

Proyecto

Guillermo Franco y Alessia Yi

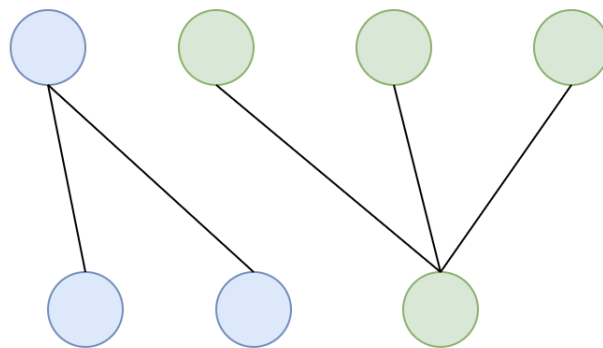
May 2020

1. Secuencias

Introducción

Presentaremos cuatro distintos algoritmos para resolver el problema de Secuencias. Implementaremos un algoritmo voraz, otro recursivo, otro memoizado y finalmente otro con programación dinámica. El objetivo del algoritmo consiste en calcular el mínimo peso de un matching de dos arreglos ya existentes y devolver dicho matching. Un matching corresponde a una transformación entre bloques de A hacia los bloques de B, tal que algunos bloques de A son divididos y otros bloques de son agrupados. El peso total de un matching es la suma de los pesos de todos los bloques que se dividen y los pesos todos los bloques que se agrupan, el peso de una división se calcula $\frac{|\text{bloque de } A|}{\text{suma de } |\text{bloques de } B_j|}$ y el peso de una agrupación de calcula $\frac{\text{suma de } |\text{bloques de } A_i|}{|\text{bloque de } B|}$.

Un ejemplo de división son los de color azul y un ejemplo de agrupación son los de color verde.



Pregunta 1 (Voraz)

Pseudocodigo

Primero, presentamos un subalgoritmo que calcula los bloques de los arreglos en tiempo lineal, ya que recorre dicho arreglo una sola vez. Cabe notar que este subalgoritmo se usará en todos los algoritmos futuros. Abajo de este se encuentra el algoritmo voraz principal.

Recibe: un arreglo de ceros y unos

Devuelve: un arreglo de tamaños de los bloques

GETBLOCKS(*array*)

```
1: i = 1
2: while i < array.size
3:   while array[i] == 0
4:     i = i + 1
5:   size = 0
6:   if array[i] == 1
7:     while array[i] == 1
8:       size = size + 1
9:       i = i + 1
10:  blocks.push(size)
11: return blocks
```

Recibe: dos arreglos de ceros y unos

Devuelve: un matching entre los dos arreglos y su peso

GREEDYMINMATCHING(*A*, *B*)

```
1: blocksA = GETBLOCKS(A)
2: blocksB = GETBLOCKS(B)
3: m = blocksA.size
4: n = blocksB.size
5: if m > n
6:   maxi = ratio = m/n
7:   j = 1
8:   for i = 1 TO m
9:     if i ≤ maxi
10:      num = num + blocksA[i]
11:    else
12:      w = w + num/blocksB[j]
13:      num = blocksA[i]
14:      maxi = maxi + ratio
15:      j = j + 1
16:  matching.push(i, j)
```

```

17:   $w = w + num/blocksB[j]$ 
18: else
19:   $max_j = ratio = n/m$ 
20:   $i = 1$ 
21:  for  $j = 1$  TO  $n$ 
22:    if  $j \leq max_j$ 
23:       $den = den + blocksB[j]$ 
24:    else
25:       $w = w + blocksA[i]/den$ 
26:       $den = blocksB[j]$ 
27:       $max_j = max_j + ratio$ 
28:       $i = i + 1$ 
29:     $matching.push(i, j)$ 
30:   $w = w + blocksA[i]/den$ 
31: return  $matching, w$ 

```

Analisis

Dejando de lado el cálculo de los bloques de cada arreglo, el tiempo de ejecución del algoritmo propuesto es de $O(max\{m, n\})$. Esto se debe al condicional de la línea 5. Si m , la cantidad de bloques del arreglo A , es mayor, se itera asignándole un solo *matching* a cada bloque de A . En el caso contrario, se hace lo mismo pero con los bloques de B . El calculo del peso del *matching* w se hace durante este proceso, por lo que solo le aumenta tiempo constante a la iteración.

Implementación

La implementación del algoritmo en C++ se puede encontrar en el siguiente [link](#).

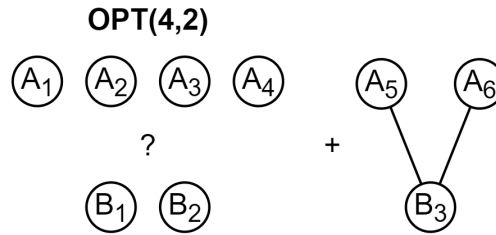
Pregunta 2 (Recurrencia)

$$\begin{aligned}
 OPT(i, j) &= \begin{cases} A_1/sum(B_1..B_j) & \text{si } i = 1 \\ sum(A_1..A_i)/B_1 & \text{si } j = 1 \\ OPT(i, j) = \min(Agrupar(i, j), Dividir(i, j)) & \text{en otro caso} \end{cases} \\
 Agrupar(i, j) &= \min(OPT(x, j-1) + sum(A_{x+1}..A_i)/B_j) : x \in \{1..i-1\} \\
 Dividir(i, j) &= \min(OPT(i-1, y) + A_i/sum(B_{y+1}..B_j)) : y \in \{1..j-1\}
 \end{aligned}$$

La siguiente imagen representa cómo funciona la recurrencia. Para calcular el $OPT(6,3)$, mostrado de color verde, se necesita evaluar todos los de color amarillo. La subrutina Agrupar consta de la fila evaluada, mientras que Dividir evalúa la columna.

Por ejemplo, cuando se calcula el $OPT(6,3)$, una de las combinaciones a evaluarse es $(4,2)$. Se calcula dicho OPT , y se le suma el peso del *matching* restante, como se ilustra en la figura a continuación.

		i						
		1	2	3	4	5	6	7
j	1							
	2							
	3							
	4							



Pregunta 3 (Recursivo)

Pseudocódigo

Este primer algoritmo es el principal. Transforma los arreglos a bloques y llama a la subrutina recursiva con los valores obtenidos. Esta subrutina, mostrada a continuación del primero, es la transformación de la fórmula de recurrencia presentada previamente a pseudocódigo.

Recibe: dos arreglos de ceros y unos

Devuelve: un matching de peso mínimo entre los dos arreglos, y el respectivo peso

RECURSIVEMINMATCHING (A, B)

- 1: $blocksA = \text{GETBLOCKS}(A)$
- 2: $blocksB = \text{GETBLOCKS}(B)$
- 3: $m = blocksA.size$
- 4: $n = blocksB.size$
- 5: **return** RECURSIVEMINMATCHINGREC($blocksA, blocksB, m, n$)

Recibe: dos arreglos de tamaños de bloques, y la cantidad de bloques a usar

Devuelve: un matching de peso mínimo entre los bloques usados y su peso

RECURSIVEMINMATCHINGREC ($blocksA, blocksB, i, j$)

```

1: if  $i = 1$ 
2:    $w = \text{blocksA}[1]/(\text{sum}(\text{blocksB}[1]..\text{blocksB}[j]))$ 
3:    $\text{matching.push}((1, 1)..(1, j))$ 
4:   return  $\text{matching}, w$ 
5: if  $j = 1$ 
6:    $w = \text{sum}(\text{blocksA}[1]..\text{blocksA}[i])/\text{blocksB}[1]$ 
7:    $\text{matching.push}((1, 1)..(i, 1))$ 
8:   return  $\text{matching}, w$ 
9:  $w = INF$ 
10: for  $x=1$  TO  $i-1$ 
11:    $\text{group}, w_g = \text{RECURSIVEMINMATCHINGREC}(\text{blocksA}, \text{blocksB}, x, j - 1)$ 
12:    $w_g = w_g + \text{sum}(\text{blocksA}[x + 1]..\text{blocksA}[i])/\text{blocksB}[j]$ 
13:    $\text{group.push}((x + 1, j)..(i, j))$ 
14:   if  $w_g < w$ 
15:      $w = w_g$ 
16:      $\text{matching} = \text{group}$ 
17: for  $y=1$  TO  $j-1$ 
18:    $\text{divide}, w_d = \text{RECURSIVEMINMATCHINGREC}(\text{blocksA}, \text{blocksB}, i - 1, y)$ 
19:    $w_d = w_d + \text{blocksA}[i]/\text{sum}(\text{blocksB}[y + 1]..\text{blocksB}[j])$ 
20:    $\text{divide.push}((i, y + 1)..(i, j))$ 
21:   if  $w_d < w$ 
22:      $w = w_d$ 
23:      $\text{matching} = \text{divide}$ 
24: return  $\text{matching}, w$ 

```

Análisis

Como la llamada recursiva siempre reduce los valores de m y n por lo menos en 1, el árbol de recursión tendrá, a lo máximo, altura de $\min(m, n)$. Además, para cualquier pares de m y n en los que ambos son mayores a 1, siempre llamarán dos veces a la recursión con valores $m - 1$ y $n - 1$, dejando de lado todas las otras llamadas. Esto quiere decir que, solo tomando una pequeña fracción de las llamadas recursivas, estas se doblan con cada nivel del árbol. Así, tenemos una cota muy inferior de $\Omega(2^{\min(m, n)})$.

Si evaluamos la recursión de forma más completa, una llamada con valores m y n resultan en $m + n - 2$ llamadas recursivas, ya que *Agrupar* y *Dividir* llaman a la recursión $m - 1$ y $n - 1$ respectivamente. Así, podemos afirmar que la cantidad de llamadas recursivas depende más del valor máximo entre m y n . Por lo tanto, es más preciso decir que la recursión tiene un tiempo de ejecución de $\Omega(\max(m, n)^{\min(m, n)})$, y eso es si el algoritmo en sí toma tiempo constante, lo cual está muy alejado de la realidad.

Implementación

La implementación del algoritmo en C++ se puede encontrar en el siguiente [link](#).

Pregunta 4 (Memoizado)

Pseudocódigo

Este primer algoritmo es el principal. Transforma los arreglos a bloques, crea los arreglos donde se guardarán los pesos con sus respectivos matching, llama a la subrutina recursiva. Esta subrutina, mostrada a continuación del primero, es el algoritmo recursivo transformado a memoizado. Los pesos y sus respectivos matching solo se calculan si es que no han sido obtenidos previamente.

Recibe: dos arreglos de ceros y unos

Devuelve: un matching de peso mínimo entre los dos arreglos, y el respectivo peso

MEMOMINMATCHING (A, B)

```
1:  $blocksA = \text{GETBLOCKS}(A)$ 
2:  $blocksB = \text{GETBLOCKS}(B)$ 
3:  $m = blocksA.size$ 
4:  $n = blocksB.size$ 
5:  $matching[m][n]$ 
6:  $memo[m][n] = INF$ 
7: return MEMOMINMATCHINGREC( $blocksA, blocksB, m, n, matching, memo$ )
```

Recibe: dos arreglos de tamaños de bloques, la cantidad de bloques a usar, y la memoria

Devuelve: un matching entre los bloques usados de peso mínimo

MEMOMINMATCHINGREC ($blocksA, blocksB, i, j, matching, memo$)

```
1: if  $memo[i][j] \neq INF$ 
2:   return  $matching[i][j], memo[i][j]$ 
3: if  $i = 1$ 
4:    $w = blocksA[1] / (sum(blocksB[1]..blocksB[j]))$ 
5:    $matching[i][j].push((1, 1)..(1, j))$ 
6:    $memo[i][j] = w$ 
7:   return  $matching[i][j], w$ 
8: if  $j = 1$ 
9:    $w = sum(blocksA[1]..blocksA[i]) / blocksB[1]$ 
10:   $matching[i][j].push((1, 1)..(i, 1))$ 
11:   $memo[i][j] = w$ 
12:  return  $matching[i][j], w$ 
13:  $w = INF$ 
14: for  $x=1$  TO  $i-1$ 
15:   $group, w_g = \text{MEMOMINMATCHINGREC}(blocksA, blocksB, x, j - 1, matching, memo)$ 
16:   $w_g = w_g + sum(blocksA[x + 1]..blocksA[i]) / blocksB[j]$ 
```

```

17:  group.push((x + 1, j)..(i, j))
18:  if  $w_g < w$ 
19:       $w = w_g$ 
20:      matching[i][j] = group
21:  for y=1 TO j-1
22:      divide,  $w_d$  = MEMOMINMATCHINGREC(blocksA, blocksB, i - 1, y, matching, memo)

23:   $w_d = w_d + \text{blocksA}[\textit{i}]/\text{sum}(\text{blocksB}[\textit{y} + 1]..\text{blocksB}[\textit{j}])$ 
24:  divide.push((i, y + 1)..(i, j))
25:  if  $w_d < w$ 
26:       $w = w_d$ 
27:      matching[i][j] = divide
28:  memo[i][j] =  $w$ 
29:  return matching[i][j],  $w$ 

```

Análisis

El tiempo de complejidad del algoritmo es $O(mn)$. Se crea una tabla *memo* de tamaño $m * n$ para almacenar los resultados calculados por primera vez. Es decir, la recurrencia solo demora $m * n$ llamadas en llenar la tabla, y después todas las llamadas son constantes.

Implementación

La implementación del algoritmo en C++ se puede encontrar en el siguiente [link](#).

Pregunta 5 (Programación dinámica)

Pseudocódigo

Este primer algoritmo es el principal. Transforma los arreglos a bloques, crea los arreglos donde se guardarán los pesos con sus respectivos matching y llama al algoritmo donde se comienza a llenar la tabla. Los pesos y sus respectivos matching se obtendrán de valor ya calculados anteriormente.

Recibe: dos arreglos de ceros y unos

Devuelve: un matching de peso mínimo entre los dos arreglos, y el respectivo peso

```

DPMINMATCHING (A, B)
1: blocksA = GETBLOCKS(A)
2: blocksB = GETBLOCKS(B)
3:  $m = \text{blocksA.size}$ 
4:  $n = \text{blocksB.size}$ 
5: matching[m][n]
6: for i=1 TO m

```

```

7:  peso[i] = [n]
8:  for j=1 TO n
9:      DPMINMATCHINGREC(blocksA, blocksB, i, j, matching, peso)
10: return peso[m][n]

```

Recibe: dos arreglos de tamaños de bloques, la cantidad de bloques a usar, y la memoria
Devuelve: un matching entre los bloques usados de peso mínimo

```

DPMINMATCHINGREC (blocksA, blocksB, i, j, matching, peso)
1: if i = 1
2:   w = blocksA[1]/(sum(blocksB[1]..blocksB[j]))
3:   matching[i][j].push((1, 1)..(1, j))
4:   peso[i][j] = w
5: if j = 1
6:   w = sum(blocksA[1]..blocksA[i])/blocksB[1]
7:   matching[i][j].push((1, 1)..(i, 1))
8:   peso[i][j] = w
9: else
10:  peso[i][j] = INF
11:  for x=1 TO i-1
12:    w = peso[x][j - 1]
13:    w = w + sum(blocksA[x + 1]..blocksA[i])/blocksB[j]
14:    newMatching = matching[x][j - 1]
15:    newMatching.push((x + 1, j)..(i, j))
16:    if w < peso[i][j]
17:      peso[i][j] = w
18:      matching[i][j] = newMatching
19:  for y=1 TO j-1
20:    w = peso[i - 1][y]
21:    w = w + blocksA[i]/sum(blocksB[y + 1]..blocksB[j])
22:    newMatching = matching[i - 1][y]
23:    newMatching.push((i, y + 1)..(i, j))
24:    if w < peso[i][j]
25:      peso[i][j] = w
26:      matching[i][j] = newMatching

```

Análisis

Se crea una tabla *matching* de tamaño $m * n$ para almacenar los resultados. Esta se va llenando cada casilla de arriba a abajo y de izquierda a derecha calculando cada nuevo valor con los resultados anteriores de la misma tabla. Por lo que tenemos una complejidad de $O(mn)$ al terminar de recorrer toda la tabla.

Implementación

La implementación del algoritmo en C++ se puede encontrar en el siguiente [link](#).

2. Transformación de imágenes

Introducción

El objetivo de estos algoritmos son encontrar el peso mínimo de una transformación de dos matrices dadas. Una transformación consiste en un conjunto de matchings y el peso de esta transformación es calculado por la suma del peso de todos los matchings.

Pregunta 6 (Voraz)

Pseudocódigo

Acá se toma el algoritmo voraz planteado previamente, y se aplica por cada fila de las dos matrices, guardando los pesos de los matching en una suma total.

Recibe: dos matrices de ceros y unos

Devuelve: una transformación entre las dos matrices y su peso

GREEDYMINTRANSFORM(A, B)

```
1:  $m = A.size$ 
2: for  $i=1$  TO  $m$ 
3:    $matching[i], w_g = \text{GREEDYMINMATCHING}(A[i], B[i])$ 
4:    $w+ = w_g$ 
5: return  $matching, w$ 
```

Recibe: un arreglo de ceros y unos

Devuelve: un arreglo de tamaños de los bloques

GETBLOCKS($array$)

```
1:  $i = 1$ 
2: while  $i < array.size$ 
3:   while  $array[i] == 0$ 
4:      $i = i + 1$ 
5:    $size = 0$ 
6:   if  $array[i] == 1$ 
7:     while  $array[i] == 1$ 
8:        $size = size + 1$ 
9:        $i = i + 1$ 
10:   $blocks.push(size)$ 
```

11: **return** *blocks*

Recibe: dos arreglos de ceros y unos

Devuelve: un matching entre los dos arreglos y su peso

GREEDYMINMATCHING(*A*, *B*)

```
1: blocksA = GETBLOCKS(A)
2: blocksB = GETBLOCKS(B)
3: m = blocksA.size
4: n = blocksB.size
5: if m > n
6:   maxi = ratio = m/n
7:   j = 1
8:   for i = 1 TO m
9:     if i ≤ maxi
10:      num = num + blocksA[i]
11:    else
12:      w = w + num/blocksB[j]
13:      num = blocksA[i]
14:      maxi = maxi + ratio
15:      j = j + 1
16:      matching.push(i, j)
17:    w = w + num/blocksB[j]
18:  else
19:    maxj = ratio = n/m
20:    i = 1
21:    for j = 1 TO n
22:      if j ≤ maxj
23:        den = den + blocksB[j]
24:      else
25:        w = w + blocksA[i]/den
26:        den = blocksB[j]
27:        maxj = maxj + ratio
28:        i = i + 1
29:        matching.push(i, j)
30:      w = w + blocksA[i]/den
31: return matching, w
```

Análisis

Al tener dos matrices de tamaño $p * q$, el tiempo de complejidad de todo el algoritmo aumenta a $O(pq)$ ya que la función extra *GreedyMinMatchingMatrix* recorre cada fila de ambas matrices y calculando el matching, con un algoritmo voraz, en cada una de ellas.

Implementación

La implementación del algoritmo en C++ se puede encontrar en el siguiente [link](#).

Pregunta 7 (Programación dinámica)

Pseudocódigo

Se añade un algoritmo adicional a los de la pregunta 5, el cual recorre cada línea de las matrices y hace el matching con el algoritmo de programación dinámica entre ellas retornando la suma del peso de todos los matchings.

Recibe: dos matrices de ceros y unos

Devuelve: una transformación entre las dos matrices y su peso

DPMINTRANSFORM(A, B)

```
1:  $m = A.size$ 
2: for  $i=1$  TO  $m$ 
3:    $matching[i], w_g = DPMINMATCHING(A[i], B[i])$ 
4:    $w+ = w_g$ 
5: return  $matching, w$ 
```

Recibe: dos arreglos de ceros y unos

Devuelve: un matching de peso mínimo entre los dos arreglos, y el respectivo peso

DPMINMATCHING (A, B)

```
1:  $blocksA = GETBLOCKS(A)$ 
2:  $blocksB = GETBLOCKS(B)$ 
3:  $m = blocksA.size$ 
4:  $n = blocksB.size$ 
5:  $matching[m][n]$ 
6: for  $i=1$  TO  $m$ 
7:    $peso[i] = [n]$ 
8:   for  $j=1$  TO  $n$ 
9:     DPMINMATCHINGREC( $blocksA, blocksB, i, j, matching, peso$ )
10: return  $peso[m][n]$ 
```

Recibe: dos arreglos de tamaños de bloques, la cantidad de bloques a usar, y la memoria

Devuelve: un matching entre los bloques usados de peso mínimo

DPMINMATCHINGREC ($blocksA, blocksB, i, j, matching, peso$)

```
1: if  $i = 1$ 
```

```

2:   $w = \text{blocksA}[1] / (\text{sum}(\text{blocksB}[1].. \text{blocksB}[j]))$ 
3:   $\text{matching}[i][j].\text{push}((1, 1)..(1, j))$ 
4:   $\text{peso}[i][j] = w$ 
5:  if  $j = 1$ 
6:     $w = \text{sum}(\text{blocksA}[1].. \text{blocksA}[i]) / \text{blocksB}[1]$ 
7:     $\text{matching}[i][j].\text{push}((1, 1)..(i, 1))$ 
8:     $\text{peso}[i][j] = w$ 
9:  else
10:    $\text{peso}[i][j] = INF$ 
11:   for  $x=1$  TO  $i-1$ 
12:      $w = \text{peso}[x][j - 1]$ 
13:      $w = w + \text{sum}(\text{blocksA}[x + 1].. \text{blocksA}[i]) / \text{blocksB}[j]$ 
14:      $\text{newMatching} = \text{matching}[x][j - 1]$ 
15:      $\text{newMatching}.\text{push}((x + 1, j)..(i, j))$ 
16:     if  $w < \text{peso}[i][j]$ 
17:        $\text{peso}[i][j] = w$ 
18:        $\text{matching}[i][j] = \text{newMatching}$ 
19:   for  $y=1$  TO  $j-1$ 
20:      $w = \text{peso}[i - 1][y]$ 
21:      $w = w + \text{blocksA}[i] / \text{sum}(\text{blocksB}[y + 1].. \text{blocksB}[j])$ 
22:      $\text{newMatching} = \text{matching}[i - 1][y]$ 
23:      $\text{newMatching}.\text{push}((i, y + 1)..(i, j))$ 
24:     if  $w < \text{peso}[i][j]$ 
25:        $\text{peso}[i][j] = w$ 
26:        $\text{matching}[i][j] = \text{newMatching}$ 

```

Análisis

Al tener dos matrices de tamaño $p * q$, el tiempo de complejidad de todo el algoritmo aumenta a $O(pq^2)$ ya que la función extra *DPMinMatchingMatrix* recorre cada fila de ambas matrices y calculando el matching, con un algoritmo de programación dinámica, en cada una de ellas.

Implementación

La implementación del algoritmo en C++ se puede encontrar en el siguiente [link](#).

Pregunta 8 (Lectura de imágenes)

Pseudocódigo

Lee una imagen y como arrays de chars, cada componente R, G y B es representado por un char. Luego estos chars son convertidos a ints y se multiplica con una constante para calcular la luminancia y se compara con el umbral y se define si es 1 o 0.

Recibe: una imagen, el umbral, y las tres constantes de R, G y B

Devuelve: una matriz de 0 y 1

```
GETMATRIXFROMPNG(file, umbral, R, G, B)
1: image = load(file)
2: matrix[height][width]
3: for i=1 TO height
4:   for j=1 TO width
5:     pixelOffset = image + (j + width * i) * 3
6:     Y = R * pixelOffset[0] + G * pixelOffset[1] + B * pixelOffset[2]
7:     if Y > umbral
8:       vec[j] = 0
9:     else
10:      vec[j] = 1
11:    matrix[i] = vec
12: return matrix
```

Implementación

La implementación del algoritmo en C++ se puede encontrar en el siguiente [link](#).

Pregunta 9 (Animación)

Pseudocódigo

Se calcula el promedio de cada pixel de amnas imagenes para crear la imagen resultante, y se recibe un parametro extra para definir cual algoritmo se quiere utilizar para realizar el matching.

Recibe: Dos imágenes y un parámetro para elección de algoritmo

Devuelve: El peso del matching y la transformación de la imagen

```
TRANSFORMIMAGES(imageA, imageB, method)
1: matrixA = load(imageA)
2: matrixB = load(imageB)
3: if method == 0
4:   peso = GreedyMinTransform(matrixA, matrixB, transform)
5: else if method == 1
6:   peso = DPMinTransform(matrixA, matrixB, transform)
7: else if method == 2
8:   peso = DPMinTransformMean(matrixA, matrixB, transform)
9: GETTRANSITION(IMAGEA, IMAGEB)
10: return peso
```

Recibe: Dos imágenes

Devuelve: Una transformación de imagen

GETTRANSITION(*imageA*, *imageB*)

```
1: A = load(imageA)
2: B = load(imageB)
3: for i = 0 TO A.size
4:   transition[i] = (A[i] + B[i])/2
5: write_jpg(transition)
```

Implementación

La implementación del algoritmo en C++ se puede encontrar en el siguiente [link](#).

Pregunta 10 (Programación dinámica)

Pseudocódigo

Se agrega una resta luego de cada calculo de agrupación y división $\bar{w} = w - \mu$ y se utiliza el algoritmo de la pregunta 7.

Recibe: dos matrices de ceros y unos

Devuelve: una transformación entre las dos matrices y su peso

DPMINTRANSFORMMEAN(*A*, *B*)

```
1: m = A.size
2: for i=1 TO m
3:   matching[i], wg =DPMINMATCHING(A[i], B[i])
4:   w += wg
5: return matching, w
```

Recibe: dos arreglos de ceros y unos

Devuelve: un matching de peso mínimo entre los dos arreglos, y el respectivo peso

DPMINMATCHINGMEAN (*A*, *B*)

```
1: blocksA = GETBLOCKS(A)
2: blocksB = GETBLOCKS(B)
3: m = blocksA.size
4: n = blocksB.size
5: matching[m][n]
6:  $\mu = \text{sum}(\text{blocksA}[i]) / \text{sum}(\text{blocksB}[j])$ 
7: for i=1 TO m
```

```

8:  peso[i] = [n]
9:  for j=1 TO n
10:    DPMINMATCHINGMEANREC(blocksA,blocksB,i,j,matching,peso, $\mu$ )
11: return peso[m][n]

```

Recibe: dos arreglos de tamaños de bloques, la cantidad de bloques a usar, y la memoria
Devuelve: un matching entre los bloques usados de peso mínimo

DPMINMATCHINGMEANREC (*blocksA*,*blocksB*,*i*,*j*,*matching*,*peso*, μ)

```

1: if i = 1
2:   w = blocksA[1]/(sum(blocksB[1]..blocksB[j]))
3:   matching[i][j].push((1,1)..(1,j))
4:   peso[i][j] = |w -  $\mu$ |
5: if j = 1
6:   w = sum(blocksA[1]..blocksA[i])/blocksB[1]
7:   matching[i][j].push((1,1)..(i,1))
8:   peso[i][j] = |w -  $\mu$ |
9: else
10:  peso[i][j] = INF
11:  for x=1 TO i-1
12:    w = peso[x][j - 1]
13:    w = w + sum(blocksA[x + 1]..blocksA[i])/blocksB[j]
14:    newMatching = matching[x][j - 1]
15:    newMatching.push((x + 1,j)..(i,j))
16:     $\bar{w}$  = |w -  $\mu$ |
17:    if  $\bar{w}$  < peso[i][j]
18:      peso[i][j] =  $\bar{w}$ 
19:      matching[i][j] = newMatching
20:  for y=1 TO j-1
21:    w = peso[i - 1][y]
22:    w = w + blocksA[i]/sum(blocksB[y + 1]..blocksB[j]))
23:    newMatching = matching[i - 1][y]
24:    newMatching.push((i,y + 1)..(i,j))
25:     $\bar{w}$  = |w -  $\mu$ |
26:    if  $\bar{w}$  < peso[i][j]
27:      peso[i][j] =  $\bar{w}$ 
28:      matching[i][j] = newMatching

```

Análisis

El tiempo de ejecución es igual que el de la pregunta 7 $O(pq^2)$, ya que solo se agrega una resta al peso cada vez que se calcula el peso de una división o una agrupación,

Implementación

La implementación del algoritmo en C++ se puede encontrar en el siguiente [link](#).

Repositorio

El repositorio con todos los algoritmos se encuentran en el siguiente [link](#).