Algoritmos y Estructura de Datos 2

Recuperatorios

X2, X4, X7, 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 X2

2. Ejercicio X2 — TADs

En la playa de Mar del Plata venden empanadas y queremos llevar un registro de sus movimientos. Inicialmente contamos con un grupo de repartidores de empanadas, que se juntan en la confitería productora de las mismas. Cuando tiene ganas, algún repartidor se lleva varias empanadas consigo (la cantidad de quiera). A medida que pasa el tiempo los grupos de personas que están frente al mar llaman a algún repartidor y le piden la cantidad de empanadas que quieran, claramente no le pueden pedir más empanadas de las que tiene el repartidor en ese momento. Cuando esté cansado el repartidor vuelve a la base inmediatamente a descansar (si le sobran empanadas se las devuelve a la confitería). Si se le acaban las empanadas, el repartidor también debe volver y descansa hasta que le toque volver a salir.

Nos interesan saber varias cosas, primero cuál fue la compra más numerosa, es decir, cuál fue la máxima cantidad de empanadas vendidas en una sola compra y por cual vendedor (en caso de empate devuelvan todas las que correspondan). Por otro lado nos interesa saber cuales fueron los vendedores que más veces vendieron todas sus empanadas.

Modelar con el TAD EMP el sistema descripto. La especificación debe estar completa, incluyendo observadores, generadores, otras operaciones, restricciones y la axiomatización correspondiente. Se sugiere escribir la igualdad observacional, pero no es de caracter obligatorio.

```
Detalle de colores:
    rojo:    observadores
    azul:    generadores
    violeta: otras operaciones y auxiliares
    naranja: comentarios

Repartidor es String

TAD EMP
    géneros emp

    igualdad observacional
        (∀ e, f: emp)( e =obs f ⇐⇒
                    repartidores(e)  =obs  repartidores(f) ∧
                    comprasRecord(e) =obs  comprasRecord(f)∧
                    (∀ r: repartidor) r ∈ repartidores(e) ⇒
                        cantEmpanadas(e, r)        =obs cantEmpanadas(e, f)
                        vecesQueVendióTodas(e, r)  =obs vecesQueVendióTodas(e, r)

    observadores
        repartidores:         emp  → conj(repartidor)
        comprasRecord:        emp  → tupla(nat, conj(repartidor))
        cantEmpanadas:        emp e x repartidor r  → nat    {r ∈ repartidores(e)}
        vecesQueVendióTodas:  emp e x repartidor r  → nat    {r ∈ repartidores(e)}
```

```
generadores
    iniciar:              conj(repartidor) cr            → emp        {¬vacío?(cr)}
    tomarEmpanadas:       emp e x repartidor r x nat n → emp
                                                           {r ∈ repartidores(e)}
    venderEmpanadas:      emp e x repartidor r x nat n → emp
                                                       {r ∈ repartidores(e) ∧L
                                                        n ≤ cantEmpanadas(e, r)}


otras operaciones
    vendieronTodasMásVeces: emp → conj(repartidor)

axiomas    ∀e: emp, ∀cr: conj(repartidor), ∀r1, r2: repartidor, ∀n: nat
    repartidores(iniciar(cr)) ≡ cr
    repartidores(tomarEmpanadas(e, r, n)) ≡ repartidores(e)
    repartidores(venderEmpanadas(e, r, n)) ≡ repartidores(e)

    cantEmpanadas(iniciar(cr), r) ≡ 0
    cantEmpanadas(tomarEmpanadas(e, r1, n), r2) ≡
                    if r1 = r2 then
                        n
                    else
                        cantEmpanadas(e, r2)
                    fi
    cantEmpanadas(venderEmpanadas(e, r1, n), r2) ≡
                    if r1 = r2 then
                        cantEmpanadas(e, r2) - n
                    else
                        cantEmpanadas(e, r2)
                    fi

    comprasRecord(iniciar(cr)) ≡ < 0, ∅ >
    comprasRecord(tomarEmpanadas(e, r1, n)) ≡ comprasRecord(e)
    comprasRecord(venderEmpanadas(e, r, n)) ≡
                    if n = π1(comprasRecord(e)) then
                        // Venta igual a record presente
                        if r ∉ π2(comprasRecord(e)) then
                            // Agrego nombre al conjunto de la tupla
                            < π1(comprasRecord(e)),
                              Ag(r, π2(comprasRecord(e))) >
                        else
                            // Vendedor ya en records,
                            // no modifica comprasRecords.
                            comprasRecord(e)
                        fi
```

```
                    else if n > π1(comprasRecord(e)) then
                                // Nuevo record! Pisa valores anteriores
                                < n, Ag(r, ∅) >
                          else
                                // Venta sub-record, no modifica comprasRecords.
                                comprasRecord(e)
                          fi
                    fi


vecesQueVendióTodas(iniciar(cr), r) ≡ 0
vecesQueVendióTodas(tomarEmpanadas(e, r1, n), r2) ≡ vecesQueVendióTodas(e,
r2)

vecesQueVendióTodas(venderEmpanadas(e, r1, n), r2) ≡
                    if (cantEmpanadas(e, r1) - n) = 0 then
                          // Vendedor r1 vendió todas sus empanadas
                          if r1 = r2 then
                                1 + vecesQueVendióTodas(r2)
                          else
                                vecesQueVendióTodas(r2)
                          fi
                    else
                          // Vendedor r1 no vendió todas, no cambia nada
                          vecesQueVendióTodas(r2)
                    fi


vendieronTodasMásVeces(in e: emp) → conj(repartidor)
      filtrarMáximos(e, repartidores(e))

filtrarMáximos(in e: emp, in cr: conj(repartidor)) → conj(repartidor)
      // Devuelve repartidores con ventas completas igual al máximo
      if #(cr) = 0 then
            ∅
      else
            if vecesQueVendióTodas(e, dameUno(cr))
                  =
               mayorCantDeVentasCompletas(e, repartidores(e)) then
                  // Es uno de los que más vendió todas, lo agrego
                  Ag( dameUno(cr), filtrarMáximos(e, sinUno(cr)) )
            else
                  filtrarMáximos(e, sinUno(cr))
            fi
      fi
```

```
        mayorCantDeVentasCompletas(in e: emp, in cr: conj(repartidor)) ⟶ nat
             // Computa el máximo de ventas completas entre repartidores
             if #(cr) = 0 then
                  ∅
             else
                  if #(cr) = 1 then
                       vecesQueVendióTodas(e, dameUno(cr))
                  else
                       máx( vecesQueVendióTodas(e, dameUno(cr)),
                            mayorCantDeVentasCompletas(e, sinUno(cr)) )
                  fi
Fin TAD
```

Nota:

El tad especificado asume un **comportamiento automático** de los vendedores en donde descansan cuando tienen ganas, y de hacerlo cuando todavía tienen empanadas, **las devuelven automáticamente a la confitería**.

Esto puede verse explícitamente en la axiomatización de

cantEmpanadas(TomarEmpanadas(...))

donde al tomar nuevas empanadas, "se pisa" cualquier cantidad de empanadas que podría haber tenido el vendedor, lo que intuitivamente se corresponde con que el vendedor siempre devuelve las empanadas que le sobraron antes de pedir nuevas.

Ejercicio X4

4. Ejercicio X4 — Notación "O"

Sean $a \in \mathbb{R}$ y $b \in \mathbb{Z}^+$ constantes, $f, g\colon \mathbb{N} \to \mathbb{R}^+$, y $h(n) = q(n) + 1$, donde $q(n)$ es la cantidad de veces que 2 divide exactamente a n (por ejemplo, $q(1) = 0$, $q(2) = 1$, $q(8) = 3$, $q(10) = 1$).
Decidir si las siguientes afirmaciones son verdaderas o falsas. Justificar detalladamente.

1. $f(n) \in \Omega(f(n)^2) \Rightarrow f(n) \in O(1)$
2. $f(n) \in \Omega(g(n)) \Rightarrow f(n)^2 \in \Omega(g(n))$
3. $n! \in \Theta((n+1)!)$
4. $(n+a)^b \in \Theta(n^b)$.
5. $nf(m) + mg(n) \in O(f(nm) + g(nm))$
6. $h(n) \in O(n)$, y es la cota más justa posible.
7. $\log_2 n \in O(h(n))$
8. Hay una cantidad infinita de funciones $i(n)$, tal que $i(n) \in O(h(n))$

1. VERDADERO.

Si $f(n) \in \Omega(f(n)^2)$, por def de Ω :

$$\exists\, n_0, k > 0 \ tal\ que\ si\ n \geq n_0 \Rightarrow f(n) \ \geq k * f(n)^2$$

Como f(n) está en R+, puedo dividir por f(n) de ambos lados

$$\Rightarrow 1 \ \geq k * f(n)$$

Reescribo

$$f(n) \leq \tilde{k} * 1$$

con $\tilde{k} > 0$.

Esta es justamente la **definición de** $f(n) \in O(1)$:

$$\exists\, n_0, \tilde{k} > 0 \ tal\ que\ si\ n \geq n_0 \Rightarrow f(n) \ \leq \tilde{k}$$

Por lo tanto, es verdadera.

2. VERDADERO.

2) $f(n) \in \Omega(g(n)) \Rightarrow f(n)^2 \in \Omega(g(n))$

Por def:

$$\exists n_0, k>0 \;/\; n \geq n_0 \Rightarrow f(n) \geq k \cdot g(n)$$

divido por $f(n) > 0$

$$\Rightarrow \frac{f(n)}{f(n)} \geq k \cdot \frac{g(n)}{f(n)}$$

$$\Rightarrow 1 \geq k \cdot \frac{g(n)}{f(n)}$$

$k>0$

$$\Rightarrow \frac{1}{k} \geq \frac{g(n)}{f(n)}$$

Obtuve una cota superior para el cociente g(n)/f(n), dada por una constante $\tilde{k} \neq 0$ finita (pues existe k>0 finita) por lo que por propiedad de Big-Ω, Por prop. 8:

Si existe $\lim\limits_{n\to\infty} \frac{g(n)}{f(n)} = \tilde{K}$

y $K \neq 0$, $K < \infty$

entonces

$\Omega(f) = \Omega(g)$

Como $\Omega(f) = \Omega(g)$

$\Rightarrow f(n) \in \Omega(g(n))$

$\overset{prop}{\Rightarrow} f(n) \in \Omega(f(n))$

Usando el resultado del ejercicio 1:

$f(n) \in \Omega(f(n)) \Rightarrow f(n)^2 \in \Omega(f(n))$

Volviendo a que $\Omega(f) = \Omega(g)$

$f(n) \in \Omega(f(n)) \Rightarrow f(n)^2 \in \Omega(g(n))$ //

3. FALSO

3) $\underbrace{n!}_{f(n)} \in \Theta(\underbrace{(n+1) \cdot n!}_{g(n)})$

$$\lim_{n\to\infty} \frac{n!}{(n+1) \cdot n!} = \lim_{n\to\infty} \frac{1}{n+1} = 0 \overset{\text{prop.8}}{\Longrightarrow} \Theta(g) \neq \Theta(f)$$

$\therefore$ por contrarrecíproco de prop.2:

$$f \in \Theta(g) \Rightarrow \Theta(f) = \Theta(g)$$

$$f \notin \Theta(g)$$

$$n! \notin \Theta((n+1)!)$$

//

4. VERDADERO

4. $(n+a)^b \in \Theta(n^b)$

$$\lim_{n\to\infty} \frac{n^b}{(n+a)^b} = \lim_{n\to\infty} \left(\frac{n}{n+a}\right)^b$$

$$= \lim_{n\to\infty} \frac{n}{n}\cdot\frac{1}{\left(1+\underbrace{\frac{a}{n}}_{\to 0}\right)} = 1$$

Por prop, como el límite es finito y $\neq 0$

$\Rightarrow \Theta(f) = \Theta(g)$

$\therefore$ Como $(n+a)^b \in \Theta\left((n+a)^b\right)$

uso $\Theta(f) = \Theta(g)$

$\Rightarrow (n+a)^b \in \Theta(n^b)$ //

5. FALSO

$$5.\quad n.f(m) + m.g(n) \in O\big(f(n.m) + g(n.m)\big)$$

Sospecho que es falso, pues implicaría que

$$n.f(m) \in O\big(f(n.m)\big)$$

y

$$m.g(n) \in O\big(g(n.m)\big)$$

Construyo contraejemplo

$$\text{Si: } f(m) = 2^{1/m}$$

$$\Rightarrow f(n.m) = 2^{1/(n.m)}$$

Analizo cociente

$$\lim_{\substack{n\to\infty \\ m\to\infty}} \frac{2^{\frac{1}{n.m}}}{n.2^{\frac{1}{m}}} = \lim_{\substack{n\to\infty \\ m\to\infty}} \overbrace{\frac{1}{n}}^{\to 0} \cdot 2^{\overbrace{\frac{1}{n.m}}^{\to 0} - \overbrace{\frac{1}{m}}^{\to 0}} = 0$$

Por prop de Big-O

Como el cociente es cero :

$$n \cdot 2^{\frac{1}{m}} \notin O\left(2^{\frac{1}{n \cdot m}}\right)$$

$$g \notin O(f)$$

$$n \cdot f(m) \notin O(f(n \cdot m))$$

Equivalentemente,

$$m \cdot g(n) \notin O(g(n \cdot m))$$

Entonces

$$n \cdot f(m) + m \cdot g(n) \notin O(f(n \cdot m)) + O(g(n \cdot m))$$

$$n \cdot f(m) + m \cdot g(n) \notin O(f(n \cdot m) + g(n \cdot m)) \quad //$$

6. VERDADERO

6) $h(n) \in O(n)$

$$h(n) = q(n) + 1$$

Def: $h(n) \in O(n)$

$$\exists\, n_0, k > 0 \,/\, n \geq n_0 \Rightarrow h(n) \leq k \cdot n$$

$$\Rightarrow q(n) + 1 \leq k \cdot n$$

Como $n \geq q(n)$ $\forall n \in \mathbb{N}$

(pues no se puede dividir un número n por 2 n veces)

acoto $q(n)$ por n

$$\Rightarrow q(n) + 1 \leq n + 1 \leq k \cdot n$$

lo cual vale para $k \geq 2$

$$\therefore \quad h(n) \in O(n)$$

7. FALSO

7) $\log_2 n \in O(h(n))$

$\in O(q(n)+1)$

$\log_2 n = x$ se corresponde con $2^x = n$

O sea: cuántas veces divide 2 a n

Pero $q(n)$ cuenta veces exactas

$\Rightarrow$ si n es impar $\Rightarrow q(n) = 0$

$\Rightarrow h(n) = 1$

$\Rightarrow \log_2 n \in O(1)$

Abs!

$\therefore \log_2 n \notin O(h(n))$ //

8. VERDADERO

8) $i(n) \in O(h(n))$

Sé que

$$h(n) \in O(h(n))$$

$$q(n)+1 \in O(h(n))$$

y a partir de ella puedo crear infinitas funciones

$$i_1(n) = q(n) + 1 \in O(h(n))$$

$$i_2(n) = 2.(q(n)+1) \in O(2.h(n))$$

por def de Big-O

$$\in O(h(n))$$

$$i_k(n) = k \cdot (g(n)+1) \in O(k \cdot h(n)) \quad k \in \mathbb{N}$$

por def de Big-O

$$\in O(h(n))$$

$\therefore$ existen infinitas funciones $i_k(n)$,

y en particular

infinitas $i(n) \in O(h(n))$ //

Ejercicio X7

7. Ejercicio X7 — Ordenamiento

En la famosa playa La Perla quieren sacar una foto área artística donde todas las sombrillas estén ordenadas formando un cuadro de colores único. Cada sombrilla podríamos considerarla como una tupla de $\langle$Color, Diámetro$\rangle$ donde un Color es un código hexadecimal de u dígitos y Diámetro un natural positivo. Como las sombrillas tienen tamaños estándar, nos indican que hay a lo sumo k posibles tamaños de sombrillas (pero no se aclara a priori cuáles k). La muchachada nos solicita un algoritmo que ordene las sombrillas por color creciente y en caso de empate por tamaño decreciente. En concreto, diseñar el siguiente algoritmo:

$$\texttt{SombrillasSort}(\textbf{in } A : \text{arreglo(Sombrilla)}, \textbf{in } B : \text{arreglo(Diámetro)}) \longrightarrow C : \text{arreglo(Sombrilla)}$$

con **complejidad** $O(n \cdot u + (k+n) \log k)$ **en peor caso**, que genere un arreglo con todas las sombrillas ordenadas como se mencionó previamente. El arreglo A de tamaño n contiene las sombrillas y el B de tamaño k los diámetros posibles. Por ejemplo:

```
SombrillasSort([<"0x6A319",21>, <"0x6A319",27>, <"0xA1719",11>, <"0xA1720",21>], [27,11,21])
==> [<"0x6A319", 27>, <"0x6A319", 21>, <"0xA1719", 11>, <"0xA1720", 21>]
```

En este ejemplo: $u = 5$ y $k = 3$.

Observaciones:

Cada caracter del color, al ser un valor hexadecimal, estará acotado por una constante: 16

Asumo que u es la cantidad de dígitos **sin** contar el "**0x**" del comienzo.

Para los for loop uso que *i in 0 to **4*** ≡ *i in [0,1,2,**3**]* (o sea, no inclusive el limite derecho)

Uso ***tupla.prim*** y ***tupla.segu*** para acceder a primer y segundo elemento de una tupla.

Uso operador **//** como **división entera** que **redondea hacia abajo**

Uso función **MergeSort** pasando parámetro ***"orden"*** que establece el orden de ordenamiento (no cambia la complejidad del algoritmo, solo invierte comparaciones de orden por defecto si es "decreciente").

```
SombrillasSort(in A: arreglo(Sombrilla), in B: arreglo(Diámetro))→ C: arreglo(Sombrilla)
    int n ← tam(A)                                              O(1)
    int k ← tam(B)                                              O(1)
    // Ignoro caracteres "0x" en u
    int u ← tam(A[0].prim) - 2                              O(1)

    arreglo(Sombrilla) C ← arreglo(n)                           O(n)

    // Ordeno tamaños en B con algo. estable Merge Sort
    B_ord ← MergeSort(B, orden="decreciente")                   O(k log k)

    // Hay una correspondencia directa entre los índices
```

```
// y los tamaños B_ord ordenados decrecientemente
// ej: [27, 21, 11] se corresponde con [0, 1, 2]
arreglo(lista(Sombrilla)) A_tamaños ← crearArreglo(k)       O(k)
for t in 0 to k do                                          O(k)
      A_tamaños[t] ← Vacía()                                O(1)
end for

// Ordeno sombrillas por tamaño decreciente
int idx ← 0                                                 O(1)
for sombri in A do                                          O(n * log k)
      idx ← deTamañoAIdx(B_ord, sombri.segu)                O(log k)
      AgregarAtrás(A_tamaños[idx], sombri)                  O(1)
end for

// Vuelvo a convertir A en un único arreglo
A ← concatenarEnArreglo(A_tamaños, n)                 O(n * u)

// Antes de volver a ordenar, creo acceso rápido a
// los dígitos de los colores ordenados
// Gano acceso a string como arreglo de chars
arreglo(arreglo(char)) colores ← crearArreglo(n)      O(n)
i ← 0                                                       O(1)
for sombri in A do                                          O(n * u)
      colores[i] ← deStringAArreglo(sombri.prim)            O(u)
      i++                                                   O(1)
end for

// Diccionario auxiliar para mapear hexadecimal (char) a decimal (int)
dicc(char, int) deHexaADec ← diccAVL()                O(1)
i ← 0                                                 O(1)
for c in “0123456789ABCDEF” do                        O(16 * 1) ≡ O(1)
      definir(deHexaADec, c, i)                       O(log 16) ≡ O(1)
      i++                                             O(1)
end for

// Ordeno colores con RadixSort, recorriendo dígitos desde las unidades
// Uso dos arreglos de forma intercalada
arreglo(lista(Sombrilla)) A_colorUno ← crearArreglo(16)     O(1)
arreglo(lista(Sombrilla)) A_colorDos ← crearArreglo(16)     O(1)

for i in 0 to 16 do                                         O(1)
      A_colorUno[i] ← Vacía()                               O(1)
      A_colorDos[i] ← Vacía()                               O(1)
end
```

```
    int idx      ← 0                                                   O(1)
    int toWrite ← 0                                               O(1)
    for dígito in 0 to u do                                            O(   )
        // Recorro todos los dígitos de atrás hacia adelante...
        for i in 0 to n do
            // ...para cada una de las Sombrillas de A
            // Obtengo índice del bucket (entre 0 y 15 inclusives)
            idx ← obtener(deHexaADec, colores[i][u-1-dígito])     O(log 16) ≡ O(1)
            if dígito = u-1 then                                           O(1)
                // Primera vez copia de A
                AgregarAtrás(A_colorUno[idx], A[i])                        O(1)
                toWrite ← 2                                                O(1)
            else if toWrite = 1 then
                // Paso de ordDos a ordUno
                AgregarAtrás(A_colorUno[idx], A_colorDos[i])               O(1)
                A_colorDos[i] ← Vacía()                                    O(1)
                toWrite ← 2                                                O(1)
            else
                // Paso de ordUno a ordDos
                AgregarAtrás(A_colorDos[idx], A_colorUno[i])               O(1)
                A_colorUno[i] ← Vacía()                                    O(1)
                toWrite ← 1                                                O(1)
            end if
        end for
    end for

    // Guardo referencia para acceder al último array de sombrillas ordenadas
    if toWrite = 1 then
        ordenadas ← A_colorDos                                             O(1)
    else
        ordenadas ← A_colorUno                                             O(1)
    end if

    // Concateno todas las listas en una única (ordenada)
    C ← concatenarEnArreglo(ordenadas, n)                              O(n * u)



deTamañoAIdx(in tamañosOrd: arreglo(Diámetro), in t: Diámetro) → idx: int
    // Pre: t existe en el arreglo tamañosOrd
    int k       ← tam(tamañosOrd)                                      O(1)
    int izq     ← 0                                                    O(1)
    int der     ← k-1                                                  O(1)
    bool encontrado ← false                                            O(1)
    // Uso búsqueda binaria
```

```
    int idx    ⟵ (der - izq) // 2                                    O(1)
    while (der-izq) > 0 and encontrado = false do                    O(log k)
        if t > tamañosOrd[idx] then                                  O(1)
            // Busco en primera mitad
            der ⟵ idx                              O(1)
            idx ⟵ izq + (der - izq) // 2                             O(1)
        else
            if t < tamañosOrd[idx] then                              O(1)
                // Busco en segunda mitad
                if idx = k-2 then                                    O(1)
                    // Salvo caso borde derecho
                    idx ⟵  k-1                               O(1)
                    encontrado ⟵  true                               O(1)
                else
                    izq ⟵ idx                                        O(1)
                    idx ⟵ izq + (der-izq) // 2               O(1)
                end if
            else
                // Lo encontré
                encontrado ⟵ true                                    O(1)
            end if
        end if
    end while


concatenarEnArreglo(in X: arreglo(lista(Sombrilla)), in n: int)
                                          ⟶ aplanado: arreglo(Sombrillas)
    // Función que guarda secuencialmente los elementos de las listas
    // de un arreglo de listas en un único arreglo
    arreglo(Sombrilla) aplanado ⟵ crearArreglo(n)            O(n)
    int i ⟵ 0                                                O(1)
    for lis in X do                                          O(n * u)
        for som in lis do
            aplanado[i] ⟵ som                                O(u)
            i++                                              O(1)
        end for
    end for

deStringAArreglo(in str: string) ⟶ arr: arreglo(char)
    // Ignoro los primeros dos caracteres
    u ⟵ long(string) - 2                                 O(1)
    arreglo(char) arr ⟵ arreglo(u)                           O(u)
    int i ⟵ 0                                                O(1)
    for c in str do                                          O(u)
        if i >= 2 then                                       O(1)
```

```
        arr[i-2] ← c                                        O(1)
    end if
    i++                                                     O(1)
end for
```

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).

<u>Implementación</u>: 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, i, j) ⟶ int
    // Cantidad de elementos sobre los que opero
    int n ⟵ j - i                                                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    ← (i + j) // 2                                             Θ(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)                 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, i, j)
    int n ← Tam(A)                                          O(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                              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:

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

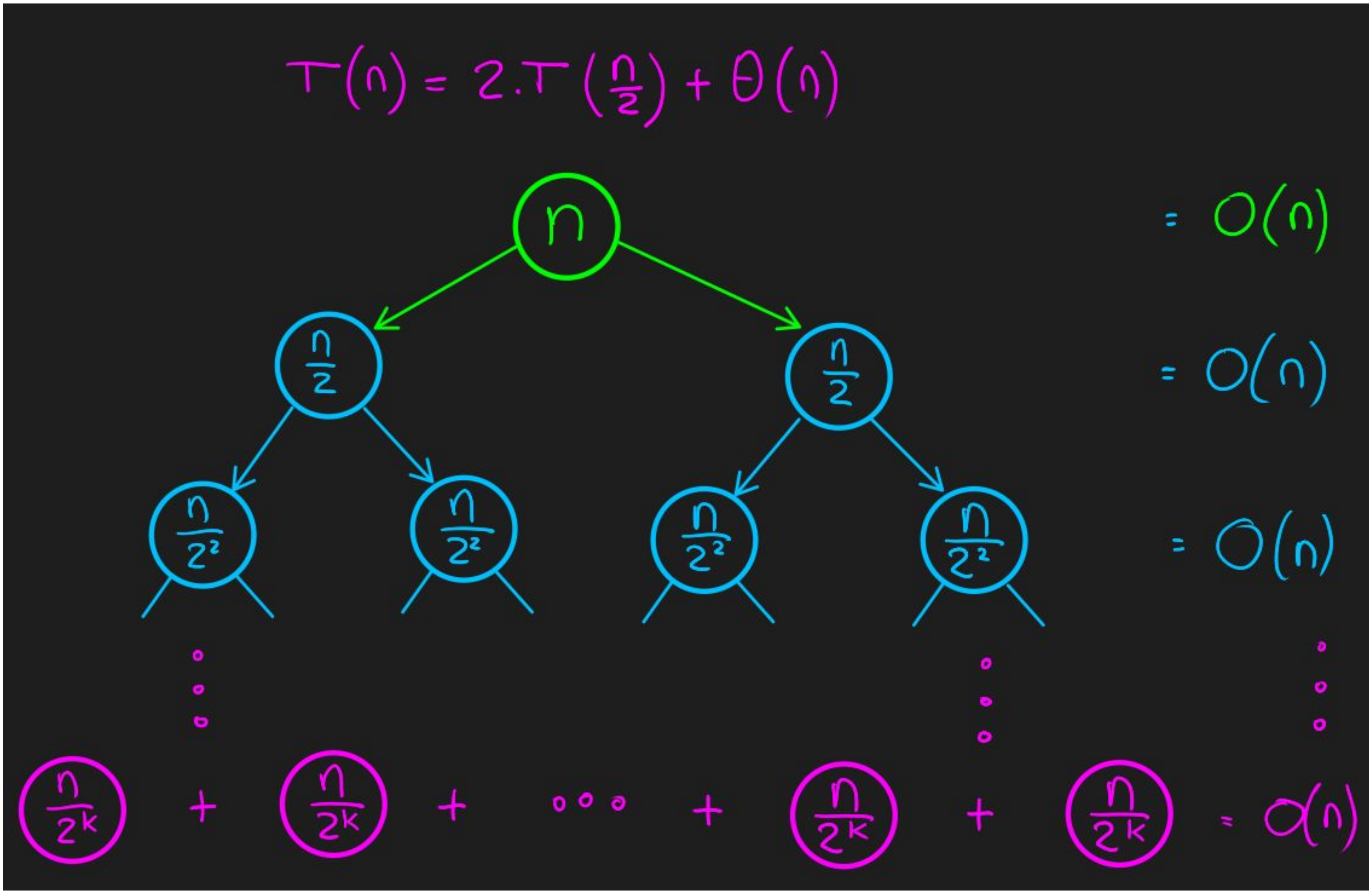
reemplazando

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

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:

$$K = \log n$$

$\Rightarrow$ 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)

$$
\begin{array}{llll}
 & \text{Nivel } 0 \text{ (raíz)} & : & 1 \text{ nodo } (2^0) \\
 & \text{Nivel } 1 & : & 2 \text{ nodos } (2^1) \\
+ & \text{Nivel } 2 & : & 4 \text{ nodos } (2^2) \\
 & \vdots & & \vdots \\
 & \text{Nivel } K & : & 2^K \text{ nodos} \\
\hline
 & \text{Nodos totales} & : & \displaystyle\sum_{i=0}^{K} 2^i
\end{array}
$$

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

Juntando Todo

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

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

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

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

$k.n > n$

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

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

fin :)