ALGORITMOS PROGRAMACIÓN DINÁMICA Y DEMÁS

Análisis y Diseño de Algoritmos

3CM1

Jorge Luis Vargas Hernández

11 de diciembre de 2023

0.1. Introducción

En este trabajo exploraremos diversos problemas resueltos mediante distintos métodos de programación, los cuales hemos estudiado a lo largo del curso de análisis y diseño de algoritmos. Analizaremos el tiempo de ejecución para determinar cuál método es más eficiente y consume menos recursos.

0.2. Fibonacci

0.2.1. Fibonacci Iterativo

```
n = 9

if n == 0:
    print("0")
elif n == 1 or n == 2:
    print("1")
else:
    n1, n2 = 0, 1

    for i in range(2, n + 1):
        pos = n1 + n2
        n1, n2 = n2, pos

    print(pos)
```

En este código, se determina el noveno término de la secuencia de Fibonacci de manera iterativa. Se inicia verificando casos base donde, si n es igual a 0, imprime "0", y si es 1 o 2, imprime "1". En caso contrario, se emplea un bucle para calcular los términos sucesivos de la secuencia sumando los dos términos anteriores en cada iteración. Los valores iniciales (n1 y n2) se actualizan en cada paso del bucle. Finalmente, se imprime el resultado, que representa el valor del noveno término de la secuencia de Fibonacci. Este enfoque iterativo evita la necesidad de recursión y calcula eficientemente el término deseado.

0.2.2. Fibonacci Recursivo

```
def fibonacci_recursivo(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci_recursivo(n - 1) + fibonacci_recursivo(n - 2)

n = 9
resultado = fibonacci_recursivo(n)
print(resultado)
```

Este código resuelve el problema de obtener el n-ésimo término de la secuencia de Fibonacci mediante una función recursiva. Si la entrada n es menor o igual a 0, la función devuelve 0. En los casos donde n es 1 o 2, retorna 1. Para valores mayores que 2, la función calcula el término de Fibonacci sumando los resultados recursivos de los términos anteriores (n-1 y n-2). La recursión continúa hasta alcanzar los casos base, y finalmente, el resultado se imprime para el valor específico de n. Es importante tener en cuenta que el enfoque recursivo puede ser menos eficiente para valores grandes de n debido a la repetición de cálculos.

0.2.3. Fibonacci Dinámico

```
def fibonacci_dinamico(n):
    dp = [-1] * (n + 1)
    dp[0] = 0
    if n > 0:
        dp[1] = 1

    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]

# Ejemplo de uso
n = 9
resultado = fibonacci_dinamico(n)
print(resultado)
```

Este código implementa la generación del n-ésimo término de la secuencia de Fibonacci utilizando programación dinámica. Se inicializa una lista dp para almacenar los valores intermedios. Luego, se utiliza un bucle para calcular y almacenar los términos de la secuencia de manera iterativa, aprovechando los resultados previamente calculados. La función fibonacci dinamico devuelve el valor del n-ésimo término. Este enfoque mejora la eficiencia en comparación con la versión recursiva, ya que evita recalcular los mismos términos varias veces y optimiza l uso de la memoria.

0.3. Cambio de Monedas

0.3.1. Cambio de Monedas Voraz

```
def cambio_moneda(monedas, cantidad):
    monedas.sort(reverse=True)
    cambio = []

    for moneda in monedas:
        while cantidad >= moneda:
```

```
            cantidad -= moneda
            cambio.append(moneda)

    return cambio


monedas_disponibles = [500, 200, 100, 50, 20, 10, 5, 2, 1]
cantidad_a_cambiar = 355

cambio_resultante = cambio_moneda(monedas_disponibles, cantidad_a_cambiar)
print("Cambio a dar:", cambio_resultante)
```

El código implementa un algoritmo de cambio de monedas utilizando la estrategia voraz. Dada una cantidad a cambiar y un conjunto de denominaciones de monedas, el algoritmo selecciona de manera iterativa la moneda de mayor valor posible que aún no haya superado la cantidad restante. Este proceso se repite hasta que se alcanza la cantidad deseada. La eficacia del algoritmo se basa en la elección voraz de las monedas de mayor denominación en cada paso, buscando optimizar la cantidad de monedas utilizadas. Es importante destacar que, aunque este enfoque suele funcionar bien en algunos casos, puede no garantizar siempre la solución óptima en todos los contextos, ya que no considera todas las combinaciones posibles.

0.3.2. Cambio de Monedas Dinámico

```
def cambio_moneda_dinamica(monedas, cantidad):
    mon = [float('inf')] * (cantidad + 1)
    mon[0] = 0

    for i in range(1, cantidad + 1):
        for moneda in monedas:
            if i - moneda >= 0:
                mon[i] = min(mon[i], mon[i - moneda] + 1)

    cambio = []
    i = cantidad
    while i > 0 and mon[i] != float('inf'):
        for moneda in monedas:
            if i - moneda >= 0 and mon[i] == mon[i - moneda] + 1:
                cambio.append(moneda)
                i -= moneda
                break

    return cambio

monedas_disponibles = [500, 200, 100, 50, 20, 10, 5, 2, 1]
cantidad_a_cambiar = 355

cambio_resultante = cambio_moneda_dinamica(monedas_disponibles, cantidad_a_cambiar)
```

```
print("Cambio a dar:", cambio_resultante)
```

0.4. Mochila Binaria Dinámica

```
def mochila_binaria(valores, pesos, capacidad):
    n = len(valores)
    dp = [[0] * (capacidad + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(capacidad + 1):
            if pesos[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], valores[i - 1] + dp[i - 1][w - pesos[i -
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][capacidad]

# Ejemplo de uso
valores = [20, 50, 80]
pesos = [10, 20, 30]
capacidad_mochila = 220

resultado = mochila_binaria(valores, pesos, capacidad_mochila)
print("El valor máximo que se puede llevar en la mochila es:", resultado)
```

El código implementa un algoritmo de resolución del problema de la mochila binaria mediante programación dinámica. Dada una lista de valores y pesos de elementos, así como la capacidad máxima de la mochila, el algoritmo determina la máxima suma de valores que se pueden colocar en la mochila sin exceder su capacidad. Utiliza una matriz ‘dp‘ para almacenar las soluciones parciales a subproblemas, donde ‘dp[i][w]‘ representa la máxima suma de valores considerando los primeros ‘i‘ elementos y una capacidad máxima de ‘w‘. El algoritmo utiliza un enfoque iterativo para llenar dinámicamente la matriz ‘dp‘ y calcular la solución óptima. Al finalizar, la función devuelve el valor máximo que se puede llevar en la mochila. Este enfoque de programación dinámica asegura la obtención de la solución óptima al problema de la mochila binaria, mejorando la eficiencia en comparación con métodos de fuerza bruta.

0.5. Tiempos de Ejecución y conclusión

El tiempo de ejecución puede variar dependiendo de las especificaciones de la máquina en la que se ejecute el código. En general, los algoritmos iterativos y de programación dinámica tienden a tener tiempos de ejecución más rápidos que los algoritmos recursivos. Sin embargo, también es importante señalar que la eficiencia de los algoritmos puede depender del tamaño de los datos de entrada.

En el script proporcionado, la sección que compara los tiempos de ejecución para el algoritmo de Fibonacci incluye tres enfoques: iterativo, recursivo y dinámico. El resultado puede variar, pero en términos generales, el enfoque dinámico (fibonacci$_{d}inamico)tiendeasermsficient$

Para obtener una respuesta específica sobre cuál tuvo el menor tiempo de ejecución, puedes ejecutar el script en tu entorno local y observar los resultados en tu máquina. Ten en cuenta que la eficiencia de los algoritmos puede diferir en distintos entornos de ejecución.

Por motivos de errores en LaTex no fue posible poner las capturas de las graficas realizadas en el código. Para su comprobación hay que checar el código dispoble en GitHub que implementa todos los códigos vistos junto con con sus tiempos de ejecución y su grafica.

0.6. Referencias

Bertsekas, D.P., "Dynamic Programming; Deterministic and Stochastic Models", Academic Press, 1987.

Dreyfus, S.E. y Law, A.M., "The Art and Theory of Dynamic Programming", Academic Press, 1977.