Lista 4

Kamil Matuszewski

10 maja 2016

1	2	3	4	5	6	7	8	9	10
	✓	✓	✓	✓		✓	✓	✓	✓

Zadanie 2

Ułóż algorytm najtańszego przejścia przez tablicę, przy założeniu, że z pola (i, j) możemy przejść na $(i+1, j), (i-1, j), (i-1, j+1), (i, j+1), (i+1, j+1)$ gdzie i to numer wiersza a j - kolumny.

To się robi jakoś podobnie jak na wykładzie. Niech *cena* oznacza wartość na danym polu, a *koszt* oznacza najmniejszą cenę wejścia na te pole. Wtedy:
Wypełnimy pierwszą kolumnę tak, że $koszt = cena$. Każdą kolejną wypełniamy w ten sposób:

$$koszt(i, j) = min(koszt(i-1, j-1), koszt(i, j-1), koszt(i+1, j-1)) + cena(i, j)$$

Oczywiście, nie chcemy wyjść za tablicę (można np dorobić dodatkowe dwa wiersze wypełnione nieskończonościami). Teraz, musimy zaktualizować nasz koszt, biorąc pod uwagę to, że mogliśmy przyjść z góry:

$$koszt(i, j) = min(koszt(i, j), koszt(i-1, j) + cena(i, j))$$

Oraz to samo z dołu:

$$koszt(i, j) = min(koszt(i, j), koszt(i+1, j) + cena(i, j))$$

W ten sposób otrzymujemy trzy pętle które wykonają się w sumie $3m$ razy, do tego doliczamy pętlę po wszystkich kolumnach i mamy złożoność $O(nm)$. To, że to działa jest w miarę intuicyjne. Najpierw wypełniamy kolumnę jakbyśmy przyszli z prawej, jeśli szybciej do danego pola możemy dojść z góry to tak robimy, jeśli szybciej do danego pola możemy dojść z dołu to to robimy.

Teraz jak odtworzyć tą ścieżkę? Tak samo jak na wykładzie. Bierzemy minimum z ostatniej kolumny, i patrzmy na miejsca z których mogliśmy tutaj dojść. Zawsze biorąc minimum z tych miejsc będziemy szli tą samą ścieżką tylko, że w tył. Dowód poprawności jest prosty. Nie w prost załóżmy, że algorytm w k'tym kroku wybierze pole, które nie wchodzi w skład oryginalnej ścieżki. To by oznaczało, że na pole, z którego przyszliśmy tutaj naszym algorytmem, można dojść mniejszym kosztem, niż w naszej oryginalnej ścieżce, co oznacza, że ta ścieżka nie była minimalna.

Zadanie 3

Ułóż algorytm na znajdowanie odległości edycyjnej ciągów.

Z tego co mi się wydaje, łatwiej najpierw spojrzeć na coś, co ma sporą złożoność pamięciową, tj wypełniamy dwie tablice nxm gdzie n to długość x a m długość y.
Najpierw, ustalmy sobie x, i spróbujmy uzyskać y dozwolonymi operacjami. Potem zrobimy odwrotnie - ustalimy y i spróbujemy na niego przekształcić x. Niech $A(i, j)$ oznacza minimalny koszt zamienienia x o i elementach na y o j elementach, a $B(i, j)$ - minimalny koszt zamiany y o j

elementach na x o i elementach. Wynikiem całości będzie $min(A(n,m), B(n,m))$ Widać, że:

$$A(i,j) = \begin{cases} j \cdot koszt\ insert & jesli\ i == 0 \\ i \cdot koszt\ delete & jesli\ j == 0 \\ min(A(i-1,j-1)+f, A(i-1,j)+koszt\ delete, A(i,j-1)+koszt\ insert) & w.p.p \end{cases}$$

Gdzie $f = 0$ jeśli $x_i = y_j$ albo $f = koszt\ replace$ w.p.p. Dlaczego? Jeśli ciąg x jest pusty ($i == 0$) to musimy dołożyć j elementów, żeby uzyskać y. Jeśli y jest pusty - musimy usunąć i elementów z x żeby uzyskać y.
Rozpatrzmy teraz co możemy zrobić z naszymi ciągami. Możemy albo zmienić $i-ty$ element z x na $j-ty$ element z y, i wywołać problem dla ciągów z usuniętymi tymi elementami ($A(i-1,j-1)+f$, gdzie f to koszt zamiany, czyli jeśli elementy są takie same to jest to koszt zerowy), albo usunąć element $i-ty$ z x ($A(i-1,j)$+koszt usunięcia tego elementu), albo dodać do x'a na pozycję $i+1$'szą element z y, i wywołać dla y'ka krótszego o 1 i x'a tej samej długości ($A(i,j-1)$+koszt wstawienia tego elementu). Oczywiście interesuje nas minimum z tego.
Analogicznie będzie wyglądać $B(i,j)$, z tym, że będą trochę inne operacje.

$$B(i,j) = \begin{cases} j \cdot koszt\ delete & jesli\ i == 0 \\ i \cdot koszt\ insert & jesli\ j == 0 \\ min(A(i-1,j-1)+f, A(i-1,j)+koszt\ insert, A(i,j-1)+koszt\ delete) & w.p.p \end{cases}$$

Teraz, pewnie da się to jakoś ładnie zwinąć do jednego wzoru, ale wydaje mi się, że w ten sposób więcej widać. Możemy albo zamienić x na y, albo y na x. Weźmiemy minimum z tego.
Czasowo mamy to samo, bo liczymy zamiast jednej tabelki dwie. Pamięciowo jest pewnie znacznie gorzej, ale da się to przerobić na wypełnianie jednej tabelki. Wypełniając tą jedną tabelkę musimy trzymać tylko dwie kolumny. No ale po co komplikować, jak ważna jest intuicja, a optymalna implementacja to zupełnie inna sprawa.

Zadanie 4

Znajdź zbiór niezależny drzewa T.

Okej. Trzymajmy dla drzewa parę w postaci (A, B). Teraz wyjaśnię co to znaczy.

A oznacza najliczniejszy zbiór niezależny synów danego wierzchołka **ZAWIERAJĄCY** ten wierzchołek.
B oznacza najliczniejszy zbiór niezależny synów danego wierzchołka **NIE ZAWIERAJĄCY** danego wierzchołka. Wtedy mamy:
$A(i) = 1$ jeśli i jest liściem.
$A(i) = 1 + \sum_{j\ -\ synach} B(j)$
$B(i) = 0$ jeśli i jest liściem.
$B(i) = \sum_{j\ -\ synach} max(A(j), B(j))$
Nasz wynik to oczywiście $max(A(k), B(k))$ gdzie k jest korzeniem.
Teraz, co my tak naprawdę robimy. Zwróćmy uwagę, że dany wierzchołek może albo być albo nie być w naszym maksymalnym zbiorze niezależnym. Jeśli jest, to jego synowie nie mogą być, jeśli nie jest, to jego synowie mogą albo być albo nie być. Stąd jeśli wierzchołek jest w naszym zbiorze niezależnym, to bierzemy sumę po wartościach synów, którzy nie są w tym zbiorze. Jeśli natomiast nie jest, bierzemy sumę po maksimach z tego, czy syn jest, czy też nie jest w zbiorze.
Wypełniamy wartości dla wszystkich wierzchołków, dla każdego wierzchołka zaglądając tylko do jego synów. Możemy to więc zrobić w czasie $O(n)$.

Zadanie 5

Mamy trzyelementowy zbiór $A = \{a, b, c\}$ na którym określamy operację $\circ$. Nie jest ona ani przemienna ani łączna. Ułóż algorytm, który sprawdza, czy da się tak ponawiasować ciąg $x_1 \circ x_2 \circ \cdots \circ x_n$, by wartość wyrażenia wynosiła a.

Wprowadźmy sobie działanie $x \diamond y = z \Leftrightarrow z \circ y = x$. Wszystko dla uproszczenia zapisu.

Spójrzmy na x_n. Otwierając nawias przed x_1 a zamykając przed x_n nie zmienimy kolejności wykonywania działań. Zamykając nawias za x_n nie zmienimy zupełnie nic. Zamykamy więc nawias przed x_n. Możemy go otworzyć w $n-1$ miejscach. Okej. Niech $a \diamond x_n = x$ a $x \diamond y = z$.
Otwierając nawias na początku, redukujemy nasz problem do jednego problemu: czy $x_1 \circ \cdots \circ x_{n-1}$ można ponawiasować tak, żeby otrzymać x. Wtedy $x \circ x_n = a$.
Jeśli otworzymy nawias przed jakimś x_i, mamy inny problem. Czy można ponawiasować $x_1 \circ \cdots \circ x_{i-1}$ tak, by dostać z, oraz, czy możemy tak ponawiasować $x_i \circ \cdots \circ x_{n-1}$ by dostać y. Wtedy $z \circ y = x$ i $x \circ x_n = a$. Niech T będzie rozwiązaniem tego problemu, i indeksem początkowym, j indeksem końcowym,$i < k < j$ miejscem w którym dzielimy, w wartością którą chcemy otrzymać, $w \diamond x_j = x$ a $x \diamond y = z$, to mamy:

$T(i,j,w) = T(i,j-1,x)\ or\ (T(i,k-1,z)\ and\ T(k,j,y))\ dla\ jakiegokolwiek\ k,x,y,z\ spelniajacych\ zalozenia$

Wynik to oczywiście $T(1,n,a)$. Możemy zbudować tablicę o trzech wymiarach. Pierwsza współrzędna będzie nam mówić o indeksie początkowym, druga o końcowym, a trzecia o literze którą chcemy otrzymać. Tablicę wypełniamy zerojedynkowo. Wynik będzie we współrzędnej $1,n,a$. Dla współrzędnych (i,j,x) takich, że $i > j$ możemy wpisać 0 i nawet nie próbować ich liczyć. Zaczynając od przekątnej (wszystkich odległości równych 0, którą wypełniamy 0), możemy obliczać odległości o 1 większe. W ten sposób wypełniamy jakieś $\frac{1}{2} \cdot 3 \cdot n \cdot m$ pól, wynik jest w polu $1,n,a$, złożoność to jakieś $O(n \cdot m)$

Zadanie 7

Rozwiąż problem 3-podziału.

Niech $t(i,j,k)$ mówi nam, czy ciąg $a_1..a_k$ można podzielić na dwa rozłączne podzbiory, sumujące się do i i do j. Łatwo zauważyć, że $t(0,0,0) = true$, $t(i,j,0) = false$ dla $i+j > 0$. Teraz układamy zależność rekurencyjną:

$$t(i,j,k) = \begin{cases} true & dla\ i = 0\ \wedge\ j = 0\ \wedge\ k = 0 \\ false & dla\ i+j > 0\ \wedge\ k = 0 \\ t(i,j,k-1) \vee t(i-a_k,j,k-1) \vee t(i,j-a_k,k-1) & wpp \end{cases}$$

Co jest raczej oczywiste. Wynik znajduje sie oczywiście na polu $t(s/3,s/3,n)$ gdzie s to suma wszystkich elementów.

Zadanie 8

Na każdym polu tablicy $4xn$ znajduje się liczba naturalna. Na każdym z pól może leżeć maksymalnie jeden kamień, a jeśli na jakimś polu leży kamień, to na polach stykających się bokiem z tym polem, kamyki leżeć nie mogą. Ułóż algorytm maksymalizujący sumę liczb na polach na których leżą kamyki.

Ustalmy sobie następujące maski bitowe:

$$\begin{cases} 0000 = 0 \\ 0001 = 1 \\ 0010 = 2 \\ 0100 = 4 \\ 0101 = 5 \\ 1000 = 8 \\ 1001 = 9 \\ 1010 = 10 \end{cases}$$

Dodatkowo powiemy, że konfiguracje są kompatybilne, kiedy nie ma sytuacji, że przy ustawieniu ich jedna pod drugą mamy pod sobą jedynki. Innymi słowy, że $x\&y == 0$. Możemy dla każdej konfiguracji wcześniej wypełnić sobie które konfiguracje są kompatybilne, i dla uproszczenia uznam, że $kompatybilne(x)$ zwraca wszystkie konfiguracje kompatybilne z x.

Stwórzmy sobie tablicę $8xn$ w której $t(i,j)$ oznaczać będzie, jaka jest maksymalna wartość planszy utworzonej przez pierwsze i kolumn przy danej konfiguracji j kolumny $i-tej$. Wtedy

$$t(i,j) = \begin{cases} 0 & dla\ i < 0 \\ max_{k=kompatybilne(j)} t(i-1,k) + w(i,k) \end{cases}$$

Gdzie $w(i,k)$ to wartość kolumny i przy konfiguracji k, czyli inaczej suma wartości pól na których jest jedynka.

Wszystko co musimy zrobić to wypełnić tablicę $8xn$ odwołując się do maksymalnie 8 innych pól (w rzeczywistości sporo mniej). Wynik to oczywiście $max_{k=konfiguracje} t(n,k)$

Zadanie 9

Znajdź najdłuższy podciąg rosnący ciągu $a_1 \dots a_k$ (longest increasing subsequence - LIS) (zadanie z bobrami - bobry to indeksy, ich wybranki to wartości. Zauważmy, że wtedy elementy są różne).

Chciałbym stworzyć tablicę $t_i(j)$ która będzie nam mówić, że $t_i(j)$ jest najmniejszym ostatnim elementem jakiegoś podciągu rosnącego długości j prefiksu $a_1 \dots a_i$. Jeśli taki element nie istnieje, niech $t_i(j) = \infty$. W ten sposób, największy indeks skończonego elementu tablicy t_n będzie nam mówić, jaki jest rozmiar najdłuższego podciągu rosnącego ciągu $a_1 \dots a_n$, co jest naszą szukaną wartością.

Udowodnijmy najpierw kilka rzeczy.

- W tablicy t_k dla danego prefiksu $a_1 \dots a_k$ wszystkie skończone elementy będą ustawione rosnąco.

  Dowód. Jeśli w prefiksie $a_1 \dots a_k$ mamy podciąg długości i zakończony przez $t_k(i)$, to oczywiście w $a_1 \dots a_k$ mamy podciąg długości $i-1$ zakończony $t_k(i)$, bo wystarczy usunąć pierwszy element podciągu długości i. Zachodzi więc $t_k(i-1) < t_k(i)$. Jeśli $t_k(i) = \infty$ to nasze twierdzenie tego elementu nie dotyczy. □

- Każde dwie tablice t_{k-1} oraz t_k różnią się w dokładnie jednym miejscu.

Dowód. Rozważmy poszerzenie naszego prefiksu o a_k. Innymi słowy zmianę z t_{k-1} na t_k. Skoro ciąg jest rosnący to istnieje taki indeks tablicy, powiedzmy j, że elementy $t_{k-1}(1) \dots t_{k-1}(j)$ są mniejsze od a_k a elementy $t_{k-1}(j+1) \dots t_{k-1}(n)$ są większe od a_k (bo ciąg a_i jest różnowartościowy). Teraz tak:

1. Wiemy, że elementów $t_{k-1}(1) \dots t_{k-1}(j)$ nie poprawimy elementem a_k, ponieważ są tam już podciągi odpowiedniej długości, a a_k jest większy od wszystkich tych elementów.
2. $t_{k-1}(j)$ mówi nam, że w prefiksie $a_1 \dots a_{k-1}$ jest podciąg rosnący o długości j zakończony elementem $t_{k-1}(j)$, a skoro $t_{k-1}(j) < a_k$ to na koniec tego podciągu możemy dodać a_k i otrzymamy podciąg długości $j+1$ zakończony elementem $a_k < t_{k-1}(j+1)$, czyli $t_{k-1}(j+1) = a_k$
3. Elementy $t_{k-1}(j+1) \dots t_{k-1}(n)$ są większe od a_k więc nie możemy przedłużyć tych podciągów za pomocą a_k, więc nie zmienimy elementów $t_{k-1}(j+2) \dots t_k(n)$

□

- Skoro zachodzą dwa powyższe warunki to możemy znajdować indeks do zmiany elementu w tablicy za pomocą wyszukiwania binarnego.

Skoro tak, to mamy algorytm:

Dane: Tablica liczb $a_1 \dots a_n$.
Wynik: Liczba s taka, że s jest rozmiarem najdłuższego podciągu rosnącego w $a_1 \dots a_k$.

Niech t - tablica rozmiaru n;
for $i \leftarrow 1$ **to** n **do**
  $t[i] \leftarrow \infty$;
end
for $i \leftarrow 1$ **to** n **do**
  $j \leftarrow BinarySearch(a_i, t)$;
  $\backslash\backslash BinarySearch(a_i, t)$ zwraca najmniejszy taki indeks j, że $a_i < t[j]$
  $t[j] \leftarrow a_i$;
end
$s \leftarrow 1$;
while $t[s] \neq \infty \ AND \ s < n$ **do**
  $s{+}{+}$;
end
return s

Algorytm 1: Długość LIS

Mamy pętlę przechodzącą n razy, a w każdej z nich dominującą operacją będzie wyszukiwanie binarne - $\log n$. Skoro tak, to złożoność czasowa naszego algorytmu to $O(n \log n)$. Przy złożoności pamięciowej potrzebujemy jednej tablicy t rozmiaru n, bo elementy możemy na bieżąco wrzucać do tablicy, co daje złożoność pamięciową $O(n)$.

Zadanie 10

Problem:
Dane : ciąg par liczb rzeczywistych $a_1 = (x_1, y_1), a_2 = (x_2, y_2), \dots a_n = (x_n, y_n)$ określający kolejne wierzchołki $n - kta$ wypukłego P
Szukane : Znaleźć zbiór S nieprzecinających się przekątnych, które dzielą P na trójkąty, taki, by długość najdłuższej przekątnej była możliwie najmniejsza.

Najpierw posortujmy wierzchołki tak, by odcinek pomiędzy kolejnymi punktami należał do wielokąta.

Utwórzmy sobie tablicę nxn w której element $t(i,j)$ oznacza minimalną najdłuższą przekątną możliwą do uzyskania z wielokąta $a_i \dots a_j$. Oczywiście, mając taki wielokąt możemy podzielić go na dwa wielokąty $a_i \dots a_k \ a_k \dots a_j$ tworząc dwa odcinki $a_i a_k$ oraz $a_k a_j$ które, wraz z odcinkiem $a_i a_j$ tworzą trójkąt o przekątnych $a_i a_k$ oraz $a_k a_j$. W ten sposób można łatwo dojść do związku rekurencyjnego:

$$t(i,j) = \begin{cases} 0 & dla\ j = i \wedge j = i+1 \\ min_{k=i+1,\dots,j-1}\left(max(|a_i,a_k|,|a_k,a_j|,t(i,k),t(k,j))\right) & \end{cases}$$

Wynik to oczywiście cały wielokąt, tj $t(1,n)$.