Universidad de Buenos Aires

Facultad de Ingeniería

Trabajo Practico 2

[75.29/95.06] Teoría de Algoritmos
Segundo cuatrimestre 2022

- Alumno: Aguirre Juan - 102227 - jaguirreb@fi.uba.ar
- Alumno: Mosquera Sebastian - 106744 - smosquera@fi.uba.ar
- Alumno: Ramos Federico Andres - 101640 - faramos@fi.uba.ar

Índice

1. Parte 1: El experimento científico

En primer lugar planteamos el problema a resolver, Tenemos una matriz M de $n \times n$, de la cual tanto las filas como las columnas están ordenadas de menor a mayor, a índice mas grande el valor es mayor. Nos interesa poder buscar el índice de un valor especifico dentro de la matriz. A continuación vamos a ver distintas estrategias para resolver el problema.

1.1. Fuerza Bruta

La estrategia por fuerza bruta seria recorrer la matriz hasta encontrar el numero que se busca. Por cada paso se realiza la siguiente comparación, siendo n el numero que buscamos y A la matriz

$$M[i, j] == n$$

Si se encuentra el valor se retorna la posición $[i, j]$, en otro caso $[-1, -1]$. Entonces suponiendo que nuestra matriz es de $n \times n$, en el peor de los casos tendríamos que recorrer toda la matriz realizando n^2 comparaciones, resultando así una complejidad temporal de

$$O(n^2)$$

La complejidad espacial es $O(1)$ porque solo hay que almacenar n.

1.2. División y Conquista

Para desarrollar una solución que utilice división y conquista, nos vamos a aprovechar de las propiedad de ordenamiento de las filas y las columnas.

1.2.1. Algoritmo y Estructuras de Datos

Primero damos algunas definiciones

- A es una matriz de $n \times n$, en la cual tanto las filas como las columnas tienen números ordenados de forma creciente.
- DIM es una tupla de dos elementos que contiene las dimensiones de la matriz,

$$DIM = (filas,\ columnas)$$

- $LIMITE$ es una tupla que representa un índice de la matriz, inicializado en

$$(0, n)$$

- v es el valor que buscamos

El algoritmo entonces es

1. Nos paramos en la esquina superior derecha de la matriz. Siendo mas precisos en la posición $A[0, n]$, siendo n el numero de columnas de la matriz.
2. Verificamos si el valor en esa posición es el que buscamos, si lo es retornamos el índice.
3. Si el valor en esa posición es mayor al que buscamos, eliminamos la fila actual de la matriz; si es menor eliminamos la columna actual.
4. Realizamos el llamado recursivo con la nueva matriz.
5. Si nos quedamos sin filas o columnas, eso significa que el algoritmo finalizo

A continuación mostramos el pseudocódigo

Algorithm 1 Algoritmo Parte 1: El Experimento Científico

```
procedure BUSCARNUMERO(A, DIM, LIMITE, v)
    if LIMITE.columna < 0 or LIMITE.fila > (DIM.filas - 1) then
        return nil
    end if

    if A[LIMITE.fila, LIMITE.columna] == v then
        return LIMITE
    end if

    if A[LIMITE.fila, LIMITE.columna] < v then
        LIMITE = (LIMITE.fila + 1, LIMITE.columna)
        return BUSCARNUMERO(A, DIM, LIMITE, v)
    else
        LIMITE = (LIMITE.fila, LIMITE.columna - 1)
        return BUSCARNUMERO(A, DIM, LIMITE, v)
    end if

end procedure
```

1.2.2. Relación de Recurrencia y Complejidad

La complejidad temporal de la solución es $O(n)$, ya que en el peor de los casos se realizan $2n$ comparaciones (Siendo A una matriz de $n \times n$). Luego la complejidad espacial de la solución es de $O(1)$ ya que lo único que se guarda es el valor en **LIMITE**.

1.2.3. Dimensión de la matriz

Si se modificara la dimensión de la matriz, de manera tal que ahora A es una matriz de $n \times m$. El algoritmo no se debería modificar, ya que en ningún momento se hace uso de la información de que la matriz es cuadrada. El algoritmo propuesto es independiente de la forma de los matriz

1.2.4. Ejemplo de funcionamiento

Presentamos el siguiente ejemplo donde la matriz A es de dimensión 4×4, y nos interesa buscar la posición del 18

$$A = A_0 = \begin{bmatrix} 1 & 5 & 9 & 25 \\ 2 & 7 & 11 & 28 \\ 3 & 18 & 98 & 101 \\ 4 & 42 & 105 & 108 \end{bmatrix}$$

En primer lugar nos paramos en la posición $(0, 3)$, donde $A[0, 3] = 25$. Como $25 > 18$, vamos a eliminar la columna mas a la derecha resultando

$$A_1 = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 7 & 11 \\ 3 & 18 & 98 \\ 4 & 42 & 105 \end{bmatrix}$$

Ahora $A = A_1$, es una matriz de 4×3, entonces ahora nos paramos en la posición $(0, 2)$,

obteniendo así $A[0,2] = 9$, el valor es menor al que buscamos por lo que eliminamos la fila

$$A_2 = \begin{bmatrix} 2 & 7 & 11 \\ 3 & 18 & 98 \\ 4 & 42 & 105 \end{bmatrix}$$

Al proceso se lo repite hasta llegar a la matriz A_4; como $A[0,2] = 11 < 18$, se elimina la fila actual

$$A_3 = \begin{bmatrix} 3 & 18 & 98 \\ 4 & 42 & 105 \end{bmatrix}$$

Por ultimo, $A[0,2] = 98 > 18$, por lo que se elimina la columna

$$A_4 = \begin{bmatrix} 3 & 18 \\ 4 & 42 \end{bmatrix}$$

Vemos en A_4 que si evaluamos A en la posición $(0,1)$ resulta $A[0,1] = 18$. Este valor coincide con el que buscamos. Podemos recuperar el índice, sumando 1 a la fila actual en la que se encuentra el valor por cada fila que eliminamos. Como eliminamos dos filas el índice real resulta

$$(2,1)$$

Por lo tanto el valor 18 se encuentra en el índice $(2,1)$ de la matriz A.

2. Parte 2: El Almacén

2.1. Enunciado

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.

2.2. Solución Greedy

2.2.1. Introducción

Se propone el siguiente algoritmo:
Se agregan tantas cajas como sean posibles a la repisa hasta superar el ancho de la repisa. Una vez que se llega a llenar la repisa, se sigue la misma lógica con la siguiente repisa hasta no tener más cajas que agregar a la bodega.

2.2.2. Optimalidad

En este algoritmo nuestra elección Greedy no es óptima debido a que, tener la mayor cantidad de cajas por repisa (nuestra resolución Greedy del subproblema) no nos lleva a la solución global óptima. Veamos un ejemplo:

Tenemos las siguientes 3 cajas y cada repisa tiene un máximo de 2 de ancho:

- Caja 1: Ancho 1 - Altura 1.
- Caja 2: Ancho 1 - Altura 3.
- Caja 3: Ancho 1 - Altura 3.

Si utilizamos nuestro algoritmo Greedy, nos quedarían 2 repisas, la primera que la Caja 1 y la Caja 2, mientras que la segunda repisa nos quedaría con la Caja 3.Ambas repisas tienen altura 3, lo que da como resultado una altura total de 6. La solución óptima seria, la primer repisa solo con la Caja 1 y la segunda repisa con las cajas restantes. Esta distribución resulta en una altura total 4, menor a nuestra solución Greedy, demostrando que el algoritmo propuesto no es óptimo.

2.3. Solución Dinámica

2.3.1. Introducción

Se propone el siguiente algoritmo:

Se va iterando por cada una de las cajas, cada caja empieza en su propio estante y se suma su altura a una variable que almacena la altura total de todas las repisas. Luego, se itera hacia atrás insertando cajas anteriores en la nueva repisa, por cada iteración se calcula nuevamente la altura total. Una vez que se queda sin espacio esta nueva repisa, se guarda la configuración con la menor altura total.

2.3.2. Optimalidad

Analizando el algoritmo, podemos ver que cumple con todas las propiedades de un algoritmo de programación Dinámica. Al almacenar los resultados de los subproblemas anteriores, la altura total mínima para las cajas anteriores, cumplimos con la propiedad de subproblemas superpuestos. En el caso de la subsestructura óptima, cada iteración resuelve el subproblema de minimización de la altura total para las i cajas, siendo i el index de la iteración.

Ya demostrado que es un algoritmo de programación dinámica, falta demostrar que sea la solución óptima. Al empezar poniendo la caja en una nueva repisa, los unicos movimientos que nos quedan disponibles es mover la mayor cantidad de cajas a la nueva repisa. Si una de estas cajas mejora la altura total no se va a omitir debido a como esta estructurada la solución. Siendo S una solución óptima y un conjunto R que es el resultado de nuestra solución, si S no puede tener más repisas porque es un absurdo, debido a que siempre agregamos la nueva caja en una nueva repisa. En el caso de que S tenga menos repisas, significa que hay más cajas en alguna repisa, de tal manera que la altura total disminuya, pero devuelta es absurdo, debido a que siempre que se agrega una nueva caja, se prueba todas las cajas anteriores que puedan entrar en la misma repisa y se busca el orden que obtenga la menor altura total.

2.4. Implementación

Para implementar el algoritmo se utilizan las siguientes definiciones:

- Definimos **N** como la cantidad de cajas a almacenar.
- Un **Array** de tamaño **N** donde se almacena la altura y ancho de cada caja. Definimos **C** tal que para cada caja i, **C**[i] sea la altura y el ancho de la caja.
- Un **Array** de tamaño **N+1** donde se almacena la altura total mínima despues de agregar una caja. El primer elemento del **Array** es 0 y representa una bodega sin ninguna caja. Definimos **AT (Altura total)** tal que para cada caja i, **AT**[$i+1$] sea la altura total mínima del problema hasta la caja i.
- Un **Array** de tamaño **N** donde se almacena a que repisa pertenece cada caja. Definimos **R (Repisas)** tal que para cada caja i, **R**[i] sea la repisa en donde se ubica i.

2.4.1. Pseudocódigo

Algorithm 2 Algoritmo Parte 2: Almacén

```
C Array de tamaño N                                  ▷ Almacena la altura y ancho de cada caja
AT Array de tamaño N+1 inicializada en 0             ▷ Almacena la altura total hasta la caja
R Array de tamaño N                                  ▷ Almacena a que repisa pertenece cada caja
AnchoMaxRepisa int                                   ▷ Ancho máximo de la repisa
BoxesInShelve int      ▷ Cajas en la repisa, utilizado para saber que cajas fueron movidas
NumberOfShelves int                                  ▷ Cantidad de repisas
for i = 1; i <= N; i++ do                            ▷ Itero por cada una de las cajas
    NumberOfShelves + = 1
    alturaCajaActual, anchoCajaActual = C[i - 1]

    anchoRepisa = anchoCajaActual
    alturaRepisa = alturaCajaActual

    alturaTotal = AT[i-1] + alturaCajaActual

    for j = i - 1; j >= 0; j++ do ▷ Itero las cajas anteriores hasta que no entren en la repisa
        alturaCaja, anchoCaja = C[j - 1]
        if anchoCaja + anchoRepisa > AnchoMaxRepisa then
            break
        end if

        anchoRepisa + = anchoCaja
        alturaRepisa = Max(alturaCaja, alturaRepisa)
        nuevaAlturaTotal = AT[j - 1] + alturaRepisa

        if nuevaAlturaTotal < alturaTotal then
            alturaTotal = nuevaAlturaTotal
            BoxesInShelve = range(j, i + 1)          ▷ Los index de las cajas que movimos
        end if

    end for
    AT[i] = alturaTotal
    for w = 0; w < N; w++ do          ▷ Actualizamos la repisa de cada caja que movimos
        R[BoxesInShelve[w] - 1] = NumberOfShelves
    end for
end for

initialNumberOfShelve = R[0]
Counter = 1
for w = 0; w < N); w++ do ▷ Se cuenta por cada valor de repisa distinto para que quede un
Array que empieza en 1
    if initialNumber ! = R[w] then
        counter += 1
        initialNumberOfShelve = R[w]
    end if
    R[w] = counter
end for
```

2.4.2. Complejidad Temporal

Sea **n** la cantidad de cajas que queremos almacenar en la bodega, el algoritmo itera las **n** caja. Por cada iteración, prueba cuantas cajas procesadas anteriormente entran en la repisa. En el peor de los casos, estamos en la ultima caja y todas las cajas anteriores (las **n** cajas) entran en la repisa, obteniendo $O(n)$. También, luego de esta iteración debe actualizar en que repisa se almacena cada una de las cajas, que en el peor de los casos también es $O(n)$. Entonces como resultado, llegamos a $(n \times (n + n)) + n$ que nos da una complejidad temporal $O(n^2)$.

2.4.3. Complejidad Espacial

Sea **n** la cantidad de cajas que queremos almacenar en la bodega, se utiliza un array de tamaño **n** para almacenar a que repisa corresponde cada caja. Además se utiliza un array de tamaño **n+1** para almacenar el resultado de cada iteración. Finalmente, nos queda una expresión $n + (n + 1)$ que resulta en una complejidad espacial de $O(n)$

2.5. Implementación del Algoritmo en Python

2.5.1. Como ejecutar el script

El **main.py** se puede ejecutar con cualquier versión de Python igual o superior a **3.9.10**. Para ejecutar el programa se debe pasar por argumento:

- −−n: Cantidad de cajas.
- −−w: Ancho máximo de cada repisa.
- −−f: Nombre del archivo.

Ejemplos:

```
python3 main.py −−n 9 −−w 3 −−f test1.txt
python3 main.py −−n 6 −−w 2 −−f test2.txt
```

2.5.2. Complejidad del problema

Debido a que solo utilizamos Arrays en el algoritmo propuesto, solo debemos analizar una operación a la hora de calcular la complejidad temporal en Python. Esta operación es el indexado en una lista, que es una operación $O(1)$ según la documentación de Python. Con este análisis se puede concluir que la complejidad temporal de la solución escrita en Python se mantiene $O(n^2)$

3. Parte 3: Un poco de teoría

3.1. Greedy

Una solución *Greedy* consiste en dividir el problema en distintos subproblemas con una jerarquía entre ellos, e ir resolviéndolos iterativamente mediante una decisión heurística, que elija la mejor solución local, llamada *Elección Greedy*. Esta heurística nos va a habilitar nuevos subproblemas. La idea es que a medida que encontremos las soluciones óptimas locales, estas nos acerquen a la solución óptima global.

Para un mismo problema pueden existir distintos algoritmos Greedy, pero no todos resultan en la solución óptima. Para que un problema puede resolverse mediante una metodología Greedy debe cumplir con las siguientes dos propiedades.

- Elección Greedy
- Subestructura óptima

Como se explico anteriormente la elección Greedy consiste en una heurística que nos acerque a la solución óptima global seleccionando la mejor solución local. Los subproblemas se encuentran condicionados siempre por las elecciones tomadas en los anteriores problemas y condicionan a los siguientes.

Por ultimo, tiene que existir una *Subestructura óptima*, esto implica que la solución global del problema debe contener en su interior las soluciones óptimas de sus subproblemas. Esto significa que al aplicar iterativamente la elección Greedy, resolveremos los subproblemas óptimamente, lo que nos acercara al óptimo global.

3.2. División y conquista

Resolver un problema utilizando *División y Conquista* consiste en dividir la instancia del problema, en subproblemas de menor tamaño, los cuales son independientes entre si. Estos subproblemas tienen la particularidad de que conservan todas las características del problema original, lo que permite que estos subproblemas puedan volver a subdividirse en problemas de menor tamaño. Esta división continua recursivamente hasta llegar al caso base, donde el problema el problema tiene un tamaño lo suficientemente pequeño como para resolverlo de forma trivial. La parte de conquista de la estrategia se da cuando se produce resolución de todos los subproblemas. Por ultimo se combinan las soluciones obtenidas para obtener la del problema general.

Para que un problema se puede resolver utilizando división y conquista el mismo tiene que presentar una *Subestructura Óptima*. Dicho de otra forma, la resolución óptima de los subproblemas, nos lleva a la solución óptima global.

3.3. Programación Dinámica

La programación dinámica es una metodología para resolver problemas de optimización, que al igual que en los casos anteriores divide el problema a resolver en nuevos subproblemas que poseen una jerarquía entre ellos.La idea del método es obtener todas las soluciones y determinar cual es la óptima. La particularidad de esta metodología se encuentra en que los subproblemas pueden volver a ser utilizados en subproblemas mayores, para esto se hace uso de una técnica llamada *Memorización*. La misma consiste en almacenar los resultados de los subproblemas calculados previamente, para que puedan ser utilizados posteriormente. El uso de esta técnica hace que se reduzca la cantidad total de subproblemas a calcular, lo que reduce la complejidad temporal de las soluciones. Notamos que la memorización no es gratuita, sino que provoca que la complejidad espacial empeore al tener que almacenar los resultados.

Para que un problema pueda resolverse de forma óptima mediante un algoritmo de programación dinámica, el mismo debe cumplir con las siguientes dos propiedades

- Subestructura óptima
- Subproblemas superpuestos.

La primera ya fue explicada en los casos anteriores. La segunda propiedad es la que distingue al método, ya que que un problema tenga subproblemas superpuestos implica, que en la resolución de los subproblemas vuelven aparecer subproblemas resueltos previamente.

3.4. ¿Cual metodología elijo?

Primero vamos a notar que no hay una metodología que sea inherentemente mejor que la otra, cual elijamos va a depender de nuestro problema que queremos resolver y las restricciones del sistema sobre el cual lo queremos resolver.

Si suponemos que el problema que queremos resolver se puede resolver por las tres metodologías, la elección de cual vamos a usar va a depender de distintos factores. En primer lugar, la situación mas trivial seria un problema de tamaño pequeño en el cual las diferencias entre

las complejidades asintóticas de las soluciones (si las hubiera), serian despreciables, debido al tamaño del problema. Entonces bajo estas condiciones los métodos son indistinguibles, ya que los tres nos llevan al resultado que queremos.

Ahora, si el tamaño del problema se vuelve apreciable, entonces las diferencias entre las complejidades temporales y espaciales empiezan a pesar mas (nuevamente, si es que las hay). En esta situación vamos a preferir la opción mas eficiente, teniendo en cuenta las limitaciones físicas del sistema. Por ejemplo, en sistemas con memoria reducida, una solución con programación dinámica por ahí no es la mejor solución debido a al alto uso de memoria debido a la memorización.