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 in 0 to 4 ≡ i 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-LZT0RprCoPyolJ6lFBbO?usp=sharing

```
// Primera llamada al algoritmo
contarMDPs(A) → int res
    int res ← contarMDPsRec(A, 0, Tam(A))                    T(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, izq, der) → int
    // Cantidad de elementos sobre los que opero
    int n ← der - izq                                                  Θ(1)
    // Casos base: Asumo n>0
    if n < 3 then                                                      Θ(1)
        return 0                                                       Θ(1)
```

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

    mitad    ← (izq + der) // 2                                           θ(1)
    cant_izq ← contarMDPsRec(A, izq, mitad)                               T(n/2)
    cant_der ← contarMDPsRec(A, mitad, der)                               T(n/2)
    cant_cen ← contarMDPsEnElCentro(A, mitad, izq, der)                   θ(n)

    return (cant_izq + cant_der + cant_cen)                               θ(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, izq, der)
    int n ← Tam(A)                                                O(1)
    // rango = n si izq,der abarcan todo A
    int rango ← der - izq                                         O(1)
    // Si A[i:j] es un arreglo vacío o muy chico, devuelvo 0
    if n < 3 or rango < 3 then                                    O(1)
        return 0                                                  O(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)                      O(1)
    arreglo(string) unaPal ← crearArreglo(1)                      O(1)
    dosPal[0] ← "mar"                                             O(1)
    dosPal[1] ← "del"                                             O(1)
    unaPal[0] ← "plata"                                           O(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)
```

```
                                                                    Θ(rango/2) ≡ Θ(n)
    int P_der  ← buscarSubsecu(A, desde=mitad, hasta=der, subsecu=unaPal)        Θ(n)

    // Caso 2: "mar" del lado izq, “del” y “plata” del derecho
    // Actualizo palabras a buscar
    unaPal[0]  ← “mar”                                                          O(1)
    dosPal[0]  ← “del”                                                          O(1)
    dosPal[1]  ← “plata”                                                        O(1)
    int M_izq  ← buscarSubsecu(A, desde=izq, hasta=mitad, subsecu=unaPal)      Θ(n)
    int DP_der ← buscarSubsecu(A, desde=mitad, hasta=der, subsecu=dosPal)      Θ(n)

    // Devuelvo resultado total entre todas las subsecuencias contadas
    return MD_izq * P_der + M_izq * DP_der                                      O(1)

buscarSubsecu(A, desde, hasta, subsecu)
    // Cantidad de palabras de la subsecuencia
    // (*) Nota: k es 1 ó 2 para el uso de contarMDPsEnElCentro()
    int k ← Tam(subsecu)                                                  O(1)
    int rango ← hasta - desde                                             O(1)

    // Creo contador acumulador de ocurrencias
    arreglo(arreglo(int)) cont ← crearArreglo(rango + 1)            O(rango+1) ≡ Θ(n)
    for pal in 0 to (rango + 1) do                                  O(rango+1) ≡ Θ(n)
        cont[pal] ← crearArreglo(k + 1)                                       O(1)
        // Los primeros elementos  de cada sub arreglo inician en 1
        cont[pal][0] ← 1                                                      O(1)
        // Los contadores comienzan en 0
        for m in 1 to (k + 1) do                                        O(k) ≡ O(1)*
            cont[pal][m] ← 0                                                  O(1)
        end for
    end for

    // Recorro rango de A contando palabras coincidentes
    // (Estoy buscando en una MITAD del array original)
    for pal in (desde + 1) to (hasta + 1) do                Θ(rango) // Θ(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] ←   cont[pal-desde - 1][m - 1] + \
                                     + cont[pal-desde - 1][m]             O(1)
            else
                // Solo arrastro contador previo
```

```
                    cont[pal-desde][m] ← 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) = Θ(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:

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

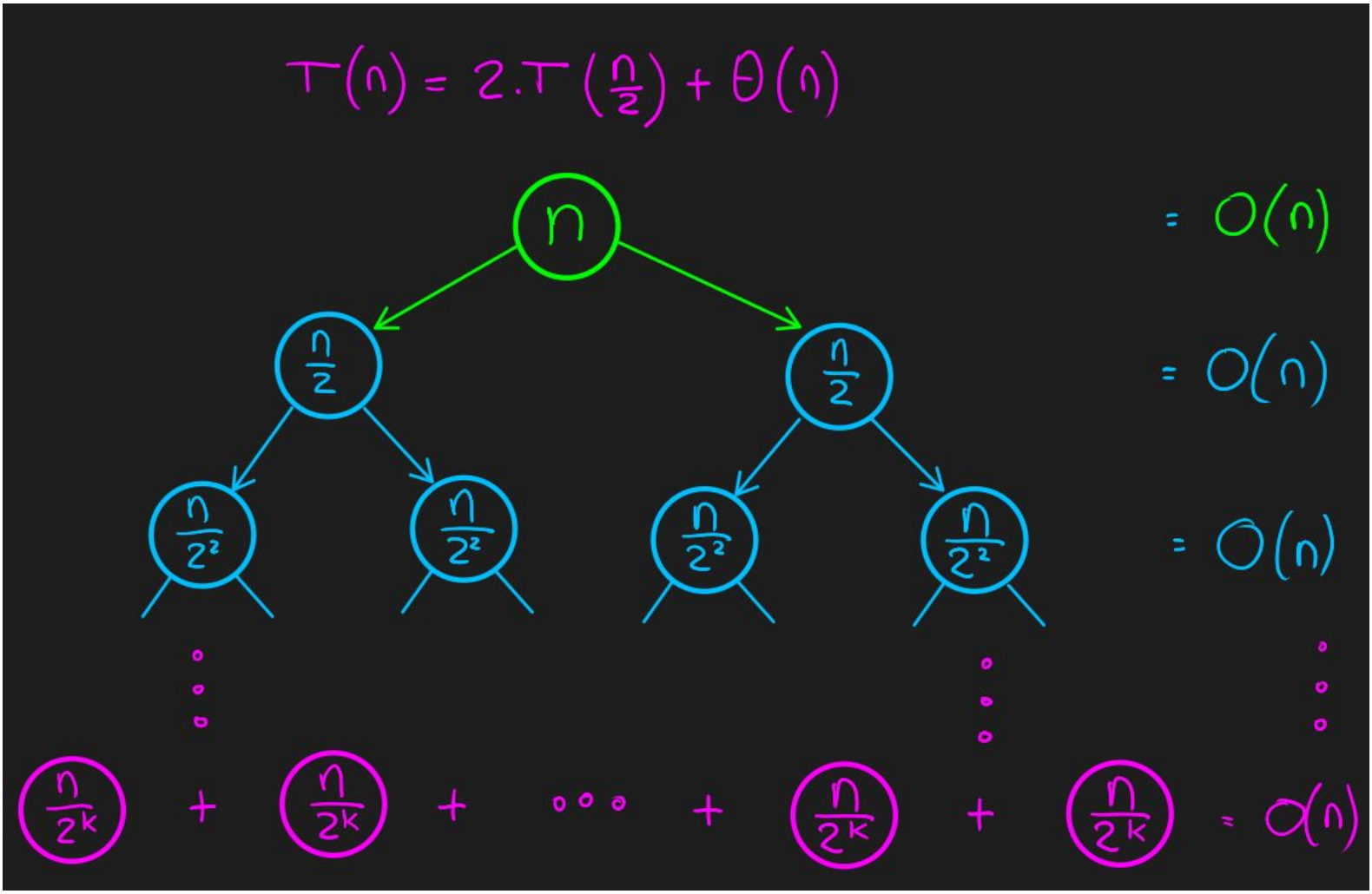
reemplazando

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

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



Donde k es la altura de un arbol binario completo de n elementos, es decir:

$$k = \log n$$

=> 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)
⋮		⋮
Nivel k	:	2^k nodos

+

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

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

Juntando todo

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

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

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

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

$k \cdot n > n$

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

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

---

fin :)