

Análisis y diseño de algoritmos

4. Programación Dinámica

Juan Morales García,
Victor M. Sánchez Cartagena,
Jose Luis Verdú Mas

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

12 de enero de 2025



- 1 Ejemplo introductorio: El problema de la mochila
- 2 Otro ejemplo Introductorio: Corte de tubos
- 3 La programación dinámica
- 4 Cálculo del coeficiente binomial



- 1 Ejemplo introductorio: El problema de la mochila
- 2 Otro ejemplo Introductorio: Corte de tubos
- 3 La programación dinámica
- 4 Cálculo del coeficiente binomial



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
 - No se conoce ningún algoritmo que lo resuelva en un tiempo razonable (polinómico)
 - Tampoco está demostrado que tal algoritmo no existe
- **Simplificación que permite abordarlo mediante PD:** Los pesos son cantidades discretas o discretizables
 - Se utilizarán para indexar un almacén de resultados parciales
 - Una versión menos general que suaviza su dificultad



- Es un problema de optimización:
 - hay numerosas alternativas sujetas a restricciones
 - se busca la óptima tomando decisiones
- En este caso:
 - 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:
 - $\sum_{i=1}^n x_i w_i \leq W$
 - $\forall i : 1 \leq i \leq n, x_i \in \{0, 1\}$

Solución mediante Divide y vencerás (DV)

- Representamos mediante $\text{knapsack}(k, C)$ al problema de la mochila con los objetos 1 hasta k y capacidad C
 - El problema inicial es, por tanto, $\text{knapsack}(n, W)$
- Según hemos visto en la técnica DV, hay que decidir:
 - Cómo reducir el problema y cuántos subproblemas habrá,
 - Cómo combinar las soluciones de esos subproblemas,
 - Cuándo un problema es lo suficientemente pequeño para que tenga solución fácil
 - Cuál es la solución de un problema “pequeño”



Solución

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

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

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

Es decir:

- 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 hasta que no se resuelvan todos los subproblemas, de menor a mayor (implícitamente quedan enumerados).



Subestructura óptima

Definición:

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

- Es decir, la subestructura de los subproblemas puede ser usada para encontrar la solución óptima de el problema completo.
 - Depende de los subproblemas escogidos, pero si existe una descomposición en subproblemas que lo cumple entonces se dice que el problema presenta una subestructura óptima.
- Esto también se conoce como **Principio de optimalidad**
- Es **condición necesaria** para que se puede aplicar Programación dinámica (PD) (también lo es para DV)
 - Para demostrar que se cumple, basta con probar que las soluciones de los subproblemas han de ser necesariamente óptimas si también lo es la solución del problema en su conjunto.



Subestructura óptima o Principio de optimalidad

- 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.^a

^aEste tipo de demostraciones se denominan “cut and paste”

- **El problema de la mochila discreta presenta una subestructura óptima**



Para pensar...

- 1 En cuanto al problema de la mochila discreta con pesos discretos, ¿Podrías decir otra descomposición en subproblemas que no cumpla la propiedad de subestructura óptima?
- 2 La propiedad de subestructura óptima es condición necesaria para aplicar PD, ¿es también condición suficiente? ¿Qué ocurre en el caso DV?
- 3 El problema de la mochila discreta con pesos continuos (versión general), ¿cumple la propiedad de subestructura óptima?
- 4 En el caso de que la respuesta anterior fuera afirmativa, ¿porqué no se conoce una solución polinómica para este problema?



Punto de partida para resolver mediante PD

Solución matemática (DV)

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

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

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

- La solución matemática basada en el esquema DV, es el punto de partida de una solución de PD. A través de ella se conoce información relevante para aplicar la estrategia:
 - Cuáles son los subproblemas
 - Parámetros que caracterizan e identifican de forma única cada subproblema
 - Dominio y recorrido de la función (tipos de datos en los valores de entrada y de salida, y sus rangos)
 - Subproblemas que corresponden al caso base, y sus soluciones
 - Orden en el que se tienen que resolver los subproblemas en un algoritmo iterativo



Una solución recursiva (sigue siendo DV)

Es la transcripción literal de la recurrencia matemática

Solución recursiva (o solución *naive* -ingenua-)(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 }
```

Versión recursiva: Complejidad temporal

- En el mejor de los casos:
ningún objeto cabe en la mochila, se tiene $T(n) \in \Omega(n)$
- En el peor de los casos:

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

El término general queda como:

$$T(n) \stackrel{k}{=} 2^k - 1 + 2^k T(n-k)$$

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

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



La innecesaria repetición de cálculos

- ¿Por qué es ineficiente?
 - Los problemas se reducen en subproblemas de tamaño similar ($n - 1$).
 - Un problema se divide en dos subproblemas, y así sucesivamente.
⇒ Esto lleva a complejidades prohibitivas (p.e. exponenciales)
- Pero, ¡el total de subproblemas diferentes no es tan grande!
 - ¡solo hay nW problemas distintos!

Por lo tanto:

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

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



Versión recursiva: Subproblemas repetidos

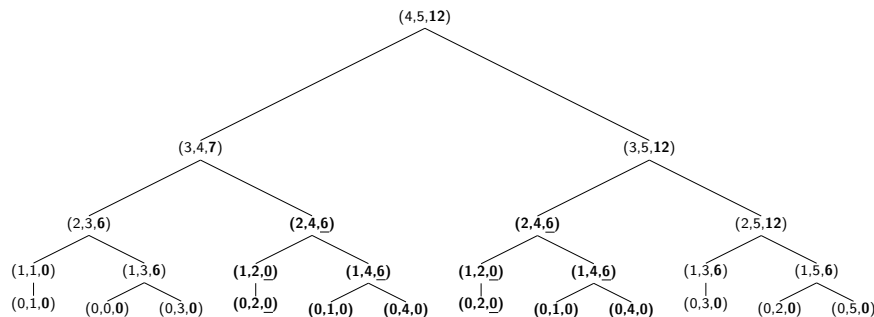
En efecto, se resuelven muchos subproblemas repetidos:

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

• Ejemplo: $w = (3, 2, 1, 1)$

$$v = (6, 6, 2, 1)$$

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



¿Cómo evitar repeticiones?

- Por cada (sub)problema que se presenta:
 - Si no ha sido resuelto previamente:
 - Se resuelve y se guarda la solución en un almacén
 - Si ya ha sido resuelto:
 - Devolver la solución guardada en el almacén
- Esta técnica se denomina **memoización** (*memoization*¹)
- **Programación dinámica se caracteriza por el uso de un almacén para evitar la repetición de cálculos innecesarios.**
 - Es aplicable cuando se puede resolver un problema basándose en subproblemas que ya han sido resueltos y sus soluciones guardadas en el almacén.

¹memoization: “recordar el resultado de una función”

Programación dinámica recursiva (memoización)

Necesitamos:

- Encontrar una forma de identificar de forma única cada subproblema, para lo localizarlo en el almacén $[[n][W]]$
- Identificar un tipo de dato aceptable para la solución `[double]`
- Definir una estructura donde almacenar los resultados
 - `vector<vector<double>> M`
- Decidir un valor centinela que permita saber si el subproblema ha sido ya resuelto
 - Debe ser un valor que no corresponda a una posible solución `[-1.0]`
- Modificar el programa para:
 - inicializar la estructura al centinela
 - `vector<vector<double>> M(n+1, vector<double>(W+1,-1.0))`
 - se declara una fila y una columna más para albergar el caso base
 - al comenzar cada llamada recursiva, comprobar si el problema ya está resuelto
 - si ya se ha resultado previamente devolver la solución almacenada (sin más).
 - al terminar, almacenar la solución



Programación dinámica recursiva (memoización)

Una solución recursiva con almacén (Memoización)

```
1 const double SENTINEL = -1.0;
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];                  // solution 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     vector< vector< double >> M( n+1, vector<double>( W+1, SENTINEL)); // init.
19     return knapsack( M, v, w, n, W );
20 }
```

Programación dinámica recursiva. Ejemplo de almacén

- 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

- El 60 % de las celdas no se han utilizado por lo tanto:

**El subproblema asociado no ha sido resuelto
¡Ahorro computacional!**



40 Solución al problema
Contorno o perfil
Celdas sin uso

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

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

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

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

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 los paréntesis por corchetes.
 - Los retornos de valores se sustituyen almacenamientos
- Construcción del algoritmo:
 - 1 Se declara el almacén incorporando espacio para albergar los casos base
 - 2 Utilizar los casos base de la solución recursiva para rellenar el contorno del almacén
 - 3 A partir del caso general en la función recursiva, diseñar la estrategia que permita crear los bucles que completen el almacén a partir de los subproblemas ya resueltos (recorrido ascendente o *bottom-up*)
 - resolvemos los subproblemas de menor a mayor
 - la solución del problema en su conjunto será el último almacenamiento realizado.



Solución iterativa

Solución iterativa con complejidad espacial mejorable

```
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  
11     for( int n = 1; n <= last_n; n++ )  
12         for( unsigned W = 1; W <= last_W; W++ ) {  
13             double S1 = M[n-1][W];  
14             double S2 = numeric_limits<double>::lowest();  
15             if( W >= w[n-1] ) // if it fits ...  
16                 S2 = v[n-1] + M[n-1][W-w[n-1]]; // try to put it  
17             M[n][W] = max( S1, S2 ); // store the best  
18         }  
19  
20     return M[last_n][last_W];  
21 }
```

Almacén de resultados parciales

- 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	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

$M[i][j] \equiv$ Ganancia máxima con los i primeros objetos y con una carga máxima j . Por tanto, solución en $M[5][11]$

40 Solución al problema
Contorno o perfil

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

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

Iterativo: Complejidad temporal y espacial

- Complejidad temporal

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

Por tanto,

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

- Complejidad espacial

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

- la complejidad espacial es mejorable ...



Solución iterativa con complejidad espacial mejorada

- En algunos casos (como este) la complejidad espacial asintótica se puede reducir.
 - puede no ser necesario almacenar las soluciones de todos los subproblemas, sino un subconjunto de estos.

- En este caso, tenemos:

$$M[n][W] = \text{máx}(M[n-1][W], M[n-1][W - w_n] + v_n)$$

- es decir, para calcular los elementos de la fila n , solo se necesitan los de la fila $n-1$; las filas anteriores ya no se necesitarán.
- por lo tanto, la matriz M puede ser reemplazada por 2 vectores ($V1$ y $V2$) que corresponderían a 2 filas consecutivas ($M[n-1]$ y $M[n]$)

$$V2[W] = \text{máx}(V1[W], V1[W - w_n] + v_n)$$

- cada vez que se completa $V2$, se vuelca su contenido en $V1$ y se repite el proceso con el siguiente valor de n .
- **La complejidad temporal no cambia.**



Solución iterativa

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 }
```

Extracción de las decisiones óptimas

- Las decisiones que corresponden al valor óptimo se pueden extraer del almacén, ya sea el de la versión recursiva o la iterativa
 - En el caso de la iterativa hay que conservar todas las soluciones de los subproblemas (sin reducir la complejidad espacial)
- Procedimiento:
 - 1 Comenzamos en la casilla del almacén que corresponde a la solución global ($M[n][W]$)
 - 2 Localizamos la casilla que le dio valor. En este problema solo pueden ser dos:
 - $M[n-1][W]$, que corresponde al caso en el que el objeto n se rechaza.
 - $M[n-1][W - w[n]] + v[n]$, que corresponde a seleccionar el objeto n .
 - 3 El objeto n se acepta o se rechaza según qué casilla sea
 - Si las dos alternativas coinciden, entonces tomamos una cualquiera (secuencia de decisiones distintas que producen el mismo resultado óptimo)
 - 4 Repetimos el proceso desde esa casilla, hasta llegar al primer objeto



Extracción de la selección

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     unsigned W,                                           // num. of objects & Knapsack limit  
5     vector<bool> &sol                                     // true -> take it, false -> don't  
6 ){  
7     int n = v.size();  
8  
9     for( int o = n-1; o >= 0; o-- ) {  
10  
11         double S1 = M[o][W];  
12  
13         double S2 = numeric_limits<double>::lowest();  
14         if (W >= w[o] )  
15             S2 = v[o] + M[o][W-w[o]];  
16  
17         sol[o] = S2 > S1;  
18         if( sol[o] )  
19             W -= w[o];  
20     }  
21 }
```

- 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 (puede que inviable)
- 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



- 1 Ejemplo introductorio: El problema de la mochila
- 2 Otro ejemplo Introductorio: Corte de tubos
- 3 La programación dinámica
- 4 Cálculo del coeficiente binomial

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	27

- ¿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 $2^{n-1}!$!)



- 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:

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



Solución recursiva (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 ~ -∞>  
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



Complejidad de la solución recursiva

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

Como:

$$T(n-1) = n-1 + \sum_{j=0}^{n-2} T(j) \Rightarrow 2T(n-1) = n-1 + \sum_{j=0}^{n-1} T(j)$$

y teniendo en cuenta que $T(n) = n + \sum_{j=0}^{n-1} T(j)$ llegamos a

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

Por tanto:

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



Solución recursiva (corte óptimo)

```
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 }
```

Extrayendo los cortes óptimos

```
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 -= 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 ...
```

Solución recursiva con almacén (Memoización)

Corte de tubos. 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: $O(n^2)$



Solución Recursiva con almacén (Memoización)

Corte de tubos. Corte óptimo

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



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	27	
Ganancia M	0	1	5	8	10	13	17	18	22	25	27

- Si $l=10$ entonces tenemos las siguientes secuencias óptimas de decisiones:
 - 2 cortes a 2 metros; 1 corte a 6 metros.
 - No cortar el tubo
- Si $l=5$:
 - 1 cortes a 2 metros; 1 corte a 3 metros.



Corte de tubos

```
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: $O(n^2)$

- La complejidad espacial no se puede reducir.



- 1 Ejemplo introductorio: El problema de la mochila
- 2 Otro ejemplo Introductorio: Corte de tubos
- 3 La programación dinámica
- 4 Cálculo del coeficiente binomial



¿Qué hemos aprendido con estos ejemplos introductorios?

Hay problemas

- con soluciones recursivas formalmente elegantes, compactas e intuitivas,
 - pero prohibitivamente lentas porque resuelven repetidamente los mismos problemas.

Hemos aprendido a:

- **Evitar repeticiones guardando resultados** (*memoización*): usar un almacén (sacrificando complejidad espacial) para evitar estos cálculos repetidos mejora instantáneamente el coste de las soluciones descendentes.
- Aprovechar la **subestructura óptima**: cuando la solución global a un problema incorpora soluciones a problemas parciales más pequeños que podemos resolver independientemente, podemos escribir una solución muy eficiente.



¿Qué hemos aprendido con estos ejemplos introductorios?

Si la solución de un problema, especialmente si es de optimización,

- es decir, cuando hay numerosas alternativas
- y se busca la óptima tomando decisiones

se puede obtener

- evitando cálculos repetidos, y
- aprovechando la subestructura óptima

puede convenir aplicar los métodos vistos.

A esto se le llama **programación dinámica**.



- Identificación:

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



Paso de Divide y Vencerás a Programación Dinámica

Esquema Divide y Vencerás (DC)

```
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 (DP, 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 }
```

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.



PD recursiva vs PD iterativa

- En ambas, la complejidad temporal asintótica suele ser la misma
 - pero la versión recursiva puede ser más eficiente (depende del problema)
 - pues solo resuelve los subproblemas que necesita
 - la versión iterativa debe resolver todos los subproblemas pues no sabe los que necesitará más adelante (inconveniente del recorrido *bottom-up*).
- Ambas hacen uso del almacén
 - pero la versión iterativa puede requerir menor complejidad espacial (depende del problema)
 - en la versión recursiva no es posible reducir el tamaño del almacén.
- Por tanto, la programación dinámica no implica necesariamente una transformación a iterativo
 - En sus orígenes, la transformación a iterativo era un valor añadido puesto que los lenguajes de programación no admitían recursividad.
- Aún así, programación dinámica suele referirse a la versión iterativa



Ejemplos de aplicación

- Problemas clásicos para los que resulta eficaz la programación dinámica
 - El problema de la mochila 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)



- 1 Ejemplo introductorio: El problema de la mochila
- 2 Otro ejemplo Introductorio: Corte de tubos
- 3 La programación dinámica
- 4 Cálculo del coeficiente binomial



Obtener el valor del coeficiente binomial

- Identidad de Pascal:

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r} \text{ con } n \geq r; \quad \binom{n}{0} = \binom{n}{n} = 1$$

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)

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



La solución recursiva es ineficiente

- Aproximando a una relación de recurrencia lineal:
- si suponemos que:

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

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

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

- Por tanto:

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



La solución recursiva es ineficiente

- Si 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:

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

O combinando los dos:

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

¡Esta solución recursiva no es aceptable!



Algunos números

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

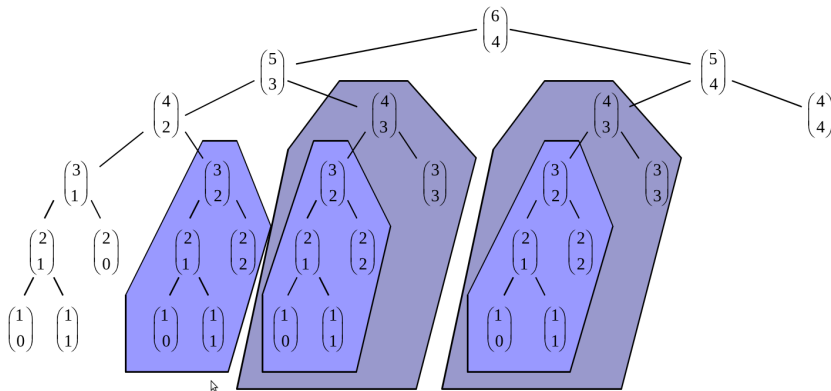
$(n, r) = \binom{n}{r}$	Pasos
(2, 1)	3
(4, 2)	11
(6, 3)	39
(8, 4)	139
(10, 5)	503
(12, 6)	1847
(14, 7)	6863
(16, 8)	25739
(18, 9)	97239
(20, 10)	369511
(22, 11)	1410863
(24, 12)	5408311

- Caso más costoso: $n = 2r$; crecimiento aprox. 2^n .
- los resultados son claramente **prohibitivos**



La innecesaria repetición de cálculos

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



- **INCONVENIENTE:** subproblemas repetidos.
 - Pero sólo hay nr subproblemas diferentes: \Rightarrow uso de almacenes



¿Cómo evitar la repetición de cálculos?

⇒ almacenar los valores ya calculados para no recalcularlos:

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
8     if( r == 0 || r == n )
9         return M[n][r] = 1;
10
11     M[n][r] = binomial(M, n-1, r-1) + binomial(M, n-1, r);
12
13     return M[n][r];
14 }
15
16 unsigned binomial( unsigned n, unsigned r ) {
17     vector<vector<unsigned>> M( n+1, vector<unsigned>(r+1, SENTINEL));
18     return binomial( M, n, r);
19 }
```

Memoización (para varios problemas)

Memoización

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

```
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 M[n][r] = 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 }
```

Memoización (para varios problemas)

Memoización (functores)

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

```
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 M[n][r] = 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);
```


Algunos números

$(n, r) \equiv \binom{n}{r}$	Ingenuo	Mem.
(40, 0)	1	1
(40, 1)	79	79
(40, 2)	1559	116
(40, 3)	19759	151
(40, 4)	182779	184
(40, 5)	1.3×10^{06}	215
(40, 7)	3.7×10^{07}	271
(40, 9)	5.4×10^{08}	319
(40, 11)	4.6×10^{09}	359
(40, 15)	8.0×10^{10}	415
(40, 17)	1.8×10^{11}	432
(40, 20)	2.8×10^{11}	440

$(n, r) \equiv \binom{n}{r}$	Ingenuo	Mem.
(2, 1)	3	3
(4, 2)	11	8
(6, 3)	20	15
(8, 4)	139	24
(10, 5)	503	35
(12, 6)	1847	48
(14, 7)	6863	64
(16, 8)	25739	80
(18, 9)	97239	99
(20, 10)	369511	120
(22, 11)	1410863	143
(24, 12)	5408311	168

- En el caso $n = 2r$, el crecimiento es del tipo $(n/2)^2 + n \in \Theta(n^2)$.
- Los resultados mejoran muchísimo cuando se añade un almacén



¿Cómo evitar la recursividad?

- ¿Se puede evitar la recursividad? En este caso sí
 - Resolver los subproblemas de menor a mayor
 - **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

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

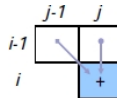


Recorrido de los subproblemas

- Resolviendo los subproblemas en sentido ascendente y almacenando sus soluciones:

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

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







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



Una solución 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 }
```

Una solución polinómica (mejorable)

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)



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 espacial:** $\Theta(rn)$ ¿Se puede mejorar?



- 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?

