Subsecuencia contigua creciente más larga

Leonardo Flórez-Valencia

2014-2019

1. Descripción y formalización del problema

El problema, informalmente, se define como: encontrar el sub-arreglo creciente más largo de un arreglo.

Así, para el arreglo $S = [9, 7, 7, 1, 2, 3, 4, 6, 5]$ el sub-arreglo más largo sería $LICS = [1, 2, 3, 4, 6]$.

Formalmente, se dice que: Dada una secuencia S de elementos $a_i \in \mathbb{T}$, donde se define la relación de orden total $<$, informar la secuencia $L \in S$ donde los elementos contiguos cumplan la relación de orden total $\leq$. Ahora, la definición del contrato sería:

- **Entradas**: Una secuencia S de n números: $S = \langle a_1, a_2, \cdots, a_n \rangle$ donde $a_i \in \mathbb{T}$ y en $\mathbb{T}$ está definida la relación de orden total $\leq$.
- **Salidas**: Una subsecuencia $L = \langle a_i, \cdots, a_j \rangle \mid a_i \leq a_{i+1} \leq \cdots \leq a_{j-1} \wedge L \in S$

2. Fuerza bruta

Algorithm 1 Algoritmo de LICS fuerza bruta.

```
procedure LICS(S)
    maxS ← −∞
    maxI ← 0
    maxJ ← 0
    for i ← 1 to |S| do
        for j ← i to |S| do
            isOrdered ← True
            for k ← i + 1 to j do
                if S[k] ≤ S[k − 1] then
                    isOrdered ← False
                end if
            end for
            if isOrdered ∧ maxS < (j − i) then
                maxS ← j − i
                maxI ← i
                maxJ ← j
            end if
        end for
    end for
    return [maxI, maxJ]
end procedure
```

Este algoritmo presenta una cota superior de $O(n^3)$, calculada por simple inspección de código.

3. Dividir y vencer

Algorithm 2 Algoritmo de LICS dividir y vencer.

```
procedure LICSAux(S, b, e)
    if b = e then
        return [b, b]
    else if e < b then
        return [0, −1]
    else
        q ← ⌊(b + e) ÷ 2⌋
        [L_i, L_j] ← LICSAux (S, b, q − 1)
        [R_i, R_j] ← LICSAux (S, q + 1, e)
        [C_i, C_j] ← LICSPivot (S, b, q, e)
        if (L_j − L_i) > (R_j − R_i) ∧ (L_j − L_i) > (C_j − C_i) then
            return [L_i, L_j]
        else if (R_j − R_i) > (L_j − L_i) ∧ (R_j − R_i) > (C_j − C_i) then
            return [R_i, R_j]
        else
            return [C_i, C_j]
        end if
    end if
end procedure
```

Algorithm 3 Algoritmo de LICS dividir y vencer: función del pivote.

```
procedure LICSPivot(S, b, q, e)
    i ← q
    while b < i ∧ S[i − 1] ≤ S[i] do
        i ← i − 1
    end while
    j ← q
    while j < e ∧ S[j] ≤ S[j + 1] do
        j ← j + 1
    end while
    return [i, j]
end procedure
```

Algorithm 4 Algoritmo de LICS dividir y vencer: función principal.

```
procedure LICS(S)
    return LICSAux(S, 1, |S|)
end procedure
```

4. Invariantes

El algoritmo de "fuerza bruta" tiene como invariante:
El algoritmo "dividir-y-vencer" tiene como invariante:

5. Análisis de complejidad

En el caso del algoritmo de "fuerza bruta", resulta evidente por inspección de código que su orden de complejidad es $O(n^3)$.

Para la versión "dividir-y-vencer", se tiene la ecuación de recurrencia:

$$T(n) = \begin{cases} O(1) & ; \quad b \geq e \\ 2T\left(\frac{n}{2}\right) + O(n) & ; \quad b < e \end{cases} ; \tag{1}$$

que tiene un orden de complejidad $\Theta(n \log n)$, después de usar el teorema maestro.