

“Año del Bicentenario, de la consolidación de nuestra Independencia, y de la conmemoración de las heroicas batallas de Junín y Ayacucho”

Universidad Nacional Mayor de San Marcos

Universidad del Perú, Decana de América



Facultad de Ingeniería de Sistemas e Informática

Escuela Profesional de Ingeniería de Sistemas

Proyecto de fin de curso

Profesor: Luis Guerra Grados

Integrantes del grupo N° 2:

Campos García, Henry Leonardo

Escribas Alan, Daniel Leonardo

Meléndez Blas, Jhair Roussell

Morales Damasco, Cristian Ricardo

Ortis Herrera, Fabrizio Peter

23 / junio / 2024

Índice

Índice	1
1. Introducción	2
2. Objetivos	2
2.1. Objetivo General	2
2.2. Objetivos Específicos	3
3. Planteamiento del Problema	3
4. El problema del cambio de monedas	3
4.1. Ejemplo 1	3
4.1.1. Código	4
4.1.2. Resultado	5
4.1.3. Análisis de complejidad	6
4.2. Ejemplo 2	7
4.2.1. Código	9
4.2.2. Resultado	11
4.2.3. Análisis de complejidad	11
5. El Problema de la Mochila	12
5.1. Ejemplo 1	12
5.2. Código	12
5.3. Resultado	12
6. El Problema de las Distancias Más Cortas	12
6.1. Ejemplo 1 Algoritmo de Floyd-Warshall	13
6.2. Código	13
6.3. Resultado	13
7. Conclusiones	13

1. Introducción

Imagina estar en una tienda y necesitar dar el cambio exacto con la menor cantidad de monedas posible; en una empresa de transporte marítimo que debe seleccionar las mercancías más valiosas sin sobrecargar el barco; o en un repartidor que necesita encontrar la ruta más corta para entregar pedidos a varias casas. Estos escenarios cotidianos ilustran problemas de optimización complejos que pueden ser resueltos eficientemente mediante la programación dinámica. Esta técnica se basa en dividir problemas grandes en subproblemas más pequeños y manejables reutilizando las soluciones de estos subproblemas para construir una solución óptima de manera eficiente.

La programación dinámica no solo es eficaz sino también versátil se puede encontrar aplicaciones en diversas áreas como la logística la gestión de recursos la ingeniería de redes y la toma de decisiones. La complejidad algorítmica de los algoritmos de programación dinámica varía según la estructura del problema y el tamaño de los parámetros de entrada. Esto significa que la eficiencia de estos algoritmos está directamente relacionada con el tamaño del problema haciendo que la programación dinámica sea una herramienta invaluable en la resolución de problemas de gran escala.

En ese sentido el presente informe se centra en tres problemas clásicos resueltos mediante programación dinámica: el problema del cambio de monedas, el problema de la mochila y el problema de las distancias más cortas. A través del análisis y la implementación de algoritmos para estos problemas se demuestra la eficacia de la programación dinámica en la optimización de recursos y la toma de decisiones ofreciendo soluciones prácticas y eficientes en contextos diversos.

2. Objetivos

2.1. Objetivo General

Desarrollar y analizar algoritmos eficientes basados en programación dinámica para resolver problemas clásicos de optimización demostrando su aplicabilidad y eficiencia en diferentes contextos.

2.2. Objetivos Específicos

- Implementar un algoritmo de programación dinámica para resolver el problema del cambio de monedas con restricciones en la cantidad disponible.
- Desarrollar un algoritmo basado en programación dinámica para maximizar los ingresos en el problema de la mochila.
- Aplicar el algoritmo de Floyd-Warshall para encontrar las rutas de menor coste en un grafo optimizando la logística de entrega.
- Analizar la complejidad algorítmica de cada algoritmo implementado destacando los factores que afectan su rendimiento.

3. Planteamiento del Problema

¿De qué manera la programación dinámica resulta ser una solución eficiente y precisa para resolver problemas de optimización como el cambio de la moneda, el problema de la mochila o la selección de rutas óptimas en diversos contextos comerciales?

4. El problema del cambio de monedas

El problema del cambio de monedas, un desafío clásico en el campo de la optimización combinatoria y las ciencias de la computación, involucra determinar el número mínimo de monedas que se requieren para sumar una cantidad específica de dinero. Este problema se complica aún más cuando las monedas disponibles están limitadas en cantidad. La programación dinámica ofrece una metodología robusta para abordar este tipo de problemas permitiendo una solución eficiente y efectiva.

4.1. Ejemplo 1

Un cliente paga una cuenta de 6.20 soles con un billete de 10 soles. El cajero necesita dar el cambio con la menor cantidad de monedas posible. Las denominaciones disponibles son monedas de 1 sol, 50 céntimos, 20 céntimos y 10 céntimos con cantidades limitadas de cada una.

El reto es calcular la forma óptima de entregar 3.80 soles de vuelto usando las denominaciones mencionadas. Considere que tiene 2 monedas de 1 sol, seis de 50 céntimos, cinco de 20 céntimos, y diez de 10 céntimos.

4.1.1. Código

```
def min_coins_change(total, denominations, counts):  
    # Convertimos las cantidades a centimos para manejarlas como enteros  
    total_cents = int(total * 100)  
    denominations_cents = [int(d * 100) for d in denominations]  
  
    # Inicializamos la tabla de DP con infinito para todas las cantidades  
    # excepto para 0 que necesita 0 monedas  
    dp = [float('inf')] * (total_cents + 1)  
    dp[0] = 0  
  
    # Para rastrear el uso de las monedas correctamente  
    used_coins = [[0 for _ in denominations] for _ in range(total_cents + 1)]  
  
    # Procesamos cada tipo de moneda  
    for i, coin in enumerate(denominations_cents):  
        for j in range(coin, total_cents + 1):  
            if dp[j - coin] != float('inf') and counts[i] > used_coins[j - coin][i]:  
                if dp[j] > dp[j - coin] + 1:  
                    dp[j] = dp[j - coin] + 1  
                    used_coins[j] = used_coins[j - coin][:]  
                    used_coins[j][i] += 1  
  
    if dp[total_cents] == float('inf'):  
        return "No se puede dar el cambio exacto"
```

```

    else:
        return dp[total_cents], used_coins[total_cents]

# Denominaciones de las monedas disponibles y sus cantidades
denominations = [1.0, 0.50, 0.20, 0.10] # en soles
counts = [2, 6, 5, 10] # cantidad de monedas disponibles

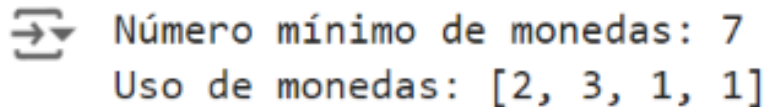
# Cantidad total a devolver
total_change = 3.80 # en soles

min_coins, coin_usage = min_coins_change(total_change, denominations,
    counts)

print("Numero mínimo de monedas:", min_coins)
print("Uso de monedas:", coin_usage)

```

4.1.2. Resultado



```

➡ Número mínimo de monedas: 7
  Uso de monedas: [2, 3, 1, 1]

```

Figura 1: imagen del resultado del código Python.

4.1.3. Análisis de complejidad

```
def min_coins_change(total, denominations, counts):  
    # Convertimos las cantidades a centimos para manejarlas como enteros } A  
    total_cents = int(total * 100)  
    denominations_cents = [int(d * 100) for d in denominations]  
  
    # Inicializamos la tabla de DP con infinito para todas las cantidades  
    # excepto para 0, que necesita 0 monedas } B  
    dp = [float('inf')] * (total_cents + 1)  
    dp[0] = 0  
  
    # Para rastrear el uso de las monedas correctamente } C  
    used_coins = [[0 for _ in denominations] for _ in range(total_cents + 1)]  
  
    # Procesamos cada tipo de moneda  
    for i, coin in enumerate(denominations_cents):  
        for j in range(coin, total_cents + 1):  
            if dp[j - coin] != float('inf') and counts[i] > used_coins[j - coin][i]:  
                # Comprobamos si usando una moneda más aún se mejora } D  
                if dp[j] > dp[j - coin] + 1:  
                    dp[j] = dp[j - coin] + 1  
                    used_coins[j] = used_coins[j - coin][:]  
                    used_coins[j][i] += 1  
  
    # Resultado en términos de monedas usadas } E  
    if dp[total_cents] == float('inf'):  
        return "No se puede dar el cambio exacto"  
    else:  
        return dp[total_cents], used_coins[total_cents]  
  
    # Denominaciones de las monedas disponibles y sus cantidades  
    denominations = [1.0, 0.50, 0.20, 0.10] # en soles  
    counts = [2, 6, 5, 10] # cantidad de monedas disponibles  
  
    # Cantidad total a devolver  
    total_change = 3.80 # en soles  
  
    # Obtener el resultado  
    min_coins, coin_usage = min_coins_change(total_change, denominations, counts)  
    print("Número mínimo de monedas:", min_coins)  
    print("Uso de monedas:", coin_usage) } F
```

Figura 2: Analisis del codigo.

$$\begin{aligned}
 T(n) &= A + B + C + D + E + F \\
 A &= O(1) \\
 B &= O(m+1) \\
 C &= \sum_{i=0}^m \sum_{j=1}^n 1 = (m+1)n \\
 D &= \sum_{i=1}^n \sum_{j=1}^m (O(1)) \\
 &= \sum_{i=1}^n \sum_{j=1}^m (O(1)) \\
 &= \sum_{i=1}^n O(m) \\
 &= O(n \cdot m) \\
 E &= O(1) + O(n) = O(n) \\
 F &= O(1) \\
 T(n) &= O(1) + O(m) + O(m \times n) + O(n \times m) + O(n) + O(1) \\
 T(n) &= O(n \times m) \\
 n &\rightarrow \text{n}^\circ \text{ de denominaciones de monedas} \\
 m &\rightarrow \text{monto total en centimos.}
 \end{aligned}$$

Figura 3: analisis del codigo.

La complejidad de este algoritmo está dada por la cantidad de denominaciones y la cantidad total a devolver, lo que resulta en una complejidad de $O(n \cdot m)$, donde n es el número de denominaciones y m es la cantidad total a devolver.

4.2. Ejemplo 2

En primer lugar, debemos pensar cómo plantear el problema de forma incremental. Consideremos el tipo de moneda de mayor valor, X_N . Si $X_N > C$ entonces la descartamos y pasamos a

considerar monedas de menor valor. Si $X_N \leq C$ tenemos dos opciones: o tomar una moneda de tipo X_N y completar la cantidad restante $C - X_N$ con otras monedas, o no tomar ninguna moneda de tipo X_N y completar la cantidad C con monedas de menor valor. De las dos opciones, nos quedamos con la que requiera un número menor de monedas. El problema lo podemos expresar de la siguiente forma cuando consideramos N tipos de monedas:

$$Cambio(N, C) = \begin{cases} cambio(N-1, C) & \text{si } X_N > C \\ \min\{cambio(N-1, C), cambio(N, C - X_N) + 1\} & \text{si } X_N \leq C \end{cases}$$

Podemos razonar análogamente para monedas de valores k menores que N y para cantidades C' menores que C :

$$Cambio(k, C') = \begin{cases} cambio(k-1, C') & \text{si } X_k > C' \\ \min\{cambio(k-1, C'), cambio(k, C' - X_k) + 1\} & \text{si } X_k \leq C' \end{cases}$$

Llegamos a los casos base de la recurrencia cuando completamos la cantidad C :

$$cambio(k, 0) = 0 \quad \text{si } 0 \leq k \leq n$$

o cuando ya no quedan más tipos de monedas por considerar, pero aún no se ha completado la cantidad C :

$$cambio(0, C') = \infty \quad \text{si } 0 < C' \leq C$$

Podemos construir una tabla para almacenar los resultados parciales que tenga una fila para cada tipo de moneda y una columna para cada cantidad posible entre 1 y C . Cada posición $t[i, j]$ será el número mínimo de monedas necesario para dar una cantidad j con $0 \leq j \leq C$ utilizando solo monedas de los tipos entre 1 e i , con $0 \leq i \leq n$. La solución al problema será, por tanto, el contenido de la casilla $t[n, C]$. Para construir la tabla, empezamos rellenando los casos base

$t[i, 0] = 0$, para todo i con $0 \leq i \leq n$. A continuación, podemos rellenar la tabla bien por filas de izquierda a derecha, o bien por columnas de arriba a abajo.

Siguiendo el método de la **programación dinámica**, se rellenará una tabla con las filas correspondientes a cada valor para las monedas y las columnas con valores desde el 1 hasta el N (12 en este caso). Cada posición (i, j) de la tabla nos indica el número mínimo de monedas requeridas para devolver la cantidad j con monedas con valor menor o igual al de i :

	0	1	2	3	4	5	6	7	8	9	10	11	12
$x = 1$	0	1	2	3	4	5	6	7	8	9	10	11	12
$x = 6$	0	1	2	3	4	5	1	2	3	4	5	2	2
$x = 10$	0	1	2	3	4	5	1	2	3	4	5	2	2

En el caso anterior, hay que pagar 12 con monedas de 1, 6, 10. Supongamos que queremos pagar 12 o pagamos con 12 monedas de 1, o 2 monedas de 6, o con una de 10 y 2 de 1. Como la mejor opción es la de las monedas de 6, me quedo con esa y es la que marco en la tabla. Obsérvese que a pesar de que con monedas de 10 me haría falta 3 monedas, marco solo 2 porque es la mejor opción.

4.2.1. Código

```
def min(a, b):
    return a if a < b else b

class Cambio:
    def __init__(self, cantidad, monedas):
        self.cantidad = cantidad
        self.vectorMonedas = monedas
        self.matrizCambio = self.calcularMonedas(cantidad, monedas)
        self.vectorSeleccion = self.seleccionarMonedas(cantidad,
                                                         monedas, self.matrizCambio)

    def getVectorSeleccion(self):
        return self.vectorSeleccion
```

```

def calcularMonedas(self, cantidad, monedas):
    matrizCambio = [[0 if j == 0 else 99 for j in range(cantidad +
        1)] for i in range(len(monedas) + 1)]

    for i in range(1, len(monedas) + 1):
        for j in range(1, cantidad + 1):
            if j < monedas[i - 1]:
                matrizCambio[i][j] = matrizCambio[i - 1][j]
            else:
                matrizCambio[i][j] = min(matrizCambio[i - 1][j],
                    matrizCambio[i][j - monedas[i - 1]] + 1)

    return matrizCambio

def seleccionarMonedas(self, c, monedas, tabla):
    i, j = len(monedas), c
    seleccion = [0] * len(monedas)

    while j > 0:
        if i > 1 and tabla[i][j] == tabla[i - 1][j]:
            i -= 1
        else:
            seleccion[i - 1] += 1
            j -= monedas[i - 1]

    return seleccion

# Ejemplo de uso
cantidad = 11
monedas = [1, 5, 6, 9]

# Crear una instancia de Cambio con los valores del ejemplo
cambio_instance = Cambio(cantidad, monedas)

# Obtener el resultado de vectorSeleccion

```

```
vector_seleccion = cambio_instance.getVectorSeleccion()
print("Resultado:", vector_seleccion)
```

4.2.2. Resultado

Para el ejemplo dado con una cantidad de 11 y monedas de denominaciones [1, 5, 6, 9], el resultado del algoritmo es el siguiente:

- Utiliza 1 moneda de 5
- Utiliza 1 moneda de 6
- No utiliza monedas de 1
- No utiliza monedas de 9

Esto se refleja en el vector de selección: [0, 1, 1, 0].

4.2.3. Análisis de complejidad

Inicialización de Variables y Estructuras:

- `calcularMonedas(int cantidad, int[] monedas):`
 - Se inicializa una matriz de tamaño $(monedas.length + 1) \times (cantidad + 1)$.
 - Complejidad: $O(m \cdot n)$.
- Llenado de la Matriz con Valores Iniciales:
 - Complejidad: $O(m + n)$.
- Cálculo de la Matriz de Cambio:
 - Complejidad: $O(m \cdot n)$.
- Selección de Monedas:
 - Complejidad: $O(m + n)$.

■ Función de Utilidad `min(int a, int b)`:

- Función llamada cálculo de la matriz.
- Complejidad: $O(1)$ por cada llamada.

La complejidad total del algoritmo es: $O(m \cdot n)$. El algoritmo tiene una eficiencia polinómica en términos del número de denominaciones y la cantidad total, lo cual es razonable para muchos problemas prácticos de cambio de monedas. La matriz `matrizCambio` utiliza $O(m \cdot n)$ memoria, lo cual puede ser una limitación si m o n son muy grandes.

5. El Problema de la Mochila

El problema de la mochila es un clásico en el ámbito de la optimización combinatoria y la teoría de algoritmos. Se trata de seleccionar un subconjunto de artículos, cada uno con un peso y un valor, de manera que se maximice el valor total sin exceder la capacidad de peso permitida. Este problema es fundamental en campos como la logística, la gestión de recursos y la toma de decisiones en la ingeniería. La programación dinámica y las técnicas de aproximación ofrecen métodos eficientes para resolver este problema.

5.1. Ejemplo 1

5.2. Código

5.3. Resultado

6. El Problema de las Distancias Más Cortas

El problema de las distancias más cortas es uno de los temas más estudiados en la teoría de grafos por sus numerosas aplicaciones, incluyendo la ingeniería de redes, la logística, inteligencia artificial y la investigación operativa. El problema trata sobre un grafo que es una estructura compuesta por nodos y aristas. Cada arista es una conexión entre vértices y puede tener un peso asociado que representa la "distancia" o el "costo" de viajar entre dichos nodos. Se desea hallar la ruta de menor coste entre dos nodos.

6.1. Ejemplo 1 Algoritmo de Floyd-Warshall

6.2. Código

6.3. Resultado

7. Conclusiones

La programación dinámica se demuestra como una herramienta poderosa y versátil para resolver problemas de optimización clásicos como el cambio de monedas, el problema de la mochila y el problema de las distancias más cortas. A través de los ejemplos y análisis realizados, se evidencia su aplicabilidad en diferentes contextos, así como la importancia de entender la complejidad algorítmica para mejorar la eficiencia de las soluciones propuestas.