Jacek Tomasiewicz Treść zadania, Opracowanie

Aleksander Łukasiewicz Program

Dostępna pamięć: 128 MB. *OI, etap III, dzień pierwszy, 16.04.2015*

Wilcze doły

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

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

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

Wejście

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

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

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

Wyjście

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

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
5
```

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

Testy „ocen”:

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

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

Rozwiązanie

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

Rozwiązanie siłowe $O(n^3)$

Każdy spójny fragment ciągu możemy rozpatrzeć osobno poprzez sprawdzenie, czy jest on poprawny. Sprawdzenie poprawności fragmentu możemy łatwo wykonać w czasie liniowym: wystarczy przykładać początek deski w każdym możliwym miejscu, jednocześnie aktualizując sumę elementów poza deską. Przy przesuwaniu deski o jedną pozycję w prawo, sumę elementów poza deską można aktualizować w czasie stałym. Jako że wszystkich spójnych fragmentów jest $O(n^2)$, a pojedyncze sprawdzenie zajmuje czas liniowy względem długości fragmentu, złożonością czasową tego rozwiązania jest $O(n^3)$.

Rozwiązanie to zaimplementowane jest w pliku `wils1.cpp`. Za poprawne zaprogramowanie takiego rozwiązania na zawodach można było uzyskać około 20% punktów.

Rozwiązanie wolne $O(n^2 \log n)$

Zauważmy, że jeżeli istnieje poprawny fragment długości s, to istnieją też krótsze poprawne fragmenty o długościach $s - 1$, $s - 2$, $\ldots$, d. Natomiast jeśli nie istnieje poprawny fragment o długości s, to nie istnieją też poprawne fragmenty o długościach większych niż s. Dzięki tej obserwacji, długość najdłuższego fragmentu może zostać wyszukana binarnie. W ten sposób zredukujemy nasz problem do logarytmicznej liczby pytań czy istnieje poprawny fragment o ustalonej długości.

Wszystkich fragmentów o ustalonej długości s jest $O(n)$. Jeśli każdy z nich sprawdzimy w czasie liniowym względem długości rozpatrywanego fragmentu, to otrzymamy całkowitą złożoność czasową $O(n^2 \log n)$.

Implementacja takiego rozwiązania znajduje się w pliku `wils2.cpp`. Rozwiązanie tego typu otrzymywało na zawodach około 30% punktów.

Rozwiązanie szybkie $O(n \log^2 n)$

Podobnie jak w rozwiązaniu wolnym, najdłuższy poprawny fragment będziemy wyszukiwać binarnie. Następnie przejrzymy wszystkie fragmenty o zadanej długości s. Przypomnijmy, że poprawny fragment to taki, którego suma elementów z pominięciem pewnego spójnego kawałka o długości d, nie przekracza liczby p. Opłaca się zatem, żeby pominięty kawałek miał jak największą sumę elementów.

Aby móc efektywnie obliczać sumy elementów w dowolnym fragmencie, w pierwszym kroku wyznaczymy ciąg sum częściowych ciągu (w_i). i-tą sumę częściową (dla $1 \leqslant i \leqslant n$) definiujemy jako $a_i = w_1 + w_2 + \ldots + w_i$, jednocześnie przyjmując $a_0 = 0$. Zauważmy, że $a_i = a_{i-1} + w_i$, więc ciąg $a_1, \ldots, a_n$ można obliczyć w czasie $O(n)$. Wartości a_i pozwalają obliczyć sumę elementów w dowolnym fragmencie $w_i, \ldots, w_j$ w czasie stałym ze wzoru $a_j - a_{i-1}$.

Pozostaje pokazać, jak znajdować kawałek długości d o maksymalnej sumie. W tym celu utwórzmy ciąg liczb reprezentujących sumy elementów w kolejnych kawałkach o długości d. Dokładniej, niech $x_i = w_i + w_{i+1} + \ldots + w_{i+d-1} = a_{i+d-1} - a_{i-1}$, gdzie $1 \leqslant i \leqslant n - d + 1$. Zauważmy, że optymalny kawałek o długości d zawarty we fragmencie $w_i, \ldots, w_j$ (gdzie $j - i + 1 \geqslant d$) ma sumę $\max(x_i, x_{i+1}, \ldots, x_{j-d+1})$.

Poniższy algorytm sprawdza czy istnieje poprawny fragment o długości $s \geqslant d$:

```
for i := 1 to n − s + 1 do begin
  j := i + s − 1; { wyznaczamy koniec przedziału }
  suma := a_j − a_{i−1}; { liczymy sumę całego przedziału }
  if suma − max(x_i, x_{i+1}, ..., x_{j−d+1}) ⩽ p then
    return true;
end
return false;
```

Znajdowanie maksymalnego (lub minimalnego) elementu w zadanym spójnym fragmencie ciągu jest dosyć częstym i standardowym problemem, znanym pod nazwą *RMQ* (ang. Range Minimum Query). Istnieje kilka klasycznych struktur danych rozwiązujących ten problem. Drzewo przedziałowe (opisane na przykład w opracowaniu zadania *Tetris 3D* z I etapu XIII Olimpiady Informatycznej [13]) pozwala znajdować największą wartość we fragmencie w czasie $O(\log n)$. Przy użyciu pamięci rzędu $O(n \log n)$, można także zastosować strukturę podobną do słownika podsłów bazowych (opisanego np. w [24]), pozwalającą odpowiadać na zapytania w czasie stałym. Ta metoda była jednak trudna do wykorzystania na zawodach właśnie ze względu na duży narzut pamięciowy. Znana jest także optymalna struktura danych dla problemu RMQ, która po przetworzeniu ciągu w czasie $O(n)$, odpowiada na zapytania w czasie $O(1)$. Jest ona jednak dość trudna w implementacji i dlatego jest rzadko stosowana w praktyce.

Z tych względów, do zaimplementowania tego rozwiązania najlepiej było użyć drzewa przedziałowego. Wtedy całe rozwiązanie działa w czasie $O(n \log^2 n)$. Takie rozwiązanie otrzymywało na zawodach około 80% punktów i zostało zaimplementowane w pliku `wils5.cpp`.

Rozwiązanie prawie optymalne $O(n \log n)$

Aby otrzymać prostsze i bardziej efektywne rozwiązanie naszego zadania, można zauważyć, że występujące w nim zapytania o maksymalną wartość w przedziale są bardzo szczególnej postaci – przedział, o który pytamy, „pełznie” przez tablicę. Wystarczy nam zatem struktura danych podobna do kolejki, udostępniająca operacje wstawiania elementu na koniec, usunięcia elementu z początku oraz odczytywania maksimum. Jeśli umielibyśmy wykonywać takie operacje w zamortyzowanym czasie stałym, sprawdzenie pojedynczej długości w wyszukiwaniu binarnym moglibyśmy zrealizować w czasie liniowym.

Szczęśliwie, taka struktura danych wystąpiła już w rozwiązaniach zadań olimpijskich: w zadaniu *Temperatura* z II etapu XVIII Olimpiady Informatycznej [18] oraz w zadaniu *Piloci* z III etapu XVII Olimpiady Informatycznej [17].

Całkowita złożoność czasowa tego rozwiązania wynosi zatem $O(n \log n)$. Jego implementacja znajduje się w pliku `wils11.cpp`. Za poprawny program tego typu można było uzyskać około 90% punktów.

Rozwiązanie wzorcowe $O(n)$

Aby otrzymać algorytm optymalny, zmodyfikujemy poprzednie rozwiązanie poprzez pozbycie się wyszukiwania binarnego. Zauważmy, że jeśli fragment $w_i, \ldots, w_j$ jest poprawny, to nieznacznie krótszy fragment $w_{i+1}, \ldots, w_j$ (gdzie $j - i \geqslant d$), również jest poprawny. Dzięki tej obserwacji możemy użyć tzw. metody gąsienicy – będziemy wydłużali z prawej strony fragment będący kandydatem na optymalne rozwiązanie, dopóki będzie on poprawny. Gdy takie rozszerzenie nie będzie możliwe, przesuniemy początek fragmentu-kandydata o jedną pozycję w prawo. W ten sposób dla każdej możliwej pozycji i początku fragmentu znajdziemy najdalszą pozycję j taką, że fragment $w_i, \ldots, w_j$ jest poprawny. W szczególności, tym sposobem na pewno nie pominiemy żadnego z optymalnych fragmentów.

```
wynik := d
j := d;
for i := 1 to n − d + 1 do begin
   j := max(j, i + d − 1);
   { niezmiennik: w_i, ..., w_j jest poprawny }
   while j+1 ⩽ n and (a_{j+1} − a_{i−1}) − max(x_i, x_{i+1}, ..., x_{j−d+2}) ⩽ p do begin
      j := j + 1;
   end
   wynik := max(wynik, j − i + 1);
end
return wynik;
```

Zauważmy, że w powyższym algorytmie wartość zmiennej j nie maleje, a przy każdym obrocie pętli **while** jest zwiększana. Dodatkowo $j \leqslant n$, więc wewnętrzna pętla wykona $O(n)$ obrotów.

Tak jak poprzednio, aby obliczać wartości $\max(x_i, x_{i+1}, \ldots, x_{j-d+2})$, potrzebujemy struktury danych, która wyznacza maksymalną wartość w spójnym fragmencie ciągu (x_i). Jeżeli użyjemy drzewa przedziałowego, uzyskamy rozwiązanie działające w czasie $O(n \log n)$. Jeśli natomiast skorzystamy z dwustronnej kolejki utrzymującej maksimum z przechowywanych wartości, uzyskamy optymalny algorytm liniowy.

Implementacje rozwiązań z użyciem kolejki znajdują się w plikach `wil.cpp` oraz `wil1.cpp`, a rozwiązanie używające drzewa przedziałowego znajduje się w pliku `wil3.cpp`. Poprawne zaprogramowanie dowolnego z tych rozwiązań było nagradzane maksymalną liczbą punktów.

Testy

Testy były podzielone na 9 grup. Każda z nich zawierała testy następujących typów:

- całkowicie losowe,
- większość dołów głębokich plus oaza o płytkich dołach,
- doły tworzące lejek,
- poprzeplatane płytkie i głębokie doły.