Problema de la «sub-secuencia homogénea» y de «buscar valor»

Nicolás Camacho-Plazas

18 de agosto de 2020

Resumen

En este documento se presentan los problemas: encontrar la sub-secuencia homogénea contigua más larga y buscar un valor en una secuencia. Para cada uno se presentan dos algoritmos, una solución inocente y una que utiliza el paradigma dividir-y-vencer, además de su análisis, diseño, seudo-código y análisis tanto del invariante como de su complejidad.

Parte I
Sub-secuencia homogénea

1. Análisis

El problema, informalmente, se puede describir como: buscar, en una secuencia de elementos, en donde existe la mayor cantidad de repeticiones contiguas de un elemento (entiéndase un elemento como un número o letra). Se habla de secuencias de elementos como:

$$S = \langle s_1, s_2, \cdots, s_n \rangle = \langle s_i \in \mathbb{T} \rangle$$

donde n es la cardinalidad (i.e. cantidad de elementos) de la secuencia e i es el índice de cada elemento (note que el primer índice es 1 y no 0).

Para evitar ambigüedades, vamos a definir a T como un conjunto de elementos comparables, es decir, elementos con los cuales se puede utilizar el operador de comparación que expresa la relación de homogeneidad entre los datos comparados. Podría entonces establecerse que:

- a y b son homogéneos, si y solo si, $a = b$

De modo que la sub-secuencia homogénea contigua más larga está definida por:

$$S'_j = \langle s_1, s_2, \cdots, s_{n'} \rangle = \langle s_i \in S \wedge s_i = s_{i+1} \forall s \rangle$$

2. Diseño

Con las observaciones presentadas en el análisis anterior, podemos escribir el diseño de un algoritmo que solucione el problema de encontrar la sub-secuencia homogénea contigua más larga. El «contrato» de los algoritmos que solucionen el problema está dado por las siguientes condiciones:

Definición. Entradas:

1. Una secuencia $S = \langle s_1, s_2, \cdots, s_n \rangle = \langle s_i \in \mathbb{T} \rangle$ de $n \in \mathbb{N}$ elementos que pertenecen a un conjunto $\mathbb{T}$. En este conjunto debe estar definida la relación de homogeneidad o equivalencia (representada en varios lenguajes de programación mediante el operador de comparación ==).

Definición. Salidas:

1. Una sub-secuencia $S'_j = \langle s_1, s_2, \cdots, s_{n'} \rangle = \langle s_i \in S \land s_i = s_{i+1} \forall s \rangle$ de la secuencia de entrada S.

3. Algoritmos

3.1. Inocente

Este algoritmo es la solución que se le ocurrió más rápido al autor y detrás de la cual no existen esfuerzos por disminuir su complejidad (por esto el nombre de Ïnocente").

Algorithm 1 Comparación de longitudes

```
procedure SimpleMaxSubArray(S)
    n ← 0
    nAux ← 1
    let S′ ∧ S′Aux be new arrays
    for j ← 2 to |S| do
        if S[j] = S[j − 1] then
            if nAux == 1 then
                S′Aux[nAux] = S[j − 1]
                S′Aux[nAux + 1] = S[j]
                nAux = nAux + 1
            else
                nAux = nAux + 1
                S′Aux[nAux]
            end if
        else
            if nAux ≥ n then
                copy S′Aux into S′
                n = nAux
            end if
            empty S′Aux
            nAux = 1
        end if
    end for
    return S′
end procedure
```

3.1.1. Complejidad

Tras realizar inspección de código, basado en el único for, se concluyó que el algoritmo tiene un orden de complejidad de: $O(|S|)$.

3.1.2. Invariante

- Inicio: Se inicializan las variables n, $nAux$ determinando que aún no se han registrado invariantes. Además, se declaran dos secuencias en donde se almacenarán el mayor subarreglo (S') y el subarreglo auxiliar ($S'Aux$).

- Avance: En el for se verifica la homogeneidad contigua de cada elemento ($S[j] = S[j-1]$), de modo que en cada iteración en la que se cumpla con dicha característica, se actualiza el número de coincidencias. Cuando dicha propiedad no se cumple, entonces:

    - Si $nAux \geq n$, entonces se almacenan los elementos de $S'Aux$ en S' y se actualiza el tamaño de el presunto máximo sub-arreglo, es decir: $n = nAux$.

- En caso contrario se descartan los datos auxiliares para estudiar el posible nuevo subarreglo.

- Terminación: se retorna el máximo subarreglo S' después de que se recorriera S por completo.

3.2. Dividir-y-vencer

Este algoritmo se basa en el principio de dividir y vencer, y el algoritmo de encontrar el máximo subarreglo adaptado para que en lugar de sumar las derivadas discretas, tenga como criterio el número de repeticiones contiguas de un elemento. Primero está el método que adapta el retorno para el usuario:

Algorithm 2 Adaptador para el usuario

$start, end, size = FindMaxHomogeneousSubarray(S)$
return $S[start..end]$

El anterior algoritmo necesita de $FindMaxHomogeneousSubarray$ descrito a continuación:

Algorithm 3 Encontrar el máximo sub-arreglo contiguo

procedure FINDMAXHOMOGENEOUSSUBARRAY(S)
    if $high \leq low$ **then**
        return $(low, high, 1)$
    else
        $mid = \lfloor (low + high)/2 \rfloor$
        $(leftlow, lefthigh, leftsum) = FindMaxHomogeneousSubarray(S, low, mid)$
        $(rightlow, righthigh, rightsum) = FindMaxHomogeneousSubarray(S, mid+1, high)$
        $(crosslow, crosshigh, crossSum) = FindMaxCrossHomogeneousSubarray(S, low, mid, high)$
        if $leftsum \geq rightsum \wedge leftsum \geq crossSum$ **then**
            return $(leftlow, lefthigh, leftsum)$
        else if $rightsum \geq leftsum \wedge rightsum \geq crossSum$ **then**
            return $(rightlow, righthigh, rightsum)$
        else
            return $(crosslow, crosshigh, crossSum)$
        end if
    end if
end procedure

Este algoritmo necesita de dos algoritmos auxiliares:

- FindMaxCrossHomogeneousSubarray, que encuentra el subarreglo que pasa por el pivote, es decir, el punto medio que divide a la secuencia original en dos subsecciones.

Ahora se describirá el algoritmo de FindMaxCrossHomogeneousSubarray:

Algorithm 4 Encontrar el máximo sub-arreglo contiguo cruzado

```
procedure FINDMAXCROSSHOMOGENEOUSSUBARRAY(S)
    leftsum = −∞
    sum = 0
    for i ← mid downto low do
        if S[i] = S[i + 1] then
            sum = sum + 1
        else
            sum = 0
            pass
        end if
        if sum ≥ leftsum then
            leftsum = sum
            maxleft = i
        end if
    end for
    for i ← mid to high do
        if S[i] = S[i − 1] then
            sum = sum + 1
        else
            sum = 0
            pass
        end if
        if sum ≥ rightsum then
            rightsum = sum
            maxright = i
        end if
    end for
    return (maxleft, maxright, leftsum + rightsum)
end procedure
```

3.2.1. Complejidad

- $FindMaxCrossHomogeneousSubarray$: Tras realizar inspección de código se determinó que el algoritmo tiene un orden de complejidad de: $O\left(|S|\right)$ por los dos ciclos no anidados.

- $FindMaxHomogeneousSubarray$: Al ser un método dividir-y-vencer, y según el Teorema maestro se establece que:

$$T(n) = \begin{cases} O(0), & \text{caso base,} \\ 2T(\frac{n}{2}) + O(n), & \text{si } n \in |S| \end{cases}$$

De modo que al utilizar el "Teorema maestro"se concluye que el algoritmo concuerda con el caso 2 de modo que se determina que:

$$T(n) \in \theta(nlog_2 n)$$

3.2.2. Invariante

En cada iteración for, la variable sum guarda el número de elementos contiguos homogéneos de cada subsección (bien sea del pivote mid hasta low o del pivote mid a $high$). Al finalizar los for, $leftsum$ tiene el mayor número de elementos contiguos y $maxleft$ su posición en la sección comprendida entre low y mid ; $rightsum$ y $maxright$ el de la sección comprendida entre mid y $high$. Lo anterior se garantiza porque cuando:

- $sum \geq leftsum \vee rightsun$, se guarda su valor e índice.
- En el caso contrario, se reinicia el conteo y se ignora el índice.

Parte II
Buscar valor

4. Análisis

El problema, informalmente, se puede describir como: buscar un valor en una secuencia de elementos. Se habla de secuencias de elementos como:

$$S = \langle s_1, s_2, \cdots, s_n \rangle = \langle s_i \in \mathbb{Z} \rangle$$

donde n es la cardinalidad (i.e. cantidad de elementos) de la secuencia e i es el índice de cada elemento (note que el primer índice es 1 y no 0).

Para evitar ambigüedades, vamos a definir a Z como un conjunto de elementos ordenables, es decir, elementos que cumplan relación de orden parcial. Podría entonces establecerse los elementos o valores de la secuencia deben cumplir las siguientes premisas:

- Reflexividad: $a \leq a$.
- Antisimetría: si $a \leq b$ y $b \leq a$, entonces $a = b$.
- Transitividad: si $a \leq b$ y $b \leq c$, entonces $a \leq c$.

5. Diseño

Con las observaciones presentadas en el análisis anterior, podemos escribir el diseño de un algoritmo que solucione el problema de encontrar un valor en una secuencia. El «contrato» de los algoritmos que solucionen el problema está dado por las siguientes condiciones:

Definición. Entradas:

1. Una secuencia $S = \langle s_1, s_2, \cdots, s_n \rangle = \langle s_i \in \mathbb{Z} \rangle$ de $n \in \mathbb{N}$ elementos que pertenecen a un conjunto $\mathbb{Z}$. En este conjunto debe estar definida la relación $\leq$.

2. El valor $v \in S$ que se pretende buscar.

Definición. Salidas:

1. Un indice j que implica la posición del valor v en S de modo que $v = s_i$ en donde $s_i \in S \wedge 1 \leq i \leq n \wedge n = |S|$. Además j será -1 si no se encontró v en la secuencia S.

6. Algoritmos

6.1. Inocente

Este algoritmo es la solución que se le ocurrió más rápido al autor y detrás de la cual no existen esfuerzos por disminuir su complejidad (por esto el nombre de Ïnocente").

Algorithm 5 Recorrer y comparar

```
procedure SimpleIndexOf(S,v)
    j = −1
    for i ← 1 to |S| do
        if S[i] = v then
            j = i
            break
        end if
    end for
    return j
end procedure
```

6.1.1. Complejidad

Al realizar la inspección de código es evidente determinar que tiene un orden de complejidad de $O(|S|)$, dado que solo hay un For, no existe recursividad y no se usan algoritmos auxiliares.

6.1.2. Invariante

- Inicio: Se inicializa $j = -1$ para indicar que no se ha encontrado el valor.

- Avance: si $S[i] = v$ se modifica el valor de j para indicar su indice y termine, itere de lo contrario.
- Terminación: si al retornar j, su valor no cambió, significa que no se encontraron coincidencias, en caso contrario implicará que j tiene el indice de el valor v en S.

6.2. Dividir-y-vencer

Este algoritmo se basa en el uso de las ideas *QuickSort* relacionadas a la elección de un pivote aleatorio y del orden parcial en un lado del pivote.

Algorithm 6 Búsqueda por ordenamiento basado en pivotes aleatorios

```
procedure Q(u)ickSortSearchS, v
    return QuickSortSearchAux(S, 0, len(S) − 1, v)
end procedure
```

El anterior algoritmo organiza los parámetros que requiere *QuickSortSearchAux* descrito a continuación:

Algorithm 7 Lógica recursiva

```
procedure QuickSortSearchAux(S, p, r, v)
    if p<r then
        q = RandomizedPartition(S, p, r, v)
        if S[q] = v then
            return q
        else if S[q] > v then
            return QuickSortSearchAux(S, p, q − 1, v)
        else
            return QuickSortSearchAux(S, q + 1, r, v)
        end if
    else
        return S[r]
    end if
end procedure
```

El anterior algoritmo requiere de *RandomizedPartition* que elige de forma aleatoria un pivote y sobre el ordena parcialmente una sección del arreglo.

Algorithm 8 Elección pivote aleatorio

```
procedure RandomizedPartition(S,p,r,v)
    Let i be a random number between p and r
    Swap(S[r], S[i])
    return Partition(S, p, r, v)
end procedure
```

Para lograr el ordenamiento parcial, el anterior algoritmo se apoya en *Partition* descrito a continuación.

Algorithm 9 Ordenamiento parcial

```
procedure PARTITION(S,p,r,v)
    x = S[r]
    i = p − 1
    for j ← p to r do
        if thenS[j] ≤ x
            i = i + 1
            Swap(S[i], S[j])
        end if
    end for
    Swap(S[i + 1], S[r])
    return i + 1
end procedure
```

6.2.1. Complejidad

Para analizar la complejidad de *QuickSortSearch* es necesario tener en cuenta:

- El orden de complejidad de *Partition*, encontrado por inspección de código, es $O\left(|S|\right)$.
- Por lo anterior, el orden de complejidad de *RandomizedPartition* es $O\left(|S|\right)$. Cabe resaltar que la generación de un número aleatorio es $O\left(|1|\right)$
- De modo que, para la complejidad de *QuickSortSearchAux*, teniendo en cuenta la cantidad de llamados recurrentes (1), el número de divisiones (2) y la complejidad afuera de la recursión ($O\left(|S|\right)$), se concluye que:

  $$T(n) = \begin{cases} O(0), & \text{caso base,} \\ T(\frac{n}{2}) + O(n), & \text{si } n \in |S| \end{cases}$$

  De modo que, por el Teorema maestro, se determina a el orden de complejidad total como $\theta(n)$.

6.2.2. Invariante

Para encontrar el invariante se analiza *Partition*. En la posición q queda el pivote aleatorio y todos los elementos a su izquierda y derecha, están parcialmente ordenados. Eventualmente, a medida que se generan particiones, en *QuickSortSearchAux* la variable q tendrá el índice de el valor buscado ($S[q] = v$) y por lo tanto se retornará.