Proyecto Análisis y Diseño de Algoritmos

Renato Bacigalupo Ortiz-Arrieta

Junio 2020

1. Introduccion

En este documento se hará el diseño de los algoritmos pedidos en el proyecto del curso de Análisis y Diseño de Algoritmos. Los códigos implementados pueden encontrarse en las carpetas adjuntadas o en este repositorio de github.

2. Secuencias

Pregunta 1 (Voraz)

Este primer algoritmo estará dividido en dos subrutinas, la subrutina Match y la subrutina Min-Matching. Esto lo hacemos para que sea más entendible el algoritmo. Implementación: ver anexo o github Secuencias/Pregunta1.cpp

Lo que hace el primer algoritmo Match es encontrar y guardar en un arreglo los bloques de los arreglos de unos y ceros de entrada. Es decir, que este primer algoritmo calcula el índice de inicio y final de cada uno de cada bloque en los dos arreglos.

Al final Match llama a la subrutina Min-Matching.

Primero analizaremos el tiempo de ejecución de Match. No contaremos dentro de este algoritmo el tiempo de ejecución de la llamada a Min-Matching, ya que este tiempo lo calcularemos después, al final juntaremos los dos tiempos. Tomaremos un peor caso como el de inputs como los siguientes:

$$A = \{0, 1, 0, 1, 0, 1, 0, 1, 0, 1\}$$

$$B = \{0, 1, 0, 1, 0, 1, 0, 1, 0, 1\}$$

Require: Dos arreglos A y B con ceros y unos de tamaño p.
Ensure: Dos arreglos de los bloques de unos de cada arreglo resperctivo.

Match(A, B, p)	*cost*	*times*
$M_1, M_2 = \emptyset$	$c1$	1
$n_1 = -1$	$c2$	1
$m_1 = -1$	$c3$	1
for $i = 0$ **to** p	$c4$	$p + 1$

if $A[i] == 1$ **and** $n_1 == -1$	$c5$	p
$\quad n_1 = i$	$c6$	$p/2$
else if $A[i] == 0$ **and** $n_1 \neq -1$	$c7$	$p/2$
$\quad M_1 = M_1 \cup (n_1, i-1)$	$c8$	$p/2$
$\quad n_1 = -1$	$c9$	$p/2$
if $i == p-1$ **and** $A[i] == 1$ **and** $n_1 \neq -1$	$c10$	p
$\quad M_1 = M_1 \cup (n_1, i-1)$	$c11$	0
else if $i == p-1$ **and** $A[i] == 1$ **and** $n_1 == -1$	$c12$	1
$\quad M_1 = M_1 \cup (i-1, i-1)$	$c13$	1
if $B[i] == 1$ **and** $m_1 == -1$	$c14$	p
$\quad m_1 = i$	$c15$	$p/2$
else if $B[i] == 0$ **and** $m_1 \neq -1$	$c16$	$p/2$
$\quad M_2 = M_2 \cup (m_1, i-1)$	$c17$	$p/2$
$\quad m_1 = -1$	$c18$	$p/2$
if $i == p-1$ **and** $B[i] == 1$ **and** $m_1 \neq -1$	$c19$	p
$\quad M_2 = M_2 \cup (m_1, i-1)$	$c20$	0
else if $i == p-1$ **and** $B[i] == 1$ **and** $m_1 == -1$	$c21$	1
$\quad M_2 = M_2 \cup (i-1, i-1)$	$c22$	1
return MIN-MATCHING(M_1, M_2)	?	1

Ahora sumamos todos los tiempos y las veces de ejecución: $T(A,B,p) = c1 + c2 + c3 + c4(p+1) + c5p + c6\frac{p}{2} + c7\frac{p}{2} + c8\frac{p}{2} + c9\frac{p}{2} + c10p + c11*0 + c12 + c13 + c14p + c15\frac{p}{2} + c16\frac{p}{2} + c17\frac{p}{2} + c18\frac{p}{2} + c19\frac{p}{2} + c20*0 + c21 + c22$

Con esto concluimos que el tiempo de ejecución de MATCH (sin contar el tiempo de ejecución de MIN-MATCHING) es $O(p)$

Ahora calcularemos el tiempo de ejecución de MIN-MATCHING considerando un peor caso para el algoritmo, este será el caso en el que se hace match de uno a uno de todos los bloques hasta que uno llega al a su bloque final y tiene que agrupar o dividir los restantes obligatoriamente.

Require: Dos arreglos A y B con ceros y unos de tamaño p.
Ensure: Un conjunto con el maching no necesariamente optimo, y su peso.

MIN-MATCHING(M_1, M_2)	*cost*	*times*
$S = \emptyset$	$c1$	1
$i, j, temp = 0$	$c2$	1
$peso, num = 0$	$c3$	1
while $i \leq M_1.length$ **and** $j \leq M_2.length$	$c4$	$max(n,m)$
$\quad$ **if** $j == M_2.length$	$c5$	1
$\quad\quad num = 0$	$c6$	1
$\quad\quad$ **while** $i \leq M_1.length$	$c7$	$n-m+1$
$\quad\quad\quad S = S \cup (M_1[i], M_2[j])$	$c8$	$n-m$
$\quad\quad\quad num+ = M_1[i].w$	$c9$	$n-m$
$\quad\quad\quad i+ = 1$	$c10$	$n-m$
$\quad\quad peso+ = num/M_2[j].w$	$c11$	1
$\quad$ **else if** $i == M_1.length$	$c12$	0

13: $num = 0$	$c13$	0
14: **while** $j \leq M_2.length$	$c14$	0
15: $S = S \cup (M_1[i], M_2[j])$	$c15$	0
16: $num+ = M_2[j].w$	$c16$	0
17: $j+ = 1$	$c17$	0
18: $peso+ = M_1[i].w/num$	$c18$	0
19: **else if** $M_1[i].w > M_2[j].w$	$c19$	0
20: $temp = M_1[i].w$	$c20$	0
21: $num = 0$	$c21$	0
22: **while** $temp > 0$	$c22$	0
23: $S = S \cup (M_1[i], M_2[j])$	$c23$	0
24: $temp = temp - M_2[j].w$	$c24$	0
25: $num+ = M_2[j].w$	$c25$	0
26: $j+ = 1$	$c26$	0
27: $peso+ = M_1[i].w/num$	$c27$	0
28: $i+ = 1$	$c28$	0
29: **else if** $M_1[i].w < M_2[j].w$	$c29$	0
30: $temp = M_2[j].w$	$c30$	0
31: $num = 0$	$c31$	0
32: **while** $temp > 0$	$c32$	0
33: $S = S \cup (M_1[i], M_2[j])$	$c33$	0
34: $temp = temp - M_1[j].w$	$c34$	0
35: $num+ = M_1[i].w$	$c35$	0
36: $i+ = 1$	$c36$	0
37: $peso+ = num/M_2[j].w$	$c37$	0
38: $j+ = 1$	$c38$	0
39: **else if** $M_1[i].w == M_2[j].w$	$c39$	$min(n, m) + 1$
40: $S = S \cup (M_1[i], M_2[j])$	$c40$	$min(n, m)$
41: $peso+ = \frac{M_1[i].w}{M_2[j].w}$	$c41$	$min(n, m)$
42: $i, j+ = 1$	$c42$	$min(n, m)$
43: **return** $S, peso$	$c44$	1

Ahora sumamos todos los valores de costo y tiempo: $T(M_1, M_2) = c1 + c2 + c3 + c4max(n, m) + c5 + c6 + c7(n - m + 1) + c8(n - m) + c9(n - m) + c10(n - m) + c11 + c39min(n, m) + 1 + c40min(n, m) + c41min(n, m) + c42min(n, m) + c43min(n, m) + c44$

En este peor caso particular para el algoritmo como se ejecuta la línea 5, eso quiere decir que el $min(n, m) = m$ y el $max(n, m) = n$. Intercambiaremos estos valores para poder darnos cuenta al final del tiempo de ejecución real del algoritmo. Entonces: $T(M_1, M_2) = c1 + c2 + c3 + c4n + c5 + c6 + c7(n - m + 1) + c8(n - m) + c9(n - m) + c10(n - m) + c11 + c39m + 1 + c40m + c41m + c42m + c44$

Esto simplificado queda: $T(M_1, M_2) = cn + c_1$, pero esto se consideró para este caso donde $max(n, m) = n$, lo que quiere decir, que también existe un peor caso donde $max(n, m) = m$, lo que quiere decir que este algoritmo tiene tiempo de ejecución: $O(max(n, m))$

Pregunta 2 (Recurrecia)

Sean A y B arreglos que contienen los pesos de los bloques.

Sea $OPT(i,j)$ el peso de una solución optima para el subproblema que solo considera a los i primeros bloques de A y a los j primeros bloques de B.

$$OPT(i,j) = \begin{cases} \text{mín}\{\text{mín}_{a=i-1}^{1}\{OPT(a-1,j-1) + \frac{\sum_{b=a}^{i} A[b]}{B[j]}\}, & \\ \text{mín}_{a=j-1}^{1}\{OPT(i-1,a-1) + \frac{A[i]}{\sum_{b=a}^{j} B[b]}\}, & \\ OPT(i-1,j-1) + \frac{A[i]}{B[j]}\} & i > 0, j > 0 \\ \frac{A[i]}{\sum_{a=0}^{j} B[a]} & i = 0, j > 0 \\ \frac{\sum_{a=0}^{i} A[a]}{B[j]} & i > 0, j = 0 \\ \frac{A[i]}{B[j]} & i = 0, j = 0 \end{cases}$$

Demostraremos por inducción que la recurrencia tiene tiempo de ejecución $\Omega(2^{max(n,m)})$, es decir $OPT(i,j) = \Omega(2^{max(i,j)})$. En el caso base tenemos $i = 1$ y $j = 1$:

$$\frac{A[1]}{B[1]} \geq c2^{max(1,1)}$$

Si $c = \frac{A[1]/B[1]}{2}$ entonces:

$$\frac{A[1]}{B[1]} \geq \frac{A[1]}{B[1]}$$

Por ende, el caso base cumple. Ahora probaremos el caso inductivo:

$$OPT(i,j) \geq c2^{max(i,j)}$$

Sabemos que: $OPT(i,j) = \text{mín}\{\text{mín}_{a=i-1}^{1}\{OPT(a-1,j-1) + \frac{\sum_{b=a}^{i} A[b]}{B[j]}\}, \text{mín}_{a=j-1}^{1}\{OPT(i-1,a-1) + \frac{A[i]}{\sum_{b=a}^{j} B[b]}\}, OPT(i-1,j-1) + \frac{A[i]}{B[j]}\}$

Ahora probaremos que cada miembro de los parámetros de la subrutina mín son $\Omega(2^{max(i,j)})$. De esta forma no importa cual de los 3 se eliga o si es que se ejecutan los 3 para verificar, el tiempo de ejecución seguirá siendo $\Omega(2^{max(i,j)})$.

Primero tenemos:

$$\underset{a=i-1}{\overset{1}{\text{mín}}}\{OPT(a-1,j-1) + \frac{\sum_{b=a}^{i} A[b]}{B[j]}\} \geq c2^{max(i,j)}$$

Por hipótesis de inducción:

$$\underset{a=i-1}{\overset{1}{\text{mín}}}\{c \cdot 2^{max(a-1,j-1)} + \frac{\sum_{b=a}^{i} A[b]}{B[j]}\} \geq c2^{max(i,j)}$$

Evaluaremos los extremos de la subrutina mín para determinar si estos dos son mayores a el tiempo dicho, si es que lo son entonces cualquier valor en medio lo será también:

Primero el caso del extremo mayor, es decir $a = i - 1$:

$$c \cdot 2^{max(i-2,j-1)} + \frac{\sum_{b=i-1}^{i} A[b]}{B[j]} \geq c2^{max(i,j)}$$

Diremos que $\frac{\sum_{b=i-1}^{i} A[b]}{B[j]} = k$ al ser constante:

$$c \cdot 2^{max(i-2,j-1)} + k \geq c2^{max(i,j)}$$

Para un $c = \frac{1}{2}$ y un $k > 0$:

$$2^{max(i-1,j)} + k \geq 2^{max(i-1,j-1)}$$

Por ende, cumple para el caso donde $a = i - 1$.

Después para el caso del extremo menor, es decir $a = 1$:

$$c \cdot 2^{max(1,j-1)} + \frac{\sum_{b=1}^{i} A[b]}{B[j]} \geq c2^{max(i,j)}$$

Diremos que $\frac{\sum_{b=i-1}^{i} A[b]}{B[j]} = k$ al ser constante:

$$c \cdot 2^{max(1,j-1)} + k \geq c2^{max(i,j)}$$

Para un $c = \frac{1}{2}$ y un $k > 0$:

$$2^{max(0,j)} + k \geq 2^{max(i-1,j-1)}$$

Por ende, cumple para el caso donde $a = 1$.

Es claro que para el caso del segundo parámetro se hace un proceso igual, entonces pasaremos a comprobar el tercer parámetro:

Tenemos:

$$OPT(i-1, j-1) + \frac{A[i]}{B[j]} \geq c2^{max(i,j)}$$

Por hipótesis de inducción:

$$c \cdot 2^{max(i-1,j-1)} + \frac{A[i]}{B[j]} \geq c2^{max(i,j)}$$

Diremos que $\frac{A[i]}{B[j]} = k$:

$$c \cdot 2^{max(i-1,j-1)} + k \geq c2^{max(i,j)}$$

$=$

$$c \cdot 2^{max(i,j)-1} + k \geq c2^{max(i,j)}$$

Para un $c = \frac{1}{2}$ y un $k > 0$:

$$2^{max(i,j)} + k \geq 2^{max(i,j)-1}$$

Por ende, cumple para el tercer parámetro.

Al estos tres parámetros cumplir con $\Omega(2^{max(i,j)})$, es claro que al ejecutarlos al mismo tiempo entonces el tiempo de ejecución será mayor, por ende $OPT(i,j) = \Omega(2^{max(i,j)})$.

Pregunta 3 (Recursivo)

Implementación: ver anexo o github Secuencias/Pregunta3.cpp

Para que sea más sencillo mantener la data en cada paso del siguiente algoritmo vamos a definir un struct, esto será lo que retorne el algoritmo. Dentro de él guardara el peso y un arreglo de los matches.

STRUCT VALORES

double w
match[] // Este guardara pares ordenados

Aparte de este struct vamos a dividir el algoritmo en dos diferentes algoritmos, como se hizo en el ejercicio 1, el inicial encontrará los diferentes bloques dentro de los arreglos de unos y ceros y el segundo algoritmo hará el proceso de matching.

Require: Dos arreglos de unos y ceros A y B con tamaño p
Ensure: Matching entre los dos arreglos y el peso del mismo

MATCH(A, B, p)

M_1, M_2 // *Estos arreglos guardaran los bloques*
$n_1 = -1$
$m_2 = -1$
$\_i, \_j = 0$
for $i = 0$ **to** p
    if $A[i] == 1$ **and** $n_1 == -1$
        $n_1 = i$
    else if $A[i] == 0$ **and** $n_1 \neq -1$
        $M_1.push\_back(< n_1, i-1 >)$
        $n_1 = -1$
        $\_i += 1$
    if $i == p-1$ **and** $A[i] == 1$ **and** $n_1 \neq -1$
        $M_1.push\_back(< n_1, i-1 >)$
        $\_i += 1$
    else if $i == p-1$ **and** $A[i] == 1$ **and** $n_1 == -1$
        $M_1.push\_back(< i-1, i-1 >)$
        $\_i += 1$
    if $B[i] == 1$ **and** $m_1 == -1$
        $m_1 = i$
    else if $B[i] == 0$ **and** $m_1 \neq -1$
        $M_2.push\_back(< m_1, i-1 >)$
        $m_1 = -1$
        $\_j += 1$
    if $i == p-1$ **and** $B[i] == 1$ **and** $m_1 \neq -1$
        $M_2.push\_back(< m_1, i-1 >)$
        $\_j += 1$
    else if $i == p-1$ **and** $B[i] == 1$ **and** $m_1 == -1$

$M_2.push\_back(< i-1, i-1 >)$

$\_j += 1$

return $Opt(M_1, M_2, \_i - 1, \_j - 1)$

Este primer algoritmo es igual al algoritmo MATCH que mostramos en el ejercicio 1, solo que tiene dos contadores extra para saber exactamente el número de bloques de cada arreglo.

Sin embargo, este cambio no afecta el tiempo de ejecución de este algoritmo que sigue siendo $O(p)$.

El siguiente algoritmo recibe lo que el algoritmo anterior le da como parámetro, es decir dos arreglos con los pesos de los bloques y sus tamaños respectivos.

Al este algoritmo ser el mismo que explica la recurrencia de la pregunta 2, solo que con unos pasos extra podemos afirmar que este algoritmo será $\Omega(2^{max(i,j)})$.

Se debe tomar en cuenta que $A[i].peso$ o $B[j].peso$ y las declaraciones parecidas calculan $A[i].second - B[i].first + 1$ o $B[j].second - B[j].first + 1$ respectivamente.

Require: Dos arreglos de pesos de bloques A y B con sus tamaños respectivos i y j.
Ensure: Matching mínimo entre los dos arreglos y el peso del mismo.

OPT(A, B, i, j)	*cost*	*times*
$values\ v$	$c1$	1
if $i == 0$ **and** $j > 0$	$c2$	1
$\quad temp = 0$	$c3$	1
$\quad$ **for** $a = 0$ **to** j	$c4$	1
$\qquad temp+ = B[a].peso$	$c5$	1
$\qquad v.match = v.match \cup (A[i], B[a])$	$c6$	1
$\quad v.w = A[i].peso/temp$	$c7$	1
$\quad$ **return** v		
else if $i > 0$ **and** $j == 0$	$c8$	1
$\quad temp = 0$	$c9$	1
$\quad$ **for** $a = 0$ **to** i	$c10$	1
$\qquad temp+ = A[a].peso$	$c11$	1
$\qquad v.match = v.match \cup (A[a], B[j])$	$c12$	1
$\quad v.w = temp/B[j].peso$	$c13$	1
$\quad$ **return** v		
else if $i == 0$ **and** $j == 0$	$c14$	1
$\quad v.match = v.match \cup (A[i], B[j])$	$c15$	1
$\quad v.w = A[i].peso/B[j].peso$	$c16$	1
$\quad$ **return** v		
else if $i > 0$ **and** $j > 0$	$c17$	1
$\quad values\ min_1, min_2, min_3$	$c18$	1
$\quad min_1.w = inf$	$c19$	1
$\quad min_2.w = inf$	$c20$	1
$\quad min_3.w = inf$	$c21$	1
$\quad$ **for** $a = i - 1$ **to** 0	$c22$	1
$\qquad temp_1 = 0$	$c23$	1
$\qquad temp = 0$	$c24$	1
$\qquad$ **for** $b = a$ **to** i	$c25$	1
$\qquad\quad temp_1+ = A[b].peso$	$c26$	1
$\qquad temp_2 = Opt(A, B, a - 1, j - 1)$	$T(A, B, a - 1, j - 1)$	1
$\qquad temp = temp_2.w + (temp_1/B[j].peso)$	$c28$	1
$\qquad$ **if** $temp < min_1.w$	$c29$	1
$\qquad\quad min_1.match = temp_2.match$	$c30$	1
$\qquad\quad min_1.w = temp$	$c31$	1
$\qquad\quad a_1 = a$	$c32$	1

for $a = j - 1$ **to** 0	$c33$	1
$temp_1 = 0$	$c34$	1
$temp = 0$	$c35$	1
for $b = a$ **to** j	$c36$	1
$temp_1 += B[b].peso$	$c37$	1
$temp_3 = Opt(A, B, i - 1, a - 1)$	$T(A, B, i - 1, a - 1)$	1
$temp = temp_3.w + (A[i].peso/temp_1)$	$c39$	1
if $temp < min_2.w$	$c40$	1
$min_2.match = temp_3.match$	$c41$	1
$min_2.w = temp$	$c42$	1
$a_2 = a$	$c43$	1
$temp_4 = Opt(A, B, i - 1, j - 1)$	$T(A, B, i - 1, j - 1)$	1
$min_3.w = temp_4.w + A[i].peso/B[j].peso$	$c45$	1
$min_3.match = temp_4.match$	$c46$	1
$cond = 0$	$c47$	1
$values\ min$	$c48$	1
$min.w = min_1.w$	$c49$	1
$min.match = min_1.match$	$c50$	1
for $a = a_1$ **to** i	$c51$	1
$min.match = min.match \cup (A[a], B[j])$	$c52$	1
if $min.w > min_2.w$	$c53$	1
$min.w = min_2.w$	$c54$	1
$min.match = min_1.match$	$c55$	1
for $a = a_1$ **to** i	$c56$	1
$min.match = min.match \cup (A[i], B[a])$	$c57$	1
else if $min.w > min_3.w$	$c58$	1
$min.w = min_3.w$	$c59$	1
$min.match = min_1.match$	$c60$	1
$min.match = min.match \cup (A[i], B[j])$	$c61$	1
return min		

Pregunta 4 (Memoizado)

Implementación: ver anexo o github Secuencias/Pregunta4.cpp

Para este algoritmo se volverá a usar el struct declarado para el algoritmo anterior, de este modo será un poco más entendible el seudocódigo.

Al igual que el algoritmo anterior se partirá este algoritmo en dos algoritmos, el primero calculará los bloques dentro de los arreglos dados y el segundo hará el cálculo del matching.

El algoritmo para calcular los bloques es exactamente igual al algoritmo para calcular los bloques de la pregunta anterior, solo que al final se llama a Min-Matching-Memoizado, en vez de llamar a Opt. Por ende, no se volverá a copiar mismo código, se intuye que existe.

M será la matriz donde se guarden los valores ya calculados. Este guardara los structs values que se mencionaron anteriormente. Se debe tomar en cuenta que $A[i].peso$ o $B[j].peso$ y las declaraciones parecidas calculan $A[i].second-B[i].first+1$ o $B[j].second-B[j].first+1$ respectivamente.

Require: Dos arreglos A y B con los pesos de los bloques y sus tamaños respectivos.
Ensure: Matching mínimo entre los dos arreglos y su peso.
Min-Matching-Memoizado(A, B, i, j)

```
if i == 0 and j > 0
    if M[i][j].w == inf
        temp = 0
        for a = 0 to j
            temp+ = B[a].peso
            M[i][j].match = M[i][j].match ∪ (A[i], B[a])
        M[i][j].w = A[i].peso/temp
        return M[i][j]
    else
        return M[i][j]
else if i > 0 and j == 0
    if M[i][j].w == inf
        temp = 0
        for a = 0 to i
            temp+ = A[a].peso
            M[i][j].match = M[i][j].match ∪ (A[a], B[j])
        M[i][j].w = temp/B[j].peso
        return M[i][j]
    else
        return M[i][j]
else if i == 0 and j == 0
    if M[i][j].w == inf
        M[i][j].match = M[i][j].match ∪ (A[i], B[j])
        M[i][j].w = A[i].peso/B[j].peso
        return M[i][j]
    else
        return M[i][j]
```

```
else if i > 0 and j > 0
   if M[i][j].w == inf
      min_1, min_2, min_3 = inf
      temp, temp_1 = 0
      a_1, a_2 = 0
      for a = i − 1 to 0
         temp_1, temp = 0
         for b = a to i
            temp_1 += A[b].peso
         if M[a − 1][j − 1].w == inf
            M[a − 1][j − 1] = MIN-MATCHING-MEMOIZADO(A, B, a − 1, j − 1)
            temp = M[a − 1][j − 1].w + (temp_1/B[j].peso)
            if temp < min_1
               min_1 = temp
               a_1 = a
         else
            temp = M[a − 1][j − 1].w + (temp_1/B[j].peso)
            if temp < min_1
               min_1 = temp
               a_1 = a
      for a = j − 1 to 0
         temp_1, temp = 0
         for b = a to j
            temp_1 += B[b].peso
         if M[i − 1][a − 1].w == inf
            M[i−1][a−1] = MIN-MATCHING-MEMOIZADO(A, B, i−1, a−1) c1                1

            temp = M[i − 1][a − 1].w + (A[i].peso/temp_1)
            if temp < min_2
               min_2 = temp
               a_2 = a
         else
            temp = M[i − 1][a − 1].w + (A[i].peso/temp_1)
            if temp < min_2
               min_2 = temp
               a_2 = a
      if M[i][j].w = inf
         M[i][j] = MIN-MATCHING-MEMOIZADO(A, B, i − 1, j − 1)
         min_3 = M[i][j].w + (A[i].peso/B[j].peso)
      else
         min_3 = M[i][j].w + (A[i].peso/B[j].peso)
      cond = 0
      min = min_1
      if min > min_2
         min = min_2
```

```
72:         cond = 1
73:       else if min > min_3
74:         min = min_3
75:         cond = 2
76:       switch (cond)
77:       case 0:
78:         M[i][j].match = M[a_1 − 1][j − 1].match
79:         for a = a_1 to i
80:           M[i][j].match = M[i][j].match ∪ (A[a], B[j])
81:         break
82:       case 1:
83:         M[i][j].match = M[i − 1][a_2 − 1].match
84:         for a = a_2 to j
85:           M[i][j].match = M[i][j].match ∪ (A[i], B[a])
86:         break
87:       case 2:
88:         M[i][j].match = M[i − 1][j − 1].match
89:         M[i][j].match = M[i][j].match ∪ (A[i], B[j])
90:         break
91:       M[i][j].w = min
92:       return M[i][j]
93:     else
94:       return M[i][j]
```

Como podemos ver la ejecución del algoritmo anterior es lineal hasta el punto que se llama recursivamente para poder conseguir el valor de una entrada de la matriz M. Esto quiere decir que el tiempo de ejecución del algoritmo recae en la cantidad de llamadas a una línea de código que asigne un valor a la entrada de la matriz llamando recursivamente a la función (es decir, líneas 38,53,64). El número más alto de veces que se pueden llamar a estas líneas de código es $n \cdot m$, ya que la matriz M no tiene más espacio y el algoritmo verifica si un valor ya existe en dentro de cualquier entrada antes de asignar uno nuevo.

Entonces, podemos afirmar que el tiempo de ejecución de este algoritmo es $O(mn)$.

Pregunta 5 (Programación Dinámica)

Implementación: ver anexo o github Secuencias/Pregunta5.cpp

Para este algoritmo se volverá a usar el struct y la función que calcula los bloques igual que se usó en la anterior pregunta solo que esta vez se llamará MIN-MATCHING-PROD-DIN.

Al igual que el algoritmo anterior se partirá este algoritmo en dos algoritmos, el primero calculará los bloques dentro de los arreglos dados y el segundo hará el cálculo del matching.

M será la matriz donde se guarden los valores ya calculados. Este guardara los structs values que se mencionaron anteriormente. Además, ahora tendremos dos matrizes extra que calculan los valores se las sumas de los bloques, estas son M_1 y M_2. Se debe tomar en cuenta que $A[i].peso$ o $B[j].peso$ y las declaraciones parecidas calculan $A[i].second - B[i].first + 1$ o $B[j].second - B[j].first + 1$ respectivamente.

Require: Dos arreglos A y B con los pesos de los bloques y sus tamaños respectivos.
Ensure: Matching mínimo entre los dos arreglos y su peso.

MIN-MATCHING-PROD-DIN(A, B, i, j)

```
for _j = 0 to j
    temp = 0
    for a = 0 to _j
        temp+ = B[a].peso
        M[0][_j].match = M[0][_j].match ∪ (A[0], B[a])
    M[0][_j].peso = A[0].peso/temp
for _i = 0 to i
    temp = 0
    for a = 0 to _i
        temp+ = A[a].peso
        M[_i][0].match = M[_i][0].match ∪ (A[a], B[0])
    M[_i][0].peso = temp/B[0].peso
M[0][0].match = (A[0], B[0])
M[0][0].peso = A[0].peso/B[0].peso
for _i = 0 to i
    first_it = false
    for a = _i to 0
        if first_it == false
            M_1[_i][a] = A[a].peso
            first_it = true
        else
            M_1[_i][a] = M_1[_i][a + 1] + (A[a].peso)
for _j = 0 to j
    first_it = false
    for a = _j to 0
        if first_it == false
            M_2[_j][a] = B[a].peso
            first_it = true
        else
```

```
        M_2[_i][a] = M_2[_i][a + 1] + (B[a].peso)
for _i = 1 to i
  for _j = 1 to j
    min1, min2, min3 = inf
    temp, temp1 = 0
    a1, a2 = 0
    for a = _i − 1 to 0
      temp1 = M1[_i][a]
      temp = M[a − 1][_j − 1].peso + temp1/B[_j].peso
      if temp < min1
        min1 = temp
        a1 = a
    for a = _j − 1 to 0
      temp1 = M2[_j][a]
      temp = M[_i − 1][a − 1].peso + A[_i].peso/temp1
      if temp < min2
        min1 = temp
        a2 = a
    min3 = M[_i − 1][_j − 1].peso + A[_i].peso/B[_j].peso
    cond = 0
    min = min1
    if min > min2
      min = min2
      cond = 1
    else if min > min3
      min = min3
      cond = 2
    switch (cond)
    case 0:
      M[_i][_j].match = M[a1 − 1][_j − 1].match
      for a = a1 to i
        M[_i][_j].match = M[_i][_j].match ∪ (A[a], B[_j])
      break
    case 1:
      M[_i][_j].match = M[_i − 1][a2 − 1].match
      for a = a2 to j
        M[_i][j].match = M[_i][_j].match ∪ (A[_i], B[a])
      break
    case 2:
      M[_i][_j].match = M[_i − 1][_j − 1].match
      M[_i][_j].match = M[_i][_j].match ∪ (A[_i], B[_j])
      break
    M[_i][_j].w = min
    return M[_i][_j]
return M[i][j]
```

Este algoritmo es el mismo algoritmo anterior (Min-Matching-Memoizado) escrito de forma iterativa. La diferencia es que dentro de los loops de líneas 31 y 32, que es donde se calcula el min matching, es necesario tomar en cuenta todas las anteriores respuestas para poder calcular el matching con menor peso. Gracias a esto, tenemos otro loop dentro de esos dos loops de las lineas 31 y 32, esta es la diferencia más significativa entre este algoritmo y Min-Matching-Memoizado. Por ende, eso hace que el algoritmo sea de tiempo de ejecución cúbico. Entonces este algoritmo es $O(max(m,n)^3)$

3. Transformación de Imágenes

Pregunta 6 (Transformación Voraz)

Implementación: ver anexo o github Transformacion/Pregunta6.cpp

Para este algoritmo se usará la función MIN-MATCHING que se presentó anteriormente. Por ende, esa función no se volverá a copiar.

Require: Dos matrices A y B de 1's y 0's.
Ensure: Una transformación de A a B no necesariamente óptima.

TRANSFORMACION-VORAZ(A, B)	*cost*	*times*
$M = \emptyset$	c_1	1
$peso\_total = 0$	c_2	1
for $i = 0$ **to** $A.size$	c_3	p
$min\_matching =$ MIN-MATCHING$(A[i], B[i], A[i].size)$	$O(max(n, m))$	$p - 1$
$M = M \cup min\_matching.first$	c_4	$p - 1$
$peso\_total+ = min\_matching.second$	c_5	$p - 1$
return $(M, peso\_total)$		

Para el análisis de tiempo consideramos las matrices A y B, sean de la forma $(p \times q)$.

Sumando los tiempos en peor caso tenemos:

$$c_1 + c_2 + c_3 * p + O(max(n, m)) * (p - 1) + c_4 * (p - 1) + c_5 * (p - 1)$$

Sabemos que $O(max(n, m)) \leq q$, entonces:

$$c_1 + c_2 + c_3 * p + q * (p - 1) + c_4 * (p - 1) + c_5 * (p - 1)$$

De esto podemos afirmar que:

$$= O(pq)$$

Pregunta 7 (Transformación Prog. Dinámica)

Implementación: ver anexo o github Transformacion/Pregunta7.cpp

Este algoritmo es igual al anterior solo que esta vez se llama a la función MIN-MATCHING-PROG-DIN. Esta función es la que calcula los bloques de cada vector y después llama al a función con el mismo nombre pero que recibe los parametros de los bloques una vez calculados, que se presentó anteriormente. Por ende, esa función no se volverá a copiar.

Require: Dos matrices A y B de 1's y 0's.
Ensure: Una transformación de A a B no necesariamente óptima.

TRANSFORMACION-PROD-DIN(A, B)	*cost*	*times*
$M = \emptyset$	c_1	1
$peso\_total = 0$	c_2	1
for $i = 0$ **to** $A.size$	c_3	p
$min\_matching =$ MIN-MATCHING-PROG-DIN$(A[i], B[i], A[i].size)$	$O(max(n,m)^3)$	$p-1$
$M = M \cup min\_matching.first$	c_4	$p-1$
$peso\_total += min\_matching.second$	c_5	$p-1$
return $(M, peso\_total)$		

Sumando los tiempos en peor caso tenemos:

$$c_1 + c_2 + c_3 * p + O(max(n,m)^3) * (p-1) + c_4 * (p-1) + c_5 * (p-1)$$

Sabemos que $O(max(n,m)^3) \leq q^3$, entonces:

$$c_1 + c_2 + c_3 * p + q^3 * (p-1) + c_4 * (p-1) + c_5 * (p-1)$$

De esto podemos afirmar que:

$$= O(pq^3)$$

Pregunta 8 (Lectura de Imágenes)

Implementación: ver anexo o github Transformacion/Pregunta8.cpp

Pregunta 9 (Animación)

Implementación: ver anexo o github Transformacion/Pregunta9.cpp

Pregunta 10 (Transformación Prog. Dinámica Promedio)

Implementación: ver anexo o github Transformacion/Pregunta10.cpp

Para este algoritmo es necesario cambiar la función que calcula el min matching para poder tomar en cuenta el peso promedio. Por ende, a pesar de solo cambiar pocas línas de código comparado con la funcióñ Min-Matching-Prog-Din, se copiará la función de nuevo. Además, también son necesarios algunos cambios en la función de calcular los bloques, y entonces también se volverá a copiar esa función a pesar de ser pocos los cambios

Require: Dos arreglos de unos y ceros A y B con tamaño p
Ensure: Matching entre los dos arreglos y el peso promedio del mismo
Min-Matching-Promedio-Prog-Din(A, B, p)

M_1, M_2 // *Estos arreglos guardaran los bloques*
$n_1 = -1$
$m_2 = -1$
$\_i, \_j = 0$
for $i = 0$ **to** p
    if $A[i] == 1$ **and** $n_1 == -1$
        $n_1 = i$
    else if $A[i] == 0$ **and** $n_1 \neq -1$
        $M_1.push\_back(< n_1, i-1 >)$
        $n_1 = -1$
        $\_i+ = 1$
    if $i == p-1$ **and** $A[i] == 1$ **and** $n_1 \neq -1$
        $M_1.push\_back(< n_1, i-1 >)$
        $\_i+ = 1$
    else if $i == p-1$ **and** $A[i] == 1$ **and** $n_1 == -1$
        $M_1.push\_back(< i-1, i-1 >)$
        $\_i+ = 1$
    if $B[i] == 1$ **and** $m_1 == -1$
        $m_1 = i$
    else if $B[i] == 0$ **and** $m_1 \neq -1$
        $M_2.push\_back(< m_1, i-1 >)$
        $m_1 = -1$
        $\_j+ = 1$
    if $i == p-1$ **and** $B[i] == 1$ **and** $m_1 \neq -1$
        $M_2.push\_back(< m_1, i-1 >)$
        $\_j+ = 1$
    else if $i == p-1$ **and** $B[i] == 1$ **and** $m_1 == -1$
        $M_2.push\_back(< i-1, i-1 >)$
        $\_j+ = 1$
$sum\_M1, sum\_M2 = 0$
for $i = 0$ **to** $M1.size$
    $sum\_M1 = M1[i].peso$
for $i = 0$ **to** $M2.size$

$sum\_M2 = M2[i].peso$
$mu = sum\_M1/sum\_M2$
return MIN-MATCHING-PROMEDIO-PROG-DIN$(M_1, M_2, _i-1, _j-1, mu)$

Require: Dos arreglos A y B con los pesos de los bloques y sus tamaños respectivos.
Ensure: Matching promedio entre los dos arreglos y su peso.
MIN-MATCHING-PROMEDIO-PROD-DIN(A, B, i, j, mu)

for $\_j = 0$ **to** j
  $temp = 0$
  for $a = 0$ **to** $\_j$
    $temp+ = B[a].peso$
    $M[0][\_j].match = M[0][\_j].match \cup (A[0], B[a])$
  $M[0][\_j].peso = |(A[0].peso/temp) - mu|$
for $\_i = 0$ **to** i
  $temp = 0$
  for $a = 0$ **to** $\_i$
    $temp+ = A[a].peso$
    $M[\_i][0].match = M[\_i][0].match \cup (A[a], B[0])$
  $M[\_i][0].peso = |(temp/B[0].peso) - mu|$
$M[0][0].match = (A[0], B[0])$
$M[0][0].peso = |(A[0].peso/B[0].peso) - mu|$
for $\_i = 0$ **to** i
  $first\_it = false$
  for $a = \_i$ **to** 0
    if $first\_it == false$
      $M\_1[\_i][a] = A[a].peso$
      $first\_it = true$
    else
      $M\_1[\_i][a] = M\_1[\_i][a+1] + (A[a].peso)$
for $\_j = 0$ **to** j
  $first\_it = false$
  for $a = \_j$ **to** 0
    if $first\_it == false$
      $M\_2[\_j][a] = B[a].peso$
      $first\_it = true$
    else
      $M\_2[\_i][a] = M\_2[\_i][a+1] + (B[a].peso)$
for $\_i = 1$ **to** i
  for $\_j = 1$ **to** j
    $min_1, min_2, min_3 = inf$
    $temp, temp_1 = 0$
    $a_1, a_2 = 0$
    for $a = \_i - 1$ **to** 0
      $temp_1 = M_1[\_i][a]$
      $temp = M[a-1][\_j-1].peso + |(temp_1/B[\_j].peso) - mu|$

if $temp < min_1$
$min_1 = temp$
$a_1 = a$
for $a = \_j - 1$ **to** 0
$temp_1 = M_2[\_j][a]$
$temp = M[\_i - 1][a - 1].peso + |(A[\_i].peso/temp_1) - mu|$
if $temp < min_2$
$min_1 = temp$
$a_2 = a$
$min_3 = M[\_i - 1][\_j - 1].peso + |(A[\_i].peso/B[\_j].peso) - mu|$
$cond = 0$
$min = min_1$
if $min > min_2$
$min = min_2$
$cond = 1$
else if $min > min_3$
$min = min_3$
$cond = 2$
switch $(cond)$
case 0**:**
$M[\_i][\_j].match = M[a_1 - 1][\_j - 1].match$
for $a = a_1$ **to** i
$M[\_i][\_j].match = M[\_i][\_j].match \cup (A[a], B[\_j])$
break
case 1**:**
$M[\_i][\_j].match = M[\_i - 1][a_2 - 1].match$
for $a = a_2$ **to** j
$M[\_i][j].match = M[\_i][\_j].match \cup (A[\_i], B[a])$
break
case 2**:**
$M[\_i][\_j].match = M[\_i - 1][\_j - 1].match$
$M[\_i][\_j].match = M[\_i][\_j].match \cup (A[\_i], B[\_j])$
break
$M[\_i][\_j].w = min$
return $M[\_i][\_j]$
return $M[i][j]$

En las últimas dos funciónes es claro que los cambios hechos no afectan el tiempo de ejecución, ya que en la función de calcular los bloques tenemos dos sentencias **for** extras de un número menor a p, por ende el tiempo de ejecución se mantiene. Y en la función de calcular el min matching en sí solo se cambia el calculo aritmetico del peso y esto no añade ni quita nada al tiempo de ejecución.

Require: Dos matrices A y B de 1's y 0's.
Ensure: Una transformación de A a B no necesariamente óptima.
TRANSFORMACION-PROMEDIO-PROG-DIN(A, B) *cost* *times*

$M = \emptyset$ $\quad c_1 \quad 1$
$peso\_total = 0$ $\quad c_2 \quad 1$
for $i = 0$ **to** $A.size$ $\quad c_3 \quad p$
  $min\_matching =$
  $\textsc{Min-Matching-Promedio-Prog-Din}(A[i], B[i], A[i].size)$ $\quad O(max(n,m)^3) \quad p-1$
  $M = M \cup min\_matching.first$ $\quad c_4 \quad p-1$
  $peso\_total += min\_matching.second$ $\quad c_5 \quad p-1$
return $(M, peso\_total)$

Como tiempo de ejecución de el algoritmo, *Min-Matching-Promedio-Prog-Din*, que llama esta función es igual al de *Min-Matching-Prog-Din*, eso hace que el analisis de tiempo de este algoritmo sea exactamente igual al analisis de la pregunta 7, entonces podemos afirmar que el tiempo de ejecución es: $O(pq^3)$