3A – Wystawa

autor zadania: Mateusz Radecki

Treść

Dany jest ciąg par liczb (a_i, b_i) oraz liczba k. Należy stworzyć ciąg $c_1, c_2, \cdots, c_n$. W tym celu, musimy wybrać dokładnie k pozycji i ustalić na nich $c_i := a_i$. Na pozostałych pozycjach należy ustalić $c_i := b_i$. Naszym celem jest zminimalizowanie maksymalnej sumy liczb tworzących spójny przedział ciągu $c_1, c_2, \ldots, c_n$. Dodatkowo, jesteśmy proszeni o odtworzenie wyniku, tzn. skonstruowanie dokładnego ciągu $c_1, c_2, \ldots, c_n$.

Rozwiązanie

Spróbujmy stworzyć optymalny ciąg bez względu na parametr k. W tym celu dla każdej pozycji i wybieramy mniejszą z dwóch wartości: a_i oraz b_i. W tak stworzonym ciągu któregoś wyboru dokonaliśmy za dużo razy – bez straty ogólności załóżmy, że za dużo razy wybraliśmy a_i. Możemy zatem wywnioskować, że na każdej pozycji, na której wybraliśmy b_i, był to wybór którego nie należy zmieniać. Możemy zatem na tych pozycjach ustalić $a_i := b_i$. Dzięki temu zyskujemy założenie, że dla każdego i zachodzi $a_i \leq b_i$, które okaże się być wygodne podczas implementacji. Musimy teraz wybrać dokładnie k pozycji, na których zdecydujemy się na wybór a_i. Zauważmy, że wspomniane przypisanie sprawia, że możemy myśleć o wybraniu co najwyżej k pozycji, na których zdecydujemy się na wybór a_i, co również może okazać się wygodne, zależnie od szczegółów implementacji.

Zacznijmy od bardzo wolnego rozwiązania i optymalizujmy je. W tym celu warto zastanowić się, jakie znamy algorytmy liczące przedział o największej sumie i który będzie dla nas przydatny. Skorzystamy tutaj z algorytmu, który iteruje się po kolejnych prefiksach i dla każdego z nich oblicza maksymalną sumę liczb na jakimś sufiksie tego prefiksu. Powiedzmy, że rozważyliśmy już prefiks długości i, a największa suma liczb na jego sufiksie wynosi s. Przechodząc do prefiksu długości $i + 1$ ustalamy $s := s + a_{i+1}$. Następnie, mamy możliwość wyzerować długość sufiksu o największej sumie – jeśli największy sufiks zawiera ostatni element, to przed nim na pewno używamy dokładnie tego sufiksu, który był optymalny dla prefiksu krótszego o 1. Jeśli go nie zawiera, to suma liczb w optymalnym sufiksie to po prostu 0. W tym celu ustalamy $s := max(s, 0)$.

Pamiętamy również globalną zmienną m, ustaloną początkowo na 0. Po każdym przedłużeniu prefiksu ustalamy $m := max(m, s)$. Końcowa wartość zmiennej m jest szukanym wynikiem. Pozwala nam to skonstruować rozwiązanie wykładnicze. Idąc od lewej, rekurencyjnie rozważamy dla każdej pozycji, czy zdecydujemy się na wybranie na niej wartości a_i czy b_i. Interesującymi nas wartościami, są oczywiście wartości s oraz m, a także liczba pozycji na których zdecydowaliśmy się na wybór a_i. Możemy zatem opisać stan krotką czterech liczb: (i, j, s, m), gdzie i oznacza długość rozważonego prefiksu, a j oznacza liczbę pozycji, na których wybraliśmy element ciągu $a_1, a_2, \ldots, a_n$.

W podobnych sytuacjach, gdy musimy minimalizować maksimum po jakichś wartościach, warto jest rozważyć wyszukiwanie binarne po wyniku. Sprawia to, że mamy globalne założenie, którego musimy pilnować. Spróbujmy więc zastosować je tutaj i skupmy się na sprawdzeniu, czy da się sprawić, by suma liczb w żadnym przedziale nie przekraczała X. Okazuje się zmieniać to stan z (i, j, s, m) w (i, j, s) – nie interesuje nas maksymalna do tej pory wartość s, ważne jedynie, że nigdy nie przekroczyła ona X.

Pozwala to nam wysunąć obserwację: dla ustalonych wartości i oraz j nie interesują nas wszystkie możliwe wartości s, do których mogliśmy dojść bez przekraczania X – interesuje nas jedynie najmniejsza z nich! Możemy zatem użyć programowania dynamicznego: niech $DP[i][j]$ oznacza minimalną możliwą do osiągnięcia wartość s, przy założeniu, że rozważyliśmy już prefiks długości i, j razy wybraliśmy na nim element ciągu $a_1, a_2, \ldots, a_n$ i suma w żadnym przedziale do tej pory nie przekroczyła X. Dodatkowo, niektóre j mogą w ogóle nie być osiągalne. Oznacza to, że nie mogliśmy wybrać tylko tyle wartości a_i, żeby nigdzie nie przekroczyć X. Osiągalny będzie sufiks możliwych wartości j. Aby nieco uprzyjemnić implementację warto w tym momencie zmienić definicję – ustalamy zatem $k := n - k$ i mówimy, że musimy co najmniej k razy wybrać wartość b_i (czyli tę większą). Sprawi to, że tylko prefiks wartości j będzie osiągalny. Powyższe podejście ma złożoność kwadratową, więc potrzebujemy jeszcze jednej obserwacji.

Zaimplementowanie powyższego programowania dynamicznego pozwala zauważyć ważną własność: dla ustalonego i, wartości $DP[i][j]$ są oczywiście niemalejące, ale również wypukłe! Innymi słowy, wartości $DP[i][j] - DP[i][j-1]$ również są niemalejące. Doświadczeni zawodnicy powinni w tym momencie wiedzieć, co należy zrobić. Zamiast utrzymywać wszystkie wartości $DP[i][j]$, utrzymujmy jedynie różnice między nimi oraz wartość $DP[i][0]$. Zastanówmy się, jak zmieniają się owe różnice przy przejściu do następnego prefiksu. Musimy wykonać $DP[i][j] = max(min(DP[i-1][j-1] + b_i, DP[i-1][j] + a_i), 0)$. Spróbujmy rozłożyć to przejście na nieco mniejsze składowe. Po pierwsze, możemy każdą wartość zwiększyć o a_i – wystarczy w tym celu zwiększyć $DP[i][0]$ o a_i oraz nie zmieniać różnic. Następnie, dla każdego elementu jednocześnie, musimy wybrać mniejszą z dwóch możliwości – $DP'[i][j] = min(DP[i][j], DP[i][j-1] + b_i - a_i)$. Warto zauważyć, że skoro różnice były posortowane, to opisana operacja jest równoważna włożeniu wartości $b_i - a_i$ do posortowanego ciągu różnic. Sprawia to, że możemy do utrzymywania tego ciągu użyć gotowych struktur danych oferowanych przez popularne języki programowania.

Następną operacją, którą musimy wykonać, jest zwiększenie wszystkich ujemnych wartości do zera. Ponieważ nadal wartości $DP[i]$ są niemalejące, to jedynie prefiks wartości mógł stać się ujemny, musimy więc go zwiększyć. Zauważmy, że jeśli musielibyśmy zwiększyć do zera wiele pierwszych wartości, to różnice między nimi staną się zerami. Aby nasze rozwiązanie było dostatecznie szybkie, możemy zamiast trzymać w strukturze wszystkie zerowe różnice, trzymać jedynie ich liczbę – dzięki temu nie będziemy musieli się po nich wszystkich iterować i nasze rozwiązanie będzie się dobrze amortyzowało – potencjałem będzie liczba niezerowych różnic.

Ostatnią rzeczą jest ewentualne usunięcie sufiksu wartości, które przekraczają X. Tutaj musimy oprócz wartości $DP[i][0]$ i ciągu różnić pamiętać również sumę różnic. Pozwoli nam to łatwo obliczać ostatni element ciągu, stwierdzać czy jest on większy niż X i ewentualnie usuwać ostatnie wartości.

Umiemy zatem powiększać prefiks o 1 w złożoności $\mathcal{O}(\log(n))$. Po przejściu przez cały ciąg wystarczy sprawdzić, czy w strukturze mamy co najmniej k różnic, czyli czy wartość $DP[n][k]$ jest osiągalna. Jeśli tak, to mogliśmy wybrać co najmniej k różnych pozycji i ustalić na nich b_i tak, by suma liczb w żadnym przedziale nie przekroczyła X. Zauważmy również, że z przebiegu algorytmu możemy wyciągnąć wszystkie informacje potrzebne nam do odtworzenia wyniku – możemy dla każdej różnicy pamiętać której pozycji ona odpowiada.

Całe rozwiązanie, ze względu na wyszukiwanie binarne oraz użycie odpowiedniej struktury danych, ma złożoność $\mathcal{O}(n \cdot \log(n) \cdot \log(n \cdot A))$, gdzie A jest górnym ograniczeniem na wartości w ciągach $a_1, a_2, \ldots, a_n$ oraz $b_1, b_2, \ldots, b_n$.