

“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



Programación Dinámica

Aplicaciones y Análisis de Complejidad Algorítmica

Integrantes del grupo N° 2

Campos García, Henry Leonardo

Escribas Alan, Daniel Leonardo

Meléndez Blas, Jhair Roussell

Morales Damasco, Cristian Ricardo

Ortiz Herrera, Fabrizio Peter

Docente

Luis Guerra Grados

23 / junio / 2024

Índice

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

```
1 def min_coins_change(total, denominations, counts):
2     # Convertimos las cantidades a centimos para manejarlas como
3     # enteros
4     total_cents = int(total * 100)
5     denominations_cents = [int(d * 100) for d in denominations]
6
7     # Inicializamos la tabla de DP con infinito para todas las
8     # cantidades
9     # excepto para 0 que necesita 0 monedas
10    dp = [float('inf')] * (total_cents + 1)
11    dp[0] = 0
12
13    # Para rastrear el uso de las monedas correctamente
```

```

12     used_coins = [[0 for _ in denominations] for _ in range(total_cents
13                     + 1)]
14
15     # Procesamos cada tipo de moneda
16     for i, coin in enumerate(denominations_cents):
17         for j in range(coin, total_cents + 1):
18             if dp[j - coin] != float('inf') and counts[i] > used_coins[
19                 j - coin][i]:
20                 if dp[j] > dp[j - coin] + 1:
21                     dp[j] = dp[j - coin] + 1
22                     used_coins[j] = used_coins[j - coin][:]
23                     used_coins[j][i] += 1
24
25     if dp[total_cents] == float('inf'):
26         return "No se puede dar el cambio exacto"
27     else:
28         return dp[total_cents], used_coins[total_cents]
29
30 # Denominaciones de las monedas disponibles y sus cantidades
31
32 denominations = [1.0, 0.50, 0.20, 0.10] # en soles
33 counts = [2, 6, 5, 10] # cantidad de monedas disponibles
34
35 # Cantidad total a devolver
36
37 total_change = 3.80 # en soles
38
39 min_coins, coin_usage = min_coins_change(total_change, denominations,
40                                         counts)
41
42 print("Número mínimo de monedas:", min_coins)
43 print("Uso de monedas:", coin_usage)

```

4.1.2. Resultado

Número mínimo de monedas: 7

Uso de monedas: [2, 3, 1, 1]

4.1.3. Análisis de complejidad

Análisis de Complejidad

Detalle de Complejidad

1. A: Convertir a centimos y crear lista de denominaciones en centimos.

El código correspondiente es:

```
1 total_cents = int(total * 100)
2 denominations_cents = [int(d * 100) for d in denominations]
```

- La conversión del total a centimos: `total_cents = int(total * 100)` es una operación constante $O(1)$. - La conversión de cada denominación a centimos: `denominations_cents = [int(d * 100) for d in denominations]` es una operación que se realiza n veces, donde n es el número de denominaciones.

Por lo tanto, la complejidad de esta parte es:

$$A = O(n)$$

Aquí, n es el número de denominaciones.

2. B: Inicializar la tabla de DP.

El código correspondiente es:

```
1 dp = [float('inf')] * (total_cents + 1)
2 dp[0] = 0
```

- Inicializar la tabla dp con infinito: `dp = [float('inf')] * (total_cents + 1)` implica recorrer $m + 1$ posiciones, donde m es el total en centimos. - Establecer `dp[0] = 0` es una operación constante $O(1)$.

Por lo tanto, la complejidad de esta parte es:

$$B = O(m + 1)$$

Aquí, m es el total en centimos.

3. C: Inicializar used_coins.

El código correspondiente es:

```
1 used_coins = [[0 for _ in denominations] for _ in range(total_cents +
    1)]
```

- Inicializar la tabla `used_coins` implica crear una lista de listas de ceros. La lista externa tiene $m + 1$ elementos y cada lista interna tiene n elementos.

Por lo tanto, la complejidad de esta parte es:

$$C = (m + 1) \cdot n$$

$$C = O((m + 1)n)$$

4. D: Procesar cada tipo de moneda.

El código correspondiente es:

```
1 for i, coin in enumerate(denominations_cents):
2     for j in range(coin, total_cents + 1):
3         if dp[j - coin] != float('inf') and counts[i] > used_coins[j -
4             coin][i]:
5             if dp[j] > dp[j - coin] + 1:
6                 dp[j] = dp[j - coin] + 1
7                 used_coins[j] = used_coins[j - coin][:]
8                 used_coins[j][i] += 1
```

- El bucle externo sobre las denominaciones: `for i, coin in enumerate(denominations_cents):` se ejecuta n veces. - El bucle interno sobre el rango del total en centimos: `for j in range(coin, total_cents + 1)`: se ejecuta aproximadamente m veces para cada denominación. - Comprobar y actualizar `dp` y `used_coins` es una operación constante $O(1)$.

Por lo tanto, la complejidad de esta parte es:

$$D = \sum_{i=1}^n \sum_{j=i}^m O(1)$$

$$D = \sum_{i=1}^n O(m)$$

$$D = O(n \cdot m)$$

5. E: Condición final.

El código correspondiente es:

```

1 if dp[total_cents] == float('inf'):
2     return "No se puede dar el cambio exacto"
3 else:
4     return dp[total_cents], used_coins[total_cents]

```

- Verificar si se puede dar el cambio exacto: `if dp[total_cents] == float('inf')` es una operación constante $O(1)$. - Devolver el resultado es una operación constante $O(1)$.

Por lo tanto, la complejidad de esta parte es:

$$E = O(1)$$

6. F: Impresión de resultados.

El código correspondiente es:

```

1 print("Número mínimo de monedas:", min_coins)
2 print("Uso de monedas:", coin_usage)

```

- Imprimir el número mínimo de monedas y el uso de monedas es una operación constante $O(1)$.

Por lo tanto, la complejidad de esta parte es:

$$F = O(1)$$

Complejidad Total

Fórmula

$$T(m) = A + B + C + D + E + F$$

Sumando todas las complejidades:

$$T(m) = O(n) + O(m+1) + O((m+1)n) + O(n \cdot m) + O(1) + O(1)$$

$$T(m) = O(n) + O(m) + O((m+1)n) + O(n \cdot m) + O(1)$$

$$T(m) = O(n) + O(m) + O(m \cdot n) + O(m \cdot n) + O(1)$$

$$T(m) = O(n) + O(m) + O(m \cdot n)$$

$$T(m) = O(m \cdot n)$$

Variables

- n = número de denominaciones de monedas.
- m = monto total en centimos.

4.2. Ejemplo 2

En primer lugar, debemos pensar cómo plantear el problema de forma incremental. Consideramos 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 :

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

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 llenando los casos base $t[i, 0] = 0$, para todo i con $0 \leq i \leq n$. A continuación, podemos llenar 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 llenará 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

```

1 def min(a, b):
2     return a if a < b else b
3
4 class Cambio:
5     def __init__(self, cantidad, monedas):
6         self.cantidad = cantidad
7         self.vectorMonedas = monedas
8         self.matrizCambio = self.calcularMonedas(cantidad, monedas)
9         self.vectorSeleccion = self.seleccionarMonedas(cantidad,
10             monedas, self.matrizCambio)

```

```

10
11     def getVectorSeleccion(self):
12         return self.vectorSeleccion
13
14     def calcularMonedas(self, cantidad, monedas):
15
16         matrizCambio = [[0 if j == 0 else 99 for j in range(cantidad +
17             1)] for i in range(len(monedas) + 1)]
18
19         for i in range(1, len(monedas) + 1):
20
21             for j in range(1, cantidad + 1):
22
23                 if j < monedas[i - 1]:
24
25                     matrizCambio[i][j] = matrizCambio[i - 1][j]
26
27                 else:
28
29                     matrizCambio[i][j] = min(matrizCambio[i - 1][j],
30
31                         matrizCambio[i][j - monedas[i - 1]] + 1)
32
33
34         return matrizCambio
35
36
37     def seleccionarMonedas(self, c, monedas, tabla):
38
39         i, j = len(monedas), c
40
41         seleccion = [0] * len(monedas)
42
43
44         while j > 0:
45
46             if i > 1 and tabla[i][j] == tabla[i - 1][j]:
47
48                 i -= 1
49
50             else:
51
52                 seleccion[i - 1] += 1
53
54                 j -= monedas[i - 1]
55
56
57         return seleccion
58
59
60     # Ejemplo de uso
61
62     cantidad = 11
63
64     monedas = [1, 5, 6, 9]

```

```

43 # Crear una instancia de Cambio con los valores del ejemplo
44 cambio_instance = Cambio(cantidad, monedas)
45
46 # Obtener el resultado de vectorSeleccion
47 vector_seleccion = cambio_instance.getVectorSeleccion()
48 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

Un naviero tiene un buque carguero con capacidad de hasta 500 toneladas. El carguero transporta contenedores de diferentes pesos para una determinada ruta. En la ruta actual el carguero puede transportar algunos de los siguientes contenedores:

Contenedor	1	2	3	4	5
Peso en (cientos de toneladas)	1	2	1	3	4
Valor (miles de dólares)	3	5	4	6	7

Cuadro 1: Datos ejemplo 1

El analista de la empresa del armador desea determinar el envío (conjunto de contenedores) que maximiza el valor de la carga transportada.

5.1.1. Código

```
1 def algoritmo_mochila(pesos, valores, capacidad):
2     n = len(pesos)
3     matriz = []
4     for i in range(n + 1):
5         fila = []
6         for j in range(capacidad + 1):
7             fila.append(0)
8         matriz.append(fila)
9
10    for i in range(1, n + 1):
```

```

11         for w in range(1, capacidad + 1):
12             if pesos[i - 1] <= w:
13                 matriz[i][w] = max(matriz[i - 1][w], valores[i - 1] +
14                                         matriz[i - 1][w - pesos[i - 1]])
15             else:
16                 matriz[i][w] = matriz[i - 1][w]
17
18     valor_maximo = matriz[n][capacidad]
19
20 return valor_maximo
21
22 pesos = [1, 2, 1, 3, 4] # Pesos en centenas de toneladas
23 valores = [3, 5, 4, 6, 7] # Valores en miles de dólares
24 capacidad = 500           # Capacidad del carguero en toneladas
25 capacidad_ajustada = capacidad // 100 # Ajustar a centenas
26 valor_maximo = algoritmo_mochila(pesos, valores, capacidad_ajustada)
27 print("El valor máximo de la carga transportada es:", valor_maximo, "
28       miles de dólares.")

```

5.1.2. Resultado

El valor máximo de la carga transportada es: 13 miles de dólares.

5.1.3. Análisis de complejidad

Análisis de Complejidad

Detalle de Complejidad

1. A: Inicialización de la matriz.

El código correspondiente es:

```

1 matriz = []
2 for i in range(n + 1): # O(n)
3     fila = []

```

```

4     for j in range(capacidad + 1):    # O(capacidad)
5         fila.append(0)    # O(1)
6     matriz.append(fila)    # O(1)

```

- Inicializar la matriz con ceros tiene una complejidad de $O(n \times capacidad)$.

2. B: Llenado de la matriz con los valores óptimos.

El código correspondiente es:

```

1 for i in range(1, n + 1):    # O(n)
2     for w in range(1, capacidad + 1):    # O(capacidad)
3         if pesos[i - 1] <= w:    # O(1)
4             matriz[i][w] = max(matriz[i - 1][w], valores[i - 1] +
5                         matriz[i - 1][w - pesos[i - 1]])    # O(1)
6         else:
7             matriz[i][w] = matriz[i - 1][w]    # O(1)

```

- El llenado de la matriz para calcular los valores óptimos tiene una complejidad de $O(n \times capacidad)$.

3. C: Determinación del valor máximo.

El código correspondiente es:

```

1 valor_maximo = matriz[n][capacidad]    # O(1)

```

- La determinación del valor máximo es una operación constante $O(1)$.

Complejidad Total

$$T(n, capacidad) = O(n \times capacidad)$$

Variables

- n = número de artículos.
- $capacidad$ = capacidad de la mochila en centenas de toneladas.

5.2. Ejemplo 2

Una empresa de transporte marítimo de mercancías posee un barco con una bodega cuya capacidad es de 250 cm^3 . Desea transportar cuatro bienes de los que se dispone su volumen y su valor monetario. En la siguiente tabla se muestra dicha información:

Bienes	Volumen (cm^3/Tm)	Ingresos (\$)
1	70	1250
2	50	900
3	60	1000
4	75	1200

Cuadro 2: Datos del problema de la mochila. Elaboración propia.

Se trata de determinar los bienes que se deben transportar en cada bodega de forma que el ingreso sea máximo.

5.2.1. Código

```
1 def mochila(capacidad, volumenes, ingresos, n):  
2     # Crear una matriz para almacenar los ingresos máximos posibles  
3     dp = [[0 for x in range(capacidad + 1)] for x in range(n + 1)]  
4  
5     # Llenar la matriz dp de manera ascendente  
6     for i in range(n + 1):  
7         for w in range(capacidad + 1):  
8             if i == 0 or w == 0:  
9                 dp[i][w] = 0  
10            elif volumenes[i - 1] <= w:  
11                dp[i][w] = max(ingresos[i - 1] + dp[i - 1][w -  
12                                volumenes[i - 1]], dp[i - 1][w])  
13            else:
```

```

13             dp[i][w] = dp[i - 1][w]
14
15     # Recuperar los elementos seleccionados
16     res = dp[n][capacidad]
17     w = capacidad
18     bienes_seleccionados = []
19
20     for i in range(n, 0, -1):
21         if res <= 0:
22             break
23         if res == dp[i - 1][w]:
24             continue
25         else:
26             bienes_seleccionados.append(i - 1)
27             res -= ingresos[i - 1]
28             w -= volumenes[i - 1]
29
30     # Crear una lista que indique si cada bien se lleva (1) o no (0)
31     seleccion = [0] * n
32     for i in bienes_seleccionados:
33         seleccion[i] = 1
34
35     return dp[n][capacidad], bienes_seleccionados, seleccion
36
37 # Datos del problema
38 volumenes = [70, 50, 60, 75]
39 ingresos = [1250, 900, 1000, 1200]
40 capacidad = 250
41 n = len(volumenes)
42
43 # Resolver el problema
44 ingreso_maximo, bienes_seleccionados, seleccion = mochila(capacidad,
45                 volumenes, ingresos, n)
46 print(f"El ingreso maximo es: ${ingreso_maximo}")

```

```
47 print("Bienes seleccionados (índices):", bienes_seleccionados)
48 print("Selección de bienes (1 = seleccionado, 0 = no seleccionado):")
49 for i in range(n):
50     print(f"Bien {i + 1}: {seleccion[i]}")
```

5.2.2. Resultado

```
El ingreso máximo es: $3450
Bienes seleccionados (índices): [3, 2, 0]
Selección de bienes (1 = seleccionado, 0 = no seleccionado):
Bien 1: 1
Bien 2: 0
Bien 3: 1
Bien 4: 1
```

Figura 1: Imagen del resultado del código Python.

5.2.3. Análisis de complejidad

```

def mochila(capacidad, volumenes, ingresos, n):
    dp = [[0 for x in range(capacidad + 1)] for x in range(n + 1)] A
    for i in range(n + 1):
        for w in range(capacidad + 1):
            if i == 0 or w == 0:
                dp[i][w] = 0 → O(1)
            elif Volumenes[i - 1] <= w:
                dp[i][w] = max(ingresos[i - 1] + dp[i - 1][w - Volumenes[i - 1]], dp[i - 1][w]) B
            else:
                dp[i][w] = dp[i - 1][w] → O(1)
    res = dp[n][capacidad] → O(1)
    w = Capacidad → O(1)
    bienes_seleccionados = [] → O(1)

    for i in range(n, 0, -1):
        if res < 0 → O(1)
        if res == dp[i - 1][w]:
            continue → O(1)
        else:
            bienes_seleccionados.append(i - 1) → O(1)
            res -= ingresos[i - 1] → O(1)
            w -= Volumenes[i - 1] → O(1) D
    Selección = [0]*n → O(n)
    for i in bienes_seleccionados:
        Selección[i] = 1 → O(1) E
    return dp[n][capacidad], bienes_seleccionados, selección

```

Figura 2: Análisis del código.

$$T(n) = A + C + cte_1 + cte_2 + cte_3 + D + E, \text{ donde } cte: \text{constante}$$

* Hallando A:

- Creación de matriz bidimensional $(n+1) \times (\text{capacidad}+1)$
- Se intera $n+1$ filas y capacidad+1 columnas

$$\rightarrow O(n \times \text{Capacidad})$$

* Hallando C

$$C = \sum_{i=0}^{n-1} B, \quad B = \sum_{i=0}^{\text{Capacidad}-1} (cte_1)$$

$$B = \text{capacidad} (cte_1) \rightarrow C = n \times \text{Capacidad} \times cte_1$$

$$\rightarrow O(n \times \text{Capacidad})$$

* Hallando D

$$D = \sum_{i=0}^{n-1} (cte_1) = n \times cte_1$$

$$\rightarrow O(n)$$

* Hallando E

$$E = n + \sum_{i=0}^{\text{biens seleccionados}-1} cte_1$$

► En el peor de los casos todas las filas no son seleccionadas, o sea hace n iteraciones.

$$E = n + n \rightarrow O(n)$$

$$T(n) = n \times \text{Capacidad} + n \times \text{Capacidad} + cte_1 + cte_2 + cte_3 + n + n$$

$$\boxed{T(n) = n \times \text{Capacidad}}$$

donde n : número de bienes, Capacidad: capacidad de la barda

Figura 3: Análisis del código.

Como se acaba de apreciar, el presente algoritmo tiene una complejidad aproximada de $O(n \times K)$ o, equivalentemente, $n \times \text{capacidad}$, lo cual supone eficiencia en función al dato de entrada de "capacidad"; mientras más sea esto, más complejo se vuelve.

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 Imaginemos la situación de un repartidor de agua que decide crear una red de entrega de suministros a domicilio con la intención de repartir sus pedidos en el menor tiempo posible. Inicialmente sólo opera únicamente en seis casas, que puedes cubrir gracias a la facilidad de desplazarse entre ellas. Sin embargo, desea encontrar las rutas más cortas a seguir entre su local y la de sus clientes para cumplir con las entregas de la manera más óptima. Por ello, realizamos el análisis para saber todas las posibles rutas que utilizaría el repartidor.

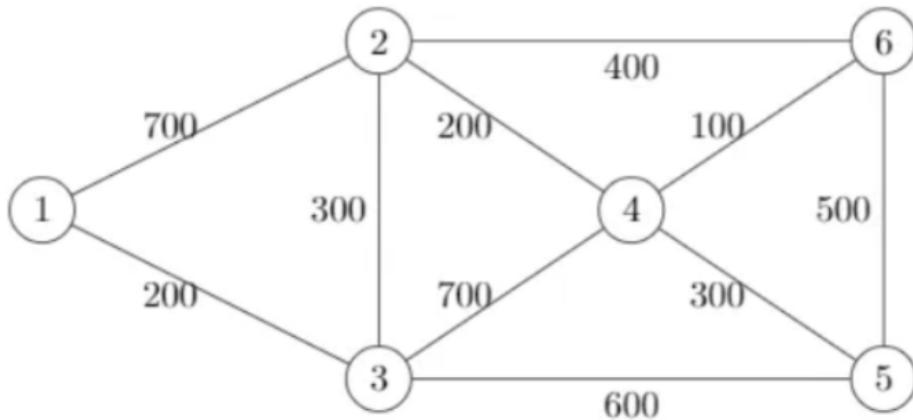


Figura 4: Representación gráfica del problema.

Uno de los algoritmos más utilizados en programación dinámica para este tipo de problemas es el algoritmo de Floyd Warshall, cuyo pseudocódigo es el siguiente

Algorithm Floyd-Warshall (A, n):

```

for  $k \leftarrow 1$  to  $n$  do
    for  $i \leftarrow 1$  to  $n$  do
        for  $j \leftarrow 1$  to  $n$  do
             $A[i][j] \leftarrow \min(A[i][j], A[i][k] + A[k][j])$ 
```

Figura 5: Pseudocódigo de Floyd Warshall.

6.1.1. Código

```

1     import sys
2
3     INF = sys.maxsize
4
5     #Implementacion
6
7     def Floyd_Warshall(graph):
8
9         n = len(graph)
10
11        dist = [[] for i in range(n)]
12
13        for i in range(n):
14            for j in range(n):
15                dist[i].append(graph[i][j]) #inicializar matriz de distancias
16
17        for k in range(n):
18            for i in range(n):
19                for j in range(n):
20                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
21
22        #Imprimir solucion
23
24        print('Distancia mas corta entre todo par de nodos:')
25
26        for i in range(n):
27            for j in range(n):
28                if dist[i][j] == INF:
29                    print("%7s" % ("INF"), end = ' ')
30                else:
31                    print("%7s" % (dist[i][j]), end = ' ')
32
33        print()
34
35        #Grafo
```

```

25     graph = [ [0, 700, 200, INF, INF, INF],
26             [700, 0, 300, 200, INF, 400],
27             [200, 300, 0, 700, 600, INF],
28             [INF, 200, 700, 0, 300, 100],
29             [INF, INF, 600, 300, 0, 500],
30             [INF, 400, INF, 100, 500, 0]
31         ]
32     Floyd_Warshall(graph)

```

6.1.2. Resultado

Distancia más corta entre todo par de nodos:

0	500	200	700	800	800
500	0	300	200	500	300
200	300	0	500	600	600
700	200	500	0	300	100
800	500	600	300	0	400
800	300	600	100	400	0

Figura 6: Resultado de la ejecución del código Python.

6.1.3. Análisis de complejidad

Análisis de Complejidad

Detalle de Complejidad

El código correspondiente es:

```

1  def Floyd_Warshall(graph):
2      n = len(graph) #... O(1)
3      dist = [[] for i in range(n)] #... O(n)
4      for i in range(n): #... Q
5          for j in range(n): #... P
6              dist[i].append(graph[i][j]) #... O(1)

```

```

7
8     for k in range(n): #...T
9         for i in range(n): #...S
10        for j in range(n): #...R
11            dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]) #...O(1)
12
13    #Imprimir solucion
14    print('Distancia mas corta entre todo par de nodos:') #...O(1)
15    for i in range(n): #...V
16        for j in range(n): #...U
17            if dist[i][j] == INF: #...O(1)
18                print("%7s" % ("INF"), end = ' ')
19            else:
20                print("%7s" % (dist[i][j]), end = ' ')
21    print() #...O(1)

```

Complejidad Total

Fórmula

La complejidad del código se determina de la siguiente manera

$$T(n) = 1 + n + n + Q + T + 1 + V$$

$$Q = \sum_{i=1}^n P \quad T = \sum_{k=1}^n S \quad V = \sum_{i=1}^n U$$

Entonces

$$T(n) = 1 + n + n + \sum_{i=1}^n P + \sum_{k=1}^n S + 1 + \sum_{i=1}^n U$$

$$P = \sum_{j=1}^n 1 \quad S = \sum_{i=1}^n R \quad U = \sum_{j=1}^n (1 + 1 + 1)$$

$$T(n) = 2 + 2n + \sum_{i=1}^n \sum_{j=1}^n 1 + \sum_{k=1}^n \sum_{i=1}^n R + \sum_{i=1}^n \sum_{j=1}^n (1 + 1 + 1)$$

$$T(n) = 2 + 2n + n^2 + \sum_{k=1}^n \sum_{i=1}^n \sum_{j=1}^n 1 + 3n^2$$

$$T(n) = 2 + 2n + n^2 + n^3 + 3n^2$$

$$T(n) = 2 + 2n + 4n^2 + n^3$$

$$T(n) = O(n^3)$$

Como es posible apreciar, el uso del algoritmo tiene un costo aproximado de $O(n^3)$, lo cual no representa muy favorable en términos de eficiencia.

Variables

- n = número de vértices.

6.1.4. Anexo Floy Warshall Análisis de complejidad

Análisis de
Complejidad
Algorítmica

```

INF = sys.maxsize -> O(1)
def Floyd_Warshall(graph):
    n = len(graph) -> O(n)
    dist = [[0] for i in range(n)] -> O(n)
    for i in range(n):
        for j in range(n):
            dist[i][j] = graph[i][j] -> O(1)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
    print("distancia") O(1)
    for i in range(n):
        for j in range(n):
            if dist[i][j] == INF:
                print("inf", end=" ")
            else:
                print(dist[i][j], end=" ")
        print() -> O(1)
    
```

Figura 7: Análisis del código.

$$\begin{aligned}
 T(n) &= 1 + m + m + Q + \Gamma + 1 - V \\
 T(n) &= 1 + m + m + \sum_{k=1}^m \sum_{i=1}^n \Delta + 1 + \sum_{i=1}^n V \\
 T(n) &= 2 + 2m + \sum_{k=1}^m \sum_{i=1}^n 1 + \sum_{k=1}^m \sum_{i=1}^n R + \sum_{i=1}^n \sum_{j=1}^m (1+1+1) \\
 T(n) &= 2 + 2m + m^2 + \sum_{k=1}^m \sum_{i=1}^n \sum_{j=1}^m 1 + 3n^2
 \end{aligned}$$

Figura 8: Análisis del código.

$$\begin{aligned}
 T(n) &= 2 + 2n + n^2 + n^3 + 8n^2 \\
 T(n) &= 2 + 2n + 4n^2 + n^3 \\
 T(n) &\in O(n^3)
 \end{aligned}$$

Figura 9: Análisis del código.

6.2. Ejemplo 2

Algoritmo de Bellman-Ford

Eres ingeniero de sistemas especializado en redes en una empresa en la que trabajas. Se calculan los tiempos que demoran para comunicarse entre distintos routers. Se te pide por tanto hallar el menor tiempo posible para comunicarte desde el Router Riobamba hacia todos los demás routers.

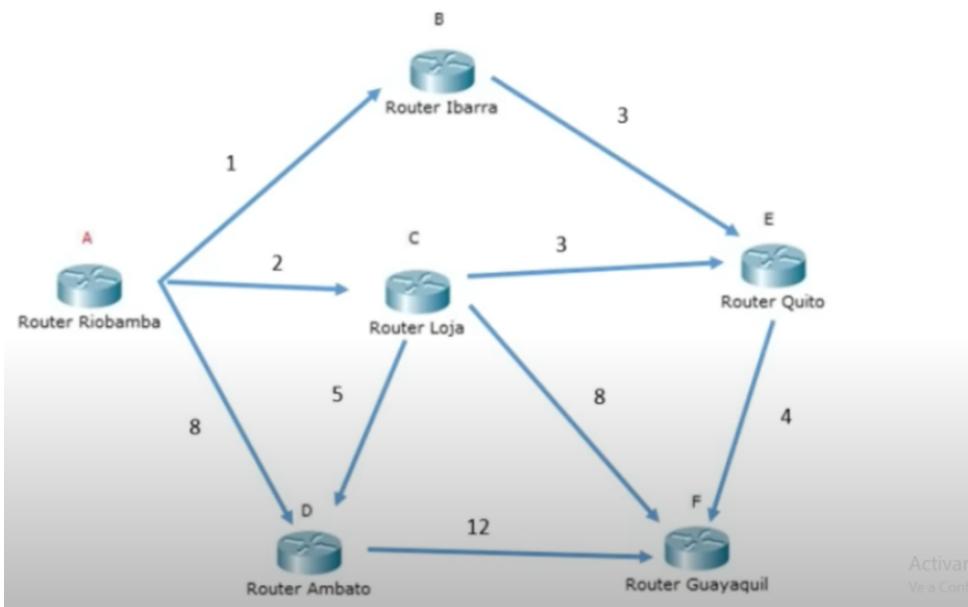


Figura 10: Representación gráfica del problema.

Uno de los algoritmos más utilizados en programación dinámica para este tipo de problemas es el algoritmo de Bellman-Ford, cuyo pseudocódigo es el siguiente:

```

BELLMAN-FORD( $G, w, s$ )
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  for  $i = 1$  to  $|G.V| - 1$ 
3    for each edge  $(u, v) \in G.E$ 
4      RELAX( $u, v, w$ )
5  for each edge  $(u, v) \in G.E$ 
6    if  $v.d > u.d + w(u, v)$ 
7      return FALSE
8  return TRUE

```

Figura 11: Pseudocódigo de Bellman-Ford.

6.2.1. Código

```

1   #Diccionario de aristas y sus pesos
2   w = {
3     (0,1) : 1,
4     (0,5) : 2,
5     (0,4) : 8,
6     (1,2) : 3,
7     (2,3) : 4,
8     (4,3) : 12,
9     (5,2) : 3,
10    (5,3) : 8,
11    (5,4) : 5
12  }
13
14  #numero de vertices
15  n = 6
16
17  def relax (u,v,w,d,p):
18    if d[v] > d[u] + w[u,v]:
19      d[v] = d[u] + w[u,v]
20      p[v] = u
21
22  def Bellman_Ford(w,n,s):
23
24    inf = 1e100

```

```

25     d = [0 for i in range(n)] ##Es la cota superior del peso del camino
26         mas corto de s a v, se inicializa en infinito
27
28     p = [0 for i in range(n)] ##Es el nodo previo en el camino del peso
29         mas corto de s a v
30
31
32     for vertex in range(n):
33         d[vertex] = inf #Asumimos infinito al principio, es decir, es
34             desconocido el limite
35         p[vertex] = 'null' #Aun no sabemos el nodo previo de vertex en el
36             camino mas corto
37
38     d[s] = 0 #O porque el peso de s a s es 0
39
40     for _ in range (n-1):
41         for u,v in w: #arista (u,v) : peso
42             relax(u,v,w,d,p) #actualizacion de las distancias
43
44     for (u,v) in w: #detectar ciclos negativos
45         if d[v] > d[u] + w[u,v]:
46             return False
47
48     print("Distancias mas cortas")
49     for i in range(n):
50         print(i, ":", d[i])
51
52     print("\nNodos previos")
53     for i in range(n):
54         print(i, ":", p[i])
55
56     return True
57
58     Bellman_Ford(w,n,0)

```

6.2.2. Resultado

```
Distancias mas cortas
0 : 0
1 : 1
2 : 4
3 : 8
4 : 7
5 : 2

Nodos previos
0 : null
1 : 0
2 : 1
3 : 2
4 : 5
5 : 0
True
```

Figura 12: Resultado de la ejecución del código Python.

6.2.3. Análisis de complejidad

Análisis de Complejidad

Detalle de Complejidad

El código correspondiente es:

```
1 def relax (u,v,w,d,p): #...A
2     if d[v] > d[u] + w[u,v]: #...O(1)
3         d[v] = d[u] + w[u,v] #...O(1)
4         p[v] = u #...O(1)
5
6 def Bellman_Ford(w,n,s):
7
8     inf = 1e100 #...O(1)
9     d = [0 for i in range(n)] #...O(n)
10    p = [0 for i in range(n)] #...O(n)
11
12    for vertex in range(n): #...P
13        d[vertex] = inf #...O(1)
```

```

14     p[vertex] = 'null' #...O(1)
15
16 d[s] = 0 #...O(1)
17
18 for _ in range(n-1): #...Q
19     for u,v in w: #...R
20         relax(u,v,w,d,p) #...A
21
22 for (u,v) in w: #...S
23     if d[v] > d[u] + w[u,v]: #...O(1)
24         return False #...O(1)
25
26 print("Distancias mas cortas") #...O(1)
27 for i in range(n): #...O(n)
28     print(i, ": ", d[i])
29
30 print("\nNodos previos") #...O(1)
31 for i in range(n): #...O(n)
32     print(i, ": ", p[i])
33
34 return True #...O(1)

```

Complejidad Total

Fórmula

La complejidad del código se determina de la siguiente manera

$$relax \implies A = O(1)$$

$$T(n) = 1 + n + n + P + 1 + Q + S + 1 + n + 1 + n + 1$$

$$T(n) = 5 + 4n + P + Q + S$$

$$P = \sum_{i=1}^n 2 \quad Q = \sum_{i=1}^{n-1} R \quad S = \sum_{\text{aristas en } W} 1$$

$$T(n) = 5 + 4n + \sum_{i=1}^n 2 + \sum_{i=1}^{n-1} R + \sum_{\text{aristas en } W} 1$$

$$R = \sum_{\text{aristas en } W} A$$

$$T(n) = 5 + 4n + (2n) + \sum_{i=1}^{n-1} \sum_{\text{aristas en } W} A + m$$

$$T(n) = 5 + 6n + \sum_{i=1}^{n-1} \sum_{\text{aristas en } W} 1 + m$$

$$T(n) = 5 + 6n + \sum_{i=1}^{n-1} m + m$$

$$T(n) = 5 + 6n + m(n - 1) + m$$

$$T(n) = mn + 6n + 5$$

$$T(n) = O(m * n)$$

El uso del algoritmo tiene un costo aproximado de $O(m*n)$. Si m fuese el numero maximo de aristas la complejidad pasaria a ser de $O(n^3)$

Variables

- n = número de vertices.

- m = número de aristas.

6.2.4. Anexo Bellman-Ford Análisis de complejidad

Análisis de Complejidad Algorítmico

$$W = \{ (0,1) : 1, (0,5) : 2, (0,4) : 3, (1,2) : 3, (2,3) : 4, (4,3) : 12, (5,2) : 3, (5,3) : 8, (5,4) : 5 \} \quad O(1)$$

$$m = 6 \rightarrow O(1)$$

$$\text{def relax}(u, v, w, d, p):$$

$$\begin{cases} \text{if } d[v] > d[u] + w[u, v]: \rightarrow O(1) \\ \quad d[v] = d[u] + w[u, v] \rightarrow O(1) \\ \quad p[v] = u \rightarrow O(1) \end{cases}$$

Figura 13: Análisis del código.

```

def BellmanFord(W, m, p):
    inf = 1e100 → O(1)
    d = [0 for i in range(m)] → O(1)
    p = [0 for i in range(m)] → O(1)
    for vertex in range(m):
        d[vertex] = inf → O(1)
        p[vertex] = 'null' → O(1)
    d[0] = 0 → O(1)
    for _ in range(m-1):
        R (for u, v in W:
            relax(u, v, w, d, p)) → Q
    for (u, v) in W:
        if d[v] > d[u] + w[u, v]: → O(1)
            return false → O(1)
    
```

Figura 14: Análisis del código.

```

print('') → O(1)
for i in range(m):
    print(i) } O(m)

print('') → O(1)
for i in range(n):
    print(i) } O(n)

return True → O(1)

```

Figura 15: Análisis del código.

$$\begin{aligned}
&\text{relax} \rightarrow A = O(1) \\
T(n) &= 1 + 1 + 1 + P + 1 + Q + S + 1 + n + 1 + n + 1 \\
T(n) &= 7 + 2n + P + Q + S \\
T(n) &= 7 + 2n + \sum_{i=1}^n 2 + \sum_{i=1}^{n-1} R + \sum_{a \in \mathcal{A}} 1 \\
T(n) &= 7 + 2n + (2n) + \sum_{i=1}^{n-1} \sum_{a \in \mathcal{A}} A + nm \\
T(n) &= 7 + 4n + \sum_{i=1}^{n-1} \sum_{a \in \mathcal{A}} 1 + nm \\
T(n) &= 7 + 4n + \sum_{i=1}^{n-1} nm + nm \\
T(n) &= 7 + 4n + (n-1)nm + nm \\
T(n) &= nm + 4n + 7 \\
T(n) &= O(mn)
\end{aligned}$$

Figura 16: Análisis del código.

```

def Bellman_Ford(W,m,p):
    inf = 1e100 -> O(1)
    d = [0 for i in range(n)] -> O(1)
    p = [None for i in range(n)] -> O(1)
    for vertex in range(n):
        d[vertex] = inf
        p[vertex] = 'null' O(1)
    d[s] = 0 O(1)
    for _ in range(n-1):
        R = {for u,v in W:
              relax(u,v,W,d,p)} Q
        for (u,v) in W:
            if d[v] > d[u] + W[u,v]: } S
            return false -> O(1)

```

Figura 17: Analisis del codigo.

Puede observar que la complejidad es $O(m*n)$, donde m es el numero de aristas y n de nodos. Si m fuera el numero máximo de aristas, la complejidad se define como $O(n^3)$

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.

Referencia

- [1] jcsis. (2013, 22 de diciembre). Programación dinámica: algoritmo cambio monedas con tipo de monedas a devolver. *jcsis Blog*. Recuperado de <https://jcsis.wordpress.com/2013/12/22/programacion-dinamica-algoritmo-cambio-monedas/>
- [2] The Knapsack Problem. *pdfcoffee.com*. (Fecha y autores no disponibles). Recuperado de <https://pdfcoffee.com/knapsack-problem-10-pdf-free.html>
- [3] Mendoza, I., & Ruedas, L. (2021, 31 de enero). Algoritmo de Floyd-Warshall: Análisis e implementación. *Medium*. Recuperado el 6 de mayo de 2024, de <https://medium.com/algoritmo-floyd-warshall/algoritmo-de-floyd-warshall-e1fd1a900d8>
- [4] Kleinberg, J., & Tardos, E. (2013). *Algorithm Design* (1^a ed.). Pearson. Recuperado de <https://www.perlego.com/book/810837/algorithm-design-pdf> (Trabajo original publicado en 2013)
- [5] Cormen, T. H., Leiserson, C. E., & Rivest, R. L. (1990). *Introduction to algorithms*. Cambridge, MA: MIT Press.
- [6] Kevin, O. [Kevin Ortega]. (2017, 2 de febrero). Algoritmo Bellman-Ford: Definición y ejemplo del algoritmo Bellman-Ford empleado por el protocolo de enrutamiento de rutas RIP. *Ing Raúl Lozada Yanez* [Video]. YouTube. Recuperado el 11 de mayo de 2024, de https://www.youtube.com/watch?v=CIz_8J6IuM0