Informatyczny kącik olimpijski (12) – najdłuższy wspólny podciąg inaczej

W tym kąciku przyjrzymy się z trochę innej perspektywy bardzo znanemu zadaniu.

Mamy dane dwa ciągi, $A_1, \ldots, A_n$ oraz $B_1, \ldots, B_m$. Zadanie polega na znalezieniu *najdłuższego wspólnego podciągu* tych dwóch ciągów, czyli najdłuższego ciągu, będącego jednocześnie podciągiem A i B.

Rozwiązanie zadania F 723.
Załóżmy, że nić wydłużyła się o Δl. Wtedy cały układ opuści się o $3\Delta l$, a środek ciężkości o $\frac{3}{2}\Delta l$. Praca wydatkowana na przesunięcie środka ciężkości jest równa pracy rozciągnięcia nici, czyli

$$\frac{3}{2}Q\Delta l = N\Delta l.$$

Otrzymujemy stąd, że

$$N = \frac{3}{2}Q.$$

Wielu Czytelników zapewne zna już rozwiązanie. Oblicza się w tym celu tablicę T, taką że $T_{i,j}$ jest długością najdłuższego wspólnego podciągu fragmentów $A_1, A_2, \ldots, A_i$ i $B_1, B_2, \ldots, B_j$. Jeśli $i = 0$ lub $j = 0$, to $T_{i,j} = 0$. W przeciwnym przypadku, jeśli $A_i = B_j$, to $T_{i,j} = T_{i-1,j-1} + 1$. W końcu, jeśli nie zachodzi żaden z poprzednich przypadków, to $T_{i,j} = \max(T_{i,j-1}, T_{i-1,j})$. W ten sposób, po wykonaniu $O(nm)$ kroków, otrzymujemy wartość $T_{n,m}$. Jeśli najpierw obliczymy pierwszy wiersz tablicy T, następnie drugi itd. (lub, analogicznie, najpierw pierwszą kolumnę, potem drugą itd.), to będziemy sięgać do komórek obliczonych najwyżej m (lub n) kroków temu – a więc złożoność pamięciową możemy ograniczyć do $O(\min(n, m))$.

Łatwo poszło! No dobrze, a teraz zmieńmy wariant. Załóżmy, że mamy gwarancję, iż żaden element nie występuje więcej niż k razy ani w A, ani w B. Możemy oczywiście pozostać przy naszym rozwiązaniu. Ale można też znaleźć lepsze!

Gdzie tak naprawdę w poprzednim rozwiązaniu dzieje się cała „magia” – innymi słowy, w którym momencie tak naprawdę odnajdujemy fragmenty ciągów, które do siebie pasują? Oczywiście wtedy, gdy dodajemy jedynkę – czyli w momencie, kiedy wykonujemy operację $T_{i,j} = T_{i-1,j-1} + 1$. Gdyby pominąć ten jeden krok, to jedynym, co byśmy robili, byłoby „przepisywanie” dotychczas uzyskanych wyników do kolejnych komórek tablicy.

Ograniczenie liczby wystąpień tego samego symbolu daje też ograniczenie liczby wykonań operacji $T_{i,j} = T_{i-1,j-1} + 1$. Jeśli wybierzemy określone i, to ile jest takich j, dla których wykonamy ten krok? Oczywiście k, ponieważ wykonujemy go tylko wtedy, gdy $A_i = B_j$. A więc, globalnie, skoro $1 \leqslant i \leqslant n$, to liczba takich kroków w całym przebiegu algorytmu jest ograniczona przez kn (jeśli z kolei oszacujemy z „drugiej strony” – tj. zapytamy, ile jest takich i dla określonego j, że $A_i = B_j$, to otrzymamy drugie ograniczenie, km).

Jakie zadanie z kolei spełnia reszta obliczeń? Powiedzmy, że para indeksów (i, j) jest *pasująca*, jeśli $A_i = B_j$. W pozostałych krokach – w pewnym sensie – kojarzymy pary *pasujące* (i, j) w ciągi rosnące $(i_1, j_1), (i_2, j_2), \ldots$, czyli takie, że $i_1 < i_2 < \ldots$, oraz $j_1 < j_2 < \ldots$. W oryginalnym problemie takich par było dużo – dokładniej, mogło ich być nawet nm, jeśli na wszystkich pozycjach w obu ciągach znajdowałby się ten sam element. Spróbujemy wykorzystać ograniczenie ich liczby, aby przyspieszyć algorytm.

Na początek należałoby wyznaczyć wszystkie pary *pasujące*. Można to robić na wiele sposobów – jednym z nich jest wrzucenie indeksów do kubełków odpowiadających elementom ciągów (można to zrobić, przeglądając raz każdy z ciągów). Potem w każdym kubełku łączymy indeksy z ciągu A z każdym indeksem w ciągu B.

Naszym celem jest ustawienie jak największego podzbioru par *pasujących* w ciąg rosnący. W szczególności, jeśli weźmiemy tylko pierwsze współrzędne tych par, to też muszą one tworzyć ciąg rosnący. Uporządkujmy więc pary według pierwszej współrzędnej. Teraz szukamy takiego podciągu tych par, żeby *różniły* się na pierwszych współrzędnych, i aby drugie współrzędne tworzyły ciąg rosnący. W tym celu zmieńmy jeszcze trochę sposób uporządkowania par – niech

$$(i_1, j_1) < (i_2, j_2) \iff i_1 < i_2 \vee (i_1 = i_2 \wedge j_1 > j_2.)$$

Taka relacja na pewno porządkuje pary rosnąco według pierwszej współrzędnej. Co więcej, jeśli (i_a, j_a) poprzedza w tym porządku (i_b, j_b) i $j_a < j_b$, to również $i_a < i_b$. W takim razie, po posortowaniu par *pasujących* względem tak zdefiniowanego porządku szukamy podciągu par, który spełnia tylko jeden warunek – drugie współrzędne mają tworzyć ciąg rosnący. Tym samym sprowadziliśmy nasz problem do problemu znajdowania najdłuższego podciągu rosnącego.

Dla problemu znajdowania najdłuższego podciągu rosnącego ciągu Q istnieją znane algorytmy, działające w czasie $O(\|Q\| \log \|Q\|)$ (o tym problemie była też mowa w *Logomotywach*, *Delta* 7/2008). W naszym przypadku daje to czas $O(k \min(n, m) \log(k \min(n, m)))$ – co, dla odpowiednio małych k, jest ulepszeniem w stosunku do $O(nm)$. W kolejnym odcinku omówimy jeden z takich algorytmów. Będzie on wykorzystywał ciekawą, prostą do zaimplementowania strukturę danych, przydatną nie tylko w tym, ale i w wielu innych zadaniach.

Filip WOLSKI