FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico Nº2

El Experimento Cientifico y *El Almacen*

Children of Dijkstra

Fecha entrega: 5/12/2022 (Tercera entrega)

[75.29] Teoria de Algoritmos I

Segundo cuatrimestre 2022

Alumno

Bauni	Chiara	102981
Leloutre	Daniela	96783
Guglielmone	Lionel	96963
Leon	Agustina	102208
Buceta	Belen	102121

Índice

1. El experimento científico

Se realiza un experimento de conductividad de un nuevo material en aleación con otro. Se formaron muestras numeradas de 1 a n. A mayor número, mayor concentración del nuevo material. Además se realizaron "n" mediciones a diferentes temperaturas de conductividad para cada muestra. Los resultados fueron expresados en una matriz M de nxn. Se observa que un mismo material cuanto mayor temperatura tiene mayor conductividad. Además para cada cuanto mayor concentración a la misma temperatura, también mayor conductividad.

En conclusión podemos, al analizar la matriz, ver dos progresiones. Cada fila tiene números ordenados de forma creciente. Es decir para la fila i, M[i,j]<M[i,k] si k>j. Cada columna tiene números ordenados de forma creciente. Es decir para la columna j, M[i,j]<M[k,j] si k>i.

Dada la matriz M, los experimentadores quieren encontrar en qué posición se encuentra un determinado número.

Se pide:

1. Proponga una estrategia por fuerza bruta para resolver el problema. ¿Cuál es su complejidad?
2. Proponga una solución superadora utilizando división y conquista. Brinde pseudocódigo y estructuras de datos a utilizar. Intente que la complejidad sea la menor posible
3. Presente relación de recurrencia de su solución. Realizar el análisis de complejidad temporal.
4. En un nuevo experimento se preparan otras "n" muestras, pero en este caso se realizan "m" mediciones para cada una. ¿Cómo Impacta en su propuesta algorítmica y su análisis?
5. Dar un ejemplo completo del funcionamiento de su solución

1.1. Estrategia por fuerza bruta

Idea: recorrer la matriz elemento a elemento hasta encontrar el elemento buscado.

Pseudo código:

```
Sea M la matriz dada
Sea A el numero que se quiere encontrar
Para fila i = 0 de la matriz M:
    Para columna j = 0 de la matriz M:
        si A = M[i,j]
            retornar coordenadas [i,j] y salir
        incrementar j
    Fin Para
    incrementar i
Fin Para
```

Tal como se indica en el pseudo código, por fuerza bruta se recorre cada posición de la matriz por filas y luego por columna (también puede hacerse por columnas y luego por filas) y se compara cada elemento

$M[i, j]$ con el numero buscado. Si hay coincidencia, se retornan las coordenadas de dicha posición y el proceso se interrumpe.

En este caso, la complejidad temporal será $O(n^2)$ ya que en el peor caso se recorre toda la matriz M de nxn.

Por otro lado, se observa que si el valor buscado A se encuentra en las posiciones mas tempranas, la complejidad temporal es incluso menor que la que se propone por división y conquista (siendo el mejor caso aquel en el que $M[1, 1] = A$).

1.2. Estrategia por división y conquista

```
Sea M la matriz de m mediciones y n muestras
Sea, ademas, este caso uno particular en el que m = n
Sea A el numero que se quiere encontrar
encontrar_numero(M, A):
    Sea X el numero que se encuentra en la celda [i,j] de la matriz M,
    con i=n/2 y j=n/2 (Si i y/o j no son enteros se redondea hacia arriba)
    Si X > A:
        M1 = M - ([1,n]x[j,n])
        M2 = M - (M1 + [i,n]x[j,n])
        encontrar_numero(M1,A)
        encontrar_numero(M2,A)
    Si X < A:
        M1 = M - ([1,n]x[1,j])
        M2 = M - (M1 + [1,i]x[1,j])
        encontrar_numero(M1,A)
        encontrar_numero(M2,A)
    Sino:
        Devolver [i,j]
```

Observación sobre las coordenadas de la celda que aloja la solución: El pseudo código presentado indica que, hallada la solución, se devuelven las coordenadas de dicha celda. Para que efectivamente se devuelvan las coordenadas originales de la celda (i.e. las coordenadas de dicha celda en la matriz M) y no un par de coordenadas subjetivas a la submatriz que contiene al valor, se propone la siguiente solución: por un lado, se utiliza una estructura de nxn para almacenar los datos de las mediciones. Por otro lado, a medida que se van construyendo las submatrices $M1$ y $M2$, se toman las longitudes de $M1$ y $M2$ para calcular el posicionamiento de la celda *media* pero cada celda de $M1$ y $M2$ guarda una referencia a la matriz original, de modo tal que si la celda *media* aloja el valor buscado, la referencia a dicha celda devuelve las coordenadas originales de la misma, esto es, las coordenadas de dicha celda en la matriz M.

Estructuras de datos Se requieren estructuras matriciales que resultan ser submatrices del nivel anterior. Además, para cada nivel siguiente, se duplica la cantidad de matrices con respecto al nivel

anterior (y se reducen en tamaño). De modo que la cantidad de matrices que se necesitan es 2^k matrices, donde k es la cantidad de niveles del árbol. Cada nuevo nivel genera nuevas estructuras matriciales cuyo coste escala a razón de una suma geométrica.

1.3. Relación de recurrencia y complejidad temporal

La resolución del problema crece en forma de árbol, ya que la estrategia consiste en dividir la matriz original en dos matrices más acotadas en cada recursión. Dado que se descarta un trozo de matriz a partir del valor medio, cada recursión genera dos subproblemas con - a lo sumo - la mitad de los elementos. Además, dentro de cada recursión se calcula únicamente el valor medio de la matriz y se efectúan dos comparaciones de menor o igual. Por lo tanto, se puede expresar la relación de recurrencia del siguiente modo:

$$T(n) = 2T(\frac{n}{2}) + O(1) \tag{1}$$

$$T(0) = 1 \tag{2}$$

Para obtener la complejidad temporal de esta relación de recurrencia se utiliza el teorema maestro. Se identifican los valores:

$a = 2$

$b = 2$

$f(n) = O(1)$

Caso 2: $f(n) = O(n^{log_2 2}) = O(n) \rightarrow$ no cumple

Caso 1: $f(n) = O(n^{1-\epsilon})$ con $\epsilon > 0$ Si $\epsilon = 1$ entonces $f(n) = O(1)$. Por lo tanto, $T(n) = \Theta(n)$.

1.4. Experimento con "n"muestras y "m"mediciones

La representación para n muestras y m mediciones es una matriz no cuadrada de n columnas y m filas:

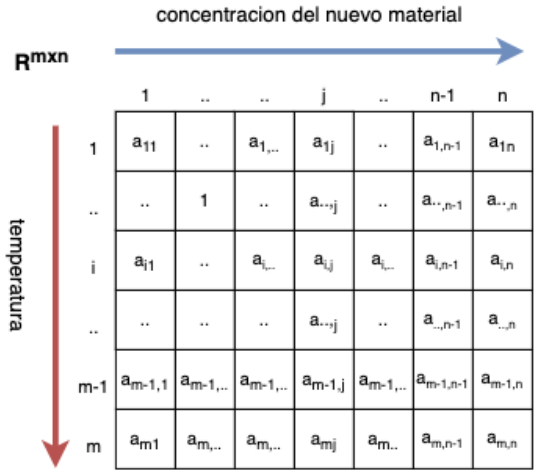

Figura 1: Matriz no cuadrada de m y n columnas

Si se aplica el algoritmo propuesto, se marca la celda $[i, j]$ que resulta de dividir la cantidad de columnas y filas entre 2. Ahora bien, como la matriz no es cuadrada, no existe una celda *central* $[i, j]$. No obstante, el algoritmo comprende este escenario y además lo resuelve redondeando hacia arriba siempre que i y j lo requieran. En este caso, el resultado es el siguiente:

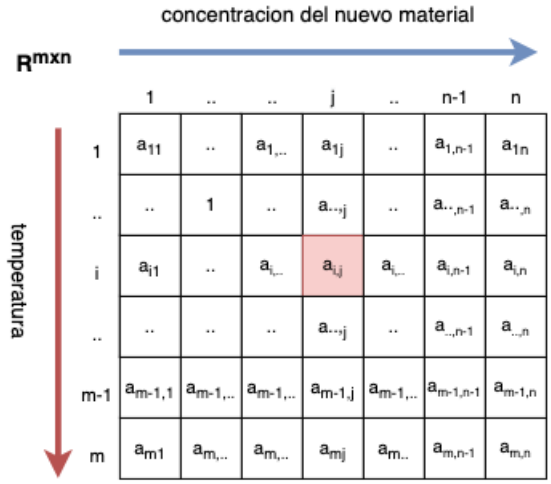

Figura 2: Celda $[i, j]$ de la matriz no cuadrada

Esta matriz además cumple con que el ordenamiento de los valores que almacena cada celda son crecientes hacia la derecha y hacia abajo. Por lo tanto, si el valor buscando es mayor al que guarda la celda $[i, j]$ se seleccionan las submatrices $M1$ y $M2$ tal como indica el pseudo código. Si es menor, la selección de $M1$ y $M2$ será distinta. Estos dos casos están contemplados por la solución propuesta y no presentan escenarios no cubiertos.

Por lo tanto, el caso particular en que la matriz es no cuadrada no impacta en la propuesta algorítmica ni en su costo.

1.5. Ejemplo del funcionamiento

A fin de mostrar el funcionamiento del algoritmo propuesto, se toma la siguiente matriz:

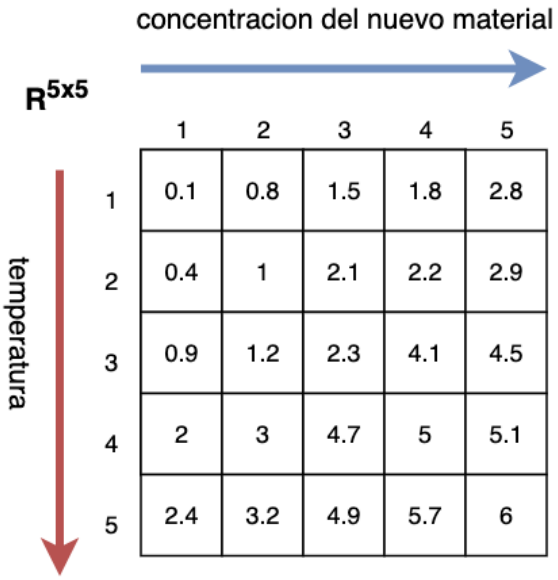

Figura 3: Matriz original M de tamaño $5x5$

El valor a buscar es $A = 2$. Tal como se puede ver, los valores están ordenados progresivamente tanto por filas como por columnas. Esto es fundamental para el correcto funcionamiento del algoritmo de división y conquista propuesto. En el caso particular de esta matriz, no hay valores repetidos. Esto no reviste mayor complejidad porque en caso de darse ese escenario y, además, de que el valor A buscado corresponda a alguno de esos valores repetidos, se devuelve el primer elemento encontrado.

Volviendo al algoritmo, se calcula i y j. En este caso:

$$i = \frac{5}{2} = 2{,}5 \rightarrow 3 \tag{3}$$

y

$$j = \frac{5}{2} = 2{,}5 \rightarrow 3 \tag{4}$$

por lo que $[i][j] = [3][3]$:

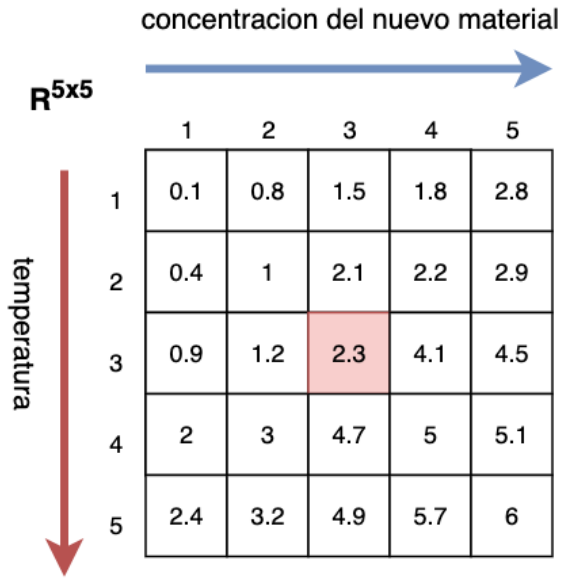

Figura 4: El elemento central [3][3] con valor 2,3

Como el valor que aloja la celda $[3,3]$ no es 2 sino que es mayor, se desecha esta celda. Además, la porción de matriz que se sabe no puede alojar al valor buscado A (es decir, lo que esté a la derecha y por debajo de 2.3) también se descarta y se obtienen las matrices $M1$ y $M2$ que son candidatas a contener a A. En el caso de $M1$:

$$M1 = [1,5]x[1,5] - [1,5]x[3,5] = [1,5]x[1,2] \tag{5}$$

En el caso de $M2$:

$$M2 = [1,5]x[1,5] - ([1,5]x[1,2] + [3,5]x[3,5]) = [1,2]x[3,5] \tag{6}$$

Las operaciones anteriores quedan reflejadas en la imagen debajo:

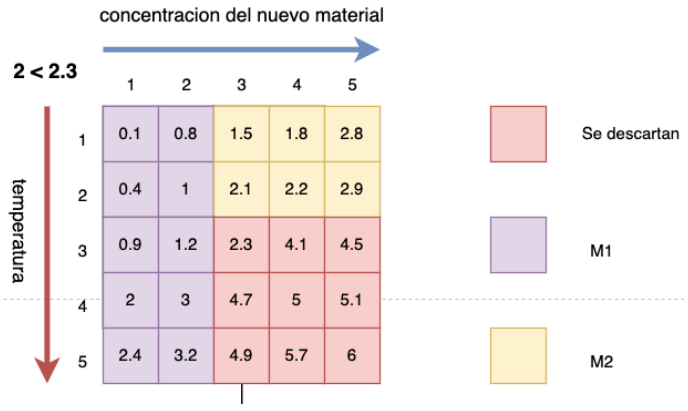

Figura 5: Recorte de la matriz original en dos sub matrices

Tal como muestra la imagen superior, solo sobreviven dos matrices $M1$ y $M2$ a las cuales se les aplica la misma lógica:

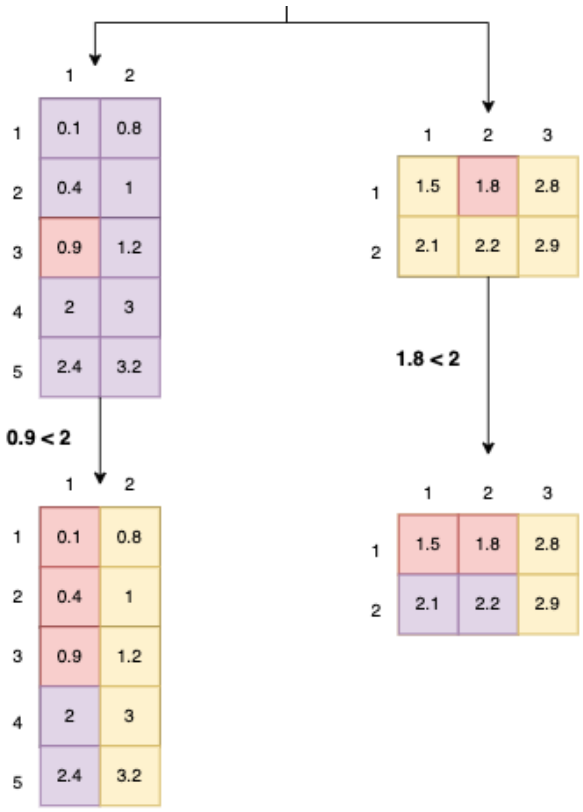

Figura 6: Aplicación recurrente de la lógica de la solución

En la rama **izquierda**, $[i, j] = [3, 1]$ pues:

$$i = \frac{5}{2} = 2{,}5 \rightarrow 3 \tag{7}$$

y

$$j = \frac{2}{2} = 1 \tag{8}$$

Además el valor de la celda $[3, 1]$ es 0,9 que es menor que 2 (se descarta). También se obtiene $M1$ de la rama izquierda (en color amarillo):

$$M1 = [1, 5]x[1, 2] - [1, 5]x[1, 1] = [1, 5]x[2, 2] \tag{9}$$

Y $M2$ de la rama izquierda (en color lila):

$$M2 = [1, 5]x[1, 2] - ([1, 5]x[2, 2] + [1, 3]x[1, 1]) = [4, 5]x[1, 1] \tag{10}$$

Por otro lado, en la rama **derecha**, $[i, j] = [1, 2]$ pues:

$$i = \frac{2}{2} = 1 \tag{11}$$

y

$$j = \frac{3}{2} = 1{,}5 \rightarrow 2 \tag{12}$$

Además el valor de la celda $[1, 2]$ es 1,8 que es menor que 2 (se descarta). También se obtiene $M1$ de la rama derecha (en color amarillo):

$$M1 = [1, 2]x[1, 3] - [1, 2]x[1, 2] = [1, 2]x[3, 3] \tag{13}$$

Y $M2$ de la rama derecha (en color lila):

$$M2 = [1,2]x[1,3] - ([1,2]x[3,3] + [1,1]x[1,2]) = [2,2]x[1,2] \tag{14}$$

Todavía no se ha encontrado la celda que guarda el valor 2 por lo que se vuelven a dividir cada matriz de cada rama.

Se aplica la misma lógica sobre cada matriz (ahora son 4 matrices en total). El valor buscado se encuentra en la matriz mas a la izquierda.

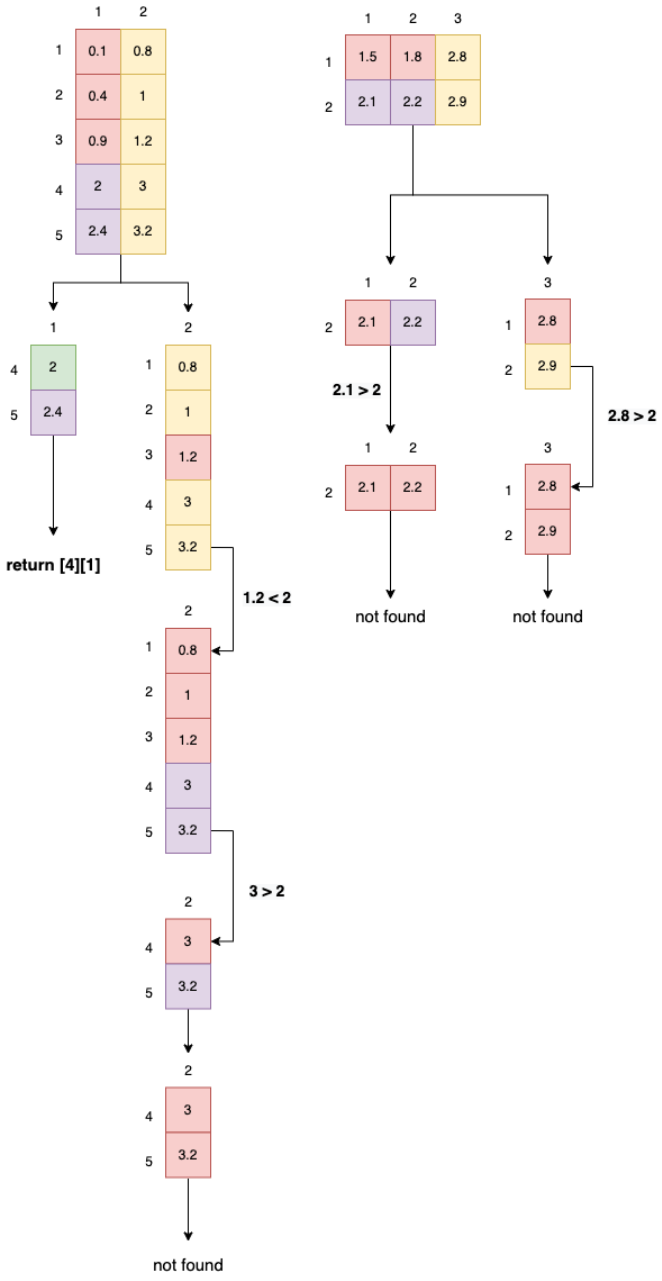

Figura 7: Hallazgo del valor A buscado

Para el primer caso, se busca la celda $[i, j]$ que resulta ser $[4, 1]$ pues:

$$i = \frac{2}{2} = 1 \rightarrow \textit{corresponde a la celda 4 de } M \tag{15}$$

y

$$j = \frac{1}{2} = 0{,}5 \rightarrow 1 \tag{16}$$

En este caso, se logró hallar A en la matriz M, el cual esta alojado en las posiciones [4][1] de la misma. Estas coordenadas se retornan. Por el contrario, si este valor no se encuentra, no se retornan coordenadas y el algoritmo finaliza satisfactoriamente.

Podemos ver que en las otras matrices se continúan con los cálculos. En la segunda rama, tenemos $[i, j] = [3][1]$ pues:

$$i = \frac{5}{2} = 2{,}5 \rightarrow 3 \tag{17}$$

y

$$j = \frac{2}{2} = 1 \tag{18}$$

Además el valor de la celda $[3, 1]$ es 1,2 que es menor que 2 (se descarta). Por lo que se descartan los valores que se encuentran por encima de 1,2 y si calculamos $M1$ obtenemos:

$$M1 = [1, 5]x[1, 1] - [1, 5]x[1, 1] = \emptyset \tag{19}$$

Y $M2$ queda (en color amarillo):

$$M2 = [1, 5]x[1, 1] - (\emptyset + [1, 3]x[1, 1]) = [4, 5]x[1, 1] \tag{20}$$

Si continuamos los cálculos en $M2$, tenemos $[i, j] = [1][1]$ pues:

$$i = \frac{2}{2} = 1 \tag{21}$$

y

$$j = \frac{1}{2} = 0{,}5 \rightarrow 1 \tag{22}$$

El valor de la celda $[1, 1]$ es 3 que es mayor que 2 (se descarta). Además se descartan los valores que se encuentran por debajo de 3, descartando así los valores restantes. Por lo que esta rama no devuelve un resultado.

En la tercera rama tenemos, $[i, j] = [1][1]$ pues:

$$i = \frac{1}{2} = 0{,}5 \rightarrow 1 \tag{23}$$

y

$$j = \frac{2}{2} = 1 \tag{24}$$

Además el valor de la celda $[1, 1]$ es 2,1 que es mayor que 2 (se descarta). Por lo que se descartan los valores que se encuentran a la derecha de 2,1(ya que van a ser mas grandes), descartando así los valores restantes. Por lo que esta rama no devuelve un resultado.

En la rama mas a la derecha tenemos, $[i, j] = [1][1]$ pues:

$$i = \frac{2}{2} = 1 \tag{25}$$

y

$$j = \frac{1}{2} = 0{,}5 \rightarrow 1 \tag{26}$$

Además el valor de la celda $[1, 1]$ es 2,8 que es mayor que 2 (se descarta). Por lo que se descartan los valores que se encuentran por debajo de 2,8 (ya que van a ser mas grandes), descartando así los valores restantes. Por lo que esta rama tampoco devuelve un resultado.

2. El almacén

Se debe organizar una bodega en un almacén. Tenemos "n" cajas. Cada caja comparte la misma profundidad pero varían en su altura y largo. El orden de las cajas no se puede modificar dado que están clasificadas mediante un código contiguo. Las cajas se ponen en repisas que comparten entre ellas la longitud L y cuya altura es regulable. El objetivo perseguido por el encargado de determinar en qué forma organizar las cajas con el objetivo de minimizar la suma de las alturas de las repisas utilizadas.

Se pide:

1. Mostrar por qué una propuesta greedy que agregue tantas cajas como sea posible por repisa no es óptima.
2. Proponer una solución al problema que utiliza programación dinámica. Incluya relación de recurrencia, pseudocódigo, estructuras de datos utilizadas y explicación en prosa.
3. Analice la complejidad temporal y espacial de su propuesta.
4. Programe la solución
5. Determine si su programa tiene la misma complejidad que su propuesta teórica.

2.1. Propuesta Greedy no óptima

Estamos ante un caso dónde necesitamos llevar cuenta de dos dimensiones: el largo de las cajas y el alto de las mismas. En lo referido al largo de cada caja, este aspecto actúa como limitante al momento de guardar nuevas cajas en un estante ya que la sumatoria de los largos l_i de cada caja no puede sobrepasar el largo total (L) de los estantes, es decir:

$$\sum l_i \leq L \tag{27}$$

La segunda dimensión del problema es la altura de las cajas, que será la que debemos optimizar para que la altura total de todos los estantes sea mínima, es decir:

$$min(\sum h_j) \tag{28}$$

donde h_j es la altura de cada estante.

Si pensamos el problema intuitivamente sin diseñar un algoritmo, naturalmente vamos a querer ordenar las cajas de mayor altura todos juntas y las de menor por otro lado, ya que la altura del estante únicamente la rige la caja más alta. No obstante, hay ciertos criterios que cumplir que hacen que esta opción no sea viable. Por tal motivo, un primer *approach* podría ser atacar el problema con una estrategia *greedy*.

La propuesta *greedy*, por su naturaleza, buscará guardar las cajas una a una a medida que llegan. Pero, ¿qué pasa cuando el estante actual aun tiene una pequeña fracción libre de longitud, no suficiente para que entre una caja y llega una nueva caja? En este caso, *greedy* fraccionará la caja, ubicará una porción en el estante actual (hasta cubrir la longitud L del estante) y el otro trozo será ubicado en un nuevo estante). Este enfoque esta alineado con la propuesta *greedy* porque optimizar el uso de la longitud de cada estante esta directamente vinculada con la cantidad de estantes que se utilizarán y esto con las

alturas acumuladas. Sin embargo, este problema especifico no admite la fragmentación de los objetos (cajas), por lo que el algoritmo *greedy* no puede plantear una solución óptima.

Si pudiésemos ignorar esta característica del fraccionamiento que realiza *greedy* y se supone que no hay fraccionamiento, se puede mostrar que aun con esta salvedad, *greedy* no es óptimo. Se describe el siguiente escenario: se acomodan las cajas según el orden en el que vienen comenzando por el primer estante de longitud L (el segundo estante es el que marca la altura del estante inferior). Entonces, de forma codiciosa se propone ajustar la altura del segundo estante solo cuando es necesario (i.e. cuando no hay mas lugar en el estante actual).

Supongamos que se recibe una caja de altura $10cm$. En ese caso, se sube el estante 2 en $10cm$ y se acomoda la caja que acaba de ingresar en el primer estante . Llega la segunda caja, también de $10cm$. No hay que ajustar la altura del estante 2. Se recibe la tercera caja de $5cm$ de alto, la cual entra sobradamente en $10cm$. No se baja el estante 2 porque bajarlo hace que las 2 primeras dos cajas no entren. Llega una cuarta caja, de $20cm$. Como la propuesta *greedy* busca el menor costo en cada iteración, la opción sera dejar la caja en el primer estante ajustando la altura del segundo. Notar que este costo es menor que subir la cuarta caja al siguiente estante porque en este caso la altura acumulada resultante es de $30cm$, contra los $20cm$ de costo que tiene dejar la cuarta caja en el estante 1 ajustando el estante 2. El proceso se repite hasta que se haya cubierto la longitud total L del primer estante. Lo mismo ocurre para los demás estantes hasta acomodar todas las cajas.

Ahora bien, por que no es óptima esta propuesta? Visualicemos el siguiente caso: llegan en total 10 cajas, y como el ancho de las cajas es igual independientemente de su altura, decimos que en cada estante entran como máximo 4 cajas.

Ahora supongamos que las primeras 3 cajas tienen altura $10cm$, las siguientes 4 cajas tienen altura $20cm$ y las siguientes 2 tienen altura $5cm$. Como acomoda el algoritmo *greedy*?

Las primeras 4 cajas van en el estante 1, las siguientes 4 en estante 2 y las 2 restantes en el 3ro, tal como se ve en el siguiente diagrama:

```
|5cm|  |5cm|.                  -> 5cm total
|20cm| |20cm| |20cm| |5cm|    -> 20cm total
|10cm| |10cm| |10cm| |20cm|   -> 20cm total
                  ____________________

                      45cm
```

Sin embargo, es fácil ver que este ordenamiento no es óptimo pues una mejor solución es colocar las 3 primeras cajas en el primer estante, las siguientes 4 en el estante 2 y las ultimas 3 en el tercer estante, como se muestra a continuación:

```
|5cm|  |5cm|  |5cm|              -> 5cm total
|20cm| |20cm| |20cm| |20cm|     -> 20cm total
|10cm| |10cm| |10cm|            -> 10cm total
                  ____________________

                      35cm
```

De modo que, aun obviando la fragmentación de la unidades, *greedy* no siempre arroja una solución óptima.

2.2. Solución teórica con Programación Dinámica

Para resolver el problema a través de Programación dinámica volvemos a la hipótesis del punto anterior. Para una mayor comprensión mostraremos los siguientes diagramas, no es lo mismo disponer las cajas como en la figura 6 que en la 7, dónde claramente h <h':

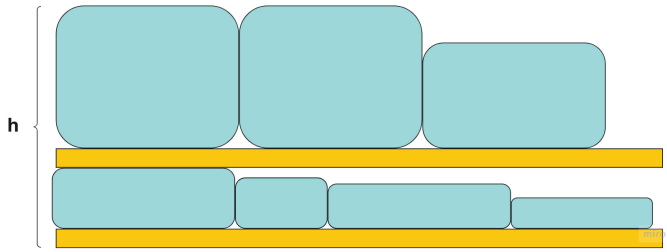

Figura 8: Disposición de cajas de mayor altura juntas

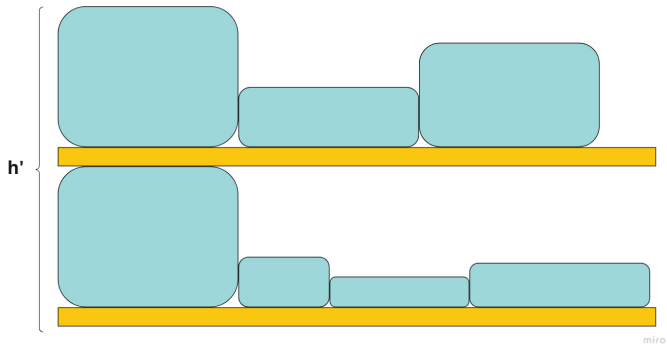

Figura 9: Disposición de cajas de forma randomizada

Podemos pensar a la altura como la ganancia que estamos tratando de maximizar pero con una limitante que es la longitud total de cada estante. Con "maximizar"la altura nos estamos refiriendo realmente buscar en cada pasada la mayor cantidad de cajas dónde la suma de sus altura sea la mayor, en el ejemplo anterior:

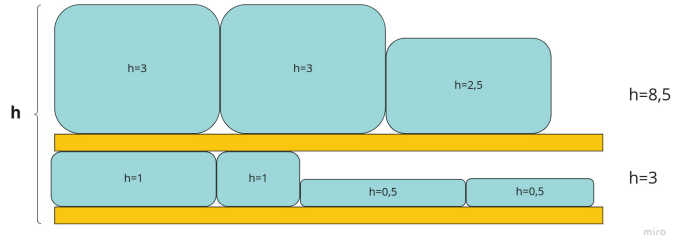

Figura 10: Disposición de cajas de mayor altura juntas

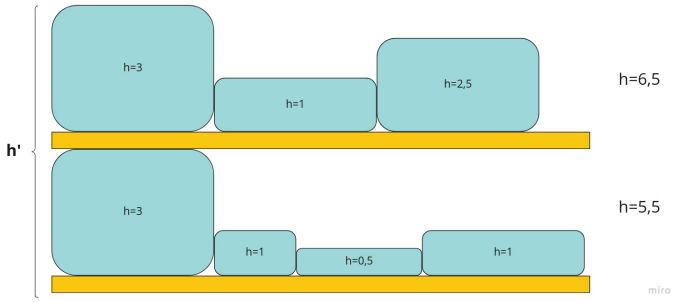

Figura 11: Disposición de cajas de forma randomizada

Aclaramos en cada pasada porque una vez que elijamos un subconjunto de cajas para el primer estante, ya vamos a haber analizado los restantes del subconjunto con lo cual sabremos que es la mejor opción para cada caso.

La idea del algoritmo es pensarlo como una matriz de n x m donde n es la cantidad de cajas y cada fila esta identificada por una caja; y m es el largo total (que es nuestro limitante). Cada columna corresponderá a la cantidad de largo que nos queda por llenar, se verá algo así:

Quedan x metros de largo dónde x en [0;m]

	0	1	2	3	4
-	0	0	0	0	0
A	0	g(A,1)	g(A,2)	g(A,3)	g(A,4)
B	0	g(B,1)	g(B,2)	g(B,3)	g(B,4)
C	0	g(C,1)	g(C,2)	g(C,3)	g(C,4)
D	0	g(D,1)	g(D,2)	g(D,3)	g(D,4)

Figura 12: Matriz de ganancia

En este caso m es igual al largo de nuestros estantes, en cada posición de la matriz evaluaremos si la caja que estamos evaluando nos dará la mayor altura. Y cada posición guardará la ganancia temporal de incluir o no esa caja faltando x metros por llenar. g(i,j) se lee: Ganancia máxima de evaluar la caja i faltando j unidades de longitud del estante.

Tendremos 4 casos para elegir o no una caja:

- Si no queda más estante, (columna 0) quiere decir que no podremos poner ninguna caja y la ganancia en altura será cero.
- Si no tenemos ninguna caja para ubicar (fila 0), no importa cuánto estante nos quede, no podremos poner ninguna caja y la ganancia en altura será cero.
- Si la caja no entra en el largo remanente, la ganancia será la anterior, es decir la ganancia máxima hasta el momento.

- Si la caja entra en el largo remanente, tendré dos escenarios: Si la ganancia actual (incluyendo la caja) es mayor a la histórica, entonces la sumo. Y sumo esa caja a los seleccionados. Si no lo es, entonces no la sumo, me quedo con la histórica y no sumo esa caja a los seleccionados.

2.3. Relación de recurrencia

Llamemos ALT a la matriz de alturas (ganancia) que debemos maximizar. i representará la i-esima caja con su largo li y su altura ai. j representará el largo remanente y j <= L.

ALT(i,0)=0
ALT(0,j)=0
ALT(i,j<li)=ALT(i 1)(L)
ALT(i,j)=MAX(ALT(i 1)(L li)+ai; ALT(i 1)(L))

En la última fila se evaluará la última caja, con lo cual en la posición (n,m) de la matriz obtendremos siempre la mayor ganancia posible para ese estante.

Esta explicación corresponde claramente a la maximización de altura de un sólo estante, lo que nosotros queremos es disponer las cajas en tantos estantes necesitemos. Es por eso que repetiremos este algoritmo hasta que no hayan más cajas. Esto se puede hacer fácilmente con un loop que evalúa si existen más cajas para posicionar en los estantes, ya que en cada iteración eliminaremos las cajas seleccionadas. Como en cada iteración elegimos las de mayor altura juntas, van a ir quedando las más bajas al final, esto se asemeja mucho a lo que en un principio habíamos pensado.

En la última fila se evaluará la última caja, con lo cual en la posición (n,m) de la matriz obtendremos siempre la mayor ganancia posible para ese estante.

Esta explicación corresponde claramente a la maximización de altura de un sólo estante, lo que nosotros queremos es disponer las cajas en tantos estantes necesitemos. Es por eso que repetiremos este algoritmo hasta que no hayan más cajas. Esto se puede hacer fácilmente con un loop que evalúa si existen más cajas para posicionar en los estantes, ya que en cada iteración eliminaremos las cajas seleccionadas. Como en cada iteración elegimos las de mayor altura juntas, van a ir quedando las más bajas al final, esto se asemeja mucho a lo que en un principio habíamos pensado.

Pseudo código

```
Sea C vector de n cajas con un codigo, un largo y una altura
Sea S_TOTAL la lista de listas de seleccionados, cada posicion corresponde
a un estante
Mientras que sigan habiendo cajas para posicionar
    Sea ALT la matriz de alturas acumuladas de (n+1)x(L+1)
    Sea S la matriz de seleccionados de un solo estante de (n+1)x(L+1)
    que guardara para cada posicion las cajas involucradas

    Desde i: 0 hasta n:
        ALT[i][0] = 0
```

```
Desde j: 0 hasta L:
    ALT[0][j] = 0

Desde i: 1 hasta n:
    Desde j: 1 hasta L:
        Si Ci.largo <= j:
            altura_temporal = Ci.altura + ALT[i-1][j - Ci.largo]
        Si no:
        // Ponemos - INFINITO para que no ingrese a la siguiente condicion
            altura_temporal = -INFINITO

        altura_historica = ALT[i-1][j]

        Si altura_temporal > altura_historica:
            ALT[i][j] = altura_temporal
            S[i][j] =  Ci.codigo + S[i-1][j - Ci.largo]
        Si no:
            ALT[i][j] = altura_historica
            S[i][j] =  S[i-1][j]

S_TOTAL.agregar(S[i][j])
C.eliminar(S[i][j])

Retornar S_TOTAL
```

2.4. Complejidad de solución teórica

2.4.1. Complejidad Temporal

While loop O(n) → En el peor de los casos donde cada caja tenga de largo L

Loop de caso base 1 O(n)

Loop de caso base 2 O(n)

Loop recorriendo 1 a n O(n)

Loop recorriendo de 1 a j O(L)

Condiciones y asignaciones a variables O(1)

Agregar al final de S TOTAL O(1)

Eliminar de C los seleccionados O(n)

```
= [While loop] *
    ([Loop de caso base 1] +
    [Loop de caso base 2] +
    [Loop recorriendo de 1 a n]
        *([Loop recorriendo de a j]+[Condiciones y asignaciones a variables]) +
    [Agregar al final de S_TOTAL] +
    [Eliminar de C los seleccionados])
    = O(n) * (O(n) + O(n) + O(n) * (O(L) + O(1)) + O(1) + O(n))
    = O(n) * (O(n) + O(n) + O(nL) + O(1) + O(n))
    = O(n) * (O(n) + O(nL))
    = O(n) * O(nL)
    = O(((n^ 2)L)) -> Complejidad temporal total
```

2.4.2. Complejidad Espacial

En el pseudocódigo se utilizan 4 estructuras de datos: un vector de cajas de tamaño n, su complejidad entonces será de O(n); un vector de seleccionados totales que siempre va a tener tamaño n porque los elementos no se repiten, entonces O(n); una matriz de tamaño (n+1) x (L+1), con un orden de O(nxL), y por último tendremos otra matriz de (n+1)*(L+1) que guardará en cada posición la cantidad de cajas involucradas en esa ganancia.

	0	1	2	3	4
-	0	0	0	0	0
A	0	1	1	1	1
B	0	2	2	2	2
C	0	3	3	3	3
D	0	4	4	4	4

Figura 13: Matriz de seleccionados

La figura 11 muestra a la matriz de seleccionados y en su interior un número representando la cantidad máxima de elementos en su interior. La cantidad de elementos totales será entonces la sumatoria de de todos esos números, en el caso de ejemplo tendremos $[1+2+3+4]*4=56$. Si queremos una expresión para todo n será algo como $[1+2+...+(n-1)+n]*L$ dónde el primer término del producto es la clásica

serie $\sum_{i=1}^{n} i$ y su resolución es $\frac{(n+1)^3-n^3-1}{6}$, que es de orden 3, entonces nuestra complejidad espacial será de $O(n^3 * L)$. Finalmente la complejidad espacial total de nuestro programa será de $O(n^3 * L)$.

2.5. Solución práctica con Programación Dinámica

La solución se realizó utilizando Python, el código junto con un archivo de ejemplo se encuentra en la carpeta almacen dentro de la carpeta del trabajo práctico. El programa se ejecuta de la siguiente manera:

```
python3 ./almacen.py file_path largo_repisas num_cajas
```

2.6. Complejidad de solución práctica

2.6.1. Complejidad Temporal

Comenzaremos mostrando las dos funciones que contienen la lógica del algoritmo, e iremos parte por parte calculando su complejidad y justificando junto con la documentación de Python.

```
72
73  def programacion_dinamica(cajas, alturas, seleccionados, L, n):
74
75      max_altura_estante = 0
76
77      for i in range(0,n):
78          for j in range(1,L+1):
79              altura_temporal = MENOS_INFINITO
80
81              if cajas[i].largo <= j:
82                  altura_temporal = cajas[i].altura + alturas[i][j-cajas[i].largo]
83
84              altura_historica = alturas[i][j]
85
86              if altura_temporal > altura_historica:
87                  alturas[i+1][j] = altura_temporal
88                  seleccionados[i+1][j] = seleccionados[i][j-cajas[i].largo].copy()
89                  seleccionados[i+1][j].append(cajas[i])
90
91                  if max_altura_estante < cajas[i].altura:
92                      max_altura_estante = cajas[i].altura
93              else:
94                  alturas[i+1][j] = altura_historica
95                  seleccionados[i+1][j] = seleccionados[i][j].copy()
96
97      return seleccionados[n][L], max_altura_estante
```

Figura 14: Función programacion dinamica

Esta función hace la lógica propiamente dicha de la programación dinámica. Tenemos dos loops que su complejidad combinada va a ser de O(nxL), igual a como se planteó en el pseudocódigo. Luego se realizan muchas asignaciones a variables y a posiciones específicas dentro de matrices pero su complejidad sigue siendo O(1). Lo que cambia del pseudocódigo es el uso de dos métodos: copy() y append(). Según la documentación de Python:

copy(): O(x) en el caso de una lista siendo x la cantidad de elementos de una lista. Como seleccionados es una matriz que guarda listas en su interior, se supone esa complejidad en el peor de los casos es O(n) ya que podemos llegar a copiar los n elementos si todos entran en un estante.

append(): O(1), ya que se agrega un elemento al final de la lista.

La complejidad total para esta función será entonces de $O((n^2) * L)$.

```
101 def disponer_cajas(cajas, L, n):
102
103     num_estantes = 1
104     altura_acumulada = 0
105
106     while n > 0:
107         seleccionados = crear_matriz_seleccionados(L, n)
108         alturas = crear_matriz_ganancia(L, n)
109
110         seleccionados_final, max_altura_estante = programacion_dinamica(cajas, alturas, seleccionados, L, n)
111
112         print("En el estante "+str(num_estantes)+": Se dispondrán las cajas", end=" ")
113
114         for caja_seleccionada in seleccionados_final:
115             print(caja_seleccionada.cod,end=" ")
116             cajas.remove(caja_seleccionada)
117
118
119         print("con una altura máxima de "+str(max_altura_estante)+"\n")
120
121         n = len(cajas)
122
123         num_estantes+=1
124         altura_acumulada+=max_altura_estante
125
126     print("La altura de todos los estantes será: "+str(altura_acumulada)+"\n")
```

Figura 15: Función disponer cajas

Esta función contiene la lógica general que llama a la función programacion dinamica para las cajas que van quedando dentro del vector de cajas.

La complejidad del while será la misma del pseudocódigo: O(n) en el peor de los casos. Dentro tiene dos funciones que inicializan ambas matrices (las dejamos fuera ya que no hacen a la lógica del funcionamiento). A esta complejidad se le multiplicará la calculada anteriormente de $O((n^2) * L)$ por la llamada a la función. También se realizan algunas asignaciones a variables que son de O(1). Luego, tendremos un for loop que iterará por los elementos de la lista devuelta por la función (seleccionados final), dentro de ese bloque se remueven las cajas ya seleccionadas mediante un remove(). Finalmente, utilizamos la función len() para recalcular n. Según la documentación de Python:

for loop de listas: O(x) siendo x la cantidad de elementos de la lista. En nuestro caso, será de O(n) porque en el peor de los casos tendremos todas las cajas seleccionadas en una iteración.

remove(): O(x) siendo x la cantidad de elementos de la lista. En nuestro caso será de O(n) porque en el peor de los casos deberá recorrer todo el vector de cajas (por ejemplo, en la primera iteración del while).

len(): O(1)

La complejidad total de nuestro programa será entonces de $O((n^3 L)$. Hay un aumento en la complejidad del algoritmo por los métodos específicos usados y que no contemplamos que se iban a usar, como el copy() (para no crear referencias entre las listas) y el for loop para remover a las cajas seleccionadas.

Referencia a la documentación: https://wiki.python.org/moin/TimeComplexity

2.6.2. Complejidad Espacial

Se utilizan 3 estructuras de datos: una lista de cajas que contendrá en sus inicios las n cajas, una matriz de (n+1)x(L+1) que contendrá la acumulación de las alturas por cada estante y una matriz de seleccionados también de (n+1)x(L+1) pero con la complejidad que dijimos anteriormente ya que es una matriz que contiene varios elementos por cada posición y su complejidad es de $O(n^3 * L)$. Cambia la cantidad de estructuras a utilizar pero la complejidad no , hay una complejidad espacial total de $O(n^3 * L)$ (no utilizamos S TOTAL porque imprimimos los valores directamente de la matriz de seleccionados).

3. Un poco de teoría

1. Hasta el momento hemos visto 3 formas distintas de resolver problemas. Greedy, división y conquista y programación dinámica.

   a) Describa brevemente en qué consiste cada una de ellas

   b) ¿Qué propiedades requiere el problema para poder ser resueltos por ellos?

2. Considere un problema teórico y suponga que se puede resolver por cualquiera de las tres metodologías. Determine qué cuestiones tendría en cuenta para seleccionar una sobre las otras. ¿Siempre existirá una metodología mejor que las otras?

3.1. Breve descripción de cada metodología

Los algoritmos **Greedy** se utilizan para resolver problemas de optimización, intentan maximizar o minimizar alguna cantidad. La idea es hacer una división jerárquica en subproblemas tal que la resolución de cada subproblema habilite un nuevo subproblema para resolver. En cada subproblema se aplica un criterio de elección para la solución, basándose en la situación local.

La estrategia de **División y conquista** consiste en la división del problema en subproblemas de menor tamaño para facilitar la resolución del mismo. Los subproblemas se resuelven de forma independiente del resto de los subproblemas generados. Como el subproblema mantiene las mismas características y propiedades del problema original, este puede ser subdividido en nuevos subproblemas de menor tamaño. Esta subdivisión continua de manera recursiva hasta llegar a un caso base, el cual puede resolverse de forma trivial. De esta manera se van resolviendo los subproblemas, hasta tener todas las soluciones y combinarlas para obtener un solución global al problema original.

Al igual que Greedy, la metodología de **programación dinámica** resuelve problemas de optimización donde se quiere maximizar o minimizar un resultado. Una particularidad de programación dinámica es su propiedad de memorización, esto es que a medida que atraviesa los subproblemas y los va resolviendo, va almacenando los resultados obtenidos. Además, se dice que el problema tiene la propiedad de solapamiento de subproblemas si existe la posibilidad de aprovechar los resultados almacenados para resolver otros subproblemas.

3.2. Propiedades requeridas

En el caso de **greedy**, si bien puede crearse un algoritmo de este tipo para cualquier problema de optimización, la solución que se obtiene no será siempre óptima. Para garantizar su optimalidad, el problema debe cumplir ciertas propiedades:

- Subestructura óptima: la solución al problema contiene dentro de sí la solución óptima de sus subproblemas
- Elección greedy: en cada paso, se debe hacer una elección entre todos los posibles subproblemas.

En cuanto a **División y conquista**, no todo problema se puede resolver mediante esta metodología, al igual que Greedy debe cumplir con la propiedad de subestructura óptima. Es decir que el problema

debe poder separarse en subproblemas de manera recursiva y al encontrar la solución óptima de estos, encontrar la solución óptima global. La diferencia es que la división, por División y conquista se puede pensar en anchura, en cambio en Greedy es en profundidad.

En el caso de **programación dinámica**, al igual que las metodologías Greedy y División y conquista, tiene la característica de dividir el problema en diferentes subproblemas. Por lo tanto, el problema debe cumplir con la característica de subestructura óptima para poder ser resuelto en forma óptima.

3.3. Elección de metodología

Si tomamos un problema que puede resolverse por cualquiera de los tres algoritmos, en principio no se tendría una técnica superior a la otra, dado que dependerá del problema mismo y las prioridades que se prioricen a la hora de resolver. Podemos considerar que si queremos obtener una solución que implique una mejor complejidad temporal, nos vamos a inclinar por el algoritmo de Programación dinámica dado que esta tecnica guarda información que se obtiene en cada subproblema que se resuelve, y de ese modo podríamos lograr optimizar iteraciones en subproblemas de un grado mayor. Aunque también podría ocurrir que nos topemos con un problema que no tenga un considerable número de subproblemas que se repitan entre sí y esto implicaría que se van a generar múltiples resoluciones a almacenar y luego no serán reutilizadas. De modo que usaríamos un enorme espacio de más.
Por otro lado, el algoritmo de división y conquista si bien tiene similitud con la programación dinámica, cada subproblema no tiene relación alguna con otro subproblema, ya que son independientes, y la resolución de cada uno de estos subproblemas llevará más tiempo.
Por último, la opción greedy resulta un estrategia a utilizar que suele ser óptima en la mayoría de los casos ya que buscan justamente optimización. Además en esta técnica se resolverá siempre sobre un mismo arreglo de modo que conservará la complejidad.
Finalmente, no podemos concluir en cuál estrategia será mejor debido a que esa respuesta está directamente relacionada con los requisitos y prioridades que se especifiquen en un enunciado (o pedido) y a su vez condicionados por los recursos que dispongamos en cuanto a los ordenadores que se utilicen para aplicar una de las estrategías y llegar a una solución,

4. Segunda Entrega

4.1. Comentarios Parte 1

Correcta solución por fuerza bruta y su complejidad temporal. Falta mencionar la complejidad espacial.

Se plantea una buena solución por división y conquista.

El pseudocódigo es correcto y bastante claro. Es buena la aclaración de las coordenadas relativas vs absolutas. Como detalle, hubiera sido conveniente que esta lógica se incluya en el pseudocódigo.

Menciona correctamente las estructuras a utilizar aunque mencionan que en cada iteración se generan nuevas matrices. Por la explicación dada asumo que se están generando copias de las matrices. Esto impacta en la complejidad aunque no se tuvo en cuenta en la aplicación del teorema maestro. Como recomendación: trabajar con índices de referencias a la matriz original.

Dice que se generan 2^k matrices siendo k la cantidad de niveles del árbol. Entiendo a que apuntan, pero expliquen las cosas con parámetros de la instancia del problema. En este caso: n.

Hay un problema con la solución, que impacta en el análisis de complejidad. En cada iteración se crean 2 nuevas matrices, una de ellas claramente no cuadrada. Sin embargo, todo el análisis temporal se hace tomando el parametro n de una matriz cuadrada. Se dice: "Dado que se descarta un trozo de matriz a partir del valor medio, cada recursión genera dos subproblemas con - a lo sumo - la mitad de los elementos". ¿Puedo afirmar esto? Yo veo que en cada iteración se crea una matriz de n/2 x n/2 y otra de n x n/2. Como cota superior tendría matriz cuadrada de n. Como consejo, intentar crear 3 matrices cuadradas en vez de 2 no cuadradas en cada iteración. Revisar todo esto y por consiguiente el análisis de complejidad.

En la variación para matrices no cuadradas, no se menciona al nuevo parámetro de entrada m. Es cierto que su algoritmo funciona para matrices no cuadradas, ¿pero que complejidad tiene? Dice seguir siendo O(n), ¿no importa el tamaño de m?.

Excelente el ejemplo de ejecución, muy claro y fácil de seguir!

4.2. Correcciones Parte 1

4.2.1. Estrategia por fuerza bruta

En lo que respecta a la complejidad espacial, se utiliza una única estructura auxiliar mientras se recorre la matriz: una tupla de 2 elementos que guarda la posición $[i, j]$ de la celda actual de forma tal que si $M[i][j]$ aloja el valor A que se busca, entonces se devuelve dicha tupla. Si por otro lado, el valor no coincide, se sobreescribe el valor de la tupla con los nuevos índices $[i + 1][j + 1]$ y el proceso se repite. La complejidad espacial pues es $O(1)$.

4.2.2. Pseudocódigo de la solución

```
Sea M la matriz de m mediciones y n muestras
Sea, ademas, este caso uno particular en el que m = n
Sea A el numero que se quiere encontrar
```

```
fila_inicio = 1
fila_fin = m
col_inicio = 1
col_fin = n
resultado = encontrar_numero(M, A, fila_inicio, fila_fin, col_inicio, col_fin)

encontrar_numero(M, A, fila_inicio, fila_fin, col_inicio, col_fin):

    Si (fila_fin - fila_inicio = 1)
        Para i en (fila_inicio, fila_fin)
            Para j en (col_inicio, col_fin)
                Si (M[i][j] == A)
                  Devolver i, j

    Sino
        Sea X = M[i][j],
        con
            i=[(fila_fin - fila_inicio) / 2 + fila_inicio]
            j=[(col_fin - col_inicio)/2 + col_inicio]
        (Si i y/o j no son enteros se redondea hacia arriba)

        % MATRIZ 2
        fila_inicio_M2 = fila_inicio
        fila_fin_M2 = i
        Si [(fila_fin - fila_inicio + 1) y (col_fin - col_inicio + 1) son pares]:
            columna_inicio_M2 = j + 1
        Sino:
            columna_inicio_M2 = j
        columna_fin_M2 = col_fin
        encontrar_numero(M,A,fila_inicio_M2,fila_fin_M2,
        columna_inicio_M2,columna_fin_M2)

        % MATRIZ 3
        Si [(fila_fin - fila_inicio + 1) y (col_fin - col_inicio + 1) son pares]:
            fila_inicio_M3 = i + 1
        Sino:
            fila_inicio_M3 = i
        fila_fin_M3 = fila_fin
        columna_inicio_M3 = col_inicio
```

```
        columna_fin_M3 = j
        encontrar_numero(M,A,fila_inicio_M3,fila_fin_M3,
        columna_inicio_M3,columna_fin_M3)



        Si X > A:
            fila_inicio_M1 = fila_inicio
            fila_fin_M1 = i
            columna_inicio_M1 = col_inicio
            columna_fin_M1 = j
            encontrar_numero(M,A,fila_inicio_M1,fila_fin_M1,
            columna_inicio_M1,columna_fin_M1)

        Si X < A:
            Si [(fila_fin - fila_inicio + 1) y (col_fin - col_inicio + 1) son pares]:
                fila_inicio_M1 = i + 1
            Sino:
                fila_inicio_M1 = i
            fila_fin_M1 = fila_fin
            Si [(fila_fin - fila_inicio + 1) y (col_fin - col_inicio + 1) son pares]:
                columna_inicio_M1 = j + 1
            Sino:
                columna_inicio_M1 = j
            columna_fin_M1 = col_fin
            encontrar_numero(M,A,fila_inicio_M1,fila_fin_M1,
            columna_inicio_M1,columna_fin_M1)
        Sino:
            Devolver i, j
```

Con esta corrección al pseudo código, se distinguen 3 nuevas características:

- se evita la construcción de nuevas matrices en cada iteración que causa un empeoramiento en el valor de complejidad de la solución. Por el contrario, con el uso de una única matriz M no se escala en complejidad como si ocurre en la solución presentada en la primera entrega.
- El cambio sustancial de la nueva propuesta de división y conquista consiste, no solo en el uso de referencias para referirse a la matriz original, sino también en la forma en que se 'divide' la matriz M para su posterior procesamiento. En esta nueva versión del algoritmo, se 'divide' la matriz actual en 3 matrices mas pequeñas de igual tamaño en forma recursiva, llegando finalmente al caso base, una matriz de $2x2$ la cual se recorre buscando el valor A deseado.

4.2.3. Análisis paso a paso del nuevo algoritmo

En este apartado, se muestra el funcionamiento de la nueva versión del algoritmo sobre la matriz M que se empleó en la primera entrega.

Sea A el valor que se desea buscar en la matriz M, una matriz cuadrada de $5x5$, es decir con n impar, tal como se muestra en la figura debajo:

Figura 16: Matriz original M, donde el valor buscado A = 2

1. Se obtienen los valores de $fila\_inicio$, $fila\_fin$, col_inicio y col_fin de la matriz actual de la siguiente forma:

$$\begin{aligned} fila\_inicio &= 1 \\ fila\_fin &= 5 \\ col\_inicio &= 1 \\ col\_fin &= 5 \end{aligned} \tag{29}$$

Con estos valores se obtienen las coordenadas (i, j) que corresponden a la celda media de M. Para calcular i (la fila media) se toma la mitad de la diferencia entre la ultima y la primera fila y a este resultado se le aplica un ajuste de acuerdo a la primera fila, es decir:

$$i = \frac{fila\_fin - fila\_inicio}{2} + fila\_inicio \tag{30}$$

En este caso resulta:

$$i = \frac{5-1}{2} + 1 \rightarrow i = 3 \tag{31}$$

Análogamente, se obtiene j (la columna media):

$$j = \frac{col\_fin - col\_inicio}{2} + col\_inicio \tag{32}$$

En este caso resulta:

$$j = \frac{5-1}{2} + 1 \rightarrow j = 3 \tag{33}$$

Es decir que se obtiene $(i, j) = (3, 3)$. Con este par de coordenadas, se encuentra el valor medio X:

$$X = M[3][3] = 2{,}3 \tag{34}$$

Como el valor buscado $A = 2$ no coincide con $X = 2{,}3$, entonces se pasa a 'dividir' la matriz en 3 partes iguales.

2. Para 'dividir' la matriz M en 3 partes iguales, se construyen los valores de $fila\_inicio$, $fila\_fin$, col_inicio y col_fin para cada una de la submatrices de la siguiente manera:

   Para la submatriz $M1$:

$$\begin{aligned} fila\_inicio\_M1 &= fila\_inicio = 1 \\ fila\_fin\_M1 &= i = 3 \\ col\_inicio\_M1 &= columna\_inicio = 1 \\ col\_fin\_M1 &= j = 3 \end{aligned} \tag{35}$$

   Para la submatriz $M2$:

$$\begin{aligned} fila\_inicio\_M2 &= fila\_inicio = 1 \\ fila\_fin\_M2 &= i = 3 \\ col\_inicio\_M2 &= j = 3 \\ col\_fin\_M2 &= col\_fin = 5 \end{aligned} \tag{36}$$

   Para la submatriz $M3$:

$$\begin{aligned} fila\_inicio\_M3 &= i = 3 \\ fila\_fin\_M3 &= fila_fin = 5 \\ col\_inicio\_M3 &= columna\_inicio = 1 \\ col\_fin\_M3 &= j = 3 \end{aligned} \tag{37}$$

   Y, visualmente, $M1$, $M2$ y $M3$ son:

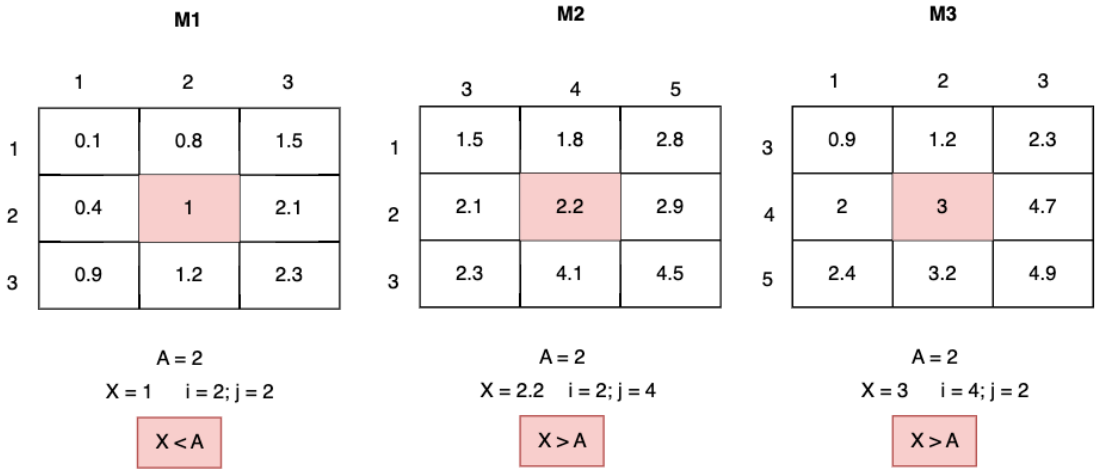

Figura 17: Matrices M1, M2 y M3 'subdivididas' a partir de la matriz original M

3. A continuación, se procede a tomar el valor medio de $M1$, tal como se hizo con M. En este caso, como $i = j = 2$, resulta que $X = M[2][2] = 1$, el cual no es el A buscado. Por lo tanto, se procede a subdividir $M1$ en 3 partes iguales pero mas pequeñas que $M1$.

4. Para construir las 3 submatrices a partir de $M1$ se sigue la misma logica que en el caso de M: se calculan los valores de $fila\_inicio$, $fila\_fin$, col_inicio y col_fin para cada una de la submatrices de $M1$ de la siguiente manera:

   Para la submatriz M1.1:

$$\begin{aligned} fila\_inicio\_M1{,}1 &= i\_M1 = 2 \\ fila\_fin\_M1{,}1 &= fila\_fin\_M1 = 3 \\ col\_inicio\_M1{,}1 &= j\_M1 = 2 \\ col\_fin\_M1{,}1 &= columna\_fin\_M1 = 3 \end{aligned} \tag{38}$$

   Para la submatriz M1.2:

$$\begin{aligned} fila\_inicio\_M1{,}2 &= fila\_inicio\_M1 = 1 \\ fila\_fin\_M1{,}2 &= i\_M1 = 2 \\ col\_inicio\_M1{,}2 &= j\_M1 = 2 \\ col\_fin\_M1{,}2 &= col\_fin\_M1 = 3 \end{aligned} \tag{39}$$

   Para la submatriz M1.3:

$$\begin{aligned} fila\_inicio\_M1{,}3 &= i\_M1 = 2 \\ fila\_fin\_M1{,}3 &= fila_fin\_M1 = 3 \\ col\_inicio\_M1{,}3 &= columna\_inicio\_M1 = 1 \\ col\_fin\_M1{,}3 &= j\_M1 = 2 \end{aligned} \tag{40}$$

Y, visualmente, M1.1, M1.2 y M1.3 son:

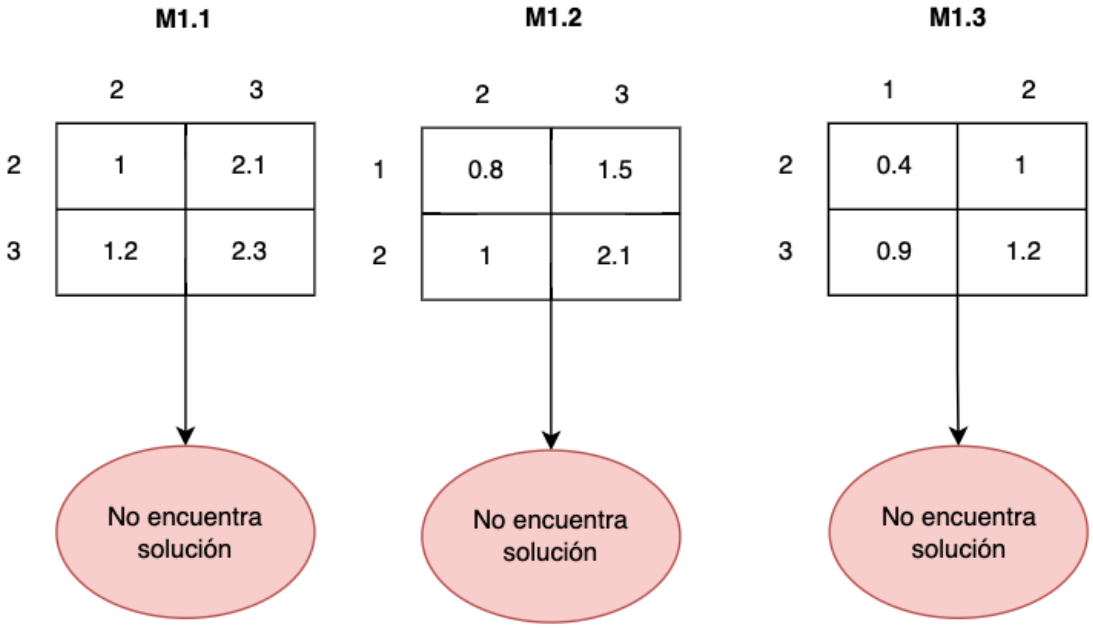

Figura 18: Matrices M1.1, M1.2 y M1.3 'subdivididas' a partir de la matriz M1

5. Dado que en la siguiente iteración se llega al caso base, esto es, una submatriz de $2x2$, entonces como indica el pseudocódigo, se procede a recorrer cada una de estas buscando el valor $A = 2$. Como en ningún caso se cumple la igualdad $X = A$ no se retorna ninguna solución (**null**).

6. Para el caso de la matriz $M2$ el procedimiento es análogo, y al no coincidir el valor medio de $M2$ con el valor A buscado, se aplica una 'división' de $M2$ y se obtienen las siguientes submatrices derivadas:

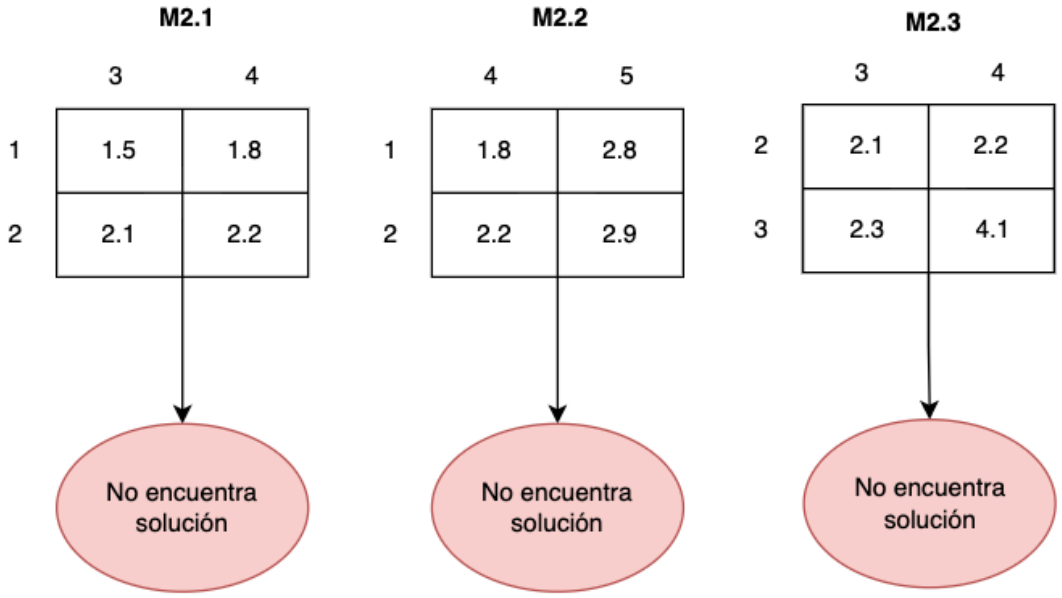

Figura 19: Matrices M2.1, M2.2 y M2.3 'subdivididas' a partir de la matriz M2

Tal como ocurrió con las submatrices derivadas de $M1$, tampoco se halla solución para las submatrices derivadas de $M2$.

7. Finalmente, se procede a aplicar el procesamiento sobre la submatriz $M3$. Al no coincidir el valor medio de $M3$ con el valor A buscado, se obtienen sus submatrices derivadas:

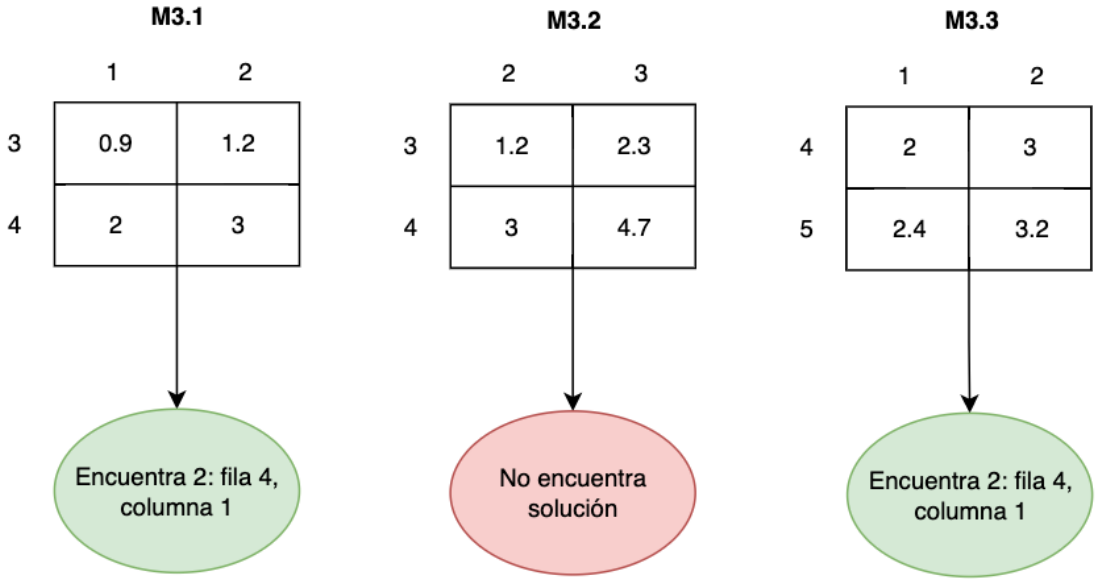

Figura 20: Matrices M3.1, M3.2 y M3.3 'subdivididas' a partir de la matriz M2. En este caso, se hallan 2 soluciones.

Como se llega al caso base, esto es, se llegan a matrices derivadas de $2x2$, se recorren las celdas en busqueda del valor $A = 2$. Tal como se ve en la figura, se encuentran 2 resultados y el algoritmo devuelve una de las soluciones.

Comentario sobre los tipos de matrices y su tratamiento: Tal como se detalla en el pseudo código, existen 2 tipos de matrices cuadradas que puede tomar el algoritmo: matrices cuadradas con $n = m$ par o impar. El caso mostrado paso a paso es el de $m = n$ impar. En el caso de $m = n$ par, el tratamiento de la matriz es similar salvo por la construcción de las matrices derivadas que se construyen a partir de la matriz actual. Fuera de ese criterio, no hay diferencia en como el algoritmo planteado resuelve el problema de hallar A.

4.2.4. Análisis de complejidad

Con la nueva propuesta algorítmica por división y conquista, se divide la matriz actual en 3 partes iguales. Asimismo, se procesan las 3 nuevas submatrices derivadas de la matriz actual de la misma manera. Por lo tanto, se tiene que:

$$\begin{aligned} b &= 3 \\ a &= 3 \\ f(n) &= O(1) \end{aligned} \tag{41}$$

Por lo tanto, la ecuación de recurrencia resulta ser:

$$T(n) = 3T(\frac{n}{3}) + O(1) \tag{42}$$

$$T(0) = 1 \tag{43}$$

Entonces, aplicando el teorema maestro:

Caso 2: $f(n) = O(n^{log_3 3}) = O(n) \rightarrow$ no cumple

Caso 1: $f(n) = O(n^{log_3 3 - \epsilon})$ con $\epsilon > 0$ Si $\epsilon = 1$ entonces $f(n) = O(1)$. Por lo tanto, $T(n) = \Theta(n)$.

4.2.5. Algoritmo para matrices no cuadradas y análisis de complejidad

> 'En la variación para matrices no cuadradas, no se menciona al nuevo parámetro de entrada m. Es cierto que su algoritmo funciona para matrices no cuadradas, ¿pero que complejidad tiene? Dice seguir siendo O(n), ¿no importa el tamaño de m?'

Se presenta una adaptación del algoritmo original con una variación para tratar a las matrices no cuadradas. La idea fundamental consiste en 'completar' la matriz no cuadrada de forma tal que se obtenga una matriz cuadrada que pueda procesar el algoritmo anterior, tal como se muestra a continuación:

```
Sea M la matriz de m mediciones y n muestras
Sea A el numero que se quiere encontrar
mayorValor = M[m][n]
Si m > n:
    Para i entre (1,m):
        Para j entre (n,m):
            Agregar M[i][j] = mayorValor
    fila_inicio = 1
    fila_fin = m
    col_inicio = 1
    col_fin = m
Sino:
    Para i entre (m,n):
        Para j entre (1,n):
            Agregar M[i][j] = mayorValor
    fila_inicio = 1
    fila_fin = n
    col_inicio = 1
    col_fin = n
Sea, ahora, el caso en el que m = n
resultado = encontrar_numero(M, A, fila_inicio, fila_fin, col_inicio, col_fin)

encontrar_numero(M, A, fila_inicio, fila_fin, col_inicio, col_fin):
```

```
Si (fila_fin - fila_inicio = 1)
    Para i en (fila_inicio, fila_fin)
        Para j en (col_inicio, col_fin)
            Si (M[i][j] == A) y [(m > n y j < n) o (m < n y i < m)]:
              Devolver i, j

Sino
    Sea X = M[i][j],
    con
        i=[(fila_fin - fila_inicio) / 2 + fila_inicio]
        j=[(col_fin - col_inicio)/2 + col_inicio]
    (Si i y/o j no son enteros se redondea hacia arriba)

    % MATRIZ 2
    fila_inicio_M2 = fila_inicio
    fila_fin_M2 = i
    Si [(fila_fin - fila_inicio + 1) y (col_fin - col_inicio + 1) son pares]:
        columna_inicio_M2 = j + 1
    Sino:
        columna_inicio_M2 = j
    columna_fin_M2 = col_fin
    encontrar_numero(M,A,fila_inicio_M2,fila_fin_M2,
    columna_inicio_M2,columna_fin_M2)

    % MATRIZ 3
    Si [(fila_fin - fila_inicio + 1) y (col_fin - col_inicio + 1) son pares]:
        fila_inicio_M3 = i + 1
    Sino:
        fila_inicio_M3 = i
    fila_fin_M3 = fila_fin
    columna_inicio_M3 = col_inicio
    columna_fin_M3 = j
    encontrar_numero(M,A,fila_inicio_M3,fila_fin_M3,
    columna_inicio_M3,columna_fin_M3)



    Si X > A:
        fila_inicio_M1 = fila_inicio
        fila_fin_M1 = i
```

```
            columna_inicio_M1 = col_inicio
            columna_fin_M1 = j
            encontrar_numero(M,A, fila_inicio_M1 , fila_fin_M1 ,
            columna_inicio_M1 , columna_fin_M1)

        Si X < A:
            Si [(fila_fin - fila_inicio + 1) y (col_fin - col_inicio + 1) son pares]:
                fila_inicio_M1 = i + 1
            Sino:
                fila_inicio_M1 = i
            fila_fin_M1 = fila_fin
            Si [(fila_fin - fila_inicio + 1) y (col_fin - col_inicio + 1) son pares]:
                columna_inicio_M1 = j + 1
            Sino:
                columna_inicio_M1 = j
            columna_fin_M1 = col_fin
            encontrar_numero(M,A, fila_inicio_M1 , fila_fin_M1 ,
            columna_inicio_M1 , columna_fin_M1)
        Si (m > n y j < n) o (m < n y i < m):
            Devolver i, j
```

Complejidad temporal : Tal como se puede ver, el análisis de la complejidad temporal es idéntico al caso de las matrices cuadradas porque se convierte una matriz no cuadrada en no cuadrada.

Complejidad espacial : A diferencia del caso de la matriz cuadrada, se incurre en un aumento de la complejidad espacial por la fabricación de celdas necesarias para convertir la matriz no cuadrada en una cuadrada. Por lo tanto, la complejidad espacial es de $O(|n-m|)$.

4.3. Comentarios Parte 2

El análisis inicial de que greedy por naturaleza buscaría fraccionar cajas no tiene sentido. Uno puede trabajar con greedy perfectamente imponiendo números enteros. El análisis subsiguiente hace caso a esto. No hay nada en la naturaleza de greedy que busque un fraccionamiento. Tampoco es necesario complicarse tanto con explicaciones. Para demostrar que algo es falso basta con un contraejemplo.

Buen caso en el que greedy no funcionaría de manera óptima.

Se brinda una solución que no resuelve el problema pedido. Hay casos en el que las cajas se desordenan, yendo en contra de una de las condiciones del problema. Cito: Çomo en cada iteración elegimos las de mayor altura juntas, van a ir quedando las más bajas al final". Yo no puedo elegir que cajas quedan al final, el orden es fijo.

Más allá de esto, la solución propuesta es bastante compleja y confusa. Es difícil de comprender por qué se pasó de un problema de minimizar la sumatoria de alturas a maximizar alturas. Recomiendo buscar

una solución más simple. Se puede mejorar tanto la complejidad temporal como espacial (especialmente la espacial).

Comentarios programa Brinda código fuente, instrucciones de ejecución y un ejemplo. El algoritmo, por lo explicado en la parte 2 no resuelve el problema pedido.

Más allá de esto, el codigo es claro y legible.

4.4. Correcciones Parte 2

4.4.1. Solución por Greedy

El siguiente ejemplo muestra una propuesta *greedy* no óptima. Llegan 10 cajas en el siguiente orden: las primeras cuatro tienen altura de 10 cm, las siguientes tres son de 20 cm y las ultimas 3 de 5 cm. Dado que el ancho de las cajas es idéntico y en cada nivel entran a lo sumo 4 cajas, el algoritmo *greedy* acomoda las cajas de la siguiente forma:

```
|5cm|  |5cm|.                  -> 5cm total
|20cm| |20cm| |20cm| |5cm|    -> 20cm total
|10cm| |10cm| |10cm| |20cm|   -> 20cm total
                                ----------
                                45cm
```

Sin embargo, es fácil ver que este ordenamiento no es óptimo pues una mejor solución es colocar las 3 primeras cajas en el primer estante, las siguientes 4 en el estante 2 y las ultimas 3 en el tercer estante, como se muestra a continuación:

```
|5cm|  |5cm|  |5cm|              -> 5cm total
|20cm| |20cm| |20cm| |20cm|     -> 20cm total
|10cm| |10cm| |10cm|             -> 10cm total
                                 ----------
                                 35cm
```

Por lo tanto, una propuesta *greedy* no siempre brinda una solución óptima.

4.4.2. Solución teórica con Programación Dinámica

Nuestra solución para este problema en particular utilizando Programación Dinámica consta en evaluar para cada caja la altura total que tendrían todos los estantes si se combinase con cajas anteriores a ella, siempre y cuando entren todas en una misma repisa. Para cada caja nos quedaremos con la combinación de cajas que nos provea con la menor altura total.

Tenemos n cajas y nos toca evaluar la caja i (i es la posición en donde se encuentra la caja, ya que están ordenadas). Para ello evaluaremos $k = i$ situaciones. En la que en cada una agregaremos una caja anterior a la repisa de la caja i para ver qué altura total es menor. En una primera instancia evaluaremos la altura total si no pusiésemos ninguna caja en el estante en donde se encuentra la caja i. Esto sería $alturaTotal1 = alturaTotal + alturaCaja_i$. Luego evaluaremos agregar la caja anterior al estante, por lo que $alturaTotal2 = alturaTotal + max(alturaCaja_i; alturaCaja_{i-1})$, donde $alturaTotal$ es la altura

antes de evaluar la caja $i-1$. Esto nos dejará una simple comparación entre altura 1 y altura 2 en dónde me quedaré con la combinación de cajas que me dé la altura menor.

Este análisis seguirá hasta que no haya más lugar en el estante. Por ejemplo, si aún quedara espacio de la primera pasada, y la $alturaTotal_2$ fuese menor a la $alturaTotal_1$, se compararía la $alturaTotal_2$ contra la $alturaTotal_3 = alturaTotal + max(alturaCaja_i; alturaCaja_{i-1}; alturaCaja_{i-2})$. Cuando no haya más lugar en el estante, se pasará a la siguiente caja y se procederá a hacer lo mismo. El algoritmo termina cuando ya se evaluaron todas las combinaciones posibles para todas las cajas.

Se trata de un problema de programación dinámica porque contiene los siguientes aspectos: una subestructura óptima, ya que en cada subproblema que se nos plantea (agregar una caja a una repisa o no) buscamos la solución óptima que nos acerca a la solución óptima global. En cada comparación de alturas estamos buscando la combinación que nos de la menor altura; y por otro lado, contamos con subproblemas superpuestos, esto es así porque entre todos los subproblemas hay subproblemas en común. Por ejemplo, la caja 3 puede evaluar agregar en el estante la caja 2 porque entra, luego la caja 4 puede evaluar agregar la caja 3 y también la caja 2 porque entra también en la repisa. Esto nos da la posibilidad de realizar memorización que consiste en reutilizar las soluciones óptimas de subproblemas calculados anteriormente para buscar la solución óptima de subproblemas próximos a calcular.

Pseudocódigo:

```
Sea E[i] el estante donde se debe colocar la caja i
Sea HF[i] la altura total minima alcanzada al colocar la caja i
Sea C la lista de cajas
Sea N la cantidad de cajas
Sea H[i] la altura de la caja i
Sea W[i] el largo de la caja i
anchoMaximo = L
Para cada caja i desde 1 a N+1:
    alturaEstante = H[i-1]
    largoEstante = W[i-1]
    HF[i] = HF[i-1] + alturaEstante
    Para cada caja j desde i-1 a 1:
        nuevoLargo = largoEstante + W[j-1])
        Si nuevoLargo <= anchoMaximo:
            alturaEstante = max(alturaEstante, H[j-1]
            largoEstante = nuevoLargo
            nuevaAltura = alturaEstante + HF[j-1]
            Si nuevaAltura <= HF[i]:
                HF[i] = nuevaAltura
                Agregar j a E[i]
Retornar HF[N], E
```

Relación de recurrencia:

Para guardar la altura óptima actual de cada caja y su combinación pensamos en un vector, este contendrá la altura máxima de todos los estantes para la mejor combinación posible de esa caja y sus cajas anteriores. El hecho de que guarde la mejor altura acumulada es muy importante ya que es lo que nos posibilita a reutilizar las soluciones. Este vector tendrá tamaño $n + 1$ porque usaremos la primera posición del vector como caso base. Su altura será cero, esto nos ayudará para el cálculo de las alturas.

Sea cajas el vector con todas las cajas ordenadas según su codigo, siendo i la posición de la caja evaluada actual y j la posición de alguna caja anterior a la caja i con $j < i$

```
ALT(0) = 0
ALT(i) = MIN(ALT(i);ALT(j) + cajas[i].altura)
```

Como estructuras proponemos almacenar todos los datos en distintas listas indexadas por el número de caja.

4.4.3. Análisis de complejidad de la solución teórica

En cuanto a la complejidad temporal, el primer ciclo For tiene complejidad $O(N)$ ya que debe recorrer todas las cajas, las asignaciones de variables dentro del ciclo tienen complejidad $O(1)$. Luego, dentro hay otro ciclo For de complejidad $O(N)$ ya que en el peor caso deberá recorrer todas las cajas nuevamente. Las operaciones dentro de este ciclo son $O(1)$ ya que son comparaciones y operaciones de suma. La complejidad temporal es entonces $O(n^2)$. En cuanto a la complejidad espacial, esta será de $O(N)$ ya que los datos de alturas y estantes se almacenarán en listas de a lo sumo $N + 1$ elementos.

4.4.4. Comandos para ejecutar solución práctica

La solución se realizó utilizando Python, el código junto con un archivo de ejemplo se encuentra en la carpeta almacen dentro de la carpeta del trabajo práctico. El programa se ejecuta de la siguiente manera:

```
python3 ./tp2.py file_path largo_repisas num_cajas
```

4.4.5. Análisis de complejidad de la solución práctica

Respecto a la complejidad temporal:

- Dentro de la función *readFile* se recorre el archivo con la información de las cajas, por lo cual tendrá complejidad $O(N)$, donde N es la cantidad de cajas.
- La función *imprimirCajas* tiene un ciclo While que se ejecuta, en el peor caso, N veces; dentro de este ciclo hay un ciclo For que recorre cada estante, en el peor caso tendrá una complejidad de $O(N)$ si hubiese una caja en cada estante. Por lo tanto, la complejidad es de $O(N^2)$
- La función *disponerCajas* tiene dos ciclos que en el peor caso tendrán una complejidad de $O(N^2)$. El primer ciclo recorre cada caja y el segundo vuelve a recorrer las cajas pero en sentido inverso. Las demás operaciones son $O(1)$.

Respecto a la complejidad espacial, todos los datos se almacenan en listas. La lista que guarda la información de en qué estante debe colocarse cada caja, tendrá a lo sumo N elementos ya que en el peor

caso se necesitará un estante por caja; asimismo la lista que guarda la información de las alturas tendrá a lo sumo N elementos. Entonces, la complejidad espacial es $O(N)$

Referencia a la documentación: https://wiki.python.org/moin/TimeComplexity

4.5. Comentarios Parte 3

Se brinda una breve explicación de los 3 métodos. Muchas de las respuestas son muy similares al apunte. Traten de que no sea un copy-paste con algunos pequeños cambios.

En la comparación de métodos es correcto que mencionen que Programación Dinámica pierde valor si un algoritmo no suele usar resultados anteriores. Más allá de esto, faltaría hacer una mejor comparación entre los métodos: complejidades, dificultad de implementación, legibilidad de código, recursividad, lo que se les ocurra.

De greedy dice "en esta técnica se resolverá siempre sobre un mismo arreglo de modo que conservará la complejidad". ¿Que sería conservar la complejidad?¿Por qué podría afirmar esto para una resolución greedy?

4.6. Correcciones Parte 3

4.6.1. Un poco de teoría

Breve descripción de cada metodología Los algoritmos **Greedy** se utilizan como una metodología de resolución de problemas de optimización, ya sea para minimizar o maximizar una variable. La idea es dividir el problema en subproblemas con una jerarquía entre ellos. Cada subproblema se va resolviendo iterativamente mediante una elección, basándose en la situación local. A medida que los subproblemas se van resolviendo según la heurística elegida, se habilitan nuevos subproblemas a resolver.

La estrategia de **División y Conquista** consiste en dividir el problema en subproblemas de menor tamaño tal que los subproblemas puedan resolverse de manera independiente. Además cada subproblema conserva todas las características del problema original. Esto permite que este mismo, pueda a su vez ser subdividido en nuevos subproblemas de un menor tamaño. Se produce un *proceso recursivo* donde se continúa esta división hasta un *caso base* donde el problema tiene un tamaño lo suficientemente pequeño para resolverse de forma trivial. De esta manera se van resolviendo los subproblemas, hasta tener todas las soluciones y *combinarlas* para obtener un solución global al problema original.

Al igual que Greedy, la metodología de **Programación Dinámica** resuelve problemas de optimización donde se quiere maximizar o minimizar un resultado. Divide el problema en subproblemas con una jerarquía entre ellos. Además, cada subproblema puede ser reutilizado en diferentes subproblemas mayores. Esto es beneficioso por su propiedad de *memorización*, técnica que consiste en almacenar los resultados de los subproblemas previamente calculados para evitar repetir su resolución cuando vuelva a requerirse.

Propiedades requeridas En el caso de **Greedy**, si bien puede crearse un algoritmo de este tipo para cualquier problema de optimización, la solución que se obtiene no será siempre óptima. Para garantizar su optimalidad, el problema debe cumplir ciertas propiedades:

- Subestructura óptima: la solución al problema contiene dentro de sí la solución óptima de sus subproblemas
- Elección greedy: en cada paso, se debe hacer una elección entre todos los posibles subproblemas.

En cuanto a **División y Conquista**, no todo problema se puede resolver mediante esta metodología, al igual que Greedy debe cumplir con la propiedad de subestructura óptima. Es decir que el problema debe poder separarse en subproblemas de manera recursiva y al encontrar la solución óptima de estos, encontrar la solución óptima global. La diferencia es que la división, por División y conquista se puede pensar en anchura, en cambio en Greedy es en profundidad.

En el caso de **Programación Dinámica**, al igual que las metodologías Greedy y División y conquista, tiene la característica de dividir el problema en diferentes subproblemas. Por lo tanto, el problema debe cumplir con la característica de subestructura óptima para poder ser resuelto en forma óptima. Además debe cumplir con la propiedad de subproblemas superpuestos, es decir que a la hora de resolver el problema, existe la posibilidad de aprovechar los resultados almacenados de ciertos subproblemas(caso base) para resolver otros subproblemas.

Elección de metodología Si tomamos un problema que puede resolverse por cualquiera de los tres algoritmos, en principio no se tendría una técnica superior a la otra, dado que dependerá del problema mismo y las prioridades que se tomen a la hora de resolverlo.
Podemos considerar que si queremos obtener una solución que implique una mejor complejidad temporal, nos vamos a inclinar por el algoritmo de Programación Dinámica. Esta técnica guarda información que se obtiene en cada subproblema que se resuelve, y de ese modo podríamos lograr optimizar iteraciones en subproblemas de un grado mayor. Aunque también podría ocurrir que nos topemos con un problema que no tenga un considerable número de subproblemas que se repitan entre sí y esto implicaría que se van a generar múltiples resoluciones a almacenar y luego no serán reutilizadas. De modo que usaríamos un enorme espacio de más.
Por un lado, el algoritmo de División y Conquista si bien tiene similitud con la Programación Dinámica, cada subproblema no tiene relación alguna con otro subproblema, ya que son independientes, y la resolución de cada uno de estos subproblemas llevará más tiempo. Por lo que, en cuanto a complejidad temporal y espacial probablemente sea el menos elegido(de todas maneras habría que evaluar cada caso particular).
Por otro lado, la opción Greedy resulta una estrategia a utilizar que suele ser óptima en la mayoría de los casos ya que logra resolver el problema de manera óptima. Es por esto que podría llegar a lograr una muy buena complejidad temporal.
Finalmente, no podemos concluir en cuál estrategia será mejor debido a que esa respuesta está directamente relacionada con los requisitos y prioridades que se especifiquen en un enunciado (o pedido) y a su vez condicionados por los recursos que dispongamos en cuanto a los ordenadores que se utilicen para aplicar una de las estrategias y llegar a una solución. Lo que se puede afirmar es que si se desea minimizar el uso de recursos de memoria por parte de una computadora entonces sería prudente utilizar el algoritmo Greedy. Ahora bien si lo que se quiere es preservar un tiempo de ejecución menor posible con un uso

menor del procesador, inclusive si esto implica usar más memoria, la elección del algoritmo indicado sería el de Programación Dinámica.

5. Tercera Entrega

5.1. Comentarios Parte 2

5.1.1. Solución práctica

Impresión de cajas y estantes Se corrige la impresión de cajas en estantes y pasa la pruebas mandada por el corrector.

En este caso se crea un vector dónde cada posición corresponde a la ultima caja de un estante x y el valor dentro de cada posición corresponde a la primera caja que se agregará a ese estante. Se adicionan además las cajas que están entre medio de los dos extremos. No se modifica la complejidad.

La solución se realizó utilizando Python, el código junto con un archivo de ejemplo se encuentra en la carpeta almacen dentro de la carpeta del trabajo práctico. El programa se ejecuta de la siguiente manera:

```
python3 ./tp2.py <file_path> <largo_repisas> <num_cajas>
```