Proyecto Final
Diseño y Análisis de Algoritmos

Luis Alejandro Rubiano 202013482
Isai Daniel Chacón 201912015
Kevin Gámez 201912514

I. Explicación del algoritmo

El algoritmo consiste en utilizar programación dinámica e ir llenando una matriz bidimensional (*dp*), dada por la ecuación de recurrencia:

$$dp(i,j) = \begin{cases} 1 & i = 0, j = 0 \\ 0 & i = 0, 0 < j \leq m \\ dp(i-1, j-l_i) + dp(i, j-l_i) & i \neq 0, \sum_{a=0}^{i} l_a \leq j < m+1 \\ 0 & \text{de lo contrario} \end{cases}$$

Cada entrada representa el número de formas de llegar al punto (iniciando en 0) j con saltos de un tamaño múltiplo de l_i (en donde $l_0 = 0$), de esta forma la respuesta buscada sería la suma de las entradas en la última columna. l_1 sería igual a k, l_2 igual a $k+1$ y así sucesivamente.

Para el caso base existe una única forma de llegar al punto 0 tomando saltos de longitud múltiplo de 0, de igual forma no existe manera de llegar a los demás puntos.

Posteriormente, el tercer caso de la ecuación de recurrencia representa todos lo puntos a los que se puede llegar dado un l_i, teniendo en cuenta los puntos posibles desde los que se puede partir, que están acotados entre la máxima distancia en pasos anteriores y $m+1$.

La implementación se desarrolló de la siguiente manera:

1. Se creó la variable $k_{multiples} := 0$, la cual guarda la suma de $\sum_{a=0}^{i-1} l_a$ para poder hacer los cálculos más fácilmente.
2. Se inició dp como una matriz de dimensiones $1 \times m+1$ a la que se le añaden filas a medida que se necesiten. Esta matriz usa la relación de recurrencia con $i = 0$, y se ve de la forma: $[1, 0, 0, \cdots, 0]$
3. Se inicializó i en 1 ya que el caso de $i = 0$ ya había sido cubierto.
4. Se creó un while que simboliza: *mientras existan elementos en la cota del tercer caso de la relación de recurrencia*
5. Se creó la variable $numero\_de\_caminos := 0$ que guarda la suma de los últimos elementos en cada fila. Para no tener que sumar la última columna en otro ciclo al final.
6. Dentro del ciclo se añade una nueva fila a la matriz dp asociada, inicializada en ceros. Representando todos lo puntos a los que se puede llegar dado un l_i.
7. La nueva fila de dp se actualiza con la relación de recurrencia.
8. Se actualizan en orden los valores de $numero\_de\_caminos$, $k\_multiples$, $k = l_{i+1}$ e i.

II. Complejidad temporal y espacial

count_divisible_paths: Se encarga de ir contando la cantidad de caminos que hay para llegar desde la posición 0 hasta la posición m, primero se hace un ciclo sobre la cantidad máxima de movimientos que se puede hacer y luego se hace la ecuación de recurrencia sobre la matriz desde el múltiplo de k en que nos encontramos hasta m.

El **número de movimientos** máximos que pueden ocurrir viene siendo la sumatoria de $k+k+1+k+2+...$ hasta llegar a n movimientos, lo que esta dado por la sumatoria

$$nk + \sum_{i=1}^{n} i = nk + \frac{n*(n+1)}{2} \leq m$$

Este número de movimientos siempre tiene que ser menor o igual al valor m. Por lo que tenemos que en el peor caso el número de movimientos (n) es $\mathcal{O}(\sqrt{m})$.

La **complejidad espacial** viene dada por el tamaño de la matriz *dp* que creamos para almacenar la cantidad de formas en que se puede llegar hasta m. El número de columnas de la matriz esta dado por m, mientras que la cantidad de filas en el peor de los casos es el número máximo de movimientos (n) que se pueden dar, que como explicamos anteriormente es $C\sqrt{m}$ para un $C > 0$. Luego la complejidad espacial esta dada por,

$$\mathcal{O}(m \cdot C\sqrt{m}) = \mathcal{O}(m \cdot \sqrt{m})$$

Por su parte, la **complejidad temporal** está dada por el ciclo interno, que se ejecuta en el peor de los casos m veces, cuando k es 1. En cada una de estas iteraciones se hacen una serie de operaciones sobre los elementos de la matriz que son constantes. Luego el ciclo externo se ejecuta la el número máximo de movimientos $\mathcal{O}(\sqrt{m})$ Por lo que la complejidad temporal de esta solución es

$$\mathcal{O}(m * \sqrt{m})$$

. Tenemos que la complejidad espacial es una cota de nuestra complejidad temporal. Esto es debido a que las iteraciones que se hacen en el ciclo interno no se hacen sobre el tamaño total de la fila, si no que desde k hasta m+1. A pesar de que k puede ser uno, este va aumentando también en cada iteración. Por lo que no necesariamente se recorren todas las columnas para una fila dada.

$$\mathcal{O}(temporal) \leq \mathcal{O}(espacial)$$

III. Comprensión de problemas algorítmicos

ESCENARIO 1: suponga que tiene un número limitado de movimientos t para llegar a un valor determinado m.
Este escenario presupone un reto menor para nuestra solución. Esto debido a que nuestra implementación crea una matriz por medio de programación dinámica y va agregando 1 fila en cada iteración siempre y cuando los saltos dados por la variable k no excedan a m. De esta manera, simplemente sería necesario agregar la condición (operador AND) que lleve un contador de las filas de la matriz *dp* (en nuestro código podría ser la misma variable i) tal que se verifique que contador_filas $\leq t$ en el *while*, i.e, en el ciclo externo.

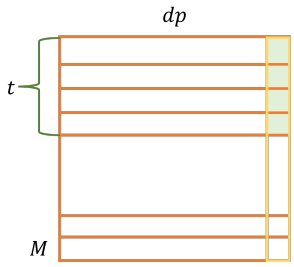

Figura 1. Representación gráfica del Escenario 1 dado una restricción t

La representación gráfica de este escenario se muestra en la Figura 1, en donde M es el número de movimentos permitidos sin restricción. Generalmente, la solución viene dada por,

$$\sum_{i=1}^{M} dp[i][-1]; \;\; t \geq M \tag{1}$$

en donde $t \geq M$ se tiene por el operador AND, la matriz no tendrá más filas que M. Ahora bien, con la restricción dada la ecuación sería,

$$\sum_{i=1}^{t} dp[i][-1]; \;\; t < M \tag{2}$$

ESCENARIO 2: El robot no parte de cero sino de un número $p > 0$ primo.
Este escenario tampoco representa un reto mayor para nuestra solución. Esto teniendo en cuenta que lo realmente importante no es la posición inicial (ya sea 0 o un número primo) sino los saltos que se dan entre estos 2 números dados por la longitud del intervalo. En este orden de ideas, dada una pareja (p, m) se puede definir una función tal que,

$$(p, m) \rightarrow (0, m - p); \;\; m > p$$

en donde $(0, m-p, k)$ es una tripla que nuestro algoritmo sabe resolver dado un k arbitrario. En términos de código, solo se necesitaría una línea de códio dada por $m = m - p$ y llame a nuestra función sobre este número.

IV. Ejecución del código

Para ejecutar el código se debe hacer cd en terminal a la carpeta en donde se encuentra proyecto.py y ejecutar:

python3.10 proyecto3.py < P0.in > P1.out