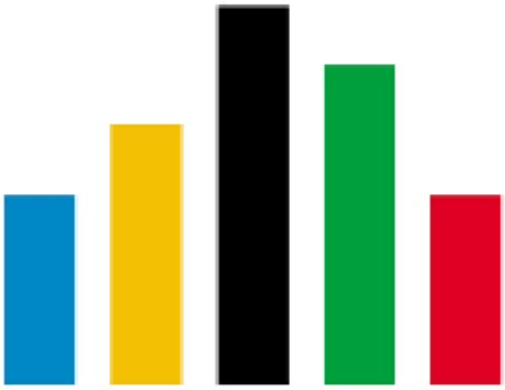
Komitet Główny Olimpiady Informatycznej Gimnazjalistów



OLIMPIADA
INFORMATYCZNA
GIMNAZJALISTÓW

V OLIMPIADA INFORMATYCZNA GIMNAZJALISTÓW

Zawody indywidualne

Treści i opracowania zadań

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

MINISTERSTWO
EDUKACJI
NARODOWEJ

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany ze środków Unii Europejskiej
w ramach Europejskiego Funduszu Społecznego.

Olimpiada Informatyczna Gimnazjalistów jest corocznym przedmiotowym konkursem przeznaczonym dla uczniów szkół gimnazjalnych, organizowanym przez **Ministerstwo Edukacji Narodowej**. Przedsięwzięcie realizuje Stowarzyszenie Talent – partner MEN, współorganizatorem jest miasto Gdynia.

Celem głównym Olimpiady Informatycznej Gimnazjalistów jest **zainteresowanie informatyką uczniów szkół gimnazjalnych**, poprzez szlachetną rywalizację w rozwiązywaniu ciekawych i inspirujących zadań i problemów informatycznych, z zastosowaniem podejścia algorytmicznego i podejmowania decyzji z wykorzystaniem komputera.

Olimpiada Informatyczna Gimnazjalistów przeprowadzana jest w dwóch rodzajach zawodów: **indywidualnych** i **drużynowych**.

Zawody indywidualne rozgrywane są w trzech etapach. Rozwijają kreatywność, wytrwałość w dążeniu do celu i umiejętność podejmowania samodzielnych decyzji. Opiekę merytoryczną nad zawodami indywidualnymi sprawuje Instytut Informatyki Uniwersytetu Warszawskiego.

- etap I ma charakter otwarty i polega na samodzielnym rozwiązywaniu przez uczestnika zadań ustalonych dla tych zawodów oraz przekazaniu rozwiązań w podanym terminie.
- etap II i III polega na rozwiązywaniu zadań w warunkach kontrolowanej samodzielności.

Zawody drużynowe składają się z etapu szkolnego, etapu okręgowego i centralnego. Uczniowie tworzą czteroosobowe zespoły. Przedmiotem zawodów drużynowych są zadania dotyczące zastosowania informatyki w innych naukach ścisłych. Za pomocą specjalnie przygotowanej **platformy edukacyjnej** możliwa jest **systematyczna praca** i przygotowywanie się do zawodów przy pomocy udostępnianych poprzez witrynę materiałów.

- etap I składa się z rund treningowych i dwóch rund konkursowych (konkurs szkolny i powiatowy).
- etap II (okręgowy) i III (centralny) zawodów drużynowych przeprowadzany jest w warunkach kontrolowanej samodzielności. Do etapu centralnego przechodzi najlepsza drużyna z każdego województwa i najlepsze drużyny z całego kraju.

Gala finałowa tegorocznej Olimpiady objęta została Honorowym Patronatem Prezydenta Rzeczypospolitej Polskiej Bronisława Komorowskiego. V Olimpiada Informatyczna Gimnazjalistów objęta została patronatami honorowymi Ministerstwa Edukacji Narodowej i Prezydenta Miasta Gdyni, a także patronatem Fundacji Rozwoju Systemu Edukacji.

Więcej informacji na temat projektu: **http://www.oig.edu.pl/**

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI
MINISTERSTWO EDUKACJI NARODOWEJ
UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY

Projekt współfinansowany ze środków Unii Europejskiej
w ramach Europejskiego Funduszu Społecznego.

OLIMPIADA
INFORMATYCZNA
GIMNAZJALISTÓW

Etap I

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

MINISTERSTWO
EDUKACJI
NARODOWEJ

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany ze środków Unii Europejskiej
w ramach Europejskiego Funduszu Społecznego.

Zadanie: PAR
Park

OLIMPIADA INFORMATYCZNA GIMNAZJALISTÓW

V OIG, etap I. Plik źródłowy `par.*` **Dostępna pamięć: 32 MB.** *10.01–7.02.2011*

Bajtocki Park Narodowy słynie z długiego (choć niezbyt szerokiego) pasma górskiego, rozciągającego się przez cały park z zachodu na wschód. Co roku przyjeżdżają do niego tłumy turystów, którzy często nie są zbyt rozgarnięci. Dlatego zarząd parku postanowił przygotować mapę całego pasma, podzieloną na fragmenty równej długości. Przy każdym punkcie podziału zarząd zamierza umieścić wysokość tego punktu oraz dwie inne liczby: wysokość najwyższego punktu podziału na zachód od niego oraz na wschód od niego.

Cała mapa jest już właściwie gotowa. Pozostaje jedynie obliczyć maksymalne wysokości na zachód i na wschód od każdego punktu podziału. Zarząd parku poprosił Cię o napisanie programu, który wyznaczy te wartości.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 1\,000\,000$) oznaczająca długość pasma górskiego. W każdym z następnych n wierszy znajduje się po jednej liczbie całkowitej w_i ($1 \leqslant w_i \leqslant 1\,000\,000\,000$) oznaczającej wysokość i-tego punktu podziału. Punkty te podane są w kolejności z zachodu na wschód.

W testach wartych przynajmniej 40% punktów zachodzi dodatkowy warunek $n \leqslant 10\,000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście dokładnie n wierszy, odpowiadających kolejnym punktom podziału (w kolejności z zachodu na wschód). W każdym z tych wierszy powinny znaleźć się dwie liczby całkowite a_i oraz b_i oddzielone pojedynczym odstępem — wysokość najwyższego punktu podziału na zachód od punktu i oraz na wschód od niego. W przypadku, gdy na zachód od punktu i nie ma szczytu wyższego niż w_i, przyjmujemy $a_i = w_i$. Podobnie, jeśli na wschód od punktu i nie ma szczytu wyższego niż w_i, to przyjmujemy $b_i = w_i$.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

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

Opracowanie: PAR
Park

1 Rozwiązanie

W zadaniu dany jest ciąg liczb naturalnych $w_1, w_2, \ldots, w_n$, reprezentujący wysokości kolejnych punktów podziału mapy. W tym ciągu dla każdej liczby i od 1 do n należy obliczyć:

1. maksimum z liczb $w_1, \ldots, w_i$
2. maksimum z liczb $w_i, \ldots, w_n$

Możemy to zrobić bezpośrednio:

```
for i := 1 to n do begin
  maxz := w[i];
  for j := 1 to i - 1 do begin
    if (w[j] > maxz) then maxz := w[j];
  end
  maxw := w[i];
  for j := i + 1 to n do begin
    if (w[j] > maxw) then maxw := w[j];
  end
  wypisz(maxz, maxw);
end
```

Niestety to rozwiązanie jest zbyt powolne, ponieważ dla każdego i powyższy algorytm przegląda wszystkie inne komórki tablicy `w[]`, co sprawia, że koszt czasowy wyniesie $O(n^2)$ (tzn. że algorytm wykona mniej więcej n^2 pojedynczych operacji). Zgodnie z treścią zadania, za takie rozwiązanie można było uzyskać 40% punktów.

Aby przyspieszyć działanie programu, wystarczy zauważyć, że obliczone maksima dla pewnego przedziału mogą nam posłużyć w szybkim wyznaczeniu maksimum dla przedziału zawierającego jedną komórkę więcej:

- $\max\{w_1, \ldots, w_i\} = \max(\max\{w_1, \ldots, w_{i-1}\}, w_i)$
- $\max\{w_i, \ldots, w_n\} = \max(w_i, \max\{w_{i+1}, \ldots, w_n\})$

To spostrzeżenie pozwala nam napisać szybki algorytm, który jest już rozwiązaniem wzorcowym.

```
maxz := 0;
maxw := 0;

for i := 1 to n do begin
  if w[i] > maxz then maxz := w[i];
  z[i] := maxz;
end;

for i := n downto 1 do begin
  if w[i] > maxw then maxw := w[i];
  w[i] := maxw;
end;

for i := 1 to n do begin
  wypisz(z[i], w[i]);
end;
```

2 Uwaga na temat strumieni w C++

Aby mieć pewność, że program będzie wystarczająco szybko działał w języku C++, warto się jeszcze upewnić, że we właściwy sposób dokonujemy operacji wejścia i wyjścia, tzn. we właściwy sposób wczytujemy i wypisujemy dane. Otóż operacje przy pomocy strumieni (`cin >>` oraz `cout <<`) są istotnie wolniejsze od funkcji przeniesionych z języka C, które służą do tego samego (`scanf` oraz `printf`). Jeżeli jednak bardzo chcemy wykorzystywać strumienie, to możemy przyspieszyć operacje wykonywane za ich pomocą, wpisując następujący wiersz na początku funkcji main():

```
std::ios_base::sync_with_stdio(0);
```

Polecenie to wyłącza tzw. synchronizację ze standardowym wejściem i wyjściem, co spowoduje, że używanie zarówno strumieni jak i funkcji `scanf` i `printf` może dawać nieoczekiwane efekty. Jednak przyspieszy ono istotnie operacje wykonywane z użyciem strumieni.

Zadanie: SKR
Skracalne liczby pierwsze

OLIMPIADA INFORMATYCZNA GIMNAZJALISTÓW

V OIG, etap I. Plik źródłowy `skr.*` Dostępna pamięć: 32 MB. *10.01–7.02.2011*

Przypomnijmy, że *liczba pierwsza* to taka dodatnia liczba całkowita, która ma dokładnie dwa różne dzielniki: jedynkę i samą siebie. Mówimy, że liczba a jest *prefiksem* liczby b, jeśli liczba a powstaje przez usunięcie pewnej liczby cyfr z końca liczby b. Na przykład, liczba 1231 jest prefiksem liczby 12314433. *Skracalna liczba pierwsza* to taka liczba, której wszystkie prefiksy niezerowej długości są liczbami pierwszymi. Przykładowo, liczba 23 jest skracalną liczbą pierwszą, gdyż jej niepuste prefiksy 2 i 23 są liczbami pierwszymi.

Twoim zadaniem jest napisanie programu, który dla zadanych dwóch liczb całkowitych dodatnich a, b ($a \leqslant b$) wyznaczy, ile jest liczb całkowitych, które są skracalnymi liczbami pierwszymi i należą do domkniętego przedziału $[a, b]$.

Wejście

W jedynym wierszu standardowego wejścia znajdują się dwie liczby całkowite a, b ($1 \leqslant a \leqslant b \leqslant 10^{18}$).

Możesz założyć, że testach wartych 50% punktów zachodzi dodatkowo warunek $b \leqslant 1\,000\,000$.

Wyjście

W jedynym wierszu standardowego wyjścia powinna znaleźć się jedna liczba całkowita będąca liczbą skracalnych liczb pierwszych nie mniejszych od a i nie większych od b.

Przykład

Dla danych wejściowych:	poprawnym wynikiem jest:
`20 24`	`1`

Wyjaśnienie do przykładu: W przedziale $[20, 24]$ jest tylko jedna skracalna liczba pierwsza i jest nią 23.

Opracowanie: SKR
Skracalne liczby pierwsze

1 Wprowadzenie

Zadanie polega na wyznaczeniu, ile jest *skracalnych liczb pierwszych* należących do pewnego zadanego przedziału. *Skracalna liczba pierwsza* to taka liczba, której każdy prefiks (w tym również cała liczba) jest liczbą pierwszą.

2 Rozwiązanie

Zanim zabierzemy się za rozwiązywanie zadanego problemu, zastanówmy się, jak można w miarę szybko i prosto sprawdzić, czy liczba jest pierwsza.

2.1 Sprawdzanie czy liczba jest pierwsza

Niech n będzie pewną liczbą, której pierwszość chcielibyśmy zbadać. Zauważmy, że jeśli n nie jest liczbą pierwszą, to muszą istnieć dwie liczby a, b $(1 < a \leqslant b < n)$ takie, że $ab = n$. Dodatkowo $a \leqslant \sqrt{n}$. Gdyby tak nie było, to $ab > n$. To daje od razu następujący algorytm:

```
x := sqrt(n);
czyJestPierwsza := TAK;
for a := 2 to x do
  if (n mod a = 0) then czyJestPierwsza := NIE;
```

Złożoność powyższego algorytmu to $O(\sqrt{n})$.

2.2 Prawdziwe zadanie

Mając już takie narzędzie w ręku, jesteśmy w stanie rozwiązać nasze zadanie. Dla każdej liczby z zakresu sprawdzamy, czy jest pierwsza, następnie czy ta liczba bez ostatniej cyfry jest pierwsza itd.

Takie rozwiązanie jest jednak dużo za wolne. Spróbujmy je przyspieszyć. W tym celu mogą nam się przydać następujące spostrzeżenia:

Spostrzeżenie 1.
Jedynymi jednocyfrowymi *skracalnymi liczbami pierwszymi* są 2, 3, 5, 7.

Spostrzeżenie 2.
Jeśli *skracalna liczba pierwsza* jest co najmniej dwucyfrowa, to musi kończyć się na 1, 3, 7, 9.
Gdyby tak nie było, to byłaby podzielna albo przez 2, albo przez 5.

Korzystając z powyższych spostrzeżeń, możemy naszkicować algorytm, którego zadaniem będzie wygenerowanie wszystkich *skracalnych liczb pierwszych*. Będziemy korzystać z kolejki, w której będziemy trzymać *skracalne liczby pierwsze*. Na początku znajdą się w niej 2, 3, 5, 7. Za każdym razem bierzemy element z kolejki, dodajemy go do zbioru *skracalnych liczb pierwszych*, a następnie próbujemy tę liczbę wydłużyć (doklejając na jej koniec jedną z cyfr: 1, 3, 7, 9), po czym sprawdzamy, czy liczba, jaką uzyskaliśmy, jest pierwsza, i w przypadku pozytywnej odpowiedzi dodajemy ją do kolejki. Liczba ta będzie faktycznie *skracalną liczbą pierwszą*, gdyż liczba ta jest pierwsza, a wszystkie jej prefiksy są liczbami pierwszymi (to sprawdziliśmy już wcześniej).

Pseudokod wygląda następująco:

```
kolejka := {2, 3, 5, 7};
while (kolejka nie pusta) do
  pobierz element x z kolejki;
  dodaj x do listy skracalnych liczb pierwszych;
  if (czyJestPierwsza(10 * x + 1)) then dodajDoKolejki(10 * x + 1);
  if (czyJestPierwsza(10 * x + 3)) then dodajDoKolejki(10 * x + 3);
  if (czyJestPierwsza(10 * x + 7)) then dodajDoKolejki(10 * x + 7);
  if (czyJestPierwsza(10 * x + 9)) then dodajDoKolejki(10 * x + 9);
done;
```

Uruchamiając program oparty na tym algorytmie, jesteśmy w stanie sprawdzić, ile jest wszystkich *skracalnych liczb pierwszych*. W przypadku, gdy kolejka jest pusta, oznacza to, że nie ma już żadnego kandydata, którego moglibyśmy rozszerzyć o jedną cyfrę, czyli nie ma już żadnej większej *skracalnej liczby pierwszej*.

Na pierwszy rzut oka wydaje się, że zaproponowany algorytm może działać dosyć wolno i wygenerować bardzo dużo interesujących nas liczb...

Jednak po wykonaniu tego algorytmu otrzymujemy, że skracalnych liczb pierwszych jest 83, a największa z nich nie przekracza 10^8. Możemy więc wygenerować wszystkie szukane liczby, a następnie wpisać je w swoje rozwiązanie (tablica 83 stałych), lub też generować je na bieżąco.

3 Wyzwanie

Warto spojrzeć na zadanie *Liczby antypierwsze* z VIII Olimpiady Informatycznej, które można również próbować rozwiązywać w pokazany powyżej sposób.

Zadanie: URZ
Urzędnicy

OLIMPIADA INFORMATYCZNA GIMNAZJALISTÓW

V OIG, etap I. Plik źródłowy `urz.*` Dostępna pamięć: 32 MB. *10.01–7.02.2011*

W Bajtocji ostatnimi czasy nie dzieje się najlepiej. Do władzy doszedł opanowany obsesyjnym strachem o swoje życie król Bitogrom. Już w kilka dni po objęciu tronu ukazał on swoje bezwzględne oblicze, ścinając pięciu dworzan podejrzanych o spiskowanie przeciw niemu. Na wszystkich urzędników w państwie padł strach o własne życie. Mieli oni świadomość, że każdy donos przełożonego prowadzi do szybkiej egzekucji. Sprawę pogarszał fakt, że donosiciel stawał się zaufanym człowiekiem króla, któremu tym samym nie groził już wyrok skazujący. W zastraszonym środowisku urzędników państwowych była to wystarczająca motywacja, żeby donieść na któregoś ze swoich podwładnych.

Sytuacja w urzędach bardzo zmartwiła profesora Bajtoszewskiego, który przewidywał związane z nią utrudnienia w działaniu sektorów państwowych. Poprosił Cię, abyś obliczył, ilu maksymalnie urzędników może zostać straconych wskutek donosów. Profesor wyjaśnił Ci dokładniej zasady funkcjonowania państwa:

- Każdy z n urzędników w państwie ma unikatowy identyfikator będący liczbą całkowitą z przedziału $[1, n]$.
- Każdy przełożony ma numer mniejszy od numerów wszystkich swoich podwładnych.
- Przełożonym wszystkich urzędników jest premier Bajtocji, który ma numer 1 i, tym samym, nie ma przełożonego.
- Każdy urzędnik donosi na co najwyżej jednego ze swoich podwładnych, ponieważ po pierwszym donosie jest on już zaufanym człowiekiem króla.
- W Bajtocji panuje zasada: „podwładny mojego podwładnego jest moim podwładnym”, co w praktyce oznacza, że urzędnik może donieść na urzędnika, dla którego jest przełożonym tylko pośrednio.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 1\,000\,000$) oznaczająca liczbę urzędników. W drugim wierszu znajduje się $n-1$ liczb całkowitych, z których i-ta oznacza numer przełożonego urzędnika o numerze $i+1$.

W testach wartych łącznie co najmniej 40% punktów zachodzi dodatkowo warunek $n \leqslant 1\,000$.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia powinna znaleźć się jedna liczba całkowita, będąca maksymalną liczbą urzędników, którzy mogą zostać straceni w wyniku donosów.

Przykład

Dla danych wejściowych:

```
4
1 2 2
```

poprawnym wynikiem jest:

```
2
```

Wyjaśnienie do przykładu: Urzędnik numer 1 donosi na urzędnika numer 3, a urzędnik numer 2 na urzędnika numer 4.

Opracowanie: URZ
Urzędnicy

Rozwiązanie wzorcowe

Rozwiązanie tego zadania polega na zastosowaniu pewnej często powtarzającej się w programowaniu techniki — mianowicie programowania dynamicznego.

Trudno jest, patrząc na całą strukturę państwa Bajtocji, od razu stwierdzić, jaka jest maksymalna możliwa liczba ofiar terroru króla Bitogroma. Jednak zastanówmy się, czy można to zadanie podzielić na mniejsze podzadania, które stosunkowo łatwo można łączyć w coraz większe. Otóż zauważmy, że hierarchia urzędników w naturalny sposób dzieli się na coraz mniejsze części.

Weźmy na przykład dowolnego urzędnika u i załóżmy, że jego bezpośredni podwładni to $u_1, u_2, \ldots, u_k$. Każdy z nich ma zbiór swoich wszystkich (pośrednich i bezpośrednich) podwładnych, który razem z nim samym tworzy pewną specyficzną podstrukturę państwa — nazwijmy ją *poddrzewem* urzędnika u_i *. Szczególną cechą poddrzewa jest to, że ma ono bardzo podobną strukturę do całego drzewa, czyli całej hierarchii urzędników w państwie — mogłoby np. samo w sobie tworzyć dane wejściowe w niniejszym zadaniu. Dalej, każdy z bezpośrednich podwładnych u może mieć własnych bezpośrednich podwładnych, którzy mają własne poddrzewa. I tak struktura państwa tworzy drzewo zaczynające się od premiera, każdy z jego bezpośrednich podwładnych ma własne poddrzewo itd. aż dojdziemy do urzędników na samym dole hierarchii, których poddrzewa są złożone z nich samych.

Aby posłużyć się programowaniem dynamicznym, najpierw musimy uogólnić nasz problem obliczeniowy. Zamiast pytać się o maksymalną liczbę ofiar w całym państwie, spróbujmy obliczyć maksymalną liczbę donosów w każdym poddrzewie. Niech `rozw[`u`]` oznacza maksymalną liczbę ofiar w poddrzewie urzędnika u. Chcemy więc nie tylko obliczyć `rozw[1]` (co jest odpowiedzią w zadaniu), ale też `rozw[`u`]` dla wszystkich innych urzędników u.

Dlaczego niby uogólnienie problemu ma nam pomóc w jego rozwiązaniu? Otóż dlatego, że teraz wystarczy nam znalezienie sposobu na obliczanie `rozw[`u`]` w dwóch prostych przypadkach:

- kiedy u nie ma żadnych podwładnych;
- kiedy podwładnymi u są $u_1, u_2, \ldots, u_k$, $k \geqslant 1$ oraz znamy wartości `rozw[`u_1`]`, `rozw[`u_2`]`, ..., `rozw[`u_k`]`.

Jeśli znajdziemy metodę na rozwiązanie tych dwóch mniejszych zadań, to wystarczy, jeśli przejrzymy wszystkich urzędników od największych numerów do numeru 1 — wtedy przeglądając każdego kolejnego, znajdziemy się w jednej z powyższych sytuacji. W ten sposób będziemy mogli budować rozwiązania małych problemów, otrzymując rozwiązania coraz większych, aż na końcu rozwiążemy cały wielki problem, postawiony w zadaniu.

Jak więc możemy obliczyć `rozw[`u`]` w powyższych dwóch przypadkach? Jeśli u nie ma żadnych podwładnych, to wtedy oczywiście `rozw[`u`]` $= 0$, ponieważ nie doniesie on sam na siebie. Załóżmy teraz, że u ma bezpośrednich podwładnych $u_1, u_2, \ldots, u_k$, $k \geqslant 1$ i przyjmijmy, że realizujemy scenariusz „zapisany" w każdej z komórek tablicy `rozw[]` odpowiadających bezpośrednim podwładnym u. Oznaczmy $s =$ `rozw[`u_1`]` $+ \ldots +$ `rozw[`u_k`]`. Wtedy mamy dwie możliwości:

- Każdy urzędnik w poddrzewie u (oprócz niego samego) doniósł na kogoś bądź był ofiarą donosu. Jest to równoznaczne temu, że liczba $2s + 1$ jest rozmiarem całego poddrzewa u (liczbą urzędników w nim się znajdujących). Wtedy rozmiar ten jest nieparzysty i niemożliwym jest, by w tym drzewie dokonano więcej niż s donosów — możemy więc wtedy bezpiecznie przyjąć, że u nie donosi na nikogo, oraz `rozw[`u`]` $= s$.
- W przeciwnym wypadku liczba $2s + 1$ jest mniejsza niż rozmiar całego poddrzewa, a więc jest w nim urzędnik, na którego u może donieść, dając sumaryczną liczbę donosów równą $s + 1$. Większej liczby donosów w tym poddrzewie nie można osiągnąć, bo to by oznaczało, że w pewnym poddrzewie u_i można osiągnąć więcej ofiar niż `rozw[`u_i`]`.

Otrzymujemy więc prosty sposób na obliczenie `rozw[`u`]`. Jeśli $2s+1$ jest równe rozmiarowi poddrzewa u, to `rozw[`u`]` $= s$, a w przeciwnym wypadku `rozw[`u`]` $= s + 1$.

Pozostaje jeszcze problem szybkiego obliczania wielkości poddrzew. Nachalne przechodzenie całego poddrzewa każdego urzędnika zajmie w sumie czas kwadratowy ze względu na n, co jest zbyt kosztowne. Jednak i tutaj możemy posłużyć się programowaniem dynamicznym. Oznaczmy przez `size[`u`]` rozmiar poddrzewa u i rozpatrzmy dwa przypadki, takie jak poprzednio:

- u nie ma żadnych podwładnych — wtedy `size[`u`]` $= 1$;
- bezpośrednimi podwładnymi u są $u_1, u_2, \ldots, u_k$, $k \geqslant 1$ — wtedy `size[`u`]` $= 1 +$ `size[`u_1`]` $+ \ldots +$ `size[`u_k`]`.

W ten sposób otrzymujemy rozwiązanie o koszcie czasowym $O(n)$.

*Motywacja do takiej nazwy wypływa ze struktury całej administracji w państwie; jeżeli bowiem narysujemy na kartce wszystkich urzędników od 1 do n od góry do dołu kartki i poprowadzimy linie pomiędzy każdym urzędnikiem (oprócz premiera) i jego szefem, to powstanie rysunek przypominający odwrócone drzewo; zbiór wszystkich podwładnych zadanego urzędnika ma podobną strukturę, która zawiera się w strukturze całego państwa — dlatego nazywamy ją poddrzewem.

Zadanie: WYS
Wyścig

OLIMPIADA INFORMATYCZNA GIMNAZJALISTÓW

V OIG, etap I. Plik źródłowy `wys.*` Dostępna pamięć: 64 MB. *10.01–7.02.2011*

Wyścig Tour de Bajtocja jest organizowany co roku na trasie z miasta A do miasta B. Ze względu na dziurę budżetową, w tym roku wyścig odbędzie się tylko na pewnym odcinku trasy. Nie jest jeszcze ustalone, jaki to będzie odcinek, choć ustalona jest już jego długość.

Na całej trasie rozstawione są znaki ograniczające prędkość jazdy. Ograniczenie obowiązuje do momentu zmiany tego ograniczenia przez inny znak. Wyścig Tour de Bajtocja znany jest z obowiązku przestrzegania ograniczeń prędkości.

Organizatorzy zastanawiają się, jaki fragment trasy (o długości m) wybrać, aby przestrzegając ograniczeń prędkości, można było go jak najszybciej przejechać.

Zostałeś poproszony o napisanie programu, który wyznaczy najkrótszy czas przejechania takiego fragmentu trasy.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite n, m oraz d ($1 \leqslant n \leqslant 1\,000\,000$, $1 \leqslant m \leqslant d \leqslant 10^9$), pooddzielane pojedynczymi odstępami, oznaczające odpowiednio liczbę znaków ustawionych na trasie, długość odcinka, na którym powinien odbyć się wyścig, oraz długość trasy z A do B.

Następne n wierszy zawiera opisy kolejnych znaków ustawionych na trasie. Opis znaku składa się z dwóch liczb całkowitych s_i, v_i ($0 \leqslant s_i \leqslant d$, $1 \leqslant v_i \leqslant 1\,000\,000$), oddzielonych pojedynczym odstępem, oznaczających odpowiednio odległość i-tego znaku od miasta A oraz ograniczenie prędkości obowiązujące od ustawienia tego znaku. Możesz założyć, że $0 = s_1 < s_2 < \ldots < s_n$.

W testach wartych przynajmniej 50% punktów zachodzą dodatkowe warunki $n \leqslant 1\,000$ oraz $d \leqslant 1\,000\,000$.

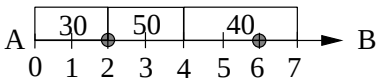
Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia Twój program powinien wypisać jedną liczbę rzeczywistą zaokrągloną do dokładnie trzech cyfr po kropce dziesiętnej, oznaczającą najkrótszy możliwy czas przejechania trasy długości m. Wybierany odcinek trasy nie może wykraczać poza trasę z miasta A do miasta B.

Przykład

Dla danych wejściowych:

```
3 4 7
0 30
2 50
4 40
```



poprawnym wynikiem jest:

```
0.090
```

Wyjaśnienie do przykładu: Optymalna trasa zaczyna się w odległości 2 od miasta A. Czas przejechania tej trasy jest równy $\frac{2}{50} + \frac{2}{40} = \frac{9}{100}$.

Wskazówka: Aby uniknąć błędów zaokrągleń, do obliczeń polecamy używać typów rzeczywistych podwójnej precyzji (`double`) oraz standardowych procedur/funkcji służących do wypisywania liczb rzeczywistych z zadaną precyzją.

Opracowanie: WYS
Wyścig

1 Rozwiązanie brutalne

Najprostszym rozwiązaniem zadania jest rozpatrzenie wszystkich możliwych tras długości m i dla każdej z nich obliczenie czasu potrzebnego na jej przejazd.

Do takiego rozwiązania przydałaby się tablica, z której moglibyśmy odczytywać ograniczenie prędkości na jednostkowym kawałku trasy. Wyznaczenie takiej tablicy jest stosunkowo proste.

Możemy przy każdym wczytanym znaku uaktualniać czas przejazdu od miejsca ustawienia znaku, aż do końca trasy.

Pseudokod takiego rozwiązania mógłby wyglądać następująco:

```
wczytaj(n, m, d);
wynik := INF;
for i := 1 to n do begin
  wczytaj(s, v);
  for j := s to d - 1 do
    predkosc[j] := v;
end;
for i := 0 to d - m do begin
  akt := 0.0;
  for j := i to i + m - 1 do
    akt := akt + 1.0 / predkosc[j];
  if akt < wynik then
    wynik := akt;
end;
wypisz(wynik);
```

Takie rozwiązanie działa zdecydowanie za wolno. Złożoność czasowa wynosi $O(d(d+n+m))$, gdyż zbudowanie tablicy z ograniczeniami prędkości zajmuje czas $O(dn)$, a obliczenie czasu potrzebnego na przejazd każdej możliwej trasy wymaga wykonania $O(dm)$ operacji.

2 Rozwiązanie powolne

Załóżmy, że mamy obliczony czas potrzebny na przejechanie odcinka $(i, i+m)$. Aby obliczyć czas przejazdu dla odcinka $(i+1, i+m+1)$, możemy wykorzystać obliczony czas dla powtarzającej się części. Powtarzającą się częścią będzie $(i+1, i+m)$. Wystarczy więc, że odejmiemy czas przejechania odcinka $(i, i+1)$ oraz dodamy czas odcinka $(i+m, i+m+1)$.

Aby polepszyć złożoność całego rozwiązania należałoby jeszcze ulepszyć generowanie tablicy do odczytywania prędkości na jednostkowym kawałku trasy.

Można to zrobić w dość prosty sposób. Na początku zapisujemy prędkości tylko dla jednostkowych kawałków trasy, zaczynających się w miejscu ustawiania znaków. Następnie przechodzimy całą trasę i uaktualniamy prędkości dla pozostałych miejsc.

Pseudokod mógłby wyglądać następująco:

```
wczytaj(n, m, d);
for i := 1 to n do begin
  wczytaj(s, v);
  predkosc[s] := v;
end;
for i := 0 to d - 1 do
  if predkosc[i] = 0 then
    predkosc[i] := predkosc[i - 1];
akt := 0.0;
for i := 0 to m - 1 do
  akt := akt + 1.0 / predkosc[i];
wynik := akt;
for j := m to d - 1 do begin
  akt := akt - 1.0 / predkosc[j - m] + 1.0 / predkosc[j];
  if akt < wynik then
```

```
    wynik := akt;
end;
wypisz(wynik);
```

Złożoność takiego rozwiązania poprawia się do $O(n+d)$, gdyż ustawienie wszystkich znaków zajmuje czas $O(n)$, a wygenerowanie tablicy z ograniczeniami prędkości i łączne obliczanie czasów przejazdów wszystkich poprawnych odcinków to czas rzędu $O(d)$.

3 Rozwiązanie wzorcowe

Zauważmy, że jedyne fragmenty trasy, które warto rozpatrywać, to te, które kończą się lub rozpoczynają w miejscu zmiany ograniczenia prędkości.

Faktycznie, rozważmy fragment, który nie zaczyna ani nie kończy się w miejscu ustawienia znaku. Wtedy jeśli ograniczenie prędkości na jednym kawałku jest mniejsze niż na drugim, to możemy przesuwać fragment, poprawiając czas przejechania trasy. Jeśli natomiast ograniczenie prędkości jest takie same, to przesuwając fragment w jedną ze stron, nie pogorszymy czasu przejechania tego fragmentu, a będzie się on zaczynał lub kończył w miejscu ustawienia znaku.

Początki wszystkich fragmentów możemy wyznaczyć podczas wczytywania danych. Pseudokod ich wyznaczenia może wyglądać następująco.

```
wczytaj(n, m, d);
for i := 1 to n do begin
  wczytaj(s, v);
  if s - m > 0 then
    nowyPoczatek(s - m);
  if s + m < d then
    nowyPoczatek(s);
end;
// dodajemy fragment, który kończy się w miejscu zakończenia całej trasy
nowyPoczatek(d - m);
```

Aby móc efektywnie rozwiązać zadanie, powinniśmy uporządkować niemalejąco początki tras. Można je posortować sortowaniem szybkim w czasie $O(n \log n)$ lub zauważyć, że powstaną nam dwa posortowane ciągi, które możemy scalić w czasie $O(n)$.

Mając posortowany ciąg wszystkich początków fragmentów tras, możemy przejść do obliczania wyniku. Zrobimy to podobnie jak w rozwiązaniu powolnym — do obliczania czasu pomiędzy dwoma fragmentami wykorzystamy powtarzające się fragmenty tras (odejmując początek i dodając koniec). Należy tylko skakać o większe fragmenty — do wyznaczonych początków tras. Dopracowanie szczegółów technicznych tego skakania pozostawiamy Czytelnikowi.

Złożoność takiego rozwiązania jest liniowa. Wszystkich początków tras będzie maksymalnie $2n$, więc przeglądając tylko takie fragmenty i uaktualniając je w czasie stałym, mamy liniowy czas rozwiązania całego rozwiązania.

Zadanie: ZAP
Zapałki

OLIMPIADA INFORMATYCZNA GIMNAZJALISTÓW

V OIG, etap I. Plik źródłowy `zap.*` **Dostępna pamięć: 32 MB.** *10.01–7.02.2011*

Bajtek bawi się zapałkami. Na jednym z końców zapałki znajduje się główka pokryta masą ułatwiającą zapłon. Bajtek ułożył zapałki w linii prostej jedna obok drugiej, w taki sposób, że każdy koniec zapałki sąsiaduje z końcem pewnej innej zapałki, oprócz dwóch skrajnych zapałek, które sąsiadują tylko jednym końcem.

Przykładowe ułożenie zapałek.

Bajtek chciałby podpalić pierwszą zapałkę (skrajną z lewej) tak aby wszystkie zapałki spaliły się. Pierwszą zapałkę zapali on przy użyciu zapalniczki, może więc to zrobić bez względu na jej ułożenie. Natomiast między kolejnymi zapałkami ogień przeniesie się tylko, jeśli co najmniej jedna z tych zapałek w miejscu połączenia będzie zwrócona główką. Zastanawiamy się, ile minimalnie zapałek musimy odwrócić, aby wszystkie zapałki spaliły się, jeśli podpalimy pierwszą zapałkę.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 1\,000\,000$) oznaczającą liczbę zapałek Bajtka. Drugi wiersz opisuje ułożenie kolejnych zapałek — zawiera ciąg n liczb całkowitych $x_1, x_2, \ldots, x_n$, przy czym x_i oznacza zwrot i-tej zapałki w ciągu: 0 jeśli główka zapałki znajduje się z lewej strony, zaś 1 jeśli główka zapałki znajduje się z prawej strony.

W testach wartych łącznie co najmniej 50% punktów zachodzi dodatkowo warunek $n \leqslant 10\,000$.

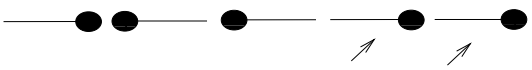
Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą równą minimalnej liczbie zapałek, jakie należy odwrócić.

Przykład

Dla danych wejściowych:

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



poprawnym wynikiem jest:

```
2
```

Opracowanie: ZAP
Zapałki

1 Wprowadzenie

Rozwiązanie zadania opiera się na obserwacji, że poprawne ustawienia zapałek mogą być tylko takie, że w pewnym miejscu sąsiadują dwie główki zapałek, a następnie z obydwu stron wszystkie zapałki zwrócone są w tym samym kierunku.

Dlaczego tylko takie przypadki są poprawne? Rozważmy miejsce i, w którym sąsiadują dwie główki zapałek. Zauważmy, że na pozycji $i+1$ oraz $i-1$ musi znajdować się co najmniej jedna główka zapałki. Jest tylko jedna możliwość ustawienia sąsiednich zapałek w taki sposób. W związku z tym każda następna, sąsiednia zapałka, którą chcemy dołożyć, musi być ustawiona w tym samym kierunku. To pokazuje, że może być maksymalnie jedna para zapałek zwróconych do siebie główkami.

Pozostają przypadki, w których nie ma żadnej takiej pary. Poprawne ustawienie otrzymujemy tylko wtedy, gdy wszystkie zapałki są ułożone w tym samym kierunku. Można przyjąć, że w tym przypadku dwie główki zapałek sąsiadują na pozycji 0 albo n.

2 Rozwiązanie powolne

Pierwszym, narzucającym się rozwiązaniem może być rozważenie wszystkich $n+1$ przypadków. Zakładamy, że na kolejnych pozycjach: $0, 1, 2, \ldots, n$ sąsiadują dwie główki zapałek, i w czasie liniowym obliczamy liczbę zapałek, które musimy obrócić, aby doprowadzić do takiej sytuacji.

Zaimplementowanie takiego rozwiązania jest stosunkowo proste. Jeśli rozważamy i-tą pozycję, to pierwsze i zapałek musi być obrócone główką w prawo, a wszystkie następne w lewo. Wystarczy więc zliczyć te zapałki, które są nieodpowiednio obrócone.

Przykładowy pseudokod mógłby wyglądać następująco:

```
wczytaj(n, zapalki[]);
wynik := n;
for i := 0 to n do begin
  ile := 0;
  for j := 1 to i do
    // zwiększam, jeśli zapałka jest obrócona w lewo
    ile := ile + 1 - zapalki[j];
  for j := i + 1 to n do
    // zwiększam, jeśli zapałka jest obrócona w prawo
    ile := ile + zapalki[j];
  wynik := min(wynik, ile);
end;
wypisz(wynik);
```

Takie rozwiązanie jest niestety za wolne. Jego złożoność wynosi $O(n^2)$, ponieważ dla każdej pozycji sąsiednich główek zapałek, szukamy liniowo liczby zapałek, które musimy obrócić. Zgodnie z treścią zadania, za takie rozwiązanie można było uzyskać 50% punktów.

3 Rozwiązanie wzorcowe

Zastanówmy się, jak dla pozycji i odpowiedzieć szybko na pytania:

- ile z pierwszych i zapałek jest obróconych w lewo?
- ile z ostatnich $n-i$ zapałek jest obróconych w prawo?

Jeśli umielibyśmy odpowiadać na takie pytania szybciej niż liniowo, to moglibyśmy to wykorzystać do zliczenia źle obróconych zapałek, zamiast robić to w brutalny sposób jak powyżej.

Wygenerowanie takich odpowiedzi jest całkiem proste. Możemy je obliczyć i zapamiętać zaraz po wczytaniu danych. Wyznaczymy tzw. sumy częściowe prefiksowe i sufiksowe.

W komórce `pref[i]` będziemy pamiętać, ile z i pierwszych zapałek jest obróconych w lewo:

```
pref[0] := 0;
for i := 1 to n do
  pref[i] := pref[i - 1] + 1 - zapalki[i];
```

W komórce `suf[i + 1]` będziemy pamiętać, ile z $n - i$ ostatnich zapałek jest obróconych w prawo:

```
suf[n + 1] := 0;
for i := n downto 1 do
  suf[i] := suf[i + 1] + zapalki[i];
```

Mając wyliczone tablice `pref[]` i `suf[]`, główną pętlę obliczającą wynik możemy zapisać następująco:

```
for i := 0 to n do
  wynik := min(wynik, pref[i] + suf[i + 1]);
```

Złożoność czasowa takiego rozwiązania wynosi $O(n)$, gdyż dla każdej pozycji sąsiednich główek zapałek, w czasie $O(1)$ obliczamy liczbę zapałek, które musimy obrócić. Czas potrzebny na wyznaczenie tablic `pref[]` i `suf[]` wynosi również $O(n)$. Złożoność pamięciowa tego rozwiązania jest liniowa.

OLIMPIADA
INFORMATYCZNA
GIMNAZJALISTÓW

Etap II

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

MINISTERSTWO
EDUKACJI
NARODOWEJ

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany ze środków Unii Europejskiej
w ramach Europejskiego Funduszu Społecznego.

Zadanie: MON
Monety

OLIMPIADA INFORMATYCZNA GIMNAZJALISTÓW

V OIG, etap II. Plik źródłowy `mon.*` Dostępna pamięć: 16 MB. *02.04.2011*

Bajtazar jest niezwykle dumny ze swojej kolekcji rzadkich monet. Zbierał je przez wiele lat, dbając o to, by żadne dwie nie były podobne. Obecnie ma n monet ponumerowanych w taki sposób, że i-ta moneta ma rozmiar dokładnie i.

Jako że kolekcja Bajtazara ostatnio powiększyła się, był on zmuszony kupić nowy klaser. Jest w nim dokładnie n przegród na monety, każda o określonym rozmiarze. Oczywiście żadnej monety nie można włożyć do zbyt małej przegrody. Nic nie stoi jednak na przeszkodzie, by włożyć ją do przegrody większej.

Bajtazar zastanawia się teraz, do których przegród włożyć poszczególne monety. Po sprawdzeniu wielu kombinacji zaintrygowało go również pytanie, na ile sposobów może zapełnić klaser. Ponieważ liczba ta może być bardzo duża, Bajtazarowi wystarczy jej reszta z dzielenia przez $10^9 + 7$. Napisz program, który zaspokoi jego ciekawość.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 1\,000\,000$). W następnym wierszu znajduje się n liczb całkowitych a_i ($1 \leqslant a_i \leqslant n$) pooddzielanych pojedynczymi odstępami. Liczba a_i oznacza, jaką największą monetę można włożyć do i-tej przegrody.

Możesz założyć, że w testach wartych co najmniej 50% punktów zachodzi dodatkowo warunek: $n \leqslant 1000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście jedną liczbę całkowitą — resztę z dzielenia liczby sposobów zapełnienia klasera przez $10^9 + 7$. Jeśli nie istnieje żaden sposób zapełnienia klasera monetami, prawidłowym wynikiem jest 0.

Przykład

Dla danych wejściowych:

```
4
4 2 4 2
```

poprawnym wynikiem jest:

```
4
```

Opracowanie: MON
Monety

1 Analiza problemu

Przypomnijmy, że ciąg a_i ($1 \leqslant i \leqslant n$) oznacza rozmiary przegród na monety. Jako, że monety Bajtazara mają wielkości dokładnie $1, 2, \ldots, n$, zadanie sprowadza się do obliczenia wszystkich kolejności liczb $1, 2, \ldots, n$, takich, że i-ta z kolei liczba jest mniejsza lub równa a_i.

2 Rozwiązanie brutalne

Najprostszym rozwiązaniem jest wygenerowanie wszystkich kolejności liczb $1, 2, \ldots, n$ (czyli permutacji) i dla każdej z nich sprawdzenie powyższego warunku. Niestety liczba permutacji zbioru złożonego z n elementów wynosi $n!$, czyli $1 \cdot 2 \cdot \ldots (n-1) \cdot n$, co nawet dla niewielkich n może być ogromną liczbą ($10! = 3628800$, $15! = 1307674368000$). Można co prawda przerywać obliczenia, jeśli zauważymy, że któraś nierówność nie jest spełniona i nie generować więcej permutacji, ale nawet takie rozwiązanie nie ma szans zmieścić się w limitach czasowych.

3 Rozwiązanie wzorcowe

Jak widać musimy być sprytniejsi. Najpierw zauważmy, że wynik nie zależy od kolejności liczb a_i. Dlaczego tak jest? Jak już powiedzieliśmy kolejność liczb $1, 2, \ldots, n$ opisuje poprawne zapełnienie klasera wtedy i tylko wtedy gdy i-ta z kolei liczba jest mniejsza lub równa a_i. Wyobraźmy sobie sytuację, że zamieniamy między sobą pewne przegródki, ale wraz z nimi przesuwamy monety, które próbowaliśmy do nich włożyć. Wtedy poprawne przyporządkowanie oczywiście się nie popsuje, a niepoprawne nadal pozostanie niepoprawne.

Skoro nie musimy dbać o kolejność liczb a_i, to możemy przeglądać je w najwygodniejszej kolejności czyli niemalejącej. Do najmniejszej przegródki możemy włożyć oczywiście a_1 najmniejszych monet. Do drugiej przegródki zmieści się a_2 monet, ale pamiętajmy, że jedną już zużyliśmy, zatem pierwsze 2 przegródki zapełnimy na $a_1(a_2 - 1)$ sposobów. Może dalej będzie podobnie?

Załóżmy, że $k-1$ pierwszych przegródek jest już zapełnione. Każdą niewykorzystaną monetę mniejszą lub równą a_k możemy włożyć na k-te miejsce, a takich monet jest $a_k - k + 1$. Dzięki temu otrzymujemy elegancki wzór na wynik: $a_1(a_2 - 1)(a_3 - 2) \cdots (a_n - n + 1)$. Zauważmy, że klasera nie da się zapełnić tylko wtedy, gdy $a_k < k$ dla pewnego k. Zgadza się to z naszym wzorem, bo kolejne czynniki mogą zmniejszać się maksymalnie o 1 i $a_k < k$ pociąga pojawienie się 0 w iloczynie.

Zanim zaczniemy pisać rozwiązanie zwróćmy uwagę na haczyk — jeśli po prostu wymnożymy powyższe n liczb, możemy wyjść poza zakres liczb 64-bitowych! Aby tego uniknąć możemy po każdym mnożeniu pamiętać tylko resztę z dzielenia wyniku przez $10^9 + 7$. Oto pseudokod powyższego rozwiązania (a `modulo` b oznacza resztę z dzielenia liczby a przez b).

```
sort(a[1]...a[n]);
wynik = 1;
dzielnik = 1 000 000 007;

for i := 1 to n do
    wynik = (wynik * (a[i] - i + 1)) modulo dzielnik;

return wynik;
```

Przedstawiony algorytm działa w czasie $O(n)$ czyli liczba operacji jest proporcjonalna do n — o wiele szybciej niż pierwszy pomysł. Nie wolno zapomnieć o czasie sortowania liczb a_i — tu możemy użyć sortowania przez zliczanie, które również działa w czasie $O(n)$. Takie rozwiązanie można znaleźć w plikach `mon.cpp` oraz `mon.pas`. Jeśli użyjemy sortowania w czasie $O(n \log n)$, otrzymamy rozwiązanie niewiele wolniejsze.

4 Rozwiązanie alternatywne

Drugie rozwiązanie jest nieco trudniejsze do zrozumienia, ale również jest ciekawe. Polega na odwróceniu sposobu myślenia. Otóż zamiast zastanawiać się, ile monet możemy włożyć do jednego pojemnika, możemy zliczać, do ilu pojemników może trafić dana moneta. Oznaczmy przez $F(k)$ liczbę sposobów włożenia k najmniejszych monet do k najmniejszych przegródek. Wynikiem działania programu będzie oczywiście $F(n)$. Aby jednak otrzymać algorytm, musimy znaleźć sposób na obliczanie kolejnych wartości $F(k)$.

Najpierw posortujmy przegródki niemalejąco, podobnie jak w poprzednim rozwiązaniu. Początek obliczania funkcji F jest prosty: $F(1) = 1$, ponieważ $a_1 \geqslant 1$. Załóżmy, że znamy $F(k-1)$. Jak wtedy policzyć $F(k)$? Włożyliśmy $k-1$ monet i musimy znaleźć miejsce na monetę wielkości k. Możliwych miejsc jest tyle, ile przegródek wielkości co najmniej k wśród pierwszych (czyli najmniejszych) k przegródek — oznaczmy tę liczbę przez b_k. Jeśli $b_k = 0$, to widać, że się nam nie powiedzie i możemy przerwać obliczenia. Załóżmy więc, że $b_k > 0$. A co zrobić z monetą, która była wcześniej na wybranym miejscu? Ma ona rozmiar mniejszy od k, więc możemy wstawić ją w k-tą przegródkę (tutaj korzystamy z tego, że posortowaliśmy dane!).

Zatem z każdej poprawnej permutacji pierwszych $k-1$ monet otrzymujemy b_k poprawnych permutacji pierwszych k monet. Trzeba tutaj pokazać, że w ten sposób nie stworzymy dwukrotnie tej samej permutacji, ale zostawiamy to jakie łatwe ćwiczenie dla czytelnika. Daje nam to wzór $F(k) = b_k F(k-1)$ dla $k > 1$.

Zatem wynik to po prostu $b_2 \cdot b_3 \cdots b_n$ — zgadza się to z obserwacją, że jeśli któreś b_i jest równe 0 to nie uda nam się w żaden sposób wypełnić klasera.

Pozostaje kwestia wyznaczenia liczb b_i. Niech c_i oznacza pierwszą pozycję, na której $a_{c_i} \geqslant i$ (pamiętajmy, że ciąg a_i jest posortowany niemalejąco). Zauważmy, że $b_i = i - c_i + 1$. Liczby c_i można już policzyć w bardzo prosty sposób:

```
j := 1;
for i := 1 to n do
begin
    while j <= a[i] do
    begin
        c[j] := i;
        j := j + 1;
    end;
end;
```

W przypadku, kiedy liczby `a[i]` są przechowywane w kubełkach, to obliczanie wartości `c[]` jest jeszcze prostsze. To rozwiązanie również działa w czasie $O(n)$. Zaimplementowano je w plikach `mon2.cpp` oraz `mon2.pas`.

Zadanie: OBW
Obwarzanki

OLIMPIADA INFORMATYCZNA GIMNAZJALISTÓW

V OIG, etap II. Plik źródłowy `obw.*` Dostępna pamięć: 32 MB. *02.04.2011*

Witek wybrał się na jarmark. W mgnieniu oka zlokalizował stoisko z najlepszymi obwarzankami. Nie zastanawiając się długo, kupił jedną porcję obwarzanków. Warto wiedzieć, że Bajtockie obwarzanki są zawsze nawlekane na patyk, a nie na kółko, jak na większości naszych jarmarków. Obwarzanki można zdejmować z lewego bądź prawego końca patyka. Każdego obwarzanka charakteryzują dwie wartości: średnica zewnętrzna (Z) i średnica wewnętrzna (W). Jest to przedstawione na rysunku.

Gdy chcemy wyjąć pewien obwarzanek znajdujący się pomiędzy innymi obwarzankami, to możemy spróbować przełożyć jednego obwarzanka przez drugiego. Uda nam się to tylko wtedy, gdy średnica wewnętrzna któregoś z tych dwóch obwarzanków jest nie mniejsza od średnicy zewnętrznej drugiego z nich. W przeciwnym wypadku musimy najpierw zdjąć obwarzanka z lewej bądź prawej strony.

Witkowi spodobał się pewien obwarzanek i zastanawia się, ile minimalnie innych obwarzanków będzie musiał zdjąć, zanim dostanie się do swojego wybranego.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n oraz m ($1 \leqslant m \leqslant n \leqslant 1\,000\,000$), oddzielone pojedynczym odstępem i oznaczające odpowiednio liczbę obwarzanków znajdujących się na patyku oraz numer obwarzanka (licząc od lewej strony), którego wybrał Witek.

W n następnych wierszach znajdują się opisy kolejnych obwarzanków nawleczonych na patyk, począwszy od lewej strony. Każdy z tych wierszy zawiera dwie liczby całkowite w_i oraz z_i ($1 \leqslant w_i < z_i \leqslant 1\,000\,000\,000$) oddzielone pojedynczym odstępem, oznaczające odpowiednio średnicę wewnętrzną oraz średnicę zewnętrzną i-tego obwarzanka.

Możesz założyć, że w testach wartych co najmniej 35% punktów zachodzi dodatkowy warunek: $n \leqslant 20\,000$.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą równą minimalnej liczbie dodatkowych obwarzanków, jakie powinien zdjąć Witek, aby dostać się do wybranego. W wyniku nie należy uwzględniać obwarzanka wybranego przez Witka.

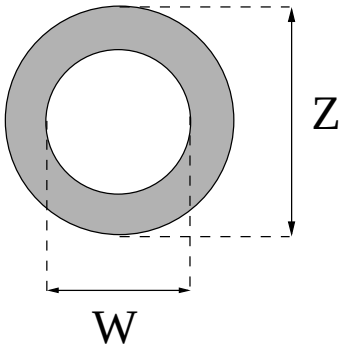
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
1
```



Opracowanie: OBW
Obwarzanki

1 Wstęp

Pierwszym krokiem w rozwiązaniu zadania jest prosta obserwacja, że jeśli będziemy próbowali przesunąć obwarzanek w jedną ze stron, to nie obchodzą nas obwarzanki występujące po drugiej stronie. W rozwiązaniu sprawdzamy, ile wynosi minimalna liczba obwarzanków, które musimy zdjąć, przesuwając wybrany obwarzanek oddzielnie w prawą i w lewą stronę, a następnie wybieramy minimum z tych wartości. Od tej pory bez straty ogólności zakładamy, że przesuwamy obwarzanek tylko w jedną stronę.

2 Rozwiązanie powolne

Jak obliczyć liczbę obwarzanków, które musimy zdjąć? Żeby zdjąć wybrany obwarzanek z patyka musimy go przesunąć do jednego z końców. Podczas pojedynczej próby przesunięcia mogą zdarzyć się dwie sytuacje:

- możemy zamienić miejscami dwa obwarzanki,
- wielkości ich zewnętrznych i wewnętrznych średnic nie pozwalają tego zrobić

W pierwszym przypadku, dosyć jasne jest, że jeśli możemy przesunąć obwarzanek, to robimy to — tym samym zmniejsza się liczba przeszkód na drodze obwarzanka.

Co powinniśmy zrobić w drugim przypadku? Ponieważ nie możemy zamienić miejscami tych dwóch obwarzanków, to musimy zdjąć je oba. W związku z tym powiększa nam się liczba obwarzanków do zdjęcia. Przyjmijmy więc, że w każdym momencie trzymać będziemy listę obwarzanków do zdjęcia.

Rozpatrzmy teraz dowolny kolejny obwarzanek, który napotykamy. Jeżeli ma on takie rozmiary, że każdy z obwarzanków już znajdujących się na liście można przez niego przełożyć, to nie ma problemu — może on pozostać na patyku, a więc nie dodajemy go do listy. Jeżeli natomiast na liście obwarzanków do zdjęcia jest przynajmniej jeden taki, którego nie można zamienić miejscami z aktualnie rozpatrywanym, to niestety rozpatrywany obwarzanek również musi się znaleźć na liście do usunięcia. Zauważmy, że kiedy skończymy przeglądanie patyka w ustalonym kierunku, to rozwiązaniem zadania będzie końcowa długość listy obwarzanków do usunięcia pomniejszona o 1.

Przyjrzyjmy się pseudokodowi rozwiązania, liczącego liczbę obwarzanków do zdjęcia przy założeniu, że przesuwamy obwarzanki w prawo (przesuwanie w lewo jest analogiczne):

```
wczytaj(n, m, obwarzanek[]);
dlugosc := 1;
lista[dlugosc] := obwarzanek[m];
{ przeglądamy wszystkich potencjalnych kandydatów do zdjęcia }
for i := m + 1 to n do begin
    czy_moge := true;
    { przeglądamy wszystkie obwarzanki dodane do listy }
    for k := 1 to dlugosc do begin
        if (not moge_przelozyc(lista[k], obwarzanek[i])) then
            czy_moge := false;
    end
    { aktualizujemy listę, jeśli potrzeba }
    if (not czy_moge) then begin
        dlugosc := dlugosc + 1;
        lista[dlugosc] := obwarzanek[i];
    end
end
wynik := dlugosc - 1;
```

Funkcja sprawdzająca czy można zamienić miejscami dwa obwarzanki:

```
bool moge_przelozyc(obwarzanek a, obwarzanek b)
    return (a.wewnetrzna >= b.zewnetrzna or b.wewnetrzna >= a.zewnetrzna)
```

Złożoność takiego rozwiązania wynosi $O(n^2)$, ponieważ może się zdarzyć, że każdy nowo napotkany obwarzanek będzie dodany do listy, a co więcej zanim się o tym dowiemy będziemy musieli przejrzeć wszystkie elementy listy. Przy ograniczeniach z treści zadania takie rozwiązanie jest zdecydowanie za wolne.

3 Rozwiązanie wzorcowe

Okazuje się, że nie trzeba pamiętać całej listy obwarzanków do zdjęcia. Zamiast tego możemy połączyć je wszystkie w jeden. Polega to na tym, że jako zewnętrzną średnicę takiego „zbiorczego" obwarzanka bierzemy największą z zewnętrznych średnic, a jako wewnętrzną — najmniejszą z wewnętrznych średnic obwarzanków składowych. Łatwo zauważyć, że zbiór obwarzanków nie może być przełożony przez pewnego obwarzanka wtedy i tylko wtedy, gdy obwarzanek powstały przez sklejenie tego zbioru nie może zostać przez niego przełożony.

Przykładowy pseudokod:

```
wczytaj(n, m, obwarzanek[]);
wynik := 0;
zbiorczy_obwarzanek := obwarzanek[m];
{ przeglądamy wszystkich potencjalnych kandydatów do zdjęcia }
for i := m + 1 to n do begin
    { łączymy obwarzanki, jeśli potrzeba }
    if (not moge_przelozyc(zbiorczy_obwarzanek, obwarzanek[i])) then begin
        wynik := wynik + 1;
        zbiorczy_obwarzanek := polacz_obwarzanki(zbiorczy_obwarzanek, obwarzanek[i]);
    end
end
```

Funkcja, która łączy dwa obwarzanki w jeden:

```
obwarzanek polacz_obwarzanki(obwarzanek a, obwarzanek b) begin
    obwarzanek zbiorczy;
    zbiorczy.zewnetrzna = max(a.zewnetrzna, b.zewnetrzna);
    zbiorczy.wewnetrzna = min(a.wewnetrzna, b.wewnetrzna);
    return zbiorczy;
end
```

Złożoność czasowa takiego rozwiązania jest liniowa, ponieważ każdy obwarzanek przejrzymy tylko jeden raz.

Zadanie: PIO
Pionek

OLIMPIADA INFORMATYCZNA GIMNAZJALISTÓW

V OIG, etap II, dzień próbny. Plik źródłowy `pio.*` Dostępna pamięć: 64 MB. *02.04.2011*

Rozważmy nieskończoną (we wszystkich kierunkach) planszę o kwadratowych polach. Na tej planszy, na polu o współrzędnych $(1, 1)$ stoi pionek. Pionek ten może wykonywać dwa typy ruchów: polegające na przesunięciu się o pewną ustaloną liczbę pól w prawo bądź w lewo oraz polegające na przesunięciu się o pewną ustaloną liczbę pól w górę bądź w dół.

Przy takich ograniczeniach można zwykle dojść pionkiem tylko do niektórych pól planszy. Twoim zadaniem jest określić, ile spośród takich osiągalnych pól mieści się w zadanym prostokątnym fragmencie planszy.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n oraz m ($1 \leqslant n, m \leqslant 200\,000$), oddzielone pojedynczym odstępem i oznaczające liczby ruchów poziomych i pionowych, jakie może wykonać pionek.

W drugim wierszu znajdują się cztery liczby całkowite x_1, y_1, x_2, y_2 ($-10^9 \leqslant x_1, y_1, x_2, y_2 \leqslant 10^9$), poodzielane pojedynczymi odstępami i oznaczające odpowiednio współrzędne lewego dolnego i prawego górnego wierzchołka prostokątnego fragmentu planszy, w którym należy wyznaczyć liczbę pól osiągalnych. Przyjmujemy, że pole (x, y) należy do tego prostokąta wtedy i tylko wtedy, gdy $x_1 \leqslant x \leqslant x_2$ oraz $y_1 \leqslant y \leqslant y_2$.

W każdym z następnych n wierszy znajduje się jedna liczba całkowita a_i ($1 \leqslant a_i \leqslant 10^{18}$) oznaczająca liczbę pól, o jakie pionek może przesunąć się w prawo lub w lewo, wykonując i-ty poziomy ruch.

W każdym z następnych m wierszy znajduje się jedna liczba całkowita b_j ($1 \leqslant b_j \leqslant 10^{18}$) oznaczająca liczbę pól, o jakie pionek może przesunąć się w górę lub w dół, wykonując j-ty pionowy ruch.

Możesz założyć, że w testach wartych co najmniej 40% punktów zachodzi dodatkowy warunek: $n, m \leqslant 1\,000$ oraz $x_2 - x_1, y_2 - y_1 \leqslant 1\,000\,000$.

Wyjście

Twój program powinien wypisać w pierwszym i jedynym wierszu standardowego wyjścia jedną liczbę całkowitą oznaczającą liczbę pól w obrębie zadanego prostokątnego fragmentu planszy, do których pionek może się dostać.

Przykład

Dla danych wejściowych:

```
1 2
1 1 10 2
1
2
5
```

poprawnym wynikiem jest:

```
20
```

Opracowanie: PIO
Pionek

1 Przypadek jednowymiarowy

Zastanówmy się najpierw, jak wyglądałoby zadanie w prostszym, jednowymiarowym przypadku. Wtedy pionek poruszałby się po jednowymiarowej, nieskończonej osi, startował z punktu 1 i w jednym ruchu mógł wykonywać skoki długości $a_1, a_2, \ldots, a_n$. Należałoby wtedy obliczyć liczbę osiągalnych pól w pewnym przedziale $[x_1, x_2]$.

Aby uprościć nasze rozważania załóżmy, że punktem początkowym jest 0, zaś przedział, w którym szukamy osiągalnych pól to $[x_1 - 1, x_2 - 1]$. Chwila zastanowienia pokazuje, że wynik dla takiego sformułowania zadania jest identyczny, jak dla oryginalnego.

Rozważmy zbiór wszystkich pól osiągalnych przez pionka z pola 0. Kluczem do rozwiązania zadania jest zrozumienie, jaką ma on postać. Okazuje się bowiem, że pola osiągalne tworzą bardzo regularny wzór na osi.

Jeżeli $n = 1$, to zbiór X jest po prostu zbiorem wszystkich liczb całkowitych (także ujemnych), podzielnych przez liczbę a_i. Aby je zaznaczyć na rysunku, należałoby zamalować wszystkie pozycje co a_1 pól, poczynając od zera (idąc zarówno w prawo jak i w lewo).

Trudniej jest, kiedy mamy do dyspozycji więcej różnych ruchów. Można je bowiem wykonywać naprzemiennie i kiedy patrzymy na pewne pole, wcale nie wydaje się jasne, czy można do niego dojść, czy nie. Żeby nieco poskromić tę trudność uprośćmy zadanie jeszcze bardziej — rozważmy przypadek $n = 2$. Warto najpierw poeksperymentować — wziąć pewne dwie liczby a_1, a_2 i zaznaczać na osi pola osiągalne. Na przykład dla liczb 4 i 6 okaże się, że osiągalne są wszystkie liczby parzyste. Natomiast dla liczb 6 i 9 osiągalne są wszystkie liczby podzielne przez 3.

Spróbujmy uogólnić nasze spostrzeżenia. Dosyć jasne wydaje się, że jeżeli pewna liczba d dzieli zarówno a_1 jak i a_2, to będzie też dzielić numer każdego osiągalnego pola, bo skoro każdy skok jest na odległość będącą wielokrotnością d, to dowolna sekwencja kroków przesunie pionek na odległość również będącą wielokrotnością d.

Zastanówmy się teraz, kiedy osiągalne będą **wszystkie** pola w odległości podzielnej przez d. Okazuje się, że wtedy, kiedy d jest największym wspólnym dzielnikiem a_1 i a_2. Żeby to uzasadnić posłużymy się algorytmem Euklidesa, służącym do obliczania `NWD` dwóch liczb.

1.1 Algorytm Euklidesa

Algorytm Euklidesa wygląda następująco:

```
function NWD(integer a, integer b) {
  while (b > 0) {
    a := a mod b;
    zamien(a, b);
  }
  NWD := a;
}
```

Powyższy pseudokod zakłada, że argumenty `a` i `b` są dodatnie, zaś funkcja `zamien(x, y)` zamienia wartości zmiennych `x` oraz `y`. To jest klasyczna wersja algorytmu Euklidesa, która działa bardzo szybko (w czasie $O(\log(\max(a, b)))$) i tą wersją należy posługiwać się w implementacji. Łatwiej jednak będzie nam spojrzeć na równoważną jej, chociaż dużo wolniejszą wersję z odejmowaniem:

```
function NWD(integer a, integer b) {
  while (b > 0) {
    while (a > b)
      a := a - b;
    zamien(a, b);
  }
  NWD := a;
}
```

Obie wersje algorytmu cały czas przechowują dwie liczby `a` i `b`, które ciągle maleją, ale ich `NWD` pozostaje taki sam, jak `NWD` oryginalnych argumentów.

Przetłumaczmy to na język naszego zadania. Liczby `a` i `b` będą reprezentować możliwe do wykonania skoki. Chcielibyśmy pokazać, że podczas działania algorytmu Euklidesa na tych długościach skoków, zbiór pól osiągalnych za ich pomocą pozostaje niezmienny. Wtedy wynikiem algorytmu będzie jeden rodzaj skoków, który sam wystarcza do osiągnięcia dokładnie tego samego zbioru pól, co skoki `a` i `b`.

Dlaczego w czasie algorytmu Euklidesa zbiór pól osiągalnych przez skoki `a` i `b` się nie zmienia? Zauważmy, że druga wersja algorytmu wykonuje tylko jeden rodzaj zmiany liczb `a` i `b` — mianowicie odjęcie jednej liczby od drugiej. Wystarczy więc pokazać, że zbiór pól osiągalnych przez skoki a i b jest taki sam jak zbiór punktów osiągalnych przez skoki a i $a-b$. W tym celu wystarczy uzasadnić, że skoki a i $a-b$ są osiągalne przez skoki a i b i odwrotnie. Skok $a-b$ jest osiągalny w oczywisty sposób przez skoki a i b — najpierw skaczemy w prawo o a pól a potem w lewo o b pól. Aby zaś osiągnąć skok b przez skoki a i $a-b$, należy wykonać skok $a-b$ w lewo a potem a w prawo.

Udowodniliśmy więc, że zbiór pól osiągalnych przez skoki a_1 i a_2 jest taki sam, jak zbiór pól osiągalnych przez skok $NWD(a_1, a_2)$. Niektórzy już zapewne przeczuwają, ku czemu zmierza to spostrzeżenie ...

1.2 Rozwiązanie na osi

Właśnie znaleźliśmy sposób na zmniejszenie liczby rodzajów skoków o jeden, zarazem jednak nie zmieniając rozwiązania zadania! Powtarzając tę sztuczkę, możemy następnie kolejne pary skoków zamieniać w jeden tak długo, aż ze zbioru $\{a_1, a_2, \ldots, a_n\}$ pozostanie nam już tylko jeden skok. Będzie on największym wspólnym dzielnikiem wszystkich liczb $a_1, a_2, \ldots, a_n$.

Tak więc problem sprowadziliśmy do przypadku $n = 1$, kiedy jedyną długością skoku jest a. Pozostaje kwestia obliczenia liczby liczb podzielnych przez a w przedziale $[x_1 - 1, x_2 - 1]$. Można to zrobić na różne sposoby:

- Możemy przejrzeć wszystkie liczby całkowite w tym przedziale i dla każdej sprawdzić, czy jest ona podzielna przez a (czas $O(x_2 - x_1)$).
- Możemy skonstruować wzór, który pozwoli obliczyć ją w czasie stałym. Niech $p = x_1 - 1, k = x_2 - 1$ będą końcami przedziału. Liczba wszystkich pól wewnątrz przedziału to $w = k - p + 1$. Potrzebna nam będzie jeszcze liczba tych pól wewnątrz przedziału, które są za ostatnim osiągalnym polem w jego wnętrzu, czyli $s = k \bmod a$. Trzeba tutaj pamiętać, że k może być liczbą ujemną i dlatego resztę z dzielenia jej przez a należy w języku programowania zapisać w następujący sposób: `((k mod a) + a) mod a`, aby otrzymać właściwy wynik.
  W końcu od w możemy odjąć s i otrzymany wynik podzielić przez a zaokrąglając w dół. Otrzymana liczba jest liczbą osiągalnych pól wewnątrz przedziału.

1.3 Złożoność

W żadnej z powyższych wersji rozwiązania przypadku jednowymiarowego nie wykorzystujemy dodatkowej pamięci, więc ich złożoność pamięciowa jest stała (mówimy, że algorytmy te działają *w miejscu*). Różna natomiast może być ich złożoność czasowa.

Zakładamy, że korzystamy z pierwszej wersji algorytmu Euklidesa, więc jej złożoność czasowa będzie wynosiła $O\left(n \cdot \log(\max(a_i))\right)$. Osobną rzeczą jest sposób znalezienia liczby liczb podzielnych przez a w zadanym przedziale — jego złożoność czasowa może być stała bądź liniowa ze względu na długość przedziału. W przypadku limitów danych z zadania obliczanie jej za pomocą wzoru jest bardzo ważne.

2 Rozwiązania przypadku dwuwymiarowego

Poprzedni rozdział pokazał, jak obliczyć wynik w przypadku jednowymiarowym. Okazuje się, że nasze rozważania bardzo łatwo przenoszą się na oryginalną, dwuwymiarową wersję zadania. Możemy bowiem osobno dla osi pionowej i poziomej znaleźć największe wspólne dzielniki dostępnych długości skoków:

$$\begin{aligned} a &= NWD(a_1, \ldots, a_n) \\ b &= NWD(b_1, \ldots, b_n) \end{aligned}$$

Następnie możemy obliczyć ostateczny wynik na kilka sposobów:

- Przejść po wszystkich punktach kratowych (o współrzędnych całkowitych) wewnątrz zadanego prostokąta i dla każdego sprawdzić, czy jego współrzędne są podzielne przez odpowiednio a i b;
- Obliczyć dla każdej osi liczbę punktów podzielnych odpowiednio przez a i b w odpowiednich przedziałach (czyli $[x_1 - 1, x_2 - 1]$ dla osi poziomej oraz $[y_1 - 1, y_2 - 1]$ dla osi pionowej) a następnie pomnożyć wyniki.

W drugim przypadku możemy otrzymać złożoność $O(x_2 - x_1 + y_2 - y_1)$ lub $O(1)$. Oczywiście rozwiązanie wzorcowe korzysta ze wzoru dającego czas stały. Dzięki temu złożoność rozwiązania wzorcowego jest zdominowana przez algorytm Euklidesa, a więc wynosi $O\left(n \cdot \log(\max_i(a_i, b_i))\right)$.

Uwaga! Ponieważ długości przedziału są rzędu 10^9, to ostateczny wynik może być rzędu 10^{18}. Ponadto liczby a_i oraz b_i również mogą być rzędu 10^{18}. Konieczne jest więc zastosowanie liczb 64-bitowych i rozwiązania nie korzystające z nich nie otrzymywały maksymalnej liczby punktów.

Zadanie: TOR
Tort

OLIMPIADA INFORMATYCZNA GIMNAZJALISTÓW

V OIG, etap II. Plik źródłowy `tor.*` Dostępna pamięć: 64 MB. *02.04.2011*

Bajtek obchodzi dzisiaj swoje urodziny. Zdmuchnął już wszystkie świeczki z tortu urodzinowego oraz podzielił go na $(n+1)^2$ kawałków. Niestety zrobił to w taki sposób, że niektóre kawałki są większe, a inne mniejsze od pozostałych. Bajtek wybiera swój kawałek jako pierwszy i chciałby wybrać k-ty pod względem wielkości, czyli taki, że $k-1$ kawałków jest nie mniejszych od niego, a $(n+1)^2-k$ kawałków jest nie większych.

Wiemy, że tort urodzinowy Bajtka ma kształt prostokąta oraz że Bajtek podzielił go n prostymi cięciami wzdłuż jednego z boków prostokąta i n prostymi cięciami wzdłuż drugiego z boków. Chcielibyśmy znać powierzchnię wybranego przez Bajtka kawałka tortu.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się cztery liczby całkowite a, b, n oraz k pooddzielane pojedynczymi odstępami ($1 \leqslant a, b \leqslant 10^9$, $0 \leqslant n \leqslant 200\,000$, $1 \leqslant k \leqslant (n+1)^2$), oznaczające odpowiednio długości boków tortu, liczbę cięć wykonanych przez Bajtka w każdym z kierunków oraz numer szukanego kawałka.

Drugi wiersz zawiera ciąg n liczb całkowitych $x_1, x_2, \ldots, x_n$ ($0 < x_1 < x_2 < \ldots < x_n < a$), przy czym x_i oznacza miejsce i-tego cięcia wzdłuż poziomego boku prostokąta (jest to odległość od lewego boku prostokąta).

Trzeci wiersz zawiera ciąg n liczb całkowitych $y_1, y_2, \ldots, y_n$ ($0 < y_1 < y_2 < \ldots < y_n < b$), przy czym y_i oznacza miejsce i-tego cięcia wzdłuż pionowego boku prostokąta (jest to odległość od dolnego boku prostokąta).

Możesz założyć, że w testach o wartości co najmniej 40% punktów zachodzi dodatkowy warunek: $n \leqslant 1\,000$.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, równą powierzchni k-tego pod względem wielkości kawałka tortu.

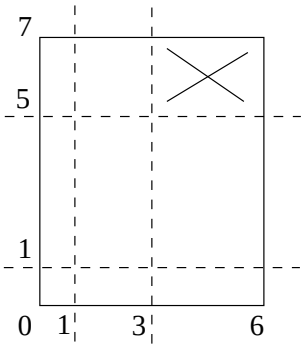
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
6
```



Opracowanie: TOR
Tort

1 Rozwiązanie brutalne

Pierwszym, narzucającym się rozwiązaniem zadania, jest obliczenie powierzchni każdego kawałka tortu, a następnie wybranie k-tego w kolejności. Znalezienie k-tego w kolejności elementu możemy zrealizować poprzez posortowanie wszystkich elementów. Pseudokod takiego rozwiązania:

```
wczytaj(a, b, n, k, x[], y[]);
x[0] := y[0] := 0;
x[n + 1] := a;
y[n + 1] := b;
liczba := 0;
for k := 1 to n + 1 do
    for i := 1 to n + 1 do begin
        liczba := liczba + 1;
        kawalki[liczba] := (x[k] - x[k-1]) * (y[i] - y[i-1]);
    end
posortuj(kawalki[]);
wypisz(kawalki[k]);
```

Głównym problemem jaki powstaje przy takim podejściu jest złożoność czasowa. Wszystkich kawałków tortu będzie $O(n^2)$, więc czas potrzebny na ich posortowanie to $O(n^2 \log(n^2))$.

2 Rozwiązanie wolne

Szukanie k-tego w kolejności elementu możemy zrealizować trochę szybciej. Istnieje algorytm *Hoare'a*, który znajduje go w czasie liniowym względem liczby wszystkich elementów. Złożoność spada wtedy do $O(n^2)$.

W algorytmie *Hoare'a* stosujemy podobną metodę jak w sortowaniu *Quicksort*. Ciąg wejściowy dzielimy na 2 podciągi: elementów mniejszych oraz elementów większych bądź równych pewnemu elementowi. Następnie w zależności od liczebności wynikowych podciągów wywołujemy rekurencyjnie funkcję dla odpowiedniego podciągu. Rekursję powtarzamy tak długo, aż w wyniku otrzymamy ciąg o 1 elemencie.

O algorytmie Hoare'a można przeczytać w książce „Algorytmy i struktury danych" L. Banachowskiego, K. Diksa oraz W. Ryttera w rozdziale 2.8 („Szybkie algorytmy wyznaczania k-tego największego elementu w ciągu").

3 Rozwiązanie wzorcowe

W rozwiązaniu wzorcowym zastosujemy trochę inne podejście, niż w poprzednich rozwiązaniach. Interesujący nas rozmiar kawałka tortu będziemy wyszukiwać binarnie po powierzchni. Wiadomo, że szukana powierzchnia kawałka będzie z przedziału od 1 do $a \cdot b$, gdzie $a \cdot b$ to wielkość całego tortu.

W każdym kroku wyszukiwania dzielimy przedział (po którym szukamy) na dwa. Wybieramy element środkowy s naszego przedziału i w zależności od tego, ile kawałków tortu jest nie mniejszych od s, wybieramy przedział lewy, bądź prawy do kolejnych iteracji. Ponieważ wielkość przedziału zmniejsza się za każdym razem dwukrotnie, to liczba wszystkich podziałów będzie rzędu $O(\log(ab))$. Jedyne, co nam pozostaje, to znalezienie szybkiego sposobu obliczenia, ile kawałków tortu jest nie mniejszych od s. Przykładowy pseudokod:

```
poczatek := 1;
koniec := a * b;
{ wyszukujemy binarnie powierzchnię kawałka }
while (poczatek < koniec) begin
    s := (poczatek + koniec + 1) / 2;
    if (liczba_nie_mniejszych(s) >= k) then
        poczatek := s;
    else
        koniec := s - 1;
end
wypisz(poczatek);
```

3.1 Zliczanie kawałków

Zauważmy, że jeśli posortujemy długości kawałków niemalejąco, to dla każdej szerokości możemy za pomocą wyszukiwania binarnego znaleźć liczbę kawałków nie mniejszych od s. Tym sposobem obliczenie liczby kawałków nie mniejszych od s kosztuje nas czas rzędu $O(n \cdot \log(n))$, a całe rozwiązanie ma złożoność $O(n \cdot \log(n) \cdot \log(ab))$.

Można to jednak zrobić jeszcze szybciej. Posortujmy zarówno długości, jak i szerokości niemalejąco. Teraz zaczniemy szukać liczby kawałków nie mniejszych od s, poczynając od najmniejszych szerokości.

Załóżmy, że znamy liczbę kawałków nie mniejszych od s dla pewnej szerokości. Jak może się ona zmienić dla kawałków o większej szerokości? Może się tylko zwiększyć, ponieważ wszystkie długości kawałków są takie same, a zwiększyliśmy szerokość. W ten sposób każdą długość i szerokość sprawdzimy dokładnie jeden raz, więc czas wyszukiwania jest rzędu $O(n \cdot \log(n))$, ze względu na sortowanie. Złożoność całego rozwiązania to $O(n \cdot \log(ab))$, ponieważ $a \cdot b > n$. Pseudokod:

```
liczba_nie_mniejszych(s) begin
    wynik := 0;
    k := n;
    { przeglądamy szerokości od najmniejszych do największych }
    for i := 0 to n do begin
        { znajdujemy liczbę kawałków, których powierzchnia jest nie mniejsza od s }
        while (k > 0 and szerokosc[i] * dlugosc[k-1] >= s)
            k := k - 1;
        wynik := wynik + n - k;
    end
    liczba_nie_mniejszych := wynik;
end
```

Należy pamiętać, aby obliczyć wcześniej wszystkie szerokości i długości kawałków, zapisać w tablicach `szerokosc[]`, `dlugosc[]` oraz posortować je niemalejąco. Pozostawiamy to jako proste ćwiczenie dla czytelnika.

OLIMPIADA
INFORMATYCZNA
GIMNAZJALISTÓW

Etap III

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

MINISTERSTWO
EDUKACJI
NARODOWEJ

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany ze środków Unii Europejskiej
w ramach Europejskiego Funduszu Społecznego.

Zadanie: BRA
Bracia

OLIMPIADA INFORMATYCZNA GIMNAZJALISTÓW

OIG, etap III. Plik źródłowy `bra.*` Dostępna pamięć: 32 MB. *20.05.2011*

W szeregu ustawiło się n chłopców. Wielu z nich jest braćmi z tych samych rodzin. Z szeregu możemy wyprosić pewne osoby, dążąc do tego, aby bracia z każdego rodzeństwa stali obok siebie. Jednak osoby stojące w szeregu są bardzo solidarne ze swoimi braćmi — jeżeli usunięta zostanie dowolna osoba, to wszyscy jej bracia obrażają się i również odchodzą z szeregu.

Oblicz, jaka jest największa liczba rodzeństw, które mogą pozostać w szeregu w wyniku takich zmian, tak aby bracia z każdego pozostałego w szeregu rodzeństwa stali obok siebie. Uwaga: jedynak liczy się jak całe rodzeństwo!

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 1\,000\,000$) oznaczającą liczbę osób ustawionych w szeregu. Drugi wiersz wejścia zawiera n liczb całkowitych $l_1, l_2, \ldots, l_n$ ($1 \leqslant l_n \leqslant 1\,000\,000$) pooddzielanych pojedynczymi odstępami, przy czym l_i oznacza numer rodzeństwa, do którego należy i-ty chłopiec.

Możesz założyć, że w testach wartych przynajmniej 50% punktów zachodzi dodatkowy warunek $n \leqslant 10\,000$.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą równą maksymalnej liczbie rodzeństw, jakie mogą pozostać w szeregu.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2
```

Opracowanie: BRA
Bracia

1 Wstęp

Kluczowe w rozwiązaniu zadania jest spostrzeżenie, że zamiast oryginalnego szeregu możemy, nie zmieniając wyniku, rozważać szereg, w którym znajduje się jedynie pierwszy i ostatni z braci z każdego rodzeństwa. Pozycje skrajnych braci możemy wyznaczyć liniowo podczas wczytywania danych w następujący sposób:

```
procedure wczytaj_szereg;
begin
  wczytaj(n);
  for k := 1 to max_numer_rodzenstwa do poczatek[k] := 0;
  for i := 1 to n do
  begin
    wczytaj(szereg[i]);
    if poczatek[szereg[i]] = 0 then poczatek[szereg[i]] := i;
    koniec[szereg[i]] := i;
  end;
end;
```

2 Rozwiązanie dynamiczne

W jaki sposób wyznaczyć maksymalną liczbę rodzeństw, które mogą pozostać w szeregu? Spróbujemy posłużyć się programowaniem dynamicznym.

Aby zredukować rozmiar problemu do mniejszego ciągu, dla którego odpowiedź moglibyśmy mieć obliczoną wcześniej, przypatrzmy się ostatniemu rodzeństwu w szeregu. Jeżeli zaczyna się ono na pozycji i (a kończy się na pozycji n), to możemy podjąć jedną z dwóch decyzji:

- pozostawimy to rodzeństwo w szeregu — wtedy pozbawiamy się możliwości wybrania rodzeństw, których reprezentanci stoją na pozycjach $[i+1, n-1]$, a więc pozostałych braci będziemy mogli wybrać jedynie z rodzeństw, których ostatni reprezentanci znajdują się na pozycjach $[1, i-1]$;
- wyprosimy ostatnie rodzeństwo — w tej sytuacji w szeregu będą mogły pozostać rodzeństwa na pozycjach $[1, n-1]$.

Powyższy schemat można wprost zapisać w postaci następującego programu dynamicznego:

```
wczytaj_szereg;
wynik[0] := 0; { dla pustego szeregu nie możemy wybrać nikogo }
for i := 1 to n do
begin
  wynik[i] := wynik[i-1];
  if koniec[szereg[i]] = i then
    wynik[i] := max(wynik[i], wynik[poczatek[szereg[i]]-1]+1);
end;
wypisz(wynik[n]);
```

Złożoność takiego rozwiązania wynosi $\Theta(n)$.

3 Rozwiązanie zachłanne

Okazuje się, że problem ten możemy rozwiązać w bardziej bezpośredni, mianowicie zachłanny, sposób. Otóż rozważajmy rodzeństwa w kolejności rosnących pozycji ostatnich reprezentantów. Jeżeli ostatni z braci poprzednio wybranego rodzeństwa nie znajduje się pomiędzy braćmi z rozważanego rodzeństwa, to rodzeństwo to pozostaje w szeregu, w przeciwnym przypadku zostaje z niego usunięte.

Dlaczego takie podejście jest poprawne? Niech p_r, k_r będą odpowiednio pozycją pierwszego i ostatniego brata z r-tego rodzeństwa. Niech $i_1, i_2, \ldots, i_w$ będzie maksymalnym uporządkowanym wg rosnących pozycji ciągiem rodzeństw, które mogą pozostać w szeregu. Jeżeli l_1 będzie rodzeństwem, którego ostatnia pozycja jest najmniejsza, to $l_1, i_2, i_3, \ldots, i_w$ też jest dopuszczalnym zbiorem rodzeństw pozostawionych w szeregu, gdyż $k_{l_1} \leqslant k_{i_1}$. Analogicznie postępujemy dla kolejnych rodzeństw otrzymując ciąg $l_1, l_2, \ldots, l_w$. Zauważmy, że l_j dla $j = 2, \ldots, w$ istnieje, gdyż dobrym kandydatem jest i_j.

Rozwiązanie wymaga rozważania rodzeństw w kolejności rosnących pozycji ostatnich reprezentantów. Nie musimy jednak sortować rodzeństw — ich kolejność możemy odczytać z tablicy `koniec` otrzymując liniowy algorytm.

```
wczytaj_szereg;
wynik := 0;
granica := 0;
for i := 1 to n do
begin
  if (koniec[szereg[i]] = i) and (granica < poczatek[szereg[i]]) then
  begin
    wynik := wynik+1;
    granica := i;
  end;
end;
wypisz(wynik);
```

Złożoność rozwiązania to $\Theta(n)$.

Zadanie: NAW
Nawiasy

OLIMPIADA INFORMATYCZNA GIMNAZJALISTÓW

V OIG, etap III, dzień próbny. Plik źródłowy `naw.*` Dostępna pamięć: 32 MB. *19.05.2011*

Wyrażeniem nawiasowym nazwiemy niepusty ciąg składający się z nawiasów otwierających i zamykających. Powiemy, że wyrażenie nawiasowe jest *poprawne*, jeżeli każdy nawias otwierający można sparować z nawiasem zamykającym, występującym po nim, tak aby ciąg nawiasów znajdujących się pomiędzy nimi również był poprawnym ciągiem nawiasowym.

Na przykład `(()())()` jest poprawnym wyrażeniem nawiasowym, ale `)(` i `()(` już poprawne nie są.

Bajtazar w swojej pracy naukowej skorzystał z programu wypisującego pewne poprawne wyrażenie nawiasowe S, mające kluczowe znaczenie dla jego badań. Niestety, słowo to zaginęło w gąszczu innych nawiasów, które przez przypadek mogły być wypisane zarówno przed nim, jak i po nim. Bajtazar otrzymał więc słowo nawiasowe, które zawiera w sobie jako spójny fragment szukane słowo S, jednak nie wie, gdzie się ono zaczyna i gdzie kończy.

Zrozpaczony, poprosił Ciebie o pomoc w wyznaczeniu wszystkich możliwych położeń poprawnych słów nawiasowych w otrzymanym słowie. Ma bowiem nadzieję, że jest ich niewiele ...

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 2\,000\,000$), oznaczającą długość słowa, które odczytał Bajtazar. W drugim wierszu znajduje się n nawiasów (bez odstępów) — jest to odczytane słowo nawiasowe.

Możesz założyć, że w testach wartych przynajmniej 50% punktów zachodzi dodatkowy warunek $n \leqslant 5\,000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście jedną liczbę całkowitą oznaczającą liczbę fragmentów odczytanego słowa, które są poprawnymi słowami nawiasowymi.

Przykład

Dla danych wejściowych:

```
10
)(())()(()
```

poprawnym wynikiem jest:

```
5
```

Opracowanie: NAW
Nawiasy

1 Wstęp

Wyrażenia nawiasowe pojawiają się bardzo często w różnych kombinatorycznych zadaniach, toteż warto wiedzieć, jak się zabierać do rozwiązywania problemów z nimi związanych.

Otóż wygodniej jest myśleć o ciągach złożonych z 1 i -1, zamiast z nawiasów. Niech 1 oznacza nawias otwierający, zaś -1 zamykający. Wtedy $c_1 + c_2 + \ldots + c_k$ oznacza *głębokość zagnieżdżenia* za k-tym nawiasem.

Łatwo zauważyć, wyrażenie $c_1 c_2 \ldots c_n$ jest poprawne wtedy i tylko wtedy, gdy głębokość w żadnym miejscu nie spada poniżej 0, zaś po n-tym nawiasie równa się 0.

Można to zapisać wzorami:

$$\forall_{1 \leqslant k \leqslant n}\ c_1 + c_2 + \ldots + c_k \geqslant 0 \tag{1}$$

$$c_1 + c_2 + \ldots + c_n = 0 \tag{2}$$

gdzie $\forall_{1 \leqslant k \leqslant n}$ oznacza „dla wszystkich k takich, że $1 \leqslant k \leqslant n$".

2 Rozwiązanie sześcienne

Rozwiązanie brutalne polega na sprawdzeniu powyższych nierówności dla każdego z $O(n^2)$ podsłów. Złożoność czasowa rozwiązania to $O(n^3)$ – dla każdego podsłowa musimy wykonać bowiem $O(n)$ sprawdzeń.

3 Rozwiązanie kwadratowe

Zauważmy, że jeśli $c_a + c_{a+1} + \ldots + c_b < 0$, to wszystkie poprawne wyrażenia zaczynające się w a muszą kończyć się przed b. Prowadzi to do następującego rozwiązania działającego w czasie $O(n^2)$.

Dla każdego a rozpoczynamy sumowanie ciągu (c_i) od c_a. Jeśli w którymś miejscu osiągniemy 0, to znaczy, że właśnie znaleźliśmy kolejne poprawne wyrażenie i zwiększamy wynik o 1. Jeśli suma równa się -1, to przerywamy zliczanie, gdyż wiemy, że wszystkie dalsze wyrażenia na prawo będą niepoprawne. Oto pseudokod powyższego rozwiązania, obliczający wynik dla tablicy nawiasów $t[1 \ldots n]$:

```
wynik := 0;

for a := 1 to n do
begin
    suma := 0;
    i := a;

    while i <= n and suma >= 0 do
    begin
        if t[i] = '(' then
            suma += 1;
        else // t[i] = ')'
            suma -= 1;

        if suma = 0 then
            wynik += 1;

        i += 1;
    end;
end;

wypisz(wynik);
```

3.1 Rozwiązanie wzorcowe

Powyższe rozwiązanie działa nadal zbyt wolno dla podanych limitów na rozmiar danych, więc postarajmy się wykonać zliczanie jeszcze sprytniej.

Poprawne wyrażenie nazwiemy *minimalnym*, jeśli $\forall_{1 \leqslant k < n}\ c_1 + c_2 + \ldots + c_k > 0$. Innymi słowy, *głębokość* w środku wyrażenia minimalnego musi być cały czas dodatnia.

Zauważmy teraz, że każde poprawne wyrażenie jest konkatenacją („sklejeniem") kilku wyrażeń minimalnych. Podsuwa to pomysł, aby dla każdego wyrażenia minimalnego $c_a c_{a+1} \ldots c_b$ sprawdzić, ile poprawnych wyrażeń kończy się w $a - 1$. Oznaczmy tę liczbę przez s. Wtedy będziemy mogli dokleić każde z nich z lewej do $c_a c_{a+1} \ldots c_b$, dodać $s + 1$ do wyniku i zapamiętać, że w b kończy się $s + 1$ poprawnych wyrażeń (+1 odpowiada samemu wyrażeniu $c_a c_{a+1} \ldots c_b$ — bez doklejania czegokolwiek z lewej strony).

Przykład: przeanalizujmy wyrażenie `(()(()))(` znak po znaku. Niech g_i oznacza *głębokość* po i-tym nawiasie, a w_i – liczba poprawnych wyrażeń kończących się w i. Zaznaczone są końce wyrażeń minimalnych.

1 : $g_1 = 1, w_1 = 0$

2 : $g_2 = 2, w_2 = 0$

3 : $g_3 = 1, w_3 = 1$ kończy się `()`

4 : $g_4 = 2, w_4 = 0$

5 : $g_5 = 3, w_5 = 0$

6 : $g_6 = 2, w_6 = 1$ kończy się `()`

7 : $g_7 = 1, w_7 = 2$ kończy się `(())`

8 : $g_8 = 0, w_8 = 1$ kończy się `(()(()))`

9 : $g_9 = 1, w_9 = 0$

Wynikiem dla tego przykładu jest 1 + 1 + 2 + 1 = 5. Zachęcamy czytelnika do samodzielnego uogólnienia powyższej metody.

Niecierpliwym od razu przedstawiamy pseudokod korzystający ze stosu (czyli struktury danych pozwalającej na dodawanie elementów na szczyt i zdejmowanie ze szczytu). Na zmiennej p będziemy pamiętać kolejne wartości w_i. Natomiast stos będziemy wypełniać w taki sposób, aby w każdym momencie, w którym przeczytaliśmy pewne wyrażenie minimalne $c_{i+1} c_{i+2} \ldots c_j$ na szczycie stosu znajdowała się liczba w_i. Oto pseudokod rozwiązania wzorcowego:

```
p := 0, wynik := 0;
stos := pusty stos;

for i = 1 to n do
begin
    if t[i] = '(' then begin
        stos.dodaj(p + 1);
        p := 0;
    end else begin
        // t[i] = ')'
        if not stos.pusty() then begin
            p := stos.szczyt();
            stos.zdejmij();
            wynik += p;
        end else begin
            p := 0;
        end
    end
end.

wypisz(wynik);
```

Operacje na stosie można wykonać w czasie stałym (np. przez użycie tablicy jako stosu), zatem cały algorytm wykonuje $O(n)$ operacji, co jest wynikiem optymalnym, ze względu na konieczność wczytania całego słowa nawiasowego.

Na koniec warto jeszcze zauważyć, że wynik dla $n = 2\,000\,000$ może wykraczać poza typ liczb 32-bitowych. Na przykład dla słowa `()()()()...` wynikiem jest $\frac{n^2}{8} - \frac{n}{4}$. Na szczęście zadeklarowanie zmiennej `wynik` jako 64-bitowej rozwiązuje ten problem.

Zadanie: TAM
Tamy

OLIMPIADA INFORMATYCZNA GIMNAZJALISTÓW

OIG, etap III. Plik źródłowy `tam.*` Dostępna pamięć: 64 MB. *20.05.2011*

W Bajtocji zbudowano wielki zbiornik wodny. Podzielono go na pewną liczbę jednakowej długości sektorów. Pomiędzy każdymi dwoma sąsiednimi sektorami znalazła się tama o pewnej wysokości. Tamy zbudowano także przed pierwszym oraz za ostatnim sektorem.

Obecnie poziom wody w całym zbiorniku jest taki sam. Ponieważ jednak zaczął padać ulewny deszcz, poziom wody zaczął szybko wzrastać. Król Bajtocji chce wiedzieć, ile czasu upłynie, zanim woda przeleje się przez pierwszą lub przez ostatnią tamę, wskutek czego pewnikiem dojdzie do zalania Bajtocji. Obliczenie tego utrudnia fakt, że nad każdym z sektorów deszcz może padać z różną intensywnością.

Pomóż królowi obliczyć czas, jaki pozostał do wylania się wody poza zbiornik. Jeżeli poziom wody jest równy dokładnie wysokości tamy, woda jeszcze się nie wylewa. Gdy część zbiornika już zalana wodą jest ograniczona z obu stron tamami tej samej wysokości, woda przelewa się z niej w obie strony równie szybko.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 500\,000$), oznaczającą liczbę sektorów, na jakie podzielony jest zbiornik.

Drugi wiersz zawiera ciąg $n+1$ liczb całkowitych w_i ($1 \leqslant w_i \leqslant 1\,000\,000$) pooddzielanych pojedynczymi odstępami, oznaczających wysokości (ponad początkowy poziom wody) kolejnych tam.

Trzeci wiersz zawiera ciąg n liczb całkowitych k_i ($1 \leqslant k_i \leqslant 1\,000\,000$) pooddzielanych pojedynczymi odstępami, oznaczających, o ile poziomów woda podnosi się w i-tym sektorze w ciągu jednej sekundy.

Możesz założyć, że w testach wartych przynajmniej 75% punktów wysokości tam są parami różne. Ponadto, w części tych testów, wartych co najmniej 50% punktów, zachodzi warunek $n \leqslant 1\,000$.

Wyjście

Pierwszy wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, będącą najmniejszą liczbą całkowitą nie mniejszą niż liczba sekund, po których woda wyleje się ze zbiornika.

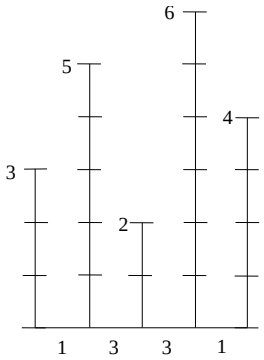
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2
```



Opracowanie: TAM
Tamy

1 Wprowadzenie

Możemy o naszym zadaniu myśleć następująco: mamy dane podłużne naczynie z poprzecznymi przegródkami (tamami), dzielącymi je na rząd sektorów. Wysokości tam mogą (w prostszej wersji - muszą) być różne. Do każdego sektora wpływa jednostajny strumień, który podnosi poziom wody w tym sektorze o k_i jednostek na sekundę. Naszym zadaniem jest obliczenie czasu, po jakim woda wyleje się z naczynia.

Zauważmy, że jeżeli pomiędzy dwoma tamami (nazwijmy je zewnętrznymi) znajdują się jedynie tamy od nich niższe, to możemy je połączyć w jedno naczynie. Aby woda wylała się przez jedną z zewnętrznych tam, musi najpierw osiągnąć poziom równy wysokości jednej z nich. Wobec tego nie ma znaczenia, jakie wysokości mają tamy wewnętrzne.

2 Rozwiązanie dla różnych wysokości tam

Uzyskujemy w ten sposób ciąg tam postaci $a_0, \ldots, a_k, \ldots, a_n$, gdzie $a_0, \ldots, a_k$ jest rosnący, a $a_k, \ldots, a_n$ jest malejący. Zauważmy, że możemy rozważać te ciągi osobno, gdyż woda wcześniej wyleje się z całego naczynia niż by przelała się przez najwyższy punkt, którym jest właśnie a_k. Jeżeli drugi z ciągów (czyli $a_k, \ldots, a_n$) odwrócimy, otrzymamy dwa naczynia o rosnących wysokościach tam, zatem mogące wylać tylko z lewej strony. Ostatecznym rozwiązaniem będzie mniejszy spośród wyników dwóch podproblemów. Dalej będziemy rozważać wyłącznie jedno z tych naczyń.

Zauważmy, że jeżeli woda z sektora i miałaby wylać się z naczynia, to sektory $1 \ldots i$ musiałyby być pełne. Musi przy tym zachodzić nierówność $czas \cdot \sum_{j=1}^{i} k_j > \sum_{j=1}^{i} p_j$, gdzie p_j to pojemność j-tego sektora. Zatem wystarczy dla kolejnych wartości i obliczyć sumę opadów oraz pojemność w sektorach $1 \ldots i$, aby z powyższego wzoru obliczyć dokładny czas, po jakim woda się wyleje. Najmniejsza z tych wartości będzie wynikiem.

Rozwiązanie ma złożoność czasową $\Theta(n)$.

3 Rozwiązanie wzorcowe

Rozwiązanie to jest symulacją przelewania wody, która ogranicza się do zdarzeń znaczących, czyli przepełnień sektorów. Kiedy żaden z sektorów się nie przepełnia, to jedyną rzeczą, jaka się dzieje, jest zwiększenie poziomu wody we wszystkich sektorach. Jednak skoro opady są jednostajne, łatwo przewidzieć moment przepełnienia każdego sektora. Zmiana tego momentu może nastąpić jedynie na skutek przepełnienia któregoś z sąsiednich sektorów. Jednak przepełnienie sąsiedniego sektora musiałoby nastąpić wcześniej, a w momencie jego przepełnienia obliczymy aktualny poziom wody w rozważanym sektorze, zaś tempo jego wzrostu powiększymy o wodę spływającą z przepełnionego sektora.

Rozwiązanie wzorcowe buduje kolejkę priorytetową zdarzeń (przepełnień sektorów), w której trzymamy numery sektorów, które kolejno ulegną przepełnieniu, zakładając, że sąsiadujące z nim sektory się nie przepełnią. Oczywiście czas przepełnienia pierwszego sektora w kolejce nie może ulec zmianie, gdyż żaden sektor nie może wcześniej ulec przepełnieniu.

Po zdjęciu pierwszego elementu z kolejki rozważamy następujące możliwości:

- jeżeli woda przelewa się z niego poza naczynie, to kończymy;
- w przeciwnym przypadku sprawdzamy, jak wylewająca się z niego woda wpłynie na czasy przepełnień sąsiednich sektorów:
  - jeżeli sektor sąsiadujący jest przepełniony, to aktualizujemy poziom wody w sektorach, do których spływa z niego woda (jego sąsiadach) i łączymy go z rozważanym sektorem,
  - następnie zwiększamy przyrost wody w sektorach sąsiednich i aktualizujemy czasy przepełnienia sąsiadów rozważanego sektora oraz odpowiadające im wartości w kolejce,
  - jeżeli sektor przepełnił się w jednym kierunku, to możemy go usunąć, gdyż woda, która do niego wpłynie z sąsiada, popłynie dalej.

Algorytm dla każdego przepełnienia wykonuje logarytmiczną ilość operacji (obsługa kolejki priorytetowej), a każdy sektor może ulec przepełnieniu tylko raz, więc całość działa w czasie $\Theta(n \cdot \log n)$.

Zadanie: TYG
Tygrysy

OLIMPIADA INFORMATYCZNA GIMNAZJALISTÓW

V OIG, etap III. Plik źródłowy `tyg.*` Dostępna pamięć: 32 MB. *20.05.2011*

Bajtockie tygrysy to niezwykłe zwierzęta, a ich nietypowe zwyczaje od zawsze fascynowały zoologów i matematyków. Ustalono niedawno, że dzielą się one na specyficzne gatunki. Tygrysa nazwiemy k-tygrysem, jeśli spotkawszy tygrysa mniejszego co najmniej k razy od siebie, zaatakuje go i zje, jednak nie odważy się tknąć żadnego większego tygrysa.

W bajtockim ZOO żyje n tygrysów. Niestety miejsce w ZOO jest ograniczone, dlatego też dyrektor stwierdził, że trzeba tak przydzielić zwierzęta do wybiegów, aby zająć ich jak najmniej. Oczywiście nie można przy tym dopuścić, by jakikolwiek tygrys został pożarty. Dyrektor ma wyraźne problemy z zakwaterowaniem tygrysów, zwrócił się więc do Ciebie po pomoc.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 500\,000$) — liczbę tygrysów w ZOO. Każdy z kolejnych n wierszy zawiera opis jednego tygrysa. Opis taki składa się z dwóch liczb całkowitych r_i oraz k_i ($1 \leqslant r_i \leqslant 1\,000\,000\,000$, $2 \leqslant k_i \leqslant 1\,000\,000$), oddzielonych pojedynczym odstępem. Oznaczają one, że i-ty tygrys jest k_i-tygrysem i ma rozmiar r_i.

Możesz założyć, że w przynajmniej 50% przypadków testowych występują jedynie 2-tygrysy.

Wyjście

Twój program powinien wypisać na standardowe wyjście dokładnie jedną liczbę całkowitą — najmniejszą liczbę wybiegów, do których można bezpiecznie przydzielić tygrysy.

Przykład

Dla danych wejściowych:	poprawnym wynikiem jest:

`5`
`8 3`
`10 2`
`15 2`
`18 2`
`28 3`
	`2`

Wyjaśnienie do przykładu: W powyższym przykładzie tygrysy o rozmiarach 28, 18, 15 mogą występować na wybiegu number 1, zaś tygrysy o rozmiarach 10, 8 mogą zostać pokazane na wybiegu numer 2.

Opracowanie: TYG
Tygrysy

1 Analiza problemu

Sprawdzenie wszystkich możliwych przyporządkowań tygrysów do wybiegów można odrzucić na samym początku — ich liczba dla $n = 500\,000$ byłaby ogromna. Lepiej poszukać strategii, która pozwoliłaby nam przydzielić zwierzęta do wybiegów w optymalny sposób. Na początku dobrze jest rozważyć najprostszy przypadek, w końcu ogólna strategia musi dawać dobry wynik również dla niego. Przyjmijmy zatem założenie, że wszystkie tygrysy są jednego gatunku, tzn. są 2-tygrysami.

1.1 Rozwiązanie prostego przypadku

Intuicyjnie wydaje się, że tygrysy o zbliżonych rozmiarach powinny być trzymane razem. Nasuwa się zatem myśl, by posortować je po rozmiarze. Sortowanie może być wykonane w czasie $O(n \log n)$, więc możemy sobie na to pozwolić. Zacznijmy od najmniejszego tygrysa (o rozmiarze r_1) i przydzielmy mu pierwszy wybieg. Nic nie stoi na przeszkodzie, żeby zamieszkały z nim wszystkie tygrysy o rozmiarze $r_i < 2r_1$, lecz każdy następny stanowiłby już zagrożenie dla najmniejszego.

Możemy powtarzać to rozumowanie, za każdym razem wybierając najmniejszego nieprzydzielonego tygrysa i dobierając mu bezpieczne towarzystwo. Zastanówmy się teraz, czy obrana strategia daje na pewno minimalną liczbę wybiegów.

Załóżmy, że tak nie jest i istnieje inne, lepsze rozwiązanie. Niech m oznacza liczbę wybiegów otrzymaną w powyższym algorytmie i niech $r_1, r_2 \dots r_m$ oznaczają rozmiary najmniejszych tygrysów w poszczególnych wybiegach. Zauważmy, że $r_{i+1} \geqslant 2r_i$, bo w przeciwnym razie oba tygrysy trafiłyby do tego samego wybiegu. W lepszym rozwiązaniu wybiegów jest mniej niż m, zatem pewne dwa spośród wyróżnionych m tygrysów musiałyby zamieszkać razem. Jest to oczywiście niemożliwe, więc lepsze rozwiązanie nie może istnieć.

2 Rozwiązanie wzorcowe

Postarajmy się teraz zmodyfikować powyższą ideę, aby działała dla dowolnych gatunków. Chcielibyśmy umieć szybko znajdować maksymalne „bezpieczne towarzystwo" dla najmniejszego bezdomnego tygrysa. Dla i-tego tygrysa oznaczmy $g_i = \lfloor \frac{r_i}{k_i} \rfloor$ ($\lfloor a \rfloor$ to największa liczba całkowita nie większa niż a). Jest to maksymalny rozmiar tygrysa, który nie może czuć się bezpieczny przy i-tym. Wybierając towarzyszy dla r_1 musimy teraz upewnić się, że dla każdego z nich $g_i < r_1$. Jak szybko znaleźć takie zwierzęta?

Nic nie stoi na przeszkodzie, by pamiętać drugą tablicę tygrysów, posortowaną względem g. Oznaczamy ją `tg[]` w odróżnieniu od pierwszej `tr[]`. Schemat jest następujący: znajdujemy najmniejszego nieprzydzielonego tygrysa w `tr[]`, przeglądamy wolne tygrysy w `tg[]` dopóki spełniona jest nierówność $g_i < r_1$ i oznaczamy je jako zameldowane, całość powtarzamy dopóki są wolne tygrysy. Oto pseudokod rozwiązania:

```
for i := 1 to n do
begin
    g[i] := r[i] / k[i];
    tr[i] := i;
    tg[i] := i;
    bezdomny[i] := true;
end;

sortuj tr[] względem r[i];
sortuj tg[] względem g[i];
j := 1;
wybiegi := 0;

for i := 1 to n do
    if (bezdomny[tr[i]]) then
    begin
        inc(wybiegi);
        while (j <= n) and (g[tg[j]] < r[tr[i]]) do
        begin
            bezdomny[tg[j]] := false;
            inc(j);
        end;
```

```
    end;
wypisz(wybiegi);
```

Dowód poprawności algorytmu jest bardzo podobny do tego dla prostego przypadku. Rozwiązanie to można znaleźć w pliku *tyg.cpp*

3 Inne rozwiązania

3.1 Rozwiązanie alternatywne 1

Zastanówmy się, co by się stało, gdybyśmy nie sortowali po g i po prostu przeglądali `tr[]`, przypisując najmniejszemu tygrysowi wszystkie aż do pierwszego i takiego, że $r_1 k_i \leqslant r_i$. Okazuje się, że ... to rozwiązanie też jest w pełni poprawne! Każdemu wybiegowi przydzieli tego samego najmniejszego tygrysa, co rozwiązanie wzorcowe, jedynie przydział reszty zwierząt może być inny. Choć rozwiązanie to jest prostsze w implementacji od wzorcowego, jego poprawność jest o wiele mniej oczywista. Zaimplementowano je w *tyg*1.*cpp*.

3.2 Rozwiązanie alternatywne 2

Tym razem przeglądamy `tr[]` od największych do najmniejszych. Aby skonstruować pierwszą klatkę idziemy po kolei od największego tygrysa do coraz mniejszych, i dla każdego z nich obliczajmy g_i. W każdym momencie tej iteracji pamiętamy największą spośród g_i (jest to największy rozmiar tygrysa, który zostałby pożarty w tej klatce) i w momencie, gdy napotkamy pierwszego tygrysa mniejszego od utrzymywanego limitu, kończymy kompletowanie pierwszej klatki i kontynuujemy algorytm w ten sam sposób dla pozostałych tygrysów.

Poprawność rozumowania wynika znów z podobnego argumentu co wcześniej: żadne dwa tygrysy spośród „zaczynających" wybiegi nie mogą mieszkać razem. Kod tego rozwiązania umieszczono w pliku *tyg*2.*cpp*.

3.3 Rozwiązanie alternatywne 3

Co by się stało, gdybyśmy w rozwiązaniu wzorcowym nie sortowali tygrysów, tylko wybierali najmniejszego bezdomnego osobnika, a potem jego towarzystwo zwyczajnie przeglądając wszystkie zwierzęta? Algorytm, choć poprawny, pewnie działałby zbyt wolno - w czasie $O(mn)$, gdzie m to liczba zajętych wybiegów. Pomyślmy jednak jak duże może być m. Niech $r_1, r_2 \ldots r_m$ oznaczają rozmiary najmniejszych tygrysów w poszczególnych wybiegach. $r_1 k_i \leqslant r_i$ i $k_i \geqslant 2$, zatem każdy kolejny z m tygrysów jest co najmniej 2 razy większy od poprzedniego i $r_1 2^m \leqslant r_m$, co z kolei prowadzi do $m \leqslant log_2 r_m \leqslant log_2 10^9 \leqslant 30$. Podane rozwiązanie działa zatem prawie tak szybko, co wzorcowe.