Instituto Politécnico Nacional

Escuela Superior de Cómputo

Evolutionary Computing

Lab Session 1: Introduction to Dynammic Programming

Alumno: David Arturo Oaxaca Pérez

Grupo: 3CV11

Profesor: Jorge Luis Rosas Trigueros

Marco teórico

En la resolución de los problemas planteados a lo largo de esta práctica se usó la programación dinámica, la cual consiste en una técnica matemática para la optimización y un método de programación que utiliza la solución de problemas seleccionados usando una serie de decisiones en forma secuencial. Esto nos proporciona un procedimiento sistemático para encontrar una combinación de decisiones que maximice la efectividad total al descomponer el problema en etapas, las cuales pueden ser completadas por una o más formas (Estados) y enlazando cada etapa a través de cálculos recursivos.

La idea general de la programación dinámica es una optimización sobre la recursividad simple, por ejemplo, cuando vemos en una solución recursiva una solución que repite llamadas para resolver problemas con los mismos inputs, el uso de la programación dinámica nos permitiría optimizar esto. Cuando hablamos de la programación dinámica como método de programación. lo que se pretende es guardar los resultados de los subproblemas resueltos para evitar tener que resolverlos nuevamente si se vuelven a necesitar posteriormente, esta simple optimización reduce la complejidad temporal de un problema, pasando de ser una complejidad exponencial a una polinomial.

En la programación dinámica existen dos acercamientos a la hora de guardar los resultados de los subproblemas que podrían utilizarse después, a estos acercamientos se les conoce como métodos de tabulación y la memoización.

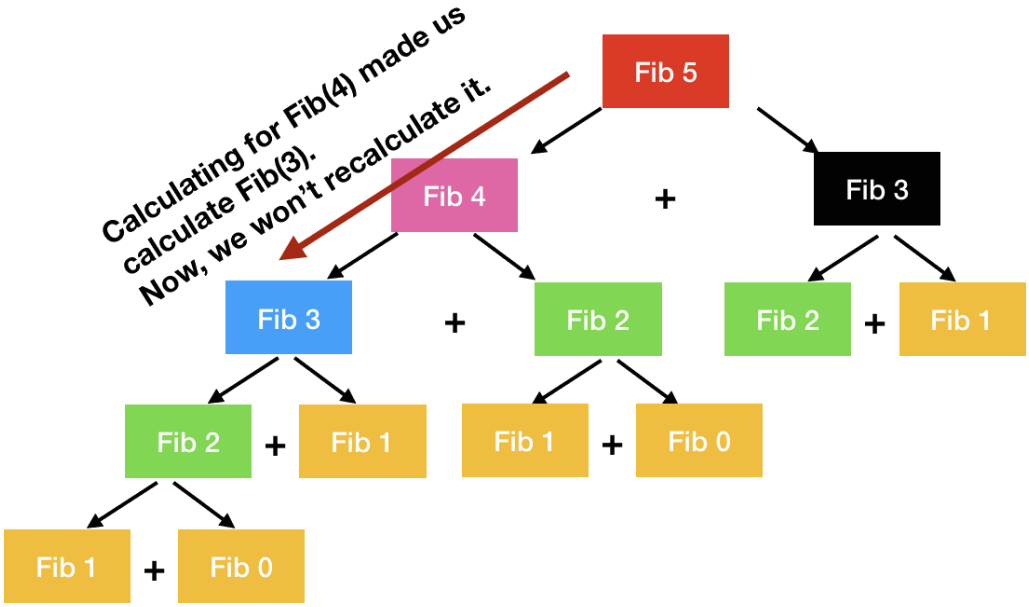
Al método de la tabulación se le conoce como “Bottom Up”, como su nombre lo sugiere, consiste en empezar desde abajo e ir acumulando respuestas hasta llegar a la solución del problema. Esto se puede ver más claro si, por ejemplo decimos, que el estado de nuestro problema de programación dinámica es m[x] siendo m[0] nuestro estado base y m[n] nuestro estado destino para la solución del problema, si empezamos la transición de desde nuestro estado base y seguimos su relación con la transición de los estados hasta alcanzar nuestro estado destino podemos ver claramente que el estado base es el “bottom” y de ahí partimos hasta avanzar el resultado deseado.

A este patrón de programación dinámica se le llama tabulación porque al irlo programando se puede ver que la tabla de programación dinámica es secuencialmente llenada y se va accediendo directamente a estados calculados previamente que se encuentran en la misma tabla hasta llegar al resultado del problema.

En cuanto el método de memoización, a este se le conoce como el método “Top-Down”, la idea detrás de este algoritmo consiste en intentar resolver el problema de manera natural e ir guardando las soluciones calculadas en el proceso para no repetir la resolución de subproblemas que se vayan necesitando hasta llegar a la

solución del problema. La siguiente imagen puede servir como ejemplo para clarificar este método:

Figura 1. Ejemplo del cálculo de un numero Fibonacci por medio de la memoización



Nota: CodesDope. (2021, 23 de septiembre). *Dynamic Programming | Top-Down and Bottom-Up approach | Tabulation V/S Memoization*. https://www.codesdope.com/course/algorithms-dynamic-programming/

Para calcular el Fibonacci de 5 mediante el método de memoización empezamos a llamar a una función recursiva y si ya hemos calculado previamente esta solución, como el caso del Fibonacci de 2 entonces ya no lo volvemos a calcular.

La principal diferencia de este método con el de tabulación es que únicamente se van resolviendo los subproblemas que son requeridos para la solución del problema.

Una comparativa de estos métodos se puede observar en la siguiente tabla que compara algunas de sus características:

Tabla 1. Comparativa de los métodos usados para la programación dinámica.

	Tabulación	Memoización
Estado	Las relaciones entre transiciones de estados puedo ser una parte complicada de idear.	La transición de estados es fácil de pensar.
Código	El código se puede complicar cuando se	El código es relativamente sencillo y menos complicado

	tiene más condiciones para el problema	
Velocidad	Es bastante rápido ya que directamente se accede a los previos estados en la tabla	Es más lento ya que se hacen varias llamadas recursivas
Solución de subproblemas	Todos los subproblemas son resueltos al menos una vez, lo cual supera a un algoritmo de memoización por un factor constante	Este método tiene la ventaja de que resuelve únicamente los subproblemas que son requeridos.
Entradas de tablas	En esta versión se empieza en la primer celda de la tabla y a partir de ahí se van llenando una por una hasta llegar a la solución final.	No todas las entradas de la tabla son necesariamente llenadas, sino que se van llenando según se vayan necesitando.

Nota: GeeksForGeeks. (2021, 22 de septiembre). *Tabulation vs Memoization - GeeksforGeeks*. GeeksforGeeks. https://www.geeksforgeeks.org/tabulation-vs-memoization/

Problema de la mochila 0-1 (Knapsack problem)

Este es un problema de optimización combinatoria que puede ser resuelto mediante un método de la programación dinámica, dada una cantidad n de objetos en los cuales cada uno tiene un peso y un valor y una mochila de capacidad máxima W se debe obtener el valor máximo total decidiendo si se guarda un objeto en la mochila o no.

Este es un problema con muchas aplicaciones, puede ir desde los ejemplos básicos, como que equipaje elegir cuando existe un límite de peso con el que vas a viajar como en los aeropuertos hasta lo que puede ser la toma de decisiones para saber que comerciales proyectar en eventos con audiencia masiva como puede ser el superbowl.

Su definición matemática puede ser expresada como:

Maximizar

$$\sum_{i=1}^{n} v_i x_i$$

Sujeto a

$$\sum_{i=1}^{n} w_i x_i \leq W \; and \; x_i \in \{0,1\}$$

El primer problema que desarrollamos a lo largo de esta práctica fue el de la mochila 0-1, también conocido como el “Knapsack problem”, lo primero que se realizó en clase para empezar a entender cómo íbamos a plantear una solución en código para este problema. Primero se planteó un ejemplo sencillo con el cual veríamos como al progresar obtendríamos una solución, esto nos ayudó a idear una solución Bottom up posteriormente.

El ejemplo planteado fue el siguiente:

i = {1, 2, 3}

v = {6, 10, 12}

w = {1, 2, 3}

W = 5

Primero determinamos que sus dimensiones de complejidad consistían en la capacidad de carga de la mochila, en este caso W, y el número de elementos que tenemos, en este caso 3, pues i es la nomenclatura con la que tenemos declarada la lista de elementos en este caso.

Tras eso, empezamos a siguiente tabla que ilustra de manera muy gráfica la forma en que se resolverá el problema por medio de una solución “Bottom up”:

Tabla 2. Solución “Bottom up” para el problema de la mochila.

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	6	6	6	6	6
2	0	6	10	16	16	16
3	0	6	10	16	18	22

Como podemos observar en la tabla, nuestro resultado deseado se encuentra en la última celda, si la consideramos como un arreglo de dos dimensiones llamado “m”, encontraríamos que el valor máximo que se puede obtener con un peso menor al W usando los elementos de i se encuentra en m[i, w], con eso podemos partir para desarrollar un algorito y una serie de condiciones que nos permitan obtener la solución a este problema.

1) **m[0,w] = 0; m[i,0] = 0**
2) **m[i, w] = m[i-1, w] if wi > w**
3) **m[I,w] = max(m[i-1, w], m[i-1, w-w_i] + v_i)**

De manera que la solución la podremos encontrar en m[n, W], esto nos ayudara a la hora de programar este problema.

Problema de cambio de monedas (Change-making problem)

Este es un problema que trata la cuestión de encontrar el mínimo numero de monedas de ciertas denominaciones hasta encontrar una cantidad determinada de dinero, es decir, minimizar el número de monedas que se requieren para sumar W. Este problema tiene consideraciones notables como el hecho de que las denominaciones de las monedas deben estar dadas en un orden ascendente, además de que en la versión del problema tratada en esta práctica se asume que contamos con una cantidad infinita de monedas para cualquiera de las denominaciones.

Su definición matemática puede ser expresada como:

Minimizar

$$\sum_{i=1}^{n} x_i$$

Sujeto a

$$\sum_{i=1}^{n} w_j x_j = W$$

En este caso, para el planteamiento del problema en clase, se hizo algo similar, pero considerando las diferencias que este problema implica, como tomar en cuenta que todas las denominaciones de las monedas, que en este caso son nuestros ítems, deben de ir en orden ascendente, la primera denominación tiene que ser 1 y la consideración de que podemos tomar varias de estas hasta alcanzar el valor N del problema.

El problema planteado que se resolvió para ir formulando una serie de pasos que nos permitieran resolver esta clase de problemas fue el siguiente:

Primero analizamos los datos planteado, las denominaciones que tenemos y la cantidad que se requiere.

d = [1, 2, 5]

N = 7

Posteriormente trabajamos con una tabla similar a la que hicimos con el problema de la mochila, pero teniendo en cuenta otras consideraciones propias del problema.

Tabla 2. Solución “Bottom up” para el problema del cambio de monedas.

d \ N	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
2	0	1	1	2	2	3	3	4

5	0	1	1	2	2	1	2	2

Una vez que fuimos resolviendo la tabla anterior, se fue planteando una serie de condiciones:

1) **m[1,n] = n; m[i, 0] = 0**
2) **m[i, N] = m[i-1, n] if d_i > n**
3) **m[I, N] = min(m[i-1, n], m[I, n – d_i + 1])**

Problema de la subsecuencia común más larga (Longest Common Subsequence)

Este problema consiste, que, dadas dos secuencias de caracteres, se tiene que encontrar la subsecuencia más larga que está presente en ambas secuencias. Una subsecuencia es una secuencia que aparece en el mismo orden relativo, pero no necesariamente de manera continua.

Por ejemplo, en la secuencia “abcdefg” existen las subsecuencias “abc”, “abg”, “bdf”, “aeg”, “acefg”, … etc.

El implementar una solución usando un método de programación dinámica reduce la complejidad temporal de este problema de manera considerable, esta pasa de ser exponencial a ser polinomial.

La definición de la función para este problema es el siguiente conjunto:

$$LCS(X_i, Y_j) = \begin{cases} 0; \quad if\ i = 0\ or\ j = 0 \\ LCS(X_{i-1}, Y_{j-1}) \ \cap \ x_i; \quad if\ x_i = y_j \\ longest\left(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)\right); \quad if\ x_i \neq y_j \end{cases}$$

Este problema adicional a los vistos en clase es de interés, pues, aunque pueda parecer un tanto sencillo, al igual que los otros dos también tiene bastantes aplicaciones como puede ser la bioinformática. Para empezar a resolverlo se usó un método similar a los vistos en clase, primero lo planteamos usando una tabla y un caso de ejemplo.

Para el planteamiento de la tabla utilizamos las siguientes cadenas:

L1 = “AGGTAB”

L2 = “GXTXAYB”

El planteamiento de la tabla fue similar a lo anteriores, consideramos a los elementos de una cadena como el índice de las columnas y a los de la otra como los índices de las filas:

Tabla 2. Solución “Bottom up” para el problema de la subsecuencia común más larga.

L2 \ L1	0	A	G	G	T	A	B
0	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0	1	1	1	2	3	3
Y	0	1	1	1	2	3	3
B	0	1	1	1	2	3	4

En esta tabla consideramos a la columna y fila inicial como 0 en el caso de que una de las cadenas estuviera vacía. Para empezar a plantear los pasos que vamos a seguir en el desarrollo de una solución analizamos como se fue haciendo esta tabla y planteamos lo siguiente:

Si ***x*** es el tamaño de la cadena L1 y ***y*** es el tamaño de la cadena L2, entonces

m[y, x] = m[y-1, x] if L1[x] != L2[y]

m[y, x] = max(m[y-1, x], m[y, x-1] + 1)

Material y equipo

Herramientas de hardware: Para la realización de esta práctica fue usada una laptop de la marca Lenovo, modelo Ideapad S540 con Ryzen 7, 8 GB de RAM y una tarjeta gráfica AMD Radeon, esta fue mi computadora personal pero la practica en general requería de un equipo de cómputo para poder realizarse.

Herramientas de software: Para esta práctica fue empleada la plataforma de Google Colaboratory donde se usó Python 3 como lenguaje para programar los problemas requeridos por la práctica.

Desarrollo de la practica

Con el planteamiento visto en clase que ha sido incluido en la parte del marco teórico para cada uno de los problemas veremos en esta parte como fue su desarrollo para hacer un programa en Python y a la vez, como se reconstruyo su solución, por ejemplo, para saber que ítems están en la mochila al obtener el valor máximo que se puede cargar.

Desarrollo del problema de la mochila

En este problema, la primera parte fue crear el arreglo de dos dimensiones que tomaremos como la tabla hecha en el marco teórico, este arreglo lo rellenamos con ceros usando la función zeros de numpy.

Algo interesante de la resolución de estos problemas fue que no se empleó la recursividad en funciones, si no que se realizaron usando dos ciclos y una serie de condicionales basadas en la serie de pasos vistos previamente, para la construcción de estas analizamos los valores de las columnas aledañas y observamos si se puede conseguir una mejor solución sacando un máximo o un mínimo según sea necesario. El código quedo de la siguiente manera:

```
import numpy as np

v = (6, 10, 12)
w = (1, 2, 3)
W = 5
N = len(v)

m = np.zeros( (N+1, W+1) )

for row in range(1,N+1):
  for col in range(1, W+1):
    if w[row-1] > col:
      m[row][col] = m[row-1][col]
    else:
      m[row][col] = max( m[row-1][col], m[row-1][col-w[row-1]]+v[row-1])

print('Tabla generada de solución Bottom Up:\n', m)
```

Cabe recalcar que esto únicamente nos ayuda a obtener la solución de cuál es el valor más grande que podemos llevar en la mochila, sin embargo, no nos dice que elementos son introducido, para eso tendremos que reconstruir la solución, esto lo haremos por medio de backtracking y tendremos que plantear algunas condiciones que nos permitirán recorrer la tabla hasta encontrar los elementos que componen la solución.

Por ejemplo, recorreremos el arreglo de dos dimensiones m empezando por la ultima casilla donde encontramos nuestra solución y regresando según vayamos encontrando cambios, por ejemplo, tenemos que si la celda de arriba de nuestra celda con la solución final es distinta entonces el elemento tomado en dicha celda fue considerado para la solución final, restamos el peso a una variable que nos ayudara a recorrer la tabla, en este caso llamada weight, y esto hará que nos movamos las filas necesarias para continuar con las comparaciones.

El código para la reconstrucción de esta solución se puede ver de la siguiente manera:

```
items_taken = []
weight = W
res = m[N][W]
```

```
for i in range(N, 0, -1):
  if res <= 0:
    break
  if res == m[i-1][weight]:
    continue
  else:
    items_taken.append(w[i-1])
    res = res - v[i-1]
    weight = weight - w[i-1]

print(f'Lista de los indices de los elementos tomados: {items_taken}')
```

Finalmente observamos el resultado obtenido de ejecutar este programa empleando los mismos datos usados que en el marco teórico y podemos observar que llegamos a la misma tabla y que los elementos coinciden con los que se toman para llegar al valor final.

Figura 2. Resultado del programa para la resolución del problema de la mochila.

```
Tabla generada de solución Bottom Up:
 [[ 0.  0.  0.  0.  0.  0.]
 [ 0.  6.  6.  6.  6.  6.]
 [ 0.  6. 10. 16. 16. 16.]
 [ 0.  6. 10. 16. 18. 22.]]
Lista de los indices de los elementos tomados: [3, 2]
```

Desarrollo del problema del cambio de monedas

En este problema, varios de los pasos son bastante similares a los del problema anterior, si acaso, es un poco más complicada la reconstrucción de la solución pues no solo requiere retornar cuales fueron las denominaciones usadas para la solución, sino que también requiere saber cuántas de las monedas de dicha denominación fueron usadas.

Para ir creando este código, igual que para el primer problema, crearemos una matriz m y la llenaremos de ceros usando una función de numpy. Posteriormente llenaremos en orden ascendente la fila donde la denominación es uno, pues ese viene siendo la cantidad de monedas que se necesitan según el valor, a partir de ahí usaremos dos ciclos, uno anidado en el otro y emplearemos los pasos detallados en el marco teórico, donde si una denominación excede el tamaño simplemente se usara el valor calculado previamente que almacenamos en la celda superior y si, por el contrario, esta denominación si nos ayudara a reducir el número de monedas

porque cabe dentro del valor establecido, verificaremos si es menor su valor con el de la celda superior para saber cuál de los dos guardaremos en la celda.

```
import numpy as np

d = [1, 2, 4]
N = 6

N_den = len(d)

m = np.zeros((N_den, N+1))

for col in range(N+1):
  m[0][col] = col

for row in range(1,N_den):
  for col in range(1, N+1):
    if d[row] > col:
      m[row][col] = m[row-1][col]
    else:
      m[row][col] = min( m[row-1,col], m[row][col-d[row]]+1)
```

Para la reconstrucción de la solución de este problema también se necesita hacer algo parecido a lo que hicimos en el problema de la mochila, pero tomaremos algunas consideraciones, asumimos que, en este problema, se agarraran tantas monedas de mayor denominación como se puedan para llegar a la solución y a partir podemos empezar a idear una solución.

Empezamos un ciclo recorriendo las filas empezando desde la celda donde está el resultado hasta la inicial, de aquí vamos a ir checando si hay un cambio en las monedas y si la denominación que se está considerando no excede la cantidad de dinero que queremos juntar, entonces añadimos el máximo de monedas que se pueden agregar a dicha denominación y así continuamos avanzando hasta que tengamos todas las monedas usadas para la solución, el código final para obtener la cantidad de monedas de cada denominación en la solución fue el siguiente:

```
for i in range(N_den-1, -1, -1):
  if (amount != m[i-1][total_value]) and (d[i] <= total_value):
    coins_per_den = [d[i], total_value//d[i]]
    coins_taken.append(coins_per_den)
    amount = m[i-1][total_value]
    total_value = total_value%d[i]

print(f'Las denominaciones de monedas y su respectiva cantidad de monedas tomadas:
{coins_taken}')
```

Y la tabla obtenida junto con la solución para los datos usados en el marco teórico obtuvimos el siguiente resultado:

Figura 3. Resultado del programa para la resolución del problema del cambio de monedas.

```
Tabla generada de solución Bottom Up:
 [[0. 1. 2. 3. 4. 5. 6.]
 [0. 1. 1. 2. 2. 3. 3.]
 [0. 1. 1. 2. 1. 2. 2.]]
Las denominaciones de monedas y su respectiva cantidad de monedas tomadas: [[4, 1], [2, 1]]
```

Desarrollo del problema del cambio de monedas

A pesar de que este problema pueda parecer bastante diferente que los otros dos anteriores, es más una cuestión de perspectiva, ya que como se pudo ver en el marco teórico, la solución es bastante parecida por el método de “bottom up” y ayuda bastante a reducir la complejidad temporal de lo que sería una solución bruta o “naive”.

De hecho, al ir construyendo el programa podemos notar que se puede hacer siguiendo algunos pasos similares, por ejemplo, para este programa también declararemos un arreglo de dos dimensiones que llenaremos de ceros usando una función de numpy. Posteriormente, en vez de emplear recursividad, usaremos dos ciclos, uno anidado dentro del otro para recorrer toda la tabla y usaremos la serie de criterios que vimos en el marco teórico para establecer las condicionales que ayudaran a calcular el resultado de este problema, si tenemos que la letra en ambas cadenas en la posición actual de la columna y la fila, donde el contador para fila es una posición de la cadena L1 y el contador de las columnas es un contador para la cadena L2, es distinta compararemos la celda superior y la celda anterior y ese será el resultado de la celda actual, evitándonos la necesidad de calcular nuevamente cuantas letras coinciden en dicha subsecuencia, ahora, si por el contrario estas letras son iguales se le aumentara el valor a la celda en diagonal hacia arriba a la derecha y así hasta que encontremos el tamaño de la subsecuencia más grande.

El código para esta solución y que ayudara a dejar más en claro lo anteriormente dicho es el siguiente:

```
import numpy as np

L1 = 'AGGTAB'
L2 = 'GXTXAYB'

x = len(L1)
y = len(L2)

m = np.zeros( (y+1, x+1) )
```

```
for row in range(1, y+1):
  for col in range(1, x+1):
    if L1[col-1] != L2[row-1]:
      m[row][col] = max(m[row-1][col], m[row][col-1])
    else:
      m[row][col] = m[row-1][col-1] + 1
```

En cuanto a la reconstrucción de la solución, en este caso será la subsecuencia en común más larga, para esto estaremos creando una nueva string añadiendo los caracteres que coinciden con la solución, iniciaremos con una variable que contiene el tamaño de la LCS y que iremos excrementando según vayamos agregando los caracteres que pertenecen a la solución, iniciamos con un ciclo que ira recorriendo las filas desde la última hasta la primera, si la variable con el tamaño de la LCS llega a cero entonces hemos reconstruido la solución, al ir pasando por este ciclo analizamos si efectivamente el carácter en el que se encuentra hace que el valor de la LCS sea más grande que la celda superior o la anterior a él, si es el caso, añadimos la letra a la cadena y nos movemos en diagonal hacia arriba a la izquierda, hasta completar la LCS. Este es quizá una de las soluciones que se tuvieron que analizar más, pues inicialmente solo se comparaba con la celda de arriba y si bien, funcionaba con el ejemplo propuesto en el marco teórico, había casos particulares donde ya no retornaba la subsecuencia más larga de manera correcta.

El código quedo de la siguiente manera:

```
while i > 0 and j > 0:

  if L1[j-1] == L2[i-1]:
    LCS = L1[j-1] + LCS
    i -= 1
    j -= 1
    res -= 1
  elif m[i][j-1] > m[i-1][j-1]:
    j -= 1
  else:
    i -= 1
```

El resultado obtenido del programa, incluyendo la subcadena y la tabla en general es el siguiente:

Figura 4. Resultado del programa para la resolución del problema de la subsecuencia común más larga.

```
Tabla generada de la solución Bottom-up:
 [[0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 1. 1. 1. 1.]
 [0. 0. 1. 1. 1. 1. 1.]
 [0. 0. 1. 1. 2. 2. 2.]
 [0. 0. 1. 1. 2. 2. 2.]
 [0. 1. 1. 1. 2. 3. 3.]
 [0. 1. 1. 1. 2. 3. 3.]
 [0. 1. 1. 1. 2. 3. 4.]]
La subsecuencia mas larga de caracteres es de 4.0
La subsecuencia mas larga de caracteres es GTAB
```

Conclusiones y recomendaciones

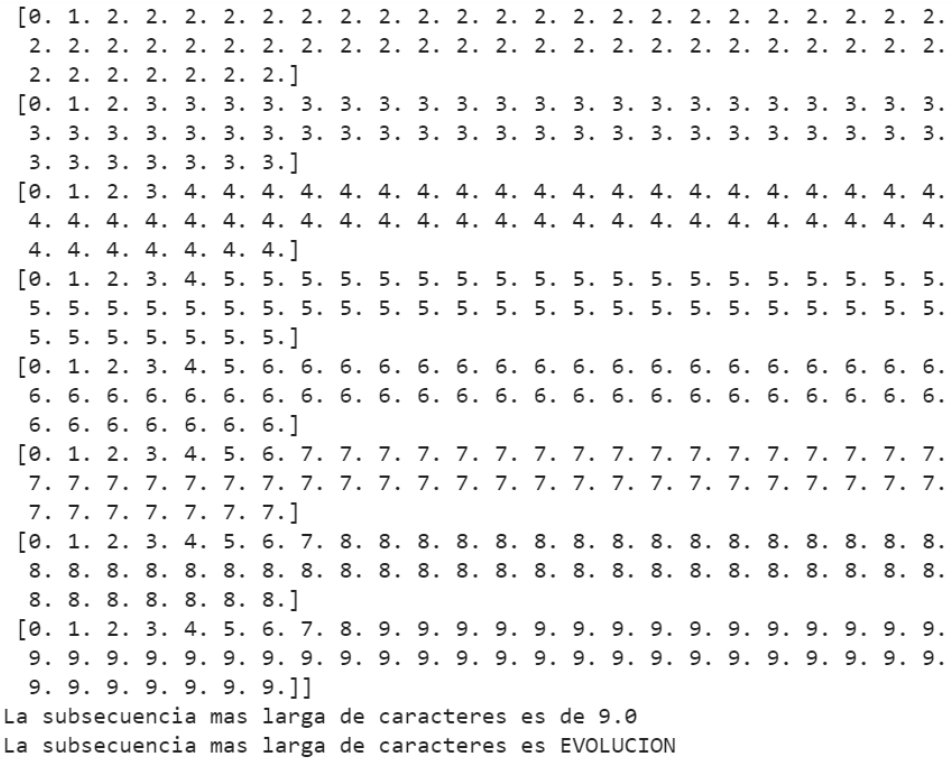
Esta fue una práctica que ayudo a repasar el tema de la programación dinámica que habíamos visto previamente en la materia de análisis de algoritmos, pero que a la vez nos adentró un poco más en lo que creo, serán conceptos importantes para la materia, como lo es el uso de algoritmos óptimos y la importancia de no resolver problemas que previamente ya habíamos resuelto para tener una mejor complejidad temporal, también considero que, personalmente, me ayudo a repasar la teoría de cómo funciona la programación dinámica y distinguir los métodos mediante los cuales se puede llegar a una solución además de que pude leer un poco más sobre sus aplicaciones como en su ocasión fueron comentadas en clase.

A partir de los programas realizados me intereso probar con “Edge Cases”, es decir probar si había un caso muy particular en que el programaría fallaría, entre otros, por ejemplo, probé que pasaría con el programa de CMP si no se le daban las monedas en orden solo por mera curiosidad y fue interesante ver qué clase de resultados arrojaba, efectivamente, la forma de calcular con subproblemas resueltos se volvía más complicado y lo previamente planteado no funcionaba. Con el problema de la subsecuencia más larga probé que pasaría si la primera cadena contenía a la primera y en otro caso, que pasaría si esta se repetía varias veces, las siguientes imágenes son de estos casos de prueba que me parecieron interesantes.

Figura 5. Cadenas para el caso de prueba en el programa de LCS donde una se repite varias veces en la otra.

```
L1 = 'EVOLUCIONEVOLUCIONEVOLUCIONEVOLUCIONEVOLUCIONEVOLUCION'
L2 = 'EVOLUCION'
```

Figura 6. Resultado de la prueba del caso para el programa de LCS donde una cadena se repite en la otra varias veces.



```
[0. 1. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2.]
[0. 1. 2. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3.
 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3.
 3. 3. 3. 3. 3. 3. 3.]
[0. 1. 2. 3. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4.
 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4.
 4. 4. 4. 4. 4. 4. 4.]
[0. 1. 2. 3. 4. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5.
 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5.
 5. 5. 5. 5. 5. 5. 5.]
[0. 1. 2. 3. 4. 5. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6.
 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6. 6.
 6. 6. 6. 6. 6. 6. 6.]
[0. 1. 2. 3. 4. 5. 6. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7.
 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7. 7.
 7. 7. 7. 7. 7. 7. 7.]
[0. 1. 2. 3. 4. 5. 6. 7. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8.
 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8. 8.
 8. 8. 8. 8. 8. 8. 8.]
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9.
 9. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9. 9.
 9. 9. 9. 9. 9. 9. 9.]]
La subsecuencia mas larga de caracteres es de 9.0
La subsecuencia mas larga de caracteres es EVOLUCION
```

Figura 7. Cadenas para el caso de prueba en el programa de LCS donde una se encuentra contenida dentro de la otra.

```
L1 = 'EAVAOALAUACAIAOANA'
L2 = 'EVOLUCION'
```

Figura 8. Resultado de la prueba del caso para el programa de LCS donde una cadena se encuentra contenida dentro de la otra.

```
Tabla generada de la solución Bottom-up:
 [[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [0. 1. 1. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]
 [0. 1. 1. 2. 2. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3. 3.]
 [0. 1. 1. 2. 2. 3. 3. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4.]
 [0. 1. 1. 2. 2. 3. 3. 4. 4. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 5.]
 [0. 1. 1. 2. 2. 3. 3. 4. 4. 5. 5. 6. 6. 6. 6. 6. 6. 6. 6. 6.]
 [0. 1. 1. 2. 2. 3. 3. 4. 4. 5. 5. 6. 6. 7. 7. 7. 7. 7. 7. 7.]
 [0. 1. 1. 2. 2. 3. 3. 4. 4. 5. 5. 6. 6. 7. 7. 8. 8. 8. 8. 8.]
 [0. 1. 1. 2. 2. 3. 3. 4. 4. 5. 5. 6. 6. 7. 7. 8. 8. 9. 9. 9.]]
La subsecuencia mas larga de caracteres es de 9.0
La subsecuencia mas larga de caracteres es EVOLUCION
```

La respuesta fue efectivamente una matriz más grande de lo que cabe en pantalla en uno de los casos, pero el resultado obtenido es correcto, pensándolo más a fondo una mejor forma de saber si una cadena es una subsecuencia de otra es usar un stack o contadores, pero aún así es interesante ver cómo podemos resolver otra clase de problemas con esta misma solución incluso si no es de la manera más óptima. En cuanto al problema de la mochila, de igual manera se realizaron algunos casos de prueba como los planteados en clase de que sucedía cuando estaban en desorden los elementos, que, en este caso, si bien la tabla cambia, si se llega al mismo resultado.

La experimentación a lo largo de la realización de esta práctica me ayudo a pensar más como un tester que como un programador, hubo casos donde podía entender efectivamente que fallaba y podía entender mejor como estaba diseñado el algoritmo, también me ayudo a analizar en qué casos este algoritmo era muy efectivo y en qué casos tal vez conviene usar uno diferente.

Bibliografía

Flores, I. (2021, 23 de septiembre). *Programación dinámica*. UNAM Facultad de Ingeniería. https://www.ingenieria.unam.mx/sistemas/PDF/Avisos/Seminarios/SeminarioV/Sesion6_IdaliaFlores_20abr15.pdf

GeeksForGeeks. (2021, 22 de septiembre). *Tabulation vs Memoization - GeeksforGeeks*. GeeksforGeeks. https://www.geeksforgeeks.org/tabulation-vs-memoization/

CodesDope. (2021, 23 de septiembre). *Dynamic Programming | Top-Down and Bottom-Up approach | Tabulation V/S Memoization*. https://www.codesdope.com/course/algorithms-dynamic-programming/

Rosas, J (2021, 20 de septiembre). Programación dinámica. Clase de Evolutionary Computing.

GeeksForGeeks. (2021a, 26 de agosto). *Longest Common Subsequence | DP-4 - GeeksforGeeks*. GeeksforGeeks. https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/