Zadania domowe. Zestaw 1.1

Maciej Poleski

13 października 2012

1

W trakcie standardowego algorytmu `LIS` dla każdej pozycji i znamy długość najdłuższego podciągu rosnącego kończącego się w pozycji i. Skoro tak, to można puścić algorytm od dwóch stron (od początku ciągu i od końca) i dodać do siebie odpowiednie wyniki (-1 z powodu podwójnego policzenia pozycji i).

```
LIS(A):
    T[0] <- 0
    for i <- 1 to n]:
        T[i] <- ∞

    for k=0 to n):
        s <- lower-bound A[k] in T[]
        T[s] <- A[k]
        S[k] <- s
    return S[]

LEFT[] <- LIS(T[])
for i <- 0 to n):
    TR[i] <- -T[n-1-i]
RIGHT[] <- LIS(TR[])

for i <- 0 to n):
    ANS[i] <- LEFT[i] + RIGHT[n-1-i] - 1
```

Funkcja `lower-bound` (omówiona na MP) zwraca indeks pierwszego elementu w zadanym ciągu nie mniejszego niż zadana wartość. Odpowiedź jest zapisana w tablicy `ANS[0..n-1]`. Złożoność jest złożonością standardowego algorytmu `LIS`.

2

Analogicznie do LCS. Tablice (słowa) `A` i `B` indeksujemy od 1.

```
for each i: T[0][i] <- i
for each i: T[i][0] <- i

for i <- 1 to n]:
```

```
    for j <- 1 to n]:
        T[i][j] <- min{
            A[i]=B[j] ? T[i-1][j-1]+1 : ∞,
            T[i-1][j]+1,
            T[i][j-1]+1}

i <- n
j <- n

S <- make-stack()

while i != 0 and j != 0:
    if A[i]=B[j] and T[i][j]=T[i-1][j-1]+1:
        push A[i] to S
        i <- i-1
        j <- j-1
    else if T[i][j]=T[i-1][j]+1:
        push A[i] to S
        i <- i-1
    else:
        push B[j] to S
        j <- j-1

while S is not empty:
    print top of S
    pop from S
```

Działa dlatego że budując optymalnie słowo dla `A[1..i]` i `B[1..j]` możemy wykorzystać optymalne słowo dla `A[1..i-1]` i `B[1..j-1]` (jeżeli `A[i]=B[j]`) oraz optymalne słowo dla `A[1..i-1]` i `B[1..j]` oraz optymalne słowo dla `A[1..i]` i `B[1..j-1]`. W każdym z tych przypadków dodając jedną brakującą literę na koniec słowa. Potem odtwarzamy rozwiązanie analizując od końca który wybór był optymalny. Przy użyciu stosu odwracam odwrócone słowo.

Pierwsze dwie pętle są liniowe, potem pojawią się pętla zagnieżdżona podwójnie (kwadratowa) i dwie pętle wykonujące się tyle razy ile wynosi długość wynikowego słowa ($\leqslant 2n$ czyli liniowe). Wobec tego złożoność całości jest kwadratowa (względem długości słów).

3

Tablice (słowa) `A` i `B` indeksujemy od 1.

```
for each i: T[0][i] <- 0
for each i: T[i][0] <- 0

max <- 0
maxi <- 0
maxj <- 0

for i <- 1 to n]:
    for j <- 1 to n]:
        if A[i]!=B[j]:
            T[i][j] <- 0
        else:
            T[i][j]=T[i-1][j-1]+1
            if T[i][j]>max:
                maxi <- i
                maxj <- j

for i <- maxi-max+1 to maxi]:
    print A[i]
```

Analizujemy wszystkie pary prefixów i dla każdej z nich wyznaczamy najdłuższy wspólny sufix. Są tylko dwie możliwości: albo `A[i]!=B[j]` - wtedy najdłuższy wspólny sufix prefixów `A[1..i]` i `B[1..j]` ma długość 0, albo `A[i]=B[j]` wtedy najdłuższy wspólny sufix prefixów `A[1..i]` i `B[1..j]` ma długość o 1 większą od najdłuższego wspólnego sufixu prefixów `A[1..i-1]` i `B[1..j-1]` a jego ostatnią literą jest `A[i]`. Pozostaje zapamiętać który sufix jest najdłuższy i go wypisać.

Złożoność pierwszych dwóch pętli jest liniowa, potem pętla zagnieżdżona - kwadrat i w końcu wypisywanie wyniku - nie dłuższego niż n. W efekcie złożoność całości jest kwadratowa (względem długości słów).