Técnicas de diseño de Algoritmos (parte II)

Prof. Cristian Cappo

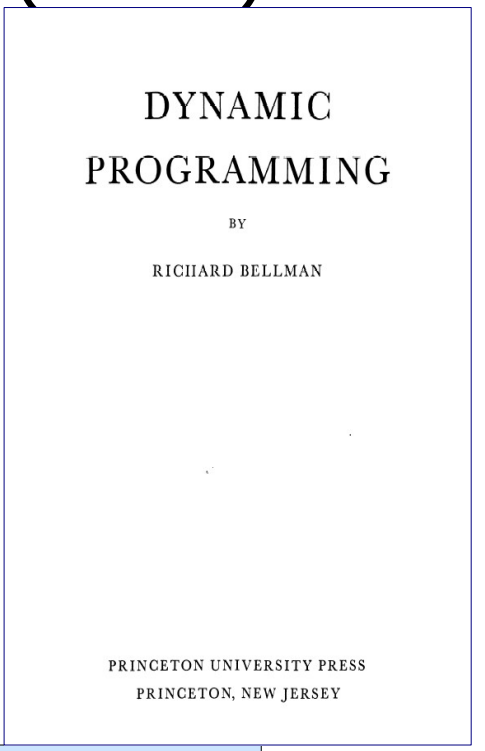
Mayo/2018

Contenido

- Programación Dinámica
- Vuelta atrás o *Backtracking*

Técnicas de diseño de Alg.
Programación Dinámica

- Técnica desarrollada por Richard Bellman*, quien propuso un esquema de solución a problemas de optimización (1957)
- Es aplicable cuando NO es posible dividir el problema y combinar las soluciones como en DyV o algoritmos greedy.
- Programación: se refiere al método tabular, no a escribir código para computadora



* Richard Bellman, *Dynamic Programming*. Princeton University Press. 1957

Programación Dinámica

- Podemos subdividir el problema en subproblemas tanto como sea necesario, solo que esto puede llevarnos a un algoritmo de tiempo exponencial.
- Pero en la mayoría de los casos solo se tiene un número polinomial de subproblemas, por lo que una estrategia posible sería no volver a calcular subproblemas para los que ya se tiene la solución.
- Para ello, podemos guardar las respuestas en una tabla y consultarla las veces que necesitamos.

Programación Dinámica

- Es similar a DyV, en el sentido de uso de recurrencia pero diferente porque se basa en que los subproblemas NO son independientes entre sí.
- *Así,* la PD es aplicable a subproblemas que NO son independientes entre sí. Resuelve un subproblema, lo guarda en una tabla de respuestas evitando así recalcular los subproblemas comunes [CLRS].
- Típicamente es aplicable a problemas de optimización.

Programación Dinámica

Para desarrollar una solución utilizando PD se debe:

- Caracterizar la estructura de una solución óptima. Debemos verificar que podemos lograr la solución como una sucesión de decisiones y verificar que esta cumple el principio de optimalidad.
- Definir recursivamente el valor de una solución óptima. Definimos así la solución en forma recursiva.
- Encontrar el valor de una solución óptima de forma *botton-up* (abajo para arriba), guardando los valores de las soluciones a los problemas parciales para luego reutilizarlos.
- Construir la solución óptima a partir de la solución calculada.

Programación Dinámica

- **Principio de optimalidad** de Bellman:

  "Cualquier subsecuencia de decisiones de una secuencia óptima de decisiones que resuelve un problema también debe ser óptima respecto al subproblema que resuelve"

- Suponga que un problema se resuelve tras tomar una secuencia $d_1, d_2,..,d_n$ de decisiones

- Si hay ***d*** opciones posibles para cada una de las decisiones, una técnica de fuerza bruta exploraría un total de $\boldsymbol{d^n}$ secuencias posibles de decisiones (lo que se llama *explosión combinatoria*).

- La técnica de PD evita explorar todas las secuencias posibles resolviendo subproblemas de tamaño creciente y almacenando en una tabla de las soluciones óptimas de los mismos facilitando así la solución del problema global.

Programación Dinámica

- Ejemplo de problema con subproblemas solapados (o NO independientes entre si)

```
fibo(1) = 1          n = 1
fibo(2) = 1          n = 2
fibo(n) = fibo(n-1) + fibo(n-2) n > 2
```

El tiempo asintótico de esta implementación es

T(n) = Ω (ϕ^n) donde ϕ es la constante aúrea $\frac{1+\sqrt{(5)}}{2} = 1,618033..$

Programación Dinámica

- Con la implementación anterior calcular F(46) en mi PC y en leng. C , tarda un poco más de 10 segundos!!
- En realidad tenemos subproblemas compartidos. Por ejemplo en F(5)

```
( F(4) + F(3) )

(( F(3) + F(2)) + ( F(2) + F(1) ))

((( F(2) +F(1)) + (F(1)+F(0)) + ((F(1)+F(0)) + F(1)))

(((F(1)+F(0))+F(1))) + (F(1)+F(0))) + ((F(1)+F(0)) + F(1)))
```

Programación Dinámica

Considerar el siguiente código en C++

```
static const int   ARRAY_SIZE = 1000;
double * array = new double[ARRAY_SIZE];

array[0]  = 1.0;
array[1]  = 1.0;


for ( int  i = 2; i < ARRAY_SIZE; ++i ) {
     array[i]  = 0.0;
}

 double F( int  n ) {
      if ( array[n] == 0.0 ) {
          array[n]  = F( n – 1 ) + F( n – 2 );
      }
      return array[n];
 }
```

Con este código tenemos un tiempo *O(n)*!!

Programación Dinámica

Podemos hacer que no tenga “límite” (1000) usando algún esquema de hashing (ej. en C++)

```
#include <map>

/* Calcular en enesimo termino de Fibonacci */
double F( int n ) {
    static std::map<int, double> memo;
            // compartido con todas las llamadas a F
            // la clave es int, el valor es double

        if ( n <= 1 ) {
            return 1.0;
        } else {
            if ( memo[n] == 0.0 ) {
               memo[n] = F(n - 1) + F(n - 2);
            }

           return memo[n];
        }
}
```

Programación Dinámica
Multiplicación de matrices

- Suponga que la matriz **A** que es $k \,.\, m$ y la matriz **B** que es $m \,.\, n$. Entonces **AB** es $k \,.\, n$ y calcular **AB** es $\Theta(kmn)$
- De hecho, el número de multiplicaciones es dado por kmn

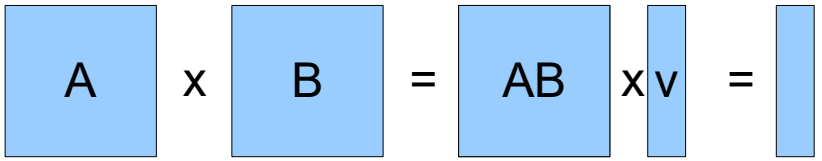
Algoritmo de multiplicación de una matriz A por otra B

```
k = A.length;
n = B[0].length;
m = A[0].length;
for ( i = 0 ; i < k ; i++)  {
   for ( j = 0; j < n; j++ ) {
       AB[i][j] = 0;
       for ( c=0; c < m ; c++)
           AB[i][j] = AB[i][j] + A[i][c]*B[c][j];
      }
}
```

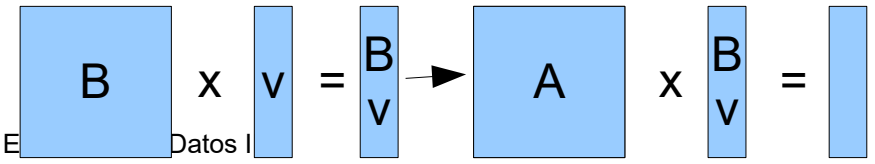
Programación Dinámica
Multiplicación de matrices

- Una selección de orden apropiado puede ser crítico en el tiempo de ejecución.
- Por ejemplo, considere un vector ***v*** , que puede ser visto como una matriz de $n \times 1$.
- Si tenemos las matrices **A** y **B** de $n \times n$, entonces
  - Calcular **(AB)v** es $\Theta(n^3)$, mientras
  - Calcular **A(Bv)** es $\Theta(n^2)$

Aplicamos propiedad asociativa de la multiplicación



(AB)v >> (n.n.n + n.n.1) = ($n^3 + n^2$) = $\Theta(n^3)$



A(Bv) >> (n.n.1) + (n.n.1)= ($n^2 + n^2$) = $\Theta(n^2)$

Programación Dinámica
Multiplicación de matrices

- Otro ejemplo: suponga que se necesite multiplicar estas matrices

Matriz	Dimensiones
A	20 × 5
B	5 × 40
C	40 × 50
D	50 × 10

Gráficamente podemos ver..

Programación Dinámica
Multiplicación de matrices

- Podemos evaluar las siguientes alternativas de multiplicación:

((AB)C)D = 54000 multiplicaciones

AB = (**20** x 5) x (5 x **40**) = 4000

(AB)C = (**20** x 40) x (40 x **50**)= 40000

((AB)C)D = (**20** x 50) x (50 x **10**) = 10000 (matriz resultante de 20x10)

(AB)(CD) = 32000 multiplicaciones

(A(BC))D = 25000 multiplicaciones

A((BC)D) = 13500 multiplicaciones

A(B(CD)) = 23000 multiplicaciones

Programación Dinámica
Multiplicación de matrices

- En este ejemplo el caso óptimo es 3 veces más rápido que el peor caso.
- El cálculo es relativamente simple para 4 matrices.
- ¿Qué pasa si tenemos un número indeterminado de matrices?

Programación Dinámica
Multiplicación de matrices

Estrategia

Enumeramos las posibles maneras que podemos multiplicar **n** matrices

$$\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4\cdots\mathbf{A}_n$$

- Nosotros podemos definir este número de forma recursiva de la siguiente forma:

  Dado P(n) que representa el número de formas diferentes en el cual el producto puede ser parentizado de forma única.

Programación Dinámica
Multiplicación de matrices

- Claramente P(1) = 1
- Para n > 1, considere la siguiente forma de dividir en subproblemas menores:

$$(\mathbf{A}_1)(\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4\cdots\mathbf{A}_n)$$

$$(\mathbf{A}_1\mathbf{A}_2)(\mathbf{A}_3\mathbf{A}_4\cdots\mathbf{A}_n)$$

$$(\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3)(\mathbf{A}_4\cdots\mathbf{A}_n)$$

$$\vdots$$

$$(\mathbf{A}_1\cdots\mathbf{A}_{n-2})(\mathbf{A}_{n-1}\mathbf{A}_n)$$

$$(\mathbf{A}_1\cdots\mathbf{A}_{n-2}\mathbf{A}_{n-1})(\mathbf{A}_n)$$

Programación Dinámica
Multiplicación de matrices

Para cualquiera de ellos podemos notar que

$$(\mathbf{A}_1)(\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4\cdots\mathbf{A}_n) \qquad \mathrm{P}(1)\mathrm{P}(n-1)$$

$$(\mathbf{A}_1\mathbf{A}_2)(\mathbf{A}_3\mathbf{A}_4\cdots\mathbf{A}_n) \qquad \mathrm{P}(2)\mathrm{P}(n-2)$$

$$(\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3)(\mathbf{A}_4\cdots\mathbf{A}_n) \qquad \mathrm{P}(3)\mathrm{P}(n-3)$$

$$\vdots$$

$$(\mathbf{A}_1\cdots\mathbf{A}_{n-2})(\mathbf{A}_{n-1}\mathbf{A}_n) \qquad \mathrm{P}(n-2)\mathrm{P}(2)$$

$$(\mathbf{A}_1\cdots\mathbf{A}_{n-2}\mathbf{A}_{n-1})(\mathbf{A}_n) \qquad \mathrm{P}(n-1)\mathrm{P}(1)$$

Programación Dinámica
Multiplicación de matrices

- Por ejemplo, para encontrar las formas de parentizar :

$$\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4\mathbf{A}_5$$

Consideramos 4 casos

Caso 1 $(\mathbf{A}_1)(\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4\mathbf{A}_5)$

Caso 2 $(\mathbf{A}_1\mathbf{A}_2)(\mathbf{A}_3\mathbf{A}_4\mathbf{A}_5)$

Caso 3 $(\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3)(\mathbf{A}_4\mathbf{A}_5)$

Caso 4 $(\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4)(\mathbf{A}_5)$

Programación Dinámica
Multiplicación de matrices

- **Caso 1** : $(A_1)(A_2A_3A_4A_5)$

  Hay una sola forma de parentizar el lado izquierdo, sin embargo el derecho requiere más P(4) = 5

$(A_2)(A_3A_4A_5)$ $(A_2)((A_3A_4)A_5)$ ó $(A_2)(A_3(A_4A_5))$

$(A_2A_3)(A_4A_5)$

$(A_2A_3A_4)(A_5)$ $((A_2A_3)A_4)(A_5)$ ó $(A_2(A_3A_4))(A_5)$

Programación Dinámica
Multiplicación de matrices

- **Caso 2**: $(A_1A_2)(A_3A_4A_5)$

  Hay una sola forma de parentizar el lado izquierdo pero dos para parentizar el lado derecho

  $(A_3A_4)A_5 \qquad A_3(A_4A_5)$

- **Caso 3**: $(A_1A_2A_3)(A_4A_5)$

  Hay dos formas de parentizar el lado izquierdo y una forma para el lado derecho (simétrico al anterior)

  $(A_1A_2)A_3 \qquad A_1(A_2A_3)$

Programación Dinámica
Multiplicación de matrices

- **Caso 4**: $(A_1A_2A_3A_4)(A_5)$

  Y las formas de parentizar es simétrico al caso 1

  $(A_1)(A_2A_3A_4)$ $(A_1)((A_2A_3)A_4)$ $(A_1)(A_2(A_3A_4))$

  $(A_1A_2)(A_3A_4)$

  $(A_1A_2A_3)(A_4)$ $((A_1A_2)A_3)(A_4)$ $(A_1(A_2A_3))(A_4)$

Finalmente el total de posibles parentizaciones es P(5) = 5 + 2 + 2 + 5 = 14

Sin embargo el proceso que utilizamos no es sencillo.

Programación Dinámica
Multiplicación de matrices

- El número de posibles ordenaciones es dada por la siguiente recurrencia

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k)\,P(n-k) & n > 1 \end{cases}$$

La recurrencia es resuelta utilizando el número catalán P(n) = C(n-1) donde C(n-1) es el n-1 número Catalán.

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!}$$

Y crece rápidamente

C(1)=1 , C(2)=2 , C(3)=5, C(4)=14, C(5)=42, C(6)=132, C(7)=429, C(8)=1430, C(9)=4862, C(10)=16792

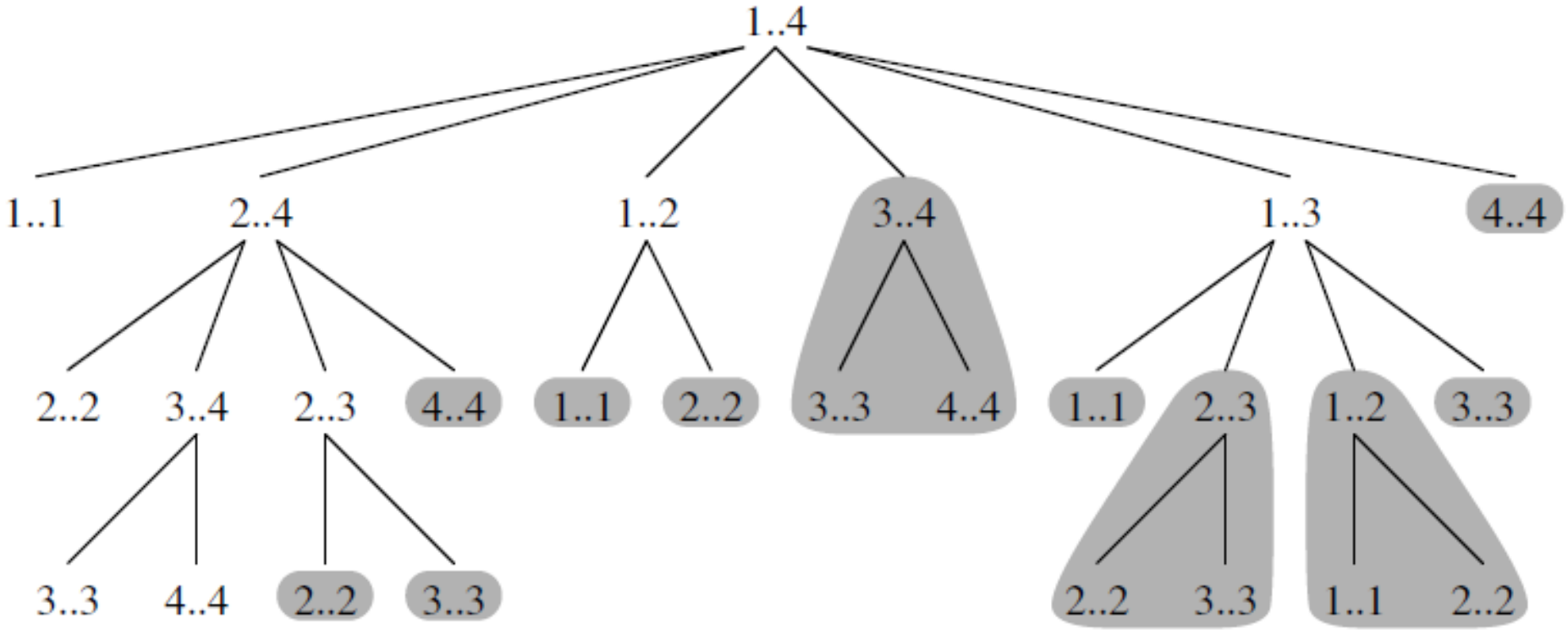
El costo asintótico es C(n) = $\Theta(4^n/n^{3/2})$

Programación Dinámica
Multiplicación de matrices

- Hasta aquí solo nos dimos cuenta que el espacio de búsqueda es muy grande ☹
- Hacemos la siguiente observación (mirando el ejemplo anterior):
  - Nosotros necesitamos solo calcular la secuencia apropiada (y consecuentemente, el número de operaciones) del producto $\mathbf{A}_i \mathbf{A}_{i+1} \cdots \mathbf{A}_{j-1} \mathbf{A}_j$ *UNA SOLA VEZ*
- En teoría, podemos considerar todas las posibles formas antes de determinar cual minimiza el número de operaciones.
- Pero el algoritmo es exponencial??

Programación Dinámica
Multiplicación de matrices

- Por fortuna, existe una considerable cantidad de solapamiento. *En el gráfico un ejemplo de posibilidades para 4 matrices (1..4) en gris los solapamientos*



Programación Dinámica
Multiplicación de matrices

Propuesta*: usamos una tabla para reducir el tiempo de forma significativa.

- Diseñamos entonces la forma de guardar el número mínimo de multiplicaciones necesarias para el producto de $\mathbf{A}_i$ a $\mathbf{A}_j$

* S. Godbole. *On efficient computation of matrix chain products*. IEEE Transactions on Computers. 22(9). 1973

A	$j=1$	2	3	4	$\cdots$	n
$i=1$	0					
2		0			$\cdots$	
3			0		$\cdots$	
4				0	$\cdots$	
					$\ddots$	
n						0

Programación Dinámica
Multiplicación de matrices

- Esta tabla tiene n^2 entradas por tanto nuestro tiempo debe ser al menos $\Omega(n^2)$
- Sin embargo, en cada paso, el tiempo es $\Theta(n^3)$
- A continuación mostramos el uso de la tabla y finalmente consideramos el tiempo requerido

Programación Dinámica
Multiplicación de matrices

Usando el ejemplo inicial de 4 matrices, completamos la tabla en diagonal que es fácil

Matriz	Dimensiones
A1	20 × 5
A2	5 × 40
A3	40 × 50
A4	50 × 10

	1	2	3	4
1	0 20×5	4000 20×40		
2		0 5×40	10000 5×50	
3			0 40×50	20000 40×10
4				0 50×10

Programación Dinámica
Multiplicación de matrices

Ahora calculamos, colocando el menor en la matriz para la posición $\mathbf{A}_{1..}\mathbf{A}_3$

$\mathbf{A}_1(\mathbf{A}_2\mathbf{A}_3)$ $\quad 20 \times 5 \times 50 + 10000 = 15000$

$(\mathbf{A}_1\mathbf{A}_2)\mathbf{A}_3$ $\quad 4000 + 20 \times 40 \times 50 = 44000$

Matriz	Dimensiones
A1	20 × 5
A2	5 × 40
A3	40 × 50
A4	50 × 10

	1	2	3	4
1	0 20×5	4000 20×40	15000 20×50	
2		0 5×40	10000 5×50	
3			0 40×50	24000 40×10
4				0 50×10

Programación Dinámica
Multiplicación de matrices

Continuamos,

$A_2(A_3A_4)$ $\quad 5 \times 40 \times 10 + 20000 = 22000$

$(A_2A_3)A_4$ $\quad 10000 + 5 \times 50 \times 10 = 12500$

Matriz	Dimensiones
A1	20 × 5
A2	5 × 40
A3	40 × 50
A4	50 × 10

	1	2	3	4
1	0 20×5	4000 20×40	15000 20×50	
2		0 5×40	10000 5×50	12500 5×10
3			0 40×50	20000 40×10
4				0 50×10

Programación Dinámica
Multiplicación de matrices

Finalmente,

$A_1((A_2A_3)A_4)$ $\quad 20 \times 5 \times 10 + 12500 \quad = 13500$

$(A_1A_2)(A_3A_4)$ $\quad 4000 + 20 \times 40 \times 10 + 20000 = 32000$

$(A_1(A_2A_3))A_4$ $\quad 15000 + 20 \times 50 \times 10 = 25000$

Matriz	Dimensiones
A1	20 × 5
A2	5 × 40
A3	40 × 50
A4	50 × 10

	1	2	3	4
1	0 20×5	4000 20×40	15000 20×50	13500 20×10
2		0 5×40	10000 5×50	12500 5×10
3			0 40×50	20000 40×10
4				0 50×10

Programación Dinámica
Multiplicación de matrices

- Así en la posición [1,4] tenemos el valor de la cantidad de operaciones mínima.
- El tiempo necesario puede verse como:

$$n-1$$

$$(n-2)\,2$$

$$(n-3)\,3$$

Puede entonces verse como la siguiente sumatoria

$$\sum_{i=1}^{n-1}(n-i)i = n\sum_{i=1}^{n-1} i - \sum_{i=1}^{n-1} i^2 = \frac{n^3-n^2}{2} - \frac{n(n-1)(2n-1)}{6} = \frac{n^3-n}{6}$$

Así el tiempo esta en el orden de $\Theta(n^3)$

Programación Dinámica
Multiplicación de matrices

- Aplicamos el principio de optimalidad
  - Si el mejor modo de realizar un producto exige dividir inicialmente las matrices *i* e (*i*+1)-esima, los productos

    $A_1A_2..A_i$ y A_{i+1},A_{i+2},A_n

    deberán ser realizados de forma óptima para que el total sea óptimo.
- Método:
  - Construir la matriz $[\boldsymbol{m}_{i,j}]$, $1 <= i <= j <= n$, donde $m_{i,j}$ da el óptimo para la parte $A_iA_{i+1}..A_j$ del producto total.
  - La solución final vendrá por $m_{1,n}$

Programación Dinámica
Multiplicación de matrices

Construcción de $[m_{ij}]$, $1 <= i <= j <= n$

- Guardar la dimensiones de las $\boldsymbol{A}_i$, $1<=i<=n$, en un vector $\boldsymbol{d}$, de $0..n$ componentes, de forma que $\boldsymbol{A}_i$, tiene dimensiones $\boldsymbol{d}_{i-1} \times \boldsymbol{d}_i$
- La diagonal s de $[m_{i,j}]$ contiene los $m_{i,j}$ tales que $j-i = s$:

$s= 0 \quad m_{i,i} = 0$

para $i = 1, 2, 3, .., n$

$s= 1 \quad m_{i,i+1} = d_{i-1} d_i d_{i+1}$

para $i= 1,2,..,n-1$

$1 < s < n : m_{i,\, i+s} = \min_{\, i <= k <= i+s-1} (m_{i,k} + m_{k+1,i+s} + d_{i-1} d_k d_{i+s})$,

para $i=1,2,..,\ n-2$

Programación Dinámica
Multiplicación de matrices

- El tercer paso representa que para calcular $A_i A_{i+1} .. A_{i+s}$ se intentaran todas las posibilidades $(A_i A_{i+1} .. A_k)(A_{k+1} A_{k+2} .. A_{i+s})$
- De forma más concisa

$$m_{ij} = \begin{cases} 0, & \text{si } i = j \\ \min\limits_{i \le k < j} \{ m_{ik} + m_{k+1,j} + d_{i-1} d_k d_j \}, & \text{si } i < j \end{cases}$$

Programación Dinámica
Multiplicación de matrices

Ejemplo 2:

Matriz	Dimensiones
A1	13 × 5
A2	5 × 89
A3	89 × 3
A4	3 × 34

- Se tiene d=(13,5,89,3,34).
- Para s=1: m_{12}=5785, m_{23}=1335, m_{34}=9078.
- Para s=2:

$$m_{13} = \min(m_{11}+m_{23}+13\times5\times3, m_{12}+m_{33}+13\times89\times3)$$
$$= \min(1530, 9256) = 1530$$
$$m_{24} = \min(m_{22}+m_{34}+5\times89\times34, m_{23}+m_{44}+5\times3\times34)$$
$$= \min(24208, 1845) = 1845$$

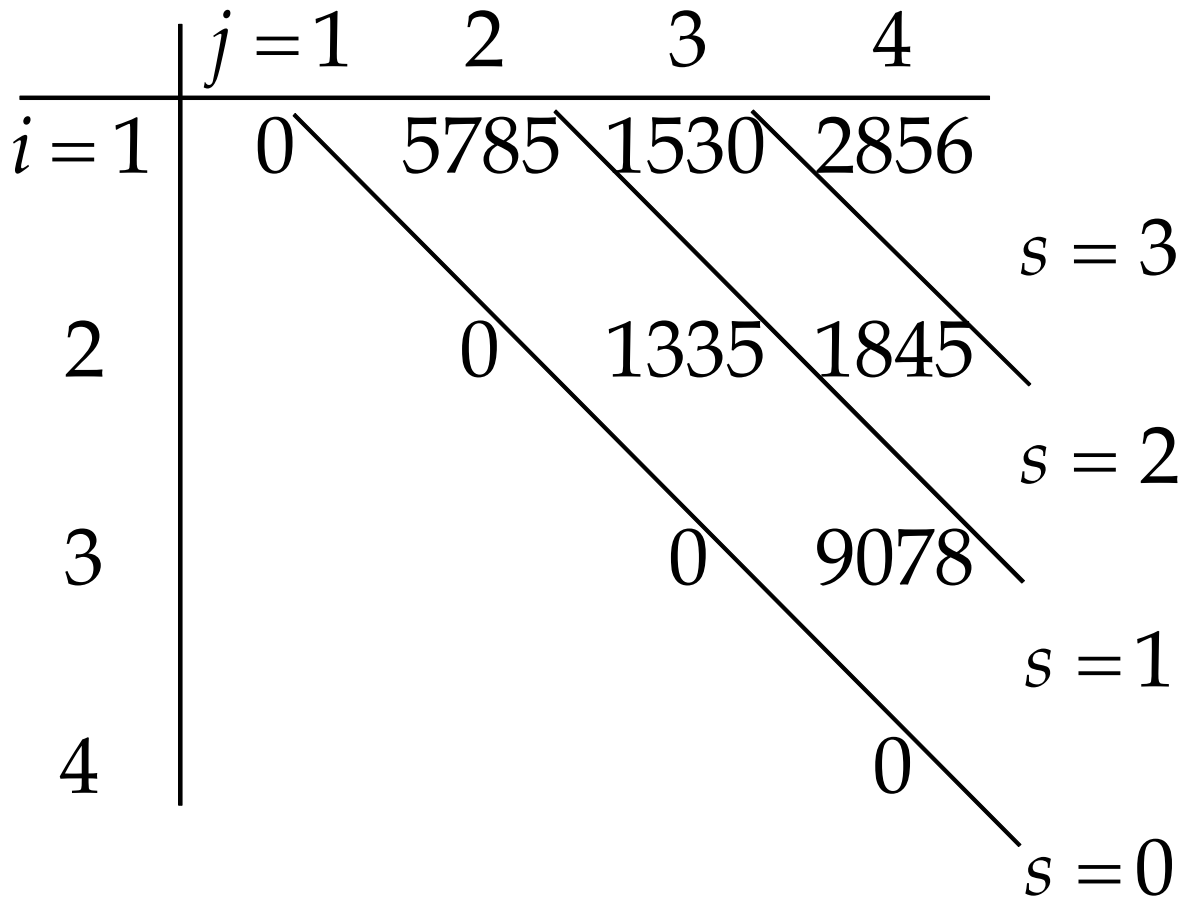
- Para s=3:

$$m_{14} = \min(\{k=1\}\ m_{11}+m_{24}+13\times5\times34,$$
$$\{k=2\}\ m_{12}+m_{34}+13\times89\times34,$$
$$\{k=3\}\ m_{13}+m_{44}+13\times3\times34)$$
$$= \min(4055, 54201, 2856) = 2856$$

m	1	2	3	4
1	0	**5785** 13x5x89	**1530** 13x5x3	**2856** 13x5x89
2		0	**1335** 5x89x3	**1845** 5x3x34
3			0	**9078** 89x3x34
4				0

Programación Dinámica
Multiplicación de matrices

- La matriz es:



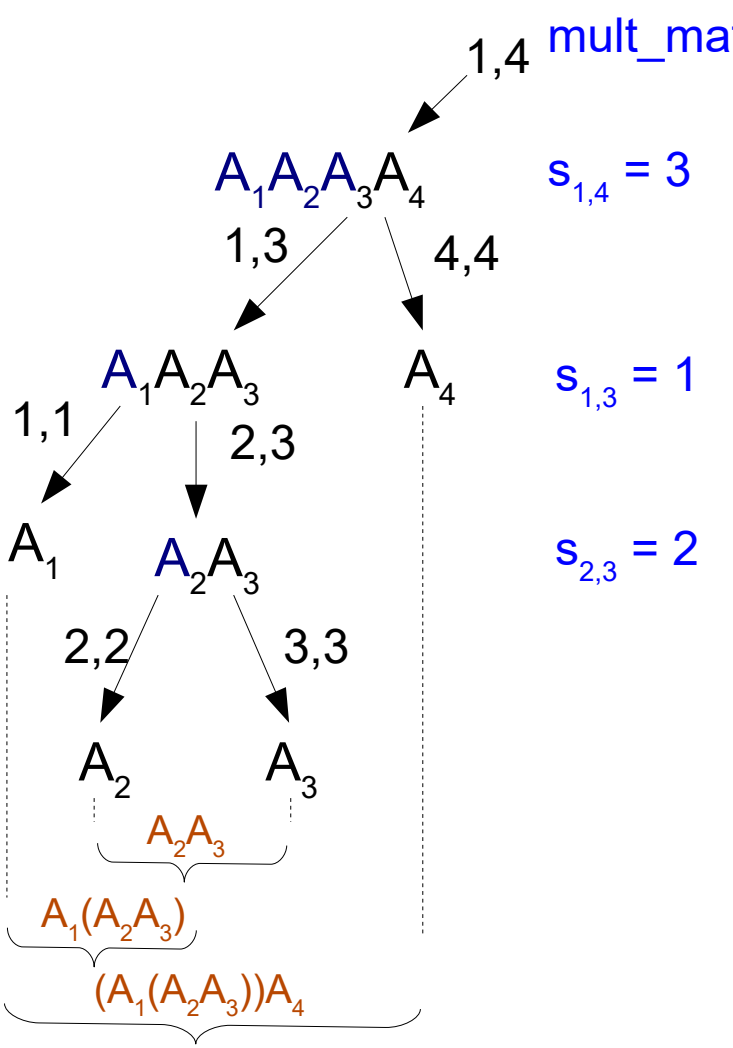
Programación Dinámica
Multiplicación de matrices

- Finalmente debemos calcular el producto
- Al encontrar la k que dividió en forma óptima el producto A_k y A_{k+1} registramos este valor en una matriz s_{ij}, $i <= i <= j <= n$. La posición $s_{i,j}$ registra entonces el valor de k óptimo.
- Para el ejemplo anterior : **(A1(A2A3))A4**

s	1	2	3	4
1	0	1	1	3
2		0	2	3
3			0	3
4				0

Programación Dinámica
Multiplicación de matrices

Empezando por la última posición $s_{1,4}$, vemos que el número es 3, indica que se debe realizar la multiplicación (A1..A3) x A4



s	1	2	3	4
1	0	1	1	3
2		0	2	3
3			0	3
4				0

(A1(A2A3))A4

```
Matriz mult_matriz( Matriz s, int i, int j: int)
Begin
  Matriz M, X, Y
  if  ( i == j ) {
     M = matrices[i]; // lista de matrices
  else
     X = mult_matriz(s,i,s[i,j]);
     Y = mult_matriz(s,s[i,j]+1,j);
     M = mult(X,Y); // Multiplica X * Y
  }
  return M;
end
```

Programación Dinámica
Multiplicación de matrices

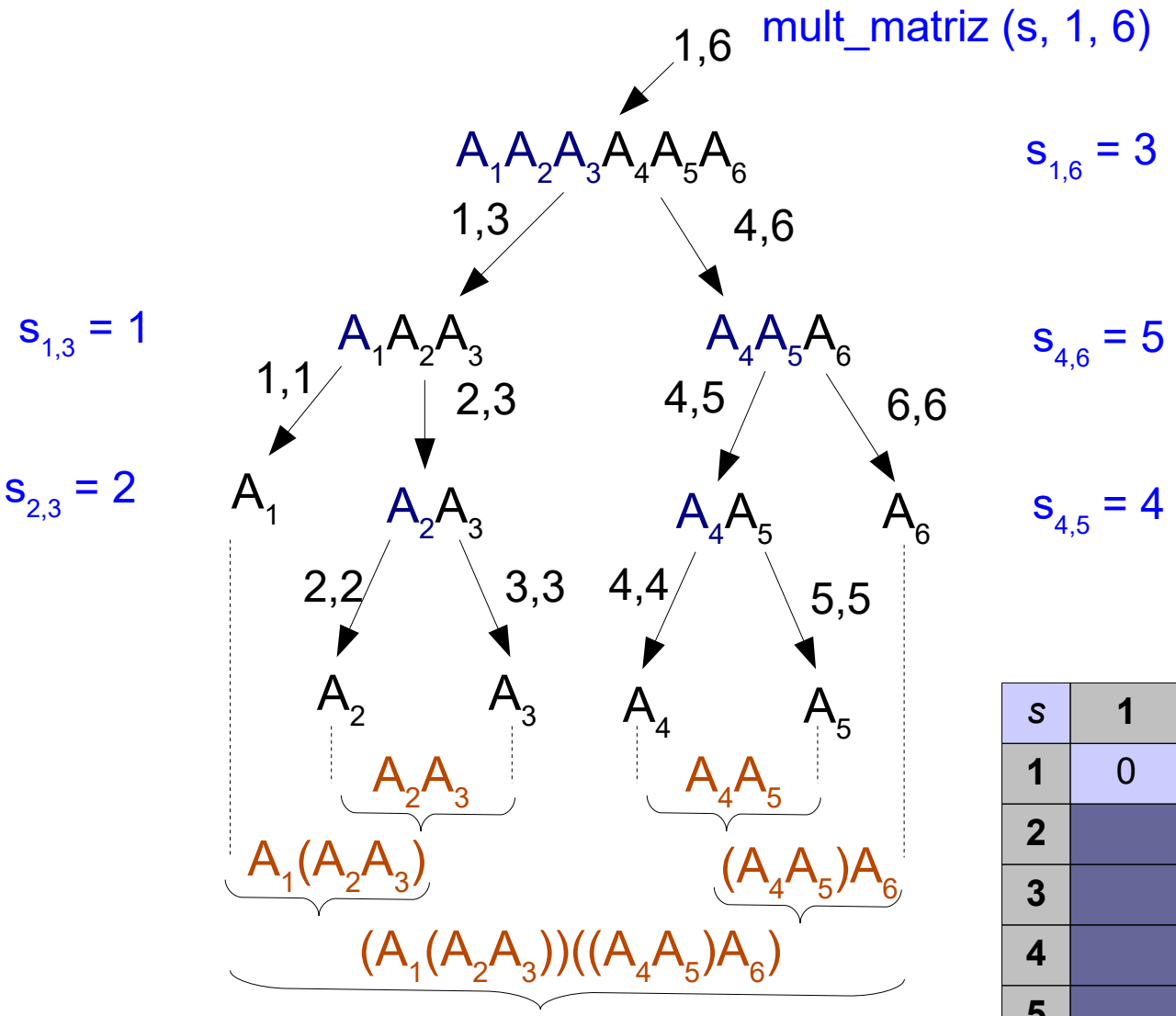
Ejercicio 1: dada la siguiente tabla de k óptimos, dar la secuencia de multiplicación..

Matriz	Dimensiones
A_1	30 × 35
A_2	35 × 15
A_3	15 × 5
A_4	5 × 10
A_5	10 × 20
A_6	20 × 25

s	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

Ejercicio 2: construya la tabla de número óptimo de multiplicaciones.

Programación Dinámica
Multiplicación de matrices



s	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

Programación dinámica

Ejercicio 3 de clase:

Encuentre la parentización óptima para la multiplicación de las siguientes matrices

Matriz	Dimensiones
A_1	5 × 10
A_2	10 × 3
A_3	3 × 12
A_4	12 × 5
A_5	5 × 50
A_6	50 × 6

Programación Dinámica

Ejemplo 2

Problema de la mochila

Se tiene ***n*** objetos fraccionables y una mochila

El objeto *i* tiene peso p_i y una fracción x_i ($0 <= x_i <= 1$) del objeto i produce un beneficio $b_i x_i$

El objetivo es llenar la mochila, de capacidad C, de manera que se maximice el beneficio.

$$\text{maximizar} \quad \sum_{1 \le i \le n} b_i x_i$$

$$\text{sujeto a} \quad \sum_{1 \le i \le n} p_i x_i \le C$$

$$\text{con} \quad 0 \le x_i \le 1,\ b_i > 0,\ p_i > 0,\ 1 \le i \le n$$

Programación Dinámica
Problema de la mochila

Una variante es la mochila “0-1”

x_i solo toma los valores 0 ó 1 , indicando que el objeto se deja fuera o se introduce a la mochila.

Los pesos p_i, y la capacidad son números naturales.

Los beneficios, b_i, son números reales positivos.

Ejemplo: $n=3 \quad C=15$

$(b_1,b_2,b_3)=(38,40,24)$

$(p_1,p_2,p_3)=(9,6,5)$

Tomar siempre el objeto que proporcione mayor beneficio por unidad de peso

aplicando una estrategia voraz se obtiene:

$(x_1,x_2,x_3)=(0,1,1)$, con beneficio 64

sin embargo, la solución óptima es:

$(x_1,x_2,x_3)=(1,1,0)$, con beneficio 78

Programación Dinámica
Problema de la mochila

Sea *mochila(k,l,P)* el problema:

$$\text{maximizar} \quad \sum_{i=k}^{l} b_i x_i$$

$$\text{sujeto a} \quad \sum_{i=k}^{l} p_i x_i \leq P$$

$$\text{con} \quad x_i \in \{0,1\}, \quad k \leq i \leq l$$

El problema de la mochila 0-1 es *mochila(1,n,C)*

Programación Dinámica
Problema de la mochila

Ecuación de recurrencia

Si $\mathring{g}_j(c)$ es el beneficio (o ganancia total) de una solución óptima de *mochila*(1,*j*,*c*), entonces

$$\mathring{g}_j(c) = max\{\mathring{g}_{j-1}(c), \mathring{g}_{j-1}(c-p_j)+b_j\}$$

Comparo entre el beneficio sin el elemento j

Y el beneficio con el elemento j

dependiendo de que el objeto *j*-ésimo entre o no en la solución (nótese que sólo puede entrar si $c\text{-}p_j \geq 0$).

Además $\mathring{g}_0(c)=0$, para cualquier capacidad *c*

Programación Dinámica
Problema de la mochila

- **Problema**: explosión exponencial.

$$T(n) = 2T(n\text{-}1) + O(1) = \Theta(2^n)$$

- Sin embargo, el número total de subproblemas a resolver no es grande:

  La función $\mathring{g}_j(c)$ tiene dos parámetros:

  El primero puede tomar n valores distintos y

  El segundo, C valores.

  ¡Luego sólo hay nC problemas diferentes!

- Por tanto la solución recursiva esta generando y resolviendo el mismo problema muchas veces.

Programación Dinámica
Problema de la mochila

- Para evitar repeticiones de cálculos, las soluciones a los subproblemas se deben almacenar en una tabla.
- Así la matriz $\mathrm{n}\times\mathrm{C}$ cuyo elemento (j,c) almacena $\mathring{g}_j(c)$

  Para el ejemplo anterior: n=3 C=15
  (b_1,b_2,b_3)=(38,40,24)
  (p_1,p_2,p_3)=(9,6,5)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$p_1=9$	0	0	0	0	0	0	0	0	0	38	38	38	38	38	38	38
$p_2=6$	0	0	0	0	0	0	40	40	40	40	40	40	40	40	40	78
$p_3=5$	0	0	0	0	0	24	40	40	40	40	40	64	64	64	64	78

$$\mathring{g}_j(c)=max\{\mathring{g}_{j-1}(c),\mathring{g}_{j-1}(c-p_j)+b_j\}$$

Solución (p_1, p_2) con beneficio 78

Dado C=10 y 5 ítems de valores 9,2,7,4,1

	0	1	2	3	4	5	6	7	8	9	10
$k_1=9$	O	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	I	$-$
$k_2=2$	O	$-$	I	$-$	$-$	$-$	$-$	$-$	$-$	O	$-$
$k_3=7$	O	$-$	O	$-$	$-$	$-$	$-$	I	$-$	I/O	$-$
$k_4=4$	O	$-$	O	$-$	I	$-$	I	O	$-$	O	$-$
$k_5=1$	O	I	O	I	O	I	O	I/O	I	O	I

- : No hay solución para $\mathring{g}_j(c)$

O : Solución con *j* omitido
I : Solución con *j* incluido
I/O : Solución con *j* incluido y omitido (más de una solución)

Ejercicio 4

Considere la siguiente tabla:

item	peso	valor
1	2	12
2	1	10
3	3	20
4	2	15

Teniendo una capacidad de C=5, encuentre la forma de llenar de forma óptima la “mochila”. Compare el resultado con una estrategia “greedy”.

¿Cómo puede saber que ítems serán incluidos?

Programación Dinámica
Problema de la mochila

Otros problemas resueltos con PD

- Subcadena común más larga (usado en biocomputación)

  Forma de encontrar similitud entre dos patrones. En biología una forma de comparar que tan similares son dos organismos por su DNA. Por ejemplo (A=adenina, G=guanina, C=citosina, T=tiamina)

  ```
  S1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
  S2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA
  ```

  La cadena común más larga es (en el mismo orden pero no necesariamente consecutiva).

  ```
  S3 = GTCGTCGGAAGCCGGCCGAA
  ```

- Programación de linea de ensamblado
- Búsqueda óptima en un árbol binario
- Distancia entre palabras (distancia de edición)
- etc

Ejercicio 5

Coeficientes binomiales:

El coeficiente binomial o número combinatorio es el número en los que se puede escoger *k* objetos de un total de *n*. Su fórmula es bien conocida

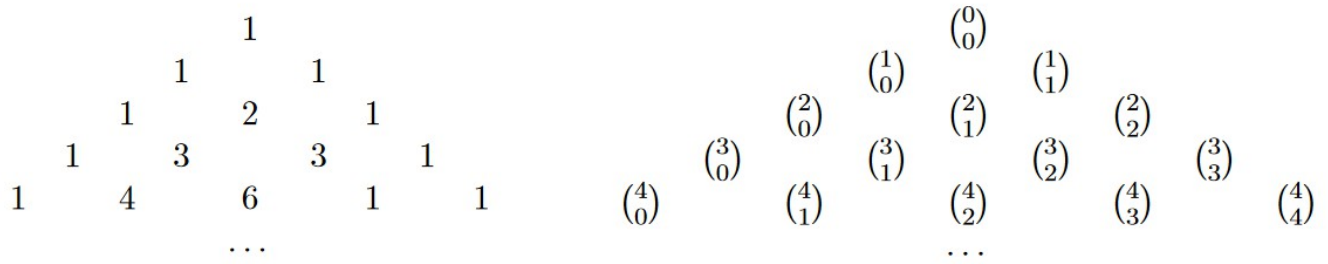
$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Donde n! = n*(n-1)*(n-2)..3*2*1 pero no es práctica desde el punto de vista computacional. ¿porqué?

Otra forma es utilizando el “Triángulo de Pascal” utilizando la siguiente propiedad:

$$\binom{n}{k} = 1 \qquad \text{si } k \text{ és } 0 \text{ o } n,$$

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \qquad \text{si } 0 < k < n.$$



a) Indique el tiempo necesario para una solución recursiva.

b) Plantee una solución utilizando la técnica de la programación dinámica e indique el tiempo que requiere.

Técnicas de Diseño de Alg.
Backtracking

Técnicas de Diseño de Alg.
Backtracking

- A veces necesitamos encontrar una solución óptima a un subproblema pero no podemos aplicar nada más que búsqueda exhaustiva
- La técnica de *backtracking* o *vuelta atrás* es una forma de búsqueda exhaustiva sistemática. Una forma de reducir el espacio de búsqueda se denomina, poda *alfa-beta.*
- Utilizada usualmente en Inteligencia Artificial y juegos.
- En su forma básica se construye un árbol de posibilidades donde las hojas corresponden a una solución posible.

Backtracking

Un posible esquema es el siguiente:

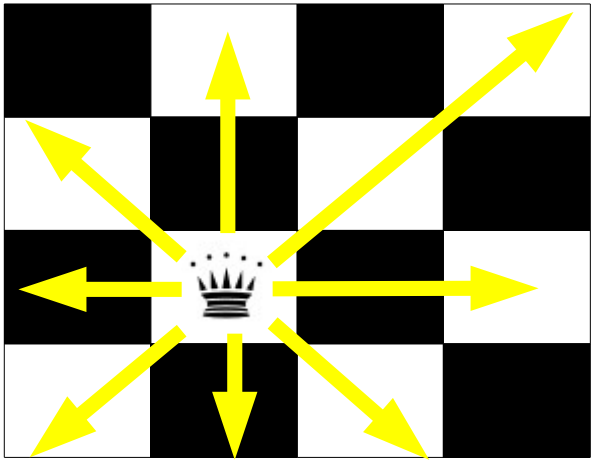
```
PROCEDURE VueltaAtras(etapa);
BEGIN
    IniciarOpciones;
    REPEAT
        SeleccionarNuevaOpcion;
        IF Aceptable THEN
            AnotarOpcion;
            IF SolucionIncompeta THEN
                VueltaAtras(etapa_siguiente);
                IF NOT exito THEN
                    CancelarAnotacion
                END
            ELSE (* solucion completa *)
                exito:=TRUE
            END
        END
    UNTIL (exito) OR (UltimaOpcion)
END VueltaAtras;
```

Backtracking

Ejemplo de las N-reinas

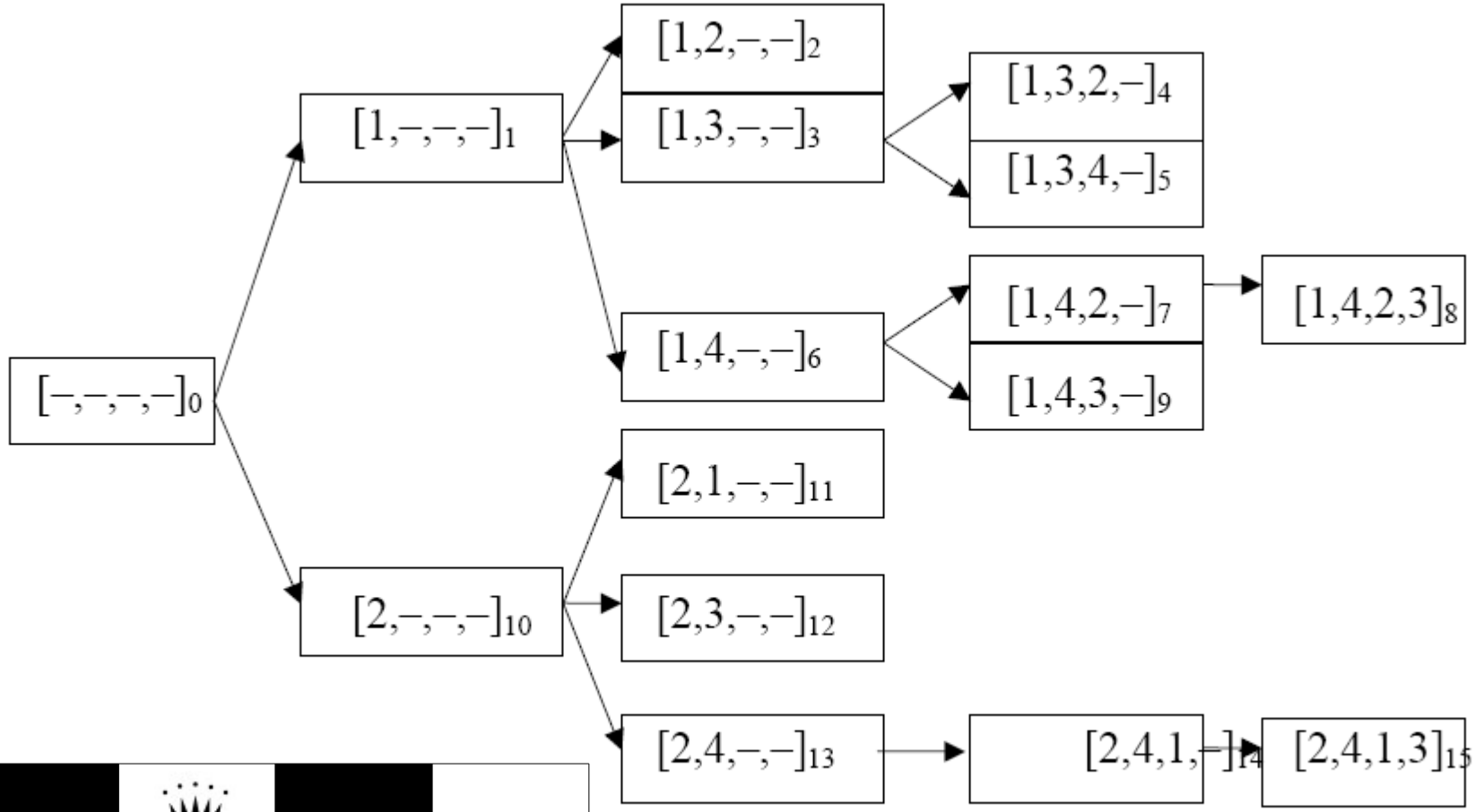
Problema: colocar en un tablero de JxJ, N reinas (de ajedrez) de forma que ninguna se amenace.

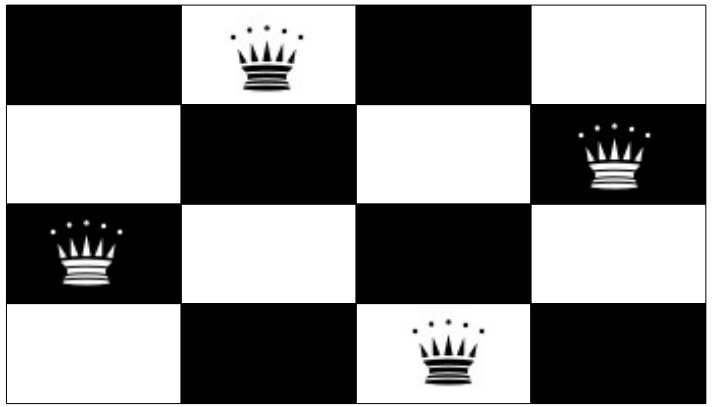
El ejemplo que veremos a continuación es encontrar todas las posibles posiciones válidas de 4 reinas en un tablero de 4x4.



(el problema general se deja como ejercicio de casa)

Backtracking



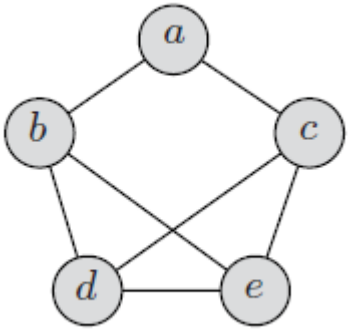


Una posible solución

Ejercicio 6

- Dado un grafo G=(V,E), se requiere colorear cada vértice en V con uno de los tres colores (1,2 ó 3), tal que dos vértices adyacentes no tengan el mismo color.

  Por ejemplo para el grafo:



  Una posible solución es (a=1,b=2,c=2,d=1,e=3)

- Como encontrar todas las posibles soluciones a este problema. Plantearlo con la estrategia de backtracking.

Bibliografía

- Estructura de Datos en Java, Mark Allen Weiss.
- Estructura de datos y algoritmos. Aho, HopCroft & Ullman
- A practical Introduction to data structures and algorithm analysys. Java Edition. Clifford A. Shaffer
- Introduction to Algorithm. Cormen, Leiserson, Rivest & Stein. 2nd Edition (2001) y 3rd Edition (2009).
- Técnicas de diseño de algoritmos. Rosa Guerequeta y Antonio Vallecillo. Universidad de Málaga.