Análisis y diseño de algoritmos

4. Programación Dinámica

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

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

26 de febrero de 2019

Índice

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

- Es un problema de optimización:
  - Secuencia de decisiones: $(x_1, x_2 \ldots x_n): \ x_i \in \{0,1\},\ 1 \leq i \leq n$
    - 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 $\displaystyle\sum_{i=1}^{n} x_i v_i$

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

La mochila 0/1. Subestructura óptima

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

Demostración:

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

[a] Este tipo de demostraciones se denominan "cut and paste"

$\Rightarrow$ 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: $x_n, x_{n-1}, \ldots x_1$
- 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 $\{\ W \geq 0,\ n > 0\ \}$

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

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

Solución recursiva (ineficiente)

```
#include <limits>

double knapsack(
    const vector<double> &v,    // values
    const vector<unsigned> &w,  // weights
    int n,                       // number of objects
    unsigned W                  // knapsack weight limit
) {
    if( n == 0 )                                            // base case
        return 0;

    double S1 = knapsack( v, w, n-1, W );        // try not to put it on

    double S2 = numeric_limits<double>::lowest();
    if( w[n-1] <= W )                        // does it fits in the knapsack?
        S2 = v[n-1] + knapsack( v, w, n-1, W-w[n-1] ); // try to put it on

    return max( S1, S2 );                                // choose the best
}
```

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 témino 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!

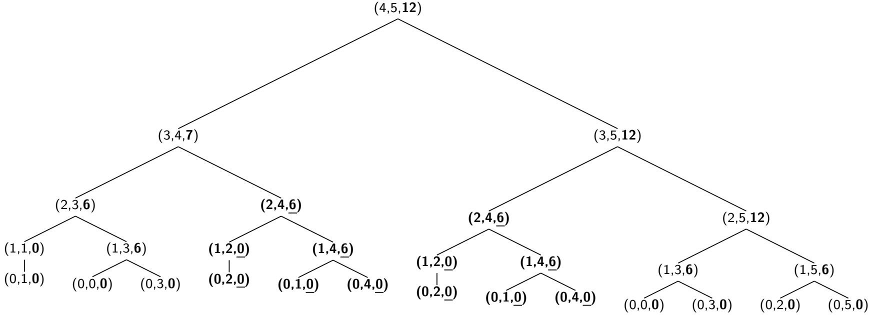
$$\text{mochila}(\{0, 1, \ldots, n\}, \{0, 1, \ldots, W\})$$

La mochila 0/1. Subproblemas repetidos

En efecto, se resuelven muchos subproblemas iguales:

- Ejemplo: $\begin{array}{l} n = 4,\ \ W = 5 \\ w = (3, 2, 1, 1) \\ v = (6, 6, 2, 1) \end{array}$

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



Solución recursiva eficiente guardando resultados parciales (= memoización)

```
const int SENTINEL = -1;
double knapsack(
    vector< vector< double >> &M,                                    // Storage
    const vector<double> &v, const vector<unsigned> &w,     // values & weights
    int n, unsigned W                        // num. of objects & Knapsack limit
) {
    if( M[n][W] != SENTINEL ) return M[n][W];               // if it is known ...
    if( n == 0 ) return M[n][W] = 0.0;                      // base case

    double S1 = knapsack( M, v, w, n-1, W );
    double S2 = numeric_limits<double>::lowest();
    if (w[n-1] <= W) S2 = v[n-1] + knapsack( M, v, w, n-1, W - w[n-1]);
    return M[n][W] = max( S1, S2 );             // store and return the solution
}
//--------------------------------------------------------------------
double knapsack(
    const vector<double> &v, const vector<unsigned> &w, int n, unsigned W
) {
    vector< vector< double >> M( n+1, vector<double>( W+1, SENTINEL)); // init.
    return knapsack( M, v, w, n, W );
}
```

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

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

El 60 % de las celdas no se usan.

40	Solución al problema
	Contorno o perfil
	Celdas sin usar

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

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

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

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

La mochila 0/1. Versión iterativa

Una solución iterativa

```
double knapsack(
     const vector<double> &v,  // values
    const vector<unsigned> &w, // weights
    int last_n,                     // assessed object
    unsigned last_W                // Knapsack limit weight
) {
    vector< vector< double >> M( last_n+1, vector<double>(last_W+1));

    for( unsigned W = 0; W <= last_W; W++ ) M[0][W] = 0;        // no objects
    for( unsigned n = 0; n <= last_n; W++ ) M[n][0] = 0;        // nothing fits

    for( unsigned n = 1; n <= last_n; n++ )
        for( unsigned W = 1; W <= last_W; W++ ) {
            double S1 = M[n-1][W];
            double S2 = numeric_limits<double>::lowest();
            if( W >= w[n-1] )                    // if it fits ...
                S2 = v[n-1] + M[n-1][W-w[n-1]]; // try to put it
            M[n][W] = max( S1, S2 );          // store the best
        }

    return M[last_n][last_W];
}
```

- 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

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

40	Solución al problema
	Contorno o perfil

¡ Se calculan todas las celdas !

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

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

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

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

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

La mochila 0/1. Versión iterativa

- 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 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 las 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

La mochila 0/1

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?

Solución iterativa economizando memoria

```
double knapsack(
    const vector<double> &v, const vector<unsigned> &w, // data vectors
    int last_n, unsigned last_W     // num. objects & Knapsack limit weight
) {
    vector<double> v0(last_W+1);
    vector<double> v1(last_W+1);

    for( unsigned W = 0; W <= last_W; W++ ) v0[W] = 0;        // no objects

    for( int n = 1; n <= last_n; n++ ) {
        for( unsigned W = 1; W <= last_W; W++ ) {
            double S1 = v0[W];
            double S2 = numeric_limits<double>::lowest();
            if( W >= w[n-1] )                      // if it fits ...
                S2 = v[n-1] + v0[W-w[n-1]]; // try to put it
            v1[W] = max( S1, S2 );        // store the best
        }
      swap(v0,v1);
    }
    return v0[last_W];
}
```

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

Una solución iterativa con almacenamiento de todas las decisiones

```
double knapsack(                        // in trace we store the taken decision
    const vector<double> &v, const vector<unsigned> &w,   // values & weights
    int last_n, unsigned last_W,         // assessed object  & Knapsack limit
    vector<vector<bool>> &trace          // trace (true->store, false->don't)
) {
    vector< vector< double >> M( last_n+1, vector<double>(last_W+1));
    trace = vector<vector<bool>>( last_n+1, vector<bool>(last_W+1));

    for( unsigned W = 0; W <= last_W; W++ ) {
        M[0][W] = 0;            // no objects
        trace[0][W] = false;    // I don't take it
    }

    for( int n = 1; n <= last_n; n++ )
        for( unsigned W = 1; W <= last_W; W++ ) {
            double S1 = M[n-1][W];
            double S2 = numeric_limits<double>::lowest();
            if( W >= w[n-1] )                        // if it fits ...
                S2 = v[n-1] + M[n-1][W-w[n-1]];      // try to put it
            M[n][W] = max( S1, S2 );                 // store the best
            trace[n][W] = S2 > S1;                   // if true I take it
        }
    return M[last_n][last_W];
}
```

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

Extracción de las decisiones que interesan

```
void parse(
    const vector<unsigned> &w,      // weights
    const vector<vector<bool>> &trace,   // solutions
    vector<bool> &sol
) {
    unsigned last_n = trace.size()-1;
    int W = trace[0].size()-1;

    for( int n = last_n; n > 0; n-- ) {
        if( trace[n][W] ) {
            sol[n-1] = true;
            W -= w[n-1];
        } else {
            sol[n-1] = false;
        }
    }
}
```

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

Extracción de la selección (directamente del almacén)

```
void parse(
    const vector<vector<double>> &M,
    const vector<double> &v, const vector<unsigned> &w,    // values & weights
    int n, unsigned W,                               // num. of objects & Knapsack limit
    vector<bool> &sol
){
    if( n == 0 ) return;

    double S1 = M[n-1][W];
    double S2 = numeric_limits<double>::lowest();
    if (W >= w[n-1] )
        S2 = v[n-1] + M[n-1][W-w[n-1]];

    if( S1 >= S2 ) {
        sol[n-1] = false;
        parse( M, v, w, n-1, W, sol );
    } else {
        sol[n-1] = true;
        parse( M, v, w, n-1, W - w[n-1], sol );
    }
}
```

Índice

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 ($i = 1, 2, \ldots, n$) 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}!)

- Buscamos una descomposición

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

  por la que se obtenga el precio máximo

- El precio es

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

- 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:

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

Problema 2: Corte de tubos

Solución recursiva (cálculo de la ganancia máxima)

```
int tube_cut(
    const vector<int> &p,        // tube length prices
    const int l                  // assessed length
) {

    if( l == 0 )
        return 0;

    int q = numeric_limits<int>::lowest();  // q ~ -∞
    for( int i = 1; i <= l; i++ )
        q = max( q, p[i] + tube_cut( p, l-i ));

    return q;
}
```

- Es ineficiente porque hay subproblemas repetidos

Solución recursiva (extracción del corte óptimo) /1

```
int trace_tube_cut( const vector<int> &p, const int n, vector<int> &trace ) {
    if( n == 0 ) {    // trace stores for each length which is the optimal cut
        trace[n] = 0;
        return 0;
    }
    int q = numeric_limits<int>::lowest();
    for( int i = 1; i <= n; i++ ) {
        int aux = p[i] + trace_tube_cut( p, n-i, trace);
        if( aux > q ) {    // Maximum gain
            q = aux;
            trace[n] = i;
        }
    }
    return q;
}
//-------------------------------------------------------------
vector<int> trace_tube_cut( const vector<int> &p, const int n ) {
    vector<int> trace(n+1);
    trace_tube_cut(p, n, trace);
    return trace;
}
```

Problema 2: Corte de tubos

Solución recursiva (extracción del corte óptimo) /2

```
vector<int> parse(
    const vector<int> &trace
) {
    vector<int> sol(trace.size(),0);     // How many cuts for each size

    int l = trace.size()-1;
    while( l != 0 ) {
        sol[trace[l]]++;                 //where to cut
        l = l - trace[l];                // the rest
    }
    return sol;
}
//-------------------------------------------------------------------
...
            vector<int> trace = trace_tube_cut( price, n );
            vector<int> sol = parse(trace);
            for( unsigned i = 0; i < sol.size(); i++)
                if( sol[i] != 0 )
                  cout << sol[i] << " cuts of length " << i << endl;
...
```

- Complejidad de la solución recursiva:

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

- 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

```
const int SENTINEL = -1;

int tube_cut(
    vector<int> &M,             // Sub-problem Storage
    const vector<int> &p, int l
) {
    if( M[l] != SENTINEL ) return M[l];// is known?

    if( l == 0 ) return M[0] = 0;

    int q = numeric_limits<int>::lowest();
    for( int i = 1; i <= l; i++ )
        q = max( q, p[i] + tube_cut( M, p, l-i));

    return M[l] = q;      // store solution & return
}

int tube_cut( const vector<int> &p, int l ) {
    vector<int> M(l+1,SENTINEL); //  initialization
    return tube_cut( M, p, l );
}
```

- 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

```
vector<int> memo_tube_cut(const vector<int> &p, int l){
    vector<int> M(l+1,SENTINEL); //  initialization
    tube_cut( M, p, l );
    return M;
}

vector<int> parse( const vector<int> &M, const vector<int> &p ) {
    vector<int> sol(M.size(), 0);

    int l = M.size() - 1;
    while( l != 0 ) {
        for( int i = 1; i <= l; i++ ) {
            if( M[l] == p[i] + M[l-i] ) {
                sol[i]++;
                l -= i;
                break;
            }
        }
    }
    return sol;
}
```

Problema 2: Corte de tubos

Solución iterativa (directa)

```
int tube_cut(
    const vector<int> &p,  // tube length prices
    int n                  // assessed length
) {
    vector<int> M(n+1);   // Sub-problem storage

    for( int l = 0; l <= n; l++ ) {

        if( l == 0 ) {            // base case
            M[0] = 0;
            continue;
        }

        int q = numeric_limits<int>::lowest();
        for( int i = 1; i <= l; i++ )
            q = max( q, p[i] + M[l-i]);
        M[l] = q;               // Store solution
    }
    return M[n];
}
```

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

Problema 2: Corte de tubos

Solución iterativa (mejor)

```cpp
int tube_cut(
    const vector<int> &p,   // tube length prices
    int n                   // assessed length
) {
    vector<int> M(n+1);   // Sub-problem storage

    M[0] = 0;                          // Base case
    for( int l = 1; l <= n; l++ ) {
        int q = numeric_limits<int>::lowest();
        for( int i = 1; i <= l; i++ )
            q = max( q, p[i] + M[l-i]);
        M[l] = q;                 // Store solution
    }

    return M[n];
}
```

- 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 $\mathtt{p}$).

Índice

¿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

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

Índice

Problema 3: El coeficiente binomial

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

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

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

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

```
unsigned binomial( unsigned n, unsigned r){

    if ( r == 0 || r == n )
        return 1;

    return binomial( n-1, r-1 ) + binomial( n-1, r );
}
```

- 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:

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

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

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

- Por tanto:

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

- Si, en cambio, suponemos:

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

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

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

- 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!

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)	1.3×10^{06}
(40, 7)	3.7×10^{07}
(40, 9)	5.4×10^{08}
(40, 11)	4.6×10^{09}
(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
- 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.

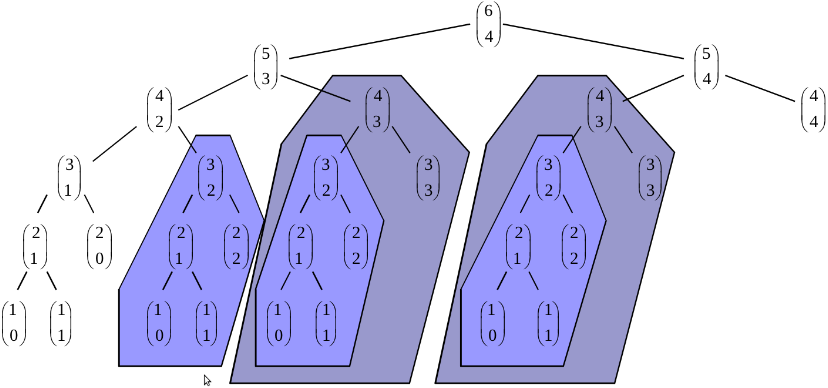
  $\Rightarrow$ Esto lleva a complejidades prohibitivas (p.e. exponenciales)
- Pero, ¡el total de subproblemas diferentes no es tan grande!
  - sólo hay nr posibilidades distintas

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

```
const unsigned SENTINEL = 0;

unsigned binomial( vector<vector<unsigned>> &M, unsigned n, unsigned r) {

    if( M[n][r] != SENTINEL )
        return M[n][r];
    if( r == 0 || r == n )
        return 1;

    M[n][r] = binomial(M, n-1, r-1) + binomial(M, n-1, r);

    return M[n][r];
}

unsigned binomial( unsigned n, unsigned r) {
    vector<vector<unsigned>> M( n+1, vector<unsigned>(r+1, SENTINEL));
    return binomial( M, n, r);
}
```

Problema 3: El coeficiente binomial

Memoización (para varios problemas) $\{n \geq r,\ n \in \mathbb{N}, r \in \mathbb{N}\}$

```
unsigned binomial( vector<vector<unsigned>> &M, unsigned n, unsigned r) {
    if( M[n][r] != 0 ) return M[n][r];
    if( r == 0 || r == n ) return 1;
    M[n][r] = binomial(M, n-1, r-1) + binomial(M, n-1, r);
    return M[n][r];
}

const unsigned MAX_N = 100;

unsigned binomial( unsigned n, unsigned r) {
    static vector<vector<unsigned>> M;
    static bool initialized = false;

    if( !initialized ) {
        M = vector<vector<unsigned>>(MAX_N, vector<unsigned>(MAX_N, SENTINEL));
        initialized = true;
    }

    return binomial( M, n, r);
}
```

Memoización (functors) $\{n \geq r,\ n \in \mathbb{N}, r \in \mathbb{N}\}$

```
const unsigned SENTINEL = 0;
const unsigned MAX_N = 100;

class Binomial {
public:
    Binomial( unsigned max_n = MAX_N ) : M(
        vector<vector<unsigned>>(max_n+1, vector<unsigned>(max_n+1, SENTINEL))
    ){};

    unsigned operator()( unsigned n, unsigned r ) {
        if( M[n][r] != SENTINEL ) return M[n][r];
        if( r == 0 || r == n ) return 1;
        M[n][r] = operator()(n-1, r-1) + operator()( n-1, r);
        return M[n][r];
    }

private:
    vector<vector<unsigned>> M;
};

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.
(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

$(n, r) = \binom{n}{r}$	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 $(n/2)^2 + n \in \Theta(n^2)$.

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:

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

Puesto que

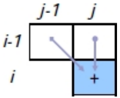
$$\binom{m}{0} = \binom{m}{m} = 1, \quad \forall m \in \mathbb{N}$$

Problema 3: El coeficiente binomial

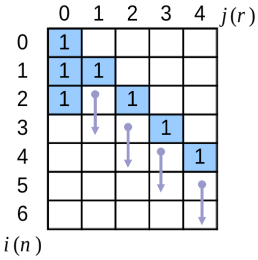
Recorrido de los subproblemas:

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

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



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



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

```
unsigned binomial(unsigned n,unsigned r){
    unsigned M[n+1][r+1];

    for (unsigned i=0; i <= n-r; i++) M[i][0]= 1;
    for (unsigned i=1; i <= r;   i++) M[i][i]= 1;

    for (unsigned j=1; j<=r; j++)
        for (unsigned i=j+1; i<=n-r+j; i++)
            M[i][j]= M[i-1][j-1] + M[i-1][j];
    return M[n][r];
}
```

Problema 3: El coeficiente binomial

Solución trivial de programación dinámica $\{n \geq r,\ n \in \mathbb{N}, r \in \mathbb{N}\}$

```
unsigned binomial(unsigned n, unsigned r){
    unsigned M[n+1][r+1];

    for (unsigned i=0; i <= n-r; i++) M[i][0]= 1;
    for (unsigned i=1; i <= r;   i++) M[i][i]= 1;

    for (unsigned j=1; j<=r; j++)
        for (unsigned i=j+1; i<=n-r+j; i++)
            M[i][j]= M[i-1][j-1] + M[i-1][j];
    return M[n][r];
}
```

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

```
unsigned binomial(unsigned n, unsigned r){
    unsigned M[n+1][r+1];

    for (unsigned i=0; i <= n-r; i++) M[i][0]= 1;
    for (unsigned i=1; i <= r;   i++) M[i][i]= 1;

    for (unsigned j=1; j<=r; j++)
        for (unsigned i=j+1; i<=n-r+j; i++)
            M[i][j]= M[i-1][j-1] + M[i-1][j];
    return M[n][r];
}
```

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

- 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?

Índice

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

```
Solution DC( Problem p ) {
  if( is_simple(p) ) return trivial(p);

  list<Solution> s;
  for( Problem q : divide(p) ) s.push_back( DC(q) );
  return combine(s);
}
```

Esquema programación dinámica (recursiva)

```
Solution DP( Problem p ) {
  if( is_solved(p) ) return M[p];
  if( is_simple(p) ) return M[p] = trivial(p); // or simply: return trivial(p)

  list<Solution> s;
  for( Problem q : divide(p) ) s.push_back( DP(q) );
  M[p] = combine(s);
  return M[p];
}
```

Programación dinámica: Estrategia de diseño

Paso a programación dinámica iterativa:

Esquema programación dinámica (iterativa)

```
Solution DP( Problem P) {
    vector<Solution> M;
    list<Problem> e = enumeration(P);

    while( !e.empty() ) {
        Problem p = e.pop_front();
        if( is_simple(p) )
            M[p] = trivial(p);
        else {
            list<Solution> s;
            for( Problem q : divide(p) ) s.push_back( M[q] );
            M[p] = combine(s);
        }
    }
    return M[P];
}
```

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)