Universidad del Valle de Guatemala
CC3041 Análisis y Diseño de Algoritmos
Sección 20
Catedrático: Gabriel Brolo

Proyecto 2 - DaC & Programación Dinámica

Diego Andrés Morales Aquino - 21762
Pablo Andrés Zamora Vásquez - 21780

Guatemala, 1 de abril de 2024

Descripción del problema a resolver

La premisa básica del problema es encontrar la cantidad mínima de monedas necesarias para dar un cambio (vuelto) específico, utilizando un conjunto dado de denominaciones de monedas.

Dado un monto total K y una serie de números enteros positivos coin[] de tamaño m, representando las diferentes denominaciones de monedas disponibles. Se requiere encontrar la combinación mínima de elementos de coin[], cuyo valor sumado sea K. Es importante considerar que se dispone de un suministro infinito para cada tipo de moneda con el valor de coin[0] hasta coin[m-1].

Ejemplo 1:

Entrada: coin[] = [1, 5, 10], K= 10
Salida: 1

Las denominaciones de monedas con las que se cuenta, permite realizar distintas combinaciones para obtener un total de K=10. Sin embargo, al utilizar una sola moneda de 10 se obtiene la cantidad mínima de monedas necesarias.

Ejemplo 2:

Entrada: coin[] = [5, 9, 10, 20], K=45
Salida: 3

Estas son algunas de las combinaciones posibles con las denominaciones de monedas con las que se cuenta. El resultado es tres, pues al utilizar tres monedas (dos monedas de 20 y una de 5) se obtiene la cantidad mínima de monedas necesaria para alcanzar el monto de cambio K=45.

Algoritmos de solución

Algoritmo Divide and Conquer

Este problema puede ser resuelto utilizando el enfoque Divide and conquer, dado que puede aplicarse un enfoque de recursividad a través de sus 3 fases:

- *Divide:* separar el problema inicial en problemas más pequeños. Para cada una de las denominaciones coin[i] en donde i=0,1,2…m, encontrar el número mínimo de monedas para devolver un cambio de cantidad k - coin[i].

- *Conquer:* En esta etapa, implementamos una solución recursiva para abordar los subproblemas. Para cada subproblema, determinamos el número mínimo de monedas necesario para alcanzar el valor de cambio deseado k, al evaluar como otro subproblema cada una de las denominaciones coin[i] y hallar el respectivo mínimo de monedas para k - coin[i]. Este proceso se repite hasta llegar al caso trivial, donde k = 0 y no se requieren monedas

- *Combine:* en este problema, la fase *combine* se produce en cada nivel de la recursión, al momento de obtener el número mínimo de monedas utilizadas al escoger primero las denominaciones i=0,1,2…m, seleccionando el número mínimo + 1. Esto se expresa como minCoins(coins[0:m], k) = min {1 + minCoins(k - coin[i])} para i = 0, 1, 2 … m, donde coin[i] ≤ k.

- Pseudocódigo

```
Algoritmo minCoins(monedas[], m, K):||s||t

    # Caso base, si el monto del cambio K es cero, no hacen falta monedas
    si K == 0:
        return 0

    # Inicializar resultado como número muy grande
    minCount = INFINITY

    # Probar cada moneda y escoger el resultado menor
    para i de 0 a m:
        si monedas[i] ≤ K:
            # Llamada recursiva para resolver subproblema
            tempResult = 1 + minCoins(monedas, m, k - monedas[i])

            # Actualizar resultado si es menor al anterior
```

```
            si tempResult < minCount:
                minCount = tempResult

    return minCount
```

- Implementación

```python
import sys
def minCoinChange (coin, m, K):
    if K == 0:
        return 0
    minCount = sys.maxsize
    for i in range (m) :
        if coin[i] <= K:
            tempResult = 1 + minCoinChange (coin, m, K - coin[i])
            if tempResult < minCount:
                minCount = tempResult + 1

    if minCount == sys.maxsize:
        return -1
    return minCount
```

Algoritmo Programación Dinámica

Es posible refinar el anterior enfoque de *Divide and Conquer* para resolver el problema en cuestión mediante programación dinámica, puesto que este cuenta con **subproblemas traslapados**; es decir, cada vez que se elige una moneda de diferente denominación, se vuelven a calcular todas las posibles combinaciones restantes que sumen el monto del cambio, en lugar de almacenar los resultados de los subproblemas para evitar realizar nuevamente dicho cálculo. Para ello, es necesario realizar los siguientes pasos:

- **Caracterizar la solución óptima:** La solución óptima es la mínima cantidad de monedas, sin importar su denominación, necesaria para alcanzar el valor del cambio "K". Esto implica que para cada valor "i" entre 0 y "K", también se debe buscar la cantidad mínima de monedas necesaria para alcanzar "i". Esta es la subestructura óptima.

- **Describir el valor de la solución óptima de forma recursiva:** La mínima cantidad de monedas necesaria para alcanzar cada valor "i" que forma parte de la solución óptima puede describirse como el mínimo entre la solución actual para "i" y 1 + la solución óptima de *"i - j"*, para cada moneda "j" entre las denominaciones disponibles. Esto se basa en la premisa de que si sabemos la solución para *"i - j"*, entonces agregando una moneda de denominación "j" a dicha solución, obtenemos una solución para el valor de "i" (de ahí por qué se suma un "1").

- **Computar el valor de la solución óptima:** Debido a la naturaleza del problema, resulta conveniente utilizar un enfoque iterativo *bottom-up*, en donde se inicia con el caso base de que 0 monedas son necesarias para alcanzar un valor de cambio “K” de 0. Luego, gradualmente se construyen todas las soluciones para las cantidades “i” desde 1 hasta “K”, utilizando los resultados de los subproblemas anteriores para informar y calcular el resultado de los subproblemas siguientes (Esto se explica más a detalle en el pseudocódigo).

- Pseudocódigo

```
Algoritmo minCoins(monedas[], m, K):

    # Caso base, si el monto del cambio K es cero, no hacen falta
monedas
    si K == 0:
        return 0

  # Inicializar un arreglo de una dimensión para almacenar los
resultados de los subproblemas. En un inicio, el resultado de cada
uno es un número muy grande.
   change = [INFINITY] * (K+1)

   # El primer resultado es el caso base: 0
   change[0] = 0

  # Se recorre cada valor “i” desde 1 hasta “K”
  para i de 1 hasta K:
    para j de 0 hasta m: #Se intenta con cada denominación
      si monedas[j] <= i:
        # Verificar la solución de un subproblema anterior
        newResult = 1 + change[i - coin[j]]
       # Si esta solución es mejor que la actual, reemplazarla
        si newResult < change[i]:
          change[i] = newResult

    return cambio[K]
```

- Implementación

```python
def minCoinChange (coin, m, K):
    if K == 0:
        return 0
    change = [sys.maxsize] * (K + 1)
    change[0] = 0
    for i in range(1, K + 1) :
        for j in range(m):
            if coin[j] <= i:
                currentCount = 1 + change[i - coin[j]]
                if currentCount < change[i]:
                    change[i] = currentCount

    if change[K] == sys.maxsize:
        return -1
    return change[K]
```

Análisis teórico

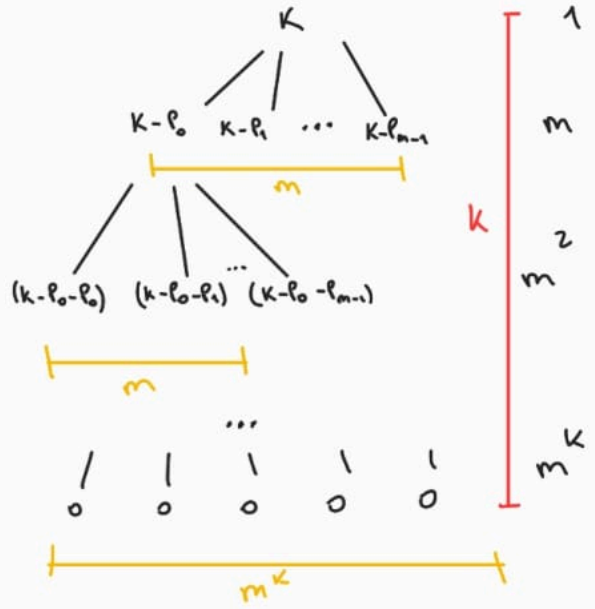
Algoritmo Divide and Conquer

```
Algoritmo minCoins(monedas[], m, K):||s||t

    # Caso base, si el monto del cambio K es cero, no hacen falta monedas
    si K == 0:      1
        return 0    1

    # Inicializar resultado como número muy grande
    minCount = INFINITY  1

    # Probar cada moneda y escoger el resultado menor
    para i de 0 a m:    m*1
        si monedas[i] ≤ K:      m
            # Llamada recursiva para resolver subproblema
            tempResult = 1 + minCoins(monedas, m, k - monedas[i])  m

            # Actualizar resultado si es menor al anterior
            si tempResult < minCount:       m
                minCount = tempResult       5

    return minCount 1
```

$$T(K,m) = \begin{cases} O(1) & ; \quad K = 0 \\ m \cdot (T(K - P_i, m) + c) & ; \quad K > 0 \end{cases}$$

P_i corresponde al valor de monedas[i] para $i = 0, 1, \dots m$

Worst-Case: $P_i \leq K$ en cada nivel de recursión.



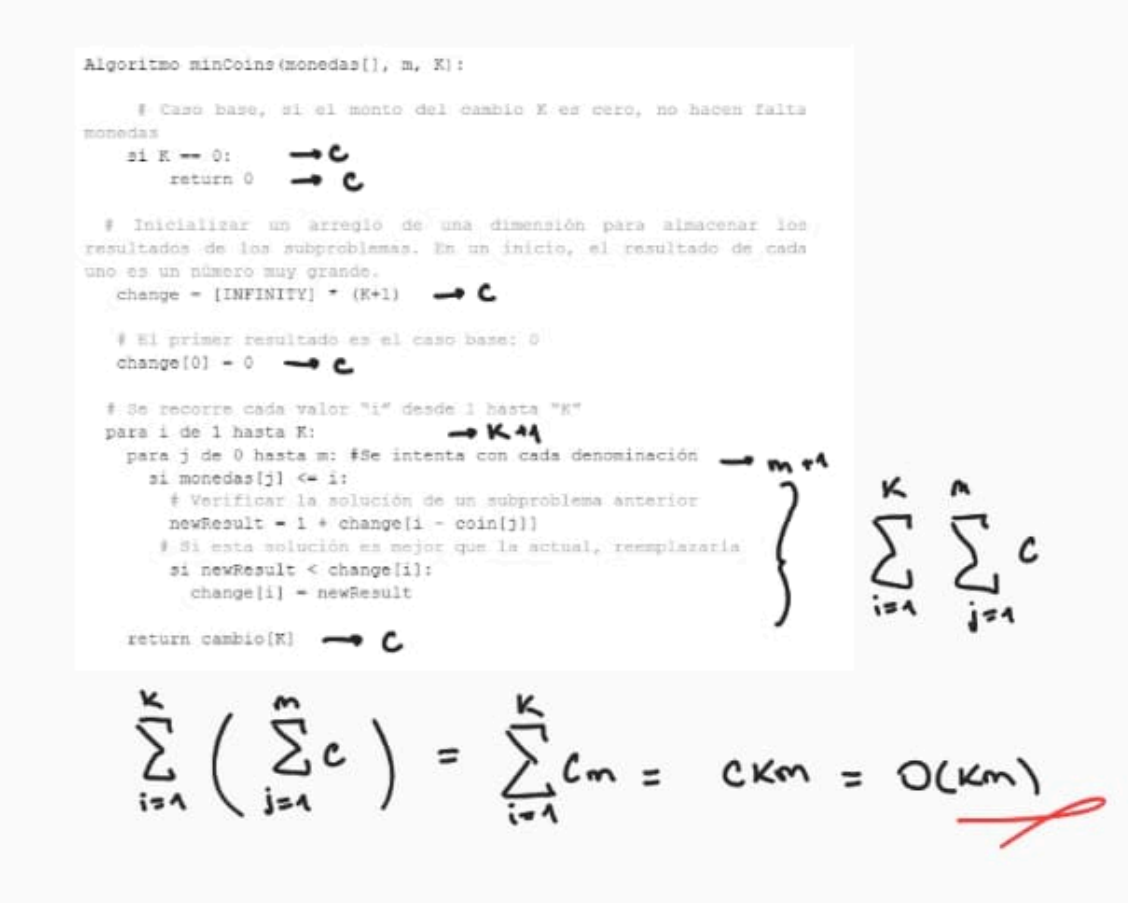
$$T(K,m) \leq c \sum_{i=0}^{m} m^i$$

$$\leq \frac{m^{K+1} - 1}{m - 1} = \frac{m(m^K) - 1}{m - 1}$$

$$T(K,m) = O(m^K)$$

La complejidad del algoritmo de *Divide and Conquer* es **$O(m^K)$.**

Algoritmo Programación Dinámica



```
Algoritmo minCoins(monedas[], m, K):

    # Caso base, si el monto del cambio K es cero, no hacen falta
monedas
    si K == 0:          → c
        return 0        → c

  # Inicializar un arreglo de una dimensión para almacenar los
resultados de los subproblemas. En un inicio, el resultado de cada
uno es un número muy grande.
   change = [INFINITY] * (K+1)    → c

   # El primer resultado es el caso base: 0
   change[0] = 0                  → c

  # Se recorre cada valor "i" desde 1 hasta "K"
  para i de 1 hasta K:            → K+1
    para j de 0 hasta m: #Se intenta con cada denominación   → m+1
      si monedas[j] <= i:
        # Verificar la solución de un subproblema anterior
        newResult = 1 + change[i - coin[j]]
       # Si esta solución es mejor que la actual, reemplazaría
        si newResult < change[i]:
          change[i] = newResult

   return cambio[K]               → c
```

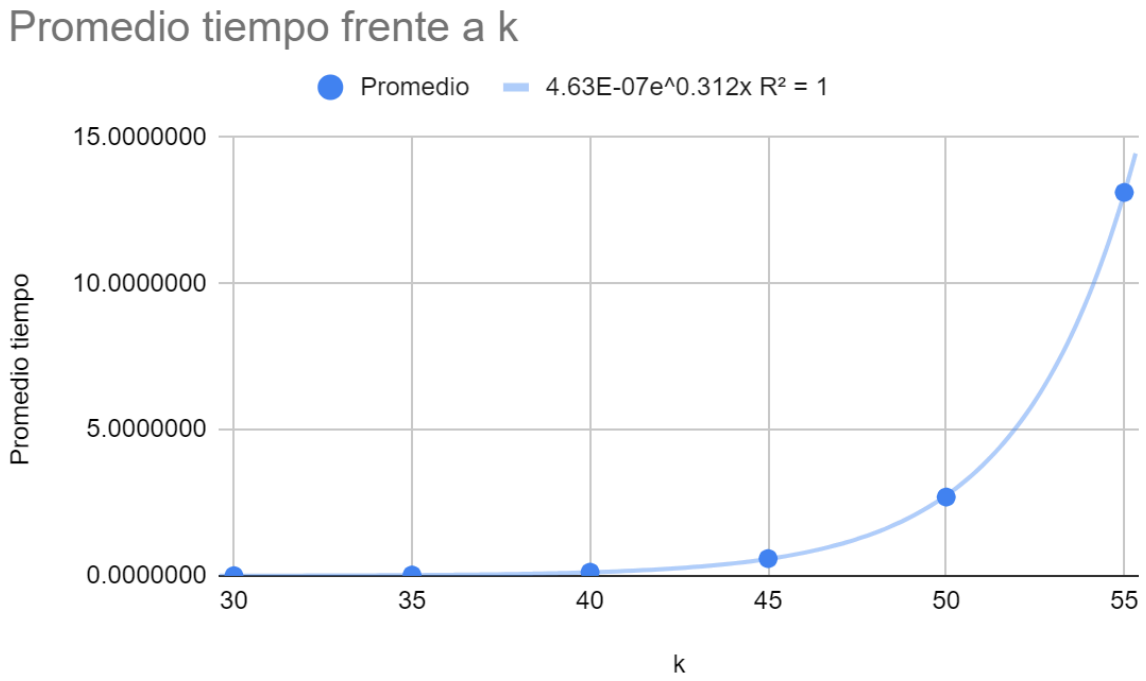
$$\sum_{i=1}^{K}\left(\sum_{j=1}^{m} c\right) = \sum_{i=1}^{K} cm = ckm = O(km)$$

La complejidad del algoritmo de Programación Dinámica es **O(km).**

Análisis empírico

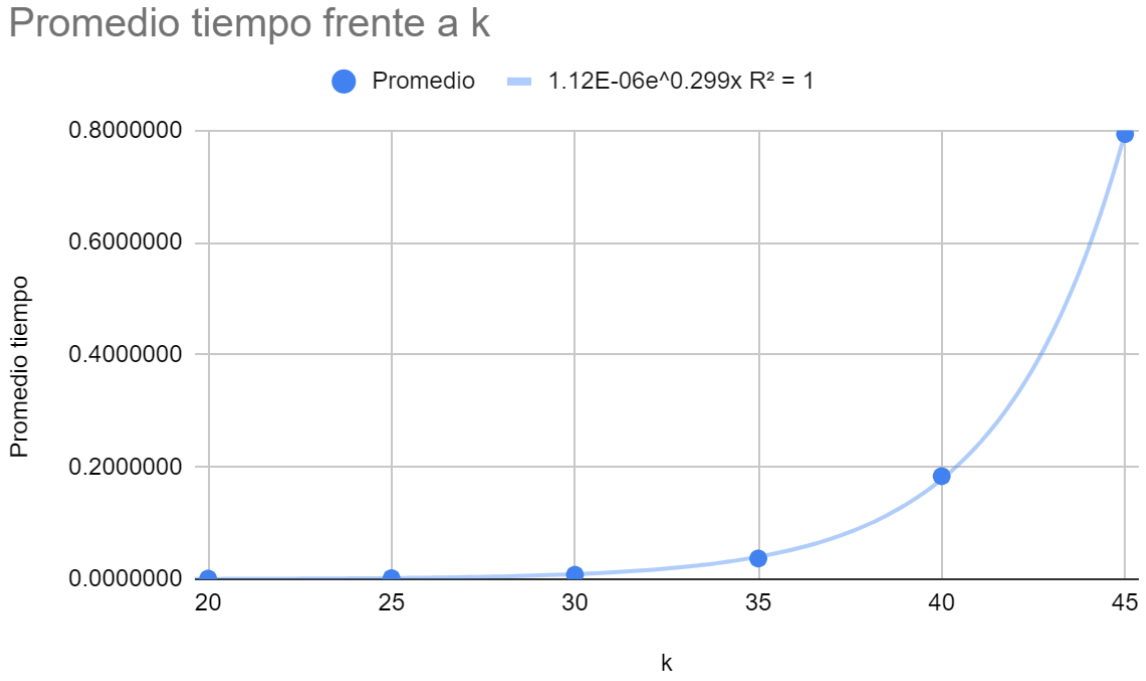
Algoritmo Divide and Conquer

m	k	Tiempos	Promedio
3	30	0.0067267	0.0063555
		0.0050056	
		0.0067365	
		0.0073531	
		0.0059557	
3	35	0.0338733	0.0283281
		0.0262423	
		0.0296428	
		0.0250056	
		0.0268767	
3	40	0.1362174	0.1252891
		0.1207256	
		0.1224442	
		0.1151412	
		0.1319170	
3	45	0.6143100	0.5906578
		0.5689466	
		0.5588951	
		0.5593266	
		0.6518104	
3	50	2.6865644	2.6972924
		2.6944511	
		2.6885917	
		2.6508191	
		2.7660356	
3	55	12.2390349	13.1229232
		13.1088772	
		13.4629645	
		13.3000851	
		13.5036542	



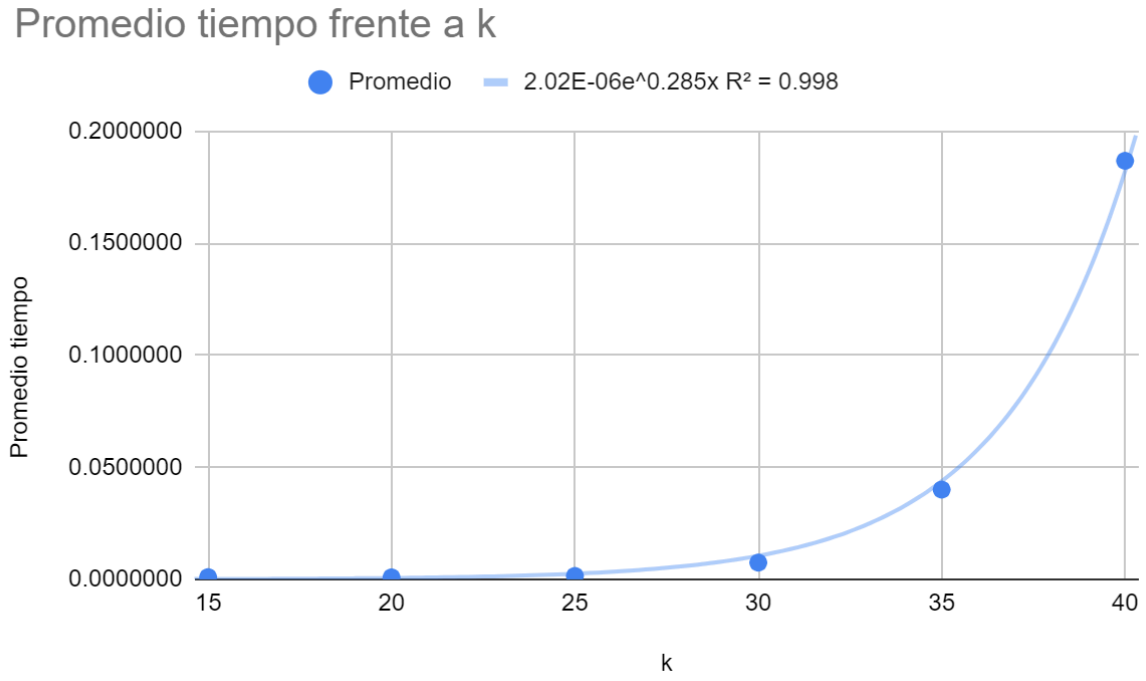
m	k	Tiempos	Promedio
4	20	0.0010257	0.0010103
		0.0010176	
		0.0010087	
		0.0009985	
		0.0010009	
4	25	0.0020001	0.0017629
		0.0019848	
		0.0022068	
		0.0009992	
		0.0016236	
4	30	0.0100260	0.0078511
		0.0078907	
		0.0072587	
		0.0069611	
		0.0071192	
4	35	0.0440910	0.0368495
		0.0347486	
		0.0359480	
		0.0318050	
		0.0376546	

4	40	0.1665132	0.1836205
		0.1719341	
		0.2345989	
		0.1825709	
		0.1624851	
4	45	0.7663794	0.7935199
		0.7768149	
		0.8106544	
		0.8196402	
		0.7941105	

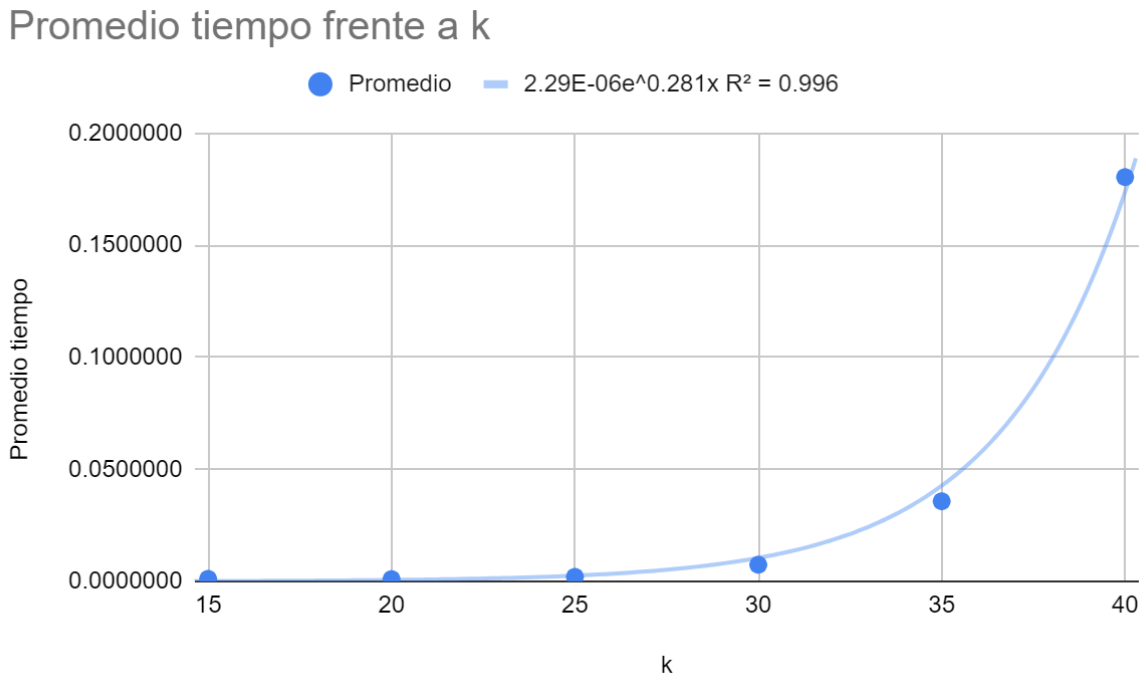


m	k	Tiempos	Promedio
5	15	0.0009937	0.0009977
		0.0009999	
		0.0009813	
		0.0009968	
		0.0010169	
5	20	0.0010004	0.0008929
		0.0009499	
		0.0010099	
		0.0010009	
		0.0005033	

5	25	0.0020111	0.0015209
		0.0009925	
		0.0016303	
		0.0009868	
		0.0019836	
5	30	0.0069940	0.0074760
		0.0069966	
		0.0070355	
		0.0097287	
		0.0066254	
5	35	0.0351286	0.0400554
		0.0387518	
		0.0392196	
		0.0465710	
		0.0406058	
5	40	0.2013206	0.1867958
		0.1771572	
		0.1904685	
		0.1824713	
		0.1825612	

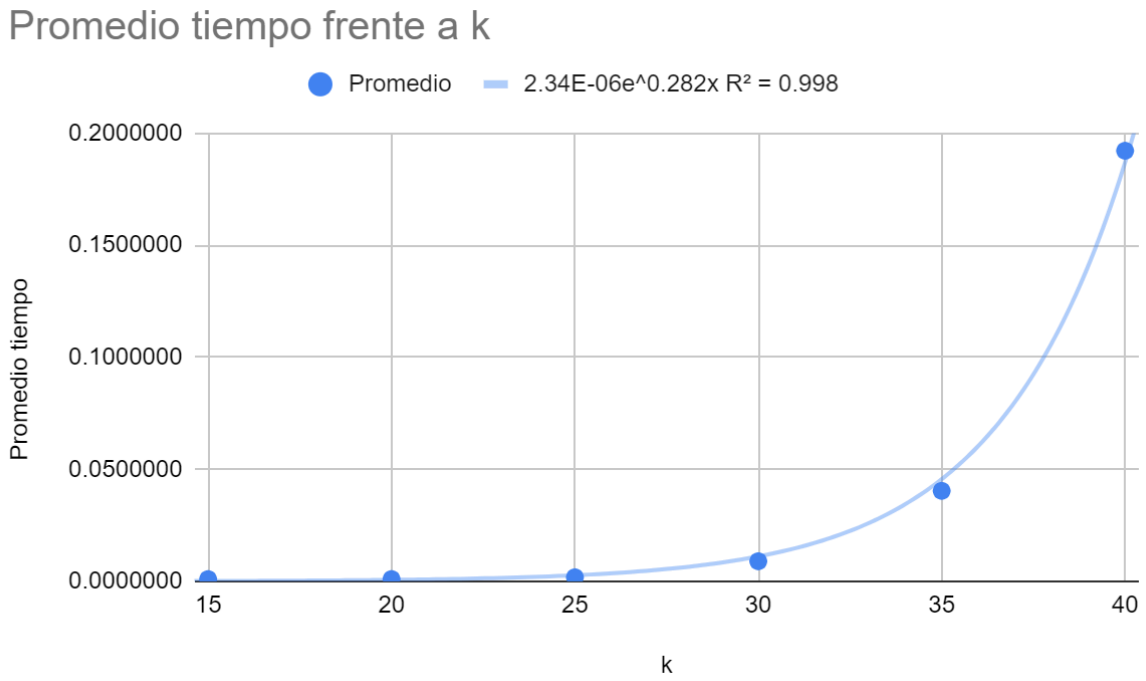


m	k	Tiempos	Promedio
6	15	0.0010130	0.0011188
		0.0016170	
		0.0010002	
		0.0010009	
		0.0009627	
6	20	0.0010195	0.0010058
		0.0009768	
		0.0009995	
		0.0010045	
		0.0010290	
6	25	0.0020368	0.0020166
		0.0020013	
		0.0020206	
		0.0009947	
		0.0030298	
6	30	0.0081875	0.0074099
		0.0070114	
		0.0076423	
		0.0069926	
		0.0072157	
6	35	0.0356505	0.0357343
		0.0350008	
		0.0370955	
		0.0359981	
		0.0349267	
6	40	0.1765230	0.1804193
		0.1811256	
		0.1713398	
		0.1737449	
		0.1993635	



m	k	Tiempos	Promedio
7	15	0.0009999	0.0010033
		0.0010166	
		0.0009761	
		0.0009997	
		0.0010240	
7	20	0.0010116	0.0010208
		0.0009902	
		0.0009987	
		0.0009995	
		0.0011039	
7	25	0.0020316	0.0018168
		0.0020564	
		0.0019979	
		0.0010002	
		0.0019979	
7	30	0.0093470	0.0089712
		0.0094047	
		0.0088682	
		0.0080032	
		0.0092330	
7	35	0.0411842	0.0403889

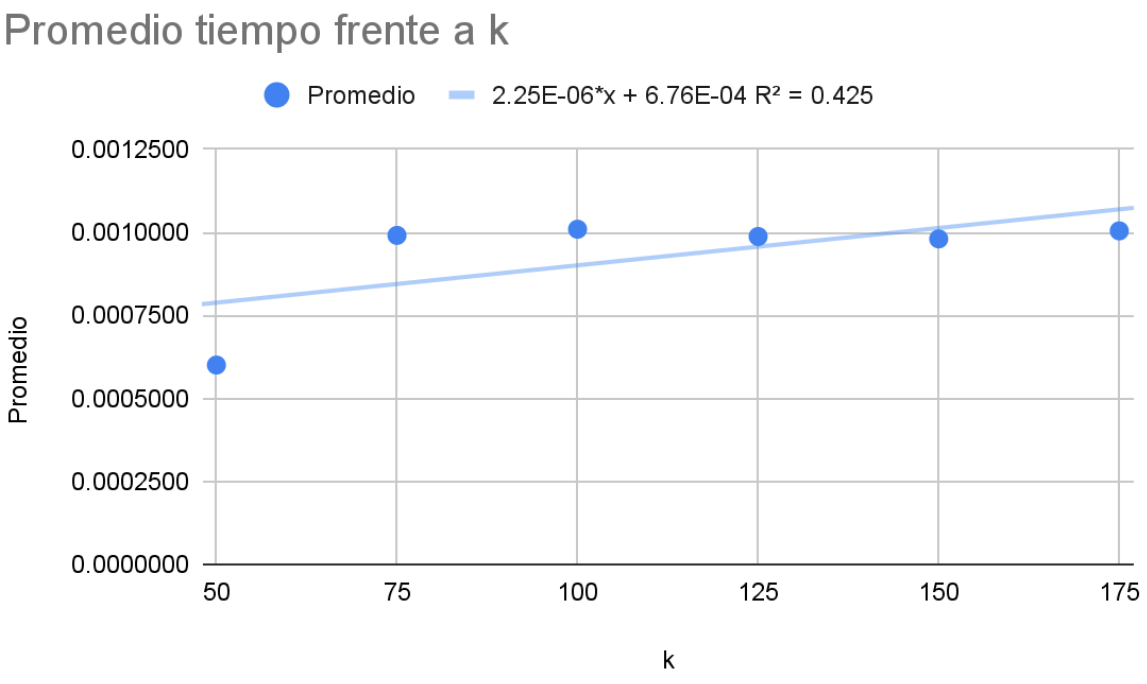
		0.0382111	
		0.0353878	
		0.0425317	
		0.0446296	
7	40	0.2062135	0.1921739
		0.1838071	
		0.1865592	
		0.1979012	
		0.1863883	



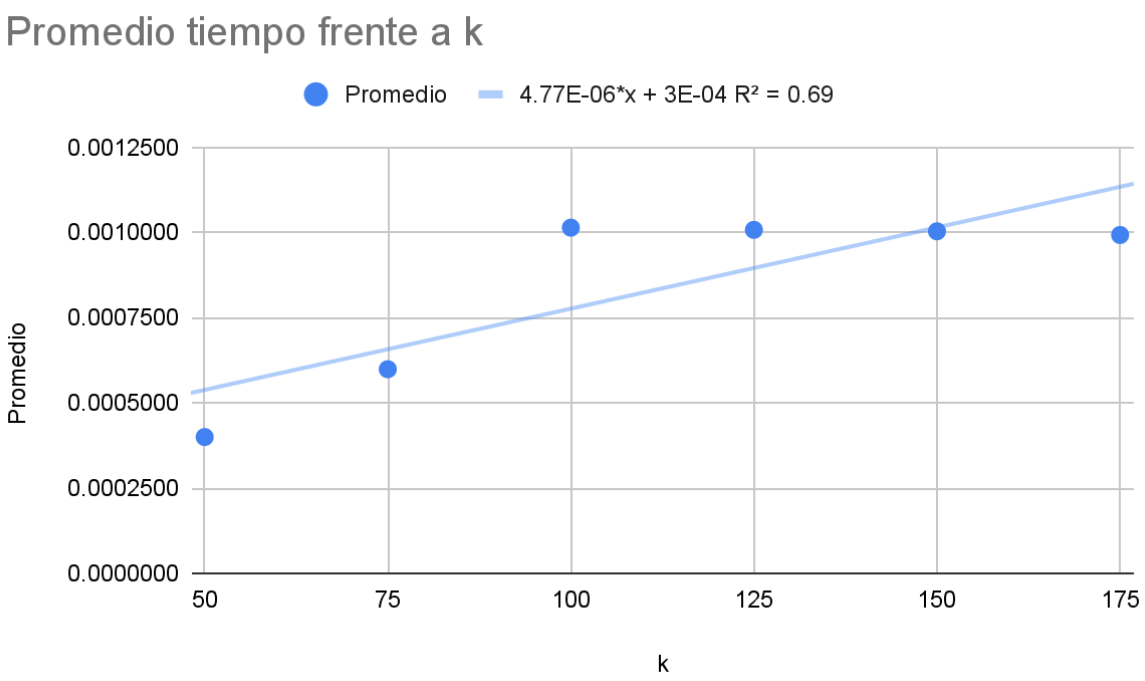
Algoritmo Programación Dinámica

m	k	Tiempos	Promedio
11	50	0.0010102	0.0006004
		0.0009923	
		0.0000000	
		0.0009995	
		0.0000000	
11	75	0.0009987	0.0009910
		0.0009978	
		0.0010149	
		0.0009990	

		0.0009446	
11	100	0.0010126	0.0010096
		0.0009985	
		0.0010266	
		0.0010002	
		0.0010102	
11	125	0.0009928	0.0009877
		0.0010080	
		0.0010266	
		0.0009151	
		0.0009959	
11	150	0.0009832	0.0009805
		0.0009367	
		0.0010002	
		0.0010004	
		0.0009818	
11	175	0.0010076	0.0010046
		0.0010126	
		0.0009983	
		0.0010142	
		0.0009906	

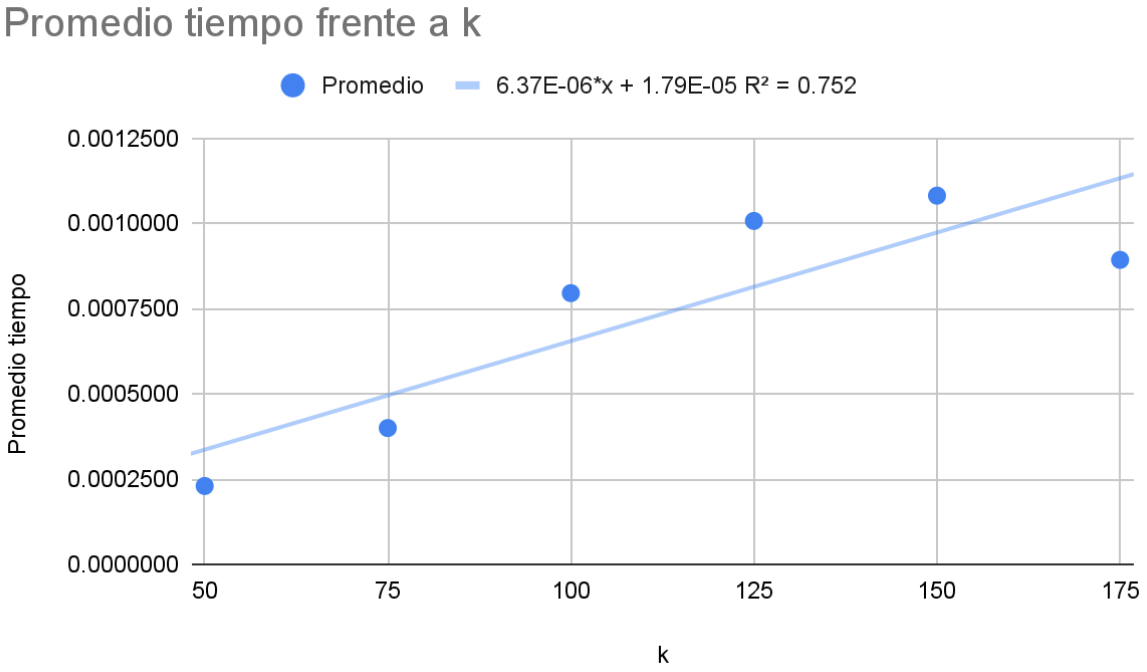


m	k	Tiempos	Promedio
12	50	0.0000000	0.0003999
		0.0009997	
		0.0000000	
		0.0009999	
		0.0000000	
12	75	0.0009809	0.0005989
		0.0010135	
		0.0000000	
		0.0009999	
		0.0000000	
12	100	0.0010064	0.0010141
		0.0010374	
		0.0009642	
		0.0010755	
		0.0009873	
12	125	0.0009968	0.0010077
		0.0010006	
		0.0010240	
		0.0010037	
		0.0010133	
12	150	0.0009964	0.0010035
		0.0009966	
		0.0009999	
		0.0010188	
		0.0010056	
12	175	0.0009546	0.0009926
		0.0010633	
		0.0009995	
		0.0010011	
		0.0009446	



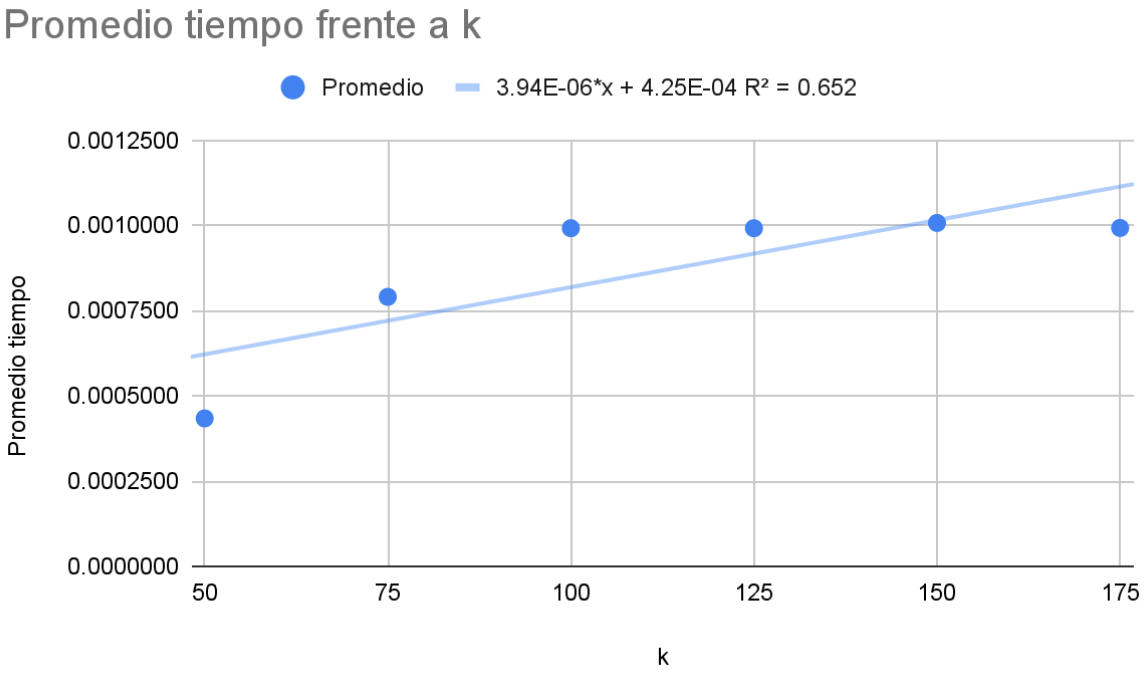
m	k	Tiempos	Promedio
13	50	0.0001547	0.0002298
		0.0000000	
		0.0009944	
		0.0000000	
		0.0000000	
13	75	0.0000000	0.0003998
		0.0009997	
		0.0000000	
		0.0000000	
		0.0009995	
13	100	0.0009952	0.0007957
		0.0009859	
		0.0000000	
		0.0009985	
		0.0009987	
13	125	0.0010023	0.0010075
		0.0010219	
		0.0010002	
		0.0010133	
		0.0009997	

13	150	0.0009956	0.0010817
		0.0010118	
		0.0010536	
		0.0012982	
		0.0010493	
13	175	0.0009742	0.0008933
		0.0010014	
		0.0009987	
		0.0004914	
		0.0010006	



m	k	Tiempos	Promedio
14	50	0.0000000	0.0004340
		0.0001788	
		0.0010023	
		0.0000000	
		0.0009890	
14	75	0.0000000	0.0007906
		0.0009377	
		0.0009997	

		0.0010133	
		0.0010023	
14	100	0.0009997	0.0009920
		0.0009995	
		0.0009816	
		0.0010054	
		0.0009737	
14	125	0.0010037	0.0009919
		0.0009170	
		0.0009940	
		0.0010445	
		0.0010004	
14	150	0.0010250	0.0010075
		0.0010021	
		0.0009997	
		0.0009995	
		0.0010111	
14	175	0.0010004	0.0009928
		0.0010076	
		0.0009511	
		0.0010056	
		0.0009992	



m	k	Tiempos	Promedio
15	50	0.0009997	0.0005998
		0.0000000	
		0.0000000	
		0.0009997	
		0.0009995	
15	75	0.0010006	0.0008894
		0.0004449	
		0.0010018	
		0.0009990	
		0.0010006	
15	100	0.0009997	0.0010200
		0.0009534	
		0.0009999	
		0.0011537	
		0.0009930	
15	125	0.0009992	0.0009925
		0.0010028	
		0.0009995	
		0.0009997	
		0.0009615	

15	150	0.0010073	0.0010820
		0.0010076	
		0.0009999	
		0.0009999	
		0.0013952	
15	175	0.0009995	0.0010727
		0.0009992	
		0.0010009	
		0.0010037	
		0.0013602	

El análisis empírico del algoritmo *DaC* demuestra un ajuste casi perfecto de los tiempos de ejecución obtenidos a una regresión exponencial, tal y como se esperaba, puesto que, sin importar *k* y *m*, en cada nivel del árbol de recursión, desde 0 hasta *k*, la cantidad de nodos siempre será m^i.

Por otro lado, el análisis empírico del algoritmo con programación dinámica demuestra que, contrario a lo que se esperaba, los tiempos de ejecución obtenidos no se ajustan muy bien a una regresión lineal. Esto se debe a que el análisis teórico se realizó para el *worst case scenario*; este es, aquel en el que ninguno de los subproblemas permita obtener el resultado de los subproblemas siguientes. Esta sería una situación muy específica en el contexto del *Minimum Coin Change Problem*, ya que, al tratarse de encontrar la cantidad mínima de monedas necesarias para alcanzar cada valor *i* entre 0 y *k*, si se logra optimizar la cantidad de monedas para tan siquiera uno de estos valores, este resultado puede ser utilizado sucesivamente en los valores de *i* siguientes, evitando que se produzca el *worst case*. Esto explica por qué los tiempos de ejecución fueron menores a los esperados *(m x k*).

Referencias

Gautam, S. (s.f.). *Minimum Coin Change Problem*. Obtenido de: https://www.enjoyalgorithms.com/blog/minimum-coin-change