Lista 6

Kamil Matuszewski

11 maja 2016

1	2	3	4	5	6
✓		✓	✓		

Zadanie 1

Ułóż algorytm, który dla danego ciągu liczbowego oblicza, ile zawiera on podciągów rosnących maksymalnej długości.

Po pierwsze odsyłam do listy 4 i zadania 9. Będziemy modyfikować tamten algorytm. Po drugie, jeśli rozumiecie tamto, to to jest proste, choć wygląda na długie, bo chciałem ładnie opisać.
Od teraz będę zakładał, że czytelnik umie znaleźć najdłuższy podciąg rosnący tamtym sposobem.

Okej. Najpierw zapomnijmy o algorytmie, ułożę go później. Podobnie jak w znajdowaniu LIS, chcemy stworzyć tablicę $t_i(j)$. Jednak będzie ona tablicą list. Zamiast zastępować element elementem mniejszym, będziemy dodawać go na koniec listy. Zauważmy, że w takim razie każda lista będzie posortowana malejąco - przyda się nam to później.
Dodatkowo, w liście nie będziemy trzymać pojedynczego elementu, ale parę elementów. Dokładniej, parę (a, b).
Trochę to dziwnie brzmi. Podsumujmy więc. Mamy tablicę $t_i(j)$, których elementami są listy par (a_k, b_k). Mówi to nam tyle, że dowolne b_k na tej liście, jest liczbą podciągów kończących się na a_k o długości j prefiksu $a_1, \ldots, a_i$.
Zwróćmy uwagę, że t_k i t_{k+1} różnią się długością którejś listy. Dokładniej w t_{k+1} gdzieś jest jeden element więcej.
Teraz przejdźmy do algorytmu. Załóżmy, że mamy już jakąś tablicę, i chcemy dołożyć do niej kolejny element. Na wejściu dostajemy tylko a. Znajdujemy więc (binarnie) największą taką pozycję $j-1$, że $a > t[j-1].last.first$ (pierwszy element pary ostatniego elementu listy). Wtedy do $t[j]$ dokładamy naszą parę (a, b). Teraz sprawdźmy jak wygląda b.
Nasz podciąg długości j został utworzony z jakiegoś podciągu długości $j-1$. a dołożyć mogliśmy tylko do jakiegoś podciągu, kończącego się na element mniejszy od a. Szukamy tych podciągów w liście $t[j-1]$. Jeśli mamy tam element (a', b'), taki, że $a > a'$, to możemy utworzyć b' podciągów długości j dokładając a do wszystkich pociągów zakończonych na a'. Skoro tak, to $b = \sum_{k:\ a_k < a} b_k$.
Teraz, skoro lista $t[j-1]$ jest posortowana, możemy zacząć od końca i szukać pierwszego takiego elementu który jest większy od a, i sumować wszystkie b_k przez które przejdziemy.
Wtedy liczba wszystkich ciągów długości i będzie sumą po wszystkich b_k na liście $t[i]$.
Teraz, działa to trochę za wolno. Ta suma strasznie nas boli. Możemy jednak to zoptymalizować, pamiętając trochę inne b. Zamiast liczby podciągów kończących się na a, b będzie oznaczać sumaryczną liczbę podciągów kończącą się na element a bądź większy.
Jak teraz wygląda dodawanie elementu? Wstawianie wygląda dokładnie tak samo. Zmienia się wyliczanie b. Teraz, jeśli wstawiliśmy (a, b) do $t[j]$, w której wcześniej ostatnim elementem było (a', b'), w $t[j-1]$ szukamy (np binarnie) par (a_k, b_k) i (a_{k+1}, b_{k+1}) (dwa kolejne elementy listy), takich, że $a_k > a > a_{k+1}$. Nasze $b = b_{k+1} - b_k + b'$ (liczba wszystkich podciągów zakończonych na a_{k+1} to $b_{k+1} - b_k$ bo od podciągów zakończonych na a_{k+1} włącznie odejmujemy podciągi zakończone na a_k włącznie. Do tego dodajemy liczbę podciągów dla elementów większych niż a, czyli b').
Widzimy, że b obliczamy teraz logarytmicznie. Skoro tak to wyszukujemy miejsce na a logarytmicznie, i znajdujemy b logarytmicznie, mamy więc $O(2 \log n) = O(\log n)$. Sumarycznie mamy

więc $O(n \log n)$.
Algorytm (omijam jakieś brzegowe przypadki, że w tej liście nie ma elementów albo że w poprzedniej jest jeden czy coś, to każdy se rozpatrzy):

Dane: Tablica liczb $a_1 \dots a_n$.
Wynik: Liczba podciągów rosnących o maksymalnej długości

Niech t - tablica rozmiaru n;
for $i \leftarrow 1$ **to** n **do**
  $t[i] \leftarrow new_list$;
end
for $i \leftarrow 1$ **to** n **do**
  $j \leftarrow BinarySearch(a_i, t)$;
  $prevlast = t[j].size - 1$;
  $t[j].push\_back(a_i, b)$;
  $k \leftarrow BinarySearch2(a_i, t[j-1])$;
  $b = t[j-1][k+1] - t[j-1][k] + t[j][prevlast]$;
end
$s \leftarrow 1$;
while $t[s] \neq \infty \ AND \ s < n$ **do**
  $s{+}{+}$;
end
return s

Algorytm 1: Liczba podciągów rosnących o maksymalnej długości

Gdzie:

$BinarySearch(a_i, t)$ zwraca najmniejszy taki indeks j, że $a_i < t[j].last.first$

$BinarySearch2(a, t[j-1])$ zwraca indeks i taki, że $t[j-1][i].first > a > t[j-1][i+1].first$

Zadanie 3

Niech $h(v)$ oznacza odległość wierzchołka v do najbliższego pustego wskaźnika w poddrzewie o korzeniu v. Rozważ drzewa binarne, zrównoważone przy użyciu warunku

$$h(v.lewy) \geqslant h(v.prawy) \ \forall v$$

do implementacji złączalnych kolejek priorytetowych.

Kolejka priorytetowa to struktura przechowująca dane obdarzone priorytetami, umożliwiająca łatwy dostęp do elementu o najwyższym priorytecie - im mniejsza liczba tym większy priorytet. Struktury te mają operacje wstawiania elementu i usuwania elementu o najwyższym priorytecie. By je zaimplementować używa się kopców, czyli drzew dla których dla każdego poddrzewa, priorytet korzenia jest większy bądź równy każdemu innemu priorytetowi w tym poddrzewie.
Do kopca binarnego (czyli kopca używającego drzewa binarnego) dokładamy warunek zrównoważenia zdefiniowany wyżej. Musimy powiedzieć, że taka struktura ułatwia nam łączenie kopców, zachowując zrównoważenie i warunek kopca. Wtedy usuwanie najmniejszego elementu to usunięcie korzenia i złączenie dwóch kopców z lewego i prawego syna, a dodanie elementu to złączenie kopca oryginalnego i kopca złożonego z jednego elementu. Jedyne co musimy powiedzieć, to jak łączyć kopce.
Aby połączyć dwa niepuste kopce, oznaczmy przez d_1 kopiec o większym priorytecie w korzeniu, a przez d_2 ten drugi. Rekurencyjnie łączymy prawe poddrzewo d_1 i d_2 dostając d_3. Nasz kopiec wynikowy d wygląda tak, że korzeń d to korzeń d_1, prawe poddrzewo d to $d' = max(h(d_1), h(d_3))$, a lewe to $d'' = min(h(d_1), h(d_3))$. Niech v - korzeń d. Wtedy $h(d) = h(d.right) + 1$.

Zadanie 4

Zmodyfikuj drzewo AVL tak, by operacje $nastepnik(v)$ i $poprzednik(v)$ były wykonywane w czasie stałym.

Najpierw doprecyzujmy: $nastepnik(v)$ to najmniejszy element w drzewie większy od zadanego, $poprzednik(v)$ to największy element w drzewie mniejszy od zadanego.

Żeby to zrobić, zapamiętamy w każdym wierzchołku *nastpnik* i *poprzednik*, które będą wskazywać na następnika i poprzednika. Teraz, co z operacjami? Wyszukiwanie zbytnio się nie zmieni. Możemy ewentualnie wyszukiwać po następniku i poprzedniku.
Usuwanie elementu. Usuwamy element jak w drzewie AVL. Pamiętamy przez n następnika usuwanego elementu a przez p jego poprzednika. Teraz idziemy do p, i jego następnika ustawiamy na n, a potem idziemy do n i jego poprzednika ustawiamy na p.
Co do dodawania elementu - dodaj go do drzewa AVL. Przed rotacją spójrz na ojca. Jeśli jesteś lewym synem ojca, jest on twoim następnikiem, twoim poprzednikiem jest jego poprzednik, a jego poprzednikiem jesteś ty. Jeśli jesteś prawym synem ojca, jest on twoim poprzednikiem, twoim następnikiem jest jego następnik, jego następnikiem jesteś ty. Idź do swojego poprzednika/następnika i zmień mu następnika/poprzednika na siebie.