punktów. Bierze to się stąd, że a i b są względnie pierwsze, w związku z tym pierwszym punktem kratowym różnym od $(0, 0)$ jest (b, a), a

wszystkie pozostałe punkty mają postać (lb, la). Zatem liczba punktów kratowych w trójkącie $\triangle ABF$, to $\frac{(ka+1)(kb+1)+(k+1)}{2}$. Przy czym nie liczymy punktów na osi OX, więc zostaje $\frac{(ka+1)(kb+1)+(k+1)}{2} - (kb+1)$ punktów. Liczba punktów w prostokącie $BCDF$ to prosty iloczyn, jednak należy zwrócić uwagę, aby nie policzyć punktów na odcinku BF, bo te już policzyliśmy, zliczając punkty dla trójkąta ABF. Pozostał trójkąt FDE, dla którego wywołamy algorytm rekurencyjnie. W ten sposób problem dla (n, a, b) sprowadziliśmy do problemu dla (r, a, b), czyli $(n \bmod b, a, b)$.



Przypadek $n < b$ oraz $a \geqslant b$

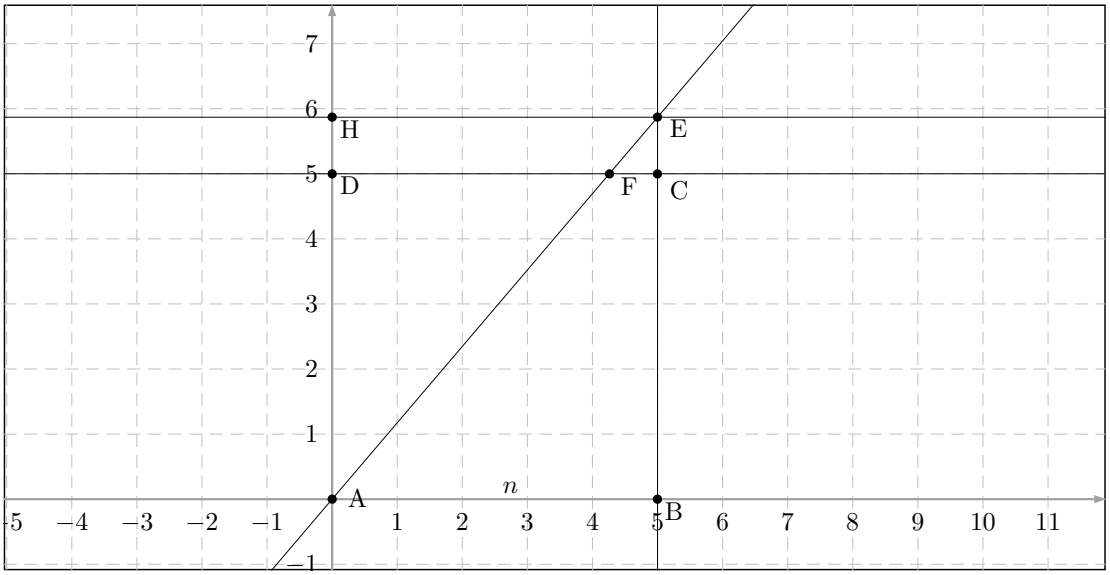
Ten przypadek najłatwiej będzie rozwiązać algebraicznie. Przyjmijmy $a = kb + r$, czyli $r = (a \bmod b)$. Przekształcimy trochę sumę (1).

$$\sum_{i=0}^{n} \left\lfloor \frac{ai}{b} \right\rfloor = \sum_{i=0}^{n} \left\lfloor \frac{(kb+r)i}{b} \right\rfloor = \sum_{i=0}^{n} \left\lfloor \frac{kbi}{b} + \frac{ri}{b} \right\rfloor = \sum_{i=0}^{n} (ki + \left\lfloor \frac{ri}{b} \right\rfloor) = \sum_{i=0}^{n} ki + \sum_{i=0}^{n} \left\lfloor \frac{ri}{b} \right\rfloor = k \sum_{i=0}^{n} i + \sum_{i=0}^{n} \left\lfloor \frac{ri}{b} \right\rfloor \tag{10.2}$$

Przy trzecim przekształceniu ki możemy wyciągnąć przed podłogę, ponieważ jest to liczba całkowita.
Suma $\sum_{i=0}^{n} i$ to suma liczb od 0 do n i jest równa $\frac{n(n+1)}{2}$.
Zauważmy, że suma $\sum_{i=0}^{n} \left\lfloor \frac{ri}{b} \right\rfloor$ jest sumą takiej samej postaci, jak ta, którą mieliśmy policzyć, tylko ze zmniejszonym argumentem a. Udało nam się zatem sprowadzić problem dla (n, a, b) do problemu z (n, r, b), czyli $(n, a \bmod b, b)$.

Przypadek $n < b$ oraz $a < b$

Ten przypadek jest najprostszy ze wszystkich trzech, więc zostawiliśmy go na koniec.



Punkt B ma współrzędne $(n, 0)$. Długość odcinka BE jest równa wartości funkcji w punkcie n. Zatem $|BE| = \frac{na}{b}$. Punkt C ma współrzędne $(n, \lfloor \frac{na}{b} \rfloor)$. Zauważmy, że liczba punktów kratowych w trójkącie ABE jest taka sama jak liczba punktów kratowych w czworokącie $ABCF$ (wewnątrz trójkąta CEF nie może być żadnego punktu kratowego, gdyż w przedziale $(\lfloor \frac{na}{b} \rfloor, \frac{na}{b})$ nie ma żadnej liczby całkowitej. Zatem liczbę punktów kratowych w trójkącie ABE można przedstawić jako różnicę liczby punktów kratowych w prostokącie $ABCD$ i trójkącie AFD. Liczbę punktów kratowych w prostokącie $ABCD$ możemy łatwo policzyć, stosując metodę podaną w punkcie 3.2. Zauważmy, że po odbiciu trójkąta AFD względem prostej $y = x$ otrzymamy problem z zadania tylko dla zmienionych danych $(\lfloor \frac{na}{b} \rfloor, b, a)$. Zwróćmy uwagę na to, że w wyniku tej operacji zamieniły się wartości b i a oraz zmniejszyła się wartość n.

Podsumowanie

Zauważmy, że nigdy 2 razy pod rząd nie wystąpi przypadek 3, ponieważ po sprowadzeniu przypadku 3 do mniejszych danych będzie zachodziło $a \geqslant b$. Ponadto, w wyniku każdego sprowadzenia, żadna wartość się nie zwiększa. Zauważmy, że w przypadku każdych dwóch kolejnych wywołań funkcji, co najmniej jedna z liczb a, b, n zmniejszy się co najmniej o połowę. Algorytm wykonujemy dopóki każda z tych liczb jest dodatnia. Zatem liczba kroków algorytmu jest logarytmiczna w stosunku do danych liczb a, b, n.

10.1.4 Zadanie Trzęsienie Ziemi

W tym roku, na obozie ONTAK pojawiło się zadanie Trzęsienie Ziemi o następującej treści.

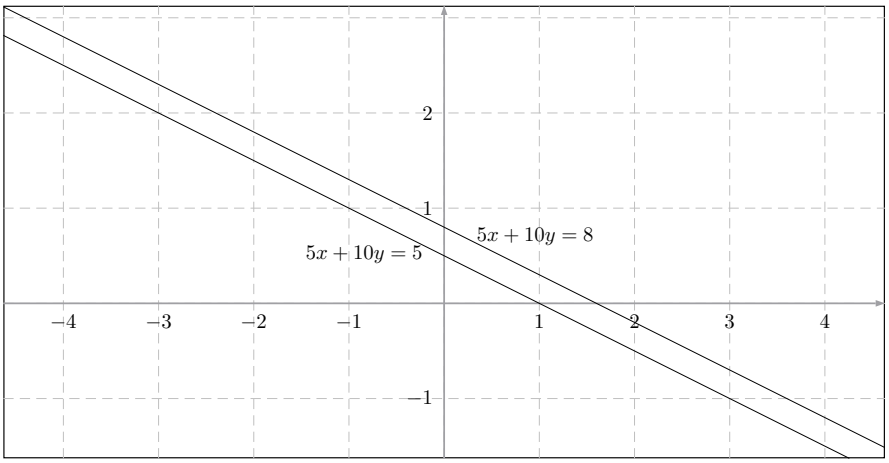
Zadanie. *Dla danych liczb $A, B, C (A, B \leqslant 10^9, C \leqslant \min(A, B) * 10^9)$ obliczyć liczbę punktów kratowych ograniczonych osiami układu współrzędnych oraz prostą $Ax + By = C$.*

Gdyby punkt przecięcia prostej $Ax + By = C$ oraz osi X–ów miał współrzędne całkowite, to mogilbyśmy bezpośrednio policzyć liczbę punktów kratowych w trójkącie za pomocą metody podanej w punkcie 3. W większości przypadków tak jednak nie jest.

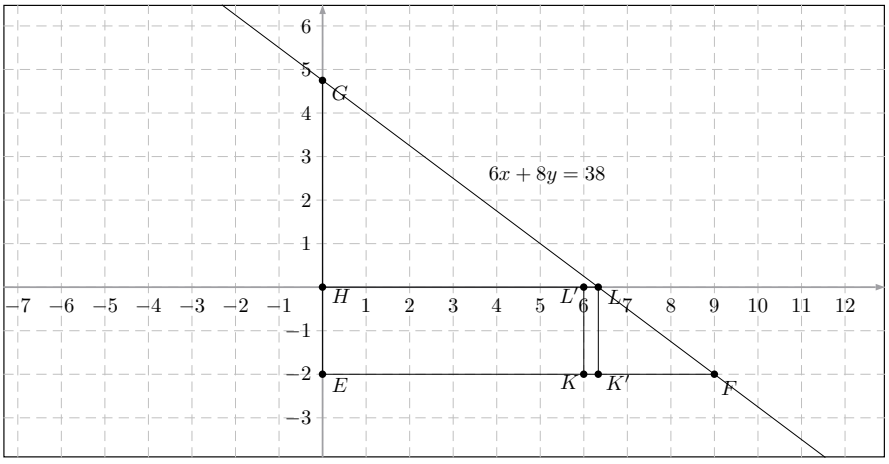
Zauważmy, że prostą $Ax+By = C$ możemy interpretować jako liniowe równanie diofantyczne, którego rozwiązaniami są punkty kratowe leżącej na tej prostej. Jeżeli zatem $nwd(A, B)$ nie

dzieli C to na prostej nie leży żaden punkt kratowy. Niech C' będzie największą wielokrotnością $nwd(A, B)$, mniejszą od C. Proste $Ax + By = C$, oraz $Ax + By = C'$ są równoległe oraz w obszarsze pomiędy tymi prostymi nie leżą żadne punkty kratowe. Dlaczego? Gdyby pomiędzy prostymi $Ax + By = C$ i $Ax + By = C'$ leżał jakiś punkt kratowy, to w szczególności leżałby na pewnej prostej $Ax + By = D$, równoległej do prostej $Ax + By = C$. Wówczas $D > C'$ oraz $D < C$, więc $nwd(A, B)$ nie dzieli D. Zatem na prostej $Ax+By = D$ nie może leżeć żaden punkt kratowy. Uzyskana sprzeczność dowodzi, że pomiędzy prostymi $Ax + By = C$ i $Ax + By = C'$ nie leży żaden punkt kratowy.

W przykładzie na rysunku mamy $A = 5, B = 10, C = 8, C' = 5$.



Ponieważ pomiędzy prostymi $Ax+By = C$ oraz $Ax+By = C'$ nie leży żaden punkt kratowy, więc rozwiązanie dla tych dwóch prostych jest takie samo. Będziemy zatem szukać rozwiązania dla prostej $Ax + By = C'$. Ponieważ $nwd(A, B)|C'$, to możemy za pomocą algorytmu z punktu 2 znaleźć dowolny punkt kratowy leżący na tej prostej. Niech punkt ten ma współrzędne (x, y). Punkt $(x+B, y−A)$, również leży na tej prostej. Ogólniej, na prostej leży punkt $(x+kB, y−kA)$ dla każdego k. Możemy łatwo znaleźć odpowiednio duże k, dla którego $y - kA \leqslant 0$. W ten sposób otrzymujemy punkt F. Zauważmy, że liczbę punktów kratowych w trójkącie HLG możemy policzyć, obliczając liczbę punktów kratowych w trójkącie EFG i odejmując ich liczbę w prostokącie $EKL'H$ i trójkącie $K'FL$. Liczbę punktów kratowych w prostokącie, to $(|L'K| + 1) * (|EK| + 1)$. Liczbę punktów katowych w trójkącie EFG obliczamy algorytmem z punktu 3 dla $n = |EF|, a = A, b = B$. Podobnie, liczbę punktów w trójkącie $K'FL$ możemy policzyć algorytmem z punktu 3, dla $n = |LK'|, a = B, b = A$. Zauważmy, że w tym przypadku, musimy tak jakby obrócić ten trójkąt, aby odcinek $|LK'|$ był w podstawie. Robimy to dlatego, że długość odcinka $|FK'|$ może nie być całkowita. W ten sposób otrzymujemy ostateczne rozwiązanie.



Bibliografia

[1] P. Stańczyk. *Algorytmika praktyczna*. PWN, Warszawa, 2009.

[2] L. Banachowski, K. Diks, W. Rytter. *Algorytmy i struktury danych*. WNT, Warszawa, 1996.

[3] T. H. Cormen, C. E. Leiseron, R. L. Rivest. *Wprowadzenie do algorytmów*. WNT, Warszawa, 1997.

[4] W. Lipski. *Kombinatoryka dla programistów*. WNT, Warszawa, 2004.

[5] Ronald L. Graham, Donald E. Knuth, Oren Patashnik. *Matematyka konkretna*. PWN, Warszawa, 1996.

[6] `http://wazniak.mimuw.edu.pl`.

[7] `http://oi.edu.pl`.