10. Programación Dinámica (Dynamic Programming - DP)

La **Programación Dinámica (DP)** es una técnica algorítmica utilizada para resolver problemas complejos dividiéndolos en subproblemas más simples, resolviendo cada subproblema solo una vez y almacenando sus soluciones. Cuando se encuentra el mismo subproblema, se reutiliza la solución almacenada, evitando cálculos redundantes. Este enfoque es particularmente útil para problemas que exhiben propiedades específicas, llevando a mejoras significativas en la eficiencia.

—

0.1. Características Clave de la Programación Dinámica

La Programación Dinámica es aplicable a problemas que cumplen dos propiedades fundamentales:

1. **Subproblemas Superpuestos (Overlapping Subproblems):** Un problema tiene subproblemas superpuestos si el mismo subproblema se resuelve varias veces. Por ejemplo, en la serie de Fibonacci recursiva ingenua ($F(n) = F(n-1) + F(n-2)$), $F(n-2)$ se calcula tanto por $F(n)$ como por $F(n-1)$, y así sucesivamente. DP resuelve cada subproblema una vez y almacena la solución en una tabla (conocida como *memoización* o *tabulación*) para futuras referencias.
2. **Subestructura Óptima (Optimal Substructure):** Un problema tiene subestructura óptima si una solución óptima del problema global puede construirse a partir de soluciones óptimas de sus subproblemas. Por ejemplo, la ruta más corta entre dos nodos en un grafo contiene subrutas más cortas entre los nodos intermedios. Esto significa que podemos combinar soluciones de subproblemas para obtener la solución del problema principal.

Si un problema no cumple ambas propiedades, la Programación Dinámica no es el enfoque más adecuado.

—

0.2. Enfoques de la Programación Dinámica

Existen dos maneras principales de implementar la Programación Dinámica:

1. **Enfoque Top-Down (Memoización):**
   - Es una aproximación “de arriba hacia abajo” que utiliza la recursión.
   - Se resuelve el problema principal recursivamente. Antes de calcular la solución para un subproblema, se verifica si ya ha sido calculada y almacenada en una tabla (generalmente un array o un hash map). Si

ya está, se devuelve el valor almacenado. Si no, se calcula, se almacena y luego se devuelve.
   - Combina la recursión con el almacenamiento de resultados para evitar recálculos.
   - Fácil de escribir para problemas que se definen recursivamente.

2. **Enfoque Bottom-Up (Tabulación):**
   - Es una aproximación “de abajo hacia arriba” que utiliza un enfoque iterativo.
   - Se resuelven los subproblemas más pequeños primero, y sus soluciones se almacenan en una tabla. Luego, se usan estas soluciones para resolver subproblemas más grandes, hasta que se resuelve el problema original.
   - Evita la sobrecarga de la pila de llamadas de la recursión.
   - A menudo es más eficiente en tiempo constante y espacio debido a la ausencia de recursión.

—

0.3. Ejemplos Clásicos de Programación Dinámica en C++

Veremos cómo aplicar la Programación Dinámica a problemas conocidos.

Serie de Fibonacci (Optimizado con DP)

Calcular el n-ésimo término de la serie de Fibonacci de manera eficiente. La versión recursiva ingenua tiene subproblemas superpuestos y una complejidad exponencial.

Listing 1: Fibonacci con Programación Dinámica (Bottom-Up - Tabulación)

```
#include <iostream>
#include <vector>

using namespace std; // Incluyendo namespace std como se solicit

// Funci n para calcular Fibonacci usando Programaci n Din mica (Bottom-Up)
long long fibonacci_dp_bottom_up(int n) {
    if (n <= 1) {
        return n;
    }

    // Crear una tabla (vector) para almacenar los resultados de los subproblemas
    vector<long long> dp(n + 1);
```

```

    // Inicializar los casos base
    dp[0] = 0;
    dp[1] = 1;

    // Llenar la tabla de abajo hacia arriba
    for (int i = 2; i <= n; ++i) {
        dp[i] = dp[i-1] + dp[i-2];
    }

    return dp[n];
}

// Funci n para calcular Fibonacci usando Programaci n Din mica (Top-Down - Memoizaci n)
vector<long long> memo; // Vector para memoizaci n , inicializado con -1
long long fibonacci_dp_top_down(int n) {
    if (n <= 1) {
        return n;
    }
    // Si ya hemos calculado este subproblema, devolver el resultado almacenado
    if (memo[n] != -1) {
        return memo[n];
    }
    // Si no, calcularlo, almacenarlo y luego devolverlo
    memo[n] = fibonacci_dp_top_down(n-1) + fibonacci_dp_top_down(n-2);
    return memo[n];
}

int main() {
    int n_fib = 10;

    // Ejemplo Bottom-Up
    cout << "Fibonacci (Bottom-Up) para n=" << n_fib << ": " << fibonacci_dp_bottom_up(n_fib) << endl; // Salida: 55

    // Ejemplo Top-Down
    memo.assign(n_fib + 1, -1); // Reiniciar memoizaci n
    cout << "Fibonacci (Top-Down) para n=" << n_fib << ": " << fibonacci_dp_top_down(n_fib) << endl; // Salida: 55

    n_fib = 40; // Para un n mero m s grande, la diferencia de eficiencia es notable
    cout << "Fibonacci (Bottom-Up) para n=" << n_fib << ": " << fibonacci_dp_bottom_up(n_fib) << endl;
```

```cpp
    memo.assign(n_fib + 1, -1);
    cout << "Fibonacci (Top-Down) para n=" << n_fib << ": " << fibonacci_dp_top_down(n_fib) << endl;

    return 0;
}
```

Problema de la Mochila (0/1 Knapsack)

Dado un conjunto de ítems, cada uno con un peso y un valor, determinar la combinación de ítems que pueden ser colocados en una mochila con una capacidad máxima dada, de modo que el valor total sea el máximo posible. Cada ítem solo puede ser incluido una vez (0 o 1).

Listing 2: Problema de la Mochila (0/1 Knapsack) con DP (Bottom-Up)

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // Para std::max

using namespace std; // Incluyendo namespace std como se solicit

// Funci n para resolver el problema de la mochila 0/1 usando DP (Bottom-Up)
int knapsack(int capacity, const vector<int>& weights, const vector<int>& values) {
    int n = weights.size();
    // dp[i][j] almacenar  el valor m ximo que se puede obtener con los primeros 'i'  tems
    // y una capacidad de mochila de 'j'
    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

    // Llenar la tabla dp
    for (int i = 1; i <= n; ++i) { // Iterar sobre los  tems
        for (int w = 1; w <= capacity; ++w) { // Iterar sobre las capacidades
            // Si el peso del  tem  actual es menor o igual a la capacidad actual
            if (weights[i-1] <= w) {
                // Opci n 1: No incluir el  tem  actual
                int val_without_item = dp[i-1][w];
                // Opci n 2: Incluir el  tem  actual + valor de los  tems anteriores con capacidad reducida
                int val_with_item = values[i-1] + dp[i-1][w - weights[i-1]];
```

```cpp

                    dp[i][w] = max(val_without_item, val_with_item);
                } else {
                    // Si el  tem  actual es demasiado pesado, no podemos incluirlo
                    dp[i][w] = dp[i-1][w];
                }
            }
        }
        // El valor m ximo estar  en dp[n][capacity]
        return dp[n][capacity];
    }

    int main() {
        vector<int> values = {60, 100, 120}; // Valores de los  tems
        vector<int> weights = {10, 20, 30}; // Pesos de los  tems
        int capacity = 50; // Capacidad m xima de la mochila

        cout << "Valores de los  tems : "; for(int v : values) cout << v << " "; cout << endl;
        cout << "Pesos de los  tems : "; for(int w : weights) cout << w << " "; cout << endl;
        cout << "Capacidad de la mochila: " << capacity << endl;

        int max_value = knapsack(capacity, weights, values);
        cout << "El valor m ximo que se puede obtener es: " << max_value << endl; // Salida: 220 (100 + 120)

        // Ejemplo de mochila con otros  tems
        vector<int> values2 = {10, 40, 30, 50};
        vector<int> weights2 = {5, 4, 6, 3};
        int capacity2 = 10;
        cout << "\nNuevo ejemplo:" << endl;
        cout << "Valores: "; for(int v : values2) cout << v << " "; cout << endl;
        cout << "Pesos: "; for(int w : weights2) cout << w << " "; cout << endl;
        cout << "Capacidad: " << capacity2 << endl;
        cout << "Valor m ximo: " << knapsack(capacity2, weights2, values2) << endl; // Salida: 90 (40 + 50)

        return 0;
    }
```

0.4. Análisis de Eficiencia de la Programación Dinámica

La Programación Dinámica mejora drásticamente la eficiencia de los algoritmos al eliminar cálculos redundantes.

- **Complejidad Temporal:**
  - Generalmente, la complejidad es el producto del número de estados (subproblemas) por el costo de calcular cada estado.
  - **Fibonacci (con DP):** $\mathcal{O}(N)$. Cada término se calcula una vez.
  - **0/1 Knapsack:** $\mathcal{O}(N \cdot W)$, donde N es el número de ítems y W es la capacidad máxima de la mochila. Hay $N \times W$ estados en la tabla DP, y cada uno se calcula en tiempo constante.
- **Complejidad Espacial:**
  - Depende del tamaño de la tabla de memoización o tabulación.
  - **Fibonacci (con DP):** $\mathcal{O}(N)$ para la tabla DP. Se puede reducir a $\mathcal{O}(1)$ si solo se necesitan los dos valores anteriores (optimización de espacio).
  - **0/1 Knapsack:** $\mathcal{O}(N \cdot W)$ para la tabla DP. En algunas variaciones, puede reducirse a $\mathcal{O}(W)$ optimizando el espacio.

La Programación Dinámica transforma problemas combinatorios de complejidad exponencial o polinómica alta a complejidades polinómicas mucho más manejables.

—

0.5. Aplicaciones Comunes de la Programación Dinámica

La Programación Dinámica se aplica a una vasta gama de problemas en informática, ciencia de datos, bioinformática, y optimización:

- **Problemas de Ruta más Corta:** En grafos (ej., Algoritmo de Floyd-Warshall para todos los pares de caminos más cortos, Algoritmo de Bellman-Ford para caminos con pesos negativos).
- **Edición de Cadenas (Levenshtein Distance):** Encontrar el número mínimo de operaciones (inserción, borrado, sustitución) para transformar una cadena en otra.
- **Problema de la Subsecuencia Común Más Larga (LCS):** Encontrar la subsecuencia común más larga entre dos cadenas.
- **Multiplicación de Cadenas de Matrices:** Determinar el orden óptimo de multiplicación de matrices para minimizar el número de operaciones.
- **Game Theory (Teoría de Juegos):** Resolver ciertos juegos.

- **Problemas de Optimización Combinatoria:** Como el problema de las monedas o el problema de la mochila (variaciones).

La Programación Dinámica es una técnica esencial para la resolución de problemas de optimización y conteo, transformando la ineficiencia de la fuerza bruta y la recursión ingenua en soluciones eficientes y escalables.