

Análisis y Diseño de Algoritmos

Tema 4: Programación Dinámica

Contenido

- Introducción
 - Conceptos generales
 - Ejemplo: números combinatorios
- Programación Dinámica
 - Elementos básicos
 - Algoritmo de Floyd
 - Cambio de monedas
- Funciones con memoria
 - Enfoque top-down
 - Cambio de monedas
- Ejemplos
 - El problema de la Mochila
 - Subsecuencia común más larga
- Referencias

Introducción

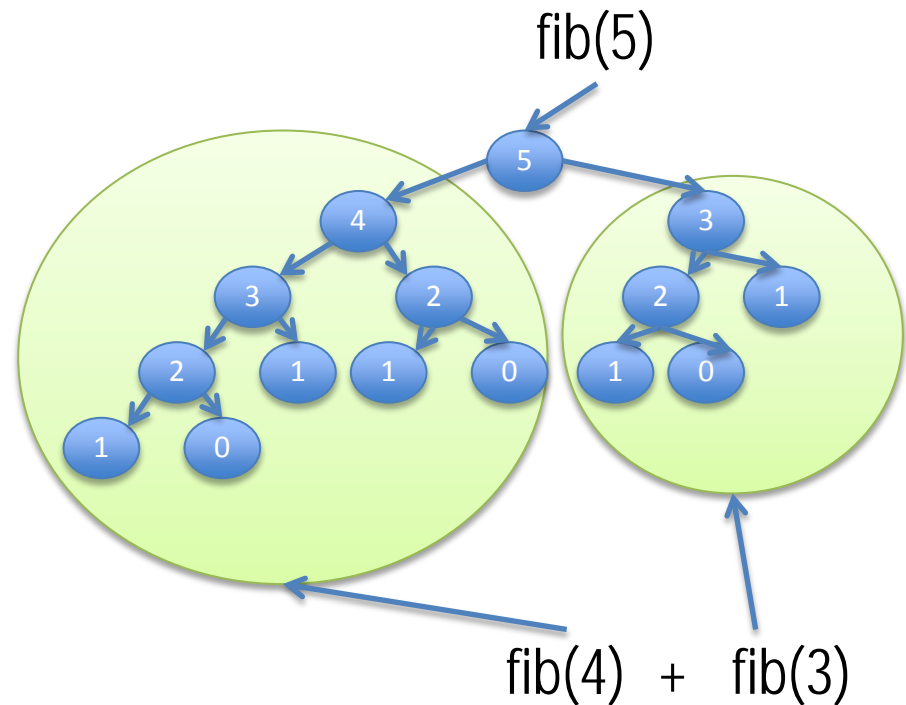
- La **programación dinámica** fue inventada por el matemático estadounidense **Richard Bellman** en 1950.
- Su campo de aplicación era la optimización de soluciones al problema de decisión multi-capas.
- Más tarde se ha visto su utilidad en problemas distintos a la optimización.
- Típicamente, **la programación dinámica puede aplicarse cuando un problema se subdivide en varios subproblemas cuyas soluciones se solapan.**
- Para evitar la recomputación de los subproblemas, sus soluciones se guardan en una tabla.
- Así que, de manera informal, la programación dinámica puede verse como la técnica **divide y vencerás más una tabla** (array u otra estructura de almacenamiento intermedio)



Introducción

- El truco de la programación dinámica está en resolver los subproblemas de forma ordenada, de manera que cuando se necesite la solución de un subproblema, si ya se ha calculado, esté disponible en la tabla.
- La programación dinámica es especialmente útil para problemas en los que la estrategia divide y vencerás produce un número exponencial de subproblemas, pero sólo hay algunos de ellos que se repiten exponencialmente.

- Por ejemplo, para calcular el quinto elemento de la sucesión de fibonacci:



Programación Dinámica: números combinatorios

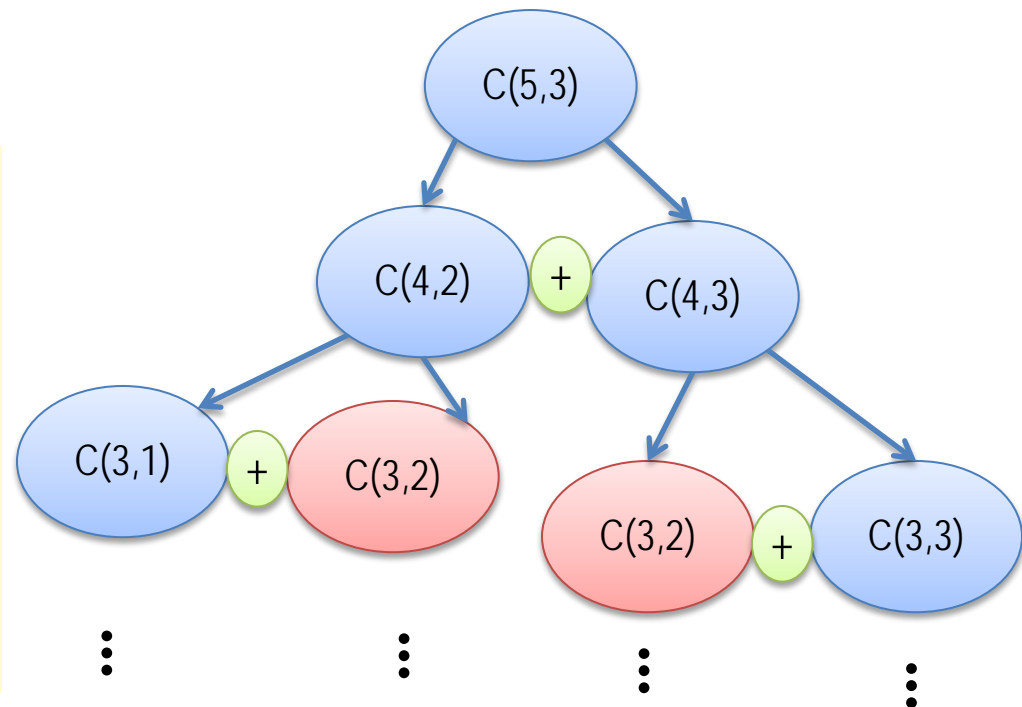
Considera la definición de los números combinatorios:

$$C(n, k) = C(n - 1, k - 1) + C(n - 1, k) \quad \text{si } n > k > 0$$

$$C(n, 0) = C(n, n) = 1$$

La naturaleza recursiva de la definición, y el solapamiento entre los subproblemas en que se divide el cálculo de $C(n, k)$ hace que este problema sea un candidato idóneo a ser resuelto mediante programación dinámica.

Observa que la implementación directa del algoritmo produciría una complejidad exponencial ($C(n, k)$ requiere 2^n llamadas).



Programación Dinámica: números combinatorios

Considera la definición de los números combinatorios:

$$C(n, k) = C(n - 1, k - 1) + C(n - 1, k) \quad \text{si } n > k > 0$$

$$C(n, 0) = C(n, n) = 1$$

$C(n, k)$ es el elemento situado en la fila n y la columna k de la tabla

La tabla contiene el conocido triángulo de Pascal

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

Programación Dinámica: números combinatorios

```
/**
 * Calcula el binomio de newton usando programación
 * dinámica
 * @param n >= k >= 0
 * @return C(n,k)
 */
public static int binomio(int n,int k){
    if (n<0 || k<0 || n<k)
        throw new IllegalArgumentException("Entrada inválida");
    int[][] tabla = new int[n+1][];
    for (int i = 0; i<=n; i++){
        tabla[i] = new int[i+1];
        for (int j = 0; j<=i; j++){
            if (j == 0 || j == i) tabla[i][j]=1;
            else
                tabla[i][j] = tabla[i-1][j-1] + tabla[i-1][j];
        }
    }
    return tabla[n][k];
}
```

operación primitiva

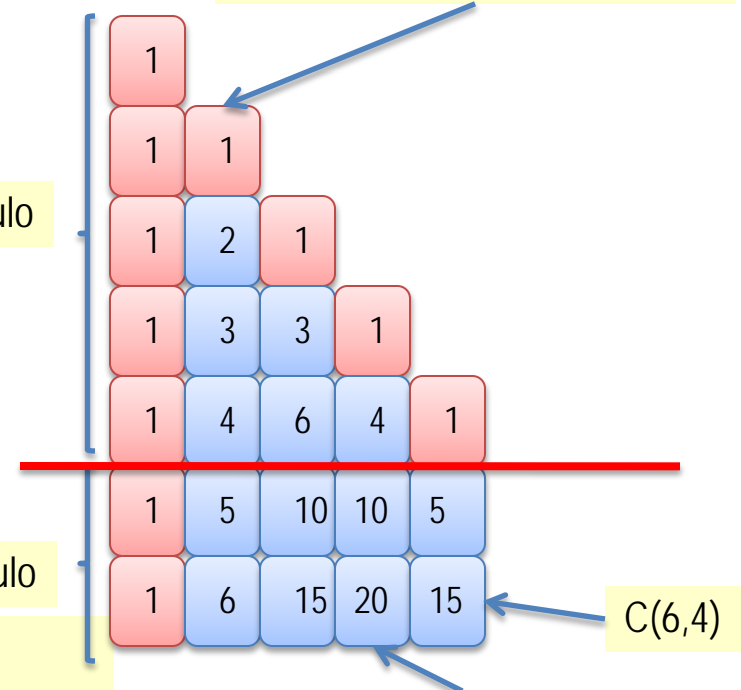
Programación Dinámica: números combinatorios-complejidad

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

Triángulo

Rectángulo

Para calcular los elementos en rojo no hace falta sumar.



Para calcular cada elemento azul hace falta 1 suma

¿Cuántas sumas hacen falta para calcular $C(n, k)$?

Tantas como elementos azules hay en la tabla.

Dividimos la tabla en el triángulo superior y el rectángulo inferior y obtenemos:

$$S(n, k) = \sum_{i=1}^k \sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^n \sum_{j=1}^k 1 = \sum_{i=1}^k (i-1) + \sum_{i=k+1}^n k = \frac{(k-1)k}{2} + k(n-k) \in \Theta(nk)$$

El principio de Optimalidad

La aplicación de la programación dinámica a problemas de optimización se basa en el ***principio de optimalidad***.

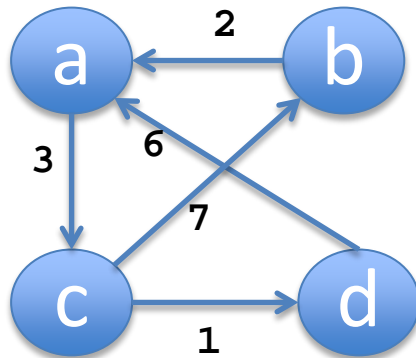
Un problema exhibe la propiedad de subestructura óptima, si ***una solución óptima al mismo*** contiene la ***solución óptima a subproblemas*** del mismo

La presencia de esta propiedad en un cierto problema puede indicarnos que su solución puede abordarse mediante la aplicación de la programación dinámica

El camino más corto

Algoritmo de Floyd

- Supongamos que tenemos un grafo dirigido y etiquetado con pesos que representan las distancias entre los nodos. (∞ significa que no hay camino directo).
- El problema a resolver consiste en encontrar la distancia menor d_{ij} que hay que recorrer para ir de un nodo i a un nodo j .



Matriz de pesos

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Matriz de distancias

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

camino más corto de c a a

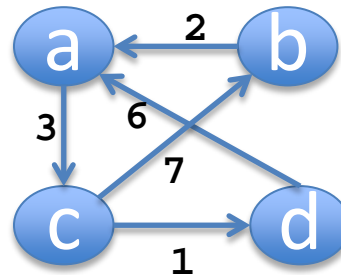
El camino más corto

Algoritmo de Floyd

Un algoritmo directo supondría probar para cada vértice todas las variaciones posibles de $n-1$ elementos tomados de 1 en 1, de 2 en 2 hasta de $n-1$ en $n-1$. Lo que supone un orden de magnitud de $n!$

Matriz de pesos

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0



... ..

Matriz de distancias

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

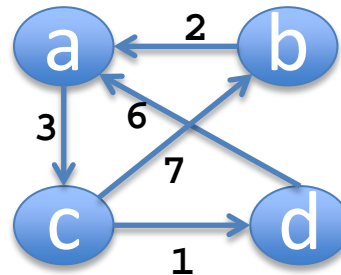
El camino más corto

Algoritmo de Floyd

Dado un grafo con n vértices, el algoritmo de Floyd se basa en calcular la matriz de las distancias más cortas construyendo una sucesión de matrices $D^0, \dots, D^{k-1}, D^k, \dots, D^n$, siendo D^0 la matriz de pesos y D^n la matriz que se quiere calcular. Cada elemento d_{ij}^k que se encuentra en la posición (i, j) de la matriz D^k contiene la longitud mínima encontrada hasta ese momento para ir del nodo i al nodo j .

Matriz de pesos

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0



... ..

Matriz de distancias

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

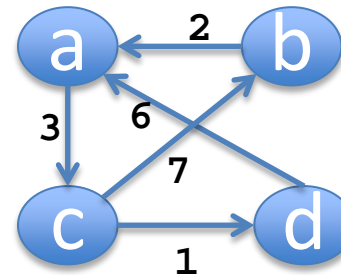
El camino más corto

Algoritmo de Floyd

Suponiendo que los nodos del grafo se numeran de 1 a n , en cada matriz D^k , el elemento d_{ij}^k es la distancia más corta para ir del vértice v_i al v_j , a través de caminos que sólo utilizan nodos numerados de 1 a k .

Así d_{ij}^0 es la distancia más corta para ir de v_i al v_j , sin nodos intermedios, es decir, la etiqueta de la flecha del nodo v_i al v_j , si existe tal flecha, o ∞ , en otro caso.

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0



El camino más corto

Algoritmo de Floyd

Supongamos que tenemos construida una matriz D^{k-1} , la matriz D^k se construye como sigue:

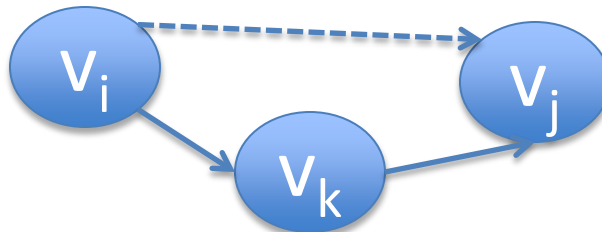
Cada componente d_{ij}^k es la distancia mínima para ir de v_i a v_j utilizando sólo vértices numerados entre 1 y k .

1. Si el camino más corto de este tipo pasa por el vértice k , entonces (como las distancias son positivas) a lo sumo pasa una vez. Por lo tanto, el camino es del tipo:

v_i , vértices entre 1 y $k-1$, v_k , vértices entre 1 y $k-1$, v_j

Este camino puede dividirse entonces en dos subcaminos:

- Un camino de v_i a v_k , con nodos intermedios entre 1 y $k-1$
- Un camino de v_k a v_j , con nodos intermedios entre 1 y $k-1$



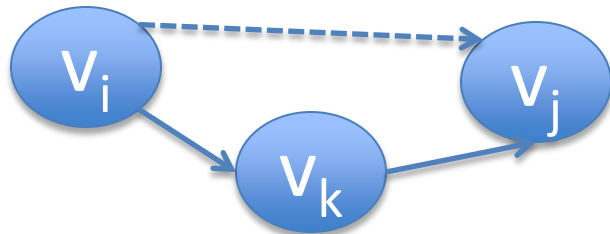
El camino más corto

Algoritmo de Floyd

Pero si v_i , vértices entre 1 y $k-1, v_k$, vértices entre 1 y $k-1, v_j$ es el camino más corto con vértices numerados entre 1 y k , entonces

- El camino v_i a v_k , con nodos intermedios entre 1 y $k-1$, es el más corto de entre este tipo de caminos, luego su distancia es d_{ik}^{k-1}
- El camino de v_k a v_j , con nodos intermedios entre 1 y $k-1$, es el más corto de entre este tipo de caminos, luego su distancia es d_{kj}^{k-1}

Por lo que, en este caso, $d_{ij}^k = d_{ik}^{k-1} + d_{kj}^{k-1}$



Principio de Optimalidad

El camino más corto

Algoritmo de Floyd

Supongamos que tenemos construida una matriz D^{k-1} , la matriz D^k se construye como sigue:

Cada componente d_{ij}^k es la distancia mínima para ir de v_i a v_j utilizando sólo vértices numerados entre 1 y k .

2. Si el camino más corto de este tipo no pasa por el vértice v_k entonces su distancia ya estaba almacenada en d_{ij}^{k-1}

Por lo tanto, $d_{ij}^k = \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$

D^k se construye a partir de D^{k-1}

El camino más corto

Algoritmo de Floyd

$$D^0 =$$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

$$D^1 =$$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	∞	7	0	1
d	6	∞	9	0

$$D^2 =$$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	9	7	0	1
d	6	∞	9	0

$$D^3 =$$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

$$D^4 =$$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

El camino más corto

Algoritmo de Floyd

```
/**
 *
 * @param m matriz de pesos asociada a un grafo.
 * m[i][j] = -1 significa que no es posible ir directamente
 * del nodo i al nodo j.
 * El método transforma m en una matriz que contiene
 * las distancias más cortas entre cada par de nodos
 * del grafo, utilizando el algoritmo de Floyd
 *
 */
public static void caminoMasCorto(int[][] m){
    for (int k=0; k<m.length;k++){
        // calcula el valor de d_{ij}^k en función
        // de los valores de D^{k-1}
        for (int i = 0; i<m.length; i++){
            for (int j = 0; j<m[i].length;j++){
                m[i][j]= minimo(m[i][j], suma(m[i][k],m[k][j]));
            }
        }
    }
}
```

El algoritmo tiene complejidad $T(n) \in \Theta(n^3)$, siendo n el número de vértices del grafo.

El camino más corto

Algoritmo de Floyd

```
/**
 * Calcula el minimo entre dos numeros, suponiendo
 * que -1 representa infinito.
 */
public static int minimo(int a,int b){
    if (a==-1 && b==-1) return -1;
    else if (a==-1) return b;
    else if (b==-1) return a;
    else return Math.min(a, b);
}

/**
 * Calcula la suma de dos números suponiendo
 * que -1 representa infinito.
 *
 */
public static int suma(int a,int b){
    if (a==-1 || b==-1) return -1;
    else return a+b;
}
```

Subproblemas solapados

- La **programación dinámica** reduce la búsqueda de una **solución óptima a un problema** a la búsqueda de **soluciones óptimas a subproblemas** más pequeños bien escogidos.
- De la solución óptima de los subproblemas se pasa a la solución óptima del problema de manera **ascendente**.
- Para aplicar la técnica hay que saber
 - escoger los subproblemas adecuados
 - combinar las soluciones de los subproblemas
- El número de subproblemas no debe ser muy grande, por ejemplo, debería ser polinómico con respecto al tamaño de la entrada.
- Para no repetir el cálculo de las soluciones a los subproblemas, se almacenan **en una tabla**, de manera que cuando se necesita una solución sólo hay que extraerla de la tabla.

El problema de las monedas

Supongamos que tenemos un suministro ilimitado de monedas de n valores distintos d_1, \dots, d_n .

Tenemos que pagar cierta cantidad M
¿Cuál es la forma de hacerlo empleando el **número mínimo** de monedas?

Si hubiera que tomar 0 ó 1 moneda de cada tipo, habría que probar 2^n casos. Luego fuerza bruta es exponencial.



El problema de las monedas

1. Si la cantidad M coincide con el valor de una de las monedas, la solución es trivial.
2. Si hay que emplear más de una moneda, hay que buscar una forma de resolver el problema en función de subproblemas óptimos más pequeños.
 - Sea s la solución óptima, y supongamos que utiliza una moneda de valor d_i . Entonces, el resto de monedas que quedan tienen que formar una solución óptima para pagar $M - d_i$. Si no fueran una solución óptima a este problema, entonces s no sería óptima, en oposición a lo que estamos suponiendo.
3. Por lo tanto, una solución óptima s para M está formada por una moneda de un valor d_i , más una solución óptima para $M - d_i$.

El problema de las monedas

Podemos plantear el problema en términos de subproblemas C_{ij} : *pagar la cantidad j con monedas de valor d_1, \dots, d_i .*
El problema a resolver es C_{nM} .

Construimos una tabla $C[1 \dots n, 0 \dots M]$ que vamos rellenando de forma incremental:

1. $C[i, 0] = 0$ para $1 \leq i \leq n$.
2. $C[1, j] = \infty$, si $j < d_1$.
3. Si $i > 1, j > 0$ hay dos opciones:
 - (a) No usar monedas de valor $d_i \Rightarrow C[i, j] = C[i - 1, j]$
 - (b) Usar 1 moneda de valor $d_i \Rightarrow C[i, j] = 1 + C[i - 1, j - d_i]$

Como queremos el mínimo de monedas:

$$C[i, j] = \min(C[i - 1, j], 1 + C[i - 1, j - d_i])$$

El problema de las monedas

- Si tenemos monedas de 1, 4 y 6 céntimos y queremos pagar 8 céntimos:

	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

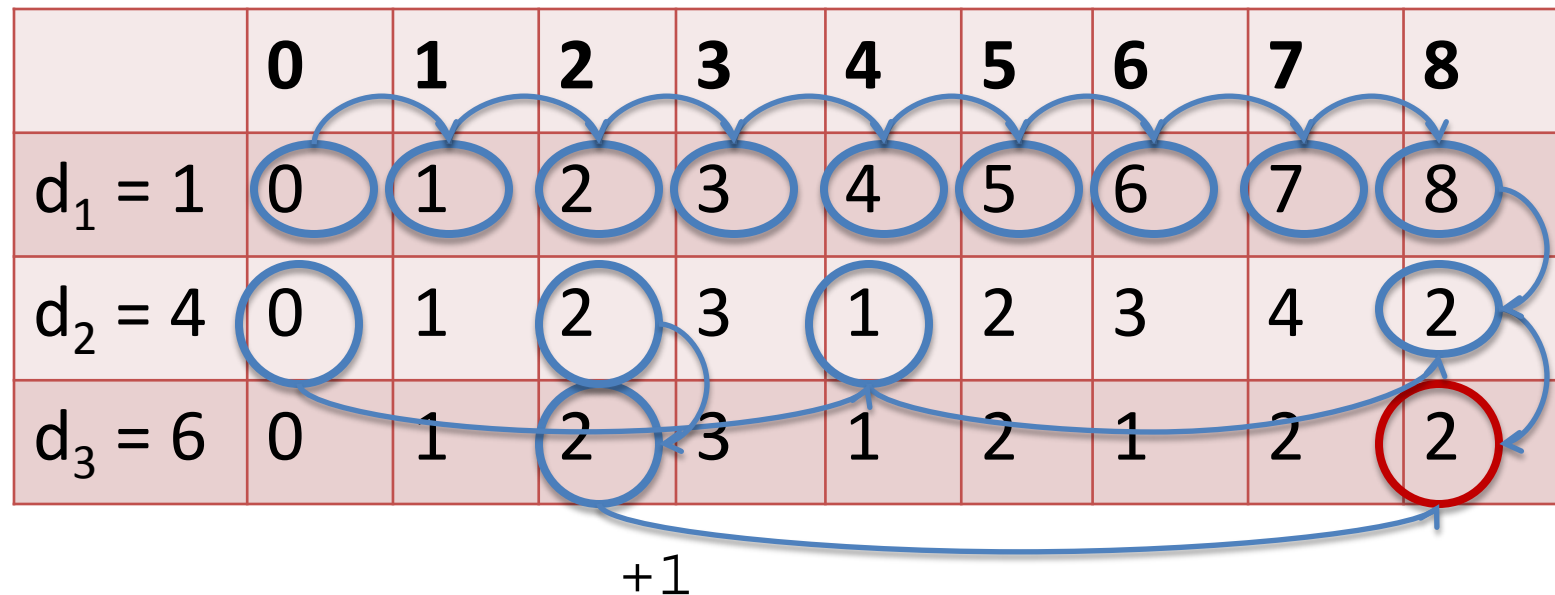
+1

El problema de las monedas

```
public static int cambioMonedas(int[] mon,int cant){
    int[][] C = new int[mon.length][];
    for (int i = 0; i<C.length; i++) C[i] = new int[cant+1];
    for (int i= 0; i<C.length; i++){
        C[i][0] = 0; //para pagar 0 hacen falta 0 monedas
    }
    for (int i= 0; i<C.length; i++){
        for (int j = 1;j < C[i].length;j++){
            if (i==0 && j<mon[0]) C[i][j] = -1;
                // la cantidad no se puede pagar
            else if (i==0) C[i][j] = suma(1,C[i][j-mon[0]]);
                // uso una moneda de valor mon[0], y calculo el resto
            else if (j < mon[i]) C[i][j] = C[i-1][j];
                // si el valor de mon[i] excede la cantidad
                // a pagar, uso monedas de menor valor
            else C[i][j] = minimo(C[i-1][j],suma(1,C[i][j-mon[i]]));
        }
    }
    return C[mon.length-1][cant];}
```

El problema de las monedas

- En realidad, para para resolver el problema no hay que rellenar la tabla completa



El problema de las monedas

- Si combinamos la versión recursiva (**top-down**) con una tabla en la que almacenamos los valores calculados, para no repetir cálculos, conseguimos calcular exactamente los valores necesarios.
- Esta técnica que combina recursión más una tabla de almacenamiento temporal se denomina **Memoización**.

El problema de las monedas

```
public static int cambioMonedas(int[] mon,int cant){
    int[][] C = new int[mon.length][];
    for (int i = 0; i<C.length; i++){
        C[i] = new int[cant+1];
    }
    for (int i= 0; i<C.length; i++){
        C[i][0] = 0;
        for (int j = 1;j<C[i].length;j++){
            C[i][j] = -2; // inicializamos la tabla
        }
        return cambioMonedas(mon.length-1,mon,cant,C);
        // llamamos al algoritmo recursivo
    }
}
```

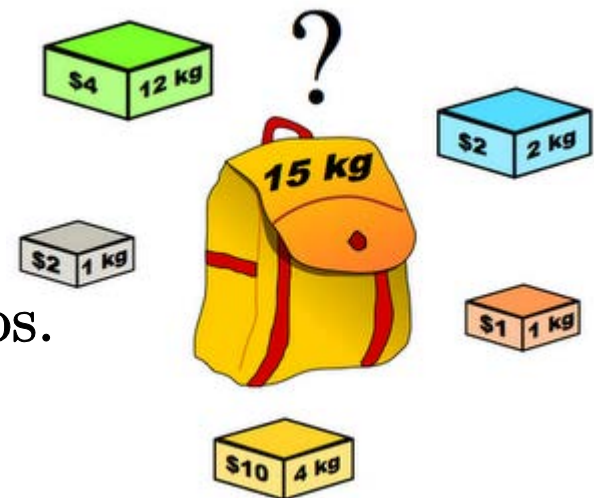
El problema de las monedas

```
/**
 * Calcula C[i][M] con las monedas que han en mon, y la tabla C
 * como auxiliar
 */
public static int cambioMonedas(int i,int[] mon,int M,int[][] C){
    if (C[i][M] == -2){ //Si la celda está vacía
        if (i==0 && M >= mon[0]){
            C[i][M]=suma(1,cambioMonedas(0,mon,M-mon[0],C));
        } else if (i>0){
            if (M < mon[i]){
                C[i][M]= cambioMonedas(i-1,mon,M,C);
            } else {
                int n1 = cambioMonedas(i-1,mon,M,C);
                int n2 = suma(1,cambioMonedas(i,mon,M-mon[i],C));
                C[i][M]= minimo(n1,n2);
            }
        }
    }
    return C[i][M];
}
```

El problema de la mochila (0, 1)

Dados n items de pesos conocidos w_1, \dots, w_n y de valor v_1, \dots, v_n , y dada una mochila de capacidad W encontrar el subconjunto de items de más valor que caben en la mochila.

Suponemos que los pesos w_1, \dots, w_n y la capacidad de la mochila W son enteros positivos. Los valores de los items v_1, \dots, v_n son reales.



El problema de la mochila (0, 1)

Para resolver el problema, aplicando la técnica de programación dinámica tenemos que encontrar una ecuación de recurrencia que dé la solución de una instancia del problema de la mochila, en función de las soluciones de instancias más pequeñas.

El problema de la mochila (0, 1)

Supongamos una instancia definida por los i primeros items ($1 \leq i \leq n$), con pesos w_1, \dots, w_i y valores v_1, \dots, v_i , y una capacidad de la mochila $j, 1 \leq j \leq W$.

Sea $V[i,j]$ el valor óptimo para esta instancia, es decir, el valor del más valioso subconjunto de los i primeros items, que caben en una mochila de tamaño j .

El problema de la mochila (0, 1)

Dividimos los subconjuntos de los i primeros items que caben en una mochila de tamaño j en dos grupos:

1. Los que no tienen al item i .

En ese caso, por definición el valor de la solución óptima es $V[i - 1, j]$.

2. Los que tienen al item i .

Entonces la solución óptima está formada por la solución óptima $v_i + V[i - 1, j - w_i]$

Por lo tanto, una solución óptima para $V[i, j]$ se obtiene calculando el máximo de estos dos valores.

El problema de la mochila (0, 1)

La solución es, por lo tanto,

$$V[i, j] = \begin{cases} V[0, j] = 0 \\ V[i, 0] = 0 \\ \max\{V[i-1, j], v_i + V[i-1, j-w_i]\} & \text{si } j - w_i \geq 0 \\ V[i-1, j] & \text{si } j - w_i < 0 \end{cases}$$

El problema de la mochila (0, 1)

ítem	peso	valor
1	2	12
2	1	10
3	3	20
4	2	15

$$W = 5$$

Capacidad j

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

Subsecuencia común más larga

- En biología, a veces, es necesario comparar el DNA de diferentes organismos.
- Una cadena de DNA está formada por una secuencia de moléculas (bases) que se denotan por su inicial {A,C,G,T}
- Por ejemplo, dos cadenas podrían ser
 - $S_1 = \text{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$
 - $S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$

Subsecuencia común más larga

- Para medir la similitud entre las dos cadenas pueden establecerse distintos criterios.
- Uno de ellos es calcular la *subsecuencia común más larga (SCL)*
- Dada la secuencia $X = \langle x_1, x_2, \dots, x_n \rangle$, otra secuencia $Z = \langle z_1, z_2, \dots, z_k \rangle$, es una **subsecuencia** de X si existe una secuencia creciente de índices $\langle i_1, i_2, \dots, i_k \rangle$ de X , tal que para todo $1 \leq j \leq k$, $x_{i_j} = z_j$
- Por ejemplo, CTCTGAA es una subsecuencia de
 - $S_1 = \text{ACCGGTCGAGTGC GCGGAAGCCGGCCGAA}$

Subsecuencia común más larga

- El problema de la *subsecuencia común más larga* consiste en, dadas dos secuencias $X = \langle x_1, x_2, \dots, x_n \rangle$, e $Y = \langle y_1, y_2, \dots, y_m \rangle$, encontrar la subsecuencia común más larga (una de ellas, si hay más de una)
- Por ejemplo, CTCTGAA es una subsecuencia común de
 - $S_1 = \text{ACCGGTCGAGTGC GCGGAA GCCGGCCGAA}$
 - $S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$aunque no es la más larga

Subsecuencia común más larga

- Para encontrar una solución al problema utilizando programación dinámica, debemos establecer la solución en función de instancias más pequeñas que tengan una subestructura óptima.
- Para ello necesitamos definir el **prefijo X_j** de una cadena **$X = \langle x_1, x_2, \dots, x_n \rangle$** , como la subcadena de **X** , **$X_j = \langle x_1, x_2, \dots, x_j \rangle$** , para **$0 \leq j \leq n$**

Subsecuencia común más larga

- Sea $Z = \langle z_1, z_2, \dots, z_k \rangle$, una SCL de las cadenas
 $X = \langle x_1, x_2, \dots, x_n \rangle$, $Y = \langle y_1, y_2, \dots, y_m \rangle$
 - Si $x_n = y_m$ entonces $x_n = y_m = z_k$ y Z_{k-1} es una SCL de X_{n-1} y Y_{m-1} .
 - Si $x_n \neq y_m$ entonces $z_k \neq x_n$ implica que Z es una SCL de X_{n-1} y Y .
 - Si $x_n \neq y_m$ entonces $z_k \neq y_m$ implica que Z es una SCL de X e Y_{m-1} .

Subsecuencia común más larga

Dadas las cadenas $X = \langle x_1, \dots, x_n \rangle$ y $Y = \langle y_1, \dots, y_m \rangle$

Para encontrar una SCL Z de X e Y tenemos tres casos:

1. Si $x_n = y_m$ entonces encontramos una SCL Z' de X_{n-1} y Y_{m-1} y le concatenamos x_m , es decir, $Z = Z'.x_m$
2. Si $x_n \neq y_m$, entonces tenemos que resolver dos casos:
 - (a) Encontrar una SCL Z_1 de X_{n-1} y Y
 - (b) Encontrar una SCL Z_2 de X y Y_{m-1}

y definir Z como la secuencia más larga de Z_1 y Z_2

Analizando la solución, se ve que en cada caso hay que calcular la SCL de X_{m-1} y Y_{n-1} , por lo que para evitar la recomputación de este valor lo almacenamos en una tabla.

Subsecuencia común más larga

Definimos el elemento $C[i, j]$ de la tabla $C[0..n, 0..m]$, como la longitud de la SCL de X_i y Y_j del modo siguiente:

$$C[i, j] = \begin{cases} 0 & \text{si } i = 0 \text{ o } j = 0 \\ C[i - 1, j - 1] + 1 & \text{si } i, j > 0 \text{ y } x_i = y_j \\ \max(C[i - 1, j], C[i, j - 1]) & \text{si } i, j > 0 \text{ y } x_i \neq y_j \end{cases}$$

Subsecuencia común más larga

```
public static String scl(String x,String y){
    int[][] C = new int[x.length()+1][];
    for (int i=0; i<C.length; i++)
        C[i] = new int[y.length() + 1];
    Direccion[][] b = new Direccion[x.length()+1][];
    for (int i = 0; i<b.length; i++)
        b[i] = new Direccion[y.length()+1];
    for (int i=0;i<x.length()+1; i++) C[i][0] = 0;
    for (int j=0;j<y.length()+1; j++) C[0][j] = 0;
    for (int i=1;i<x.length()+1;i++){
        for (int j=1;j<y.length()+1;j++){
            if (x.charAt(i-1)== y.charAt(j-1)){
                C[i][j] = C[i-1][j-1] + 1;
                b[i-1][j-1] = Direccion.DIAG;
            } else if (C[i-1][j] >= C[i][j-1]){
                C[i][j] = C[i-1][j];
                b[i-1][j-1] = Direccion.ARRIBA;
            } else {
                C[i][j] = C[i][j-1];
                b[i-1][j-1] = Direccion.IZDA;
            }
        }
    }
    return construir_SCL(x,x.length()-1,y.length()-1,b);}
}
```

C es la matriz de las longitudes

b nos permite reconstruir la subcadena encontrada

si hay una coincidencia, incremento la longitud y guardo que decremento i y j

si no hay una coincidencia, y la LCS de $X_{i-1}Y_j$ es mayor que la LCS de X_iY_{j-1} me quedo con la mayor y guardo que decremento i


en otro caso, me quedo con la LCS de X_iY_{j-1} y guardo que decremento j

Este método reconstruye la subcadena

Subsecuencia común más larga

```
public enum Direccion {  
    ARRIBA, IZDA, DIAG;  
}
```

```
public static String construir_SCL(String x,int i,int j,Direccion[][] b){  
    if (i==-1 || j==-1) return "";  
    if (b[i][j].equals(Direccion.DIAG)){  
        return construir_SCL(x,i-1,j-1,b).concat(String.valueOf(x.charAt(i)));  
    } else if (b[i][j].equals(Direccion.ARRIBA)){  
        return construir_SCL(x,i-1,j,b);  
    } else return construir_SCL(x,i,j-1,b);  
}
```



Sólo se guarda el carácter, si
hay una coincidencia, es
decir, si i y j se han
decrementado, y la dirección
guardada es *DIAG*

Subsecuencia común más larga

- Sea $X = ABCBDAB$ y $Y = BDCABA$

		j	0	1	2	3	4	5	6
			y_j	B	D	C	A	B	A
i									
0	x_i	0	0	0	0	0	0	0	0
1	A	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←	1 ↖	
2	B	0	1 ↖	1 ←	1 ←	1 ↑	2 ↖	2 ←	
3	C	0	1 ↑	1 ↑	2 ↖	2 ←	2 ↑	2 ↑	
4	B	0	1 ↖	1 ↑	2 ↑	2 ↑	3 ↖	3 ←	
5	D	0	1 ↑	2 ↖	2 ↑	2 ↑	3 ↑	3 ↑	
6	A	0	1 ↑	2 ↑	2 ↑	3 ↖	3 ↑	4 ↖	
7	B	0	1 ↖	2 ↑	2 ↑	3 ↑	4 ↖	4 ↑	

BCBA

ojo, el array **b** está desplazado una fila y una columna con respecto a la representación en el dibujo

Subsecuencia común más larga

– Para las cadenas

$S_1 = \text{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$

$S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$

una subsecuencia común más larga es

$\text{GTCGTCGGAAGCCGGCCGAA}$

$S_1 = \text{ACC}\textcolor{red}{GGTCGA}\textcolor{red}{GTGCGCGGAAGCCGGCCGAA}$

$S_2 = \textcolor{red}{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$

Referencias

- *Introduction to The Design & Analysis of Algorithms*. A. Levitin. Ed. Adison-Wesley
- *Introduction to Algorithms*. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Ed. The MIT Press