INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

ANÁLISIS DE ALGORITMOS

PROFESOR: CRISTHIAN ALEJANDRO ÁVILA SÁNCHEZ

PRÁCTICA:

ESTRATEGIA DYNAMIC PROGRAMMING

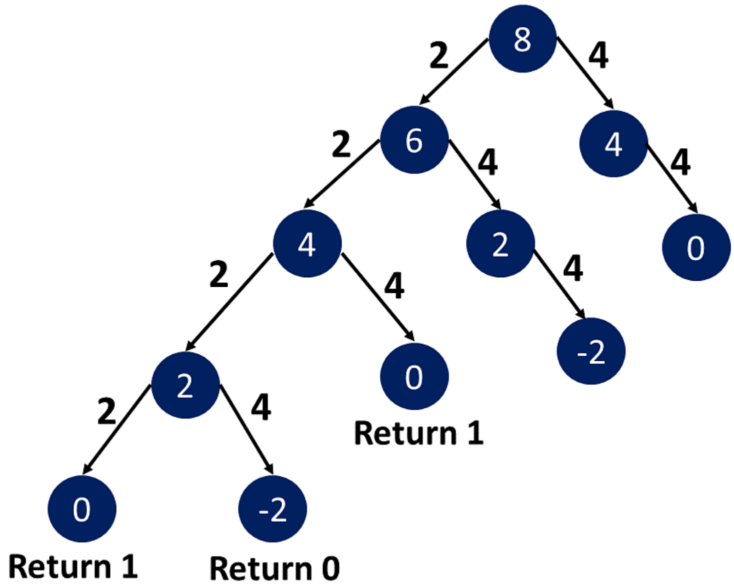
ALUMNO: GARAYOA FLORES ROBERTO ALESSANDRO

GRUPO: 3CM11

1. Introducción

Coin Change Algorithm

Se le da una serie de monedas con distintas denominaciones y una suma entera que representa la cantidad total de dinero; debe devolver la menor cantidad de monedas necesarias para completar esa suma; si esa suma no se puede construir, devuelve -1.



Se le da una secuencia de monedas de varias denominaciones como parte del problema de cambio de moneda. Por ejemplo, considere la siguiente matriz como una colección de monedas, en la que cada elemento representa una denominación diferente.

{ 2, 3, 5, 10, 20, 30, 50 };

Nuestro objetivo es usar estas monedas para acumular una cierta cantidad de dinero usando la menor cantidad (u óptima) de monedas. Además, puede asumir que una denominación dada tiene un número infinito de monedas. Dicho de otro modo, puedes usar una denominación específica tantas veces como quieras.

Por ejemplo, si desea llegar a 78 utilizando las denominaciones anteriores, necesitará las cuatro monedas que se enumeran a continuación.

{3, 5, 20, 50};

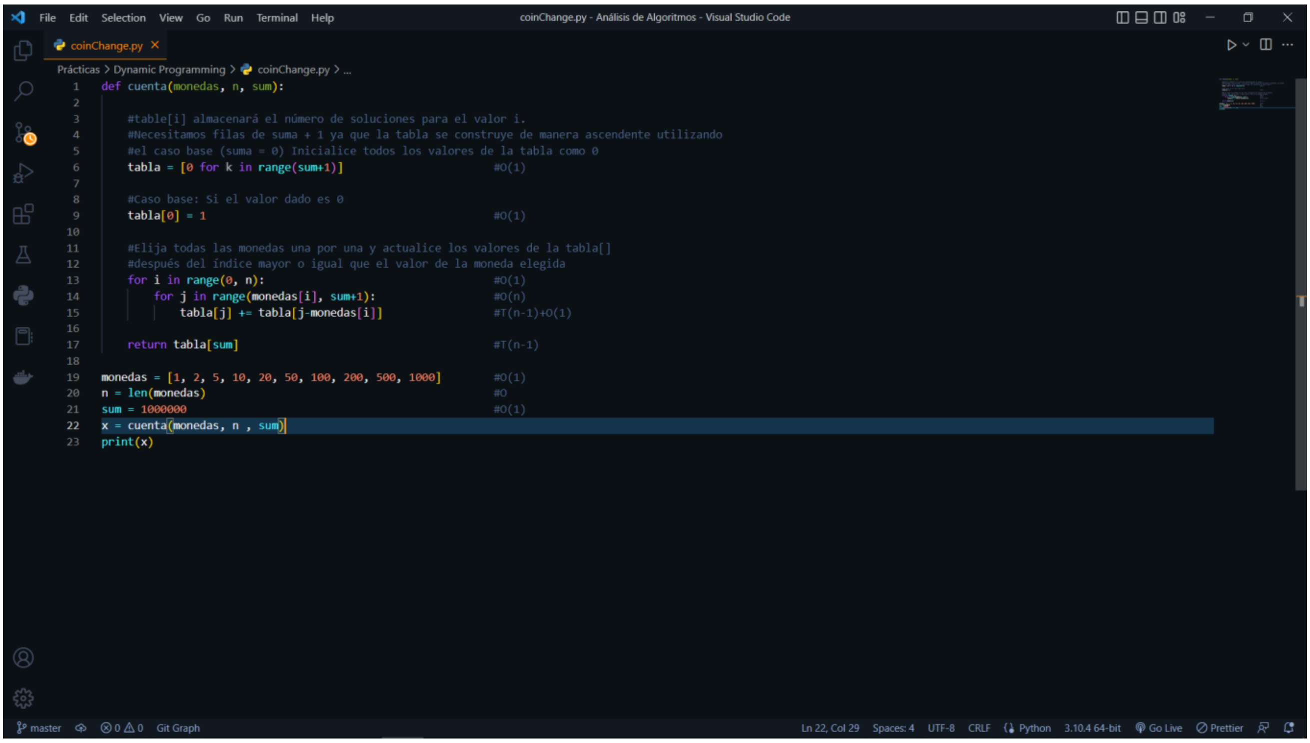
2. Planteamiento del problema

Dada una matriz entera de monedas[ ] de tamaño N que representan diferentes tipos de monedas y una suma entera, la tarea es encontrar el número de formas de hacer una suma usando diferentes combinaciones de monedas[ ].

Tenemos 2 opciones para una moneda de una denominación particular, ya sea i) para incluir, o ii) para excluir.

- ✓ *Si estamos en monedas[n-1], podemos tomar tantas instancias de esa moneda (inclusión ilimitada), es decir, contar(monedas, n, suma – monedas[n-1]) ; luego pasamos a monedas [n-2].*
- ✓ *Después de movernos a monedas[n-2], no podemos retroceder y no podemos elegir monedas[n-1], es decir, contar(monedas, n-1, suma) .*
- ✓ *Finalmente, como tenemos que encontrar el número total de formas, sumaremos estas 2 opciones posibles, es decir, contar (monedas, n, suma – monedas [n-1]) + contar (monedas, n-1, suma);*

3. Implementación



```python
def cuenta(monedas, n, sum):

    #table[i] almacenará el número de soluciones para el valor i.
    #Necesitamos filas de suma + 1 ya que la tabla se construye de manera ascendente utilizando
    #el caso base (suma = 0) Inicialice todos los valores de la tabla como 0
    tabla = [0 for k in range(sum+1)]                    #O(1)

    #Caso base: Si el valor dado es 0
    tabla[0] = 1                                         #O(1)

    #Elija todas las monedas una por una y actualice los valores de la tabla[]
    #después del índice mayor o igual que el valor de la moneda elegida
    for i in range(0, n):                                #O(1)
        for j in range(monedas[i], sum+1):               #O(n)
            tabla[j] += tabla[j-monedas[i]]              #T(n-1)+O(1)

    return tabla[sum]                                    #T(n-1)

monedas = [1, 2, 5, 10, 20, 50, 100, 200, 500, 1000]     #O(1)
n = len(monedas)                                         #O
sum = 1000000                                            #O(1)
x = cuenta(monedas, n , sum)
print(x)
```

```python
def cuenta(monedas, n, sum):

    #table[i] almacenará el número de soluciones para el valor i.
    #Necesitamos filas de suma + 1 ya que la tabla se construye de manera ascendente
utilizando
    #el caso base (suma = 0) Inicialice todos los valores de la tabla como 0
    tabla = [0 for k in range(sum+1)]                       #O(1)

    #Caso base: Si el valor dado es 0
    tabla[0] = 1                                            #O(1)

    #Elija todas las monedas una por una y actualice los valores de la tabla[]
    #después del índice mayor o igual que el valor de la moneda elegida
    for i in range(0, n):                                   #O(1)
        for j in range(monedas[i], sum+1):                  #O(n)
            tabla[j] += tabla[j-monedas[i]]                 #T(n-1)+O(1)

    return tabla[sum]                                       #T(n-1)

monedas = [1, 2, 5, 10, 20, 50, 100, 200, 500, 1000]        #O(1)
n = len(monedas)                                            #O
sum = 1000000                                               #O(1)
x = cuenta(monedas, n , sum)
print(x)
```

4. Análisis de Complejidad

Comenzamos con el análisis de la complejidad con la primera línea del código:

```python
tabla = [0 for k in range(sum+1)]                       #O(1)
```

La función *“in range”* tiene una complejidad inicial de O(1); Toda la línea de código se dedica a crear una tabla, por lo que de igual forma podemos considerar que su complejidad es:

$$O(1)$$

Pasando al caso base de “si el valor dado” a la matriz es 0, simplemente se queda en una complejidad constante.

```python
tabla[0] = 1                                            #O(1)
```

Su complejidad:

$$O(1)$$

En la parte de mayor complejidad tendremos lo siguiente.

```
for i in range(0, n):                                   #O(n)
        for j in range(monedas[i], sum+1):                  #O(n)
            tabla[j] += tabla[j-monedas[i]]                 #O(1)
```

Como se observa, tenemos 2 ciclos *for* anidados, al repetirse N veces su complejidad se vuelve O(n), pasando al siguiente *for* tenemos que su complejidad sigue siento O(n). Al entrar a la parte de lo que se va a hacer con esos 2 ciclos tenemos que al ser una simple operación aritmética, tomará una complejidad de O(1).

Al final tenemos O(n)*O(n)*O(1) = O(n^2)+O(1); Por lo tanto nuestra complejidad de esta parte de código es de:

$$O(n^2)$$

Para el último caso, estando dentro de nuestro main, tenemos

```
monedas = [1, 2, 5, 10, 20, 50, 100, 200, 500, 1000]       #O(1)
n = len(monedas)                                            #O(1)
sum = 1000000                                               #O(1)
x = cuenta(monedas, n , sum)
```

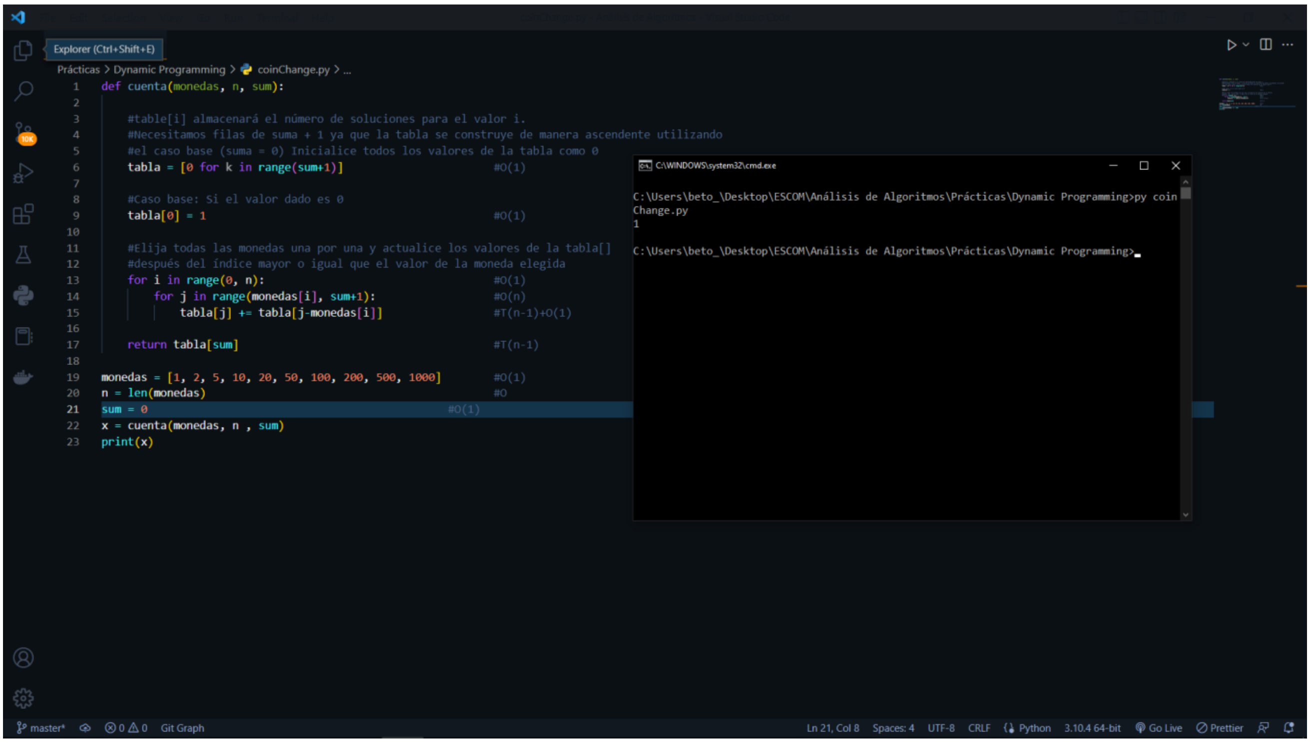
Cada una de las funciones que se representan ahí serán constantes, o sea, que van a repetirse por lo menos una única vez en todo el código.

Para concluir con la sumatoria de las complejidades de este algoritmo podemos decir que el código es un algoritmo de complejidad

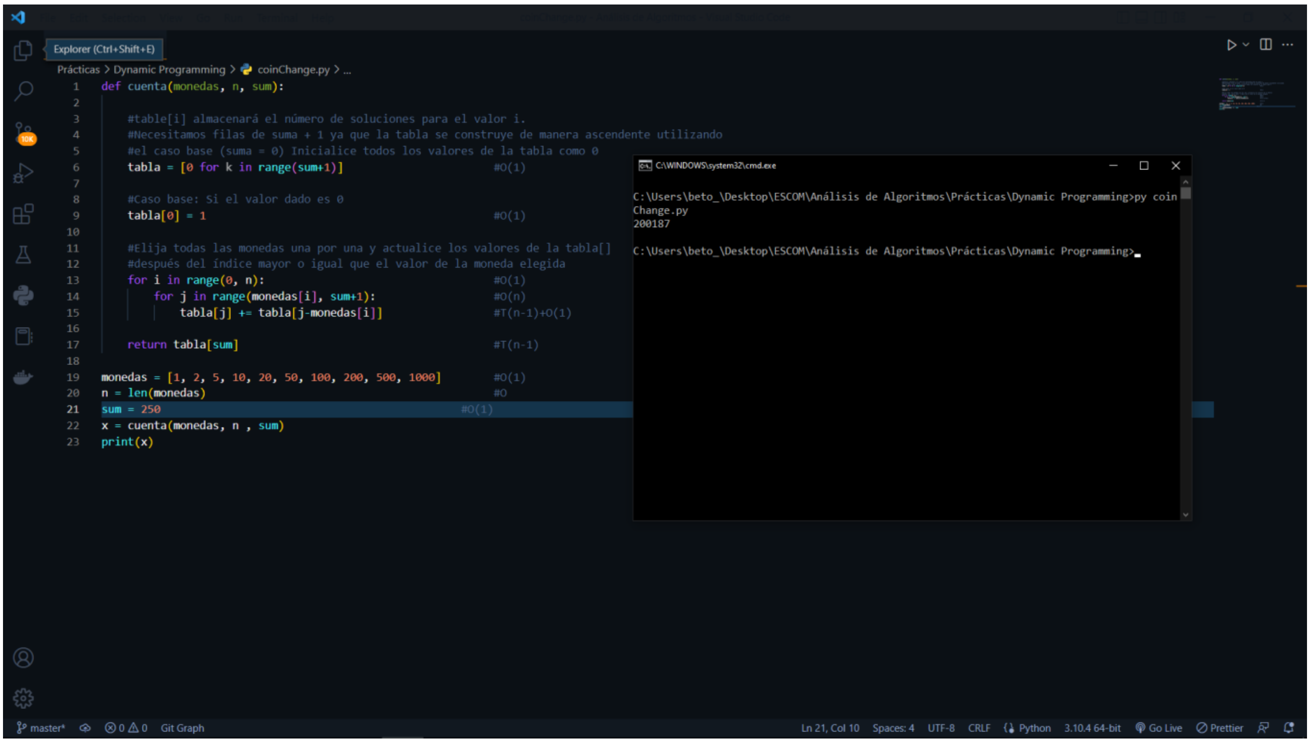
$$n^2$$

5. Pruebas

1er prueba: Tenemos como primer prueba que "me regresen" $0 de cambio, y como veíamos en el caso base, si agrega un 0 a mi tabla, directamente retorna un 1.



2da prueba: Tenemos que dar cambio de 250 pesos, por lo que habrá 200,187 formas de dar ese cambio



6. Conclusiones

Puede que en algún momento hayamos tenido problemas para dar cambio a alguien, que te quedas sin cambio o que no hay cambio, o que tienes mucho cambio pero no sabes qué monedas darle al cliente. Es por eso que se puede considerar un problema NP-completo. El algoritmo te dice cuántas formas puedes completar la tarea de dar cambio.

7. Bibliografías

- ✓ (Dasgupta, July 18, 2006, p. 318)
- ✓ https://www.simplilearn.com/tutorials/data-structure-tutorial/coin-change-problem-with-dynamic-programming
- ✓ https://www.codesdope.com/course/algorithms-coin-change/