Proyecto Parcial
Analisis y Diseño de Algoritmos
Universidad de Ingenieria y Tecnologia

Luis Ponce, Dario Toribio, Jonathan Hoyos

16 de junio de 2020

Índice

1. Implementacion del Codigo

El codigo se implemento en C++ y cuenta de 6 archivos.

- **main.cpp** ⇒ Inicio del programa
- **programa.h** ⇒ Contiene la clase Programa. Encargada de recibir los inputs y parsearlos para luego guardarlos en un atributo de tipo Secuencia.
- **secuencia.h** ⇒ Contiene la clase Secuencias. Encargada de almacenar los vectores con pesos y ejecutar el algoritmo voraz, recursivo y memorizado.
- **Makefile** ⇒ Makefile para correr el proyecto.
- **run.sh** ⇒ Script que corre el Makefile y corre el ejecutable obtenido.
- **parser.py** ⇒ Codigo util para generar la tabla de resultados
- **tests** ⇒ Una lista de 50 tests que el programa correra.
- **resultados** ⇒ Resultados de la ejecucion Voraz, recursiva y memoizada. Ademas cuenta los tiempos en ms de demora.

1.1. Variables importantes

Lista de variables que seran utiles en el desarrollo del proyecto.

- **A_vector** ⇒ Almacena el input original del primero arreglo. *Ejm:* {*1,1,0,1,0,1,1,0,0,1*}
- **B_vector** ⇒ Almacena el input original del segundo arreglo. *Ejm:* {*1,1,0,1,1,0,0,0,1,1*}
- **A_weights** ⇒ Almacena los pesos (cantidad de 1's juntos) del primer arreglo. *Ejm:* {*2,1,2,1*}
- **B_weights** ⇒ Almacena los pesos (cantidad de 1's juntos) del segundo arreglo. *Ejm:* {*2,2,2*}
- **A_acumulado** ⇒ Almacena los pesos acumulados del primero arreglo. Servira para no realizar calculos innecesarios en las funciones *OPT(i,j). Ejm :*{*2,3,5,6*}
- **B_acumulado** ⇒ Almacena los pesos acumulados del segundo arreglo. Servira para no realizar calculos innecesarios en las funciones *OPT(i,j). Ejm :*{*2,4,6*}

2. Solucion Voraz

2.1. Pseudocodigo

Recibe: A_weights, B_weights
Devuelve: Suma de Agrupaciones y Divisiones.

VORAZ_FUNC($A\_weights, B\_weights$)
1: **return** VORAZ($A\_weights.size(), B\_weights.size()$)

VORAZ(i, j)
1: **if** $i == 1$ and j!=1
2: **return** A_weights[i]/Sum{B_weights[$j : B\_weights.size()$]}
3: **if** $j == 1$ and i!=1
4: **return** Sum{A_weights[$i : A\_weights.size()$]}/B_weights[j]
5: **if** $i == 1$ and $j == 1$
6: **return** A_weights[i]/B_weights[j]
7: Auxiliar = B_weights[j]
8: **if** A_weights[i]/B_weights[j] > 1
9: **while** A_weights[i]/B_weights[j] > 1
10: $j+ = 1$
11: **if** $j == B\_weights.size()$
12: $j- = 1$
13: Auxiliar += B_weights[j]
14: **return** A_weights[i]/B_weights[j] + voraz($i + 1$,$j + 1$)

2.2. Justificacion Voraz

Este algoritmo no siempre devolvera el valor mas optimo, esto debido a que el ultimo match siemrpe es el peor, y siempre podrán haber mejores matches finales. Estamos encontrando minimos locales. La forma de encontrar los verdaderos minimos es aplicando un algoritmo Recursivo, Memoizado o Dinamico.

2.3. Complejidad de tiempo

El algoritmo Voraz realiza una recursion aumentando los i, j en 1 n caso la division del $A\_weights[i]/B\_weights[j] \leq 1$.
En caso no sea menor se dividira $A\_weights[i]$ entre la suma de los $B\_weights[j \dots k]$ hasta que sea menor o igual a 1.
Finalmente si llega al extremo del minimo, ejecutara un FOR para encontrar su suma de pesos hasta el final, $Arr\_max[pos\_actual : pos\_final]$ y realizar la division o agrupacion, dependiendo cual es el vector con mayor tamaño.
De esta manera queda demostrado que el algoritmo tiene complejidad de tiempo $O(Max\{A, B\})$

3. Solucion Recursiva

3.1. Pseudocodigo

Recibe: A_weights, B_weights
Devuelve: Minima Suma de Agrupaciones y Divisiones.

RECURSIVO_FUNC($A\_weights, B\_weights$)
1: **return** OPT($A\_weights.size(), B\_weights.size()$)

OPT(i, j)
1: **if** $i == 1$ and j!=1
2: **return** A_weights[i]/B_acumulado[j]
3: **if** $j == 1$ and i!=1
4: **return** A_acumulado[i]/B_weights[j]
5: **if** $i == 1$ and $j == 1$
6: **return** A_weights[i]/B_weights[j]
7: $Valores$[:]
8: **for** $k = j$ **to** 2
9: $Valores \Leftarrow (A\_weights$[i]/($B\_acumulado$[j] − $B\_acumulado$[k-1]) + OPT(i-1,k-1))
10: **for** $k = i$ **to** 2
11: Valores⇐ (($A\_acumulado$[i] − $A\_acumulado$[k-1])/$B\_weights$[j] + OPT(k-1,j-1))
12: **return** $minimo(Valores)$

3.1.1. Explicacion del pseudocodigo

Funcionamiento de los dos *for* en las lineas $8 - 11$ en el pseudocodigo **OPT**

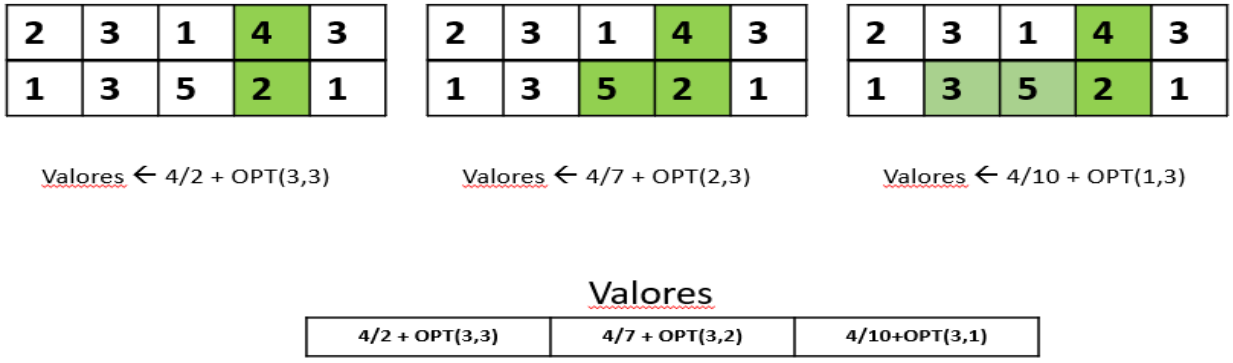

Figura 1: Datos guardados por Valores en el primer for

El fin es calcular todos los posibles casos que se podrian encontrar si utilizamos esos i,j. Luego de calcular dichos valores se retorna el minimo de este vector, el cual contiene la minima suma al usar agrupamientos y divisiones en esos indices.

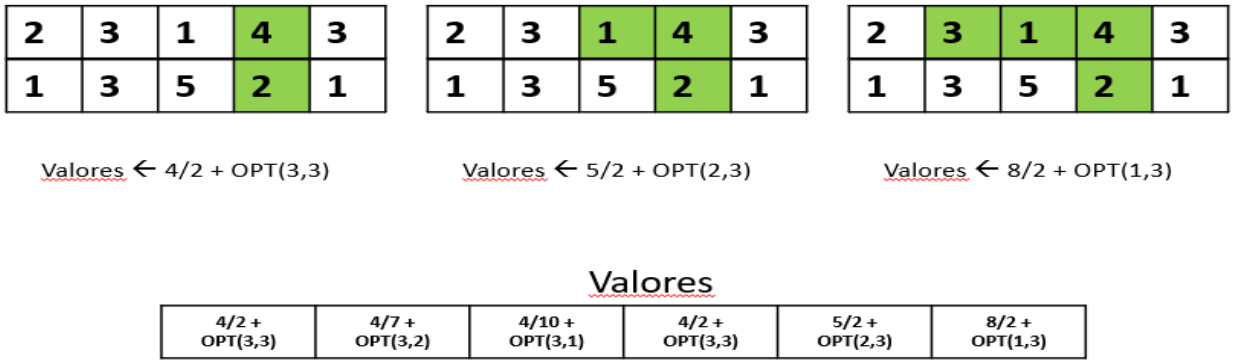

Figura 2: Datos guardados por Valores en el segundo for

3.2. Recursion

- A_w ⇒ A_weights
- A_acu ⇒ A_acumulado
- B_w ⇒ B_weights
- B_acu ⇒ B_acumulado

$$OPT(i,j) = \begin{cases} A\_w[i]/B\_w[j] & \textbf{i = 1 , j = 1} \\ A\_acu[i]/B\_w[j] & \textbf{i!= 1, j = 1} \\ A\_w[i]/B\_acu[j] & \textbf{i = 1 , j!= 1} \\ \\ \textbf{Min\{} & \\ A\_w[i]/(B\_acu[j] - B\_acu[j-1]) + OPT(i-1, j-1), & \\ A\_w[i]/(B\_acu[j] - B\_acu[j-2]) + OPT(i-1, j-2), & \\ \ldots\ldots & \\ A\_w[i]/(B\_acu[j] - B\_acu[1]) + OPT(i-1, 1), & \\ & \textbf{caso contrario} \\ (A\_acu[i] - A\_acu[i-1])/B\_w[j] + OPT(i-1, j-1), & \\ (A\_acu[i] - A\_acu[i-2])/B\_w[j] + OPT(i-2, j-1), & \\ \ldots\ldots & \\ (A\_acu[i] - A\_acu[1])/B\_w[j] + OPT(1, j-1), & \\ \textbf{\}} & \end{cases}$$

Demostraremos por induccion que $OPT(n,m) \geq ck^{max(n,m)}$.
Primero asumiremos a los tres casos no recursivos que son iguales a una variable d. Entonces:

Para $k = \frac{3}{2}$ y $c \leq \frac{2d}{3}$.
Cuando ($i = 1$ y $j = 1$),($i = 1$ y $j! = 1$) o ($i! = 1$ y $j = 1$) y tenemos que $OPT(i, j) = d \geq c$

$$\begin{aligned} OPT(i,j) &= q_1 + Min\{(OPT(i-1,j-1), OPT(i-2,j-1), \ldots, OPT(1,j-1) \\ & \quad OPT(i-1,j-1), OPT(i-1,j-2), \ldots, OPT(i-1,1)\} \\ OPT(i,j) &\geq q_1 + Min\{ck^{Max(i-1,j-1)}, ck^{Max(i-2,j-1)}, \ldots, ck^{Max(1,j-1)}, \\ & \quad ck^{Max(i-1,j-1)}, ck^{Max(i-1,j-2)}, \ldots, ck^{Max(i-1,1)}\} \end{aligned}$$

Debido a la funcion recursiva, se tienen que realizar dichos calculos de $OPT()$:

$$\begin{aligned} &= q_1 + ck^{Max(i-1,j-1)} + ck^{Max(i-1,j-2)} + \cdots + ck^{Max(i-1,1)} \ldots \\ &= q_1 + ck^{Max(i,j)}(\frac{1}{k} + \frac{1}{k^2} + \cdots + \frac{1}{k^{Max(i,j)-Min(i,j)}} + \frac{1}{k^{Max(i,j)-Min(i,j)}} + \cdots + \frac{1}{k} + \frac{1}{k} + \cdots + \frac{1}{k}) \end{aligned}$$

Reemplazamos k con $\frac{3}{2}$ y nos damos cuenta que:

$$OPT(i,j) \geq q_1 + ck^{Max(i,j)} * 1$$

Por lo tanto, el tiempo de ejecucion del algoritmo recursivo es de $\Omega(2^n)$

4. Solucion Memoizada

4.1. Pseudocódigo

Recibe: A_weights, B_weights
Devuelve: Minima Suma de Agrupaciones y Divisiones.

MEMOIZADO(i, j)
1: **if** $Matriz[i][j]$!= 0
2: **return** Matriz[i][j]
3: $Valores$[:]
4: **for** $k = j$ **to** 2
5: $Valores \Leftarrow (A\_weights$[i]$/(B\_acumulado$[j] $- B\_acumulado$[k-1]$) + OPT$(i-1,k-1))
6: **for** $k = i$ **to** 2
7: Valores$\Leftarrow ((A_acumulado$[i] $- A\_acumulado$[k-1]$)/B\_weights$[j] $+ OPT$(k-1,j-1))
8: Matriz[i][j] = minimo(Valores)
9: **return** $minimo(Valores)$

CASOBASE()
1: **for** $i = 1$ **to** A_weights.size()
2: $Matriz[i][1] = A\_acumulado$[i]$/B\_weights$[1]
3: **for** $i = 1$ **to** B_weights.size()
4: $Matriz$[1][i] $= A\_weights$[1]$/B\_acumulado$[i]

MEMOIZADO$_{func}$()
1: CasoBase()
2: **return** $Memoizado(A\_weights.size(), B\_weights.size())$

4.1.1. Explicación del pseudocódigo

La lógica aplicada en este algoritmo es la misma que en el algoritmo recursivo, solo que aquí guardamos los resultados que nos salgan para usarlos luego.
Primero, inicializamos el caso base, que viene siendo la primera fila y columna de la matriz, la cual al inicio esta llena de ceros. Una vez inicializado, aplicamos el algoritmo del recursivo, solo que aquí, antes de devolver el resultado, lo guardamos en su posición respectiva de la matriz. Por lo que, cada vez que se entre a la función recursivamente, se preguntara si esa posición 0, si es asi, continua el algoritmo, pero si este no es 0 significa que este valor ya fue calculado previamente.

4.2. Complejidades

Como podemos visualizar, el árbol recursivo tiene unos pocos niveles de más, esto es porque siempre opera todo lo que necesita, así ya lo haya calculado previamente.

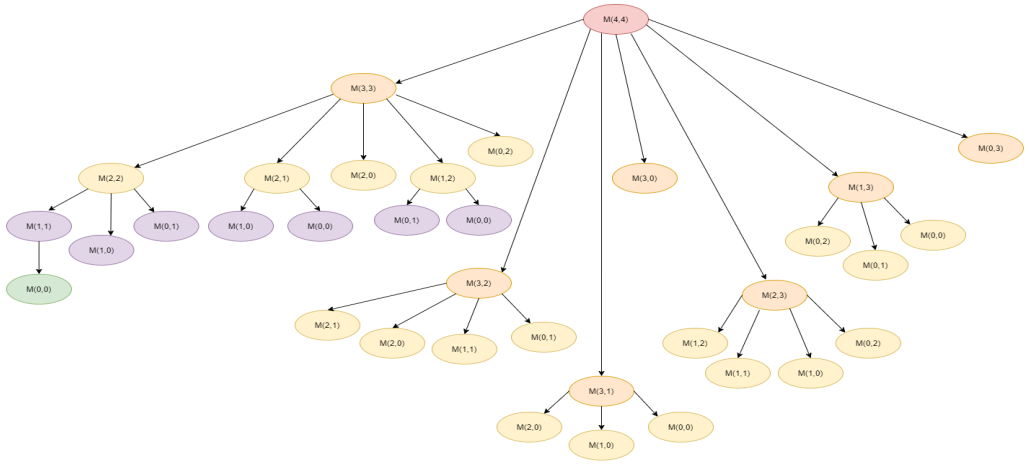

Figura 3: Árbol de recursión del algoritmo Memoizado para 2 arreglos de 4 valores

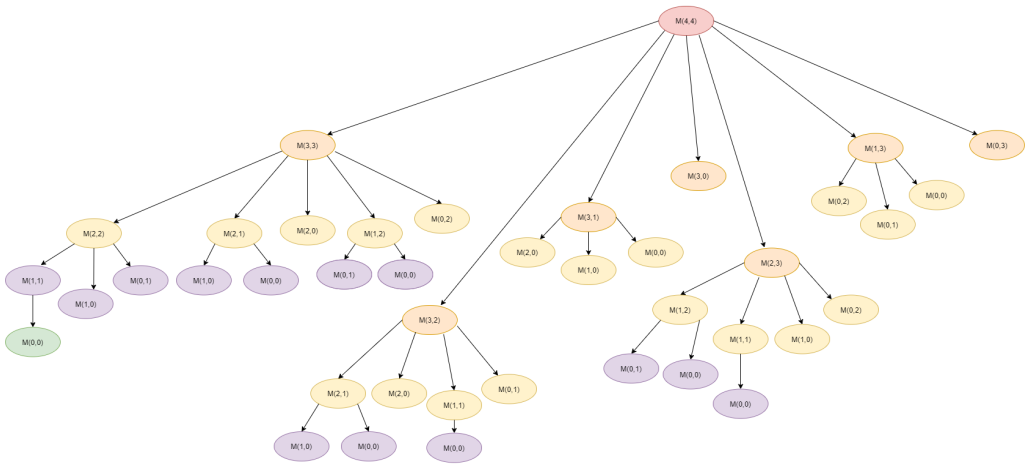

Figura 4: Árbol de recursión del algoritmo Memoizado para 2 arreglos de 4 valores

4.2.1. Complejidad Espacial

Todos los valores se guardan en una matriz, teniendo siempre el valor óptimo para la submatriz i, j. Por lo que al final el tamaño de esta matriz sería MxN.

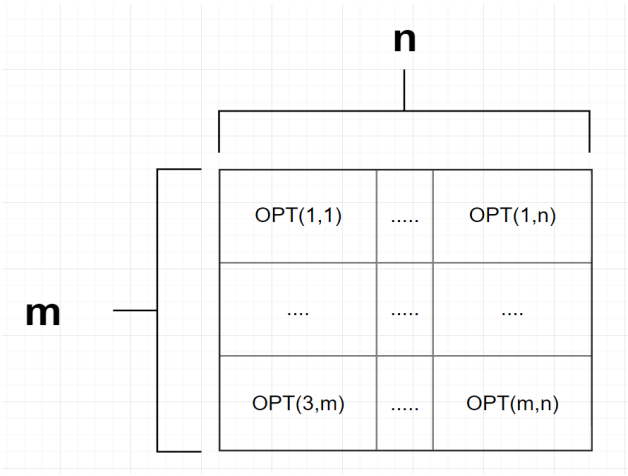

Figura 5: Complejidad de espacio

4.2.2. Complejidad de tiempo

Al inicio tienes la recursión T(m, n), el cuál comienza a generar los valores de la matriz en la primera recorrida, luego el resto de casillas se va llenando de acuerdo a lo que se necesita mas adelante. Esto hace que al calcular ciertas casillas, no tengamos que resolver OPT's anteriores porque estos a fueron guardados en la matriz. El máximo recorrido que hacemos sería desde el OPT(1,1) hasta el OPT(m,n), por lo que esta complejidad viene siendo dada por O(m*n)

5. Github

Link del Proyecto