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

18-01-2018 (379)

Índice

Índice

Obtener el valor del coeficiente binomial

- Identidad de Pascal:

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

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:

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

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

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

- Por tanto:

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

La solución recursiva es ineficiente

- Si suponemos

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

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

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

- Por tanto:

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

- Combinando ambas posibilidades:

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

¡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^{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

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.

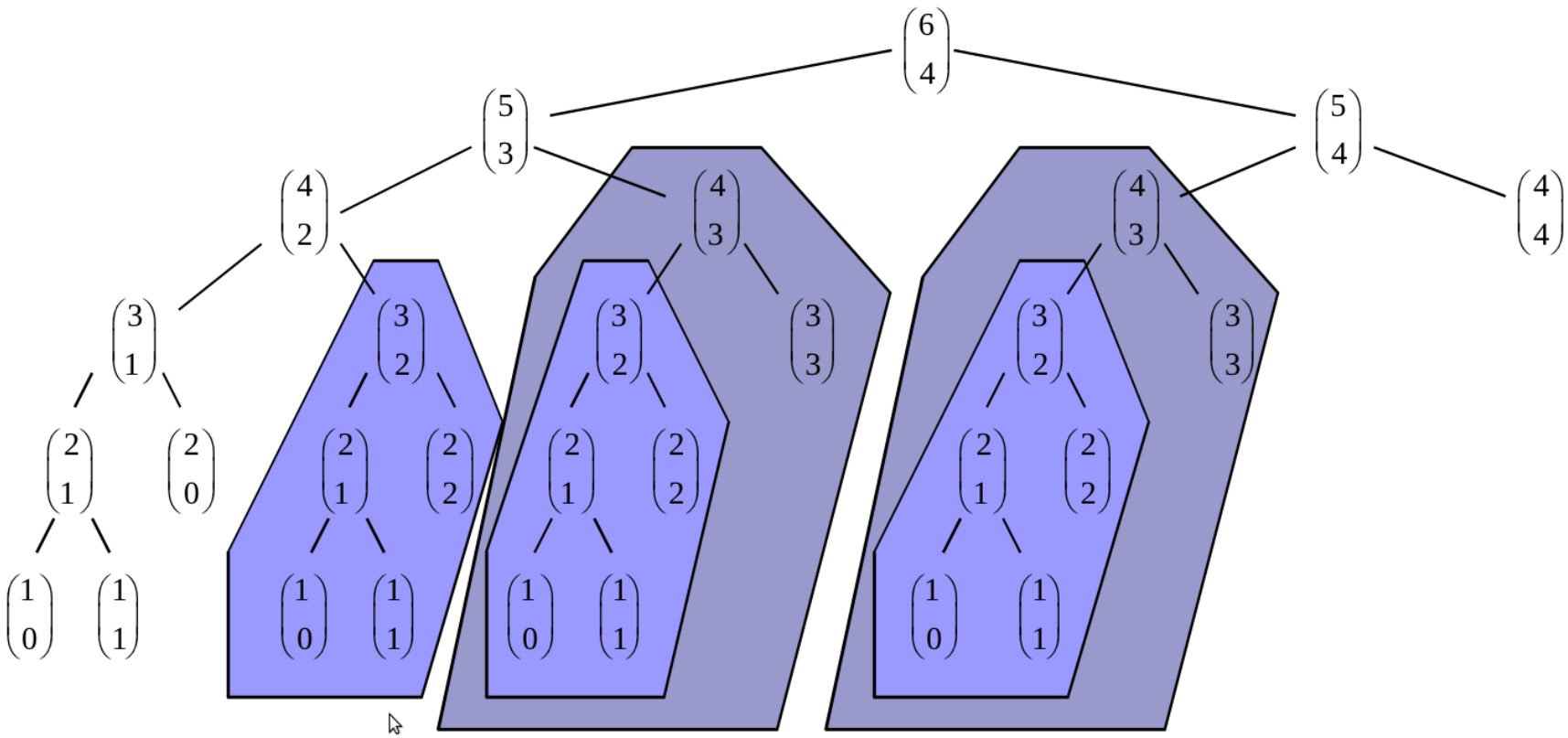
  $\Rightarrow$ Esto lleva a complejidades prohibitivas (p.e. exponenciales)
- Pero, ¡el total de subproblemas diferentes no es tan grande!
  - sólo hay nr posibilidades distintas

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

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

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?

$\Rightarrow$ almacenar los valores ya calculados para no recalcularlos:

Una solución recursiva mejorada $\{n \geq r,\ n \in \mathbb{N}, r \in \mathbb{N}\}$

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

Memoización (para varios problemas relacionados)

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

```
1  class Binomial {
2  public:
3    Binomial() {
4      for( unsigned i = 0; i < 100; i++ )
5        for( unsigned j = 0; j < 100; j++ )
6      M[i][j] = 0;
7    }
8    unsigned operator()( unsigned n, unsigned r ) {
9      if( M[n][r] != 0 )      return M[n][r];
10     if( r == 0 || r == n ) return 1;
11     M[n][r] = operator()(n-1, r-1) + operator()( n-1, r);
12     return M[n][r];
13   }
14 private:
15   unsigned M[100][100];
16 };
17
18 Binomial binomial;
```

Algunos números

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

$$M[i][0] = 1 \qquad \forall i = 1 \cdots (n-r)$$
$$M[i][i] = 1 \qquad \forall i = 1 \cdots r$$

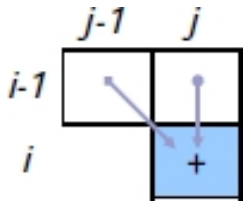
Puesto que

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

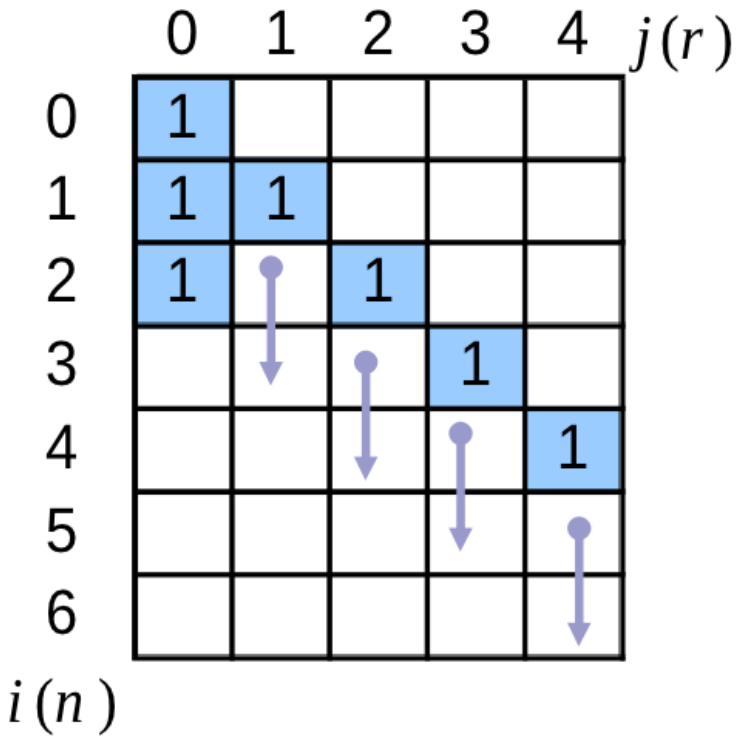
Recorrido de los subproblemas

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

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

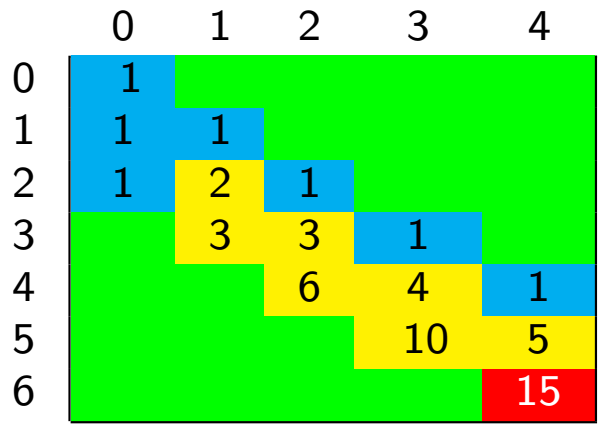


$$\forall (i,j) : (1 \leq j \leq r,\ j+1 \leq i \leq n-r+j)$$



Una solución polinómica (mejorable)

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



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)

Complejidad espacial

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?

Mejorando la complejidad espacial

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

Índice

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}!)

Corte de tubos

- 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:

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

Corte de tubos

Corte de tubos: solución recursiva descendente

```
1  precio cortar_tubo( precio p[], longitud n ) {
2
3    if (n==0)
4      return 0;
5
6    precio q = -1;
7    for ( unsigned i = 1; i <= n; i++ )
8       q = max( q, p[i] + cortar_tubo( p, n-i ) );
9
10   return q;
11 }
```

- Es ineficiente

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:

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

Corte de tubos

Corte de tubos: solución recursiva con almacén

```
1  precio cortar_tubo( precio r[], precio p[], longitud n ){
2      if ( r[n] >= 0 ) return r[n];
3      if ( n == 0 ) return 0;
4      precio q = -1;
5      for ( unsigned i = 1; i <= n; i++ )
6          q = max( q, p[i] + cortar_tubo( r, p, n-i ) );
7      r[n] = q;
8      return q;
9  }
10
11 precio cortar_tubo( precio p[], longitud n ){
12     precio r[n+1];
13     for( unsigned i = 0; i <= n; i++ )
14         r[i] = -1;
15     return cortar_tubo( r, p, n );
16 }
```

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

Corte de tubos

Corte de tubos: Solución iterativa

```
1  precio cortar_tubo(precio p[], longitud n) {
2    precio r[n+1];
3
4    r[0] = 0;
5    for ( indice j = 1; j <= n; j++ ) {
6      precio q = -1;
7      for ( indice i=1; i <= j; i++ )
8          q = max( q, p[i] + r[j-i] );
9      r[j] = q;
10   }
11
12   return r[n];
13 }
```

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

Índice

¿Qué hemos aprendido de estos ejemplos?

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 de subproblemas** (*memoización*) . . .
- . . . a expensas de aumentar la complejidad espacial.

Esto se llama Programación Dinámica.

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.

- Esto también se conoce como principio de optimalidad.
- Esta es una condición necesaria para que se puede aplicar Programación Dinámica.
- Ejemplos:
  - Cálculo del coeficiente binomial
  - Corte de tubos
  - Quicksort
  - Mergesort

Paso de DyV a PD

Esquema DyV

```
1 Solucion DyV( Problema p ) {
2   if( es_pequeno(p) ) return trivial(p);
3
4   list<Solucion> s;
5   for( Problema q : descomponer(p) ) s.push_back( DyV(q) );
6   return combinar(s);
7 }
```

Esquema Programación dinámica (recursiva)

```
1 Solucion PD( Problema p ) {
2   if( ya_resuelto(p) ) return A[p];
3   if( es_pequeno(p) ) return trivial(p);
4
5   list<Solucion> s;
6   for( Problema q : descomponer(p) ) s.push_back( PD(q) );
7   A[p] = combinar(s);
8   return A[p];
9 }
```

Paso A PD iterativa

Esquema Programación dinámica (recursiva)

```
1  Solucion PD( Problema P) {
2    vector<Solucion> A;
3    Enumeracion e(P);
4
5    while( !empty(e) ) {
6      Problem p = pop_next(e);
7      if( es_pequeno(p) )
8        A[p] = trivial(p);
9      else {
10       list<Solucion> s;
11       for( Problema q : descomponer(p) ) s.push_back( A[q] );
12       A[p] = combinar(s);
13     }
14   return A[P];
15 }
```

Le enumeración ha de cumplir:

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

Ejemplos de aplicación

- Problemas clásicos para los que resulta eficaz la programación dinámica
  - El problema de la mochilla 0-1 (que veremos a continuación)
  - 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)

Índice

El problema de la mochila (*Knapsack problem*)

- Sean n objetos con valores ($v_i \in R$) y pesos ($p_i \in R^{>0}$) conocidos.
- Sea una mochila con capacidad máxima de carga P.
- ¿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
  - Los pesos son cantidades discretas o discretizables
    - Se utilizarán para indexar una tabla
    - Una versión menos general que suaviza su dificultad

Formalización

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

Subestructura óptima (I)

- Sea $(x_1, x_2 \ldots x_n)$ una secuencia óptima de decisiones para el problema $mochila(n, P)$
  - Si $x_n = 0$ entonces $(x_1 \ldots x_{n-1})$ es una secuencia óptima para el subproblema $mochila(n-1, P)$
  - Si $x_n = 1$ entonces $(x_1 \ldots x_{n-1})$ es una secuencia óptima para el subproblema $mochila(n-1, P-p_1)$

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.

$\Rightarrow$ La solución al problema presenta una subestructura óptima

Aproximación matemática

- Se toman decisiones en orden descendente: x_n, x_{n-1}, ... 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 p_i
- Se selecciona la alternativa que mayor ganancia global resulte

Solución

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

$$\texttt{Mochila}(n, P) = \text{máx} \begin{cases} \texttt{Mochila}(n-1, P) \\ \texttt{Mochila}(n-1, P-p_n) + v_n \end{cases}$$

con:

- $\texttt{Mochila}(i, P) = -\infty$ si $P < 0$
- $\texttt{Mochila}(0, P) = 0, P \geq 0$

Una solución recursiva

- Escribiendolo en forma de programa:

Solución recursiva (ineficiente)

```
1  vector <float> v;
2  vector <unsigned> p;
3
4  float Mochila (int i, unsigned P){
5    if( i < 0 ) return 0;
6
7    float S1 = 0.0;
8    if ( p[i] <= P )  //Aun hay sitio en la mochila para el objeto
9         S1= v[i] + Mochila(i-1,P-p[i]);
10   float S2= Mochila(i-1,P); //Se descarta el objeto
11
12   return max(S1,S2); //lo mejor de entre tomarlo o no tomarlo
13 }
```

Versión recursiva: Complejidad temporal

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

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

El témino general queda como:

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

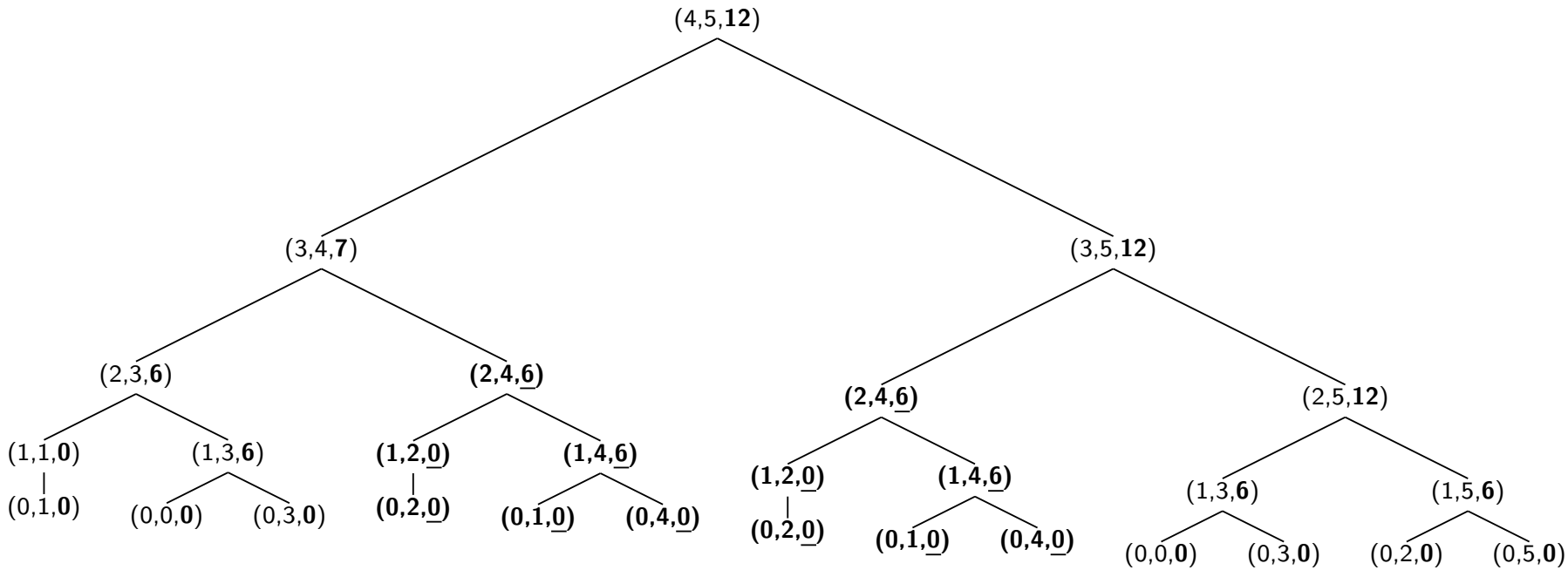
Que terminará cuando $n - i = 0$, o sea:

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

Version recursiva: Subproblemas repetidos

- Ejemplo: $n = 4,\ P = 5$
  $p = (3, 2, 1, 1)$
  $v = (6, 6, 2, 1)$

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



Almacén de resultados parciales

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

$\mathbf{T}[0\ldots5][0\ldots11]$	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
$p_1 = 2, v_1 = 1$	0	0	1	1	1	1	1	1	1	1	1	1
$p_2 = 2, v_2 = 7$	0	0	7	7	8	8	8	8	8	8	8	8
$p_3 = 5, v_3 = 18$	0	0	7	7	8	18	18	25	25	26	26	26
$p_4 = 6, v_4 = 22$	0	0	7	7	8	18	22	25	29	29	30	40
$p_5 = 7, v_5 = 28$	0	0	7	7	8	18	22	28	29	35	35	40

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

40

Solución al problema

Contorno o perfil

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

$$T[5][11] = \text{máx}\left(T[4][11], T[4][11-p_5] + v_5\right) = \text{máx}(40, 36).$$ 5 no se toma

$$T[4][11] = \text{máx}\left(T[3][11], T[3][11-p_4] + v_4\right) = \text{máx}(26, 40).$$ 4 sí se toma

$$T[3][5] = \text{máx}\left(T[2][5], T[2][5-p_3] + v_3\right) = \text{máx}(8, 18).$$ 3 sí se toma

$$T[2][0] = T[1][0] = 0.$$ 1 y 2 no se toman

Una versión iterativa

Una solución iterativa

```
1  float Mochila (int n, unsigned P){
2      float M[n+1][P+1];
3
4      for (unsigned i=0; i<=n; i++) M[i][0]=0; //sin espacio, ganancia 0
5      for (unsigned j=1; j<=P; j++) M[0][j]=0; //sin objetos, ganancia 0
6
7      for (unsigned i=1; i<=n; i++)
8          for (unsigned j=1; j<=P; j++) {
9              float S1 = 0.0;
10             if (p[i] <= j ) // Aun hay sitio en la mochila
11                 S1 = v[i] + M[i-1][j-p[i]];
12             float S2 = M[i-1][j];
13             M[i][j] = max(S1,S2);
14         }
15     return M[n][P];
16 }
```

Iterativo: Complejidad temporal y espacial

- Complejidad temporal

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

  Por tanto,

$$T(n,P) \in \Theta(nP)$$

- Complejidad espacial

$$T_S(n,P) \in \Theta(nP)$$

  - la complejidad espacial es mejorable ...

PD-recursiva

Una solución recursiva

```
1  float Mochila( unsigned n, unsigned P ) {
2    float M[100][100];
3    for( int i = 0; i <= n; n++ )
4      for( int j = 0; j <= P; j++ )
5        M[i][j] = -1;
6
7    return Mochila( M, n, P );
8  }
9
10 float Mochila( float M[100][100], unsigned n, unsigned P){
11   if( M[n][P] > 0 ) return M[n][P];
12   if( n < 0 ) return 0;
13
14   float S1 = 0.0;
15   if (p[n] <= P)
16     S1 = Mochila( n-1, P-p[i] );
17   float S2 = Mochila( n-1, P );
18   M[n][P]=max( S1, S2 );
19
20   return M[n][P];
21 }
```

PD-recursiva con almacén

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

$\mathbf{T}[0\ldots5][0\ldots11]$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0		0		0	
$p_1 = 2, v_1 = 1$	0		1	1	1	1	1			1		1
$p_2 = 2, v_2 = 7$	0				8	8	8					8
$p_3 = 5, v_3 = 18$					8	18						26
$p_4 = 6, v_4 = 22$					8							40
$p_5 = 7, v_5 = 28$												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

$$T[5][11] = \text{máx}\left(T[4][11],\ T[4][11-p_5]+v_5\right) = \text{máx}(40, 36).$$ 5 no se toma

$$T[4][11] = \text{máx}\left(T[3][11],\ T[3][11-p_4]+v_4\right) = \text{máx}(26, 40).$$ 4 sí se toma

$$T[3][5] = \text{máx}\left(T[2][5],\ T[2][5-p_3]+v_3\right) = \text{máx}(8, 18).$$ 3 sí se toma

$$T[2][0] = T[1][0] = 0.$$ 1 y 2 no se toman

Conclusiones

- La complejidad temporal de la solución obtenida mediante programación dinámica está en $\Theta(nP)$
  - Un recorrido descendente a través de la tabla permite obtener también, en tiempo $\Theta(n)$, la secuencia óptima de decisiones tomadas.
- Si P es muy grande entonces las solución obtenida mediante programación dinámica no es buena
- Si los pesos p_i o la capacidad P pertenecen a dominios continuos (p.e. los reales) entonces esta solución no sirve
- La complejidad espacial de la solución obtenida se puede reducir hasta $\Theta(P)$
- 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

Ejercicios Propuestos

- ¿Se puede reducir la complejidad espacial de la solución iterativa propuesta?
  - ¿Cuántos vectores harían falta y de qué tamaño?
  - ¿Sacrifica esto la complejidad temporal?
- Escribe una función para obtener la secuencia de decisiones óptima a partir de la tabla completada por el algoritmo iterativo
  - ¿Qué complejidad temporal tiene esa función?