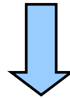
Estructuras de Datos

Tema 2. Diseño de Algoritmos

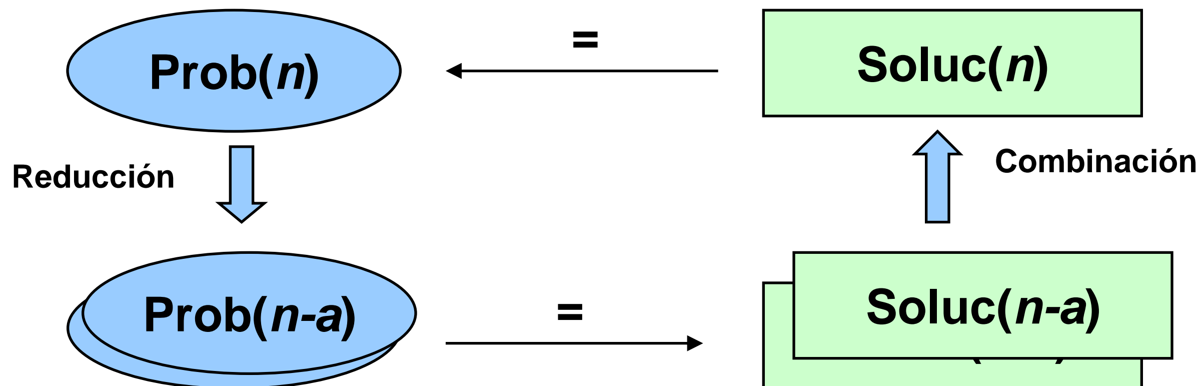
1. Recursividad

- Implica plantear la resolución del problema con otra estrategia:

¿Cómo puedo resolver el problema?

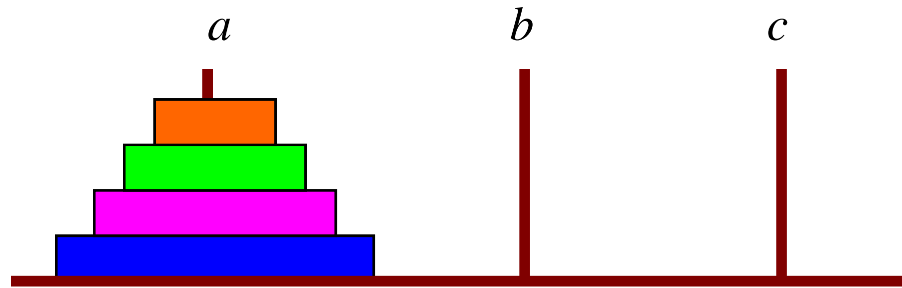


Si me dieran la solución de un problema un poco menos complejo...
¿A partir de esa solución podría obtener la solución del problema original?



- En ese caso, sigue reducciendo el problema hasta que sea trivial

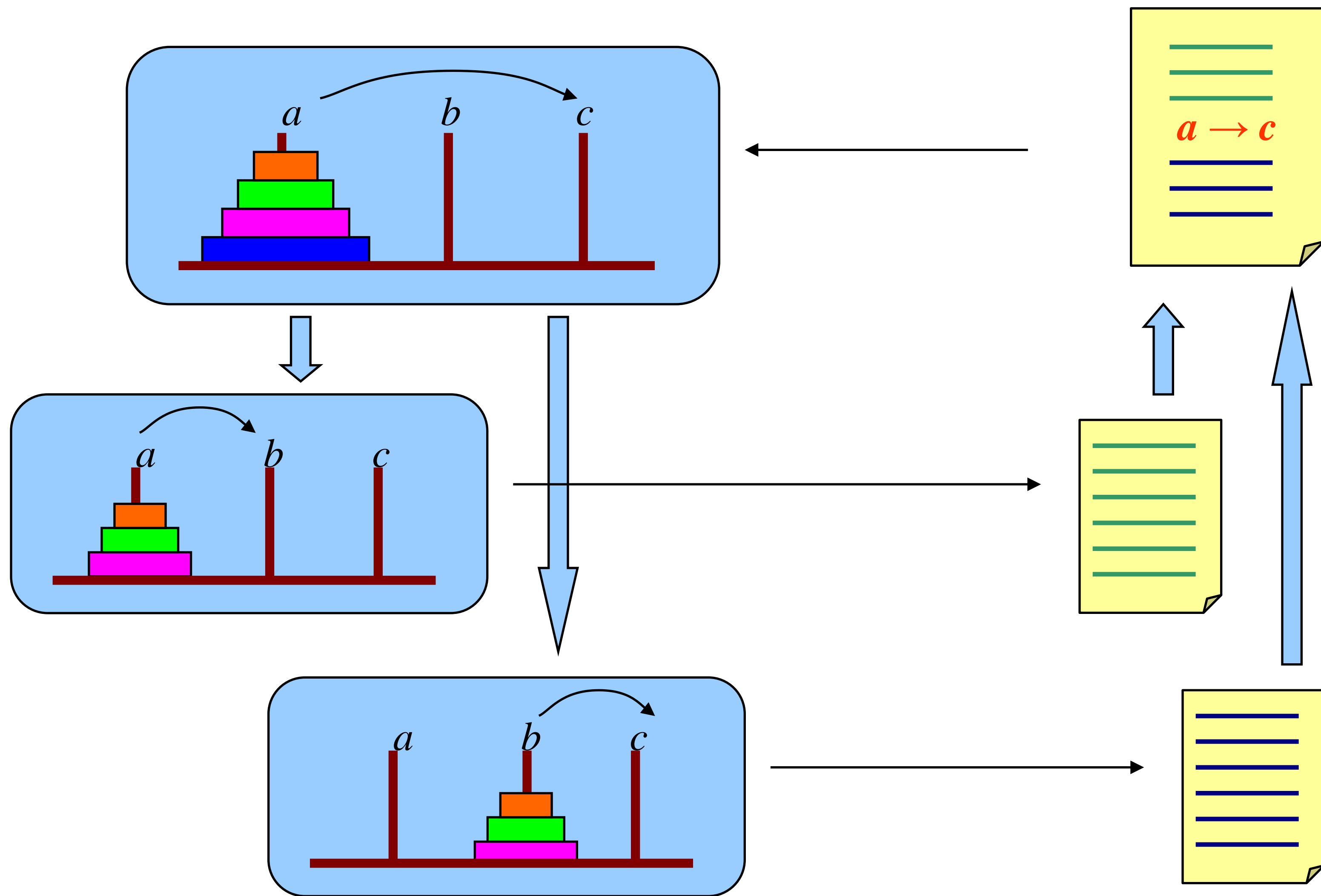
Torres de Hanoi



- Mover pirámide de *n* discos a otro poste
- Solo se mueve un disco cada vez, y tiene que ser el superior
- No se puede apilar un disco mayor sobre otro menor

La solución consiste en una lista de movimientos, cada uno de ellos en el formato poste_origen → poste_destino (mueve el disco superior de poste_origen para que pase a ser el disco superior de poste_destino)

Torres de Hanoi – Enfoque recursivo



Torres de Hanoi - Código

```
procedure Hanoi(n: integer; orig,dest,otro: char);
begin
  if n > 1 then Hanoi(n-1,orig,otro,dest);
  writeln(output,orig,' -> ',dest);
  if n > 1 then Hanoi(n-1,otro,dest,orig)
end;
```

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

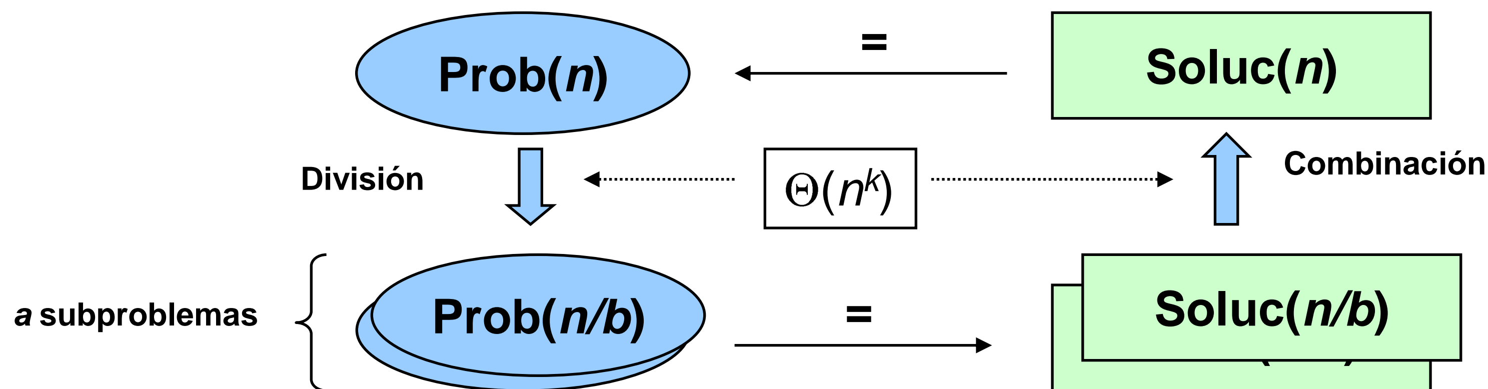
$$T(n) \in \Theta(2^n)$$

$$E(n) = E(n-1) + O(1)$$

$$E(n) \in \Theta(n)$$

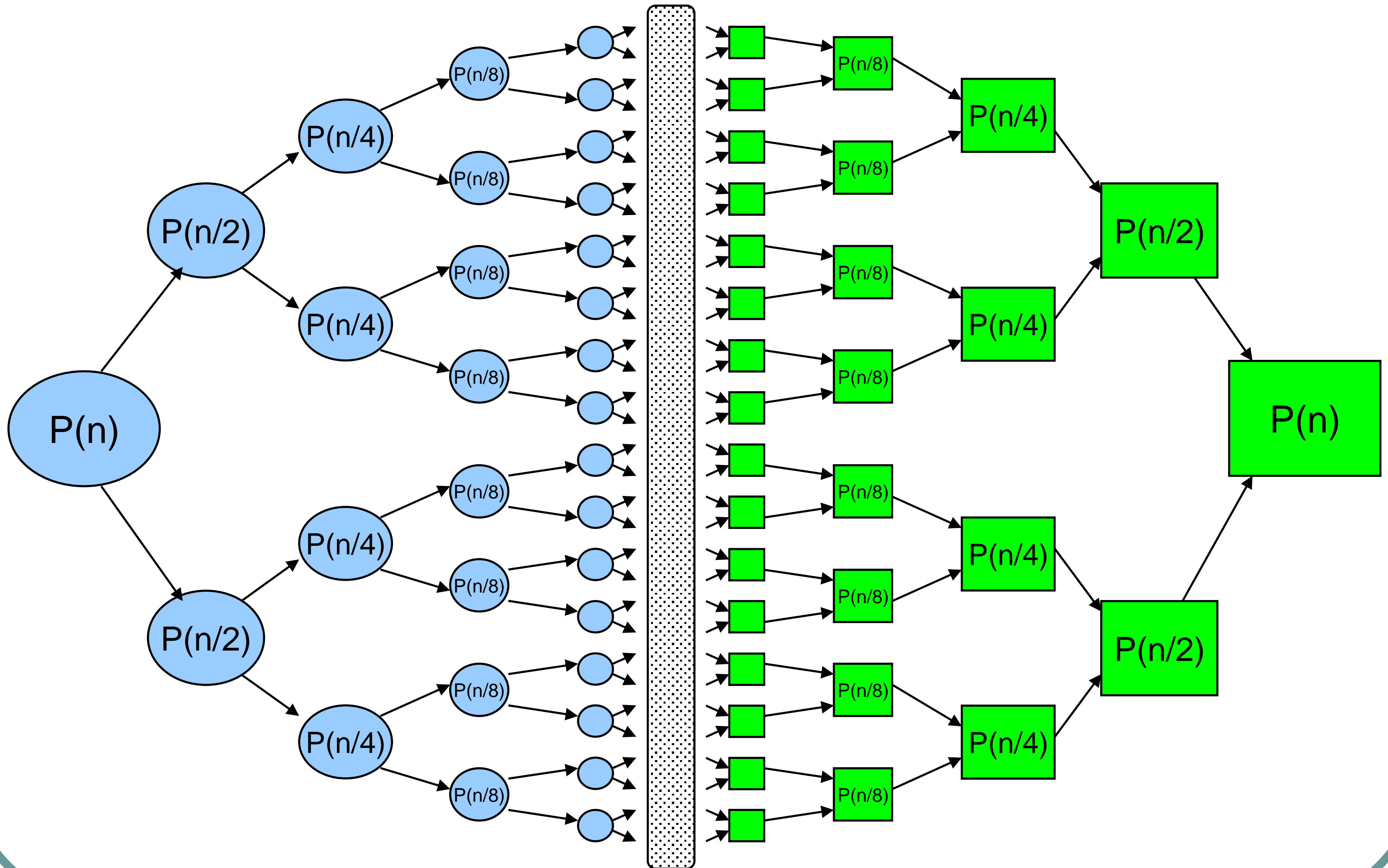
2. Divide y Vencerás

- Dividir el problema en subproblemas de tamaño dividido por una constante
- Resolver los subproblemas mediante *divide y vencerás*
- Combinar las soluciones de los subproblemas para obtener la solución del problema original



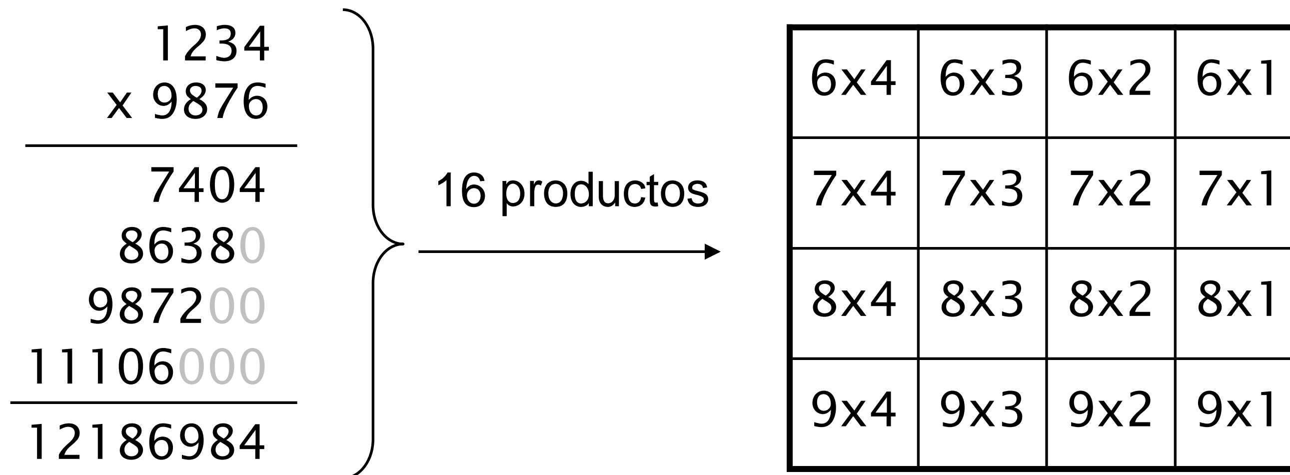
$$\mathbf{T}(n) = a \cdot \mathbf{T}(n/b) + \Theta(n^k)$$

Divide y Vencerás - Esquema



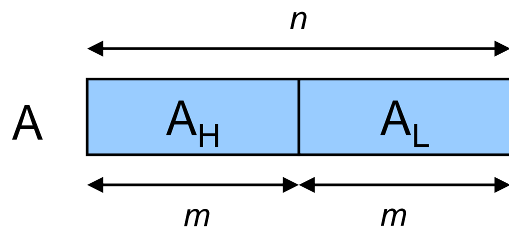
Producto de enteros grandes (PEG)

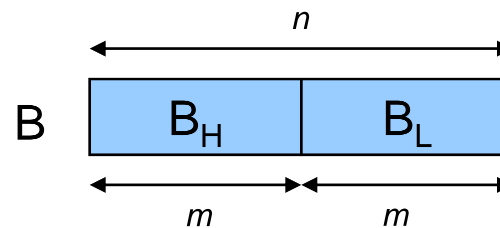
- Enteros representados como un **array** de n bits
- n hace referencia al tamaño del mayor. El otro se puede suponer que también tiene n bits, rellenando con ceros al principio.
- El algoritmo tradicional realiza $O(n^2)$ productos de bits individuales (unidad básica de medida)
- El número de sumas de bits es del mismo orden que el de productos de bits. El número resultante tiene como máximo $2n$ bits



PEG – Divide y vencerás

- Queremos hallar el producto de enteros de *n* bits combinando el resultado de varios productos de enteros de *n*/2 bits.
- Supondremos *n* = 2*m* (par). Dividimos los enteros dos mitades de *m* bits, la mitad más significativa (H) y la menos significativa (L).
- Tomando los números como si fueran polinomios podemos ver como combinarles:





$A = 2^m \cdot A_H + A_L$ $\qquad$ $B = 2^m \cdot B_H + B_L$

$$A \times B = (2^m \cdot A_H + A_L) \times (2^m \cdot B_H + B_L) =$$

$$2^{2m} \cdot (A_H \times B_H) + 2^m \cdot (A_H \times B_L) + 2^m \cdot (A_L \times B_H) + A_L \times B_L$$

PEG – Divide y vencerás directo

- Necesitamos 4 productos de enteros de tamaño $n/2$ para poder reconstruir el resultado.
- Por ejemplo (en base 10), para calcular (1234x9876) necesitamos saber (12x98), (12x76), (34x98) y (34x76) y reconstruimos el resultado así:
- $(1234 \times 9876) = 10^4 \cdot (12 \times 98) + 10^2 \cdot (12 \times 76) + 10^2 \cdot (34 \times 98) + (34 \times 76)$
- **Nota**: Por supuesto cada subproducto se calcula usando divide y vencerás: 12x98 se calcula dividiendo y calculando (1x9), (1x8), (2x9) y (2x8) y combinándolos mediante la fórmula
- Los productos por potencias de dos no cuestan nada, son equivalentes a un desplazamiento de dígitos y llevando adecuadamente los índices no son necesarias operaciones extra.
- Sumar dos números de n bits tiene un coste $O(n)$.

$$\mathbf{T}(n) = 4 \cdot \mathbf{T}(n/2) + \Theta(n)$$

$$\mathbf{T}(n) \in \Theta(n^2)$$

PEG – Algoritmo de Karutsaba

- Se parte del siguiente resultado..

$$(A_H+A_L) \times (B_H+B_L) = A_H\times B_H + A_H\times B_L + A_L\times B_H + A_L\times B_L$$

- ..para modificar la fórmula de combinación:

$$A\times B = 2^{2m}\cdot(A_H\times B_H) + 2^m\cdot(A_H\times B_L) + 2^m\cdot(A_L\times B_H) + A_L\times B_L =$$

$$2^{2m}\cdot(A_H\times B_H) + 2^m\cdot((A_H+A_L)\times(B_H+B_L) - A_H\times B_H - A_L\times B_L) + A_L\times B_L$$

- Se obtiene una fórmula más compleja pero donde sólo se necesitan 3 productos, $(A_H\times B_H)$ y $(A_L\times B_L)$ se usan dos veces pero sólo es necesario calcularlos una vez!

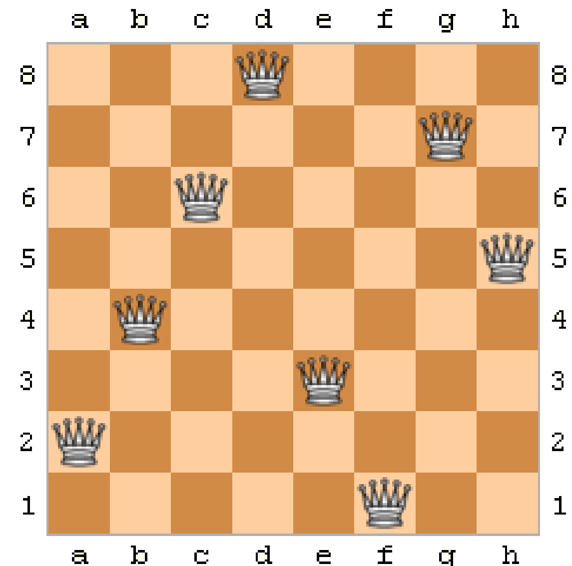
$\mathbf{T}(n) = 3\cdot\mathbf{T}(n/2) + \Theta(n)$ $\mathbf{T}(n) \in \Theta(n^{\lg 3}) = \Theta(n^{1.58})$

3. Fuerza Bruta

- Problemas **difíciles**, sin un método directo de resolución.
- Dada una posible solución, conocemos un algoritmo para saber si es **válida** o no (**criterio de verificación**).
- Tipos de problemas:
  - Una única solución válida.
  - Varias soluciones válidas: Enumerar todas.
  - Varias soluciones válidas: Obtener una cualquiera.
  - **Optimización**: Obtener la mejor de entre las soluciones válidas. Cada solución se evalúa con una **función de coste**.
- **Estrategia Fuerza Bruta**: Explorar el *espacio de posibles resultados* aplicando a cada posible solución el criterio de verificación.
- El encontrar un modo de generar todas las posibles soluciones puede no ser trivial.
- Es muy importante encontrar una **representación** de los resultados que elimine la mayor cantidad posible de soluciones no válidas

Problema de las N-Reinas

- Problema histórico, propuesto en 1848 por Max Bezzel y examinado, entre otros, por Gauss y Cantor.
- Dado un tablero de ajedrez de tamaño *n* x *n*, colocar *n* reinas sin que existan amenazas entre ellas.
- Una reina amenaza a cualquier otra que se encuentre en su misma fila, columna, diagonal izquierda o diagonal derecha.
- Dependiendo del tamaño del tablero pueden existir 0 ó varias soluciones (para el tablero de 8x8 habitual existen 92).
- Aunque no sepamos cómo resolver el problema, dada una posible solución es sencillo comprobar si lo es realmente (criterio de verificación)

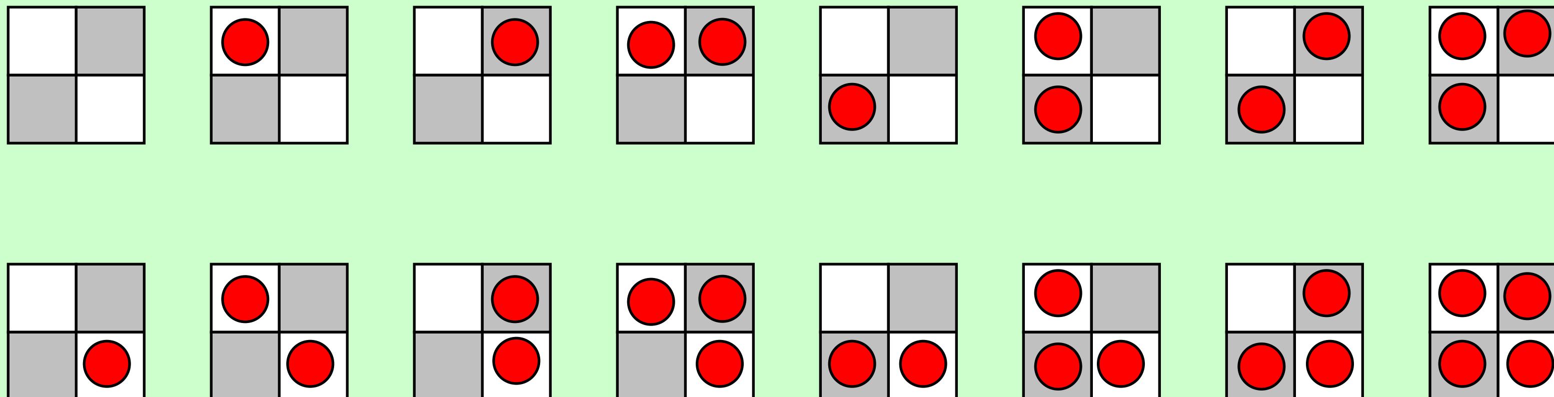


N-Reinas – Representación 1

```
type
  TTablero = array[1..N,1..N] of boolean;
```

Fila (first 1..N) — Columna (second 1..N)

2^{n^2} tableros

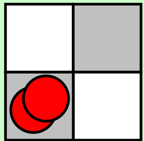
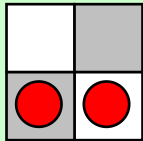
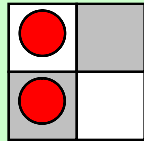
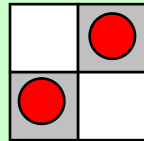
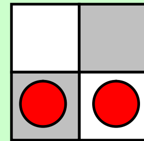
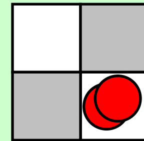
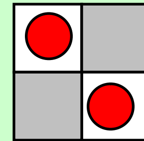
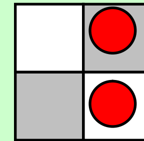
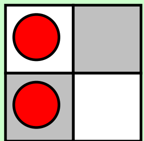
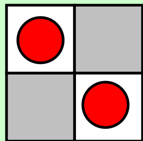
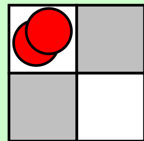
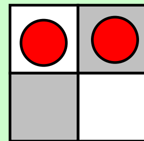
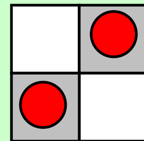
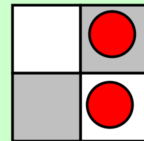
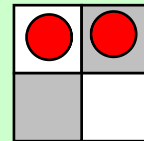
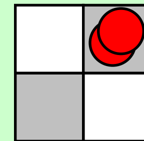


N-Reinas – Representación 2

```
type
  TTablero = array[1..N] of record
                              Fil,Col: 1..N;
                           end;
```

Indice reina (labels [1..N])

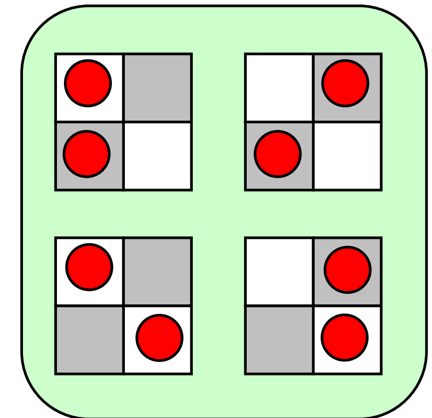
$$(n^2)^n$$

N-Reinas – Representación 3

```
type
  TTablero = array[1..N] of 1..N;
```

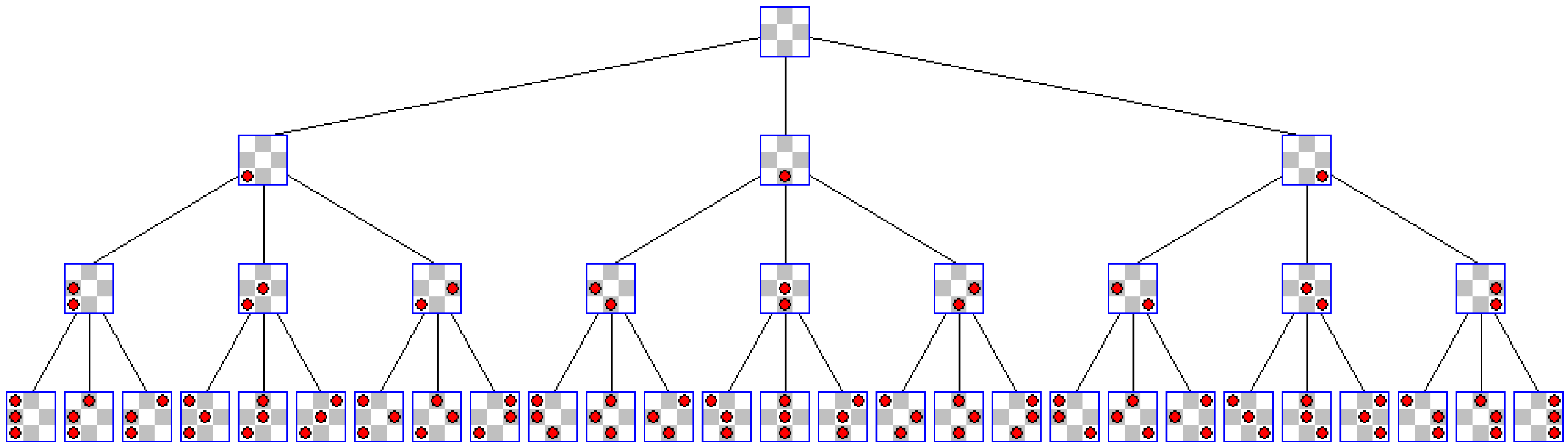
Indice reina Fila — Columna — n^n



N	2^{n^2}	$(n^2)^n$	n^n
2	16	16	4
3	512	729	27
4	65.536	65.536	256
5	33.554.432	9.765.625	3.125
6	68.719.476.736	2.176.782.336	46.656
7	562.949.953.421.312	678.223.072.849	823.543
8	18.446.744.073.709.600.000	281.474.976.710.656	16.777.216

Arbol de Soluciones

- En muchos problemas, los resultados se representan mediante un array o una estructura similar.
- En estos casos la generación de todos los posibles resultados se puede hacer recursivamente haciendo uso de resultados parciales:
  - En cada llamada recursiva, una zona de la solución ya está asignada, y el resto sin asignar (libre).
  - En un bucle se van asignando todos los valores posibles a una componente libre y se llama recursivamente para rellenar el resto.
- La representación gráfica de este método es el arbol de soluciones:



N-Reinas – Fuerza Bruta (todos)

```
function Amenazas(const T: TTablero; N: integer) : boolean;
{ Detecta si existen amenazas entre las reinas del tablero }

procedure NReinas(var T: TTablero; Fil,N: integer);
{ Genera y comprueba todos los posibles tableros
  obtenidos colocando reinas en las filas Fil..N }
var Col : integer;
begin
  if Fil > N then { tablero completo }
  begin
    if not Amenazas(T,N) then Escribir(T,N);
  end else begin { tablero parcial }
    for Col := 1 to N do
    begin
      T[Fil] := Col; { colocar reina }
      NReinas(T,Fil+1,N); { generar tableros }
    end
  end
end;
```

N-Reinas – Fuerza Bruta (1 solución)

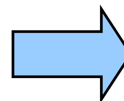
```
function NReinas(var T: TTablero; Fil,N: integer): boolean;
{ Busca un tablero solución entre aquellos obtenidos
  colocando reinas en las filas Fil..N. Devuelve true
  si lo encuentra. }
var Col : integer;
begin
  if Fil > N then { tablero completo }
  begin
    Result := not Amenazas(T,N);
  end else begin { tablero parcial }
    Col := 1;
    repeat
      T[Fil] := Col; { colocar reina }
      Result := NReinas(T,Fil+1,N); { buscar solución }
      Col := Col+1
    until Result or (Col > N);
  end
end;
```

Sudoku

- En este caso la representación es evidente, una matriz de 9x9 números del 1 al 9. Se añadirá el valor 0 para distinguir entre una celda asignada inicialmente y una celda libre.
- Aunque tenemos una matriz, es sencillo tratarla como un vector de 81 valores asignando a cada celda el índice (Fila-1)*9 + Columna.

```
type TSudoku = array[1..9,1..9] of 0..9;
```

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9



5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Sudoku – Fuerza Bruta

```
procedure Sudoku(var S: TSudoku; I: integer);
{ Genera todos los posibles sudokus rellenando las
  celdas no asignadas posteriores a la i-ésima }
var V,Fil,Col : integer;
begin
  if I > 81 then { sudoku completo }
  begin
    if SudokuCorrecto(S) then Escribir(S);
  end else begin
    Fil := (I div 9)+1; Col := (I mod 9)+1;
    if S[Fil,Col] <> 0 then Sudoku(S,I+1) else
    begin
      for V := 1 to 9 do
      begin
        S[Fil,Col] := V; Sudoku(S,I+1);
      end;
      S[Fil,Col] := 0; { borrar valor }
    end
  end
end;
```

Problema de la herencia

- Se desean repartir una serie de *n* objetos, de valores $v_1..v_n$, en dos lotes con el objetivo de que el reparto sea lo más equitativo posible.
- Es un problema de optimización. El criterio de validez es sencillo, cualquier reparto de todos los objetos en el que cada uno se asigne a uno y solo uno de los lotes es válido. Lo que se desea es averiguar cuál, de todos los repartos posibles, hace mínima la diferencia de valor entre los lotes.
- Si la solución se expresa como el vector $r_1..r_n$ donde r_i vale 0 si el objeto *i* se asigna al primer lote y 1 si se asigna al segundo lote, la función de coste sería:

$$\min \left\{ \left| \underbrace{\sum_{i=1}^{n} r_i \cdot v_i}_{\text{valor lote 2}} - \underbrace{\sum_{i=1}^{n} (1 - r_i) \cdot v_i}_{\text{valor lote 1}} \right| \right\}$$

Herencia – Fuerza Bruta

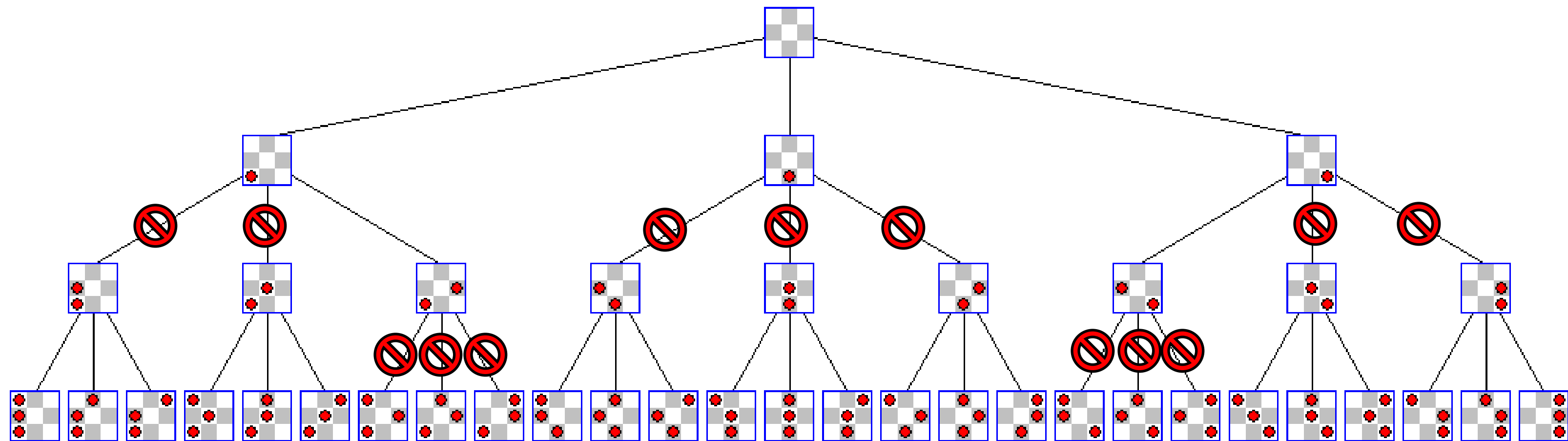
```
type
  TReparto = array[1..N] of boolean;

procedure Herencia(var R,Ropt: TReparto; I,N: integer);
{ Genera todos los posibles repartos rellenando las celdas
  no asignadas posteriores a la i-ésima }
begin
  if I > N then { reparto completo }
  begin
    { comprueba si el reparto actual es mejor
      que el mejor encontrado hasta el momento }
    if coste(R) < coste(Ropt) then Ropt := R
  end else begin { reparto parcial }
    { sólo existen dos posibles valores }
    R[I] := false; Herencia(R,Ropt,I+1,N);
    R[I] := true;  Herencia(R,Ropt,I+1,N)
  end
end;
```

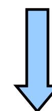
4. Backtracking

- Problemas del mismo tipo que los de Fuerza Bruta (existe criterio de verificación), en los que se puede explorar el espacio de posibles soluciones mediante el árbol de soluciones.
- Se define **criterio de poda** como todo aquél criterio que permita saber, dada una solución parcial, que no va a ser posible que a partir de ella se obtengan soluciones válidas.
- El objetivo es eliminar la exploración de la mayor cantidad posible de zonas del árbol, y por lo tanto encontrar uno o más criterios de poda lo más potentes posibles.
- Cuando el criterio de poda es cierto, a partir de la solución parcial no se pueden generar soluciones válidas.
- Cuando el criterio de poda es falso, no se sabe nada: Es posible tanto que existan como que no existan soluciones válidas a partir de la solución parcial. Se debe seguir explorando.

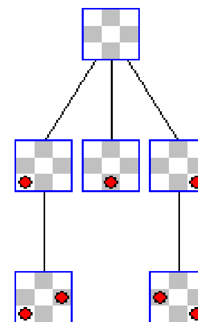
N-Reinas – Backtracking

- Un criterio de poda sencillo es que si ya tienes amenazas entre las reinas colocadas, no vas a poder obtener un tablero válido.





Arbol "podado"



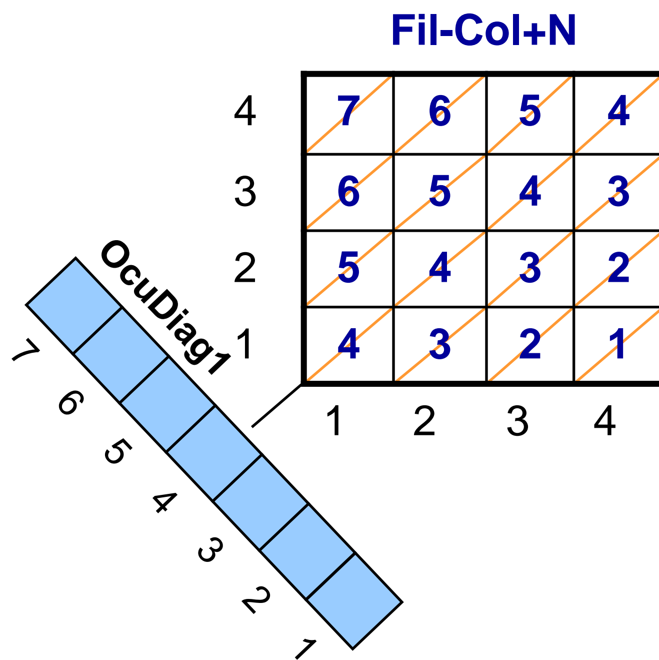
N-Reinas – Tableros comprobados

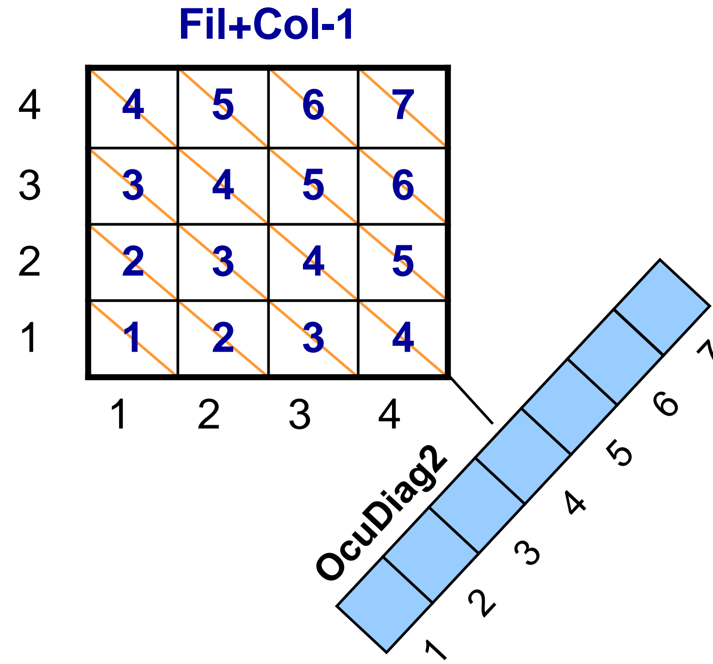
N	Fuerza Bruta	Backtracking
2	4	6
3	27	21
4	256	68
5	3.125	270
6	46.656	900
7	823.543	4.949
8	16.777.216	17.320
9	387.420.489	107.163
10	10.000.000.000	429.770
11	285.311.670.611	3.225.310
12	8.916.100.448.256	14.592.456
13	302.875.106.592.253	125.416.577
14	11.112.006.825.558.000	639.627.618

N-Reinas – Backtracking (I)

```
function Amenazas(const T: TTablero; Fil,N: integer) : boolean;
{ Detecta si existen amenazas entre las reinas 1..Fil }

procedure NReinas(var T: TTablero; Fil,N: integer);
var Col : integer;
begin
  if Fil > N then { tablero completo }
  begin
    if not Amenazas(T,N,N) then Escribir(T,N);
  end else begin { tablero parcial }
    for Col := 1 to N do
    begin
      T[Fil] := Col; { colocar reina }
      if not Amenazas(T,Fil,N) then { criterio de poda }
         NReinas(T,Fil+1,N);        { recursividad }
    end
  end
end;
```

N-Reinas – Backtracking (II)






```
type
  TTablero = record
    Tab       : array of integer; { array[1..N] of 1..N }
    OcuCol    : array of boolean; { array[1..N] of bool }
    OcuDiag1  : array of boolean; { array[1..2*N-1] of bool }
    OcuDiag2  : array of boolean; { array[1..2*N-1] of bool }
  end;
```

N-Reinas – Backtracking (II)

```
procedure NReinas(var T: TTablero; Fil,N: integer);
var Col : integer;
begin
  { caso base: tablero completo y correcto }
  if Fil > N then Escribir(T,N) else
  begin
    for Col := 1 to N do
    begin
      if not T.OcuCol[Col] and
         not T.OcuDiag1[Fil-Col+N] and
         not T.OcuDiag2[Fil+Col-1] then
      begin
        T.Tab[Fil] := Col;
        T.OcuCol[Col] := true;
        T.OcuDiag1[Fil-Col+N] := true;
        T.OcuDiag2[Fil+Col-1] := true;
       NReinas(T,Fil+1,N);
        T.OcuCol[Col] := false;
        T.OcuDiag1[Fil-Col+N] := false;
        T.OcuDiag2[Fil+Col-1] := false;
      end
    end end
end;
```

Criterio de poda (condiciones del if)

Poner reina (asignaciones a true)

Quitar reina (asignaciones a false)

Sudoku – Backtracking

```
procedure Sudoku(var S: TSudoku; I: integer);
var V,Fil,Col : integer;
begin
  if I > 81 then { sudoku completo }
  begin
    if SudokuCorrecto(S) then Escribir(S);
  end else begin
    Fil := (I div 9)+1; Col := (I mod 9)+1;
    if S[Fil,Col] <> 0 then Sudoku(S,I+1) else
    begin
      for V := 1 to 9 do
      begin
        S[Fil,Col] := V;
        if SudokuCorrecto(S) then Sudoku(S,I+1);
      end;
      S[Fil,Col] := 0; { borrar valor }
    end
  end
end;
```

Sudoku – Backtracking (II)

1	2	3
4	5	6
7	8	9

```
type
  TFil = 1..9; { filas }
  TCol = 1..9; { columnas }
  TBlo = 1..9; { bloques }
  TSudoku = record
    Sud    : array[TFil,TCol] of 0..9;
    ValFil : array[TFil] of set of 1..9;
    ValCol : array[TCol] of set of 1..9;
    ValBlo : array[TBlo] of set of 1..9;
  end;
```

```
const
  Trad : array[TFil,TCol] of TBlo =
    ((1,1,1,2,2,2,3,3,3),
     (1,1,1,2,2,2,3,3,3),
     (1,1,1,2,2,2,3,3,3),
     (4,4,4,5,5,5,6,6,6),
     (4,4,4,5,5,5,6,6,6),
     (4,4,4,5,5,5,6,6,6),
     (7,7,7,8,8,8,9,9,9),
     (7,7,7,8,8,8,9,9,9),
     (7,7,7,8,8,8,9,9,9));
```

→ Matriz para obtener el bloque al que pertenece una casilla

Sudoku – Backtracking (II)

```
procedure Sudoku(var S: TSudoku; I: integer);
var V,Fil,Col,Blo : integer;
begin
  if I > 81 then Escribir(S) else
  begin
    Fil := (I div 9)+1; Col := (I mod 9)+1; Blo := Trad[Fil,Col];
    if S.Sud[Fil,Col] <> 0 then Sudoku(S,I+1) else
    begin
      for V := 1 to 9 do
        if not (V in S.ValFil[Fil]) and
           not (V in S.ValCol[Col]) and
           not (V in S.ValBlo[Blo]) then
        begin
          S.Sud[Fil,Col] := V;
          S.ValFil[Fil] := S.ValFil[Fil] + [V];
          S.ValCol[Col] := S.ValCol[Col] + [V];
          S.ValBlo[Blo] := S.ValCol[Blo] + [V];
          Sudoku(S,I+1);
          S.ValFil[Fil] := S.ValFil[Fil] - [V];
          S.ValCol[Col] := S.ValCol[Col] - [V];
          S.ValBlo[Blo] := S.ValCol[Blo] - [V]
        end; { if y for }
      S[Fil,Col] := 0
    end
  end end;
```

- `if I > 81 then Escribir(S) else` → Caso base: completo y correcto
- `if not ... then` → Criterio de poda
- Las tres asignaciones con `+ [V]` → Poner valor
- Las tres asignaciones con `- [V]` → Quitar valor

Herencia – Fuerza Bruta

```
type
  TReparto = record
    Val : array{[1..N]} of integer; { Valores objetos }
    Rep : array{[1..N]} of boolean; { Reparto }
    Coste : integer;  { Diferencia valor lote A y lote B }
  end;

procedure Herencia(var R,Ropt: TReparto; I,N: integer);
var CosteIni : integer;
begin
  if I > N then { reparto completo }
  begin
    if abs(R.Coste) < abs(Ropt.Coste) then Ropt := R
  end else begin { reparto parcial }
    CosteIni := R.Coste;
    R.Rep[I] := false; R.Coste := R.Coste-R.Val[I]; { i ∈ lote B }
    Herencia(R,Ropt,I+1,N);
    R.Coste := CosteIni; { Deshacer cambio }
    R.Rep[I] := true;  R.Coste := R.Coste+R.Val[I]; { i ∈ lote A }
    Herencia(R,Ropt,I+1,N);
    R.Coste := CosteIni; { Deshacer cambio }
  end
end;
```

Herencia - ¿Backtracking?

- Con la representación elegida todas las soluciones son válidas, por lo que no pueden existir criterios de poda.
- Para cada solución existe otra simétrica (todos los objetos del lote A pertenecen al lote B y viceversa) con el mismo coste.
- Esto permite restringir el problema exigiendo, por ejemplo, que el lote A sea menos o igual valioso que el lote B.
- En este problema restringido si que existe un criterio de poda: Si en una solución parcial el lote A es más valioso que el valor del lote B *mas la suma de los valores de los objetos restantes* entonces es imposible obtener una solución válida.
- Sin embargo, este criterio de poda es poco potente: En el mejor caso nos eliminaría las soluciones simétricas y dividiría por dos el espacio de búsqueda, lo que es una ganancia mínima.

5. Programación Dinámica

- Estrategia de resolución de problemas (típicamente problemas de optimización) aplicable cuando éstos cumplen:
  - Subestructura óptima (principio de suboptimalidad)
  - Subproblemas tienen solapamiento
  - Se puede aplicar TRP (*memoization*)
- Subestructura óptima: La solución óptima/válida del problema se puede obtener fácilmente a partir de soluciones óptimas/válidas de subproblemas.
- Solapamiento: En el árbol de soluciones aparecen subproblemas repetidos.
- Aplicación de la técnica de la tabla de resultados parciales (TRP, *memoization*): Los parámetros de los (sub)problemas están acotados, de forma que es factible almacenar las soluciones de los subproblemas.

Tabla de Resultados Parciales (TRP)

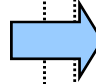
- Aplicable a funciones recursivas con parámetros acotados (a lo largo de la recursión sólo van a tomar un número discreto de posibles valores). Tipicamente son subrangos de enteros.
- Consiste en sustituir llamadas recursivas por accesos a una tabla que almacena los valores (ya calculados) de la función.
- La tabla se rellena desde los casos bases hasta el valor deseado.

```
function F(N: integer) : integer;
begin
  if N = 0 then
    Result := 1
  else
    Result := N*F(N-1)
end;
```

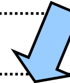
```
function F(N: integer) : integer;
var
  I : integer;
  T : array of integer;
begin
  SetLength(T,N+1);
  T[0] := 1;
  for I := 1 to N do
    T[I] := I*T[I-1];
  Result := T[N]
end;
```

Factorial TRP – Optimización de espacio

```
function F(N: integer) : integer;
var
  I : integer;
  T : array of integer;
begin
  SetLength(T,N+1);
  T[0] := 1;
  for I := 1 to N do
    T[I] := I*T[I-1];
  Result := T[N]
end;
```



```
function F(N: integer) : integer;
var I,Ant,Act : integer;
begin
  Act := 1; { 0! }
  for I := 1 to N do
  begin
    Ant := Act;
    Act := I*Ant;
  end;
  Result := Act
end;
```

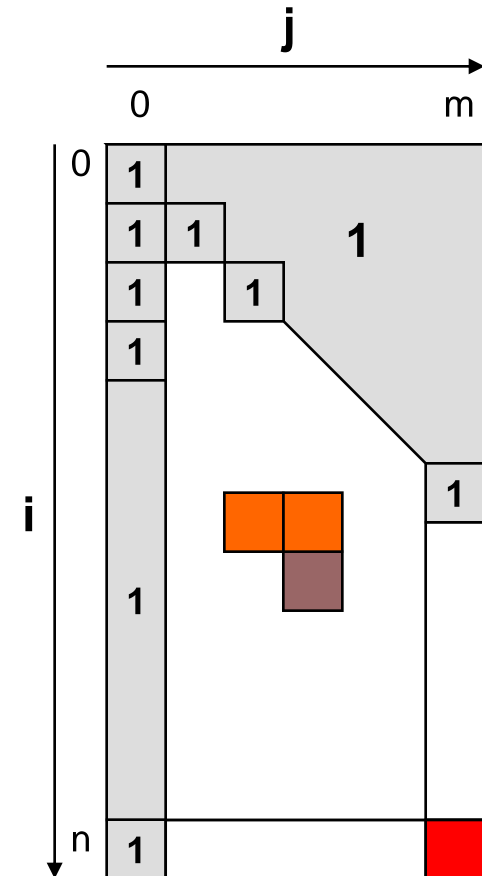


```
function F(N: integer) : integer;
var I : integer;
begin
  Result := 1; { 0! }
  for I := 1 to N do
    Result := I*Result;
end;
```

Cálculo de combinaciones – TRP

```
function C(N,M: integer) : integer;
var
  T : array of array of integer;
  I,J : integer;
begin
  if (M = 0) or (M >= N) then C := 1 else
  begin
    SetLength(T,N+1,M+1);
    T[0,0] := 1;
    for I := 1 to N do
    begin
      T[I,0] := 1;
      for J := 1 to Min(M-1,I-1) do
        T[I,J] := T[I-1,J] + T[I-1,J-1];
      T[I,Min(M,I)] := 1;
    end;
    C := T[N,M]
  end
end;
```

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$



Cálculo de combinaciones – Optimización

```
function C(N,M: integer) : integer;
var
  Lant,Lact : array of integer;
  I,J : integer;
begin
  if (M = 0) or (M >= N) then C := 1 else
  begin
    SetLength(Lant,M+1);
    SetLength(Lact,M+1);
    Lact[0] := 1;
    for I := 1 to N do
    begin
      for J := 0 to M do Lant[J] := Lact[J]; { Lant := Lact }
      Lact[0] := 1;
      for J := 1 to Min(M-1,I-1) do
        Lact[J] := Lant[J]+Lant[J-1];
      Lact[Min(M,I)] := 1;
    end;
    C := Lact[M]
  end
end;
```

Problema del Cambio en Monedas

- Supondremos un país con un sistema monetario que consiste en **n** monedas o billetes de valores faciales $m_1..m_n$ (ordenados de menor a mayor)
- Por comodidad supondremos que los valores son enteros (se escoje como unidad la moneda más pequeña, en nuestro caso el céntimo de euro), y que existe una moneda de valor 1 ($m_1 = 1$)
- El problema consiste en dar cambio de una cantidad **C** usando el menor número posible de monedas.
- La solución se puede expresar como un vector de enteros, $s_1..s_n$, donde s_i indica el número de monedas/billetes de valor m_i que se usan para dar ese cambio.
- Criterio de validez (dar cambio exacto de C): $\sum_{i=1}^{n} s_i \cdot m_i = C$
- Función de coste: $\min \left\{ \sum_{i=1}^{n} s_i \right\}$
- Ejemplo: Si disponemos de monedas de valores 1,8 y 10, la forma óptima de dar cambio de 24 es usar 3 monedas de 8.

Cambio – Fuerza Bruta

```
type TCambio = array{[1..N]} of integer;

procedure Cambio(var S,Sopt: TCambio; C,I,N: integer);
{ Devuelve el cambio óptimo de una cantidad C en el
  vector Sopt. En una llamada explora todas las
  asignaciones posibles de S[I..N] }
var X : integer;
begin
  if I > N then { solución completa }
  begin
    if ∑_{i=1}^{n} s_i·m_i = C and ∑_{i=1}^{n} sopt_i < ∑_{i=1}^{n} s_i then Sopt := S;
  end else begin { solución parcial }
    for X := 0 to (C div M[I]) do
    begin
      S[I] := X;
      Cambio(S,Sopt,C,I+1,N); { recursividad }
    end
  end
end;
```

Cambio – Backtracking

```
type
  TCambio = record
    M: array{[1..N]} of integer;  { Valores faciales }
    V: array{[1..N]} of integer;  { Solución }
    Cact : integer; { Cambio de una solución parcial }
    Nmod : integer; { Número de monedas solución parcial }
  end;

procedure Cambio(var S,Sopt: TCambio; C,I,N: integer);
var X : integer;
begin
  if I > N then { solución completa }
  begin
    if S.Cact = C then { solución válida }
      if S.Nmod < Sopt.Nmod then Sopt := S;
  end else begin { solución parcial }
    for X := 0 to (C div M[I]) do
      if S.Cact+X*S.M[I] <= C then { Criterio poda }
      begin
        S.V[I] := X;
        S.Cact := S.Cact+X*S.M[I]; S.Nmod := S.Nmod+X;
        Cambio(S,Sopt,C,I+1,N);
        S.Cact := S.Cact-X*S.M[I]; S.Nmod := S.Nmod-X;
      end
  end
end;
```

Cambio – Programación Dinámica

- **Subestructura óptima**: Los parámetros del problema son el vector **m** (cuyo contenido no cambia en los subproblemas), el cambio **C** que se desea dar y el número de tipos de monedas que puedes usar, **n**.
- Para dar cambio de **C** usando monedas de tipos **1..n** puedes basarte en la solución óptima de subproblemas donde utilizas monedas de tipos **1..n-1** y se da cambio de una cantidad distinta.
- La idea es escoger todos los posibles valores de s_n (cuántas monedas de valor m_n se usan) y preguntar la manera óptima de dar cambio de la cantidad restante. Si definimos **φ(C,n)** como el coste (número total de monedas usadas) de la solución óptima de dar cambio de **C** usando monedas de tipos **1..n**, obtenemos la siguiente definición recursiva:

Caso base

$$\varphi(C,1) = C$$

$$\varphi(C,n) = \min_{\forall k \in \{0..\lfloor C/m_n \rfloor\}} \left\{ \varphi(C - k \cdot m_n, n-1) + k \right\}$$

Núm. de monedas de tipos 1..n-1 usadas

Núm. de monedas de tipo n usadas

Número máximo de monedas de valor m_n que se pueden usar sin sobrepasar C

Cambio – Problema restringido (recursivo)

- Función recursiva para el cálculo del problema restringido → sólo deseamos conocer el coste de la solución óptima, no la solución (array m se supone que es una variable global):

```
function Cambio(C,N: integer) : integer;
var K,Coste : integer;
begin
  if N = 1 then Result := C else
  begin
    Result := Cambio(C,N-1); { K = 0 }
    for K := 1 to C div M[N] do
    begin
      Coste := Cambio(C-K*M[N],N-1) + K;
      if Coste < Result then Result := Coste;
    end
  end
end;
```

Cambio – Problema restringido (TRP)

```
function Cambio(Ctot,Ntot: integer) : integer;
var
  Coste : array of array of integer;
  C,N,K,CosteAct,CosteMin : integer;
begin
  SetLength(Coste,Ctot+1,Ntot+1); { [0..Ctot,1..Ntot] }
  { Caso base }
  for C := 0 to Ctot do Coste[C,1] := C;
  { El resto de la matriz se rellena por columnas }
  for N := 2 to Ntot do
    for C := 0 to Ctot do
    begin
      CosteMin := Coste[C,N-1]; { K = 0 }
      for K := 1 to C div M[N] do
      begin
        CosteAct := Coste[C-K*M[N],N-1] + K;
        if CosteAct < CosteMin then CosteMin := CosteAct
      end;
      Coste[C,N] := CosteMin
    end;
  Result := Coste[Ctot,Ntot]; { Solución general }
 end;
```

Cambio – Solución general (I)

```
type
  { Vector de enteros }
  TVecEnt = array of integer;
  { Matriz de enteros }
  TMatEnt = array of array of integer;

procedure Cambio(Ctot,Ntot: integer; M: TVecEnt; var S: TVecEnt);
{ S[1..Ntot] es la solución óptima al problema de dar cambio de
  Ctot usando monedas de valores faciales M[1..Ntot] }
var
  Coste : TVecEnt; { Sólo se necesita una columna de la matriz }
  { NumMod[C,N] almacena el número de monedas de tipo N que se
    usan en la solución optima del problema de dar cambio de C
    con monedas de tipos 1..N }
  NumMod : TMatEnt;
  C,N,K,Kmin,CosteAct,CosteMin : integer;
begin
  SetLength(Coste,Ctot+1);           { [0..Ctot] }
  SetLength(NumMod,Ctot+1,Ntot+1); { [0..Ctot,1..Ntot] }
  { Caso base -> N = 1 }
  for C := 0 to Ctot do Coste[C] := C;
```

Cambio – Solución general (II)

```
  { El resto de la matriz se rellena por columnas }
  for N := 2 to Ntot do
    for C := 0 to Ctot do
    begin
      CosteMin := Coste[C]; Kmin := 0; { K = 0 }
      for K := 1 to C div M[N] do
      begin
        CosteAct := Coste[C-K*M[N]] + K;
        if CosteAct < CosteMin then
        begin
          CosteMin := CosteAct; Kmin := K;
        end
      end;
      Coste[C] := CosteMin; NumMod[C,N] := Kmin
    end;
  { Reconstrucción de la solución }
  C := Ctot;
  for N := Ntot downto 1 do
  begin
    S[N] := NumMod[C,N]; C := C-NumMod[C,N]*K
  end
end;
```

Problema de la Mochila 0/1

- Se dispone de una mochila de capacidad **M** kg y de **n** objetos de pesos $p_1..p_n$ y con valor monetario $v_1..v_n$ (todas las cantidades enteras positivas).
- El problema consiste en encontrar la mochila más valiosa: Determinar qué objetos se deben incluir en la mochila de manera que su peso total no supere su capacidad y su valor total sea el máximo posible.
- La solución se puede expresar como un vector $s_1..s_n$ donde s_i es **0** si el objeto **i** no se incluye en la mochila y **1** en caso contrario.
- Criterio de validez (no se rompa la mochila): $\sum_{i=1}^{n} s_i \cdot p_i \leq M$
- Función de coste (valor de la mochila): $\max\left\{\sum_{i=1}^{n} s_i \cdot v_i\right\}$
- Existen 2^n soluciones posibles.

Mochila 0/1 – Programación Dinámica

- **Subestructura óptima**: Los parámetros del problema son los vectores **p** y **v** (cuyo contenido no cambia en los subproblemas), la capacidad **M** de la mochila y el número de objetos que se pueden utilizar, **n**.
- Para dar llenar una mochila de capacidad **M** escojiendo objetos **1..n** es posible basarse en la solución óptima de subproblemas donde los objetos disponibles son **1..n-1** y la capacidad varía.
- idea es escoger todos los posibles valores de s_n (llevar o no el objeto **n**) y preguntar la manera óptima de llenar la mochila resultante. Si definimos **φ(n,M)** como el coste (valor monetario) de la mochila óptima de capacidad **M** y objetos **1..n**, obtenemos la siguiente definición recursiva:

Caso base

$$\varphi(0,M) = 0$$

Se incluye el objeto n (si cabe)

$$\varphi(n,M) = \max\{\varphi(n-1,M),\ \varphi(n-1,M-p_n)+v_n\}$$

No se incluye el objeto n

Mochila 0/1 – Solución general (I)

```
procedure Mochila(Ntot,Mtot: integer; P,V: TVecEnt; var S: TVecEnt);
{ S[1..Ntot] es la solución óptima al problema de la mochila de
  capacidad Mtot y Ntot objetos de pesos P[1..Ntot] y valores
  V[1..Ntot]. S[I] = 1 si el objeto I se incluye en la mochila.  }
var
  { Coste[N,M] almacena el valor de la mochila óptima con capacidad
    M y objetos 1..N
    Decis[N,M] almacena la decisión (0 ó 1) tomada sobre el objeto
    N en el problema de la mochila de capacidad C y objetos 1..N   }
  Coste,Decis : TMatEnt;
  N,M,Coste0,Coste1 : integer;
begin
  SetLength(Coste,Ntot+1,Mtot+1); { [0..Ntot,0..Mtot] }
  SetLength(Decis,Ntot+1,Mtot+1); { [1..Ntot,0..Mtot] }
  { Caso base -> N = 0 }
  for M := 0 to Mtot do Coste[0,M] := 0;
  { Resto de la matriz }
```

Mochila 0/1 – Solución general (II)

```
  { Resto de la matriz }
  for N := 1 to Ntot do
    for M := 0 to Mtot do
      if M < P[N] then { objeto N no cabe en la mochila }
      begin
        Coste[N,M] := Coste[N-1,M]; Decis[N,M] := 0;
      end else begin
        Coste0 := Coste[N-1,M];
        Coste1 := Coste[N-1,M-P[N]]+V[N];
        Coste[N,M] := Min(Coste0,Coste1);
        if Coste0 < Coste1 then Decis[N,M] := 0 else Decis[N,M] := 1
      end;
  { Reconstruir la solución }
  M := Mtot;
  for N := Ntot downto 1 do
  begin
    S[N] := Decis[N,M];
    if S[N] = 1 then M := M-P[N]
  end
end;
```

Problema de la Herencia

- Encontrar el reparto de n objetos de valores $v_1..v_n$ en dos lotes (A y B) que haga que el valor de ambos lotes sea lo más parecido posible.
- La solución se puede expresar como un vector $s_1..s_n$ donde s_i valga 0 si el objeto i es asignado al lote A y 1 si es asignado al lote B.
- Si V_A y V_B son los valores del lote A y B, respectivamente, el objetivo es conseguir que la cantidad V_B-V_A sea lo más cercana a cero posible
- Un primer enfoque puede consistir en encontrar la solución óptima del problema de n objetos a partir de la solución óptima del problema con n-1 objetos.
- Contraejemplo: Si tenemos 4 objetos de valores (10,4,6,8) el reparto óptimo es claramente (A,A,B,B). Sin embargo esta respuesta no se puede obtener a partir de la solución óptima del subproblema con 3 objetos (10,4,6), ya que en este caso la solución óptima es (A,B,B).
- Tal como está planteado, este problema no cumple el **principio de suboptimalidad**: Las soluciones óptima de los subproblemas deben servir para construir la solución óptima del problema general.

Problema del Reparto Desequilibrado

- La solución viene de considerar una generalización del problema de la herencia: En realidad lo que necesitamos es conocer un reparto en el que los lotes A y B esten desequilibrados en la cantidad adecuada para que al añadir un nuevo objeto (a A ó B) el reparto sea equilibrado.
- Tenemos un parámetro extra, d, el desequilibrio deseado. Ahora el objetivo es conseguir un reparto en el que V_B-V_A sea lo más cercano posible al valor d.
- Si definimos la función de coste φ(n,d) como la diferencia entre los lotes A y B (V_B-V_A), donde el objetivo es conseguir que ésta diferencia sea lo más cercana a d posible, entonces ya que con el objeto n tenemos dos posibilidades, incluirle en A ó incluirle en B, los subproblemas (n-1 objetos) que nos interesan son aquellos que obtienen un desequilibrio más cercano a d+v_n (para que al añadirle a A nos aproximemos a d) y d-v_n (para que al añadirle a B nos aproximemos a d):

$$\varphi(0,d) = 0$$

$$\varphi(n,d) = \text{mas_cercano}\left(d, \begin{Bmatrix} \varphi(n-1, d+v_n) - v_n \\ \varphi(n-1, d-v_n) + v_n \end{Bmatrix} \right)$$

Función que escoge de los valores del 2º argumento el más cercano al 1º

Herencia – Solución general (I)

```
procedure Herencia(Ntot: integer; V: TVecEnt; var S: TVecEnt);
{ S[1..Ntot] es la solución óptima al problema de la herencia
  con Ntot objetos de valores V[1..Ntot].
  S[I] = 0 si el objeto I se incluye en el lote A. }
var
  { Coste[N,D] almacena la diferencia entre lotes (lote B - lote A)
    del mejor reparto de objetos 1..N con objetivo de conseguir
    un desequilibrio lo más cercano posible a D.
    Decis[N,M] almacena la decisión (0 ó 1) tomada sobre el objeto
    N en el problema con objetos 1..N y desequilibrio D }
  Coste,Decis : TMatEnt;
  Vsum : integer; { Suma de valor de todos los objetos }
  I,N,D,Coste0,Coste1 : integer;
begin
  Vsum := 0;
  for I := 1 to Ntot do Vsum := Vsum+V[I];
  SetLength(Coste,Ntot+1,2*Vsum+1); { [0..Ntot,-Vsum..Vsum] }
  SetLength(Decis,Ntot+1,2*Vsum+1); { [1..Ntot,-Vsum..Vsum] }
  { Caso base -> N = 0 }
  for D := -Vsum to Vsum do Coste[0,D+Vsum] := 0;
  { Resto de la matriz }
```

Herencia – Solución general (II)

```
for N := 1 to Ntot do
  for D := -Vsum to Vsum do
  begin
    { Coste de añadir a A, ∞ si no es posible }
    if D+V[N] > Vsum then Coste0 := ∞ else
      Coste0 := Coste[N-1,D+V[N]+Vsum]-V[N];
    { Coste de añadir a B, ∞ si no es posible }
    if D-V[N] < -Vsum then Coste1 := ∞ else
      Coste1 := Coste[N-1,D-V[N]+Vsum]+V[N];
    if Coste0 < Coste1 then
    begin
      Coste[N,D+Vsum] := Coste0; Decis[N,D+Vsum] := 0
    end else begin
      Coste[N,D+Vsum] := Coste1; Decis[N,D+Vsum] := 1
    end
  end;
{ Reconstruir solución }
D := 0;
for N := Ntot downto 1 do
begin
  S[N] := Decis[N,D+Vsum];
  if S[N] = 0 then D := D+V[N] else D := D-V[N]
end;
```

Problema del Producto de Matrices

- Encontrar la forma óptima de parentizar un producto de n matrices de las cuales conocemos sus dimensiones (la matriz i-ésima tiene f_i filas y c_i columnas) de manera que al evaluarlo se realicen la menor cantidad posible de productos elementales.
- El producto de matrices no es conmutativo (no se puede cambiar el orden de las matrices) pero si asociativo. Al multiplicar dos matrices de dimensiones (f_i, c_i) y (f_{i+1}, c_{i+1}) se tiene que:
  - $c_i = f_{i+1}$ para que se puedan multiplicar
  - La matriz resultante tiene dimensiones (f_i, c_{i+1})
  - Se realizan $f_i \cdot c_i \cdot c_{i+1}$ productos de elementos
- La entrada del problema son las dimensiones de las n matrices. Dado que el número de columnas de cada matriz debe ser igual al de filas de la siguiente, sólo se necesitan n+1 valores, que se proporcionan en un vector $d_1..d_{n+1}$. La matriz i-ésima tiene dimensiones (d_i, d_{i+1})
- La salida será una expresión matemática textual que indique el orden de evaluación

Producto de Matrices – Solución general (I)

```
procedure ProdMat(N: integer; D: TVecEnt);
{ Escribe la manera óptima de calcular el producto de n matrices
  con dimensiones D[1..N+1] }
var
  { Coste[I,J] almacena el número de productos elementales de la
    evaluación óptima del producto de las matrices I..J
    Decis[I,J] almacena la manera en que se debe parentizar, en
    el primer nivel, el producto de matrices I..J. Si K = Decis[I,J]
    entonces el producto debe evaluarse como (I..K)x(K+1..J) }
  Coste,Decis : TMatEnt;
  I,J,Coste0,Coste1 : integer;
begin
  SetLength(Coste,N+1,N+1); { [1..N,1..N] }
  SetLength(Decis,N+1,N+1); { [1..N,1..N] }
  { Caso base -> I = J }
  for I := 0 to Mtot do Coste[I,I] := 0;
  { Resto de la matriz (diagonal superior) }
```

Producto de Matrices – Solución general (II)

```
  { Resto de la matriz (diagonal superior) }
  for U := 2 to n do { diagonal u-ésima }
  begin
    I := 1; J := U;
    repeat
      CosteMin := ∞;
      for K := I to J-1 do
      begin
        CosteAct := Coste[I,K]+Coste[K+1,J]+D[I]*D[K]*D[J];
        if CosteAct < CosteMin then
        begin
          CosteMin := CosteAct; Kmin := K
        end
      end;
      Coste[I,J] := CosteAct;
      Decis[I,J] := Kmin;
      I := I+1; J := J+1; { Siguiente celda de diagonal }
    until (I > N) or (J > N)
  end;
  { Escribir resultado }
  EscribeProd(Decis,1,N)
end;
```

Producto de Matrices – Solución general (III)

```
procedure EscribeProd(const Soluc: TMatEnt; I,J: integer);
{ Escribe (recursivamente) el producto de las matrices I..J }
var K : integer;
begin
  if I = J then
    write('M',I)
  else
    begin
      K := Soluc[I,J];
      if K > I then write('(');
      EscribeProd(I,K);
      if K > I then write(')');
      write('x');
      if K+1 < J then write('(');
      EscribeProd(K+1,J);
      if K+1 < J then write(')')
    end
end;
```