

# Algoritmos y Estructura de Datos 2

## Recuperatorios

X8

Alumno: Leandro Carreira

LU: 669/18

---

Link a documento online (en caso que algún caracter se haya pasado mal a .pdf):

<https://docs.google.com/document/d/1YTTKsL5jPRnfuZErhZuG9wHYfTRbwfUTkjYPc99aJyU/edit?usp=sharing>

# Ejercicio X8

## 8. Ejercicio X8 — Dividir y Conquistar

Se tiene un arreglo de palabras de longitud **acotada por una constante** y se desea saber cuántas veces es posible leer el nombre de la ciudad **mar del plata** de izquierda a derecha en este arreglo, esto es, alguna manera de encontrar **mar**, en alguna posición posterior **del** y en alguna posición posterior a ésta **plata**, no necesariamente consecutivas. Si hiciera falta puede asumirse que  $n$  es una potencia de algún natural mayor que 1.

Ejemplos: en [del, mar, del, del, mar, plata, tuyú] se puede leer 2 veces, mientras que en [mar, del, playa, mar, plata, mar, tuyú, mar, del, arcoiris, mar, plata] se puede leer 6 veces.

Usando la técnica de Dividir y Conquistar, escribir un algoritmo que, dado un arreglo de  $n$  palabras cualesquiera, devuelva esta cantidad en tiempo estrictamente mejor que  $O(n^2)$  (preferentemente  $O(n)$ ).

Se pide el algoritmo en pseudocódigo, explicar con palabras las ideas volcadas en el algoritmo y justificar su complejidad temporal.

### Observaciones

Uso `//` como operador "**división entera**", que redondea **hacia abajo**.

Para los for loop uso que  $i \text{ in } 0 \text{ to } 4 \equiv i \text{ in } [0, 1, 2, 3]$  (o sea, no inclusive el limite derecho)

En el análisis de complejidad, uso **n** en casos donde debería ser un valor menor dado por el **rango** del arreglo sobre el que estoy operando. En algunos casos lo aclaro como comentario, pero en otros uso **n** directamente (ya que es la cota que nos interesa).

**Implementación:** He implementado la función `contarMDPsEnElCentro()` en **python** para que pueda ser debuggeada o analizada en detalle en caso de que no sea del todo claro el pseudocódigo:

Link a Colab: <https://colab.research.google.com/drive/1FxRtwWGTQaQ-LZT0RprCoPyo1J61FBb0?usp=sharing>

*// Primera llamada al algoritmo*

`contarMDPs(A) → int res`

`int res ← contarMDPsRec(A, 0, Tam(A))`

$T(\text{Tam}(A))$

`return res`

*// Algoritmo recursivo.*

*// En los casos base se cuentan los "mar del plata"s completos a*

*// izquierda o a derecha del centro.*

*// Si no es un caso base, se hace recursión sobre las mitades y se analiza el*

*// "centro" (ambas mitades) por separado*

`contarMDPsRec(A, i, j) → int`

*// Cantidad de elementos sobre los que opero*

`int n ← j - i`

$\theta(1)$

*// Casos base: Asumo  $n > 0$*

`if n < 3 then`

$\theta(1)$

`return 0`

$\theta(1)$

```

else if n = 3 then                                 $\theta(1)$ 
    if A[0] = "mar" and A[1] = "del" and A[2] = "plata" then     $\theta(1)$ 
        return 1                                                 $\theta(1)$ 
    else
        return 0                                                 $\theta(1)$ 
    end if
end if

mitad ← (i + j) // 2                                 $\theta(1)$ 
cant_izq ← contarMDPsRec(A, i, mitad)                 $T(n/2)$ 
cant_der ← contarMDPsRec(A, mitad, j)                 $T(n/2)$ 
cant_cen ← contarMDPsEnElCentro(A, mitad, i, j)       $\theta(n)$ 

return (cant_izq + cant_der + cant_cen)               $\theta(1)$ 

```

*// El problema de contar "mar del plata"s en un arreglo es muy similar  
// al problema de contar subsecuencias en una secuencia, con la diferencia que aquí  
// los elementos a comparar son strings (acotados) y la secuencia sobre la cual buscar,  
// un arreglo de strings.*

*// Para resolverlo, uso un contador que será un arreglo de arreglos de enteros  
// que iré llenando a medida que recorro A, de forma de contar palabras consecutivas  
// Como no quiero contar dos veces lo ya contado, solo cuento los casos donde  
// "mar del plata" quedó dividida por la mitad del arreglo*

```

contarMDPsEnElCentro(A, mitad, i, j)
    int n ← Tam(A)                                 $\theta(1)$ 
    int rango ← j - i // rango = n si i,j abarcan todo A
    // Si A[i:j] es un arreglo vacío o muy chico, devuelvo 0
    if n < 3 or rango < 3 then                     $\theta(1)$ 
        return 0                                   $\theta(1)$ 
    end if

```

*// Creo arreglos con elementos a buscar para evitar más índices  
// en la función auxiliar buscarSubsecu() y que sea más clara*

```

arreglo(string) dosPal ← crearArreglo(2)           $\theta(1)$ 
arreglo(string) unaPal ← crearArreglo(1)           $\theta(1)$ 
dosPal[0] ← "mar"                                   $\theta(1)$ 
dosPal[1] ← "del"                                   $\theta(1)$ 
unaPal[0] ← "plata"                                 $\theta(1)$ 

```

*// Cuento ocurrencias separando en casos  
// Caso 1: "mar", "del" del lado izq, "plata" del derecho*

```

int MD_izq ← buscarSubsecu(A, desde=izq, hasta=mitad, subsecu=dosPal)

```

$\theta(\text{rango}/2) \equiv \theta(n)$

```
// Recorro rango de A contando palabras coincidentes
// (Estoy buscando en una MITAD del array original)
for pal in (desde + 1) to (hasta + 1) do            $\Theta(\text{rango})$  //  $\Theta(n)$  si es todo A
    for m in 1 to (k + 1) do                        $O(1)$ 
        // Veo que sea alguna de las palabras
        if A[pal - 1] = subsecu[m - 1] do          $O(1)$  // Pues string acotados
            // Cuento palabra y arrastro contador previo
            cont[pal-desde][m]  $\leftarrow$  cont[pal-desde - 1][m - 1] + \
                                     + cont[pal-desde - 1][m]            $O(1)$ 
        else
            // Solo arrastro contador previo
            cont[pal-desde][m]  $\leftarrow$  cont[pal-desde - 1][m]            $O(1)$ 
```

```

        end if
    end for
end for

// Devuelvo ultimo elemento de ultimo sub-arreglo, que lleva la cuenta
// acumulada de todas las sub-secuencias de strings encontradas
return cont[rango][k]

```

### Complejidad del algoritmo

$$T(n) = \begin{cases} \Theta(1) & n \leq 3 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{Caso contrario} \end{cases}$$

### Usando Teorema Maestro

Donde

$a = 2$  : Cantidad de subproblemas  
 $c = 2$  : Cantidad de particiones, con  $n/c$  el tamaño de los subproblemas  
 $f(n) = \Theta(n)$  : Función de costo dada por **contarMDPsEnElCentro()**

Veamos que el costo de la recursión tendrá el mismo “peso” que tiene  $f(n)$  en el cálculo de la complejidad, por lo que estaremos en el caso 2 del Teorema donde:

$$f(n) \in \Theta(n^{\log_c(a)})$$

reemplazando

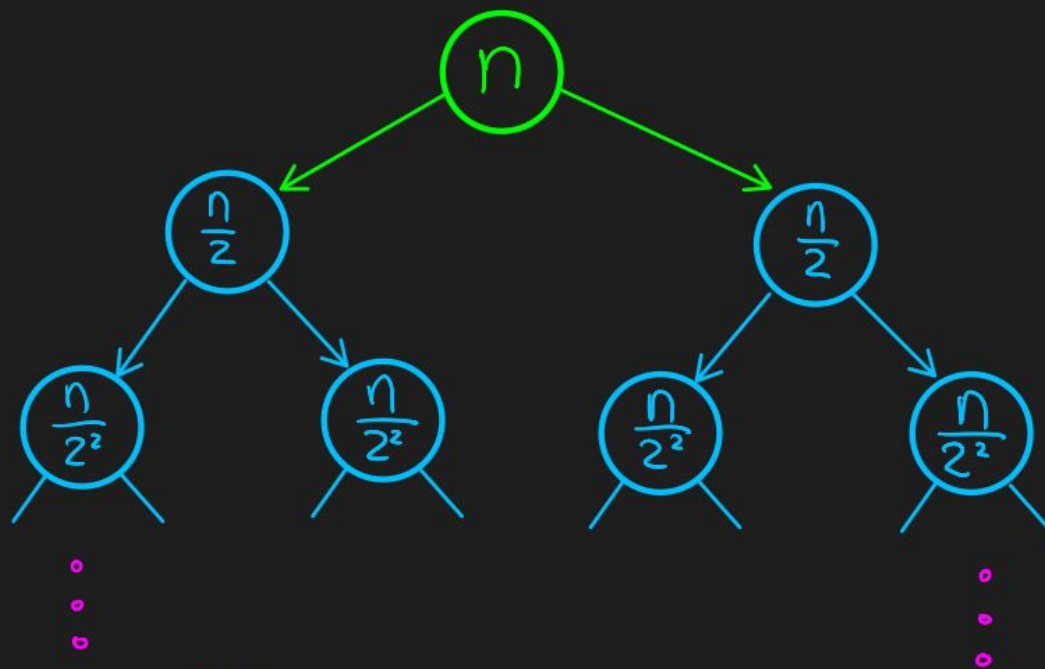
$$f(n) \in \Theta(n^{\log_2(2)}) = \Theta(n)$$

Por lo tanto, usando el Teorema Maestro, el costo de complejidad del algoritmo es de:

$$\Theta(n^{\log_2 2} \log n) = \Theta(n \log n)$$

## Analizando Árbol de Recursión

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$$



$$= O(n)$$

$$= O(n)$$

$$= O(n)$$

$$\frac{n}{2^k} + \frac{n}{2^k} + \dots + \frac{n}{2^k} + \frac{n}{2^k} = \Theta(n)$$

Donde  $K$  es la altura de un árbol binario completo de  $n$  elementos, es decir:

$$K = \log n$$

$\Rightarrow$  Como para cada nivel del árbol tenemos un costo de  $O(n)$ ,

para los  $K$  niveles del árbol tendremos

$$K \cdot O(n) = O(n \cdot K) = O(n \cdot \log n)$$

Formalmente:

Contando nodos por nivel (se duplican en cada paso)

	Nivel 0 (raíz) :	1 nodo ( $2^0$ )
	Nivel 1 :	2 nodos ( $2^1$ )
+	Nivel 2 :	4 nodos ( $2^2$ )
	$\vdots$	$\vdots$
	Nivel $K$ :	$2^K$ nodos

---

$$\text{Nodos totales} : \sum_{i=0}^K 2^i$$



Complejidad de cada nodo:  $\frac{n}{2^i}$ , con  $i$  su nivel

Juntando todo

$$\text{Complejidad} = O\left(\sum_{i=0}^k \cancel{2^i} \cdot \cancel{\frac{n}{2^i}}\right)$$

$$= O\left(\sum_{i=0}^k n\right)$$

$$= O((k+1) \cdot n)$$

$$= O(k \cdot n + n)$$

$$k \cdot n > n$$

$$= O(n \cdot k)$$

$$= O(n \cdot \log n) //$$