

TEMA 4. PROGRAMACIÓN DINÁMICA

1. Introducción.

Las técnicas de programación dinámica tienen en común con los algoritmos voraces que, típicamente, se emplean para resolver problemas de optimización y permiten resolver problemas mediante una secuencia de decisiones. Sin embargo, a diferencia del esquema voraz, se producen varias secuencias de decisiones y solamente al final se sabe cuál es la mejor de ellas.

Por otro lado, con respecto al método de divide y vencerás, en este, se ataca de inmediato el caso completo, que a continuación se divide en subcasos más pequeños. Sin embargo, la programación dinámica empieza por los subcasos más pequeños, y por tanto más sencillos. Combinando sus soluciones, obtenemos las respuestas para subcasos de tamaños cada vez mayores, hasta que finalmente llegamos a la solución del caso original.

2. Principio de optimalidad.

Las técnicas de programación dinámica están basadas en el Principio de Optimalidad de Bellman que dice que “cualquier subsecuencia de decisiones de una secuencia óptima que resuelve un problema también debe ser óptima respecto al subproblema que resuelve”. Este principio no es aplicable a todos los problemas que nos podamos encontrar. Por ejemplo cuando buscamos la utilización óptima de unos recursos limitados. Aquí la solución óptima de un caso puede no ser obtenida por la combinación de soluciones óptimas de 2 o más subcasos, si los recursos utilizados en esos subcasos suman más que el total de recursos disponibles.

Supongamos que un problema se resuelve tras tomar una secuencia d_1, d_2, \dots, d_n de decisiones. Si hay d opciones posibles para cada una de las decisiones, una técnica de fuerza bruta exploraría un total de d^n secuencias posibles de decisiones (exploración combinatoria). La técnica de programación dinámica evita explorar todas las secuencias posibles por medio de la resolución de subproblemas de tamaño creciente y almacenamiento en una tabla de las soluciones óptimas de esos subproblemas para facilitar la solución de los problemas más grandes.

Por ejemplo, si el camino más corto entre A y C pasa por B, entonces la parte del camino que va desde A hasta B también debe seguir el camino más corto posible, al igual que la parte del camino que une B con C. En esta situación, el Principio de Optimalidad es aplicable. Sin embargo, si el camino más rápido para ir desde A hasta C nos lleva a través de B, no se sigue

necesariamente que lo mejor sea ir tan rápidamente como sea posible desde A hasta B, y después ir tan rápidamente como sea posible desde B hasta C. Si utilizamos demasiada gasolina en la primera mitad del viaje, tendremos que repostar en algún sitio en la segunda parte, y así perderemos más tiempo que el ganado al conducir muy deprisa. Los subviajes desde A hasta B y desde B hasta C no son independientes, porque comparten un recurso, así que la selección de una solución óptima para un subviaje puede impedir que utilicemos una solución óptima para el otro. En esta situación, el Principio de Optimalidad no es aplicable.

3. El problema de la mochila.

Se nos da un cierto número de objetos y una mochila. Esta vez suponemos que los objetos no se pueden fragmentar en trozos más pequeños, así que podemos decidir si tomamos un objeto o lo dejamos, pero sin fraccionarlo. Para $i=1,2,\dots,n$, supongamos que el objeto i tiene un peso positivo w_i y un valor positivo v_i . La mochila puede llevar un peso que no supere W . Nuestro objetivo es llenar la mochila de tal forma que se maximice el valor de los objetos incluidos, respetando la limitación de capacidad. Sea $x_i=0$ si decidimos no tomar el objeto i o bien 1 si lo incluimos. El enunciado del problema sería:

Maximizar $\sum_{i=1}^n x_i v_i$ con la restricción $\sum_{i=1}^n x_i w_i \leq W$, donde $v_i > 0$, $w_i > 0$ y $x_i \in \{0,1\}$ para $1 \leq i \leq n$.

El algoritmo voraz no se podría aplicar porque x_i tiene que ser 0 o 1. Supongamos que están disponibles tres objetos, el primero de los cuales pesa 6 unidades y tiene un valor de 8, mientras que los otros dos pesan 5 unidades cada uno y tienen un valor de 5 cada uno. Si la mochila puede llevar 10 unidades, entonces la carga óptima incluye a los dos objetos más ligeros, con un valor total de 10. El algoritmo voraz, comenzaría por seleccionar el objeto que pesa 6 unidades, puesto que es el que tiene un mayor valor por unidad de peso. Sin embargo, si los objetos no se pueden romper, el algoritmo no podrá utilizar la capacidad restante de la mochila. La carga que produce, consta de un solo objeto, y tiene un valor de 8 nada más.

Para resolver el problema por programación dinámica, preparamos una tabla $V[1..n, 0..W]$ que tiene una fila para cada objeto disponible, y una columna para cada peso desde 0 hasta W . En la tabla, $V[i,j]$ será el valor máximo de los objetos que podemos transportar si el límite de peso es j , con $0 \leq j \leq W$, y si solamente incluimos los objetos numerados desde el 1 hasta el i , con $1 \leq i \leq n$. La solución de este caso, se puede encontrar en $V[n, W]$. Una vez

más se puede aplicar el Principio de Optimalidad. Podemos rellenar la tabla fila por fila o columna por columna. En la situación general, $V[i,j]$ es el máximo de $V[i-1,j]$ y $V[i-1, j-w_i] + v_i$. La primera de estas opciones corresponde a no añadir el objeto i a la carga. La segunda corresponde a seleccionar el objeto i , lo cual incrementa el valor de la carga en v_i , y reduce en w_i la capacidad disponible. Por tanto, llenaremos las entradas de la tabla empleando la regla general:

$$V[i,j] = \max(V[i-1,j], V[i-1, j-w_i] + v_i)$$

Para las entradas fuera de límites definidos $V[0,j]=0$ cuando $j \geq 0$, y $V[i,j] = -\infty$ para todo i cuando $j < 0$.

Límite de peso	0	1	2	3	4	5	6	7	8	9	10	11
$w_1=1, v_1=1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2=2, v_2=6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3=5, v_3=18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4=6, v_4=22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5=7, v_5=28$	0	1	6	7	7	18	22	28	29	34	35	40

Los valores por unidad de peso son 1,3, 3.6, 3.67 y 4. Si solo podemos transportar un máximo de 11 unidades de peso, entonces la tabla muestra que podemos componer una carga cuyo valor es 40. La tabla V nos permite recuperar el valor de la carga óptima y su composición. En el ejemplo comenzamos por examinar $V[5,11]$. Como $V[5,11]=V[4,11]$ pero $V[5,11] \neq V[4,11-w_5]+v_5$, una carga óptima no puede incluir el objeto 5. A continuación, $V[4,11] \neq V[3,11]$ pero $V[4,11] = V[3,11-w_4]+v_4$, así que una carga óptima debe incluir al objeto 4. Ahora $V[3,5] \neq V[2,5]$ pero $V[3,5] = V[2,5-w_3]+v_3$, así que una carga óptima debe incluir al objeto 3. Prosiguiendo de esta forma, encontramos que $V[2,0] = V[1,0]$ y que $V[1,0] = V[0,0]$, así que una carga óptima no incluye al objeto 2 ni al objeto 1. Así, solo hay una carga óptima que consta de los objetos 3 y 4.

En este ejemplo el algoritmo voraz consideraría primero el objeto 5, por ser el mayor por unidad de peso. Luego el 4, pero este objeto no se puede incluir sin violar la restricción de capacidad. Luego examinaría los objetos 3, 2 y 1 y acabaría con una carga que constaría de los objetos 5, 2 y 1 con un valor total de 35. Luego el algoritmo voraz no funciona.

4. Cálculo del coeficiente binomial.

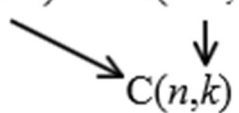
Supongamos que $0 \leq k \leq n$:

$$\binom{n}{k} = \begin{cases} 1 & k = 0 \text{ o } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 0 & \text{en otro caso} \end{cases}$$

Si calculamos directamente

función $C(n,k)$
 si $k=0$ o $k=n$ entonces devolver 1
 sino devolver $C(n-1,k-1)+C(n-1,k)$

Muchos de los valores $C(i,j)$, con $i < n$ y $j < k$ se calculan una y otra vez. Dado que el resultado final se obtiene sumando un cierto número de unos, el tiempo de este algoritmo es desproporcionado. Si utilizamos una tabla de resultados intermedios (triángulo de Pascal) entonces obtenemos un algoritmo más eficiente. La tabla debería ir llenándose línea por línea. De hecho, ni siquiera es necesario llenar toda la tabla: basta con mantener un vector k , que representa la línea actual y actualizar el vector de derecha a izquierda. Por tanto para calcular $\binom{n}{k}$, el algoritmo requiere un tiempo que está en $O(nk)$, y un espacio que está en $O(k)$, si suponemos que la suma es una operación elemental.

	0	1	2	3	...	$k-1$	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
...		
...	
$n-1$						$C(n-1,k-1) +$	$C(n-1,k)$
n							

5. El campeonato mundial.

El problema es el siguiente: en una competición hay dos equipos A y B que juegan un máximo de $2n-1$ partidas, y el ganador es el primer equipo que

consiga n victorias. No hay empates, y para cualquier partido dado hay una probabilidad constante p de que el equipo A sea el ganador, y $q = 1 - p$ de que pierda y gane el equipo B. Sea $P(i, j)$ la probabilidad de que gane el equipo A, cuando todavía necesita i victorias más para conseguirlo, mientras que el equipo B necesita j victorias para ganar. Antes del primer partido, la probabilidad de que gane el equipo A es $P(n, n)$: ambos equipos necesitan n victorias para ganar. Si el equipo A gana $P(0, i) = 1$, con $1 \leq i \leq n$. Si el equipo B gana $P(i, 0) = 0$ con $1 \leq i \leq n$. Entonces:

$$P(i, j) = pP(i-1, j) + qP(i, j-1) \quad i \geq 1, j \geq 1$$

Se puede calcular $P(i, j)$ de la siguiente forma:

```

función P(i,j)
    si i=0 entonces devolver 1
    sino si j=0 entonces devolver 0
    sino devolver pP(i-1,j)+qP(i,j-1)

```

Sea $T(k)$ el tiempo necesario en el caso peor para calcular $P(i, j)$, con $k = i + j$. Vemos que:

$$T(1) = c$$

$$T(k) \leq 2T(k-1) + d, \quad k > 1$$

con c y d constantes. Reescribiendo $T(k-1)$ en términos de $T(k-2)$ y así sucesivamente, obtenemos:

$$T(k) \leq 4T(k-2) + 2d + d, \quad k > 1$$

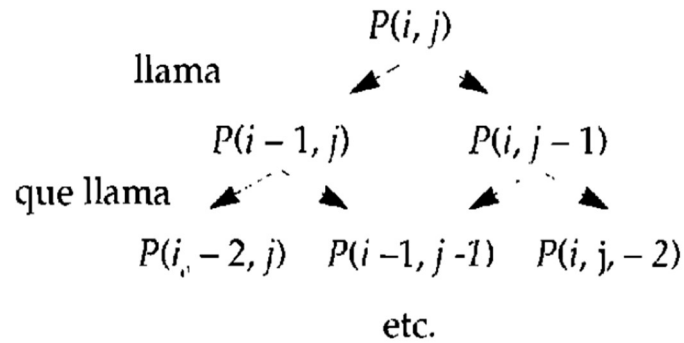
$$\dots$$

$$\leq 2^{k-1}T(1) + (2^{k-2} + 2^{k-3} + \dots + 2 + 1)d =$$

$$= 2^{k-1}c + (2^{k-1} - 1)d =$$

$$= 2^k(c/2 + d/2) - d$$

$T(k)$ está en $O(2^k)$ que es $O(4^n)$ si $i=j=n$.



De esta manera se calcula muchas veces el valor de cada $P(i, j)$. Es mejor usar una tabla para almacenar esos valores. Para acelerar el algoritmo, declaramos una matriz del tamaño adecuado y después vamos relleno las entradas, pero en vez de línea a línea trabajamos diagonal a diagonal. El algoritmo tiene que llenar una matriz $n \times n$. Tiempo: $O(n^2)$. Al igual que el triángulo de Pascal podríamos implementar el algoritmo para que el tiempo sea $O(n)$. El algoritmo para calcular $P(n, n)$ sería:

```

función serie(n,p)
    matriz P[0..n,0..n]
    q=1-p
    {Llenamos desde la esquina izquierda hasta la diagonal principal}
    para s=1 hasta n hacer
        P[0,s]=1
        P[s,1]=0
        para k=1 hasta s-1 hacer
            P[k,s-k]=pP[k-1,s-k]+qP[k,s-k-1]
    {Llenamos desde debajo de la diagonal principal hasta la esquina derecha}
    para s=1 hasta n hacer
        para k=0 hasta n-s hacer
            P[k,s-k]=pP[k-1,s-k]+qP[k,s-k-1]
    devolver P(n,n)
  
```

6. Devolver cambio.

El algoritmo voraz es muy eficiente pero funciona solamente en un número limitado de casos. Con ciertos sistemas monetarios o cuando faltan monedas de una cierta denominación o su número es limitado, el algoritmo puede encontrar una respuesta que no sea óptima o no hallar respuesta. Ejemplo: Tenemos monedas de 1, 4 y 6 unidades. Si tenemos que cambiar 8 unidades, el algoritmo voraz devuelve una moneda de 6 unidades y dos de una unidad, es decir, tres monedas. Pero podemos hacerlo mejor dando dos monedas de cuatro unidades. Supongamos que tenemos un número ilimitado de monedas

de n denominaciones diferentes y que una moneda de denominación i , con $1 \leq i \leq n$ tiene un valor de d_i unidades, $d_i > 0$. Tenemos que dar al cliente monedas por valor de N unidades empleando el menor número posible de monedas.

El problema se puede resolver mediante programación dinámica. Preparamos una tabla $c[1..n, 0..N]$ con una fila para cada denominación posible y una columna para las cantidades que van desde 0 unidades hasta N unidades. $C[i, j]$ es el número mínimo de monedas necesarias para pagar una cantidad de j unidades, con $1 \leq j \leq N$, empleando solamente monedas de las denominaciones desde 1 hasta i , con $1 \leq i \leq n$. Solución: $c[n, N]$ si lo único que necesitamos saber es el número de monedas que se necesitan. Comenzamos con $C[i, 0] = 0$ para todos los valores de i . La tabla del ejemplo sería la siguiente:

Cantidad	0	1	2	3	4	5	6	7	8
$d_1=1$	0	1	2	3	4	5	6	7	8
$d_2=4$	0	1	2	3	1	2	3	4	2
$d_3=6$	0	1	2	3	1	2	1	2	2

Nos da la solución para todos los casos que supongan un pago de 8 unidades o menos. Y el algoritmo sería el siguiente:

```

función monedas(N)
    {Devuelve el mínimo número de monedas necesarias para
    cambiar N unidades. El vector  $d[1..n]$  especifica las denominaciones: en el
    ejemplo 7 hay monedas de 1, 4 y 6 unidades}
    vector  $d[1..n] = [1, 4, 6]$ 
    matriz  $c[1..n, 0..N]$ 
    para  $i=1$  hasta  $n$  hacer  $c[i, 0] = 0$ 
    para  $i=1$  hasta  $n$  hacer
        para  $j=1$  hasta  $N$  hacer
            si  $i=1$  y  $j < d[i]$  entonces  $c[i, j] = +\infty$ 
            sino si  $i=1$  entonces  $c[1, j] = 1 + c[1, j - d[1]]$ 
            sino si  $j < d[i]$  entonces  $c[i, j] = c[i - 1, j]$ 
            sino  $c[i, j] = \min(c[i - 1, j], 1 + c[i, j - d[i]])$ 
    devolver  $c[n, N]$ 

```

7. Caminos mínimos.

Sea $G=(N, A)$ un grafo dirigido, N es el conjunto de nodos y A el conjunto de aristas. Deseamos calcular la longitud del camino mas corto entre cada par de nodos. Supongamos que los nodos de G están numerados desde 1

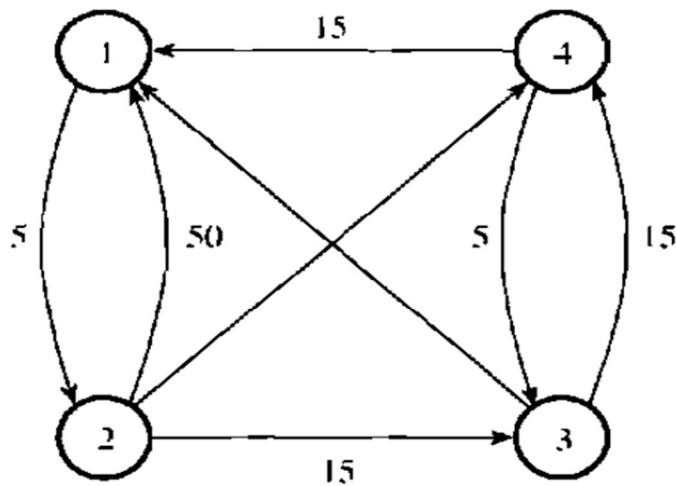
hasta n , así que $N=\{1,2,\dots,n\}$, y sea L la matriz que da las longitudes de las aristas, con $L[i,i]=0$ para $i=1,2,\dots,n$, $L[i,j]\geq 0$ para todo i y j , y $L[i,j]=\infty$ si no existe la arista (i,j) . Se puede aplicarse el Principio de Optimalidad: si k es un nodo del camino mínimo entre i y j , entonces la parte del camino que va desde i hasta k , y la parte del camino que va desde k hasta j deben ser óptimo también. Construimos una matriz D que da la longitud del camino más corto entre un par de nodos. El algoritmo da a D el valor inicial L , es decir, las distancias directas entre nodos, y luego efectúa n iteraciones. Después de la iteración k , D da la longitud de los caminos más cortos que utilizan solamente los nodos $\{1,2,\dots,k\}$ como nodos intermedios. Al cabo de n iteraciones, D nos da la longitud de los caminos más cortos que utilicen alguno de los nodos de N como nodo intermedio, que es el resultado deseado. En la iteración k , el algoritmo debe comprobar para cada par de nodos (i,j) si existe o no un camino que vaya de i a j pasando por el nodo k , y que sea mejor que el camino óptimo actual que pasa solo por los nodos $\{1,2,\dots,k-1\}$. Si D_k representa la matriz D después de la k -ésima iteración ($D_0=L$), entonces la comprobación necesaria debe implementarse en la forma:

$$D_k[i,j] = \min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j])$$

Se ha utilizado el hecho de que un camino óptimo que pase por k no visitará k dos veces. Este algoritmo se conoce como algoritmo de Floyd y es como sigue:

```
función Floyd(L[1..n, 1..n]): matriz[1..n, 1..n]
    matriz D[1..n, 1..n]
    D=L
    para k=1 hasta n hacer
        para i=1 hasta n hacer
            para j=1 hasta n hacer
                D[i,j]=min(D[i,j],D[i,k]+D[k,j])
    devolver D
```


Ejemplo:



$$D_0 = L = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

En la k-esima iteración, los valores de la k-esima fila y de la k-esima columna de D no cambian, porque $D\{k,k\}$ es siempre 0. Por tanto, no hace falta proteger estos valores al actualizar D. Esto nos permite utilizar una única D de tamaño $n \times n$.

La complejidad del algoritmo es $O(n^3)$. Con el algoritmo de Dijkstra el tiempo es el mismo pero la sencillez del algoritmo de Floyd hace que tenga una constante oculta más pequeña, y por tanto sea más rápido en práctica. Normalmente, deseamos saber cuál es el camino más corto, y no solamente su longitud. En tal caso, utilizaremos una segunda matriz P, cuyos elementos tienen todos ellos un valor inicial 0. El bucle mas interno del algoritmo pasa a ser

$$\begin{aligned} \text{si } D[i,k] + D[k,l] < D[i,l] \text{ entonces} \\ D[i,l] &= D[i,k] + D[k,l] \\ P[i,l] &= k \end{aligned}$$

Cuando se detiene el algoritmo, $P[i,j]$ contiene el número de la última iteración que haya dado lugar a un cambio en $D[i,j]$. Para recuperar el camino

más corto desde i hasta j , se examina $P[i,j]$. Si $P[i,j]=0$, entonces $D[i,j]$ nunca ha cambiado, y el camino mínimo pasa directamente por la arista (i,j) ; en caso contrario, si $P[i,j]=k$, entonces el camino más corto desde i hasta j pasa por k . Se examina recursivamente $P[i,k]$ y $P[k,j]$ para buscar cualquier otro posible nodo intermedio que este en el camino más corto. En el ejemplo anterior P pasa a ser

$$P = \begin{pmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

Dado que $P[1,3]=4$, el camino mínimo desde 1 hasta 3 pasa por 4. Examinando $P[1,4]$ y $P[4,3]$, descubrimos que entre 1 y 4 hay que pasar por 2, pero que de 4 a 3 pasamos directamente. Los recorridos desde 1 hasta 2 y desde 2 hasta 4 son directos. Por tanto, el camino mínimo desde 1 hasta 3 es 1, 2, 4, 3.