Zadanie 2
lista 6

INPUT: Ciąg $[a_1, a_2, ..., a_n]$
OUTPUT: Najdłuższy ciąg $a_{i_1}, a_{i_2}, ..., a_{i_k}$, taki że $1 \leqslant i_1 < i_2 < ... < i_k \leqslant n$ oraz $a_{i_1} < a_{i_2} < ... < a_{i_k}$.

1 Algorytm

W celu skonstruowania algorytmu znajdującego wyżej opisany podciąg wykorzystamy metodologię programowania dynamicznego.

Stwórzmy graf wszystkich dopuszczalnych przejść od jednego do drugiego elementu, tzn. ustalmy wierzchołek i dla każdego spośród a_i elementów (zbiór wierzchołków oznaczmy jako V), oraz dodajmy do niego krawędzie (i, j). Krawędź (i, j) będzie występowała, jeśli możliwe jest, aby a_i oraz a_j były następującymi po sobie elementami rosnącego podciągu, czyli gdy: $i < j$ oraz $a_i < a_j$ (zbiór krawędzi oznaczmy jako E). Dodatkowo przyjmiemy, że długość każdej krawędzi będzie równa 1.
Zauważmy, że wyżej opisany graf $\mathbf{G = (V, E, c)}$ ma następujące własności:

1) Jest skierowanym grafem acyklicznym (dag - gdyż wszystkie krawędzie (i, j) mają własność $i < j$)
2) Każdej ścieżce w tym grafie odpowiada podciąg rosnący
3) $\forall (u, v) \in E : c(u, v) = 1$

Zatem problem znalezienia najdłuższego rosnącego podciągu sprowadza się do problemu znalezienia najdłuższej ścieżki w wyżej opisanym grafie! Aby to zrobić zastosujemy poniższy algorytm:

for j = 1, 2,...,n:
  $L(j) = 1 + max\{L(i) : (i, j) \in E\}$
return $max\{L(i) : i \in \{1, 2, ..., n\}\}$

gdzie $L(j)$ - długość najdłuższej ścieżki (najdłuższego podciągu rosnącego) kończącej się na a_j (czyli na j - tym elemencie ciągu wejściowego). Zauważmy, że każda ścieżka prowadząca do wierzchołka j musi zawierać w sobie któregoś z jego poprzedników. Z tego powodu $L(j)$ wynosi 1 + maksymalna wartość L(i), gdzie i jest jego pewnym poprzednikiem (jedynkę dodajemy, gdyż chcemy zliczać wierzchołki, a nie krawędzie w ścieżce). Zauważmy, że potencjalnie każdy element ciągu $(a_1, ...a_n)$ może być ostatnim elementem najdłuższego podciągu rosnącego, stąd liczymy maksimum po wszystkich elementach ciągu.

Jednak otrzymana wartość L mówi nam tylko o tym , jaką długość ma najdłuższy z podciągów. Abyśmy byli w stanie "odtworzyć" podciąg, którego szukamy wystarczy zastosować następującą zmianę w algorytmie:

for j = 1, 2,...,n:
  $L(j) = 1 + max\{L(i) : (i, j) \in E\}$
  prev(j) = i // i jest indeksem poprzednika a_i elementu a_j, dla którego
    // wartość $L(i)$ jest największa spośród wartości $L(.)$
    // dla wszystkich poprzedników a_j
return $max\{L(i) : i \in [n]\}$

Zatem zapamiętując a_i, dla którego osiągany jest $max\{L(i) : (i, j) \in E\}$ dowiemy się, jakie dokładnie elementy występują w szukanym podciągu.

2 Złożoność obliczeniowa algorytmu

W powyższym algorytmie każdy z podproblemów rozwiązywany jest przez wyliczenie wartości:

$$L(j) = 1 + max\{L(i) : (i, j) \in E\}$$

wyrażenie wykorzystuje tylko "mniejsze" podproblemy. Wyznaczenie wyniku powyższego wyrażenia wymaga znajomości "poprzedników" wierzchołka j. Aby to zrobić należy wyznaczyć "reversed Graph" G^R (w G^R istnieje krawędź skierowana (v, u) wtedy i tylko wtedy, gdy w grafie G istnieje krawędź (u, v)). Zauważmy, że sąsiadami danego wierzchołka $t \in V$ w grafie G^R są "poprzednicy" wierzchołka t w grafie G. Wyliczenie "reversed graph" zajmuje czas $O(n^2)$, ponieważ:

- musimy przejść oryginalny graf G i "odwrócić" jego krawędzie, czyli: Jeśli graf G będziemy reprezentowali poprzez listy sąsiedztwa, aby stworzyć G^R (również reprezentowany w postaci list sąsiedztwa) wystarczy przeglą-

dać listy sąsiedztwa dla wierzchołków grafu G po kolei i umieszczać dany wierzchołek na listach sąsiedztwa wszystkich jego sąsiadów. Ponieważ każdy element każdej listy przetwarzamy jeden raz (wykonując stałą liczbę operacji), łącznie da nam to wyżej wspomniany koszt $O(n^2)$.
Zauważmy, że wtedy dla każdego $j = 1, 2, ..., n$ wykonujemy:

$$L(j) = 1 + max\{L(i) : (i, j) \in E\} = L(j) = 1 + max\{L(i) : (j, i) \in E'\}$$

gdzie E' jest zbiorem krawędzi w grafie G^R. Czyli tak naprawdę dla każdego wierzchołka j patrzymy na $L(i)$ dla wszystkich jego poprzedników i (wszystkie elementy na listach sąsiedztwa dla wierzchołka j w grafie G^R). Zatem wykonujemy łącznie pracę proporcjonalną do liczby krawędzi w grafie G, czyli $O(n^2)$. Następnie wyliczamy $max\{L(j) : j \in [n]\}$ co zajmuje czas $O(n)$. Warto zauważyć, że zapamiętywanie poprzedników, w celu "odtworzenia" szukanego podciągu nie zmienia asymptotycznej złożoności obliczeniowej. Dochodzimy więc do wniosku, iż złożoność tego algorytmu to $O(n^2)$.