

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

Nicolás Camacho-Plazas

17 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, pseudo-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:

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

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:

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

## 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  $S = \langle s_1, s_2, \dots, s_n \rangle = \langle s_i \in \mathbb{T} \rangle$  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  $S'_j = \langle s_1, s_2, \dots, s_{n'} \rangle = \langle s_i \in S \wedge s_i = s_{i+1} \forall s \rangle$  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 "Inocente").

---

**Algorithm 1** Comparación de longitudes

---

```
1: procedure SIMPLEMAXSUBARRAY( $S$ )
2:    $n \leftarrow 0$ 
3:    $nAux \leftarrow 1$ 
4:   let  $S' \wedge S' Aux$  be new arrays
5:   for  $j \leftarrow 2$  to  $|S|$  do
6:     if  $S[j] = S[j - 1]$  then
7:       if  $nAux == 1$  then
8:          $S' Aux[nAux] = S[j - 1]$ 
9:          $S' Aux[nAux + 1] = S[j]$ 
10:         $nAux = nAux + 1$ 
11:       else
12:          $nAux = nAux + 1$ 
13:          $S' Aux[nAux]$ 
14:       end if
15:     else
16:       if  $nAux \geq n$  then
17:         copy  $S' Aux$  into  $S'$ 
18:          $n = nAux$ 
19:       end if
20:       empty  $S' Aux$ 
21:        $nAux = 1$ 
22:     end if
23:   end for
24:   return  $S'$ 
25: 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

---

```

1: start, end, size = FindMaxHomogeneousSubarray(S)
2: return  $S[start..End]$ 

```

---

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

---

**Algorithm 3** Encontrar el máximo sub-arreglo contiguo

---

```

1: procedure FINDMAXHOMOGENEOUSSUBARRAY(S)
2:   if high ≤ low then
3:     return (low, high, 1)
4:   else
5:     mid =  $\lfloor (low + high)/2 \rfloor$ 
6:     (leftlow, lefthigh, leftsum) = FindMaxHomogeneousSubarray(S, low, mid)
7:     (rightlow, righthigh, rightsum) = FindMaxHomogeneousSubarray(S, mid +
8:       1, high)
9:     (crosslow, crosshigh, crossSum) = FindMaxCrossHomogeneousSubarray(S, low, mid, high)
10:    if leftsum ≥ rightsum ∧ leftsum ≥ crossSum then
11:      return (leftlow, lefthigh, leftsum)
12:    else if rightsum ≥ leftsum ∧ rightsum ≥ crossSum then
13:      return (rightlow, righthigh, rightsum)
14:    else
15:      return (crosslow, crosshigh, crossSum)
16:    end if
17:  end if
18: 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

---

```

1: procedure FINDMAXCROSSHOMOGENEOUSSUBARRAY( $S$ )
2:    $leftsum = -\infty$ 
3:    $sum = 0$ 
4:   for  $i \leftarrow mid$  downto  $low$  do
5:     if  $S[i] = S[i + 1]$  then
6:        $sum = sum + 1$ 
7:     else
8:        $sum = 0$ 
9:       pass
10:    end if
11:    if  $sum \geq leftsum$  then
12:       $leftsum = sum$ 
13:       $maxleft = i$ 
14:    end if
15:  end for
16:  for  $i \leftarrow mid$  to  $high$  do
17:    if  $S[i] = S[i - 1]$  then
18:       $sum = sum + 1$ 
19:    else
20:       $sum = 0$ 
21:      pass
22:    end if
23:    if  $sum \geq rightsum$  then
24:       $rightsum = sum$ 
25:       $maxright = i$ 
26:    end if
27:  end for
28:  return ( $maxleft, maxright, leftsum + rightsum$ )
29: end procedure

```

---

### 3.2.1. Complejidad

- *FindMaxCrossHomogeneousSubarray*: Tras realizar inspección de código se determinó que el algoritmo tiene un orden de complejidad de:  $O(|S|)$  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

concuerta con el caso 2 de modo que se determina que:

$$T(n) \in \theta(n \log_2 n)$$

### 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:

- $sum \geq leftsum \vee rightsum$ , 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:

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

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  $S = \langle s_1, s_2, \dots, s_n \rangle = \langle s_i \in \mathbb{Z} \rangle$  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 índice  $j$  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 "Inocente").

---

**Algorithm 5** Recorrer y comparar

---

```
1: procedure SIMPLEINDEXOF( $S, v$ )
2:    $j = -1$ 
3:   for  $i \leftarrow 1$  to  $|S|$  do
4:     if  $S[i] = v$  then
5:        $j = i$ 
6:       break
7:     end if
8:   end for
9:   return  $j$ 
10: 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 índice 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 índice 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

---

```

1: procedure Q(u)ickSortSearchS, v
2:   return QuickSortSearchAux( $S, 0, \text{len}(S) - 1, v$ )
3: end procedure

```

---

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

---

**Algorithm 7** Lógica recursiva

---

```

1: procedure QUICKSORTSEARCHAUX( $S, p, r, v$ )
2:   if  $p < r$  then
3:      $q = \text{RandomizedPartition}(S, p, r, v)$ 
4:     if  $S[q] = v$  then
5:       return  $q$ 
6:     else if  $S[q] > v$  then
7:       return QuickSortSearchAux( $S, p, q - 1, v$ )
8:     else
9:       return QuickSortSearchAux( $S, q + 1, r, v$ )
10:    end if
11:  else
12:    return  $S[r]$ 
13:  end if
14: 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

---

```

1: procedure RANDOMIZEDPARTITION( $S, p, r, v$ )
2:   Let  $i$  be a random number between  $p$  and  $r$ 
3:   Swap( $S[r], S[i]$ )
4:   return Partition( $S, p, r, v$ )
5: end procedure

```

---



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

---

**Algorithm 9** Ordenamiento parcial

---

```

1: procedure PARTITION( $S, p, r, v$ )
2:    $x = S[r]$ 
3:    $i = p - 1$ 
4:   for  $j \leftarrow p$  to  $r$  do
5:     if  $S[j] \leq x$ 
6:        $i = i + 1$ 
7:        $Swap(S[i], S[j])$ 
8:     end if
9:   end for
10:   $Swap(S[i + 1], S[r])$ 
11:  return  $i + 1$ 
12: 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  $O(|S|)$ .
- Por lo anterior, el orden de complejidad de *RandomizedPartition* es  $O(|S|)$ .
- 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 ( $O(|S|)$ ), 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.