Programación dinámica

La **Programación Dinámica** es una técnica de programación que consiste en resolver los subproblemas una sola vez, guardando sus soluciones en una estructura dinámica de datos para su futura utilización.

Esta técnica es útil aplicarla cuando la solución óptima a un problema puede ser definida en función de soluciones óptimas a subproblemas de tamaño menor, y cuando hay solapamiento entre estos. Si planteásemos la solución recursiva del problema, un mismo problema se resolvería más de una vez. En estos casos no resulta eficiente la solución por la repetición de cálculos que conlleva y la programación dinámica nos puede ofrecer otra solución más aceptable al problema descomponiéndolo en subproblemas solapados y así se va construyendo la solución con las soluciones de esos subproblemas.

La programación dinámica la aplicamos entonces, cuando la subdivisión de un problema conduce a una enorme cantidad de problemas, cuyas soluciones parciales se solapan y grupos de problemas de muy distinta complejidad.

Entonces, la programación dinámica se basa en la **memorización.** Esto es almacenar las soluciones de los subproblemas en alguna estructura de datos para reutilizarlas posteriormente. De esa forma, se consigue un algoritmo más eficiente que el que resuelve el mismo subproblema una y otra vez.

Los pasos a seguir en la memorización de datos son:

1. Cuando se calcula una solución, ésta se almacena.
2. Antes de realizar una llamada recursiva para un subproblema, se comprueba si la solución ha sido obtenida previamente:
   * + - 1. Si no ha sido obtenida, se hace la llamada recursiva y, antes de devolver la solución, ésta se almacena.
         2. Si ya ha sido previamente calculada, se recupera la solución directamente (no hace falta calcularla).

Usualmente, se utiliza una matriz que se rellena conforme las soluciones a los conforme las soluciones a los subproblemas son calculados.

Para que un problema pueda ser abordado por esta técnica debe cumplir dos condiciones:

1. La solución al problema ha de ser alcanzada a través de una secuencia de decisiones, una en cada etapa.
2. Dicha secuencia de decisiones ha de cumplir el principio de óptimo.

El diseño de un algoritmo de Programación Dinámica consta de los siguientes pasos:

1. Planteamiento de la solución como una sucesión de decisiones y verificación de que ésta cumple el principio de óptimo.
2. Definición recursiva de la solución.
3. Cálculo del valor de la solución óptima mediante una tabla en donde se almacenan soluciones a problemas parciales para reutilizar los cálculos.
4. Construcción de la solución óptima haciendo uso de la información contenida en la tabla anterior.

**OPTIMICAZIÓN DINÁMICA Y PRINCIPIO DE OPTIMALIDAD DE BELLMAN**

La **optimización dinámica**, como su nombre lo indica, estudia la optimización de sistemas dinámicos, es decir, sistemas que evolucionan en el tiempo. De esta manera, se trata de guiar o controlar el sistema de manera óptima a lo largo de un tiempo determinado de acuerdo a un objetivo previamente establecido.

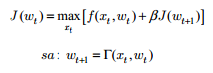
La optimización dinámica se diferencia de la estática en que esta última consiste en determinar el o los valores de una o varias variables que hacían máxima una función objetivo en un determinado instante del tiempo. Por lo contrario, en la dinámica se busca resolver un conjunto finito o infinito de problemas pero con la diferencia que estos no pueden resolverse de manera independiente.

Entonces, la optimización dinámica puede resolverse por medio de la **técnica de programación dinámica** que permite abordar problemas en tiempo continuo.

Esta técnica de programación se basa en el **Principio de Optimalidad de Bellman** que plantea:

*“Una política optima tiene la propiedad de que, cualesquiera sean el estado y las decisiones iniciales tomadas (es decir, el control), las restantes decisiones deben constituir una política optima con independencia del estado resultante de la primer decisión”.*

Dicho principio, se expresó en términos matemáticos a través de la **Ecuación de Bellman** que consiste en una relación recursiva fundamental para la programación dinámica.



Dicha ecuación planteada sostiene que el valor máximo que se puede obtener desde el estado wt es el valor máximo desde el estado siguiente más el valor máximo de la función una vez optimizada con respecto a un periodo t.

Por lo tanto, la Ecuación de Bellman permite transformar un problema de múltiples periodos en múltiples problemas de un solo período aprovechando el concepto de recursividad. Es decir, que si uno observa una subsolución de la solución óptima, ésta debe ser solución del subproblema asociado a esa subsolución.

En conclusión, donde tiene mayor aplicación la Programación Dinámica es justamente en la resolución de problemas de optimización. En este tipo de problemas se pueden presentar distintas soluciones, cada una con un valor, y lo que se desea es encontrar la solución de valor óptimo (máximo o mínimo).

Entonces, podemos decir que para aplicar programación dinámica:

1. Se comprueba que se cumple el principio de optimalidad de Bellman, para lo que hay que, encontrar la “estructura” de la solución.
2. Se define recursivamente la solución óptima del problema (en función de los valores de las soluciones para subproblemas de menor tamaño).
3. Se calcula el valor de la solución óptima utilizando un enfoque ascendente: Se determina el conjunto de subproblemas que hay que resolver (el tamaño de la tabla) y se identifican los subproblemas con una solución trivial (casos base). Después se van calculando los valores de soluciones más complejas a partir de los valores previamente calculados.
4. Se determina la solución óptima a partir de los datos almacenados en la tabla.

**EJEMPLO BASE: CÁLCULO DE LOS NÚMEROS DE FIBONACCI**

Un primer ejemplo donde podemos ver reflejada toda esta problemática es el del cálculo de los términos de la sucesión de números de Fibonacci.

Esta sucesión podemos expresarla recursivamente en términos matemáticos de la siguiente manera:

Fib(n) = 1 si n= 0, 1

Fib(n−1)+Fib(n−2) si n>1

La forma más natural de calcular los términos de esta sucesión es mediante un programa recursivo. Pero el algoritmo resultante es muy poco eficiente por su tiempo de ejecución de orden exponencial. (Como vimos en el ejemplo en clase)

Como podemos observar, la falta de eficiencia del algoritmo se debe a que se producen llamadas recursivas repetidas para calcular valores de la sucesión, que habiéndose calculado previamente, no se conserva el resultado y por tanto es necesario volver a calcularlo cada vez que se lo pide.

Para este problema es posible diseñar un algoritmo que en tiempo lineal lo resuelva mediante la construcción de una estructura que permita ir almacenando los cálculos ya realizados para poder reutilizarlos. Como únicamente son necesarios los dos últimos valores calculados para determinar cada término, solo necesitamos dos variables para almacenar los dos últimos términos.

Esta es la clave de los algoritmos de Programación Dinámica: el uso de estructuras (vectores o tablas) para eliminar la repetición de los cálculos.

En general, los algoritmos obtenidos mediante la aplicación de esta técnica consiguen tener complejidades (espacio y tiempo) bastante razonables, pero debemos evitar que el tratar de obtener una complejidad temporal de orden polinómico conduzca a una complejidad espacial demasiado elevada.

Por lo tanto, el problema de Fibonacci desarrollado sin programación dinámica sería:

public class Fibonacci

{

public static void main(String[] args)

{

System.out.println(Fibo(40));

}

public static int Fibo(int n)

{

if(n==1 || n==2)

return 1;

return Fibo(n-1)+Fibo(n-2);

}

}

Mientras que el mismo problema desarrollado con la técnica de programación dinámica seria:

public class FibonacciPD

{

private static int[] tabla = new int[40];

public static void main(String[] args)

{

for(int i = 0; i<40;i++)

{

tabla[i]=0;

}

System.out.println(Fibo(40));

}

public static int Fibo(int n)

{

if(tabla[n-1]!=0)

return tabla[n-1];

if(n==1 || n==2)

{

tabla[n-1] = 1;

return tabla[n-1];

}

tabla[n-1] = Fibo(n-1)+Fibo(n-2);

return tabla[n-1];

}

}

En esta segunda solución, se aplicó la programación dinámica al incorporar el array donde se van guardando los resultados parciales. Además, se empleó un enfoque ascendente (bottom--up) de la técnica de programación que consiste en calcular primero las soluciones óptimas para problemas de tamaño pequeño y, luego utilizando dichas soluciones, se encuentran las soluciones de los problemas de mayor tamaño.

**OTRO EJEMPLO: EL PROBLEMA DEL CAMBIO**

El problema consiste en, dada una lista de valores de billetes y una suma total, calcular la forma de pagar esa suma con la menor cantidad de billetes posibles.

En la manera tradicional, este problema se resuelve con la siguiente lógica:

1. Tomo el valor de billete más grande
2. Si la suma es mayor al valor del billete, aumentar el contador correspondiente a este tipo de billetes y restarle su valor a la suma (suma = suma – valorBillete). Si es menor, volver a hacerlo pero tomando como valor de billete al siguiente.
3. Volver al paso 1 con la suma actualizada.

Este proceso recursivo (o bien iterativo) termina cuando la suma es igual a 0.

Como mencionamos anteriormente, la idea de utilizar programación dinámica es ir guardando lo calculado anteriormente tanto para optimizar el resultado como el tiempo de ejecución.

Para la resolución de este problema utilizando programación dinámica, lo que se hace es lo siguiente:

Carga una matriz con n+1 filas siendo n la cantidad de billetes posibles y m+1 columnas siendo m el valor/suma que quiero averiguar.

La primer columna va a llenarse de 0s mientras que la primer fila se va a llenar de un valor máximo a determinar (en nuestro ejemplo el 99)

Para hacerlo más claro, nos vamos a basar en el siguiente ejemplo:

***valoresBilletes = {1,4,6} ; suma = 8***

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 |
| 0 |  |  |  |  |  |  |  |  |
| 0 |  |  |  |  |  |  |  |  |
| 0 |  |  |  |  |  |  |  |  |

Luego se va a proceder de la siguiente manera:

Se toma el primer valor de billete (el menor) y se va a poner en cada fila cuántos billetes de este valor se requiere para llegar a la suma de la columna

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 |  |  |  |  |  |  |  |  |
| 0 |  |  |  |  |  |  |  |  |

Y luego por cada fila se va a hacer lo siguiente:

Tomando el valor de billete correspondiente a cada fila (fila 1 = 1, fila 2 = 4, fila 3 = 6) voy a calcular si es menor la cantidad de billetes utilizada anteriormente para cada suma o la cantidad de billetes que me llevaría hacerlo de la forma tradicional (sin programación dinámica).

En cada columna se ubica la cantidad mínima de billetes, siendo igual a la celda superior en caso de que sea mejor lo calculado anteriormente o mayor si es mejor calcularlo de nuevo utilizando este valor de billete.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 1 | 2 | 3 | 1 | 2 | 3 | 4 | 2 |
| 0 | 1 | 2 | 3 | 1 | 2 | 1 | 2 | 2 |

Finalizada la carga de la matriz se procede a la obtención de la cantidad de billetes de cada valor que corresponde.

Para ello, nos ubicamos en la columna del valor buscado (8) y arrancamos desde la ultima fila (3). Desde ahí seguimos el siguiente algoritmo:

* Si es mayor al billete mínimo y la cantidad de billetes es igual a la cantidad de billetes que figura en la celda superior, omito este valor de billete y sigo con los valores inferiores.
* Si no, aumento el contador a este valor de billete y vuelvo a calcularlo con el valor decrementado (valor = valor – valorBillete)

En este ejemplo se ve claramente la diferencia entre ambas implementaciones:

|  |  |
| --- | --- |
| FORMA TRADICIONAL | PROGRAMACION DINAMICA |
| Cantidad billetes de 1: 2 | Cantidad billetes de 1: 0 |
| Cantidad billetes de 4: 0 | Cantidad billetes de 4: 2 |
| Cantidad billetes de 6: 1 | Cantidad billetes de 6: 0 |
| Cantidad total de billetes: 3 | Cantidad total de billetes: **2** |