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*
1: $M_1, M_2 = \emptyset$	$c1$	1
2: $n_1 = -1$	$c2$	1
3: $m_1 = -1$	$c3$	1
4: **for** $i = 0$ **to** p	$c4$	$p + 1$

5:	**if** $A[i] == 1$ **and** $n_1 == -1$	$c5$	p
6:	$n_1 = i$	$c6$	$p/2$
7:	**else if** $A[i] == 0$ **and** $n_1 \neq -1$	$c7$	$p/2$
8:	$M_1 = M_1 \cup (n_1, i-1)$	$c8$	$p/2$
9:	$n_1 = -1$	$c9$	$p/2$
10:	**if** $i == p-1$ **and** $A[i] == 1$ **and** $n_1 \neq -1$	$c10$	p
11:	$M_1 = M_1 \cup (n_1, i-1)$	$c11$	0
12:	**else if** $i == p-1$ **and** $A[i] == 1$ **and** $n_1 == -1$	$c12$	1
13:	$M_1 = M_1 \cup (i-1, i-1)$	$c13$	1
14:	**if** $B[i] == 1$ **and** $m_1 == -1$	$c14$	p
15:	$m_1 = i$	$c15$	$p/2$
16:	**else if** $B[i] == 0$ **and** $m_1 \neq -1$	$c16$	$p/2$
17:	$M_2 = M_2 \cup (m_1, i-1)$	$c17$	$p/2$
18:	$m_1 = -1$	$c18$	$p/2$
19:	**if** $i == p-1$ **and** $B[i] == 1$ **and** $m_1 \neq -1$	$c19$	p
20:	$M_2 = M_2 \cup (m_1, i-1)$	$c20$	0
21:	**else if** $i == p-1$ **and** $B[i] == 1$ **and** $m_1 == -1$	$c21$	1
22:	$M_2 = M_2 \cup (i-1, i-1)$	$c22$	1
23:	**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*
1: $S = \emptyset$	$c1$	1
2: $i, j, temp = 0$	$c2$	1
3: $peso, num = 0$	$c3$	1
4: **while** $i \leq M_1.length$ **and** $j \leq M_2.length$	$c4$	$max(n, m)$
5: **if** $j == M_2.length$	$c5$	1
6: $num = 0$	$c6$	1
7: **while** $i \leq M_1.length$	$c7$	$n - m + 1$
8: $S = S \cup (M_1[i], M_2[j])$	$c8$	$n - m$
9: $num+ = M_1[i].w$	$c9$	$n - m$
10: $i+ = 1$	$c10$	$n - m$
11: $peso+ = num/M_2[j].w$	$c11$	1
12: **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:

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

Por hipótesis de inducción:

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

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

1: *double* w
2: *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)

1: M_1, M_2 // *Estos arreglos guardaran los bloques*
2: $n_1 = -1$
3: $m_2 = -1$
4: $\_i, \_j = 0$
5: **for** $i = 0$ **to** p
6: **if** $A[i] == 1$ **and** $n_1 == -1$
7: $n_1 = i$
8: **else if** $A[i] == 0$ **and** $n_1 \neq -1$
9: $M_1.push\_back(< n_1, i-1 >)$
10: $n_1 = -1$
11: $\_i += 1$
12: **if** $i == p-1$ **and** $A[i] == 1$ **and** $n_1 \neq -1$
13: $M_1.push\_back(< n_1, i-1 >)$
14: $\_i += 1$
15: **else if** $i == p-1$ **and** $A[i] == 1$ **and** $n_1 == -1$
16: $M_1.push\_back(< i-1, i-1 >)$
17: $\_i += 1$
18: **if** $B[i] == 1$ **and** $m_1 == -1$
19: $m_1 = i$
20: **else if** $B[i] == 0$ **and** $m_1 \neq -1$
21: $M_2.push\_back(< m_1, i-1 >)$
22: $m_1 = -1$
23: $\_j += 1$
24: **if** $i == p-1$ **and** $B[i] == 1$ **and** $m_1 \neq -1$
25: $M_2.push\_back(< m_1, i-1 >)$
26: $\_j += 1$
27: **else if** $i == p-1$ **and** $B[i] == 1$ **and** $m_1 == -1$

28: $M_2.push\_back(< i-1, i-1 >)$
29: $\_j += 1$
30: **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.

	$\textsc{Opt}(A, B, i, j)$	*cost*	*times*
1:	*values* v	$c1$	1
2:	**if** $i == 0$ **and** $j > 0$	$c2$	1
3:	$temp = 0$	$c3$	1
4:	**for** $a = 0$ **to** j	$c4$	1
5:	$temp+ = B[a].peso$	$c5$	1
6:	$v.match = v.match \cup (A[i], B[a])$	$c6$	1
7:	$v.w = A[i].peso/temp$	$c7$	1
8:	**return** v		
9:	**else if** $i > 0$ **and** $j == 0$	$c8$	1
10:	$temp = 0$	$c9$	1
11:	**for** $a = 0$ **to** i	$c10$	1
12:	$temp+ = A[a].peso$	$c11$	1
13:	$v.match = v.match \cup (A[a], B[j])$	$c12$	1
14:	$v.w = temp/B[j].peso$	$c13$	1
15:	**return** v		
16:	**else if** $i == 0$ **and** $j == 0$	$c14$	1
17:	$v.match = v.match \cup (A[i], B[j])$	$c15$	1
18:	$v.w = A[i].peso/B[j].peso$	$c16$	1
19:	**return** v		
20:	**else if** $i > 0$ **and** $j > 0$	$c17$	1
21:	*values* min_1, min_2, min_3	$c18$	1
22:	$min_1.w = inf$	$c19$	1
23:	$min_2.w = inf$	$c20$	1
24:	$min_3.w = inf$	$c21$	1
25:	**for** $a = i - 1$ **to** 0	$c22$	1
26:	$temp_1 = 0$	$c23$	1
27:	$temp = 0$	$c24$	1
28:	**for** $b = a$ **to** i	$c25$	1
29:	$temp_1+ = A[b].peso$	$c26$	1
30:	$temp_2 = Opt(A, B, a - 1, j - 1)$	$T(A, B, a - 1, j - 1)$	
31:	$temp = temp_2.w + (temp_1/B[j].peso)$	$c28$	1
32:	**if** $temp < min_1.w$	$c29$	1
33:	$min_1.match = temp_2.match$	$c30$	1
34:	$min_1.w = temp$	$c31$	1
35:	$a_1 = a$	$c32$	1

36:	**for** $a = j - 1$ **to** 0	$c33$	1
37:	$temp_1 = 0$	$c34$	1
38:	$temp = 0$	$c35$	1
39:	**for** $b = a$ **to** j	$c36$	1
40:	$temp_1 += B[b].peso$	$c37$	1
41:	$temp_3 = Opt(A, B, i - 1, a - 1)$	$T(A, B, i - 1, a - 1)$	
42:	$temp = temp_3.w + (A[i].peso/temp_1)$	$c39$	1
43:	**if** $temp < min_2.w$	$c40$	1
44:	$min_2.match = temp_3.match$	$c41$	1
45:	$min_2.w = temp$	$c42$	1
46:	$a_2 = a$	$c43$	1
47:	$temp_4 = Opt(A, B, i - 1, j - 1)$	$T(A, B, i - 1, j - 1)$	
48:	$min_3.w = temp_4.w + A[i].peso/B[j].peso$	$c45$	1
49:	$min_3.match = temp_4.match$	$c46$	1
50:	$cond = 0$	$c47$	1
51:	*values min*	$c48$	1
52:	$min.w = min_1.w$	$c49$	1
53:	$min.match = min_1.match$	$c50$	1
54:	**for** $a = a_1$ **to** i	$c51$	1
55:	$min.match = min.match \cup (A[a], B[j])$	$c52$	1
56:	**if** $min.w > min_2.w$	$c53$	1
57:	$min.w = min_2.w$	$c54$	1
58:	$min.match = min_1.match$	$c55$	1
59:	**for** $a = a_1$ **to** i	$c56$	1
60:	$min.match = min.match \cup (A[i], B[a])$	$c57$	1
61:	**else if** $min.w > min_3.w$	$c58$	1
62:	$min.w = min_3.w$	$c59$	1
63:	$min.match = min_1.match$	$c60$	1
64:	$min.match = min.match \cup (A[i], B[j])$	$c61$	1
65:	**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)

```
1: if i == 0 and j > 0
2:    if M[i][j].w == inf
3:       temp = 0
4:       for a = 0 to j
5:          temp+ = B[a].peso
6:          M[i][j].match = M[i][j].match ∪ (A[i], B[a])
7:       M[i][j].w = A[i].peso/temp
8:       return M[i][j]
9:    else
10:      return M[i][j]
11: else if i > 0 and j == 0
12:   if M[i][j].w == inf
13:      temp = 0
14:      for a = 0 to i
15:         temp+ = A[a].peso
16:         M[i][j].match = M[i][j].match ∪ (A[a], B[j])
17:      M[i][j].w = temp/B[j].peso
18:      return M[i][j]
19:   else
20:      return M[i][j]
21: else if i == 0 and j == 0
22:   if M[i][j].w == inf
23:      M[i][j].match = M[i][j].match ∪ (A[i], B[j])
24:      M[i][j].w = A[i].peso/B[j].peso
25:      return M[i][j]
26:   else
27:      return M[i][j]
```

```
28: else if i > 0 and j > 0
29:   if M[i][j].w == inf
30:     min_1, min_2, min_3 = inf
31:     temp, temp_1 = 0
32:     a_1, a_2 = 0
33:     for a = i − 1 to 0
34:       temp_1, temp = 0
35:       for b = a to i
36:         temp_1 += A[b].peso
37:       if M[a − 1][j − 1].w == inf
38:         M[a − 1][j − 1] = Min-Matching-Memoizado(A, B, a − 1, j − 1)
39:         temp = M[a − 1][j − 1].w + (temp_1/B[j].peso)
40:         if temp < min_1
41:           min_1 = temp
42:           a_1 = a
43:       else
44:         temp = M[a − 1][j − 1].w + (temp_1/B[j].peso)
45:         if temp < min_1
46:           min_1 = temp
47:           a_1 = a
48:     for a = j − 1 to 0
49:       temp_1, temp = 0
50:       for b = a to j
51:         temp_1 += B[b].peso
52:       if M[i − 1][a − 1].w == inf
53:         M[i−1][a−1] = Min-Matching-Memoizado(A, B, i−1, a−1) c1          1

54:         temp = M[i − 1][a − 1].w + (A[i].peso/temp_1)
55:         if temp < min_2
56:           min_2 = temp
57:           a_2 = a
58:       else
59:         temp = M[i − 1][a − 1].w + (A[i].peso/temp_1)
60:         if temp < min_2
61:           min_2 = temp
62:           a_2 = a
63:     if M[i][j].w = inf
64:       M[i][j] = Min-Matching-Memoizado(A, B, i − 1, j − 1)
65:       min_3 = M[i][j].w + (A[i].peso/B[j].peso)
66:     else
67:       min_3 = M[i][j].w + (A[i].peso/B[j].peso)
68:     cond = 0
69:     min = min_1
70:     if min > min_2
71:       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 $O(mn)$.