Algorytmy i Struktury Danych 1

Notatki

„Narysujmy drzewo DFSu dla tego ptaka, który jest rybą."

Maciej Mikołajczak
Kraków, 2024/2025

Spis treści

1. Zestaw 1

2024-10-15

Algorytm 1. Dla danych dwóch słów długości n, m nad alfabetem $\{0, 1\}$ można wyznaczyć długość ich najdłuższego wspólnego podciągu w czasie $O\left(n^2 + m\right)$.

Dowód. Będziemy obliczać wartości `dp[n+1][n+1]`, gdzie `dp[a][b]` oznacza najmniejsze takie `k`, że `lcs(A[0:a],B[0:k]) = b` – jak mały prefix drugiego słowa jest potrzebny, by otrzymać najlepszy podciąg dla danego prefixu pierwszego słowa. Do tego będziemy trzymać tablicę `f[pos][c]`, w której trzymamy pierwsze wystąpienie znaku `c` po indeksie `pos` (włącznie). Tablicę `f` obliczamy w $O\left(2m\right)$.

Mamy warunki początkowe `dp[!= 0][0] = 0` oraz `dp[0][!= 0] =` $+\infty$. Krok ma postać `dp[a][b] = min(dp[a-1][b], f[dp[a-1][b-1]][A[a-1]]+1)` – dla danego indeksu w pierwszym słowie A możemy albo nie brać znaku pod nim, albo wziąć go i wtedy wystarczy nam znaleźć, gdzie jest pierwszy pasujący znak w drugim słowie po końcu krótszego optymalnego słowa. □

2. Zestaw 4

2024-10-29

Algorytm 2. Dane są dwa ciągi liczb całkowitych A oraz B o długościach odpowiednio n oraz m. Długość najdłuższego ciągu rosnącego, który jest podciągiem obu tych ciągów, wyznacza algorytm:

$dp[0][j] = dp[i][0] = 0$
for $i = 1 \dots n$ **do**
  $tmp = 0$
  for $j = 1 \dots m$ **do**
    if $A[i] \neq B[j]$ **then**
      $dp[i][j] = dp[i-1][j]$
      if $A[i] > B[j]$ **then**
        $tmp = \max\left(tmp, dp[i-1][j]\right)$
      end if
    else
      $dp[i][j] = tmp + 1$
    end if
  end for
end for

Dowód. Dynamik dp oznacza długość najdłuższego wspólnego podciągu rosnącego, który kończy się na zadany element B. Rozważając krok w dynamiku mamy dwie opcje: ostatni element A nie jest równy ostatniemu elementowi B i go ignorujemy albo jest równy. Wtedy bierzemy go i cofamy się do takiego wcześniejszego elementu B, który daje najdłuższy wspólny podciąg ze skróconym A i jest mniejszy od elementu $B[j]$. Wyznaczamy go w trakcie trwania algorytmu – jeśli rozważany element $A[i]$ był różny od elementu $B[j]$, to w momencie gdy był większy, to rozważana wartość z $B[j]$ będzie potencjalnym miejscem w ciągu B, do którego skoczymy, gdy $B[j]$ będzie równe $A[i]$ (rozważamy cały czas to samo i). □

3. Programowanie dynamiczne

2024-11-04

Algorytm 3 (Odległość edycyjna)**.** Mamy dane słowa A i B o długościach n i m. Dopuszczamy operacje: wstawienie dowolnego znaku w określone miejsce słowa, usunięcie znaku z dowolnego miejsca w słowie i zamiana znaku na dowolny inny. Minimalną liczbę operacji przekształcających A na B wyznacza algorytm:

$C[i][0] = i$
$C[0][j] = j$
for $i = 1 \dots n$ **do**
  for $j = 1 \dots m$ **do**
    $C[i][j] = \min\{C[i-1][j] + 1, C[i][j-1] + 1, C[i-1][j-1] + A[i] \neq B[j]\}$
  end for
end for

Działa on w czasie $O(nm)$ i takiej samej pamięci.

Dowód. Zawsze możemy wykonywać operacje w takiej kolejności, aby ostatnią operacją było: usunięcie ostatniego znaku (rozważanego prefiksu) A, dodanie na jego koniec nowego znaku lub zamiana ostatniego znaku na odpowiedni (jeśli nie są takie same). Zatem są to jedyne opcje, jakie trzeba rozważyć przechodząc do niższej instancji. □

Algorytm 4 (Hirschberga dla odległości edycyjnej)**.** Chcemy rozwiązać problem odległości edycyjnej z odtwarzaniem rozwiązania w czasie $O(nm)$ i pamięci $O(\min(n,m))$.

Mamy algorytm wyznaczający odległość edycyjną w pamięci liniowej: w normalnym algorytmie zauważamy, że wystarczy pamiętać tylko poprzedni i aktualny wiersz, starsze można zapominać. Daje to lepszą złożoność, ale nie można odtworzyć rozwiązania.

Dzielimy pierwsze słowo na części równej długości. Następnie wykonujemy przedstawiony wyżej algorytm na słowach $A[1\ldots\frac{n}{2}], B$ i $A[\frac{n}{2}+1,n]^R, B^R$ (R w wykładniku oznacza odwrócenie słowa). Dostajemy dwie tablice X, Y, jedna mówi ile potrzeba ruchów do zamiany pierwszej połowy A na prefiks B każdej możliwej długości, a druga to samo dla drugiej połowy A i sufiksów B. Teraz wyznaczamy takie k, że $X[k]+Y[m-k]$ jest minimalne. Znając k, wykonujemy cały algorytm rekurencyjnie na mniejszych słowach. Bazą są krótkie słowa, dla których można wykonać pełny algorytm odzyskiwania rozwiązania.

Dowód. Mamy $T(n,m) = T\left(\frac{n}{2},k\right) + T\left(\frac{n}{2},m-k\right) + nm$, co daje nam $T(n,m) = O(nm)$, czyli czas się zgadza. Rozbijając się na podzadania możemy korzystać z tej samej tablicy długości n (lub m, gdy jest mniejsze) i w niej trzymać wyniki obliczeń pomocniczych. Daje nam to złożoność pamięciową $O(\min(n,m))$. □

Algorytm 5 (Longest Increasing Subsequence)**.** Mając zadaną tablicę liczb $X[1..n]$ chcemy znaleźć najdłuższy podciąg rosnący jej elementów. Zrobimy to obliczając tablicę $Q[0..n]$, gdzie $Q[i]$ zawiera najmniejszy element, na jaki może się kończyć rosnący podciąg długości i. Początkowo $Q[0] = 0, Q[i] = \infty$. Działamy algorytmem

for $i = 1\ldots n$ **do**
  $ind = Q[i].\text{lowerbound}(X[i])$
  $Q[ind] = i$
  $P[i] = Q[ind-1]$
end for

który korzysta z faktu, że elementy w Q są rosnące. Zapisujemy w P mniejszy od $X[i]$ element, na który kończy się ciąg długości $ind-1$, bo jest to dobry poprzednik $X[i]$ w ciągu długości ind.

4. Algorytmy zachłanne

2024-11-10

Algorytm 6 (Kody Huffmanna)**.** Mamy zadany alfabet C wraz z częstotliwością występowania każdego znaku. Chcemy tak zakodować znaki z C za pomocą 0 i 1, żeby wartość oczekiwana długości zakodowanego ciągu była jak najmniejsza. Jednocześnie chcemy, żeby ciąg był jednoznacznie dekodowalny, czyli żaden kod nie był prefixem innego kodu. Łatwo zauważyć, że taki kod można reprezentować jako drzewo binarne, w którym znaki to liście a przejście w prawo oznacza 1 w kodzie. Budujemy drzewo za pomocą algorytmu:

$Q = C$
while $!Q.empty()$ **do**
  $left[z] = x = Q.pop()$
  $right[z] = y = Q.pop()$
  $freq[z] = freq[x] + freq[y]$
  $Q.push(z)$
end while

gdzie Q jest kolejką priorytetową a z oznacza nowo utworzony węzeł. Inaczej mówiąc, łączymy dwa węzły o najmniejszej częstotliwości w jeden.

Dowód. Zauważmy, że każdy kod jest reprezentowany przez jakieś drzewo binarne. Optymalny kod jest reprezentowany przez drzewo pełne (inaczej można wsadzić jakiś znak do pustego liścia o krótszym kodzie). Rozważmy dowolne optymalne rozwiązanie (drzewo) T. Jest to drzewo pełne, więc będą istniały dwa liście leżące na najdłuższej ścieżce w drzewie, które są swoim rodzeństwem. Zamieniając je ze znakami o najmniejszej częstotliwości x, y dostajemy rozwiązanie optymalne, w którym x, y mają wspólnego rodzica. Zauważmy, że jeśli T jest optymalnym drzewem, w którym x, y mają wspólnego rodzica, to T' powstałe przez zastąpienie x, y ich rodzicem z jest optymalne dla alfabetu $C \setminus \{x, y\} \cup \{z\}$ (gdyby tak nie było, to lepsze rozwiązanie dawałoby rozwiązanie lepsze od T dla C). Zatem można zredukować problem do mniejszej instancji i dostać poprawną odpowiedź z wykorzystaniem naszego algorytmu. □

5. Drzewa AVL

2024-11-10

Definicja 1. Drzewem AVL (Adelson-Velsky, Landis) nazywamy drzewo BST, w którym dla każdego wierzchołka v wysokość poddrzew o korzeniach left(v) i right(v) różni się o co najwyżej 1. Różnicę $b(v) = h(\text{right}(v)) - h(\text{left}(v))$ nazywamy współczynnikiem zrównoważenia (balansem).

Twierdzenie 1. Wysokość drzewa AVL o n wierzchołkach nie przekracza $\Theta(\log n)$.

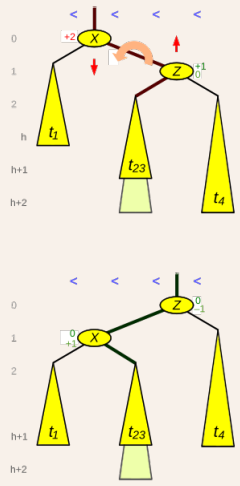
Dowód. Jeśli drzewo binarne ma n wierzchołków, to ma $n+1$ liści zewnętrznych (pustych wskaźników w wierzchołkach). Niech $N(h)$ oznacza najmniejszą możliwą liczbę liści zewnętrznych w drzewie AVL o wysokości h. Mamy $N(0) = 2$, $N(1) = 3$ oraz $N(h) = N(h-1) + N(h-2)$, bo minimalne drzewo AVL o wysokości h powstaje przez dołączenie do korzenia dwóch minimalnych drzew o wysokościach $h-1$ i $h-2$. Zatem $N(h) = F(h+3)$, gdzie $F(i)$ jest i-tą liczbą Fibonacciego. Ze wzoru Bineta $F(k) \geq \left(\frac{1+\sqrt{5}}{2}\right)^{k-2}$, a więc $h + 1 \leq \log N(h) = \log(n+1)$. □

Algorytm 7 (Rotacje). Podczas dokonywania zmian w drzewie AVL balans wierzchołków może zostać zaburzony. Po jednej operacji zaburzony wierzchołek może przyjąć wartości balansu 2 lub -2. Będziemy naprawiać takie sytuacje stosując odpowiednie rotacje wierzchołków. Niech X będzie najniższym wierzchołkiem o zaburzonym balansie i niech Z będzie jego dzieckiem o głębszym poddrzewie. Mamy cztery możliwości naruszenia warunku AVL:

1. Prawo prawe – Z jest prawym dzieckiem i $b(Z) \geq 0$
2. lewo lewe – Z jest lewym dzieckiem i $b(Z) \leq 0$
3. prawo lewe – Z jest prawym dzieckiem i $b(Z) < 0$
4. lewo prawe – Z jest lewym dzieckiem i $b(Z) > 0$

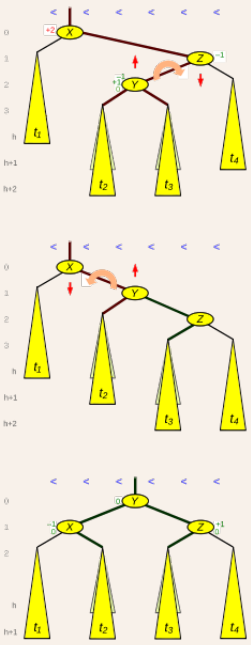
W pierwszych dwóch przypadkach stosujemy rotację prostą (odpowiednio w lewo i prawo), a w pozostałych rotację podwójną (prawo lewą i lewo prawą).

Algorytm 8 (Rotacja prosta). Mamy wierzchołek X, którego dziecko Z ma za duże poddrzewo. W przypadku, gdy większym poddrzewem Z jest jego zewnętrzne poddrzewo (dla prawego dziecka prawe, a dla lewego lewe), można zastosować rotację prostą. Zamieniamy w niej X i Z miejscami, a dzieckiem X którym wcześniej był Z staje się wewnętrzne poddrzewo Z. Po takiej zamianie nie zmienia się kolejność inorder wierzchołków, a więc dalej są dobrze uporządkowane (jak w drzewie BST).



Rysunek 1: Lewa rotacja prosta

Algorytm 9 (Rotacja podwójna)**.** Mamy wierzchołek X, którego dziecko Z ma za duże poddrzewo. W przypadku, gdy większym poddrzewem Z jest jego wewnętrzne poddrzewo Y, można zastosować rotację podwójną. Wykonujemy dwie rotacje proste – najpierw Z z Y, a potem X z Y.



Rysunek 2: Rotacja podwójna prawo lewa

Algorytm 10 (Operacje na drzewie AVL)**.** Wstawianie i usuwanie elementów z drzewa AVL realizujemy tak samo jak w zwykłym drzewie BST. Po wprowadzaniu zmiany musimy zadbać o to, aby drzewo dalej było zbalansowane. Idziemy do góry od wierzchołka, który został zmieniony i aktualizujemy odpowiednio wartości balansów. W przypadku powstania niezbalansowanego wierzchołka naprawiamy go odpowiednią rotacją.