

Algoritmia
Grado en Ingeniería Informática del Software
Escuela de Ingeniería Informática – Universidad de Oviedo

Programación dinámica

Juan Ramón Pérez Pérez

jrpp@uniovi.es



¿Quién va a ganar el Play-off de la liga ACB?

Play-off de baloncesto (I)

Dos equipos A y B disputan una eliminatoria de la fase final de la liga de baloncesto.

Jugarán $2n-1$ partidos y el ganador será el primer equipo que consiga n victorias.

El equipo **A** tiene una probabilidad constante p de ganar un partido, mientras que **B** tiene una probabilidad q de ganar ($q = 1 - p$).

Queremos conocer la probabilidad de que uno de estos equipos gane el play-off al principio de la eliminatoria y una vez jugados varios partidos.

Play-off de baloncesto (II)

$P(i,j)$, que se define como la probabilidad de que **A** gane la eliminatoria cuando le quedan por ganar i partidos a **A** y j partidos a **B**.

Se puede obtener fácilmente el valor de $P(i,j)$ según la expresión:

$$P(i, j) = \begin{cases} 1 & \text{si } i = 0 \text{ y } j > 0 \\ 0 & \text{si } i > 0 \text{ y } j = 0 \\ p * P(i-1, j) + q * P(i, j-1) & \text{si } i > 0 \text{ y } j > 0 \end{cases}$$

A partir de esta fórmula se puede obtener directamente un algoritmo con Divide y Vencerás.

Probabilidad de ganar play-offs con *Divide y Vencerás*

```
public double probabilidadDv(int i,int j)
{
    if (i==0)
        return 1.0;
    else
        if (j==0)
            return 0.0;
        else
            return probabilidadGanarA
                * probabilidadDv(i-1,j)
                + (1-probabilidadGanarA)
                * probabilidadDv(i,j-1);
}
```

Análisis algoritmo y mejora

Inconveniente: se están repitiendo cálculos innecesariamente.

La complejidad es exponencial, del orden de $O(2^{i+j})$ si $i+j=n$.

Para acelerar el algoritmo: declaramos una tabla del tamaño adecuado y vamos rellenando las entradas.

Problemas de *Divide y Vencerás*

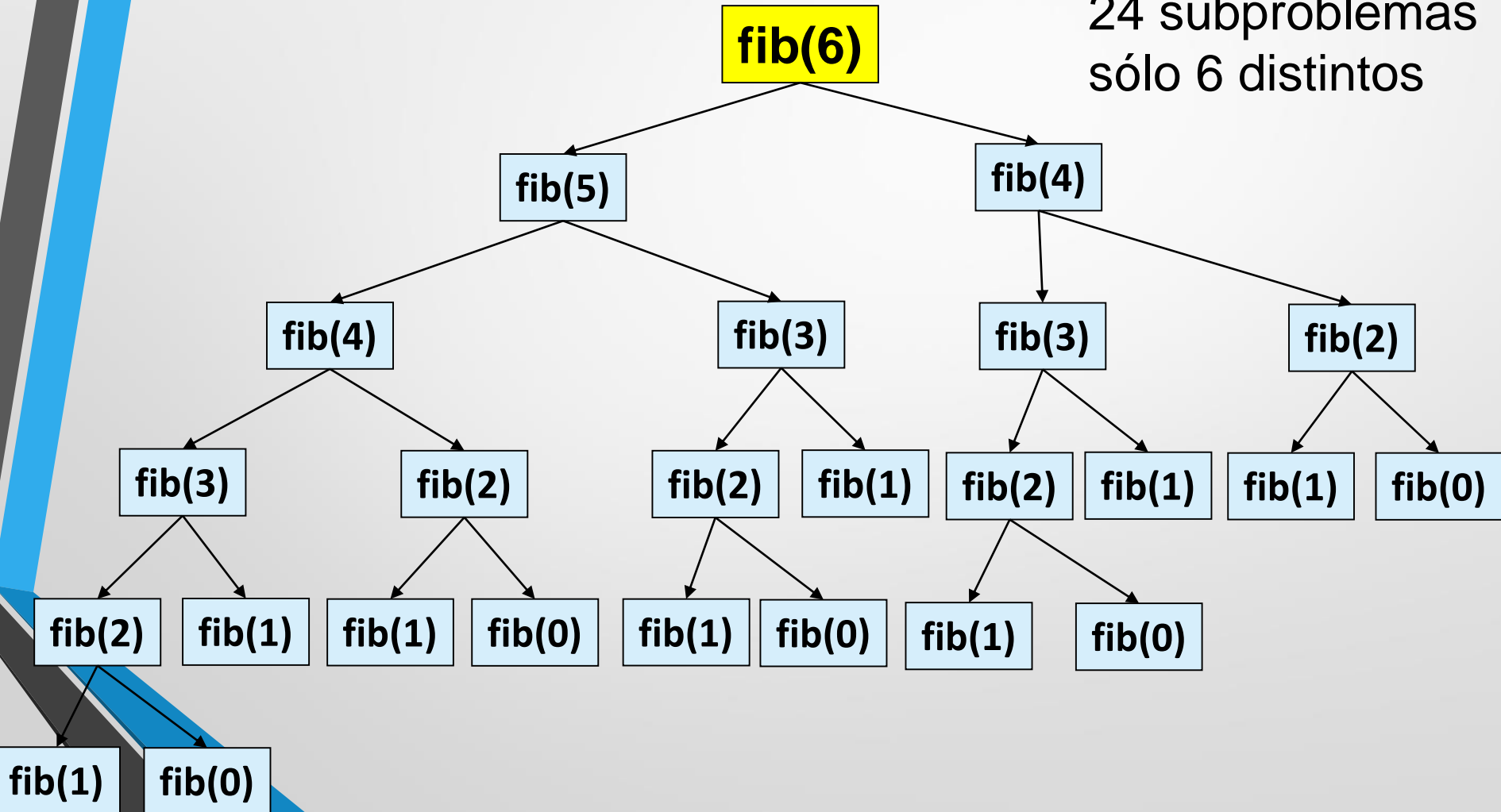
- Divide y vencerás → dividir los problemas en subproblemas y combinar las soluciones para resolver el problema original.
- Algunas veces, el diseño Divide y Vencerás fracasa porque descompone el problema inicial en un número muy alto de subproblemas (complejidad no polinómica).
- Además, puede darse el caso de que al aplicar Divide y Vencerás, se generan un cierto número de subproblemas **repetidos** que se solucionan varias veces.

Ejemplo de ineficiencia de *Divide y Vencerás*: Fibonacci.

```
public static long fibDyV (int n)
{
    if (n<=1)
        return n;
    else return fibDyV(n-1)+fibDyV(n-2);
}
```


Árbol de llamadas de la función de Fibonacci

Si $n=6$
24 subproblemas
sólo 6 distintos



Fibonacci con programación dinámica

```
public static long fibPD (int n)
{
    long[] f=new long[n+1]; // subsoluciones
    f[0]=0;f[1]=1;
    for (int i=2;i<=n;i++)
        f[i]=f[i-1]+f[i-2];
    return f[n];
}
```

Programación dinámica. Mejora de la eficiencia

- Cuando el número de problemas distintos es polinómico, podemos resolver cada subproblema una vez y **guardar su solución** para un uso posterior.
- La idea es **evitar calcular dos veces** el mismo subproblema, normalmente manteniendo una tabla de resultados conocidos.

Probabilidad de ganar play-offs con Programación dinámica

```
public double probabilidadPd(int i,int j)
{
    double[][] a= new double[i+1][j+1]; // necesitamos índices desde 0 hasta i

    for (int v= 1; v<j+1; v++) { // Rellenamos casos triviales
        a[0][v]= 1.0;
    }
    for (int u= 1; u<i+1; u++) {
        a[u][0]= 0.0;
    }

    for (int u= 1; u<i+1; u++) // Rellenamos el resto de las celdas
        for (int v= 1; v<j+1; v++) {
            a[u][v]= probabilidadGanarA * a[u-1][v]
                    + (1-probabilidadGanarA) * a[u][v-1];
        }
    return a[i][j];
}
```

Comparación *Divide y Vencerás* – Prog. dinámica

- Divide y vencerás es un método de *refinamiento progresivo (descendente)*:
 - Atacamos el caso completo que vamos dividiendo en subcasos más pequeños.
- Programación dinámica es una técnica *ascendente*:
 - Se empieza por los subcasos más pequeños y combinando las soluciones se obtienen respuestas a subcasos cada vez mayores, hasta que llegamos al caso completo.

Play-off de baloncesto con programación dinámica

$$P(i, j) = \begin{cases} 1 & \text{si } i = 0 \text{ y } j > 0 \\ 0 & \text{si } i > 0 \text{ y } j = 0 \\ p * P(i-1, j) + q * P(i, j-1) & \text{si } i > 0 \text{ y } j > 0 \end{cases}$$

se conocen de antemano ($a[u, 0]$ y $a[0, v]$) y calculando en cada pasada los elementos de la fila u .

Si tomamos $p=q=0.5$ y $n=3$ la tabla resultante sería la siguiente:

		a[0][v]			
i \ j		0	1	2	3
0			1	1	1
1		0	0,5	0,75	0,87
2		0	0,25	0,5	0.68
3		0	0,12	0.31	0.5

$$a[u, v] \rightarrow a[1, 1] = p_gane_A * a[u-1, v] + (1-p_gane_A) * a[u, v-1]$$

Análisis del algoritmo de programación dinámica

La complejidad de esta función no es en ningún caso exponencial como ocurría en DV.

Podríamos calcular la complejidad de este algoritmo de la siguiente forma:

$$T(n) = \sum_{s=1}^{i+j} (s-1) = \frac{i+j}{2} (i+j) = \frac{(i+j)^2}{2} \Rightarrow O((i+j)^2) \Rightarrow O(n^2)$$



Ejemplos de programación dinámica

Prog. dinámica. Aplicada a problemas de optimización

La mayor aplicación de la programación dinámica es en problemas de optimización, cuando fracasan otras técnicas.

Ejemplos:

- Algoritmo de Floyd-Warshall de búsqueda del camino mínimo entre dos nodos en grafos dirigidos ponderados

Principio de optimalidad de Bellman

Una secuencia óptima de decisiones que resuelve un problema

debe cumplir la propiedad de que

cualquier subsecuencia de decisiones, que tenga el mismo estado final, debe ser también óptima respecto al subproblema correspondiente.

Aplicación de este principio

- Si **siempre** se conoce cuál es la decisión que debe tomarse a continuación para obtener la secuencia óptima, el problema se resuelve con estrategia voraz.
- Principio de optimalidad, sigue siendo posible el ir tomando decisiones elementales, en la confianza de que la **combinación** de ellas seguirá siendo óptima, pero será entonces necesario **explorar muchas secuencias** de decisiones para dar con la correcta

Mochila 0/1



Mochila 0/1

- n objetos y una “mochila” para transportarlos.
- Para cada objeto $i = 1, 2, \dots, n$ tiene un peso w_i y un valor v_i .
- La mochila puede llevar un peso que no sobrepase W .
- Objetivo: maximizar valor de los objetos respetando la limitación de peso.
- Los objetos **no se pueden fragmentar**; tomamos un objeto completo o lo dejamos.

Datos de un problema concreto

- Número de objetos: $n=3$,
- Peso límite de la mochila: $W=10$

Objeto	1	2	3
w_i	6	5	5
v_i	8	5	5

Planteamiento con programación dinámica

- Tabla V
 - Filas: los objetos i .
 - Columnas: pesos máximos de la mochila.
- $V[i,j] \rightarrow$ valor máximo de los objetos que transportamos, si sólo incluimos hasta el objeto i , si el límite de peso es j .
- Solución a nuestro problema $V[n,W]$, $V[3,10]$

Función

- Función que permite calcular valores de la matriz:

$$V(i, j) = \begin{cases} -\infty & \text{si } j < 0 \\ 0 & \text{si } i = 0 \text{ y } j \geq 0 \\ \max(V(i-1, j), V(i-1, j-w_i) + v_i) & \text{en otro caso} \end{cases}$$

- i , es el número de objetos que probamos a meter en la mochila.
- j , es el peso máximo de la mochila

Programación dinámica. Mochila 0/1

Eliminación de valores directos de la tabla.

Para $n=3$ objetos, $W=10$ carga máxima.

Pesos máximos

Objetos		0	1	2	3	4	5	6	7	8	9	10
	1											
	2											
	3											

$V[i,j]$

$V[n,W]$

Programación dinámica. Mochila 0/1

Celda fuera de la tabla.

Para $n=3$ objetos, $W=10$ carga máxima.

Pesos máximos

Objetos	i \ j	Pesos máximos										
		0	1	2	3	4	5	6	7	8	9	10
	1	0	0	0								
	2	0	0	0								
	3	0	0									

Objet	1	2	3
w_i	6	5	5
v_i	8	5	5

$V[i,j]$ $\text{Max}(V(i-1,j), V(i-1,j-w_i)+v_i)$

Programación dinámica. Mochila 0/1

Celda dentro de la tabla

Para $n=3$ objetos, $W=10$ carga máxima.

Pesos máximos

Objetos	$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10
	1	0	0	0	0	0	0					
	2	0	0	0	0	0	5					
	3	0	0	0	0	0						

Objet	1	2	3
w_i	6	5	5
v_i	8	5	5

$V[i,j]$ $\text{Max}(V(i-1,j), V(i-1,j-w_i)+v_i)$

Programación dinámica. Mochila 0/1

Solución

Para $n=3$ objetos, $W=10$ carga máxima.

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	8	8	8	8	8
2	0	0	0	0	0	5	8	8	8	8	8
3	0	0	0	0	0	5	8	8	8	8	10

$$\text{Max}(V[i-1][j], V[i-1][j-w_i] + v_i)$$

$V[n, W]$

Ejercicio propuesto

Dado un sistema monetario compuesto por monedas de distinto valor, el **problema del cambio** consiste en descomponer en monedas cualquier cantidad dada, utilizando el menor número posible de monedas de dicho sistema monetario.

Siendo n el número de **tipos de monedas distintos**, j la **cantidad** que queremos descomponer y $T(1..n)$ un vector con el valor de cada tipo de moneda del sistema.

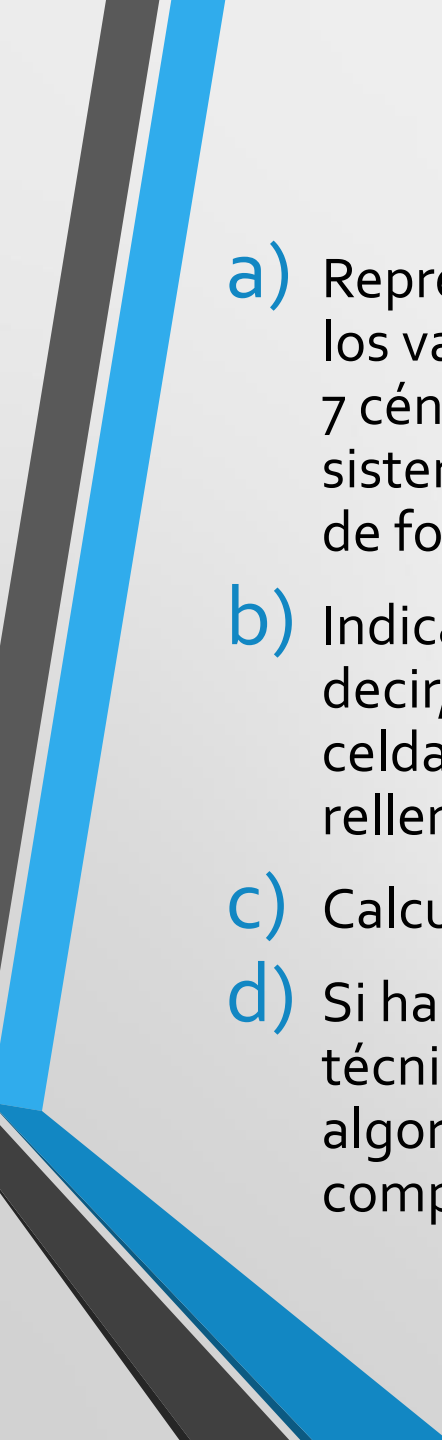
Suponemos que tenemos una cantidad suficiente de monedas de cada tipo.

Se debe calcular la cantidad mínima final de monedas para devolver dicha cantidad j planteada.

Función que resuelve el problema

- Disponemos de la función $C(i,j)$ que nos proporciona la solución al problema para obtener la cantidad j restringiéndose a los tipos T_1, T_2, \dots, T_i .

$$C(i, j) = \begin{cases} \infty & \text{si } (i = 0) \text{ ó } (j < 0) \\ 0 & \text{si } j = 0 \text{ y } i > 0 \\ \text{Min}(C(i-1, j), C(i, j-T_i) + 1) & \text{en otro caso} \end{cases}$$

- 
- a) Representar la tabla (sin rellenar) necesaria para almacenar los valores intermedios. Sabiendo que tenemos que devolver 7 céntimos con los cinco primeros tipos de monedas del sistema euro. Marcar en la tabla los valores podemos rellenar de forma directa.
 - b) Indicar el patrón de dependencia de una celda cualquiera, es decir, marcar las celdas que son necesarias para calcular una celda dada. Además, explicar de forma específica cómo rellenaríamos la celda (1,1).
 - c) Calcular los valores de la tabla.
 - d) Si hablamos de eficiencia en tiempo de ejecución, qué técnica deberíamos escoger: programación dinámica o un algoritmo voraz. Razona la respuesta planteando las complejidades medias.

Ejercicio propuesto 2

Camino Simple Menor (Floyd)

Se pide calcular el camino simple de **menor** coste entre un nodo origen y un nodo destino (origen \leftrightarrow destino) en un grafo dirigido de pesos positivos.

Camino Simple Mayor

Se pide calcular el camino simple de **mayor** coste entre un nodo origen y un nodo destino (origen \leftrightarrow destino) en un grafo dirigido de pesos positivos.