Algoritmia y Complejidad

Ejercicios tema 4

Laboratorio Miércoles 15:00-17:00

Integrantes:

Carlos Javier Hellín Asensio
Diego Gutiérrez Marcos

Curso: 2020-2021

Problema 3

Descripción del algoritmo:
Para este problema el algoritmo genera una matriz M cuyas filas representan los valores de cada moneda (por ejemplo: 1, 3 y 5) y las columnas son las posibles cantidades a pagar desde 0 hasta D, siendo D la cantidad de dinero a pagar.
Para ello se rellena la matriz teniendo en cuenta estos posibles casos:

- Si D < v[i], es decir, si la cantidad a pagar es menor que el valor de la moneda actual entonces el número de monedas a devolver es lo que hay que devolver con la anterior moneda, esto es M[i - 1][j] en la matriz generada M.
- Si D >= v[i], es decir, si la cantidad a pagar es mayor o igual que el valor de la moneda actual entonces:
    - Si hay monedas suficientes como para pagar (el valor de la moneda actual por la cantidad de esa moneda es mayor o igual que la cantidad a pagar): el mínimo sería entre lo que hay que devolver con la anterior moneda y la fila actual restando a la columna el valor de la moneda actual. A esto último se le suma uno. Esto es el min(M[i - 1][j], M[i][j - v[i]] + 1) en la matriz generada M.
    - Si no hay monedas suficientes entonces el mínimo sería entre lo que hay que devolver con la anterior moneda y la fila anterior restando a la columna el valor de la moneda actual por la cantidad de monedas de ese valor. A esto último se le suma la cantidad que hay de ese valor. Esto es el min(M[i - 1][j], M[i - 1][j - (v[i] * c[i])] + c[i]) en la matriz generada M.

Una vez con la matriz generada se obtiene el número de monedas a devolver y si se puede devolver con las monedas que se disponen, se muestra cuántos billetes de cada tipo forman la descomposición óptima. Para obtener el número de monedas o billetes necesarios se realiza una búsqueda en la matriz generada en la posición M [N-1][D-1] siendo la N los posibles valores de cada moneda y la D la cantidad de dinero a pagar. Si esa posición visitada resulta que es un math.inf no se puede devolver esa cantidad de dinero con esas monedas mientras que si ese valor es cualquiera valido el numero de monedas utilizadas para pagar se obtiene buscando en la posición M[N][D]. Para saber cuántas monedas hay de cada valor se recorre una lista, cuya longitud es la cantidad diferentes monedas que hay, desde el final . Si queda cantidad suficiente de esa moneda y si al sumarle al total no ha superado la

cantidad a pagar y sigue habiendo monedas totales disponibles se incluye esa moneda como resultado. Por ejemplo si tienes una cantidad a pagar de 11 y solo tienes 15 monedas de 1, 5 de 3 y 5 de 5. La cantidad total de monedas a utilizar era de solamente 3 se coge la moneda 5 y como es se cumplen las condiciones que he dicho 2 veces significa que hay 2 monedas de 5. Tras ello se realiza una comprobación con el 3 pero se ve que no se cumplen todas las condiciones por lo que no hay ninguna. Por tanto se mira la moneda restante y se ve que las cumple una vez llegando así al máximo de monedas a utilizar y por tanto las monedas utilizadas para pagar 11 con la menor cantidad de monedas son 2 de 5 y 1 de 1

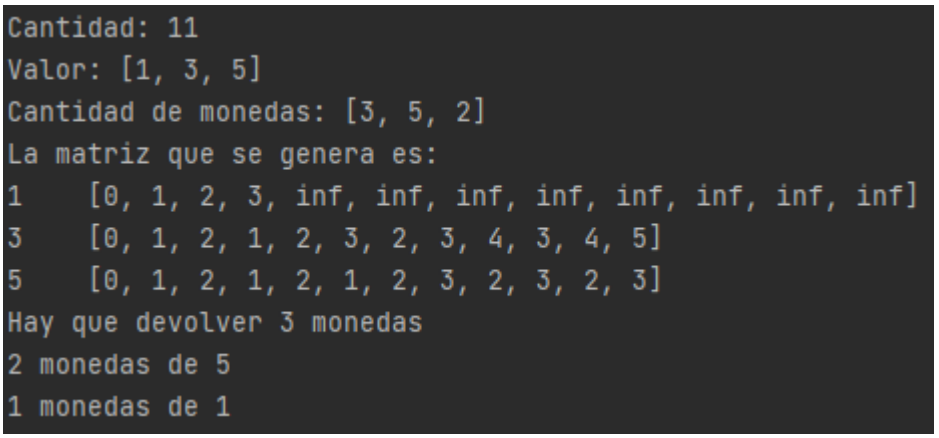
Casos de prueba:
Esta imagen de abajo es la realización de nuestro primer caso de prueba, el cual hemos realizado a partir del primer caso del anexo.

```
Cantidad: 11
Valor: [1, 3, 5]
Cantidad de monedas: [15, 5, 5]
La matriz que se genera es:
1    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
3    [0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5]
5    [0, 1, 2, 1, 2, 1, 2, 3, 2, 3, 2, 3]
Hay que devolver 3 monedas
2 monedas de 5
1 monedas de 1
```

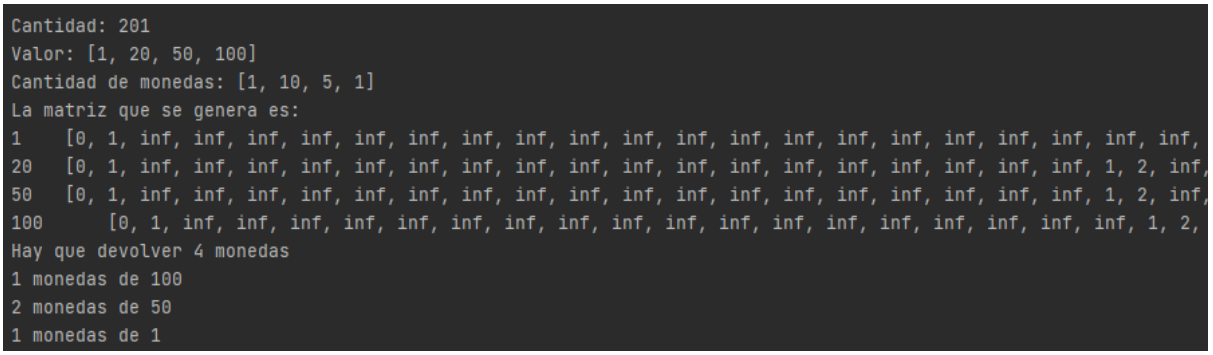
Funcionamiento del algoritmo: Este algoritmo empieza pintando por pantalla la cantidad de monedas que hay, los valores de estas además del número correspondiente de las monedas que hay por cada valor. Tras ello el algoritmo crea una matriz que inicialmente se encuentra a 0 siendo los valores de las monedas las filas y las posibles cantidades a pagar las columnas. Esta matriz se va rellenando teniendo en cuenta los posibles casos. Cuando la cantidad a pagar es menor que el valor de la moneda actual entonces el número de monedas a devolver es lo que hay que devolver con la anterior moneda y si la cantidad a pagar es mayor o igual que el valor de la moneda actual entonces se pueden dar dos posibilidades: la primera es que hay suficientes monedas para pagar y habría que realizar el mínimo entre lo que hay que devolver con la anterior moneda y la fila actual restando a la columna el valor de la moneda actual y a esto último sumandole uno. Y el otro caso caso sería no hubiese suficientes monedas para pagar de modo que habría que hacer el mínimo entre lo que hay que devolver con la anterior moneda y la fila anterior restando a la columna el valor de la moneda actual por la cantidad de monedas de ese valor. A esto último se le suma la cantidad de monedas que

hay de ese valor. En este caso se llega a la matriz mostrada arriba. Tras ello el algoritmo obtiene el número de monedas que se tiene que devolver para realizar el pago, en este caso siendo 3 y además el algoritmo muestra el número de monedas que hay que utilizar al haber realizado la descomposición óptima si es posible. En nuestro caso ha sido posible y se ha descompuesto la forma de pago en 2 monedas de 5 y 1 de 1 para conseguir el pago

Esta imagen de abajo es la realización de nuestro segundo caso de prueba, el cual hemos realizado a partir del segundo caso del anexo



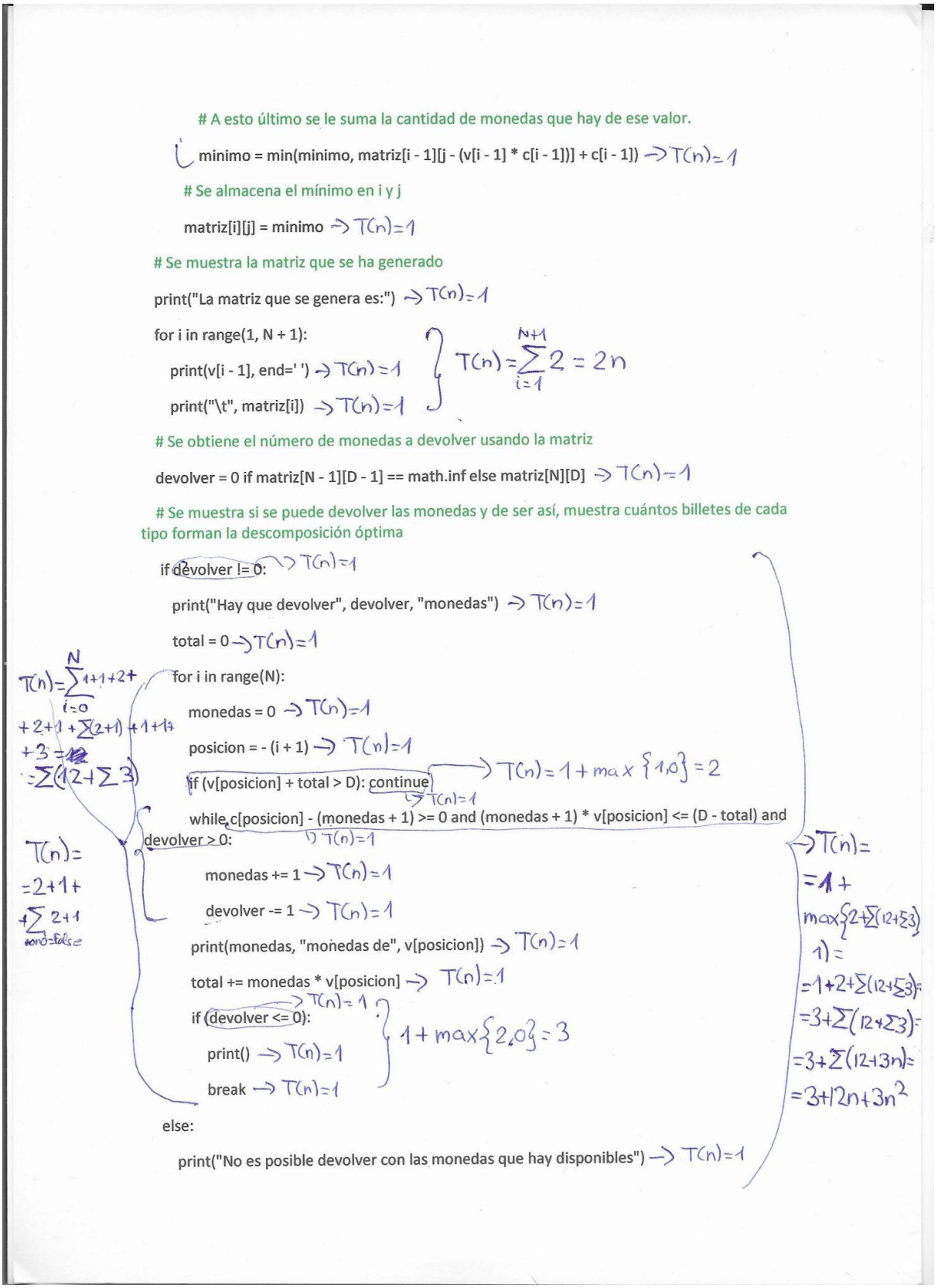
Esta imagen de abajo es la realización de nuestro tercer caso de prueba, el cual hemos realizado a partir del tercer caso del anexo



Esta imagen de abajo es la realización de nuestro cuarto caso de prueba, en el cual hemos incluido partes del enunciado

```
Cantidad: 105
Valor: [1, 2, 5, 10, 20, 50, 100]
Cantidad de monedas: [1, 2, 6, 4, 2, 5, 1]
La matriz que se genera es:
1    [0, 1, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, i
2    [0, 1, 1, 2, 2, 3, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf,
5    [0, 1, 1, 2, 2, 1, 2, 2, 3, 3, 2, 3, 3, 4, 4, 3, 4, 4, 5, 5, 4, 5, 5, 6, 6, 5, 6, 6, 7, 7, 6, 7, 7, 8, 8, 9, inf,
10   [0, 1, 1, 2, 2, 1, 2, 2, 3, 3, 1, 2, 2, 3, 3, 2, 3, 3, 4, 4, 2, 3, 3, 4, 4, 3, 4, 4, 5, 5, 3, 4, 4, 5, 5, 4, 5, 5
20   [0, 1, 1, 2, 2, 1, 2, 2, 3, 3, 1, 2, 2, 3, 3, 2, 3, 3, 4, 4, 1, 2, 2, 3, 3, 2, 3, 3, 4, 4, 2, 3, 3, 4, 4, 3, 4, 4
50   [0, 1, 1, 2, 2, 1, 2, 2, 3, 3, 1, 2, 2, 3, 3, 2, 3, 3, 4, 4, 1, 2, 2, 3, 3, 2, 3, 3, 4, 4, 2, 3, 3, 4, 4, 3, 4, 4
100      [0, 1, 1, 2, 2, 1, 2, 2, 3, 3, 1, 2, 2, 3, 3, 2, 3, 3, 4, 4, 1, 2, 2, 3, 3, 2, 3, 3, 4, 4, 2, 3, 3, 4, 4, 3,
Hay que devolver 2 monedas
1 monedas de 100
1 monedas de 5
```

Cálculo de la complejidad:



Por tanto el T(n) del algoritmo que hemos implementado es 10+ 5*n^2 + 18*n. Siendo su complejidad O(n^2)

Problema 7

Descripción del algoritmo:
Para calcular la longitud máxima posible se ha utilizado programación dinámica. Se crea una matriz M cuyas filas es la longitud de la secuencia A y las columnas de las de B. Primeramente está matriz se inicia a 0, pero la primera fila y la primera columna se calculan mirando si el bit más significativo (MSB) de A está en B al menos una vez y viceversa (el MSB de B con A) dando el valor 1 a la matriz en dicha fila y columna, es decir, en M[i][j].
Luego se recorre la matriz con dos bucles sin pasar por la primera fila y primera columna que ya han sido rellenados anteriormente. En este caso lo que se busca es si la longitud no es la máxima posible y el bit de B[j] está en A[i] siendo "j" la columna a recorrer e "i" la fila a recorrer, entonces el tamaño máximo de M[i][j] siendo M la matriz será el valor de la columna anterior más uno. En caso contrario, el tamaño máximo simplemente será el valor de la columna anterior.
Una vez la matriz esté completa, se obtiene la longitud máxima posible que se sitúa en la última fila y última columna de la matriz.

Sabiendo la longitud máxima posible, se recorre las X posibles secuencias comunes, buscando siempre que sea de la misma longitud que la longitud máxima y de no ser así, rellenar con ceros por la izquierda.
Se comprueba que si la X posible secuencia es subsecuencia de A y subsecuencia de B, entonces se denomina a X una subsecuencia común de A y B y se añade como resultado. Mostrando al final del algoritmo todas las secuencias comunes de A y B de dicha longitud máxima.

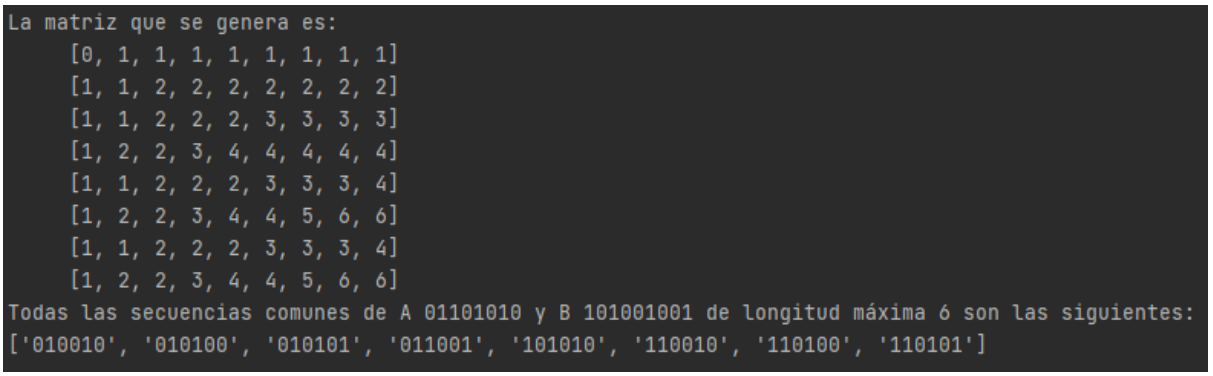
Casos de prueba:
Esta imagen de abajo es la realización de nuestro primer caso de prueba, el cual hemos realizado a partir del primer caso del anexo

```
La matriz que se genera es:
     [0, 1, 1, 1]
     [1, 1, 2, 2]
     [1, 1, 2, 2]
Todas las secuencias comunes de A 011 y B 1010 de longitud máxima 2 son las siguientes:
['01', '11']
```

Funcionamiento del algoritmo: Para poder obtener la secuencia común necesitamos en una primera instancia necesitamos obtener la longitud máxima posible para ello realizamos la creación de una matriz que inicialmente se encuentra a 0 la cual primero rellenamos la primera fila y

columna. En ellas ponemos un valor 1 en la posición correspondiente cuando se encuentra en la primera fila y el bit más significativo de A está en B al menos una vez y también cuando está en la primera columna y el bit más significativo de B está en A al menos una vez. Tras ello el algoritmo rellena el resto de la matriz de forma que si la longitud no es la más alta posible y el bit de B[j] está en A[i] el tamaño máximo será el valor de la columna anterior más uno sino el tamaño máximo será el valor de la columna anterior. Tras la generación de la matriz también se obtiene el valor máximo de la longitud que se encuentra situado en la última columna y última fila. En este caso el valor de longitud máxima es 2. Tras saber esta longitud el algoritmo realiza un recorrido de la x posibles secuencias buscando que sea en este caso de longitud 2. Tras realizar la comprobación de que esa secuencia x es subsecuencia de A y de B, pasa a ser denominada subsecuencia común de A y B y se la añade como resultado.Tras realizar todas las comprobaciones de las diferentes secuencias finalmente se muestran todas las que son comunes. En este caso de prueba únicamente hay dos secuencias comunes que son “01” y “11”

Esta imagen de abajo es la realización de nuestro segundo caso de prueba, el cual hemos realizado a partir del segundo caso del anexo

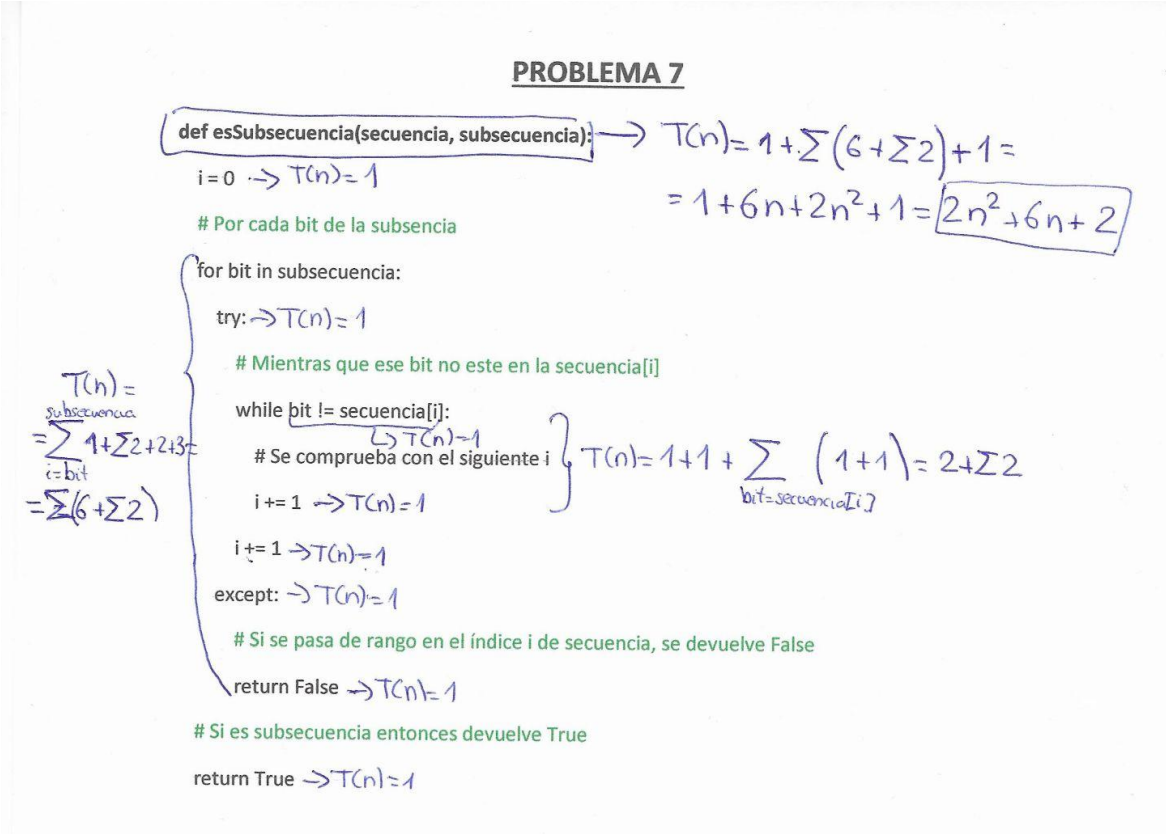


El siguiente caso de prueba se hace para cuando A y B son del mismo tamaño:

Y por último, otro caso de prueba que obtiene una longitud máxima de 4:

```
La matriz que se genera es:
    [0, 1, 1, 1, 1, 1, 1, 1, 1]
    [0, 1, 1, 2, 2, 2, 2, 2, 2]
    [1, 1, 2, 2, 2, 3, 3, 3, 3]
    [1, 2, 2, 3, 4, 4, 4, 4, 4]
    [1, 2, 2, 3, 4, 4, 5, 5, 5]
    [1, 1, 2, 2, 2, 3, 3, 4, 4]
    [1, 2, 2, 3, 4, 4, 5, 5, 6]
    [1, 1, 2, 2, 2, 3, 3, 4, 4]
    [1, 2, 2, 3, 4, 4, 5, 5, 6]
    [1, 1, 2, 2, 2, 3, 3, 4, 4]
Todas las secuencias comunes de A 1101101010 y B 010110101 de longitud máxima 4 son las siguientes:
['0000', '0001', '0010', '0011', '0100', '0101', '0110', '0111', '1000', '1001', '1010', '1011', '1100', '1101', '1110', '1111']
```

Cálculo de la complejidad:



```
def obtenerLongituxMax(A, B):
    N_A = len(A)
    N_B = len(B)
    matriz = [[0 for i in range(N_B)] for j in range(N_A)]  # Se crea una matriz inicialmente a 0.
    for i in range(N_A):  # Se rellena la primera fila y primera columna
        for j in range(N_B):
            # Para la primera fila y si el bit más significativo (MSB) de A está en B al menos una vez
            if i == 0 and A[0] in B[:j + 1]:
                matriz[i][j] = 1  # Se le pone un 1 a la matriz
            # Para la primera columna y si el bit más significativo (MSB) de B está en A al menos una vez
            elif j == 0 and B[0] in A[:i + 1]:
                matriz[i][j] = 1  # Se le pone un 1 a la matriz
    # Se empieza a rellenar la matriz sin necesidad de hacerlo para la primera fila y primera columna
    for i in range(1, N_A):
        for j in range(1, N_B):
            # Si la longitud no es la máxima posible y el bit de B[j] está en A[i]
            if matriz[i][j - 1] != i + 1 and B[j] == A[i]:
                # El tamaño máximo será el valor de la columna anterior más uno
                matriz[i][j] = matriz[i][j - 1] + 1
            # Sino el tamaño máximo será el valor de la columna anterior
            else:
                matriz[i][j] = matriz[i][j - 1]
    # Se muestra la matriz que se ha generado
    print("La matriz que se genera es:")
    for i in range(N_A):
        print("\t", matriz[i])
    # Se devuelve la longitud máxima posible que se situa en la última fila y última columna de la matriz
    return matriz[N_A - 1][N_B - 1]
```

$T(n) = 2+1+\Sigma\Sigma 1+\Sigma\Sigma 3+\Sigma\Sigma 2+1+\Sigma 1+1 = 5+n^2+3n^2+2n^2+n = \boxed{5+n+6n^2}$

$T(n) = 1+\Sigma\Sigma 1$

$T(n) = \Sigma\Sigma 3$

$T(n) = \Sigma 3$

$T(n) = 1+\max\{1,2\} = 1+2 = 3$

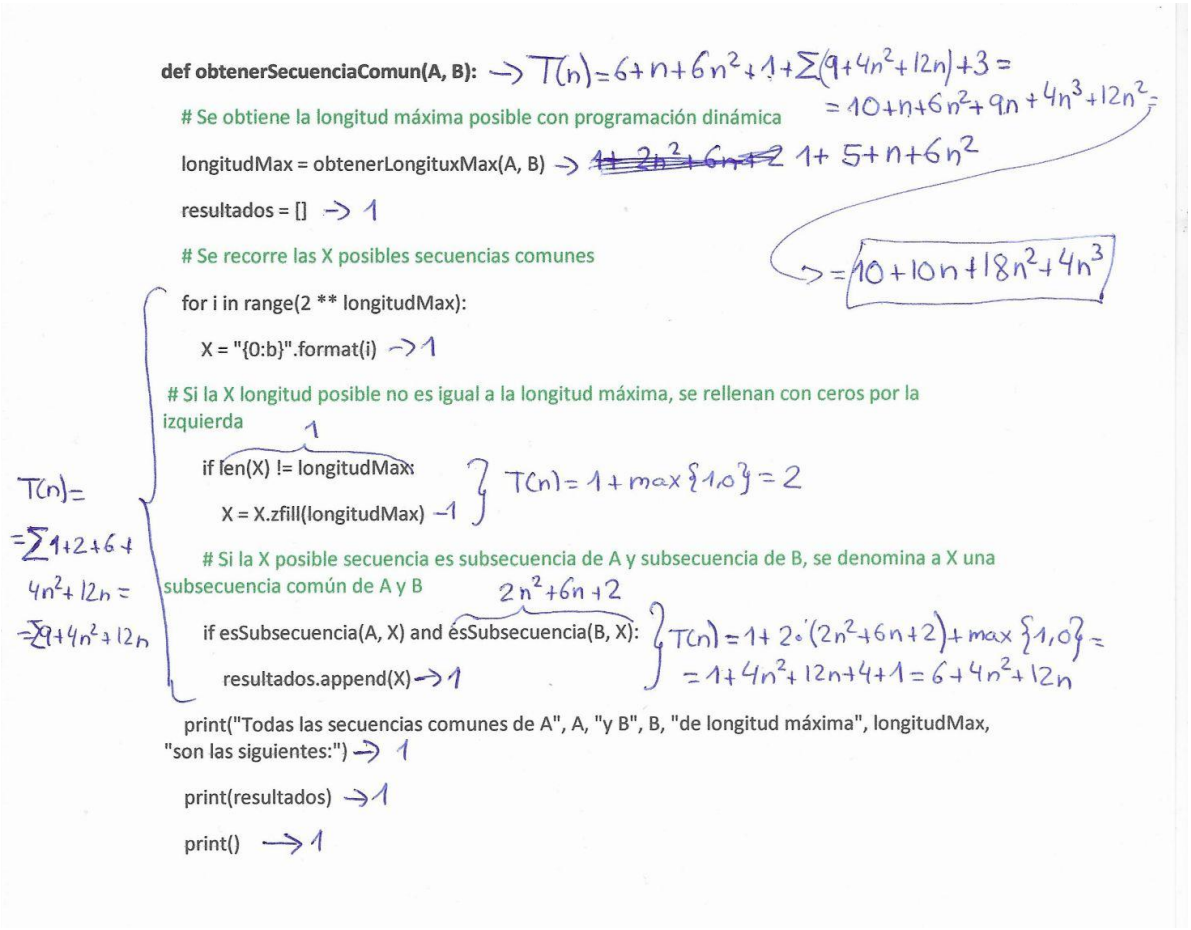
$T(n) = 1+\max\{1,0\} = 2$

$T(n) = \Sigma\Sigma 2$

$T(n) = \Sigma 2$

$T(n) = 1+\max\{1,1\} = 1+1 = 2$

$T(n) = \Sigma 1$



La T(n) del algoritmo final en la que se llaman a dos algoritmos implementados con anterioridad(el primero tiene un T(n) de 2*n^2+6*n+2 y el segundo de 5+n+6*n^2) es de 10+10*n+18*n^2+4*n^3. Por tanto, la complejidad con la que acaba el algoritmo es de O(n^3)