

# Análisis y diseño de algoritmos

## 4. Programación Dinámica

José Luis Verdú Mas, Jose Oncina,  
Mikel L. Forcada

Dep. Lenguajes y Sistemas Informáticos  
Universidad de Alicante

23 de marzo de 2020



- 1 Problema 1: la mochila 0/1
- 2 Problema 2: Corte de tubos
- 3 ¿Qué hemos aprendido con los problemas 1 y 2?
- 4 Problema 3: El coeficiente binomial
- 5 Programación dinámica: Estrategia de diseño

# Problema 1: la mochila 0/1

El problema de la mochila (Knapsack problem):



- Sean  $n$  objetos con valores ( $v_i \in \mathbb{R}$ ) y pesos ( $w_i \in \mathbb{R}^{>0}$ ) conocidos
- Sea una mochila con capacidad máxima de carga  $W$
- ¿Cuál es el valor máximo que puede transportar la mochila sin sobrepasar su capacidad?

- Un caso particular: **La mochila 0/1 con pesos discretos**
  - Los objetos no se pueden fraccionar (mochila 0/1 o mochila discreta)
    - La variante más difícil desde el punto de vista computacional
  - Los pesos son cantidades discretas o discretizables
    - Se utilizarán para indexar una tabla
    - Se trata de una versión menos general que suaviza su dificultad



# La mochila 0/1. Formalización matemática

- Es un problema de optimización:

- Secuencia de decisiones:  $(x_1, x_2 \dots x_n) : x_i \in \{0, 1\}, 1 \leq i \leq n$ 
  - En  $x_i$  se almacena la decisión sobre el objeto  $i$
  - Si  $x_i$  es escogido  $x_i = 1$ , en caso contrario  $x_i = 0$

- Una secuencia óptima de decisiones es la que maximiza  $\sum_{i=1}^n x_i v_i$

sujeto a las restricciones:

- $\sum_{i=1}^n x_i w_i \leq W$
  - $\forall i : 1 \leq i \leq n, x_i \in \{0, 1\}$
- Representamos mediante  $\text{knapsack}(i, C)$  al problema de la mochila con los objetos 1 hasta  $i$  y capacidad  $C$ 
  - El problema inicial es, por tanto,  $\text{knapsack}(n, W)$



# La mochila 0/1. Subestructura óptima

- Sea  $(x_1, x_2 \dots x_n)$  una secuencia óptima de decisiones para el problema  $\text{knapsack}(n, W)$ 
  - Si  $x_n = 0$  entonces  $(x_1 \dots x_{n-1})$  es una secuencia óptima para el subproblema  $\text{knapsack}(n-1, W)$
  - Si  $x_n = 1$  entonces  $(x_1 \dots x_{n-1})$  es una secuencia óptima para el subproblema  $\text{knapsack}(n-1, W - w_n)$

## Demostración:

Si existiera una solución mejor  $(x'_1 \dots x'_{n-1})$  para cada uno de los subproblemas entonces la secuencia  $(x'_1, x'_2 \dots x_n)$  sería mejor que  $(x_1, x_2 \dots x_n)$  para el problema original lo que contradice la suposición inicial de que era la óptima.<sup>a</sup>

---

<sup>a</sup>Este tipo de demostraciones se denominan “cut and paste”

⇒ **La solución al problema presenta una subestructura óptima**

- Es decir, la subestructura de los subproblemas puede ser usada para encontrar la solución óptima de el problema completo.



# La mochila 0/1. Aproximación matemática

- Se toman decisiones en orden descendente:  $x_n, x_{n-1}, \dots, x_1$
- Ante la decisión  $x_i$  hay dos alternativas:
  - Rechazar el objeto  $i$ :  $x_i = 0$ 
    - No hay ganancia adicional pero la capacidad de la mochila no se reduce
  - Seleccionar el objeto  $i$ :  $x_i = 1$ 
    - La ganancia adicional es  $v_i$ , a costa de reducir la capacidad en  $w_i$
- Se selecciona la alternativa que mayor ganancia global resulte
  - No se sabrá mientras no se analicen todas las posibilidades

## Solución

$$\{ W \geq 0, n > 0 \}$$

$$\text{knapsack}(0, W) = 0$$

$$\text{knapsack}(n, W) = \max \begin{cases} \text{knapsack}(n-1, W) \\ \text{knapsack}(n-1, W - w_n) + v_n & \text{if } W \geq w_n \end{cases}$$

# La mochila 0/1. Solución recursiva ineficiente (ingenua)

## Solución recursiva (ineficiente)

```
1 #include <limits>
2
3 double knapsack(
4     const vector<double> &v,      // values
5     const vector<unsigned> &w,    // weights
6     int n,                        // number of objects
7     unsigned W                    // knapsack weight limit
8 ) {
9     if( n == 0 )                  // base case
10         return 0;
11
12     double S1 = knapsack( v, w, n-1, W );    // try not to put it on
13
14     double S2 = numeric_limits<double>::lowest();
15     if( w[n-1] <= W )                // does it fits in the knapsack?
16         S2 = v[n-1] + knapsack( v, w, n-1, W-w[n-1] ); // try to put it on
17
18     return max( S1, S2 );            // choose the best
19 }
```

# La mochila 0/1. Coste temporal de la versión ingenua

- Caso mejor: ningún objeto cabe en la mochila:  $T(n) \in \Omega(n)$
- Caso peor:

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 + 2T(n-1) & \text{en otro caso} \end{cases}$$

El término general queda como:

$$T(n) = 2^i - 1 + 2^i T(n-i)$$

Que terminará cuando  $n-i=0$ , o sea:

$$T(n) = 2^n - 1 + 2^n \in O(2^n)$$

Pero ... ¡solo pueden haber  $(n+1)(W+1)$  problemas distintos!

$$\text{mochila}(\{0, 1, \dots, n\}, \{0, 1, \dots, W\})$$





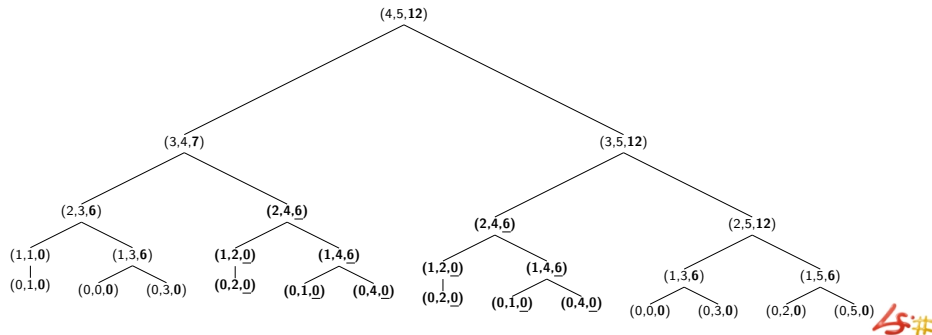
# La mochila 0/1. Subproblemas repetidos

En efecto, se resuelven muchos subproblemas iguales:

$$n = 4, \quad W = 5$$

- Ejemplo:  $w = (3, 2, 1, 1)$   
 $v = (6, 6, 2, 1)$

Nodos:  $(i, W, \text{Mochila}(i, W))$ ; izquierda,  $x_i = 1$ ; derecha,  $x_i = 0$ .



# La mochila 0/1. Solución recursiva con almacén

## Solución recursiva eficiente guardando resultados parciales (= memoización)

```
1 const int SENTINEL = -1;
2 double knapsack(
3     vector< vector< double >> &M,                                // Storage
4     const vector<double> &v, const vector<unsigned> &w,        // values & weights
5     int n, unsigned W                                           // num. of objects & Knapsack limit
6 ) {
7     if( M[n][W] != SENTINEL ) return M[n][W];                  // if it is known ...
8     if( n == 0 ) return M[n][W] = 0.0;                         // base case
9
10    double S1 = knapsack( M, v, w, n-1, W );
11    double S2 = numeric_limits<double>::lowest();
12    if (w[n-1] <= W) S2 = v[n-1] + knapsack( M, v, w, n-1, W - w[n-1]);
13    return M[n][W] = max( S1, S2 );                             // store and return the solution
14 }
15 //-----
16 double knapsack(
17     const vector<double> &v, const vector<unsigned> &w, int n, unsigned W
18 ) {
19     vector< vector< double >> M( n+1, vector<double>( W+1, SENTINEL)); // init.
20     return knapsack( M, v, w, n, W );
21 }
```



# La mochila 0/1. Memoización

- Ejemplo: Sean  $n = 5$  objetos con pesos ( $w_i$ ) y valores ( $v_i$ ) indicados en la tabla. Sea  $W = 11$  el peso máximo de la mochila.

$M[6][12]$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0		0		0	
$w_1 = 2, v_1 = 1$	0		1	1	1	1	1			1		1
$w_2 = 2, v_2 = 7$	0				8	8	8					8
$w_3 = 5, v_3 = 18$					8	18						26
$w_4 = 6, v_4 = 22$					8							40
$w_5 = 7, v_5 = 28$												40

$$M[i][j] = \max(\underbrace{M[i-1][j]}_{\text{rechazar } i}, \underbrace{M[i-1][j - p_i] + v_i}_{\text{seleccionar } i})$$

El 60 % de las celdas no se usan.

**40** Solución al problema  
 Contorno o perfil  
 Celdas sin usar

$$M[5][11] = \max(\underbrace{M[4][11]}_{\text{Contorno}}, \underbrace{M[4][11 - w_5] + v_5}_{\text{Contorno}}) = \max(40, 36) \quad 5 \text{ no se toma}$$

$$M[4][11] = \max(\underbrace{M[3][11]}_{\text{Contorno}}, \underbrace{M[3][11 - w_4] + v_4}_{\text{Contorno}}) = \max(26, 40) \quad 4 \text{ sí se toma}$$

$$M[3][5] = \max(\underbrace{M[2][5]}_{\text{Contorno}}, \underbrace{M[2][5 - w_3] + v_3}_{\text{Contorno}}) = \max(8, 18) \quad 3 \text{ sí se toma}$$

$$M[2][0] = M[1][0] = 0 \quad 1 \text{ y } 2 \text{ no se toman}$$

# La mochila 0/1. Versión iterativa

## Una solución iterativa

```
1 double knapsack(  
2     const vector<double> &v, // values  
3     const vector<unsigned> &w, // weights  
4     int last_n, // assessed object  
5     unsigned last_W // Knapsack limit weight  
6 ) {  
7     vector< vector< double >> M( last_n+1, vector<double>(last_W+1));  
8  
9     for( unsigned W = 0; W <= last_W; W++ ) M[0][W] = 0; // no objects  
10    for( unsigned n = 0; n <= last_n; n++ ) M[n][0] = 0; // nothing fits  
11  
12    for( unsigned n = 1; n <= last_n; n++ )  
13        for( unsigned W = 1; W <= last_W; W++ ) {  
14            double S1 = M[n-1][W];  
15            double S2 = numeric_limits<double>::lowest();  
16            if( W >= w[n-1] ) // if it fits ...  
17                S2 = v[n-1] + M[n-1][W-w[n-1]]; // try to put it  
18            M[n][W] = max( S1, S2 ); // store the best  
19        }  
20  
21    return M[last_n][last_W];  
22 }
```



# La mochila 0/1. Almacén de la versión iterativa

- Ejemplo: Sean  $n = 5$  objetos con pesos ( $w_i$ ) y valores ( $v_i$ ) indicados en la tabla. Sea  $W = 11$  el peso máximo de la mochila.

$M[6][12]^1$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 1$	0	0	1	1	1	1	1	1	1	1	1	1
$w_2 = 2, v_2 = 7$	0	0	7	7	8	8	8	8	8	8	8	8
$w_3 = 5, v_3 = 18$	0	0	7	7	8	18	18	25	25	26	26	26
$w_4 = 6, v_4 = 22$	0	0	7	7	8	18	22	25	29	29	30	40
$w_5 = 7, v_5 = 28$	0	0	7	7	8	18	22	28	29	35	35	40

$$M[i][j] = \max(\underbrace{M[i-1][j]}_{\text{rechazar } i}, \underbrace{M[i-1][j - p_i] + v_i}_{\text{seleccionar } i})$$

40

Solución al problema  
Contorno o perfil

¡ Se calculan todas las celdas !

$$\begin{aligned}
 M[5][11] &= \max \left( \underline{M[4][11]}, M[4][11 - w_5] + v_5 \right) = \max(40, 36). & 5 \text{ no se toma} \\
 M[4][11] &= \max \left( \underline{M[3][11]}, \underline{M[3][11 - w_4] + v_4} \right) = \max(26, 40). & 4 \text{ sí se toma} \\
 M[3][5] &= \max \left( \underline{M[2][5]}, \underline{M[2][5 - w_3] + v_3} \right) = \max(8, 18). & 3 \text{ sí se toma} \\
 M[2][0] &= M[1][0] = 0. & 1 \text{ y } 2 \text{ no se toman}
 \end{aligned}$$

<sup>1</sup>  $M[i][j] \equiv$  Ganancia máxima con los  $i$  primeros objetos y con capacidad máxima  $j$ . Por tanto, la solución estará en  $M[5][11]$

3#

- Complejidad temporal

$$T(n, W) = 1 + \sum_{i=1}^n 1 + \sum_{i=0}^W 1 + \sum_{i=1}^n \sum_{j=1}^W 1 = 1 + n + W + 1 + W(n+1)$$

Por tanto,

$$T(n, W) \in \Theta(nW)$$

- Complejidad espacial

$$S(n, W) \in \Theta(nW)$$

- la complejidad espacial es mejorable ...



# La mochila 0/1

- La complejidad temporal de la solución obtenida mediante programación dinámica está en  $\Theta(nW)$ 
  - Un recorrido descendente a través de la tabla permite obtener también, en tiempo  $\Theta(n)$ , la secuencia óptima de decisiones tomadas.
- Si  $W$  es muy grande entonces la solución obtenida mediante programación dinámica no es buena
- Si los pesos  $w_i$  o la capacidad  $W$  pertenecen a dominios continuos (p.e. los reales) entonces esta solución no sirve
- La complejidad espacial de la solución obtenida se puede reducir hasta  $\Theta(W)$
- En este problema, la solución PD-recursiva puede ser más eficiente que la iterativa
  - Al menos, la versión que no realiza cálculos innecesarios es más fácil de obtener en recursivo



Mejoras del algoritmo:

- ¿Se puede reducir la complejidad espacial del algoritmo iterativo propuesto?
  - ¿Cuántos vectores harían falta y de qué tamaño?
  - ¿Se perjudicaría la complejidad temporal?
- Escribe una función para obtener la secuencia de decisiones óptima a partir de la tabla completada para el algoritmo iterativo.
  - ¿Qué complejidad temporal tendría esa función?





# La mochila 0/1. Versión iterativa mejorando coste espacial

## Solución iterativa economizando memoria

```
1 double knapsack(  
2     const vector<double> &v, const vector<unsigned> &w, // data vectors  
3     int last_n, unsigned last_W      // num. objects & Knapsack limit weight  
4 ) {  
5     vector<double> v0(last_W+1);  
6     vector<double> v1(last_W+1);  
7  
8     for( unsigned W = 0; W <= last_W; W++ ) v0[W] = 0;           // no objects  
9  
10    for( int n = 1; n <= last_n; n++ ) {  
11        for( unsigned W = 1; W <= last_W; W++ ) {  
12            double S1 = v0[W];  
13            double S2 = numeric_limits<double>::lowest();  
14            if( W >= w[n-1] )           // if it fits ...  
15                S2 = v[n-1] + v0[W-w[n-1]]; // try to put it  
16            v1[W] = max( S1, S2 );      // store the best  
17        }  
18        swap(v0,v1);  
19    }  
20    return v0[last_W];  
21 }
```

# La mochila 0/1: Extracción de las decisiones /1

## Una solución iterativa con almacenamiento de todas las decisiones

```
1 double knapsack(                                // in trace we store the taken decision
2     const vector<double> &v, const vector<unsigned> &w,    // values & weights
3     int last_n, unsigned last_W,                    // assessed object & Knapsack limit
4     vector<vector<bool>> &trace                      // trace (true->store, false->don't)
5 ) {
6     vector< vector< double >> M( last_n+1, vector<double>(last_W+1));
7     trace = vector<vector<bool>>( last_n+1, vector<bool>(last_W+1));
8
9     for( unsigned W = 0; W <= last_W; W++ ) {
10         M[0][W] = 0;                                // no objects
11         trace[0][W] = false;                        // I don't take it
12     }
13
14     for( int n = 1; n <= last_n; n++ )
15         for( unsigned W = 1; W <= last_W; W++ ) {
16             double S1 = M[n-1][W];
17             double S2 = numeric_limits<double>::lowest();
18             if( W >= w[n-1] )                        // if it fits ...
19                 S2 = v[n-1] + M[n-1][W-w[n-1]];    // try to put it
20             M[n][W] = max( S1, S2 );                // store the best
21             trace[n][W] = S2 > S1;                  // if true I take it
22         }
23     return M[last_n][last_W];
24 }
```

# La mochila 0/1: Extracción de las decisiones /2

## Extracción de las decisiones que interesan

```
1 void parse(  
2     const vector<unsigned> &w,           // weights  
3     const vector<vector<bool>> &trace,    // solutions  
4     vector<bool> &sol  
5 ) {  
6     unsigned last_n = trace.size()-1;  
7     int W = trace[0].size()-1;  
8  
9     for( int n = last_n; n > 0; n-- ) {  
10         if( trace[n][W] ) {  
11             sol[n-1] = true;  
12             W -= w[n-1];  
13         } else {  
14             sol[n-1] = false;  
15         }  
16     }  
17 }
```



# La mochila 0/1: Extracción de decisiones (de otra forma)

## Extracción de la selección (directamente del almacén)

```
1 void parse(  
2     const vector<vector<double>>> &M,  
3     const vector<double> &v, const vector<unsigned> &w,    // values & weights  
4     int n, unsigned W,                                     // num. of objects & Knapsack limit  
5     vector<bool> &sol  
6 ){  
7     if( n == 0 ) return;  
8  
9     double S1 = M[n-1][W];  
10    double S2 = numeric_limits<double>::lowest();  
11    if (W >= w[n-1] )  
12        S2 = v[n-1] + M[n-1][W-w[n-1]];  
13  
14    if( S1 >= S2 ) {  
15        sol[n-1] = false;  
16        parse( M, v, w, n-1, W, sol );  
17    } else {  
18        sol[n-1] = true;  
19        parse( M, v, w, n-1, W - w[n-1], sol );  
20    }  
21 }
```

- 1 Problema 1: la mochila 0/1
- 2 Problema 2: Corte de tubos
- 3 ¿Qué hemos aprendido con los problemas 1 y 2?
- 4 Problema 3: El coeficiente binomial
- 5 Programación dinámica: Estrategia de diseño

## Problema 2: Corte de tubos

- Una empresa compra tubos de longitud  $n$  y los corta en tubos más cortos, que luego vende
  - El corte le sale gratis
  - El precio de venta de un tubo de longitud  $i$  ( $i = 1, 2, \dots, n$ ) es  $p_i$
- Por ejemplo:

longitud $i$	1	2	3	4	5	6	7	8	9	10
precio $p_i$	1	5	8	9	10	17	17	20	24	30

- ¿Cual es la forma óptima de cortar un tubo de longitud  $n$  para maximizar el precio total?
- Probar todas las formas de cortar es prohibitivo (¡hay  $2^{n-1}!$ )



## Problema 2: Corte de tubos

- Buscamos una descomposición

$$n = i_1 + i_2 + \dots + i_k$$

por la que se obtenga el precio máximo

- El precio es

$$r_n = p_{i_1} + p_{i_2} + \dots + p_{i_k}$$

- Una forma de resolver el problema recursivamente es:
  - Cortar el tubo de longitud  $n$  de las  $n$  formas posibles,
  - y buscar el corte que maximiza la suma del precio del trozo cortado  $p_i$  y del resto  $r_{n-i}$ ,
  - suponiendo que el resto del tubo se ha cortado de forma óptima:

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}); \quad r_0 = 0$$



## Problema 2: Corte de tubos

### Solución recursiva (cálculo de la ganancia máxima)

```
1 int tube_cut(  
2     const vector<int> &p,           // tube length prices  
3     const int l                   // assessed length  
4 ) {  
5  
6     if( l == 0 )  
7         return 0;  
8  
9     int q = numeric_limits<int>::lowest(); //  $q \sim -\infty$   
10    for( int i = 1; i <= l; i++ )  
11        q = max( q, p[i] + tube_cut( p, l-i ) );  
12  
13    return q;  
14 }
```

- Es ineficiente porque hay subproblemas repetidos





## Problema 2: Corte de tubos

### Solución recursiva (extracción del corte óptimo) /1

```
1 int trace_tube_cut( const vector<int> &p, const int n, vector<int> &trace ) {
2     if( n == 0 ) {      // trace stores for each length which is the optimal cut
3         trace[n] = 0;
4         return 0;
5     }
6     int q = numeric_limits<int>::lowest();
7     for( int i = 1; i <= n; i++ ) {
8         int aux = p[i] + trace_tube_cut( p, n-i, trace);
9         if( aux > q ) {    // Maximum gain
10             q = aux;
11             trace[n] = i;
12         }
13     }
14     return q;
15 }
16 //-----
17 vector<int> trace_tube_cut( const vector<int> &p, const int n ) {
18     vector<int> trace(n+1);
19     trace_tube_cut(p, n, trace);
20     return trace;
21 }
```

## Problema 2: Corte de tubos

### Solución recursiva (extracción del corte óptimo) /2

```
1 vector<int> parse(  
2     const vector<int> &trace  
3 ) {  
4     vector<int> sol(trace.size(),0);    // How many cuts for each size  
5  
6     int l = trace.size()-1;  
7     while( l != 0 ) {  
8         sol[trace[l]]++;                //where to cut  
9         l = l - trace[l];              // the rest  
10    }  
11    return sol;  
12 }  
13 //-----  
14 ...  
15     vector<int> trace = trace_tube_cut( price, n );  
16     vector<int> sol = parse(trace);  
17     for( unsigned i = 0; i < sol.size(); i++)  
18         if( sol[i] != 0 )  
19             cout << sol[i] << "└cuts└of└length└" << i << endl;  
20 ...
```

## Problema 2: Corte de tubos

- Complejidad de la solución recursiva:

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ n + \sum_{j=0}^{n-1} T(j) & \text{en otro caso} \end{cases}$$

- Observando que:

$$T(n) = 1 + 2T(n-1)$$

- Tenemos:

$$T(n) = 2^n - 1 + 2^n \in O(2^n)$$

- Hay  $2^{n-1}$  maneras de cortar el tubo (el árbol de recursión tiene  $2^{n-1}$  hojas).



# Problema 2: Corte de tubos

## Recursiva con almacén. Ganancia máxima

```
1  const int SENTINEL = -1;
2
3  int tube_cut(
4      vector<int> &M,           // Sub-problem Storage
5      const vector<int> &p, int l
6  ) {
7      if( M[l] != SENTINEL ) return M[l]; // is known?
8
9      if( l == 0 ) return M[0] = 0;
10
11     int q = numeric_limits<int>::lowest();
12     for( int i = 1; i <= l; i++ )
13         q = max( q, p[i] + tube_cut( M, p, l-i));
14
15     return M[l] = q;           // store solution & return
16 }
17
18 int tube_cut( const vector<int> &p, int l ) {
19     vector<int> M(l+1,SENTINEL); // initialization
20     return tube_cut( M, p, l );
21 }
```

- Complejidad espacial:  $O(n)$
- Complejidad temporal:  $O(n^2)$



## Problema 2: Corte de tubos

### Recursiva con almacén. Extracción del corte óptimo a partir del almacén

```
1 vector<int> memo_tube_cut(const vector<int> &p, int l){
2     vector<int> M(l+1,SENTINEL); // initialization
3     tube_cut( M, p, l );
4     return M;
5 }
6
7 vector<int> parse( const vector<int> &M, const vector<int> &p ) {
8     vector<int> sol(M.size(), 0);
9
10    int l = M.size() - 1;
11    while( l != 0 ) {
12        for( int i = 1; i <= l; i++ ) {
13            if( M[l] == p[i] + M[l-i] ) {
14                sol[i]++;
15                l -= i;
16                break;
17            }
18        }
19    }
20    return sol;
21 }
```

# Problema 2: Corte de tubos

## Solución iterativa (directa)

```
1 int tube_cut(  
2     const vector<int> &p,    // tube length prices  
3     int n                    // assessed length  
4 ) {  
5     vector<int> M(n+1);    // Sub-problem storage  
6  
7     for( int l = 0; l <= n; l++ ) {  
8  
9         if( l == 0 ) {          // base case  
10             M[0] = 0;  
11             continue;  
12         }  
13  
14         int q = numeric_limits<int>::lowest();  
15         for( int i = 1; i <= l; i++ )  
16             q = max( q, p[i] + M[l-i] );  
17         M[l] = q;              // Store solution  
18     }  
19     return M[n];  
20 }
```

- Complejidad espacial:  $O(n)$
- Complejidad temporal:  $O(n^2)$



## Problema 2: Corte de tubos

### Solución iterativa (mejor)

```
1 int tube_cut(  
2     const vector<int> &p,    // tube length prices  
3     int n                    // assessed length  
4 ) {  
5     vector<int> M(n+1);    // Sub-problem storage  
6  
7     M[0] = 0;              // Base case  
8     for( int l = 1; l <= n; l++ ) {  
9         int q = numeric_limits<int>::lowest();  
10        for( int i = 1; i <= l; i++ )  
11            q = max( q, p[i] + M[l-i] );  
12        M[l] = q;           // Store solution  
13    }  
14  
15    return M[n];  
16 }
```

- Complejidad espacial:  $O(n)$
- Complejidad temporal:  $O(n^2)$



## Problema 2: Corte de tubos

- las dos soluciones con almacén (recursiva descendente e iterativa ascendente) tienen el mismo coste temporal asintótico
- En el iterativo se observa claramente que este coste es  $\Theta(n^2)$
- El coste espacial es  $\Theta(n)$  (vector p).





- 1 Problema 1: la mochila 0/1
- 2 Problema 2: Corte de tubos
- 3 ¿Qué hemos aprendido con los problemas 1 y 2?
- 4 Problema 3: El coeficiente binomial
- 5 Programación dinámica: Estrategia de diseño

# ¿Qué hemos aprendido con los problemas 1 y 2?

Hay problemas ...

- ... con soluciones recursivas elegantes, compactas e intuitivas
- pero prohibitivamente lentas debido a que resuelven repetidamente los mismos problemas.

Hemos aprendido a:

- **Evitar repeticiones guardando resultados** (*memoización*): usar un almacén para evitar estos cálculos repetidos mejora instantáneamente el coste temporal de las soluciones descendentes (a consta de aumentar la complejidad espacial).
- Aprovechar la **subestructura óptima**: cuando la solución global a un problema incorpora soluciones a problemas parciales más pequeños que se pueden resolver de manera independiente, se puede escribir un algoritmo eficiente.

A esto se le llama **programación dinámica**



# ¿Qué hemos aprendido con los problemas 1 y 2?

## Definición:

Un problema tiene una **subestructura óptima** si una solución óptima puede construirse eficientemente a partir de las soluciones óptimas de sus subproblemas

- Esto también se conoce como **principio de optimalidad**
- Esta es una condición **necesaria** para que se puede aplicar Programación Dinámica



- 1 Problema 1: la mochila 0/1
- 2 Problema 2: Corte de tubos
- 3 ¿Qué hemos aprendido con los problemas 1 y 2?
- 4 Problema 3: El coeficiente binomial**
- 5 Programación dinámica: Estrategia de diseño

# Problema 3: El coeficiente binomial

- **Obtener el valor del coeficiente binomial**  $\binom{n}{r}$

Identidad de Pascal:  $\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$ ;  $\binom{n}{0} = \binom{n}{n} = 1$

(Solución analítica:  $\binom{n}{r} = \frac{n!}{r!(n-r)!}$ )

Coeficiente binomial

precondición:  $\{ n \geq r, n \in \mathbb{N}, r \in \mathbb{N} \}$

```
1 unsigned binomial( unsigned n, unsigned r){
2
3     if ( r == 0 || r == n )
4         return 1;
5
6     return binomial( n-1, r-1 ) + binomial( n-1, r );
7 }
```

- Complejidad temporal (relación de recurrencia múltiple)

$$T(n, r) = \begin{cases} 1 & r = 0 \vee r = n \\ 1 + T(n-1, r-1) + T(n-1, r) & \text{en otro caso} \end{cases}$$

45#

# Problema 3: El coeficiente binomial

La solución recursiva es ineficiente.

- Aproximando a una relación de recurrencia lineal:
- Si suponemos:

$$T(n-1, r) \geq T(n-1, r-1)$$

$$T(n, r) \leq g(n, r) = \begin{cases} 1 & n = r \\ 1 + 2g(n-1, r) & \text{en otro caso} \end{cases}$$

$$g(n, r) = 2^k - 1 + 2^k g(n-k, r) \quad \forall k = 1 \dots (n-r)$$

- Por tanto:

$$T(n, r) \sim g(n, r) \in O(2^{n-r})$$



## Problema 3: El coeficiente binomial

- Si, en cambio, suponemos:

$$T(n-1, r) \leq T(n-1, r-1)$$

$$T(n, r) \sim g(n, r) = \begin{cases} 1 & r = 0 \\ 1 + 2g(n-1, r-1) & \text{en otro caso} \end{cases}$$

$$g(n, r) = 2^k - 1 + 2^k g(n-k, r-k) \quad \forall k = 1 \dots r$$

- Por tanto:

$$T(n, r) \sim g(n, r) \in O(2^r)$$

Combinando ambos resultados:

$$T(n, r) \sim g(n, r) \in O(2^{\min(r, n-r)})$$

¡Esta solución recursiva no es aceptable!



# Problema 3: El coeficiente binomial

Estudio empírico de la complejidad:

$(n, r) = \binom{n}{r}$	Pasos
(40, 0)	1
(40, 1)	79
(40, 2)	1559
(40, 3)	19759
(40, 4)	182779
(40, 5)	$1.3 \times 10^6$
(40, 7)	$3.7 \times 10^7$
(40, 9)	$5.4 \times 10^8$
(40, 11)	$4.6 \times 10^9$
(40, 15)	$8.0 \times 10^{10}$
(40, 17)	$1.8 \times 10^{11}$
(40, 20)	$2.8 \times 10^{11}$

$(n, r) = \binom{n}{r}$	Pasos
(2, 1)	3
(4, 2)	11
(6, 3)	39
(8, 4)	139
(10, 5)	503
(12, 6)	1847
(14, 7)	6863
(16, 8)	25739
(18, 9)	97239
(20, 10)	369511
(22, 11)	1410863
(24, 12)	5408311

- Caso más costoso:  $n = 2r$ ; crecimiento aprox.  $2^n$
- los resultados son claramente **prohibitivos**
- Recurrencia aprox.:  $f(n) = 1 + 2f(n-1)$ ;  $f(1)=1$





# Problema 3: El coeficiente binomial

- ¿Por qué es ineficiente esta solución descendente (*top-down*)?
  - Los problemas se reducen en subproblemas de tamaño similar ( $n - 1$ ).
  - Un problema se divide en dos subproblemas y cada uno de estos en otros dos, y así sucesivamente.

⇒ Esto lleva a complejidades prohibitivas (p.e. exponenciales)
- Pero, ¡el total de subproblemas diferentes no es tan grande!
  - sólo hay  $nr$  posibilidades distintas

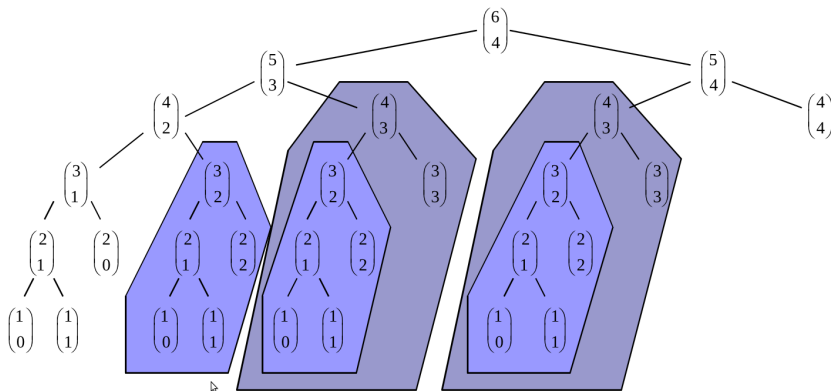
¡La solución recursiva está generando y resolviendo el mismo problema muchas veces!

- ¡Cuidado! la ineficiencia no es debida a la recursividad



### Problema 3: El coeficiente binomial

- Solución recursiva: ejemplo para  $n = 6$  y  $r = 4$



- INCONVENIENTE: subproblemas repetidos.
  - Pero sólo hay  $nr$  subproblemas diferentes: El problema se puede resolver utilizando almacenes intermedios.

# Problema 3: El coeficiente binomial

- Memoización: Almacenamiento de valores ya calculados para no volver a calcularlos.

Una solución recursiva mejorada

$\{n \geq r, n \in \mathbb{N}, r \in \mathbb{N}\}$

```
1 const unsigned SENTINEL = 0;
2
3 unsigned binomial( vector<vector<unsigned>> &M, unsigned n, unsigned r) {
4
5     if( M[n][r] != SENTINEL )
6         return M[n][r];
7     if( r == 0 || r == n )
8         return 1;
9
10    M[n][r] = binomial(M, n-1, r-1) + binomial(M, n-1, r);
11
12    return M[n][r];
13 }
14
15 unsigned binomial( unsigned n, unsigned r) {
16     vector<vector<unsigned>> M( n+1, vector<unsigned>(r+1, SENTINEL));
17     return binomial( M, n, r);
18 }
```

# Problema 3: El coeficiente binomial

Memoización (para varios problemas)

$\{n \geq r, n \in \mathbb{N}, r \in \mathbb{N}\}$

```
1 unsigned binomial( vector<vector<unsigned>> &M, unsigned n, unsigned r) {
2     if( M[n][r] != 0 ) return M[n][r];
3     if( r == 0 || r == n ) return 1;
4     M[n][r] = binomial(M, n-1, r-1) + binomial(M, n-1, r);
5     return M[n][r];
6 }
7
8 const unsigned MAX_N = 100;
9
10 unsigned binomial( unsigned n, unsigned r) {
11     static vector<vector<unsigned>> M;
12     static bool initialized = false;
13
14     if( !initialized ) {
15         M = vector<vector<unsigned>>(MAX_N, vector<unsigned>(MAX_N, SENTINEL));
16         initialized = true;
17     }
18
19     return binomial( M, n, r);
20 }
```

# Problema 3: El coeficiente binomial

## Memoización (functors)

$$\{n \geq r, n \in \mathbb{N}, r \in \mathbb{N}\}$$

```
1 const unsigned SENTINEL = 0;
2 const unsigned MAX_N = 100;
3
4 class Binomial {
5 public:
6     Binomial( unsigned max_n = MAX_N ) : M(
7         vector<vector<unsigned>>(max_n+1, vector<unsigned>(max_n+1, SENTINEL))
8     ){};
9
10    unsigned operator()( unsigned n, unsigned r ) {
11        if( M[n][r] != SENTINEL ) return M[n][r];
12        if( r == 0 || r == n ) return 1;
13        M[n][r] = operator()(n-1, r-1) + operator()(n-1, r);
14        return M[n][r];
15    }
16
17 private:
18     vector<vector<unsigned>> M;
19 };
20
21 Binomial binomial(40); // use: a = binomial(30,20);
```

# Problema 3: El coeficiente binomial

- Los resultados mejoran muchísimo cuando se añade un almacén:

$(n, r) = \binom{n}{r}$	Ingenuo	Mem.	$(n, r) = \binom{n}{r}$	Ingenuo	Mem.
(40, 0)	1	1	(2, 1)	3	3
(40, 1)	79	79	(4, 2)	11	8
(40, 2)	1559	116	(6, 3)	20	15
(40, 3)	19759	151	(8, 4)	139	24
(40, 4)	182779	184	(10, 5)	503	35
(40, 5)	$1.3 \times 10^{06}$	215	(12, 6)	1847	48
(40, 7)	$3.7 \times 10^{07}$	271	(14, 7)	6863	64
(40, 9)	$5.4 \times 10^{08}$	319	(16, 8)	25739	80
(40, 11)	$4.6 \times 10^{09}$	359	(18, 9)	97239	99
(40, 15)	$8.0 \times 10^{10}$	415	(20, 10)	369511	120
(40, 17)	$1.8 \times 10^{11}$	432	(22, 11)	1410863	143
(40, 20)	$2.8 \times 10^{11}$	440	(24, 12)	5408311	168

- En el caso  $n = 2r$ , el crecimiento es del tipo  $(n/2)^2 + n \in \Theta(n^2)$ .



# Problema 3: El coeficiente binomial

- ¿Se puede evitar la recursividad? En este caso sí
  - Resolver los subproblemas de menor a mayor (recorrido ascendente o *bottom-up*)
  - **Almacenar** sus soluciones en una tabla  $M[n][r]$  donde

$$M[i][j] = \binom{i}{j}$$

- El almacén de resultados parciales permite evitar repeticiones.
- La tabla se inicializa con la solución a los subproblemas triviales:

$$\begin{aligned} M[i][0] &= 1 & \forall i = 1 \dots (n-r) \\ M[i][i] &= 1 & \forall i = 1 \dots r \end{aligned}$$

Puesto que

$$\binom{m}{0} = \binom{m}{m} = 1, \quad \forall m \in \mathbb{N}$$



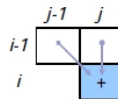
# Problema 3: El coeficiente binomial

Recorrido de los subproblemas:

- Los subproblemas se resuelven en sentido ascendente.
- Se almacenan sus soluciones pues harán falta para los siguientes subproblemas.

$$M[i][j] = M[i-1][j-1] + M[i-1][j]$$

$$\forall (i, j) : (1 \leq j \leq r, j+1 \leq i \leq n-r+j)$$



	0	1	2	3	4	$j(r)$
0	1					
1	1	1				
2	1		1			
3				1		
4					1	
5						
6						

$i(n)$









# Problema 3: El coeficiente binomial

Una solución iterativa y polinómica (mejorable)

- Ejemplo: Sea  $n = 6$  y  $r = 4$

	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3		3	3	1	
4			6	4	1
5				10	5
6					15

-  Celdas sin utilizar ¡desperdicio de memoria!
-  Instancias del caso base: perfil o contorno de la matriz
-  Soluciones de los subproblemas. Obtenidos, en este caso, de arriba hacia abajo y de izquierda a derecha
-  Solución del problema inicial.  $M[6][4] = \binom{6}{4}$

## Solución trivial de programación dinámica

$\{n \geq r, n \in \mathbb{N}, r \in \mathbb{N}\}$

```
1 unsigned binomial(unsigned n, unsigned r){
2     unsigned M[n+1][r+1];
3
4     for (unsigned i=0; i <= n-r; i++) M[i][0] = 1;
5     for (unsigned i=1; i <= r; i++) M[i][i] = 1;
6
7     for (unsigned j=1; j<=r; j++)
8         for (unsigned i=j+1; i<=n-r+j; i++)
9             M[i][j] = M[i-1][j-1] + M[i-1][j];
10    return M[n][r];
11 }
```

# Problema 3: El coeficiente binomial

## Solución trivial de programación dinámica

$\{n \geq r, n \in \mathbb{N}, r \in \mathbb{N}\}$

```
1 unsigned binomial(unsigned n, unsigned r){
2     unsigned M[n+1][r+1];
3
4     for (unsigned i=0; i <= n-r; i++) M[i][0] = 1;
5     for (unsigned i=1; i <= r; i++) M[i][i] = 1;
6
7     for (unsigned j=1; j<=r; j++)
8         for (unsigned i=j+1; i<=n-r+j; i++)
9             M[i][j] = M[i-1][j-1] + M[i-1][j];
10    return M[n][r];
11 }
```

- Coste temporal exacto:

$$T(n, r) = 1 + \sum_{i=0}^{n-r} 1 + \sum_{i=1}^r 1 + \sum_{j=1}^r \sum_{i=j+1}^{n-r+j} 1 = rn + n - r^2 + 1 \in \Theta(rn)$$

- Idéntico al descendente con memoización (almacén)



# Problema 3: El coeficiente binomial

## Solución trivial de programación dinámica

$\{n \geq r, n \in \mathbb{N}, r \in \mathbb{N}\}$

```
1 unsigned binomial(unsigned n, unsigned r){
2     unsigned M[n+1][r+1];
3
4     for (unsigned i=0; i <= n-r; i++) M[i][0]= 1;
5     for (unsigned i=1; i <= r; i++) M[i][i]= 1;
6
7     for (unsigned j=1; j<=r; j++)
8         for (unsigned i=j+1; i<=n-r+j; i++)
9             M[i][j]= M[i-1][j-1] + M[i-1][j];
10    return M[n][r];
11 }
```

- **Complejidad espacial:**  $\Theta(rn)$  ¿Se puede mejorar?
- Podéis verlo en <http://v.gd/binCoeff>.



# Problema 3: El coeficiente binomial

- Ejercicios propuestos: Reducción de la complejidad espacial:
  - Modificar la función anterior de manera que el almacén no sea más que dos vectores de tamaño  $1 + \min(r, n - r)$
  - Modificar la función anterior de manera que el almacén sea un único vector de tamaño  $1 + \min(r, n - r)$
  - Con estas modificaciones, ¿queda afectada de alguna manera la complejidad temporal?



- 1 Problema 1: la mochila 0/1
- 2 Problema 2: Corte de tubos
- 3 ¿Qué hemos aprendido con los problemas 1 y 2?
- 4 Problema 3: El coeficiente binomial
- 5 Programación dinámica: Estrategia de diseño

## Identificación:

- Diseñar una solución recursiva al problema (top-down)
- Análisis: la complejidad temporal es prohibitiva (p.ex., exponencial)
  - Subproblemas superpuestos
  - Reparto no equitativo de las tallas de los subproblemas
- Si el problema es un problema de optimización, verificar que se puede establecer una subestructura óptima.



## Transformación de recursivo a iterativo:

- El siguiente paso es la construcción de la función iterativa (*bottom-up*) a partir de la recursiva (*top-down*)
  - Las llamadas recursivas se sustituyen por accesos a la tabla-almacén
    - Se podría decir, en términos coloquiales, que en el lenguaje de programación se sustituyen paréntesis por corchetes.
  - Sustituir la orden que devuelve el valor en la función recursiva por un almacenamiento en la tabla
  - Utilizar los casos base de la solución recursiva para empezar a rellenar el contorno de la tabla
  - A partir del caso general en la función recursiva, diseñar la estrategia que permita crear los bucles que completen la tabla a partir de los subproblemas resueltos (recorrido ascendente o *bottom-up*)
    - **Es habitual que complejidades exponenciales se transformen en polinómicas**



## Programación dinámica recursiva:

- La programación dinámica recursiva consiste en hacer uso del almacén en la versión recursiva
  - La versión recursiva puede ser más eficiente que la iterativa:
    - Evitar los cálculos innecesarios puede ser más fácil en la versión recursiva que en la iterativa
- Por lo tanto, la programación dinámica no implica necesariamente una transformación a iterativo
  - En sus orígenes la transformación a iterativo era un valor añadido pero se debía a que los compiladores no admitían recursividad





# Programación dinámica: Estrategia de diseño

## Paso de divide y vencerás a programación dinámica

### Esquema divide y vencerás

```
1 Solution DC( Problem p ) {  
2     if( is_simple(p) ) return trivial(p);  
3  
4     list<Solution> s;  
5     for( Problem q : divide(p) ) s.push_back( DC(q) );  
6     return combine(s);  
7 }
```

### Esquema programación dinámica (recursiva)

```
1 Solution DP( Problem p ) {  
2     if( is_solved(p) ) return M[p];  
3     if( is_simple(p) ) return M[p] = trivial(p); // or simply: return trivial(p)  
4  
5     list<Solution> s;  
6     for( Problem q : divide(p) ) s.push_back( DP(q) );  
7     M[p] = combine(s);  
8     return M[p];  
9 }
```

# Programación dinámica: Estrategia de diseño

## Paso a programación dinámica iterativa:

### Esquema programación dinámica (iterativa)

```
1 Solution DP( Problem P) {  
2     vector<Solution> M;  
3     list<Problem> e = enumeration(P);  
4  
5     while( !e.empty() ) {  
6         Problem p = e.pop_front();  
7         if( is_simple(p) )  
8             M[p] = trivial(p);  
9         else {  
10            list<Solution> s;  
11            for( Problem q : divide(p) ) s.push_back( M[q] );  
12            M[p] = combine(s);  
13        }  
14    }  
15    return M[P];  
16 }
```

Le enumeración ha de cumplir:

- todo problema en `divide(p)` aparece antes que `p`
- el problema `P` es el último de la enumeración.



# Programación dinámica: casos de aplicación

- Problemas clásicos para los que resulta eficaz la programación dinámica
  - El problema de la mochilla 0-1
  - Cálculo de los números de Fibonacci
  - Problemas con cadenas:
    - La subsecuencia común máxima (*longest common subsequence*) de dos cadenas.
    - La distancia de edición (*edit distance*) entre dos cadenas.
  - Problemas sobre grafos:
    - El viajante de comercio (*travelling salesman problem*)
    - Caminos más cortos en un grafo entre un vértice y todos los restantes (alg. de Dijkstra)
    - Existencia de camino entre cualquier par de vértices (alg. de Warshall)
    - Caminos más cortos en un grafo entre cualquier par de vértices (alg. de Floyd)

