Técnicas de diseño de algoritmos
Programación dinámica

Luis Javier Rodríguez Fuentes
Amparo Varona Fernández

Departamento de Electricidad y Electrónica
Facultad de Ciencia y Tecnología, UPV/EHU
luisjavier.rodriguez@ehu.es
amparo.varona@ehu.es

OpenCourseWare 2015
Campus Virtual UPV/EHU

Programación dinámica: Indice

Programación dinámica: memoria que ahorra tiempo

- Algunos problemas se pueden dividir en partes, resolver cada parte por separado y combinar las soluciones. Eso es lo que hacemos de forma recursiva en la técnica conocida como *Divide y Vencerás* (DyV). En estos casos, los subproblemas generados son independientes y no hay esfuerzos redundantes.
- Sin embargo, ciertas soluciones recursivas producen los mismos subproblemas muchas veces y resultan altamente ineficientes, ya que deben resolver dichos subproblemas cada vez que se generan. Esto es lo que se conoce como **solapamiento de subproblemas** (p.e. Fibonacci recursivo).
- Manteniendo el esquema top-down de los algoritmos recursivos, es posible reducir drásticamente el esfuerzo computacional mediante lo que se conoce como **memorización**: cada vez que se ha de realizar un cálculo, primero se busca en una tabla; si está, se utiliza; si no, se realiza el cálculo y se almacena el resultado en la tabla.
- Esta pequeña modificación implica usar memoria auxiliar para almacenar los resultados ya calculados, lo cual, dependiendo del tamaño del problema, podría ser inviable, o incluso, si el espacio de parámetros no es discreto, imposible de implementar. En algunos textos, el método descrito se denomina **top-down dynamic programming**.

Programación dinámica: primero los casos pequeños

- Los algoritmos recursivos operan según un esquema *top-down*, planteando la solución a un problema en función de la solución a problemas más pequeños que derivan del primero. Para resolver éstos, se producen problemas todavía más pequeños, etc. hasta que se llega a problemas de tamaño tan pequeño que se pueden resolver directamente.
- Los algoritmos de **Programación Dinámica** (PD) operan justo al revés, según un esquema **bottom-up**: resuelven primero los problemas más pequeños, que almacenan para resolver después problemas mayores a partir de las soluciones ya obtenidas, y así van resolviendo problemas cada vez mayores hasta obtener la solución al problema planteado originalmente.
- **IMPORTANTE:** el espacio de parámetros ha de ser discreto (es decir, indexable), de modo que pueda construirse una tabla.
- Así pues, los algoritmos PD son iterativos y pueden tener fuertes requerimientos de memoria. En muchos casos, sin embargo, la solución a un cierto problema se podrá expresar en términos de unos pocos problemas más pequeños, de modo que la memoria necesaria se puede reducir drásticamente si los cálculos se organizan adecuadamente.

Programación dinámica: el principio de optimalidad

- Los algoritmos PD se aplican a problemas de optimización y parten siempre de una definición recursiva (*ecuación de Bellman*), que expresa la solución óptima a un problema como combinación de las soluciones óptimas a ciertos subproblemas que se derivan del primero.
- No todos los problemas de optimización admiten una expresión recursiva de este tipo. Sólo aquéllos que cumplen el **principio de optimalidad de Bellman**:

  > *El principio de optimalidad se cumple si la solución óptima a un problema contiene dentro de sí las soluciones óptimas a todos los subproblemas en que podemos dividirlo.*

- Considerese, por ejemplo, el siguiente problema: encontrar el **camino de coste mínimo** entre dos nodos u y v de un grafo ponderado. Si dicho camino pasa por otro nodo w, necesariamente los caminos que van de u a w y de w a v han de ser también de coste mínimo.
- Un ejemplo en sentido contrario: encontrar el **camino simple de coste máximo** entre dos nodos u y v de un grafo ponderado. En este caso, si dicho camino pasa por un nodo w, los caminos de u a w y de w a v no tienen por qué ser la solución óptima a los subproblemas correspondientes. Casi con total seguridad, las soluciones óptimas a dichos subproblemas compartirán algún nodo intermedio, de manera que al concatenarlas el camino resultante no será *simple*.

Programación dinámica: un ejemplo

- Consideremos el problema de obtener el término n de la serie de Fibonacci, que se define recursivamente como:

$$F_n = F_{n-1} + F_{n-2} \qquad \text{con } F_0 = F_1 = 1$$

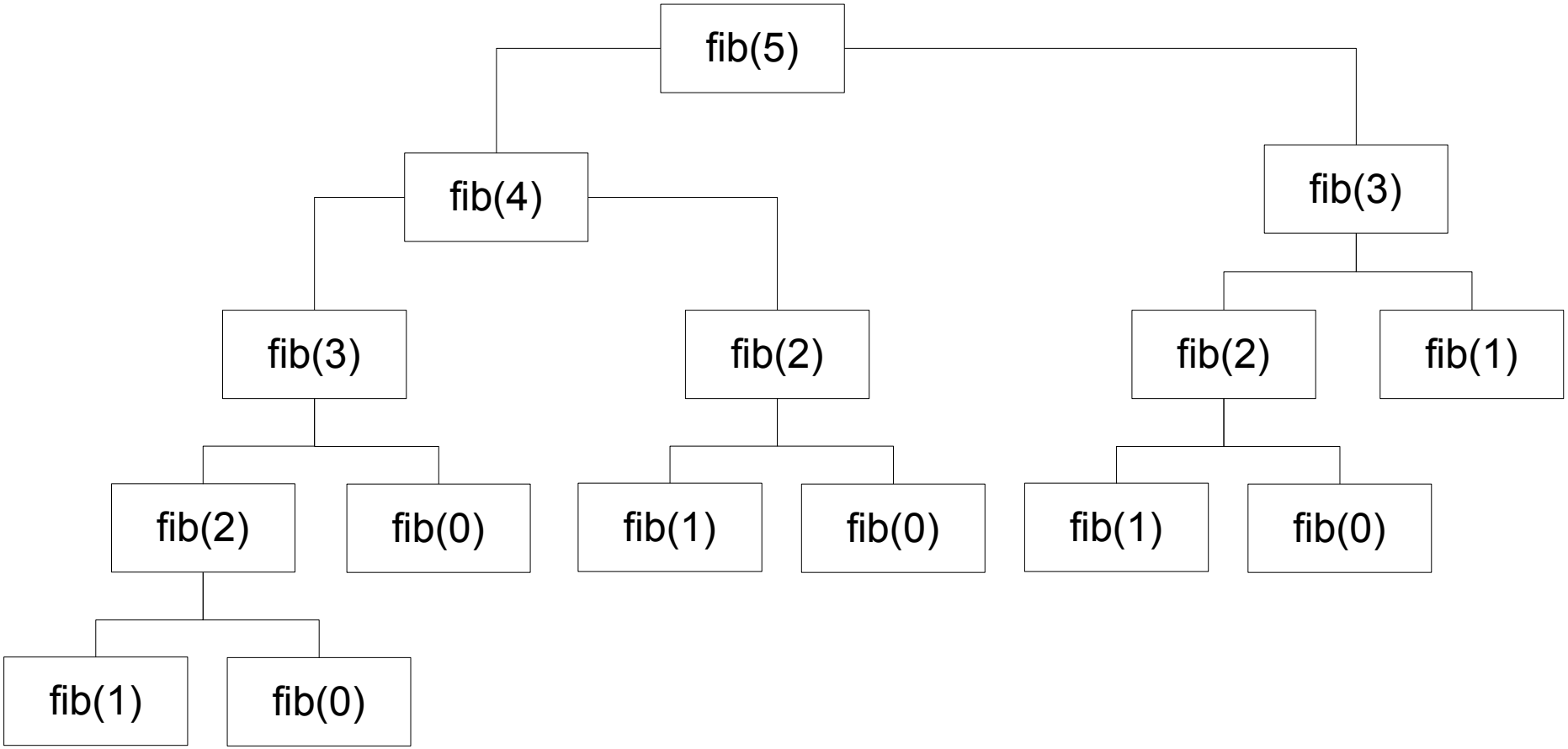
- La solución recursiva es inmediata:

```
def fib_rec(n):
    if n<2:
        return 1
    else:
        return fib_rec(n-1)+fib_rec(n-2)
```

- Pero esta solución produce (y resuelve) los mismos subproblemas muchas veces, como ya vimos en el tema de *Divide y Vencerás*, y como consecuencia el coste temporal es exponencial.
- El coste espacial de esta solución será de orden lineal, ya que el número de llamadas almacenadas en la pila del sistema por la rama izquierda del árbol llegará a ser exactamente n (por la otra rama serán $n/2$).

Programación dinámica: un ejemplo

Fibonacci - versión recursiva: árbol de llamadas



Programación dinámica: un ejemplo

- Para no repetir cálculos, podemos almacenarlos en una tabla:

```
# t se define como un diccionario y se inicializa t={0:1, 1:1}
def fib_rec_mem(n):
    global t
    if not n in t:
        t[n]=fib_rec_mem(n-1)+fib_rec_mem(n-2)
    return t[n]
```

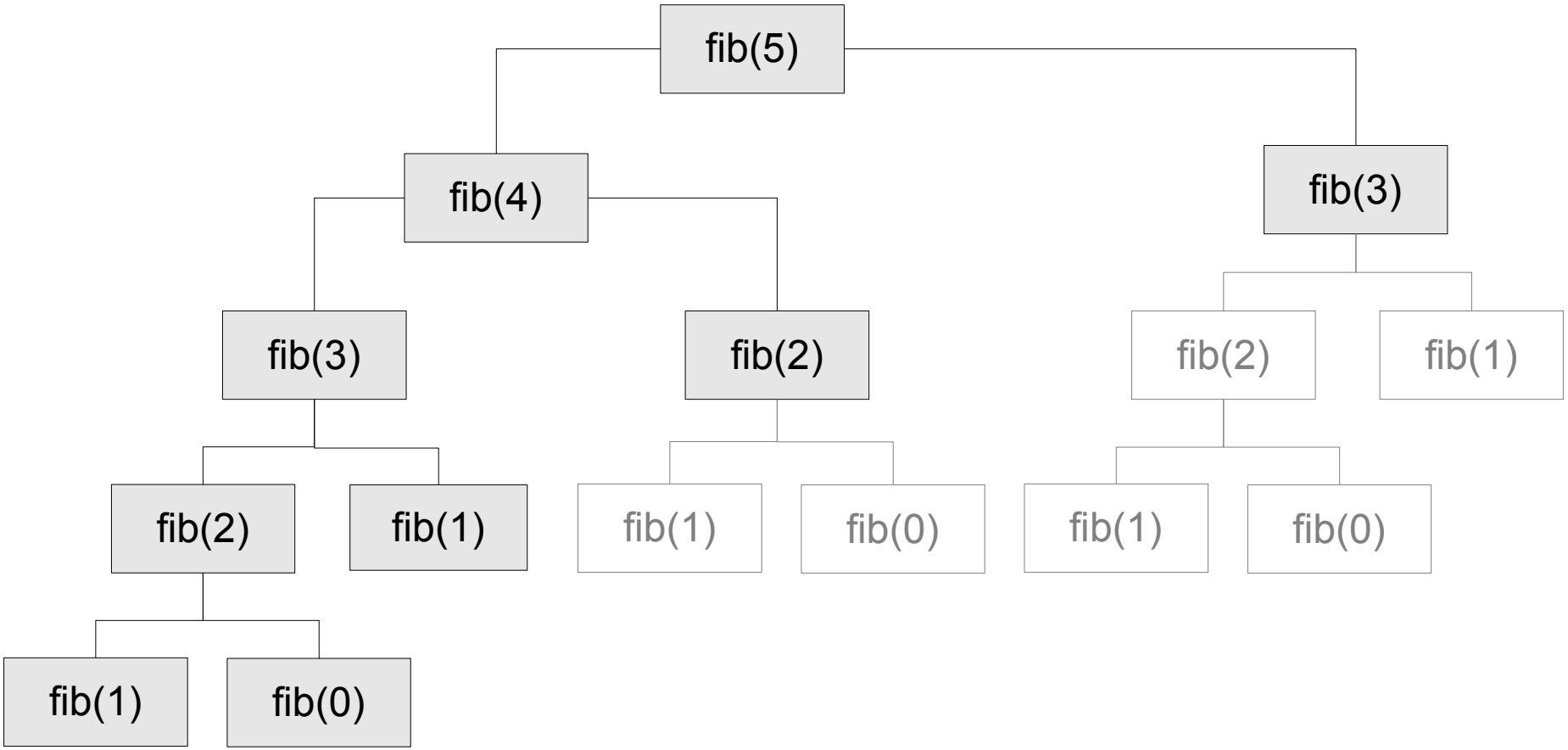
- Esta solución calculará cada término una sola vez, de modo que el coste temporal será lineal. Nótese que el coste espacial también será lineal.
- Sin embargo, la versión anterior sigue siendo recursiva y encajaría en lo que hemos llamado *top-down dynamic programming*. Los algoritmos de programación dinámica son iterativos y operan de abajo arriba. En este caso, a partir del caso base iremos calculando y almacenando los términos sucesivos de la serie:

```
def fib_ite(n):
    t={0:1, 1:1}
    for i in range(2,n+1):
        t[i]=t[i-1]+t[i-2]
    return t[n]
```

- Los costes temporal y espacial de esta solución siguen siendo lineales.

Programación dinámica: un ejemplo

Fibonacci - versión recursiva con memoria: árbol de llamadas



Programación dinámica: un ejemplo

- Sin embargo, como el término F_n depende sólo de F_{n-1} y F_{n-2}, no es necesario almacenar *toda* la tabla, sino tan sólo los dos últimos términos calculados:

```
def fib_ite(n):
    f2=1  # f2 es el termino F_{n-2}
    f1=1  # f1 es el termino F_{n-1}
    for i in range(2,n+1):
        f=f1+f2  # F_{n} = F_{n-1} + F_{n-2}
        f2=f1
        f1=f
    return f
```

- De esta forma, hemos llegado a un algoritmo de programación dinámica iterativo, de coste temporal lineal y coste espacial constante.

Programación dinámica: esquema de diseño

- Los algoritmos de programación dinámica tratan de resolver un problema de optimización, de modo que maximizarán (o minimizarán) el valor de una función objetivo que mide la bondad (o el coste) de una solución. Como resultado se obtendrá *una* solución óptima (podría haber varias).
- El proceso consta típicamente de 4 pasos:

  (1) Definir la función objetivo
  (2) Definir recursivamente (ecuación de Bellman) el valor óptimo de la función objetivo correspondiente a un problema en términos de los valores óptimos de dicha función para ciertos subproblemas
  (3) Calcular los valores óptimos de la función objetivo de abajo arriba, empezando por problemas elementales y aplicando la definición recursiva del paso (2) hasta llegar al problema planteado originalmente
  (4) Recuperar la solución óptima a partir de información almacenada durante el proceso de optimización

- El paso (4) puede omitirse si sólo interesa el valor óptimo de la función objetivo.
- **IMPORTANTE:** para reconstruir la solución óptima, será necesario guardar el camino seguido (la secuencia de decisiones) en el proceso de optimización.

El problema de la mochila (discreto)

Enunciado: Considerese una mochila capaz de albergar un peso máximo M, y n elementos con pesos $p_1, p_2, \ldots, p_n$ y beneficios $b_1, b_2, \ldots, b_n$. **Tanto M como los p_i serán enteros, lo que permitirá utilizarlos como índices en una tabla**. Se trata de encontrar qué combinación de elementos, representada mediante la tupla $x = (x_1, x_2, \ldots, x_n)$, con $x_i \in \{0, 1\}$, maximiza el beneficio:

$$f(x) = \sum_{i=1}^{n} b_i x_i \tag{1}$$

sin sobrepasar el peso máximo M:

$$\sum_{i=1}^{n} p_i x_i \leq M \tag{2}$$

- Empezaremos con una versión recursiva que, partiendo de un prefijo x de longitud k y un peso disponible r, devuelve una solución óptima junto con el máximo valor alcanzable de la función objetivo (1).
- Para ello, a partir del prefijo x, se explorarán las dos opciones posibles: añadir el objeto k (comprobando la condición (2)) o no añadirlo.

El problema de la mochila (discreto)

Versión recursiva de referencia

```
# Solucion y maximo beneficio alcanzable
# con los objetos 0..i-1, teniendo un peso m
# disponible en la mochila
def mochila_d_rec(i,m,p,b):
    # base de la recurrencia: 0 objetos
    if i==0:
        return [],0
    # opcion 1: el objeto i-1 NO se introduce
    sol_NO, max_b_NO = mochila_d_rec(i-1,m,p,b)
    # opcion 2: el objeto i-1 SI se introduce
    if p[i-1]<=m:
        sol_SI, max_b_SI = mochila_d_rec(i-1,m-p[i-1],p,b)
        if b[i-1] + max_b_SI > max_b_NO:
            return [1]+sol_SI, b[i-1]+max_b_SI
    return [0]+sol_NO, max_b_NO
```

El problema de la mochila (discreto)

Versión de programación dinámica (1)

El beneficio acumulado de la mejor solución para cada caso (i, m) se almacena en una matriz t, de tamaño $(n+1) \times (M+1)$

```
# Devuelve el maximo beneficio alcanzable con los objetos
# de pesos p y beneficios b, teniendo un peso M disponible
def mochila_d_pd1(p,b,M):
    n=len(p)
    t=[[0 for m in range(M+1)] for i in range(n+1)]
    for i in range(1,n+1):
        for m in range(1,M+1):
            # si se puede introducir el objeto i y el beneficio
            # es mayor que no haciendolo, lo introducimos
            if p[i-1]<=m and b[i-1]+t[i-1][m-p[i-1]]>t[i-1][m]:
                t[i][m]=b[i-1]+t[i-1][m-p[i-1]]
            # en caso contrario, no lo introducimos
            else:
                t[i][m]=t[i-1][m]
    return t[n][M]
```


El problema de la mochila (discreto)

Versión de programación dinámica (1) - Ejemplo

p = [2, 5, 3, 6, 1]

b = [28, 33, 5, 12, 20]

i \ m	0	1	2	3	4	5	6	7	8	9	10
0	0	0	**0**	0	0	0	0	0	0	0	0
1	0	0	28	28	**28**	28	28	28	28	28	28
2	0	0	28	28	28	33	33	61	61	**61**	61
3	0	0	28	28	28	33	33	61	61	**61**	66
4	0	0	28	28	28	33	33	61	61	**61**	66
5	0	20	28	48	48	48	53	61	81	81	**81**

El problema de la mochila (discreto)

Versión de programación dinámica (2)

El procedimiento sólo requiere dos filas de la matriz: la actual y la anterior, puesto que la optimización de la fila i sólo depende de la fila $i-1$.

```
# Devuelve el maximo beneficio alcanzable con los objetos
# de pesos p y beneficios b, teniendo un peso M disponible
def mochila_d_pd2(p,b,M):
    n=len(p)
    ant=[0 for m in range(M+1)]
    act=[0 for m in range(M+1)]
    for i in range(1,n+1):
        for m in range(1,M+1):
            # si se puede introducir el objeto i y el beneficio
            # es mayor que no haciendolo, lo introducimos
            if p[i-1]<=m and b[i-1]+ant[m-p[i-1]]>ant[m]:
                act[m]=b[i-1]+ant[m-p[i-1]]
            # en caso contrario, no lo introducimos
            else:
                act[m]=ant[m]
        ant=act[:]
    return act[M]
```

El problema de la mochila (discreto)

Versión de programación dinámica (3)

Recuperación de la solución: una matriz d guarda las decisiones tomadas; se retrocede fila a fila desde el elemento $d[n][M]$ hasta la fila 0

```
def mochila_d_pd3(p,b,M):
    n=len(p)
    ant=[0 for m in range(M+1)]
    act=[0 for m in range(M+1)]
    d=[[0 for m in range(M+1)] for i in range(n+1)]
    for i in range(1,n+1):
        for m in range(1,M+1):
            if p[i-1]<=m and b[i-1]+ant[m-p[i-1]]>ant[m]:
                act[m]=b[i-1]+ant[m-p[i-1]]
                d[i][m]=1
            else:
                act[m]=ant[m]
                d[i][m]=0
        ant=act[:]
    x=[]
    m=M
    for i in range(n,0,-1):
        x.insert(0,d[i][m])
        if d[i][m]==1:
            m=m-p[i-1]
    return x, act[M]
```


El problema de la mochila (discreto)

Versión de programación dinámica (4)

Recuperación de la solución: los vectores ant y act almacenan el beneficio máximo alcanzable junto con la lista de decisiones correspondiente. Al terminar, el elemento act[M] contiene el beneficio máximo alcanzable y la solución.

```
# Devuelve el maximo beneficio alcanzable con los objetos
# de pesos p y beneficios b, teniendo un peso M disponible,
# asi como la solucion que lo produce
def mochila_d_pd4(p,b,M):
    n=len(p)
    ant=[[0,[]] for m in range(M+1)]
    for i in range(1,n+1):
        act=[[0,[0 for j in range(i)]]]
        for m in range(1,M+1):
            if p[i-1]<=m and b[i-1]+ant[m-p[i-1]][0]>ant[m][0]:
                act.append([b[i-1]+ant[m-p[i-1]][0],ant[m-p[i-1]][1][:]+[1]])
            else:
                act.append([ant[m][0],ant[m][1][:]+[0]])
        ant=act
    return act[M]
```

El problema de la mochila (discreto)

Demostración del principio de optimalidad (1)

- Sea $x = (x_1, x_2, \ldots, x_n)$ la solución óptima del problema $(1, n, M)$, con $x_i \in \{0, 1\} \quad \forall i$
- Vamos a demostrar que cualquier subsecuencia $x' = (x_i, \ldots, x_j)$ de x es también óptima para el subproblema asociado $(i, j, M - K)$, donde K es el peso aportado a la solución óptima por los objetos $1, \ldots, i-1$ y $j+1, \ldots, n$:

$$K = \sum_{k=1}^{i-1} x_k p_k + \sum_{k=j+1}^{n} x_k p_k \tag{3}$$

- Lo haremos por reducción al absurdo. Si x' no es solución óptima para el subproblema $(i, j, M - K)$, entonces existirá otra subsecuencia diferente $y' = (y_i, \ldots, y_j)$ tal que:

$$\sum_{k=i}^{j} y_k b_k > \sum_{k=i}^{i} x_k b_k \tag{4}$$

con la restricción:

$$\sum_{k=i}^{j} y_k p_k \leq M - K$$

El problema de la mochila (discreto)

Demostración del principio de optimalidad (2)

- Si en esta última expresión sustituimos el valor de K por (3), nos queda:

$$\sum_{k=1}^{i-1} x_k p_k + \sum_{k=i}^{j} y_k p_k + \sum_{k=j+1}^{n} x_k p_k \leq M$$

- Y por otra parte (por la desigualdad (4)):

$$\sum_{k=1}^{i-1} x_k b_k + \sum_{k=i}^{j} y_k b_k + \sum_{k=j+1}^{n} x_k b_k > \sum_{k=1}^{n} x_k b_k$$

- Esto implicaría que la secuencia $y = (x_1, \ldots, x_{i-1}, y_i, \ldots, y_j, x_{j+1}, \ldots, x_n)$ sería la solución óptima al problema $(1, n, M)$, lo cual contradice nuestra hipótesis de partida. Así pues, cualquier subsecuencia x' de la solución óptima x debe ser asimismo solución óptima del subproblema asociado (principio de optimalidad).

Devolver el cambio con el número mínimo de monedas

Enunciado: Considerese un repositorio infinito de monedas, con n monedas distintas, de valores $v_1, v_2, \ldots, v_n$. El problema del cambio consiste en descomponer una cierta cantidad a devolver M (entera) en el menor número posible de monedas.

- Como se ha visto en un tema anterior, existe un algoritmo voraz que permite obtener la solución óptima al problema siempre que los valores sean de la forma $1, p, p^2, p^3, \ldots, p^n$, con $p > 1$ y $n > 0$.
- Ahora planteamos un algoritmo completamente general, según un esquema de programación dinámica (de abajo arriba), que expresa la solución óptima para una cantidad y un conjunto de monedas en términos de las soluciones óptimas para cantidades y/o conjuntos de monedas más pequeños (puesto que se cumple el principio de optimalidad).
- Además, el procedimiento obtendrá la solución óptima cualquiera que sea el orden de los valores de las monedas.

Devolver el cambio con el número mínimo de monedas

- Dado un repositorio infinito de monedas de valores $V = \{v_0, v_1, \ldots, v_{n-1}\}$, tales que $v_i > 0\ \forall i$ y $v_i \neq v_j\ \forall i \neq j$, y suponiendo que ha de devolverse una cantidad entera $M \geq 0$, construiremos una tabla c de tamaño $n \times (M+1)$ con el mínimo número de monedas necesario para resolver cada subproblema (i, m).
- El índice i indicará que se están usando las monedas $\{v_0, v_1, \ldots, v_i\}$, mientras que el índice $m \in [0, M]$ indicará la cantidad a devolver.
- Por razones obvias, los $c[i][0]$ se inicializan a 0. Los demás $c[i][m]$ se inicializan a `None`, indicando que no existe solución (de momento) para dichos subproblemas.
- Primero se calculan los $c[0][m]$, para $m = 1, 2, \ldots, M$, como sigue:

$$\begin{array}{rcl} m >= v_0 \text{ y } c[0][m - v_0] \neq \texttt{None} & \Rightarrow & c[0][m] = 1 + c[0][m - v_0] \\ \text{en caso contrario} & \Rightarrow & c[0][m] = \texttt{None} \end{array}$$

- A continuación se calculan los $c[i][m]$, para $i = 1, \ldots, n-1$ y (una vez fijado i) para $m = 1, \ldots, M$, como sigue:

$m < v_i$			$c[i][m] = c[i-1][m]$
$m >= v_i$	$c[i][m - v_i] = \texttt{None}$		$c[i][m] = c[i-1][m]$
	$c[i][m - v_i] \neq \texttt{None}$	$c[i-1][m] = \texttt{None}$	$c[i][m] = 1 + c[i][m - v_i]$
		$c[i-1][m] \neq \texttt{None}$	$c[i][m] = \min(c[i-1][m], 1 + c[i][m - v_i])$

- Finalmente, la solución al problema original se encontrará en $c[n-1][M]$.

Devolver el cambio con el número mínimo de monedas

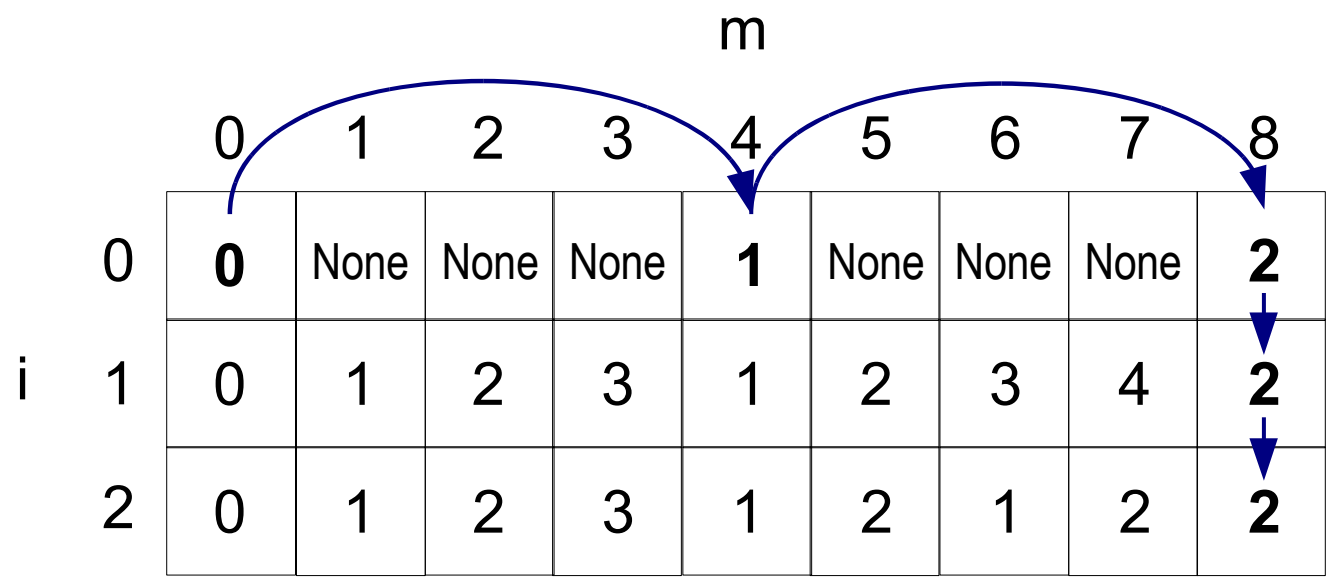
```
# monedas: vector con los valores de las monedas (distintas)
# M: valor a devolver
# devuelve el numero minimo de monedas necesario (o None)
def devolver_cambio(monedas,M):
    n=len(monedas)  # numero de monedas
    c=[[None for j in range(M+1)] for i in range(n)]
    c[0][0]=0
    for j in range(1,M+1):
        if j>=monedas[0] and c[0][j-monedas[0]]!=None:
            c[0][j]=1+c[0][j-monedas[0]]
    for i in range(1,n):
        c[i][0]=0
        for j in range(1,M+1):
            if j<monedas[i] or c[i][j-monedas[i]]==None:
                c[i][j]=c[i-1][j]
            elif c[i-1][j]!=None:
                c[i][j]=min(c[i-1][j],1+c[i][j-monedas[i]])
            else:
                c[i][j]=1+c[i][j-monedas[i]]
    return c[n-1][M]
```

Devolver el cambio con el número mínimo de monedas

Devolver el cambio - Ejemplo

monedas = [4, 1, 6]

devolver_cambio (monedas,8)



Devolver el cambio con el número mínimo de monedas

```
# Devuelve la solucion (o None, si no existe)
def devolver_cambio(monedas,M):
    n=len(monedas)  # numero de monedas distintas
    c=[[None for j in range(M+1)] for i in range(n)]
    c[0][0]=0
    for j in range(1,M+1):
        if j>=monedas[0] and c[0][j-monedas[0]]!=None:
            c[0][j]=1+c[0][j-monedas[0]]
    for i in range(1,n):
        c[i][0]=0
        for j in range(1,M+1):
            if j<monedas[i] or c[i][j-monedas[i]]==None:
                c[i][j]=c[i-1][j]
            elif c[i-1][j]!=None:
                c[i][j]=min(c[i-1][j],1+c[i][j-monedas[i]])
            else:
                c[i][j]=1+c[i][j-monedas[i]]
    if c[n-1][M]==None:
        return None
    x=[0 for i in range(n)] # RECUPERACION DE LA SOLUCION
    i=n-1
    j=M
    while i!=0 or j!=0:
        if i==0 or c[i][j]!=c[i-1][j]:
            x[i]+=1
            j-=monedas[i]
        else:
            i=i-1
    return x
```

Parentizado óptimo en la multiplicación de matrices

- Cada elemento de la matriz C, producto de dos matrices A y B, de tamaños $p \times q$ y $q \times r$, respectivamente, está dado por:

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj} \qquad i \in [1, p], \quad j \in [1, r]$$

- La expresión anterior implica $p \cdot q \cdot r$ multiplicaciones escalares.
- Considerese ahora la **multiplicación encadenada de n matrices**:

$$M = M_1 M_2 \ldots M_n$$

- La propiedad asociativa hace que este producto se pueda calcular de muchas formas (según donde se pongan los paréntesis) y el número de multiplicaciones escalares es distinto en cada caso. **Se trata de hacerlo con el mínimo número de multiplicaciones escalares posible.**

Parentizado óptimo en la multiplicación de matrices

- Por ejemplo, considerense las 4 matrices: A (13×5), B (5×89), C (89×3) y D (3×34). Hay 5 parentizados distintos, que implican distinto número de multiplicaciones escalares:
  - $((AB)C)D$: 10.582
  - $(AB)(CD)$: 54.201
  - $(A(BC))D$: 2.856
  - $A((BC)D)$: 4.055
  - $A(B(CD))$: 26.418
- Nótese que el mejor parentizado es casi 20 veces más rápido que el peor.
- El número de parentizados distintos de un producto de n matrices viene dado por la siguiente recurrencia:

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i), \qquad \text{con } T(1) = 1$$

Parentizado óptimo en la multiplicación de matrices

- $T(n)$ da lugar a los llamados *números de Catalan*, cuyos primeros valores son 1, 1, 2, 5, 14, 42, 132, 429, 1.430, 4.862, 16.796, 58.786, 208.012, etc.
- Explorar todos los posibles parentizados y quedarse con el que produzca menos productos no es computacionalmente viable. Tampoco existe un algoritmo voraz que produzca el parentizado óptimo.
- Sin embargo, **sí se cumple el principio de optimalidad**: supongamos que el mejor parentizado del producto $M_1 M_2 \ldots M_n$ implica partir el producto entre las posiciones i e $i+1$, entonces los parentizados que se obtengan para los productos $M_1 M_2 \ldots M_i$ y $M_{i+1} M_{i+2} \ldots M_n$ deberán ser asimismo soluciones óptimas de los subproblemas asociados.
- Esto sugiere una solución de programación dinámica que explore todos los posibles puntos de inserción del primer paréntesis (el de nivel más alto) y escoja aquel que conduzca a un número menor de productos escalares (usando una tabla con los valores óptimos previamente calculados para problemas más pequeños).
- El número de posibles puntos de inserción es lineal con n, así que el algoritmo de exploración es computacionalmente viable.

Parentizado óptimo en la multiplicación de matrices

- Se define una tabla m, de tamaño $n \times n$, tal que cada elemento m_{ij} almacena el mínimo número de multiplicaciones escalares necesario para realizar el subproducto $M_i M_{i+1} \ldots M_j$. La solución del problema planteado vendrá dada, por tanto, por el elemento m_{1n}.
- Las dimensiones de las matrices $M_1, \ldots, M_n$ se almacenan en un vector d, indexado de 0 a n, tal que la matriz M_i tendrá dimensiones $d_{i-1} \times d_i$.
- La matriz m se construye diagonal a diagonal: la diagonal s ($s \in [0, n-1]$) corresponde a los elementos m_{ij} tales que $j - i = s$.
- Los elementos de la diagonal $s = 0$ corresponden a *no productos*, pues sólo incluyen una matriz M_i, y será por tanto $m_{ii} = 0 \quad \forall i$.
- Los elementos de la diagonal $s = 1$ corresponden a productos de la forma $M_i M_{i+1}$, con un solo parentizado posible, de modo que $m_{i,i+1} = d_{i-1} d_i d_{i+1}$.
- Por último, los elementos $m_{i,i+s}$ de las diagonales $s > 1$ corresponden a productos de la forma $M_i M_{i+1} \ldots M_{i+s}$; en estos casos, el primer paréntesis se puede colocar en $s - 1$ sitios distintos: entre las matrices M_k y M_{k+1}, para $k = i, i+1, \ldots, i+s-1$, lo que implica $m_{ik} + m_{k+1,i+s}$ multiplicaciones escalares (calculadas previamente) más las $d_{i-1} d_k d_{i+s}$ correspondientes al producto de las dos matrices resultantes.
- El valor óptimo de $m_{i,i+s}$ se obtiene escogiendo el k que minimice el número de productos escalares.

Parentizado óptimo en la multiplicación de matrices

```
# Dentro de la matriz m, la fila 0 y la columna 0 no se usan
# La matriz m es triangular superior: la diagonal principal
# esta llena de ceros y corresponde a s=0, la siguiente a s=1, etc.
# hasta la ultima (con un solo elemento), que corresponde a s=n-1
def orden_optimo_producto(d):
    n=len(d)-1
    m=[[0 for i in range(n+1)] for i in range(n+1)]
    # recorremos las diagonales, desde s=1 hasta s=n-1
    for s in range(1,n):
        for i in range(1,n-s+1):
            # optimizacion del parentizado para m[i][i+s]
            for k in range(i,i+s):
                n_prod = m[i][k] + m[k+1][i+s] + d[i-1]*d[k]*d[i+s]
                if k==i or n_prod < minimo:
                    minimo = n_prod
            m[i][i+s]=minimo
    return m[1][n]
```

Parentizado óptimo en la multiplicación de matrices

Parentizado óptimo - Ejemplo

d = [13, 5, 89, 3, 34]

orden_optimo_producto (d)

	0	1	2	3	4
0	0	0	0	0	0
1	0	0 s=0	5785 s=1	1530 s=2	2856 s=3
2	0	0	0	1335	1845
3	0	0	0	0	9078
4	0	0	0	0	0

Caminos de coste mínimo en un grafo ponderado

Enunciado: Sea $G = (V, E, c)$ un grafo **dirigido** y ponderado. Se trata de encontrar el camino de coste mínimo entre cualquier par de vértices v_i y v_j de G.

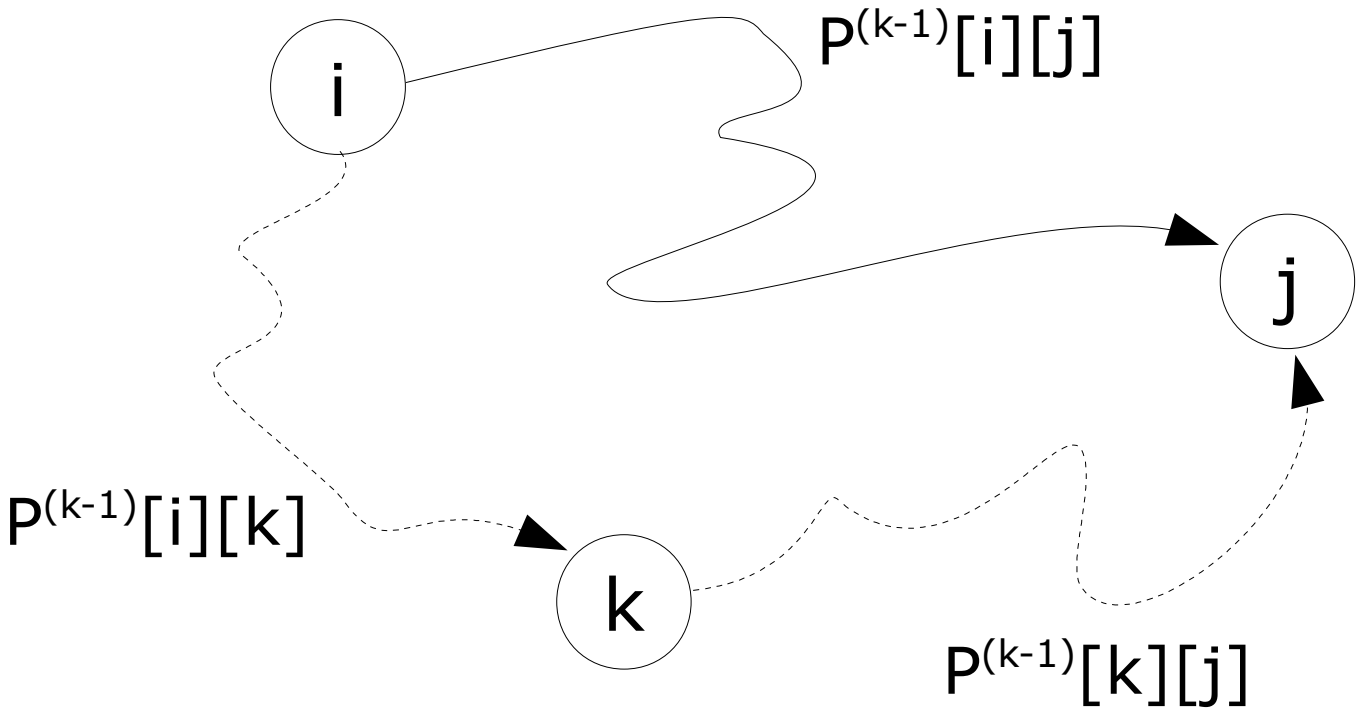
Solución: algoritmo de Floyd-Warshall

- El algoritmo iterará sobre los nodos del grafo, considerando cada uno de ellos como posible nodo intermedio en el camino óptimo de v_i a v_j. El nodo intermedio se identificará con el índice k, para $k = 1, 2, \ldots, n$ (n: número de nodos de G).
- Se irá generando una secuencia de matrices $P^{(k)}$ de tamaño $n \times n$, que almacenarán la solución óptima tras haber considerado los nodos $1, 2, \ldots, k$ como posibles nodos intermedios.
- De acuerdo a lo anterior, $P^{(0)} = c$. Por convenio, el coste de los arcos que no existen en el grafo G será infinito (∞). Además, el coste del camino directo de un nodo a sí mismo será 0.
- A continuación, cada matriz $P^{(k)}$, con $k \in [1, n]$, se calcula a partir de la matriz $P^{(k-1)}$ aplicando el siguiente criterio de optimización:

$$P^{(k)}[i][j] = \text{mín}(P^{(k-1)}[i][j], P^{(k-1)}[i][k] + P^{(k-1)}[k][j]) \qquad (5)$$

Caminos de coste mínimo en un grafo ponderado

Algoritmo de Floyd-Warshall



Caminos de coste mínimo en un grafo ponderado

- En cada iteración k sólo se utilizan la fila k y la columna k de $P^{(k-1)}$, que además no varían tras el proceso de optimización:

$$P^{(k)}[k][j] = \text{mín}(P^{(k-1)}[k][j], P^{(k-1)}[k][k] + P^{(k-1)}[k][j]) = P^{(k-1)}[k][j]$$

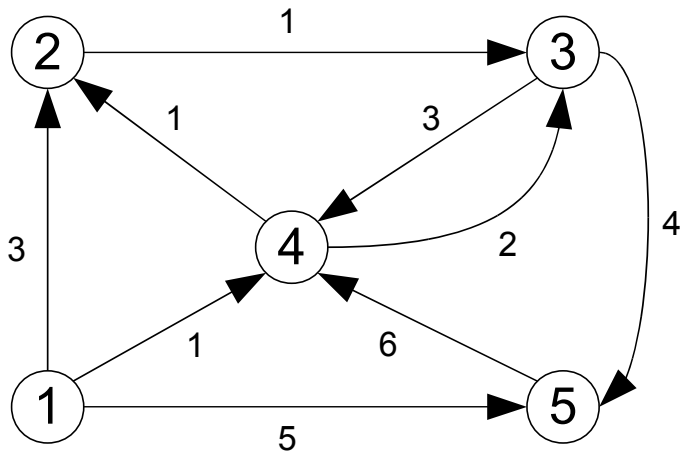
$$P^{(k)}[i][k] = \text{mín}(P^{(k-1)}[i][k], P^{(k-1)}[i][k] + P^{(k-1)}[k][k]) = P^{(k-1)}[i][k]$$

  puesto que $P^{(k-1)}[k][k] = 0$.

- **Esto permite realizar el proceso de optimización sobre una única matriz.**
  Es decir, no es necesario crear una serie de matrices, sino tan sólo inicializar la matriz P con la matriz de costes de los caminos directos (c) y a continuación ir actualizándola paso a paso según la ecuación (5).

Caminos de coste mínimo en un grafo ponderado

Ejemplo



$$P^{(0)} = \begin{pmatrix} 0 & 3 & \infty & 1 & 5 \\ \infty & 0 & 1 & \infty & \infty \\ \infty & \infty & 0 & 3 & 4 \\ \infty & 1 & 2 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad P^{(1)} = \begin{pmatrix} 0 & 3 & \infty & 1 & 5 \\ \infty & 0 & 1 & \infty & \infty \\ \infty & \infty & 0 & 3 & 4 \\ \infty & 1 & 2 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$P^{(2)} = \begin{pmatrix} 0 & 3 & 4 & 1 & 5 \\ \infty & 0 & 1 & \infty & \infty \\ \infty & \infty & 0 & 3 & 4 \\ \infty & 1 & 2 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad P^{(3)} = \begin{pmatrix} 0 & 3 & 4 & 1 & 5 \\ \infty & 0 & 1 & 4 & 5 \\ \infty & \infty & 0 & 3 & 4 \\ \infty & 1 & 2 & 0 & 6 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$P^{(4)} = \begin{pmatrix} 0 & 2 & 3 & 1 & 5 \\ \infty & 0 & 1 & 4 & 5 \\ \infty & 4 & 0 & 3 & 4 \\ \infty & 1 & 2 & 0 & 6 \\ \infty & 7 & 8 & 6 & 0 \end{pmatrix} \quad P^{(5)} = \begin{pmatrix} 0 & 2 & 3 & 1 & 5 \\ \infty & 0 & 1 & 4 & 5 \\ \infty & 4 & 0 & 3 & 4 \\ \infty & 1 & 2 & 0 & 6 \\ \infty & 7 & 8 & 6 & 0 \end{pmatrix}$$

Caminos de coste mínimo en un grafo ponderado

Implementación en lenguaje Python

```
# n: numero de nodos en el grafo (indexados de 0 a n-1)
# c: diccionario de costes de los caminos directos
#    indexado por tuplas (i,j); solo aparecen los
#    costes de los arcos existentes; None representa
#    un coste infinito
def floyd(c,n):
    # inicializacion
    p=[[None if j!=i else 0 for j in range(n)] for i in range(n)]
    for i,j in c:
        p[i][j]=c[(i,j)]
    # optimizacion para k=0,1,...,n-1
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if p[i][k]!=None and p[k][j]!=None:
                    if p[i][j]==None or p[i][j]>p[i][k]+p[k][j]:
                        p[i][j]=p[i][k]+p[k][j]
    return p
```