

Técnica de diseño #4: Backtracking



Análisis y Diseño de Algoritmos
Ing. Román Martínez M.

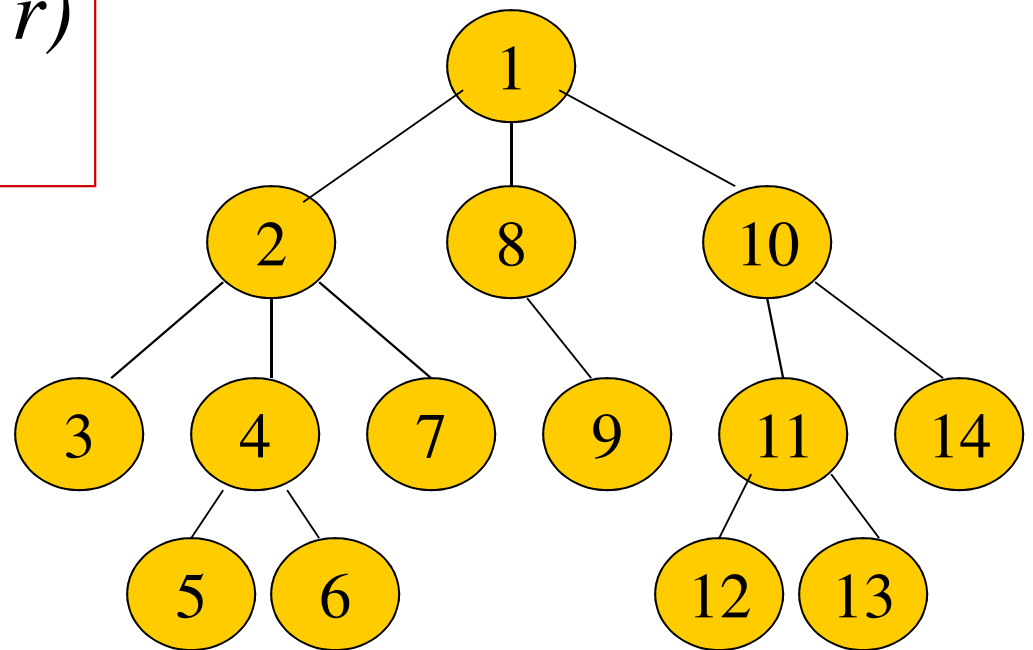
Backtracking



- Utilizado para resolver problemas en los que se tiene que seleccionar una secuencia de objetos de un conjunto dado, y la cuál cumple con cierto criterio o restricción.
- El backtracking es una modificación a la búsqueda en profundidad (**depth-first**), equivalente al recorrido en **preorden**, en el **árbol de búsqueda de soluciones** al problema.
- *NOTA:* El árbol de búsqueda de soluciones, NO necesariamente es un árbol binario.

Algoritmo Depth-first

```
void depth-first (Nodo r)  
{ Nodo h;  
    Visitar r;  
    for (cada hijo h de r)  
        depth_first(h);  
}
```



El problema de las 8 reinas...



- En un tablero de ajedrez, ¿en qué posiciones se pueden colocar 8 reinas sin que se ataquen una a otra?
- Una reina ataca a otra, cuando está en el mismo renglón, o en la misma columna, o en la misma diagonal dentro del tablero.
- El problema se puede generalizar como el problema de las n reinas, y puede tener otros contextos de aplicación.
- *¿Cómo se puede resolver el problema?*

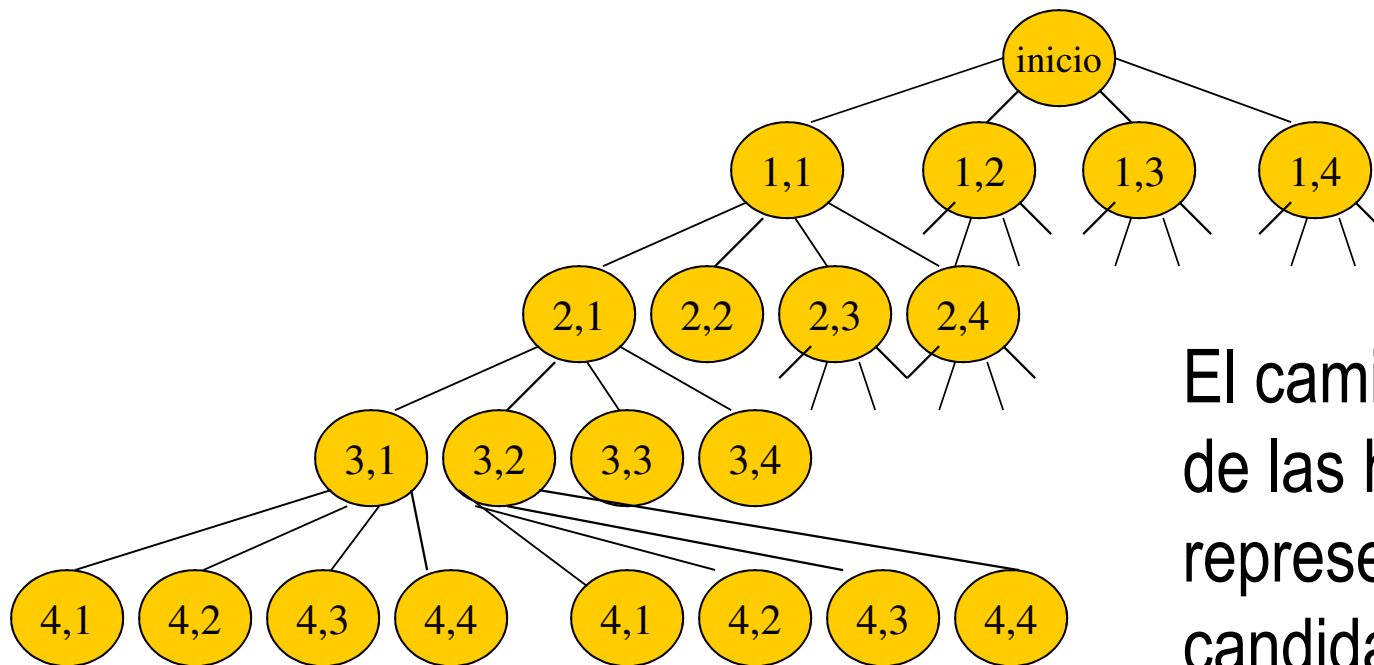
El problema de las n reinas...



- De todas las posibles posiciones dentro del tablero de $n \times n$, se tienen que seleccionar las n posiciones que cumplan la condición de no estar en el mismo renglón, columna o diagonal que otra posición seleccionada del tablero.
- ¿Cuántas posibles combinaciones de posiciones se pueden tener para de ahí seleccionar la solución?
- Por FUERZA BRUTA se obtendrían n^n conjuntos candidatos !

Ejemplo... para $n = 4$

Árbol de búsqueda de soluciones:



El camino a cada una de las hojas en el árbol representa un conjunto candidato a la solución.

$4^4 = 256$ conjuntos candidatos

El problema de las n reinas...



- La técnica de backtracking, irá generando los posibles conjuntos candidatos a solución, pero evalúa su formación con respecto al criterio o restricción del problema...
- Cuando la selección de una posición, no cumple la restricción, el algoritmo elimina la posibilidad de ese conjunto "regresando" a conformar otra posible solución...
- De esta manera se eliminan conjuntos candidatos eficientemente, y se puede encontrar el conjunto solución sin necesidad de calcular todos los conjuntos candidatos...

Algoritmo general con Backtracking

```
void verifica_nodo (Nodo r)
{
  Nodo h;
  if (el nodo r cumple criterio de selección)
    if (se obtiene la solución con r)
      desplegar la solución;
    else
      for (cada hijo h de r)
        verifica_nodo(h);
}
```

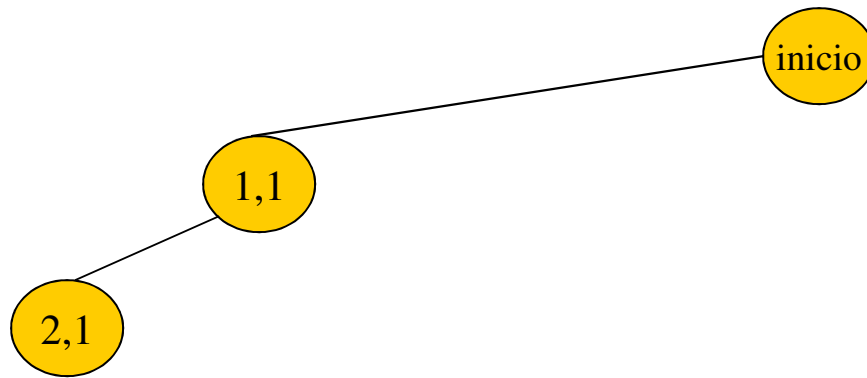


*Restricción
en la búsqueda*



Búsqueda Depth-first

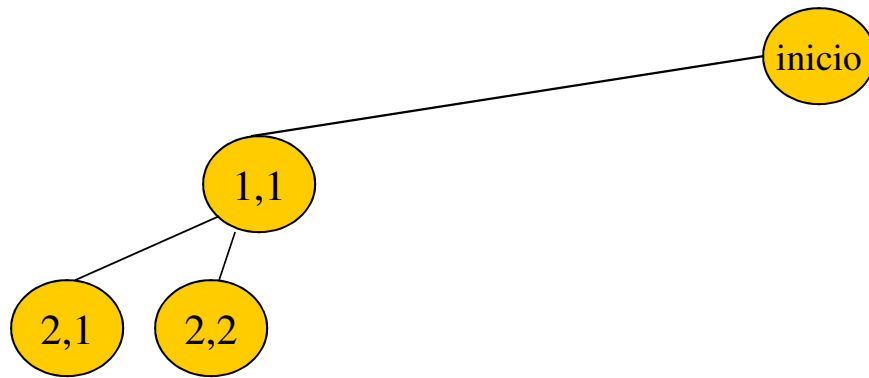
Ejemplo... para $n = 4$



*La estrategia ASEGURA
no ocupar el mismo renglón*

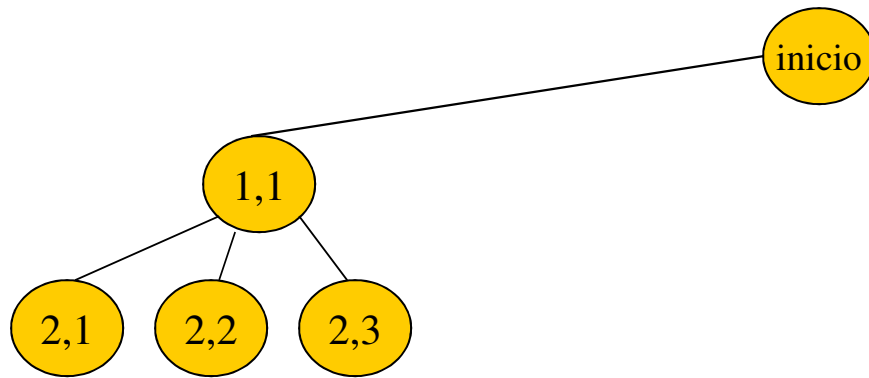
**NO se cumple el criterio
(misma columna)**

Ejemplo... para $n = 4$



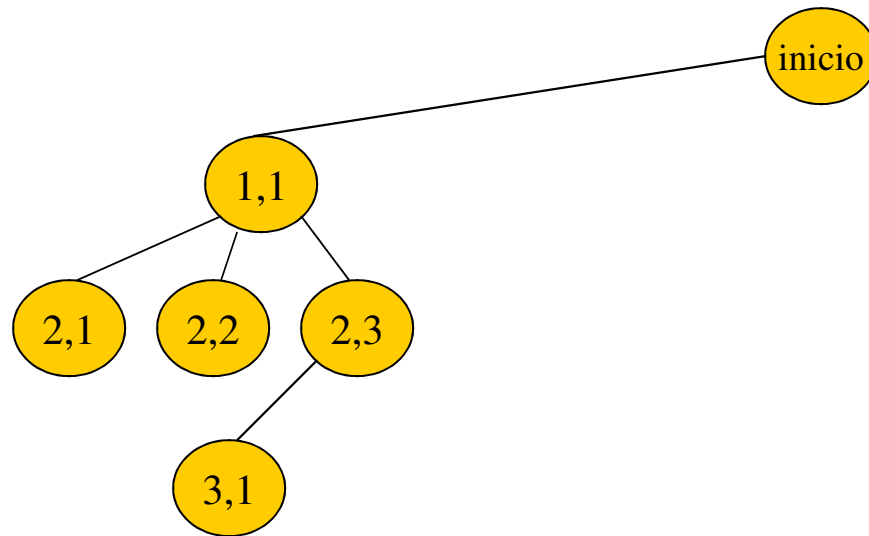
***NO se cumple el criterio
(misma diagonal)***

Ejemplo... para $n = 4$



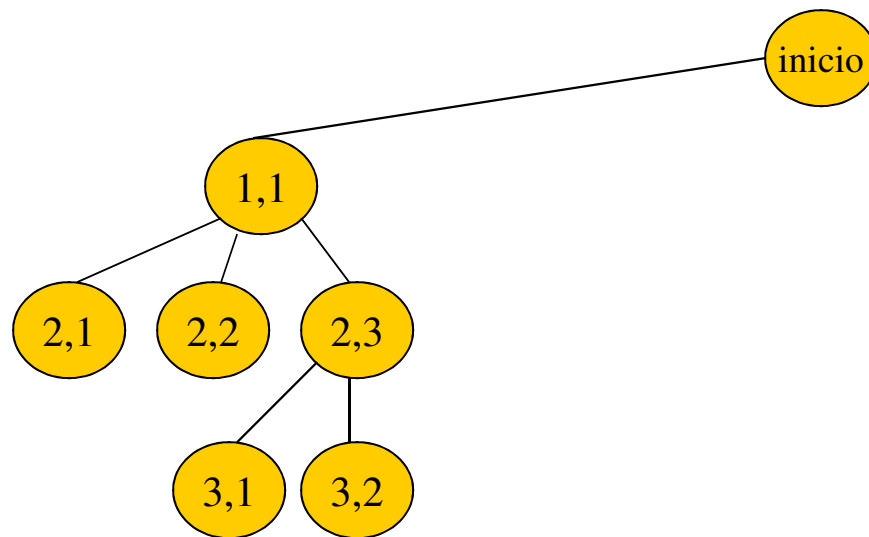
***OK... adelante en la
búsqueda !***

Ejemplo... para $n = 4$



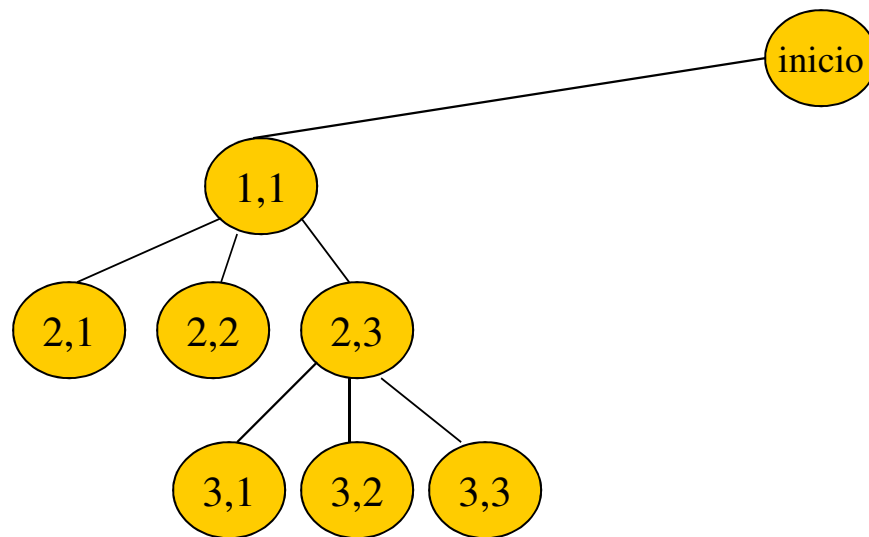
***NO se cumple el criterio
(misma columna)***

Ejemplo... para $n = 4$



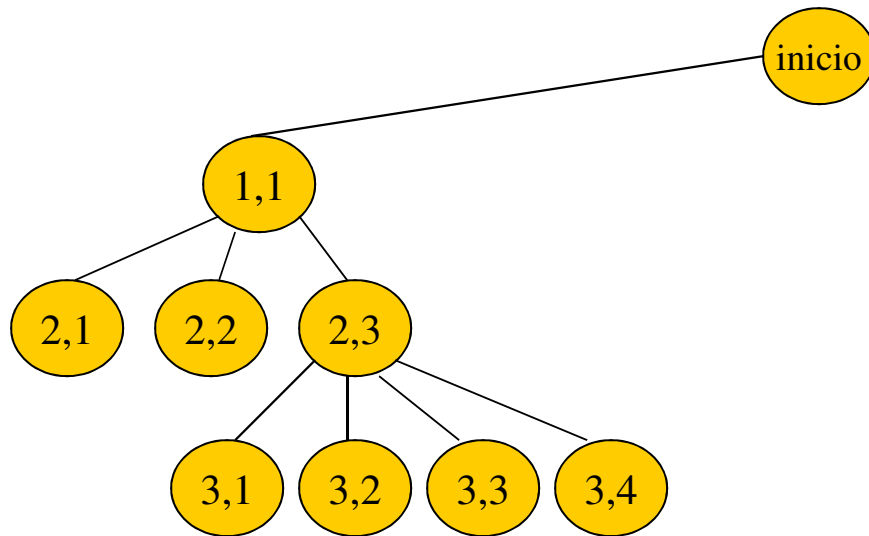
***NO se cumple el criterio
(misma diagonal que 2,3)***

Ejemplo... para $n = 4$



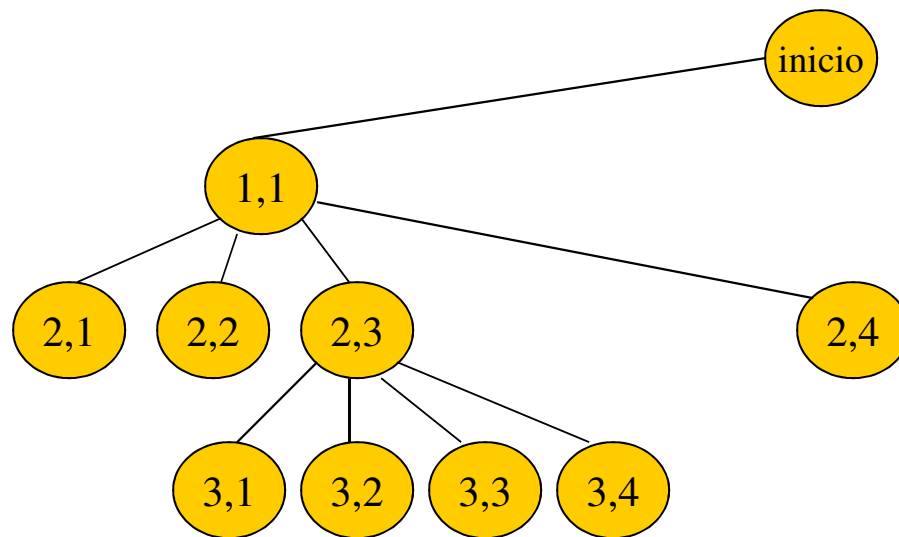
***NO se cumple el criterio
(misma diagonal que 1,1)***

Ejemplo... para $n = 4$



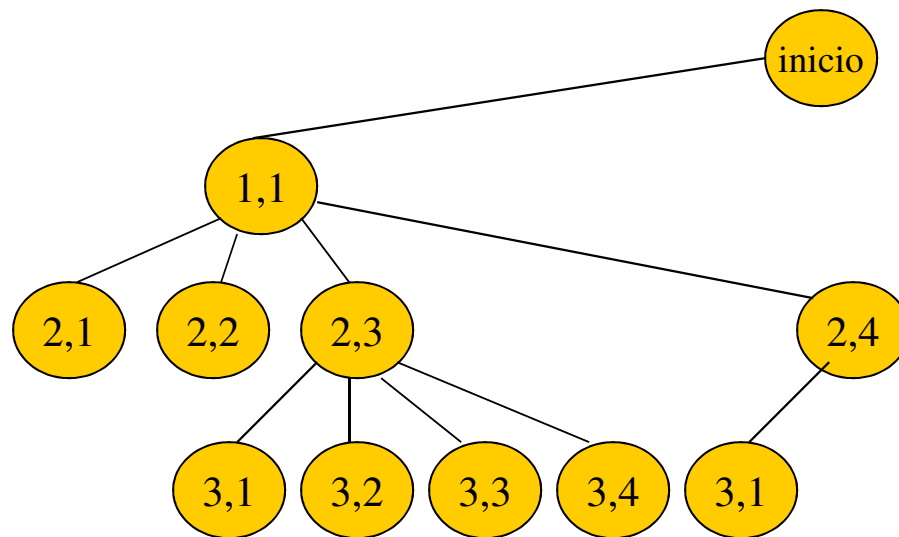
***NO se cumple el criterio
(misma diagonal que 2,3)***

Ejemplo... para $n = 4$



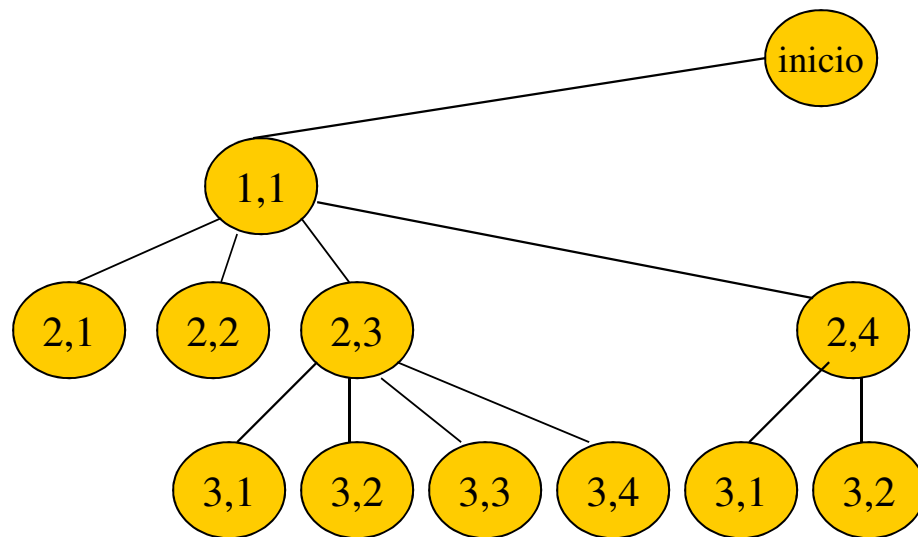
***OK... adelante con la
búsqueda!***

Ejemplo... para $n = 4$



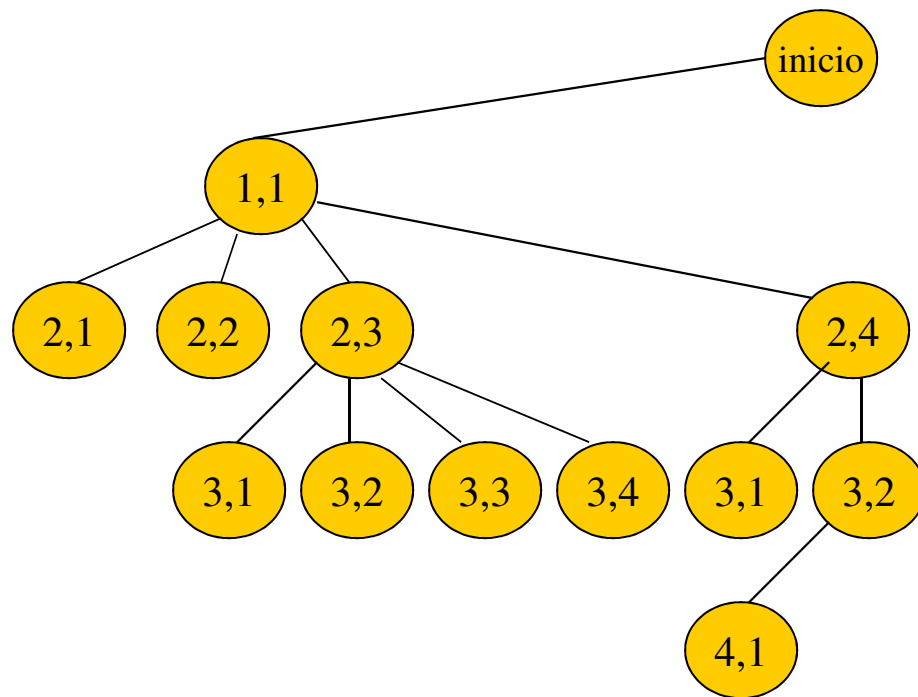
***NO se cumple criterio
(misma columna)***

Ejemplo... para $n = 4$



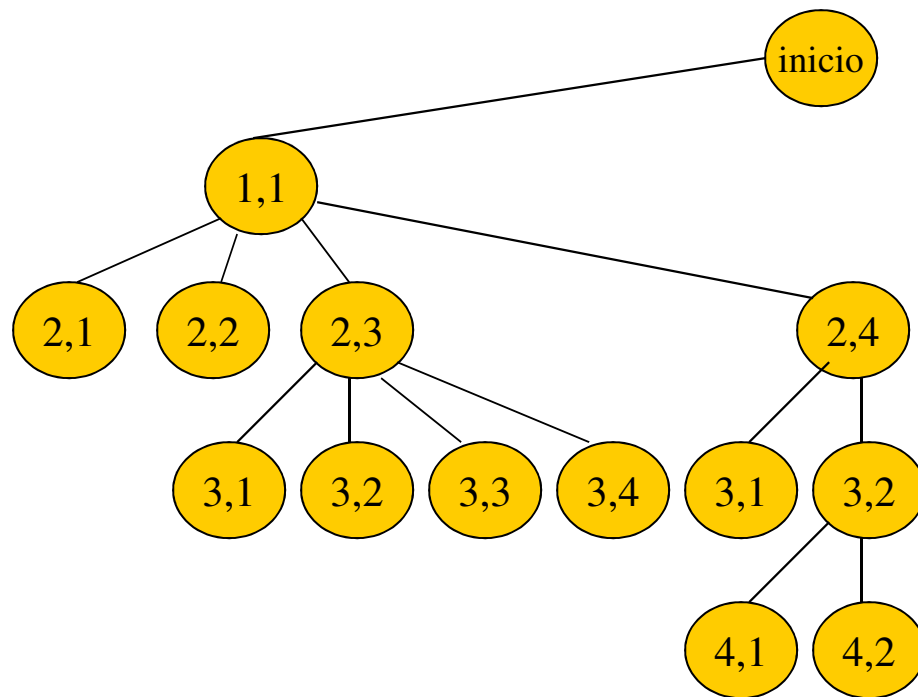
***OK... adelante con la
búsqueda!***

Ejemplo... para $n = 4$



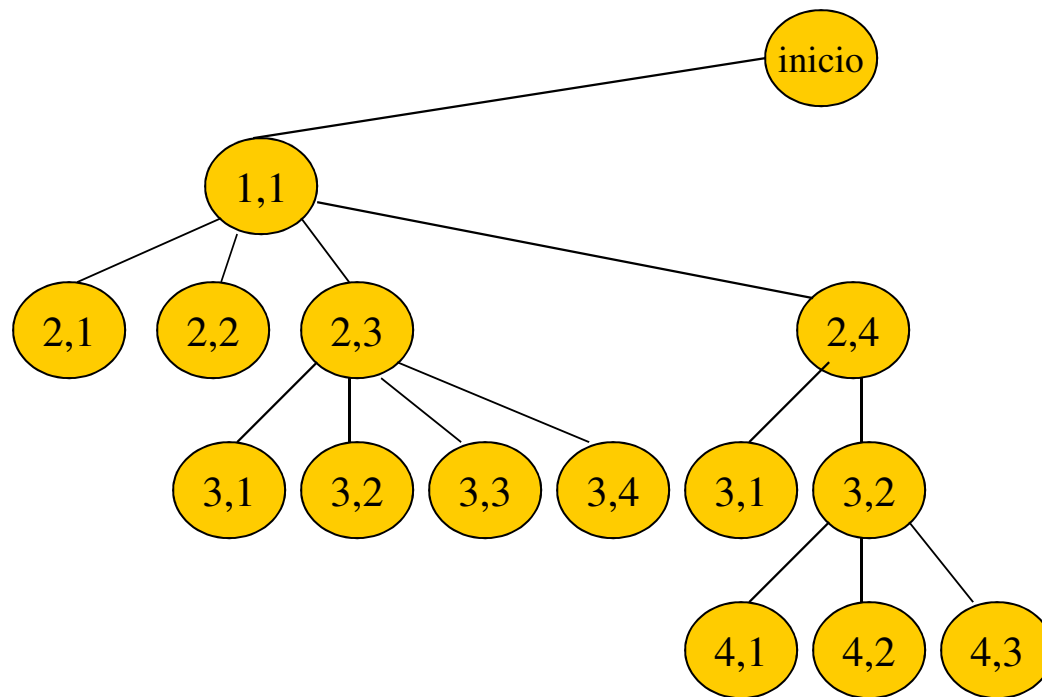
***NO se cumple criterio
(misma columna)***

Ejemplo... para $n = 4$



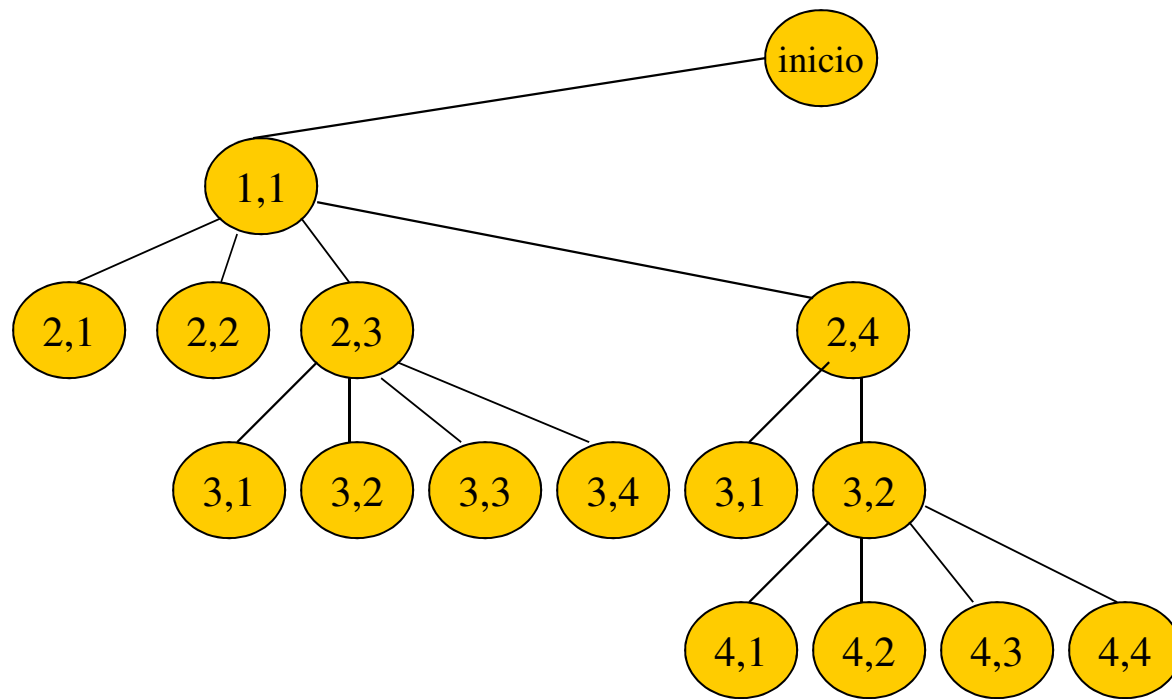
***NO se cumple criterio
(misma columna)***

Ejemplo... para $n = 4$



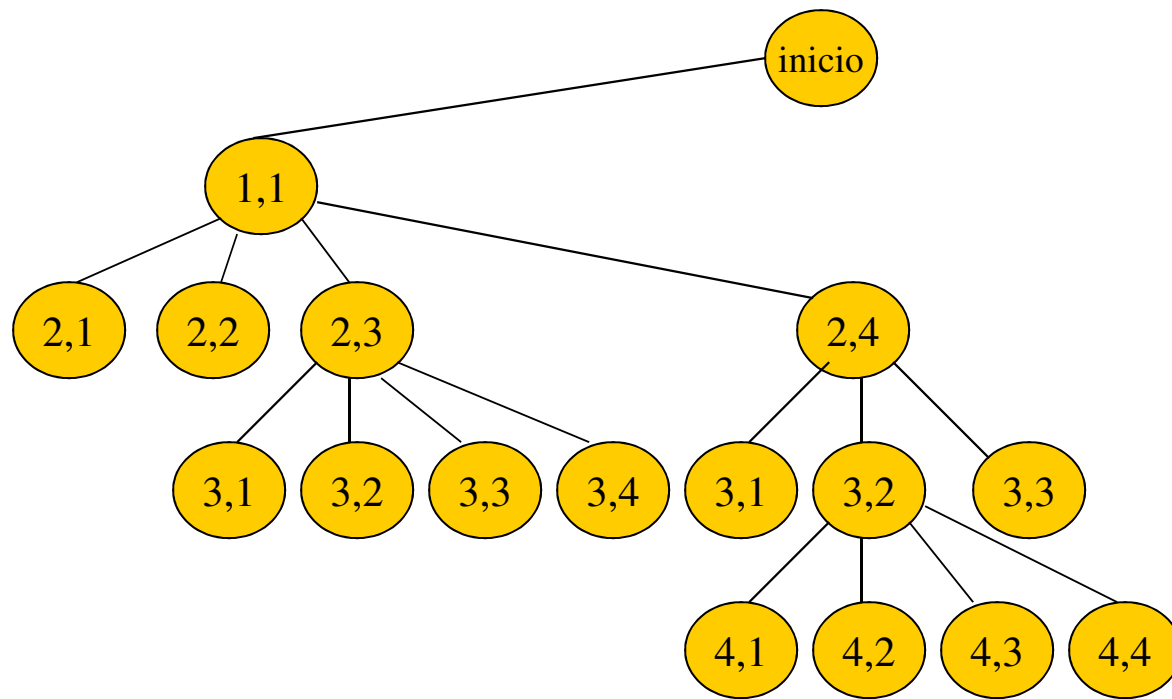
***NO se cumple criterio
(misma diagonal 3,2)***

Ejemplo... para $n = 4$



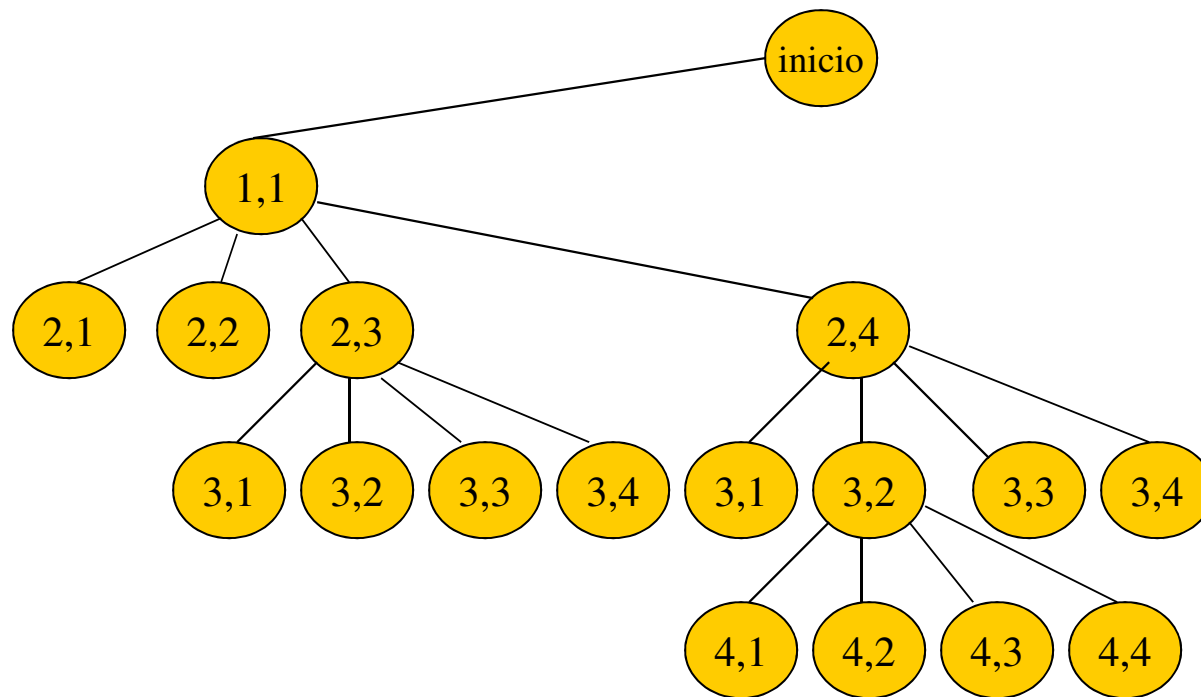
***NO se cumple criterio
(misma columna)***

Ejemplo... para $n = 4$



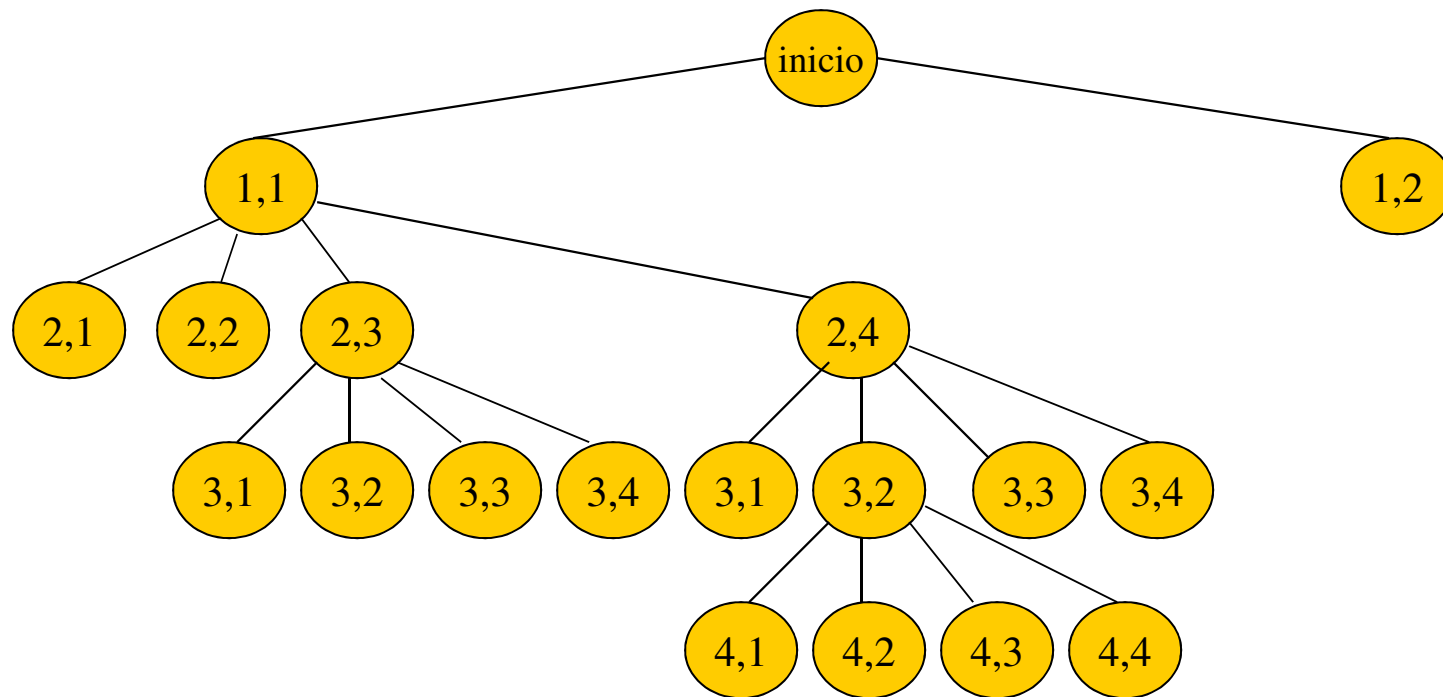
***NO se cumple criterio
(misma diagonal)***

Ejemplo... para $n = 4$



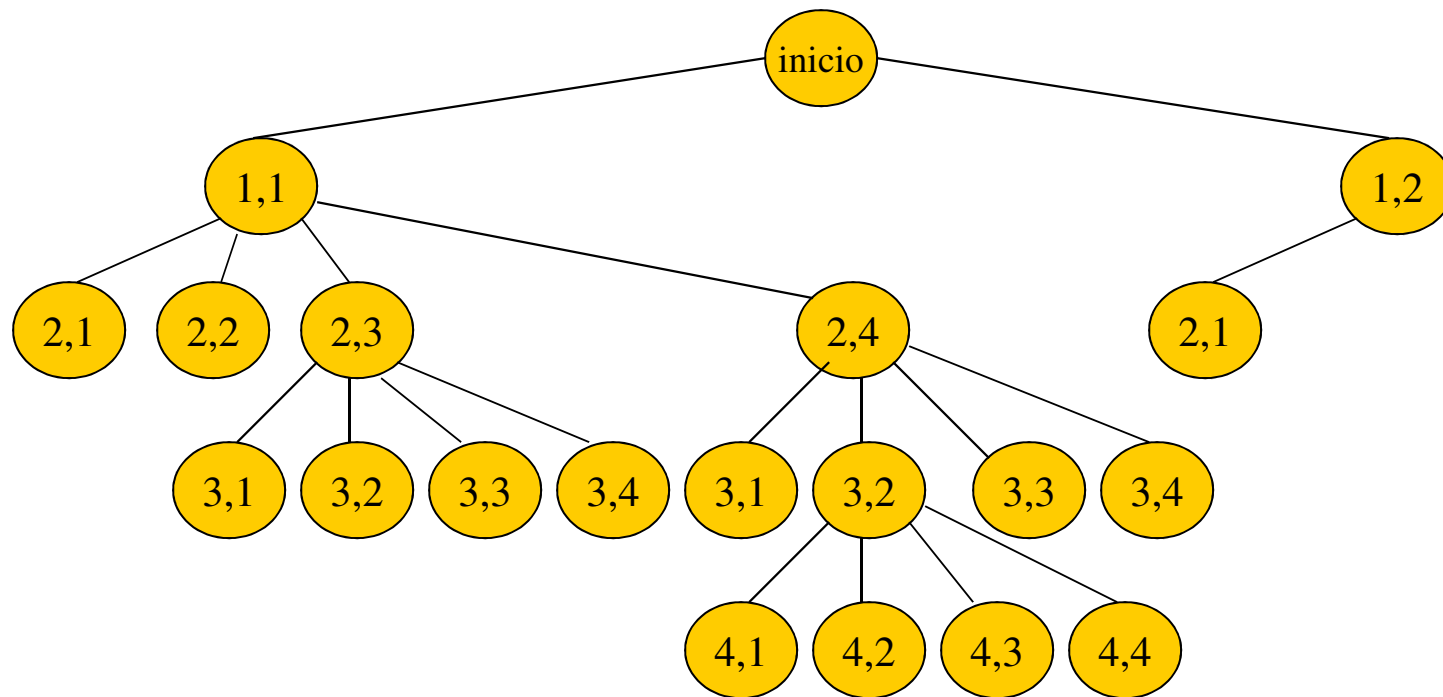
***NO se cumple criterio
(misma columna)***

Ejemplo... para $n = 4$



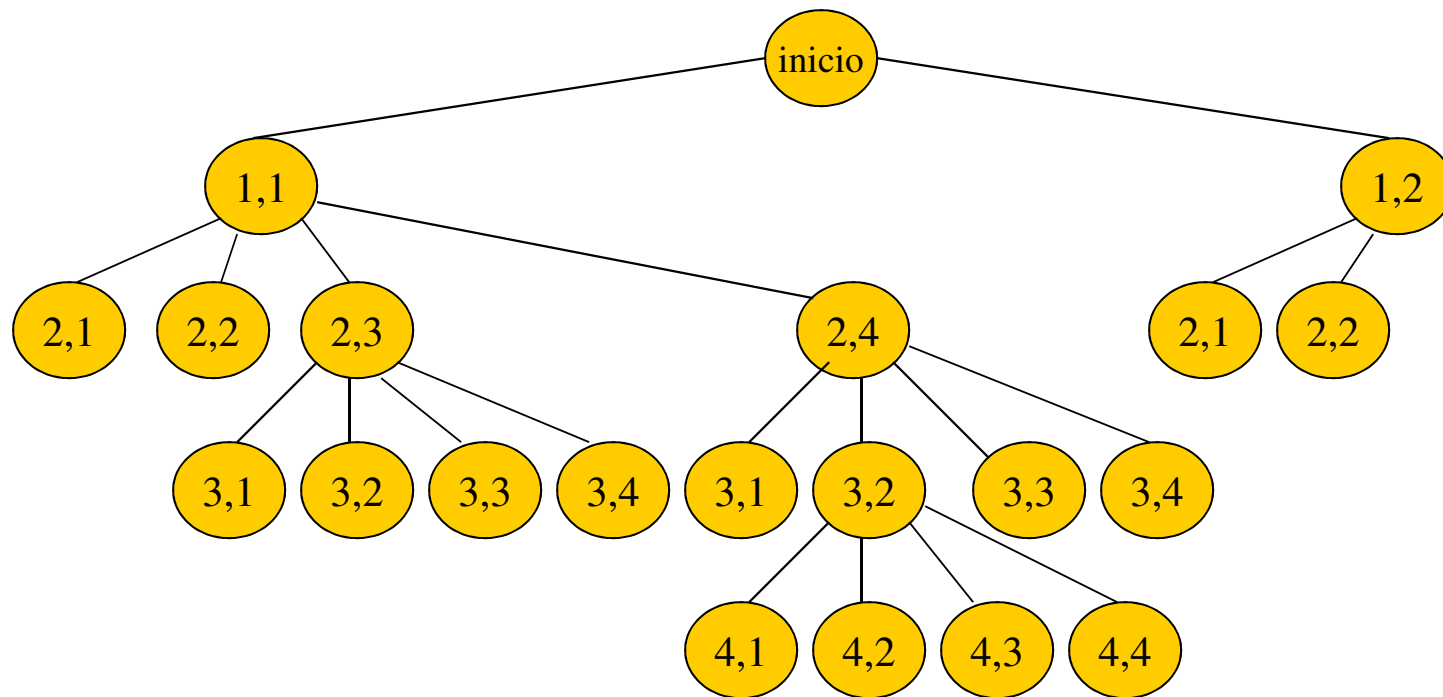
OK... adelante con la búsqueda!

Ejemplo... para $n = 4$



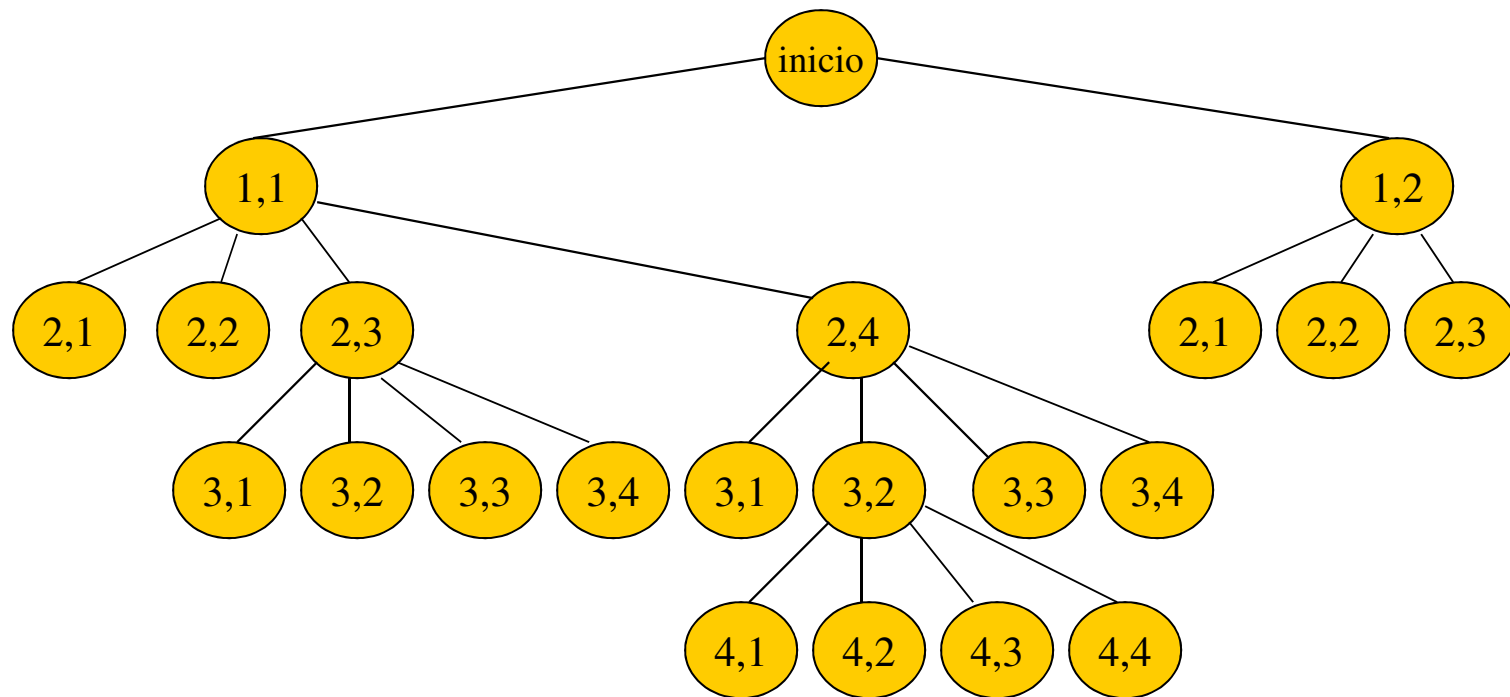
***NO cumple criterio
(misma diagonal)***

Ejemplo... para $n = 4$



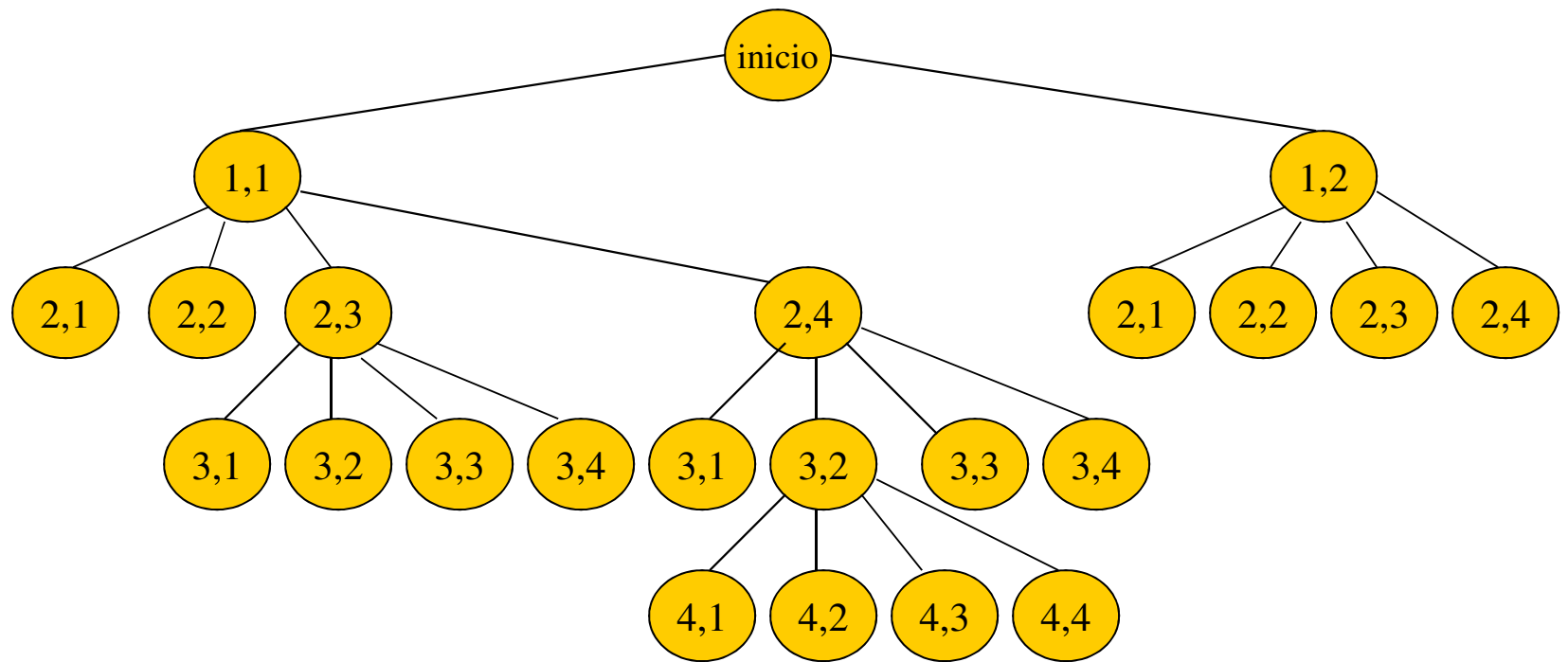
***NO cumple criterio
(misma columna)***

Ejemplo... para $n = 4$



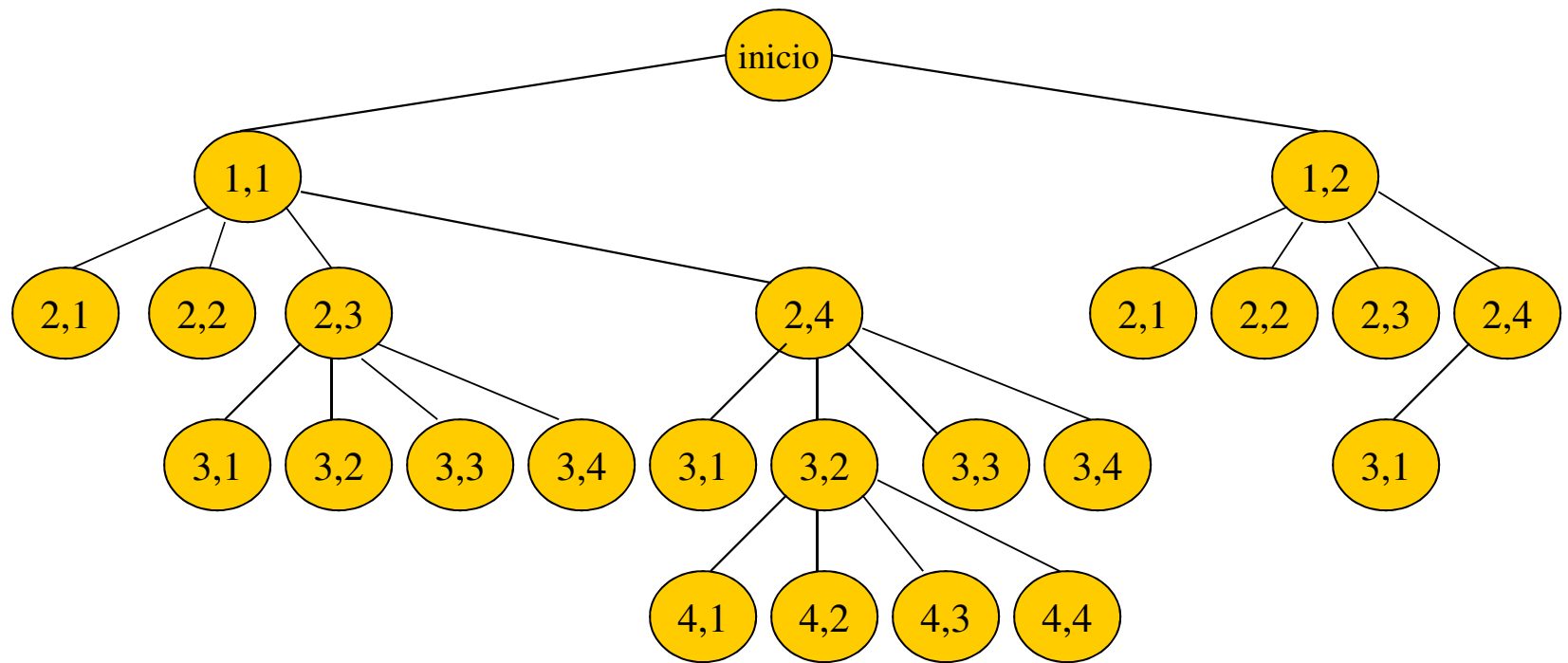
***NO cumple criterio
(misma diagonal)***

Ejemplo... para $n = 4$



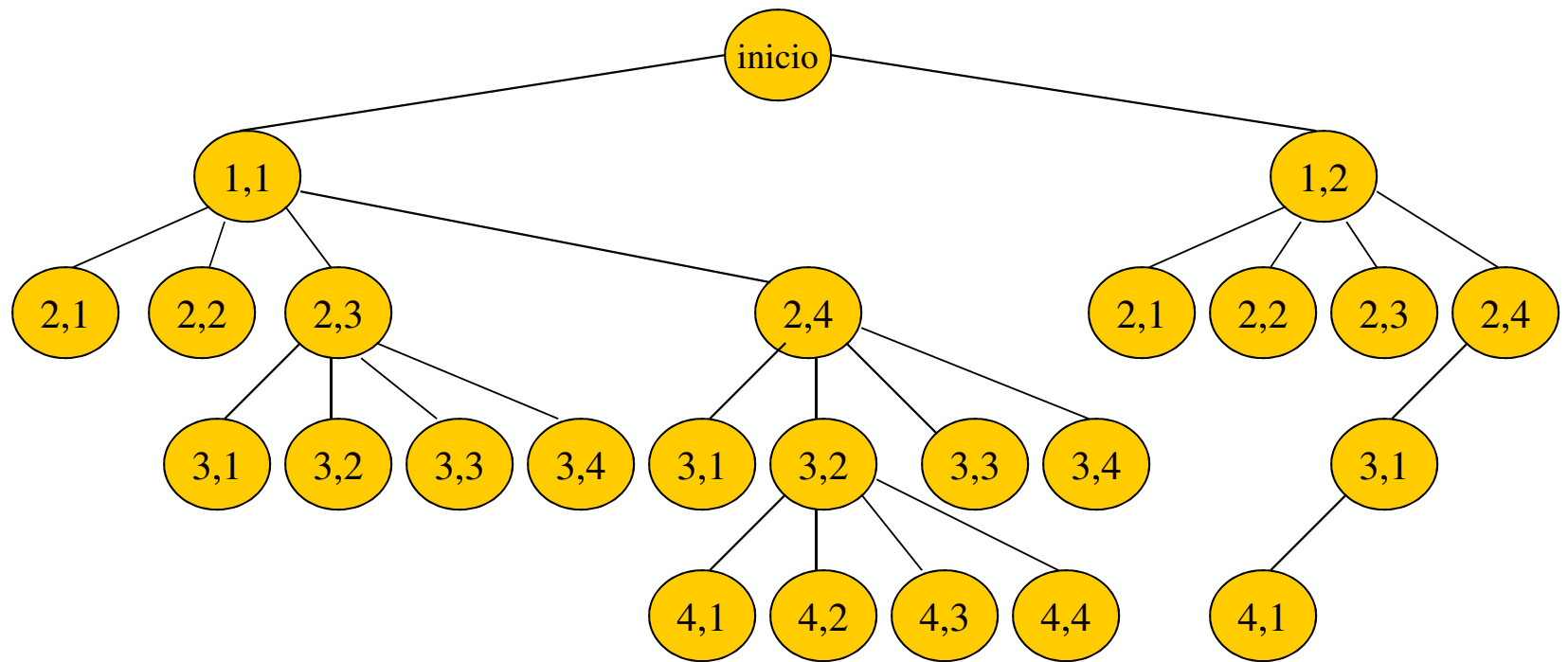
***OK... adelante con la
búsqueda!***

Ejemplo... para $n = 4$



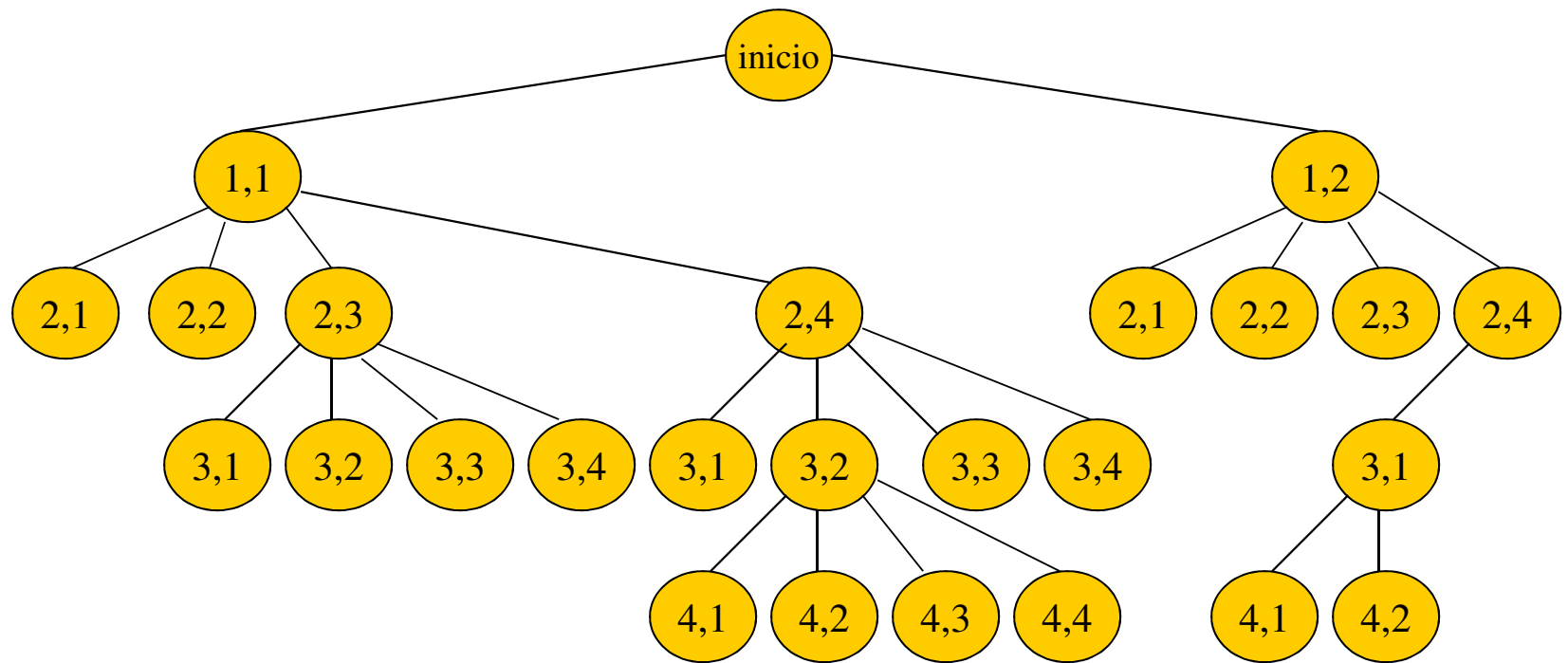
***OK... adelante con la
búsqueda!***

Ejemplo... para $n = 4$



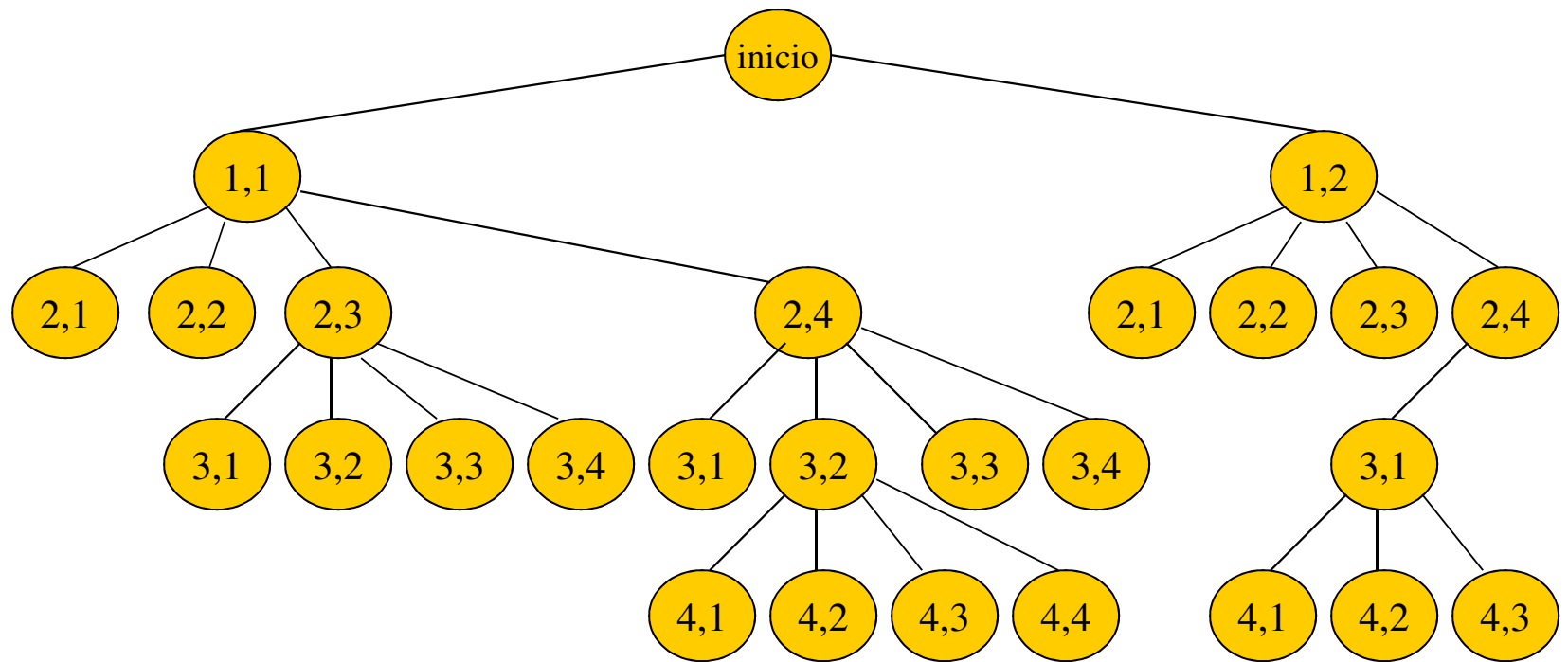
***NO cumple el criterio
(misma columna)***

Ejemplo... para $n = 4$



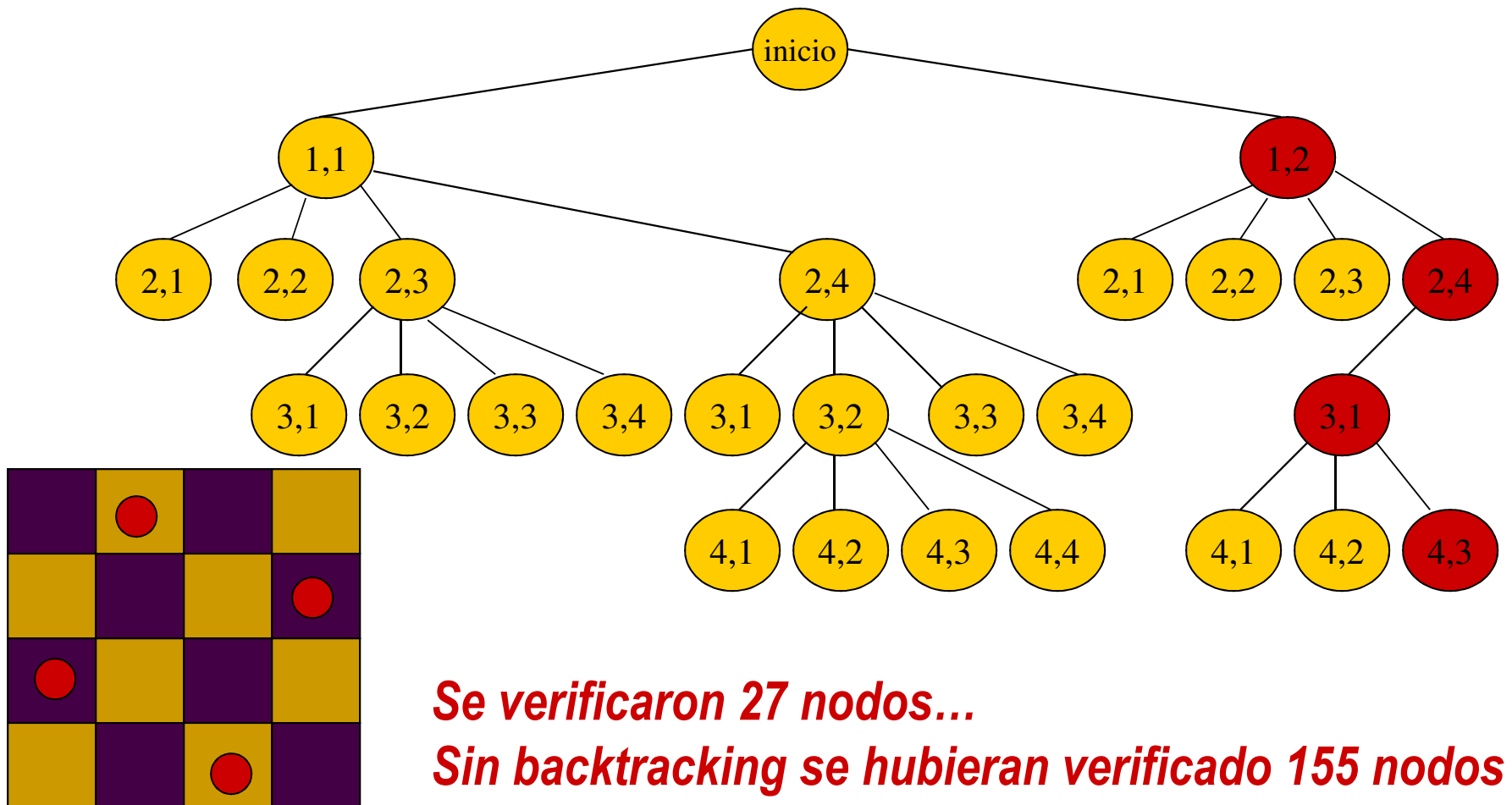
***NO cumple el criterio
(misma columna)***

Ejemplo... para $n = 4$



**OK... se encontró
solución !!**

Ejemplo... para $n = 4$



Algoritmo específico para el problema de las n reinas...

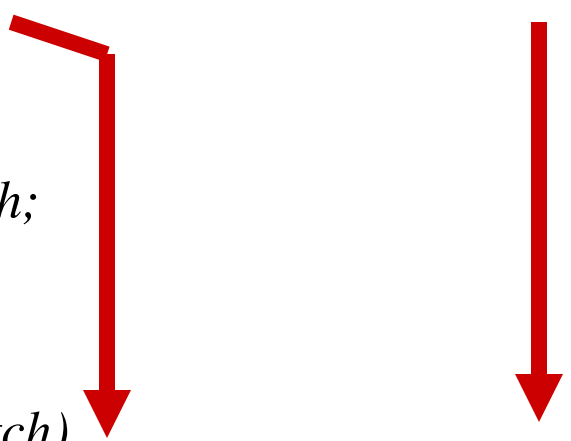
- Se utilizará un arreglo global llamado *col*, donde *col[i]* contiene la posición de la columna para la reina en el renglón *i*.

```
void reinas (indice i)
{ indice j;
  if (cumple(i))
    if (i==n)
      for(int k=1; k<=n; k++) cout << col[k];
    else
      for (j=1; j<=n; j++)
        { col[i+1] = j; reinas(i+1); }
}
```

Algoritmo específico para el problema de las n reinas...

- El cumplimiento del criterio consiste en verificar si no se está en la misma columna o en la misma diagonal.

```
bool cumple (indice i)  
{ indice k; bool switch;  
  k = 1;  
  switch = true;  
  while (k < i && switch)  
  { if(col[i] == col[k] || abs(col[i]-col[k]) == i-k)  
    switch = false;  
    k++; }  
  return switch;  
}
```



Eficiencia con backtracking



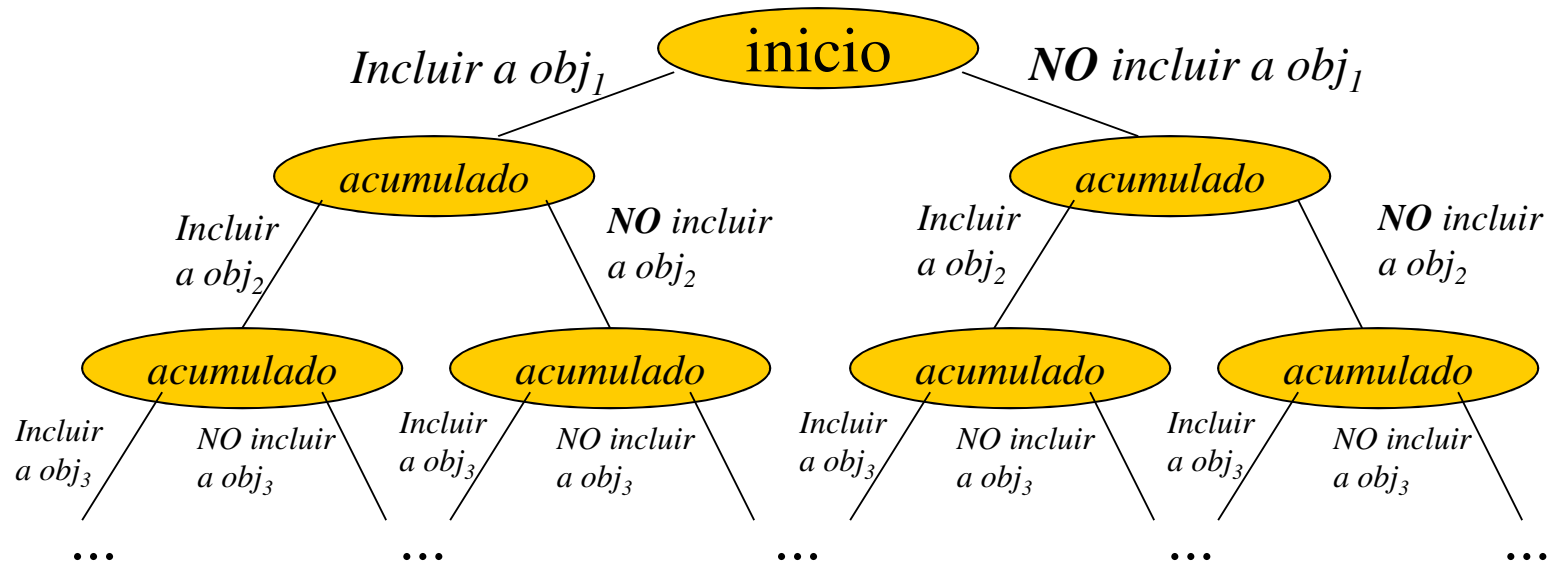
- Evidentemente, cómo el caso de las 'n' reinas lo ejemplifica, existen problemas en que aplicar la técnica del backtracking representa un beneficio significativo en la eficiencia del algoritmo...
- Sin embargo, el análisis formal de los algoritmos que utilizan backtracking es complejo...
- La técnica (algoritmo) de Monte Carlo es una forma de tener una estimación formal del comportamiento de un algoritmo con backtracking (*ver libro, sección 5.3*)...

El problema de los subconjuntos que acumulan cierto valor (*Sum-of-subsets*)

- Dado un conjunto de objetos con cierto valor asignado a cada uno de ellos, ¿qué subconjuntos de objetos se pueden seleccionar de tal manera que la suma de sus valores sea exáctamente cierto valor establecido?
- Ejemplo: Si $obj_1=5$, $obj_2=6$, $obj_3=10$, $obj_4=11$, $obj_5=16$, ¿cuáles son los subconjuntos que acumulan exáctamente 21?
 $\{obj_1, obj_2, obj_3\}$
 $\{obj_1, obj_5\}$
 $\{obj_3, obj_4\}$

Solución con Backtracking

- ***¿Cómo se representaría el árbol de búsqueda de soluciones?***



- Las hojas del árbol que tengan el acumulado buscado, indican cuáles son los subconjuntos solución.

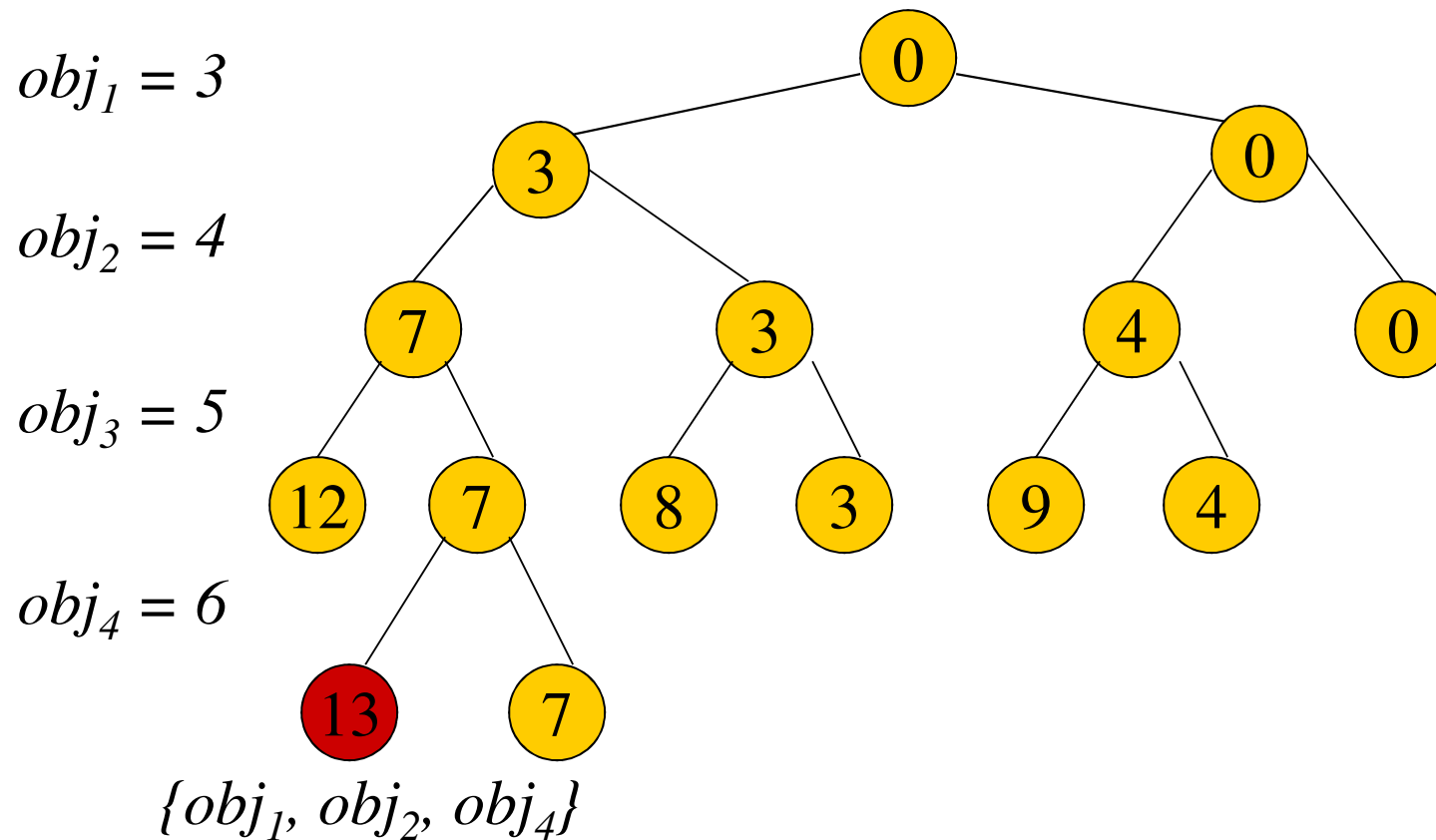
Solución con Backtracking



- ***¿Cuál es el criterio de selección de nodos que se debe aplicar en este problema?***
- Estratégicamente, se trabajará con los objetos ordenados de acuerdo a su valor de menor a mayor...
 - Si el acumulado excede el valor que se busca, se descarta ese subconjunto:
$$acum + valor\ del\ siguiente\ objeto > Valor\ buscado$$
 - Si el acumulado más la suma de los valores restantes no llega al valor buscado, se descarta ese subconjunto:
$$acum + resto\ de\ los\ valores < Valor\ buscado$$

Ejemplo

- Sea $obj_1 = 3$, $obj_2 = 4$, $obj_3 = 5$, $obj_4 = 6$ y el valor a acumular **13**...



Algoritmo...



- Se utilizará un arreglo global llamado *include*, donde *include[i]* indica si el objeto *i* forma parte de la solución o no.
- El arreglo *obj* contiene los valores ordenados de los objetos.
- *VALOR* es el acumulado que se busca.
- El acumulado y los totales, se controlan automáticamente a través de los parámetros y las llamadas recursivas.

Algoritmo...

```
void sum_of_subsets (indice i, int acum, int total)
{  if (acum+total>=VALOR && (acum == VALOR || acum+obj[i+1] <= VALOR)
    if (acum==VALOR) for(int k=1; k<=n; k++) cout << include[k];
    else
    {  include[i+1] = "si";
        sum_of_subsets(i+1, acum+obj[i+1], total-obj[i+1]); ←
        include[i+1] = "no";
        sum_of_subsets(i+1, acum, total-obj[i+1]); } ←
    }
```

Sólo dos
llamadas recursivas,
pues se forma
un árbol binario

- Llamada inicial: *sum_of_subsets(0,0,T)*; donde **T** es la sumatoria de los valores de los objetos.

Otras aplicaciones



- Coloreado de grafos (Mapas).
 - Utilizando ' n ' colores, de qué manera se pueden colorear los vértices de un grafo sin que vértices adyacentes tengan el mismo color.
- Ciclo Hamiltoniano (problema del viajero sin optimización).
 - Ciclo en el grafo en el que TODOS los vértices del grafo se visitan sólo una vez.
- Problema de la mochila

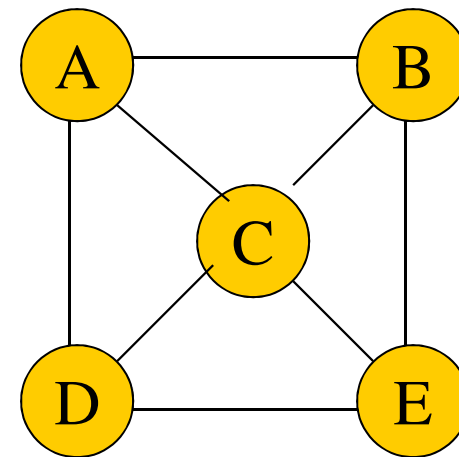
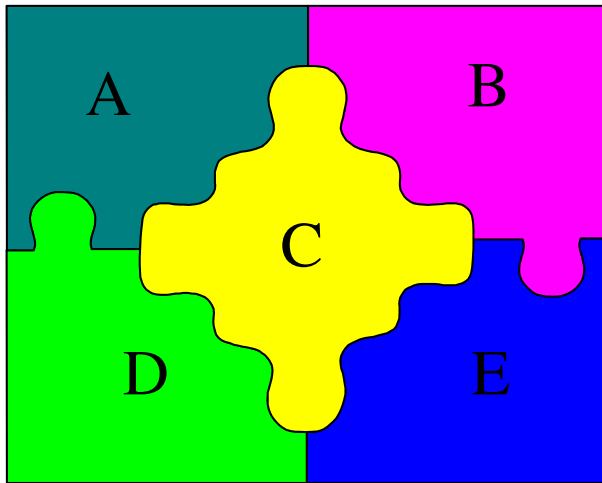
Coloreado de grafos



- Arbol de búsqueda de soluciones:
 - Cada nivel en el árbol, corresponde a un vértice del grafo.
 - Cada nodo del árbol, corresponde a un color con el que puede ser coloreado el vértice correspondiente a ese nivel.
- Criterio de selección:
 - Si el color asociado a ese vértice, no es igual al color asignado a los vértices adyacentes previamente analizados.

Coloreado de grafos

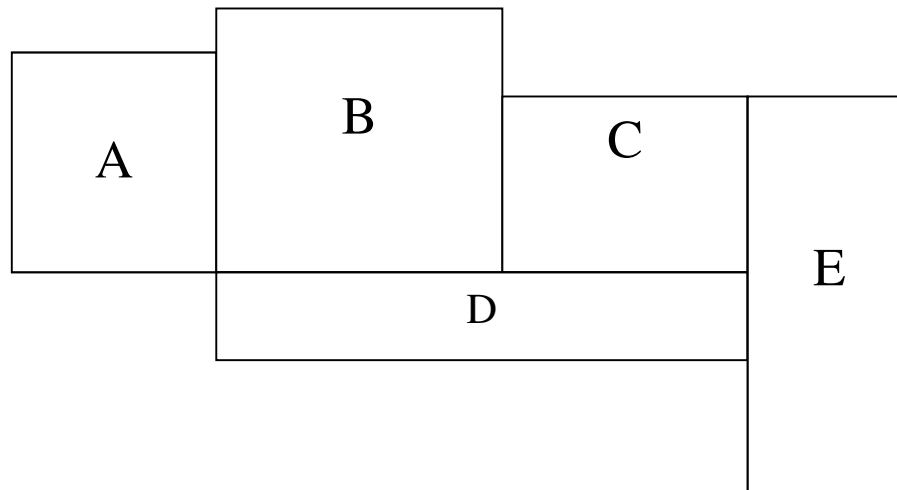
- Modelación de un mapa en un grafo...



- Arbol de 6 niveles, nodos con 'n' hijos, donde 'n' es la cantidad de colores con los que se quiere colorear...
- *Ver algoritmo específico en libro, sección 5.5...*

EJERCICIO

- El siguiente mapa se debe colorear con 3 colores, de tal manera, que cada elemento del mapa no tenga el mismo color que otro elemento adyacente. Modelar el mapa con un grafo, y aplicando la técnica de backtracking mostrar las soluciones al problema.



Ciclo Hamiltoniano



- Arbol de búsqueda de soluciones:
 - La raíz del árbol (nivel 0) es el vértice de inicio del ciclo.
 - En el nivel 1 se consideran TODOS los vértices menos el inicial.
 - En el nivel 2 se consideran TODOS los vértices menos los 2 que ya fueron visitados.
 - Y así sucesivamente hasta el nivel ' $n-1$ ' que incluirá al vértice que no ha sido visitado.

Ciclo Hamiltoniano



- Criterio de selección:
 - Un vértice en el nivel i del árbol, debe ser adyacente al vértice en el nivel $i-1$ del camino correspondiente en el árbol.
 - Un vértice en el nivel i , no puede ser alguno de los vértices de los $i-1$ niveles anteriores.
 - Un vértice en el nivel $n-1$ debe ser adyacente con el vértice del nivel 0 (raíz).
- Ver algoritmo específico en el libro, sección 5.6...

Eficiencia del Ciclo Hamiltoniano



- El problema del viajero fue resuelto con la técnica de la programación dinámica y se obtuvo un algoritmo con un comportamiento de orden exponencial
 $[T(n) = (n-1)(n-2)2^{n-3}]...$
- El algoritmo para encontrar los ciclos hamiltonianos por backtracking, en su peor caso, tiene un comportamiento peor que exponencial...
- Sin embargo, su utilidad radica en que al encontrar el primer ciclo, pudiera ser suficiente... (Ejemplo: visitar 20 ciudades vs. 40 ciudades).

EJERCICIO

- Para el grafo que se representa en la siguiente matriz de adyacencias, encontrar todos los ciclos hamiltonianos que tienen como origen el primer vértice del grafo, utilizando la técnica de backtracking.

$$\begin{pmatrix} 0 & 5 & \infty & 8 & \infty \\ 5 & 0 & 2 & 3 & 3 \\ \infty & 2 & 0 & 6 & 1 \\ 8 & 3 & 6 & 0 & 4 \\ \infty & 3 & 1 & 4 & 0 \end{pmatrix}$$

El problema de la mochila



- *Recordando*: Seleccionar el conjunto de objetos que maximice el valor que se puede guardar en la mochila sin exceder un peso específico que esta soporta.
- El problema ya fue resuelto con Programación dinámica, sin embargo, cumple las condiciones para ser resuelto con backtracking.
- A diferencia de los problemas anteriores, ESTE ES UN PROBLEMA DE OPTIMIZACIÓN, por lo que la forma de hacer el backtracking tendrá que considerarlo.

Algoritmo general de Backtracking con optimización

```
void verifica_nodo (Nodo r)  
{ Nodo h;  
    if (valor(r) es mejor que optimo)  
        optimo = valor(r);  
    if (el nodo r cumple criterio de selección)  
        for (cada hijo h de r)  
            verifica_nodo(h);  
}
```

En este caso, se explora TODO el árbol de búsqueda de soluciones, haciendo las podas correspondientes, y obteniendo al final, la solución óptima.

Problema de la mochila



- Arbol de búsqueda de soluciones:
 - Similar al del problema de "Sum-of-subsets"...
 - Cada nivel indica un objeto a incluir en la mochila...
 - Cada nodo del árbol tiene 2 hijos; uno que incluye al objeto y otro que no lo incluye...
- Criterio de selección:
 - Si el peso acumulado de los objetos incluidos no excede a la capacidad de la mochila...
 - Si el valor posible a acumular es mayor al mejor valor acumulado hasta ese momento...

Estimación del valor posible a acumular



- Si en un nodo del nivel i , se conoce cuál es el peso acumulado, y el valor acumulado...
- ¿cuántos objetos más podrían acumularse sin exceder el peso permitido, y cuál es el valor posible a acumular con estos objetos?
- Estratégicamente, conviene que los objetos estén ordenados de acuerdo a su valor proporcional de acuerdo a su peso ($\text{valor}_i / \text{peso}_i$)...
- Y que en orden descendente se vayan incluyendo en los niveles del árbol.

Estimación del valor posible a acumular

- Sea el nodo del nivel k el que hace que el peso acumulado exceda al peso permitido...
- El valor posible a acumular, se puede calcular de la siguiente forma:
 - Valor acumulado por los objetos ya incluidos, más...
 - Valor de los objetos $i+1$ hasta $k-1$, más...
 - Valor proporcional del objeto k por la fracción de peso que resta en la mochila...
- La fracción de peso que resta en la mochila se puede calcular:
 - Peso permitido en la mochila, menos...
 - Peso acumulado por los objetos ya incluidos, menos...
 - Peso de los objetos $i+1$ hasta $k-1$.

Ejemplo



- Para una mochila con capacidad de soportar 16 unidades de peso, se tienen los siguientes 4 objetos:
 - Objeto₁, valor₁: \$40, peso₁: 2, $\text{valor}_1/\text{peso}_1 = \20
 - Objeto₂, valor₂: \$30, peso₂: 5, $\text{valor}_2/\text{peso}_2 = \6
 - Objeto₃, valor₃: \$50, peso₃: 10, $\text{valor}_3/\text{peso}_3 = \5
 - Objeto₄, valor₄: \$10, peso₄: 5, $\text{valor}_4/\text{peso}_4 = \2

Ejemplo

PESO MOCHILA = 16

Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

Objeto₂, valor₂: \$30, peso₂: 5, valor₂/peso₂ = \$6

Objeto₃, valor₃: \$50, peso₃: 10, valor₃/peso₃ = \$5

Objeto₄, valor₄: \$10, peso₄: 5, valor₄/peso₄ = \$2

Vacum = \$0

Pacum = 0

Vposible = \$115

Valor óptimo = \$0

✓ **Pacum < 16**

✓ **Vposible > Valor óptimo**

Valor posible a acumular:

Se pueden acumular los objetos 1 y 2 sin exceder el peso.

$$\$40 + \$30 + (16 - 2 - 5) * \$50 / 10 = \$115$$

Ejemplo

PESO MOCHILA = 16

Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

Objeto₂, valor₂: \$30, peso₂: 5, valor₂/peso₂ = \$6

Objeto₃, valor₃: \$50, peso₃: 10, valor₃/peso₃ = \$5

Objeto₄, valor₄: \$10, peso₄: 5, valor₄/peso₄ = \$2

Incluir a obj1

Vacum = \$0
Pacum = 0
Vposible = \$115

Vacum = \$40
Pacum = 2
Vposible = \$115

Valor óptimo = \$40

✓ **Pacum < 16**

✓ **Vposible > Valor óptimo**

Valor posible a acumular:

Se pueden acumular el objeto 2 sin exceder el peso.

$$\$40 + \$30 + (16 - 2 - 5) * \$50 / 10 = \$115$$

Ejemplo

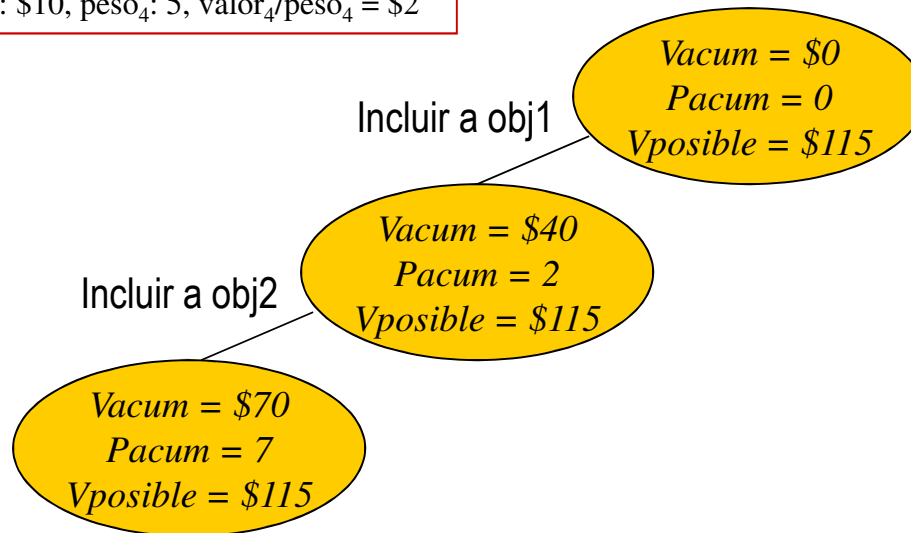
PESO MOCHILA = 16

Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

Objeto₂, valor₂: \$30, peso₂: 5, valor₂/peso₂ = \$6

Objeto₃, valor₃: \$50, peso₃: 10, valor₃/peso₃ = \$5

Objeto₄, valor₄: \$10, peso₄: 5, valor₄/peso₄ = \$2



Valor óptimo = \$70

✓ **Pacum < 16**

✓ **Vposible > Valor óptimo**

Valor posible a acumular:

NO se pueden acumular más objetos sin exceder el peso.

$$\$70 + (16 - 7) * \$50 / 10 = \$115$$

Ejemplo

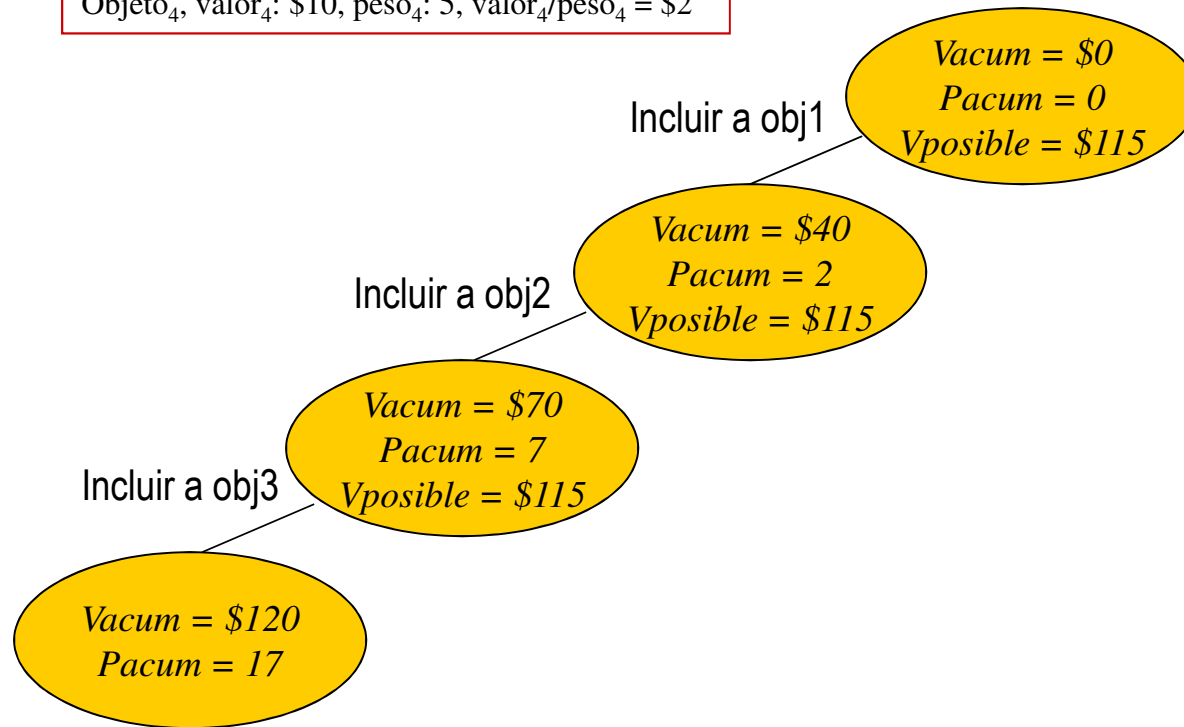
PESO MOCHILA = 16

Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

Objeto₂, valor₂: \$30, peso₂: 5, valor₂/peso₂ = \$6

Objeto₃, valor₃: \$50, peso₃: 10, valor₃/peso₃ = \$5

Objeto₄, valor₄: \$10, peso₄: 5, valor₄/peso₄ = \$2



Valor óptimo = \$70

× $Pacum < 16$

Valor posible a acumular:

NO hay necesidad de calcularlo...

Se aplica BACKTRACKING

Ejemplo

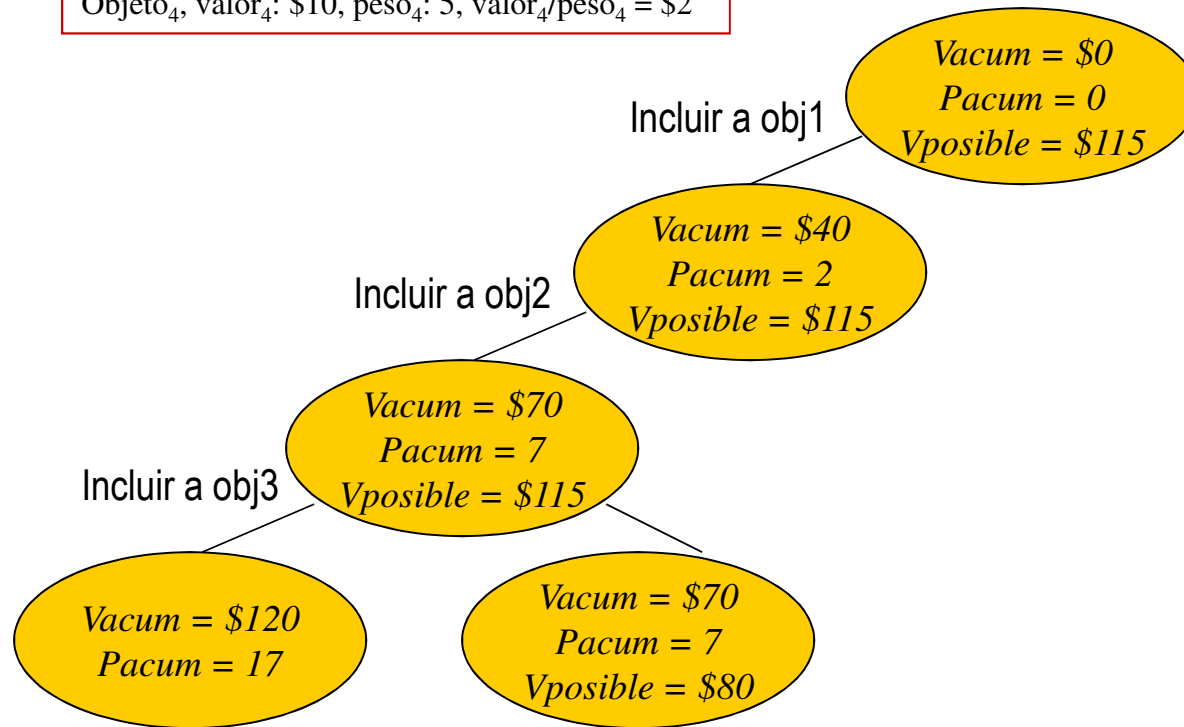
PESO MOCHILA = 16

Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

Objeto₂, valor₂: \$30, peso₂: 5, valor₂/peso₂ = \$6

Objeto₃, valor₃: \$50, peso₃: 10, valor₃/peso₃ = \$5

Objeto₄, valor₄: \$10, peso₄: 5, valor₄/peso₄ = \$2



Valor óptimo = \$70

✓ **Pacum < 16**

✓ **Vposible > Valor óptimo**

Valor posible a acumular:

Se puede incluir el objeto 4 sin exceder el peso

$$\$70 + \$10 + (16 - 7 - 5) * \$0 = \$80$$

Ejemplo

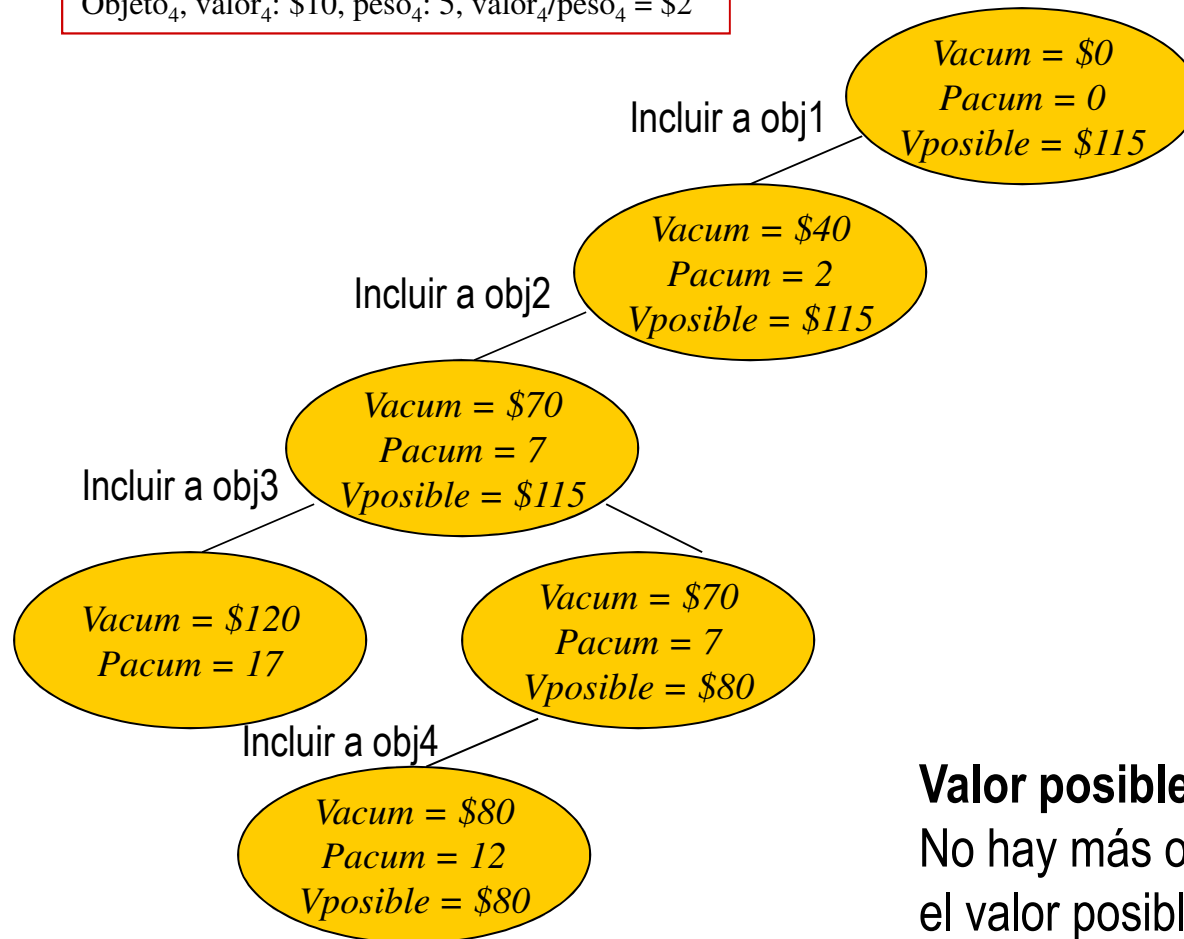
PESO MOCHILA = 16

Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

Objeto₂, valor₂: \$30, peso₂: 5, valor₂/peso₂ = \$6

Objeto₃, valor₃: \$50, peso₃: 10, valor₃/peso₃ = \$5

Objeto₄, valor₄: \$10, peso₄: 5, valor₄/peso₄ = \$2



Valor óptimo = \$80

✓ $Pacum < 16$
✗ $Vposible > \text{Valor óptimo}$

Valor posible a acumular:

No hay más objetos a incluir, por lo que coincide el valor posible con el valor acumulado.

Ejemplo

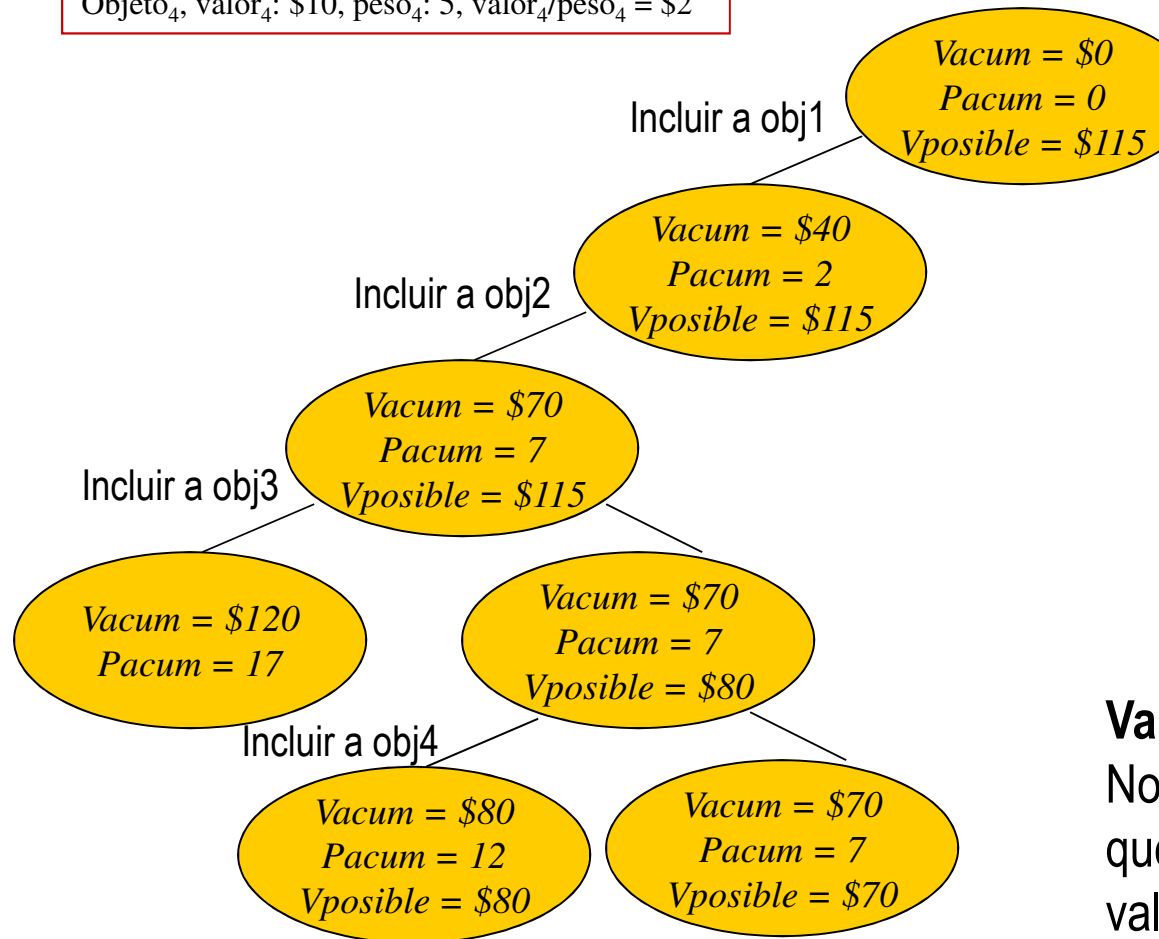
PESO MOCHILA = 16

Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

Objeto₂, valor₂: \$30, peso₂: 5, valor₂/peso₂ = \$6

Objeto₃, valor₃: \$50, peso₃: 10, valor₃/peso₃ = \$5

Objeto₄, valor₄: \$10, peso₄: 5, valor₄/peso₄ = \$2



Valor óptimo = \$80

✓ $Pacum < 16$
✗ $Vposible > \text{Valor óptimo}$

Valor posible a acumular:

No hay más objetos a incluir, por lo que coincide el valor posible con el valor acumulado.

Ejemplo

PESO MOCHILA = 16

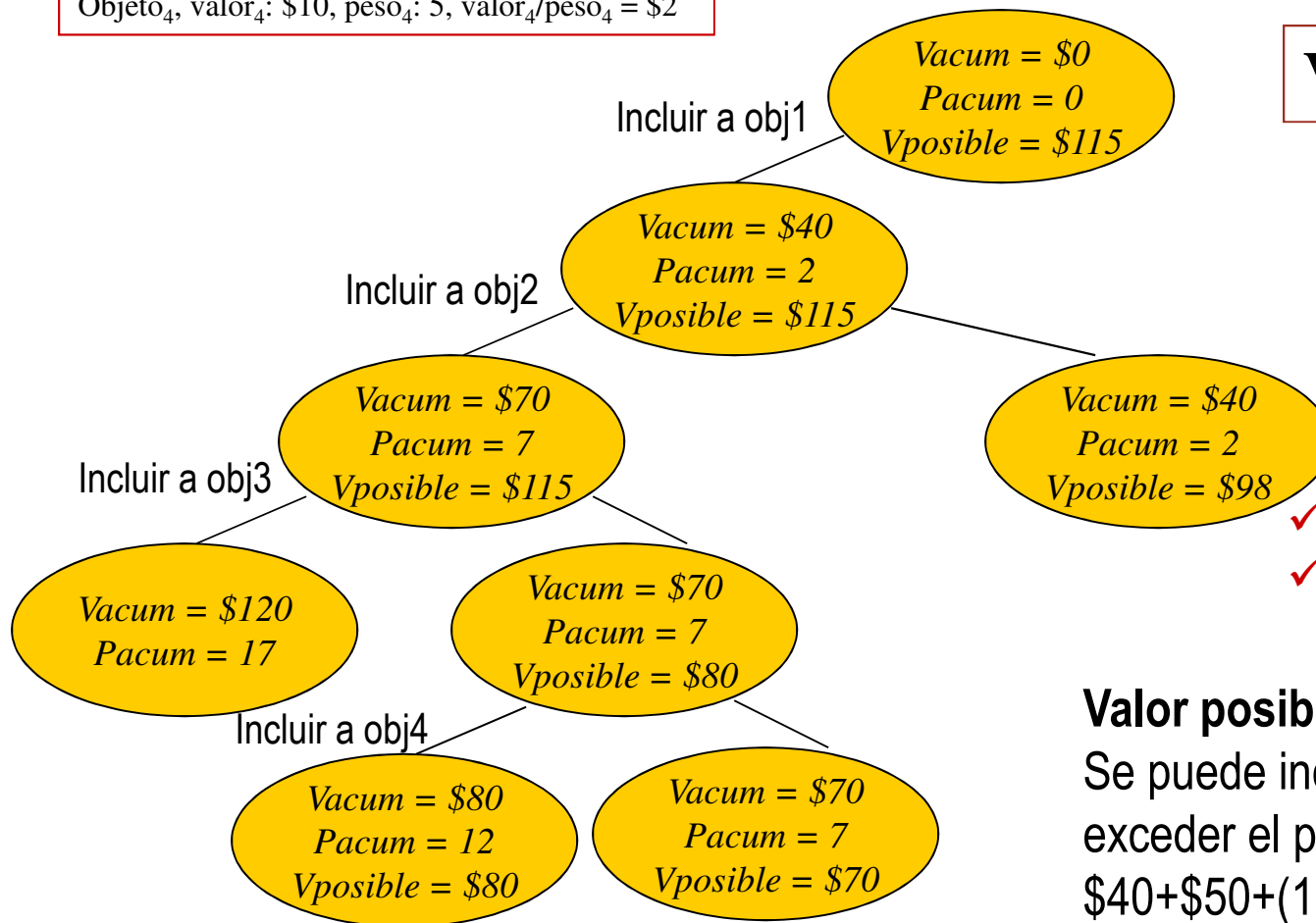
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

Objeto₂, valor₂: \$30, peso₂: 5, valor₂/peso₂ = \$6

Objeto₃, valor₃: \$50, peso₃: 10, valor₃/peso₃ = \$5

Objeto₄, valor₄: \$10, peso₄: 5, valor₄/peso₄ = \$2

Valor óptimo = \$80



✓ **Pacum < 16**
✓ **Vposible > Valor óptimo**

Valor posible a acumular:

Se puede incluir el objeto 3 sin exceder el peso

$$\$40 + \$50 + (16 - 2 - 10) * \$10 / 5 = \$98$$

Ejemplo

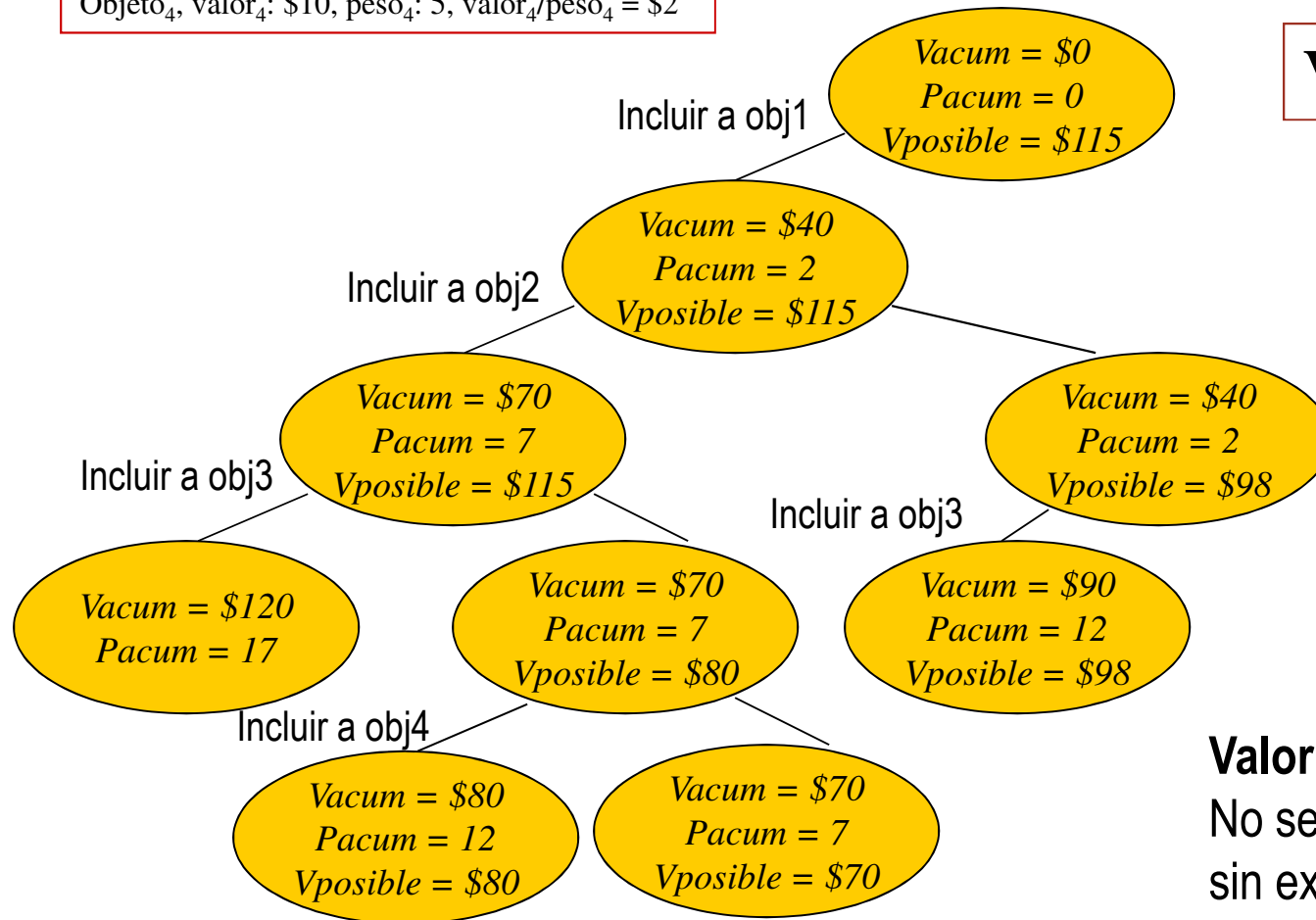
PESO MOCHILA = 16

Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

Objeto₂, valor₂: \$30, peso₂: 5, valor₂/peso₂ = \$6

Objeto₃, valor₃: \$50, peso₃: 10, valor₃/peso₃ = \$5

Objeto₄, valor₄: \$10, peso₄: 5, valor₄/peso₄ = \$2



Valor óptimo = \$90

✓ $Pacum < 16$

✓ $Vposible > \text{Valor óptimo}$

Valor posible a acumular:

No se pueden incluir más objetos sin exceder el peso.

$$\$90 + (16 - 12) * \$10 / 5 = \$98$$

Ejemplo

PESO MOCHILA = 16

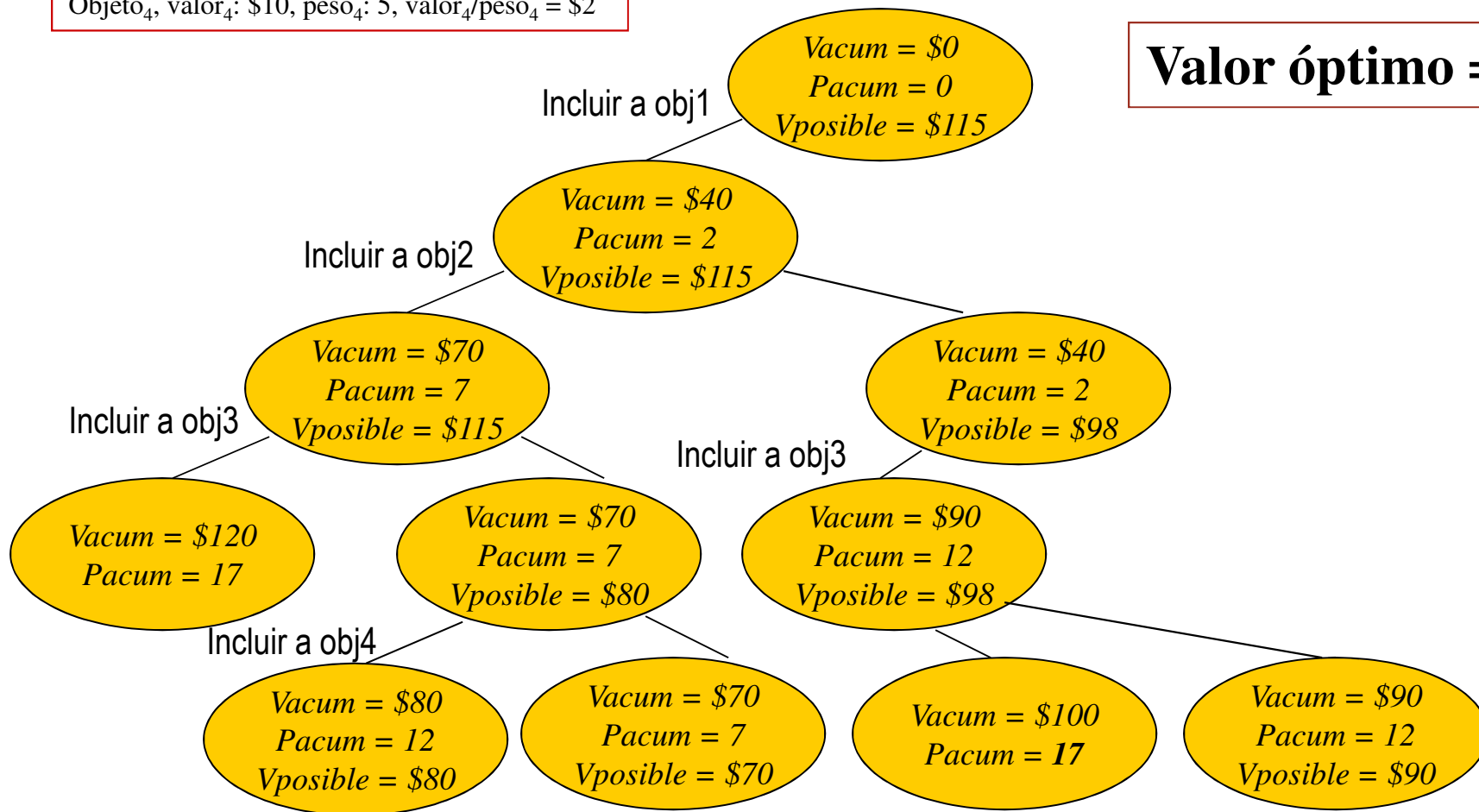
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

Objeto₂, valor₂: \$30, peso₂: 5, valor₂/peso₂ = \$6

Objeto₃, valor₃: \$50, peso₃: 10, valor₃/peso₃ = \$5

Objeto₄, valor₄: \$10, peso₄: 5, valor₄/peso₄ = \$2

Valor óptimo = \$90



Ejemplo

PESO MOCHILA = 16

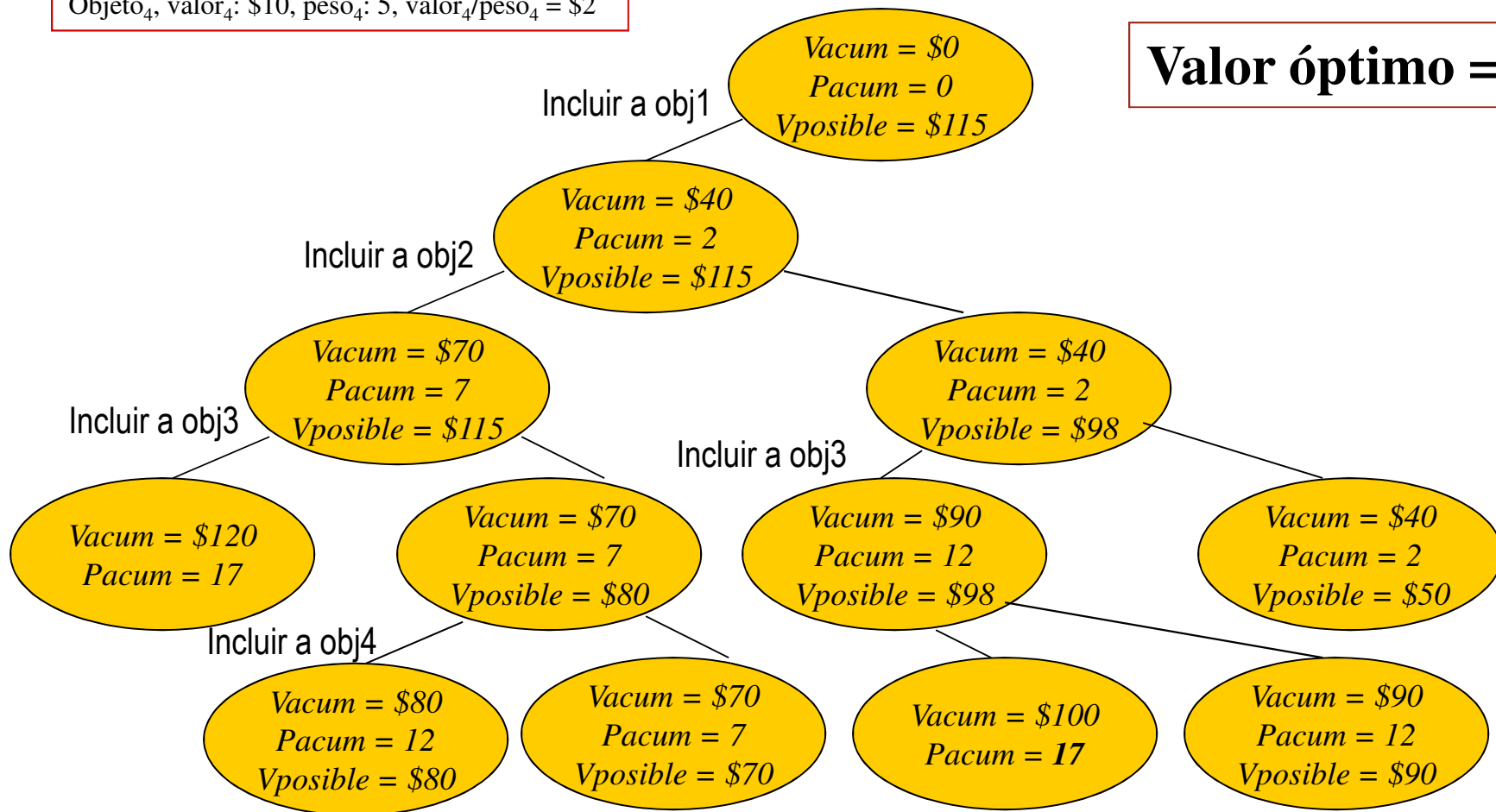
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

Objeto₂, valor₂: \$30, peso₂: 5, valor₂/peso₂ = \$6

Objeto₃, valor₃: \$50, peso₃: 10, valor₃/peso₃ = \$5

Objeto₄, valor₄: \$10, peso₄: 5, valor₄/peso₄ = \$2

Valor óptimo = \$90



Ejemplo

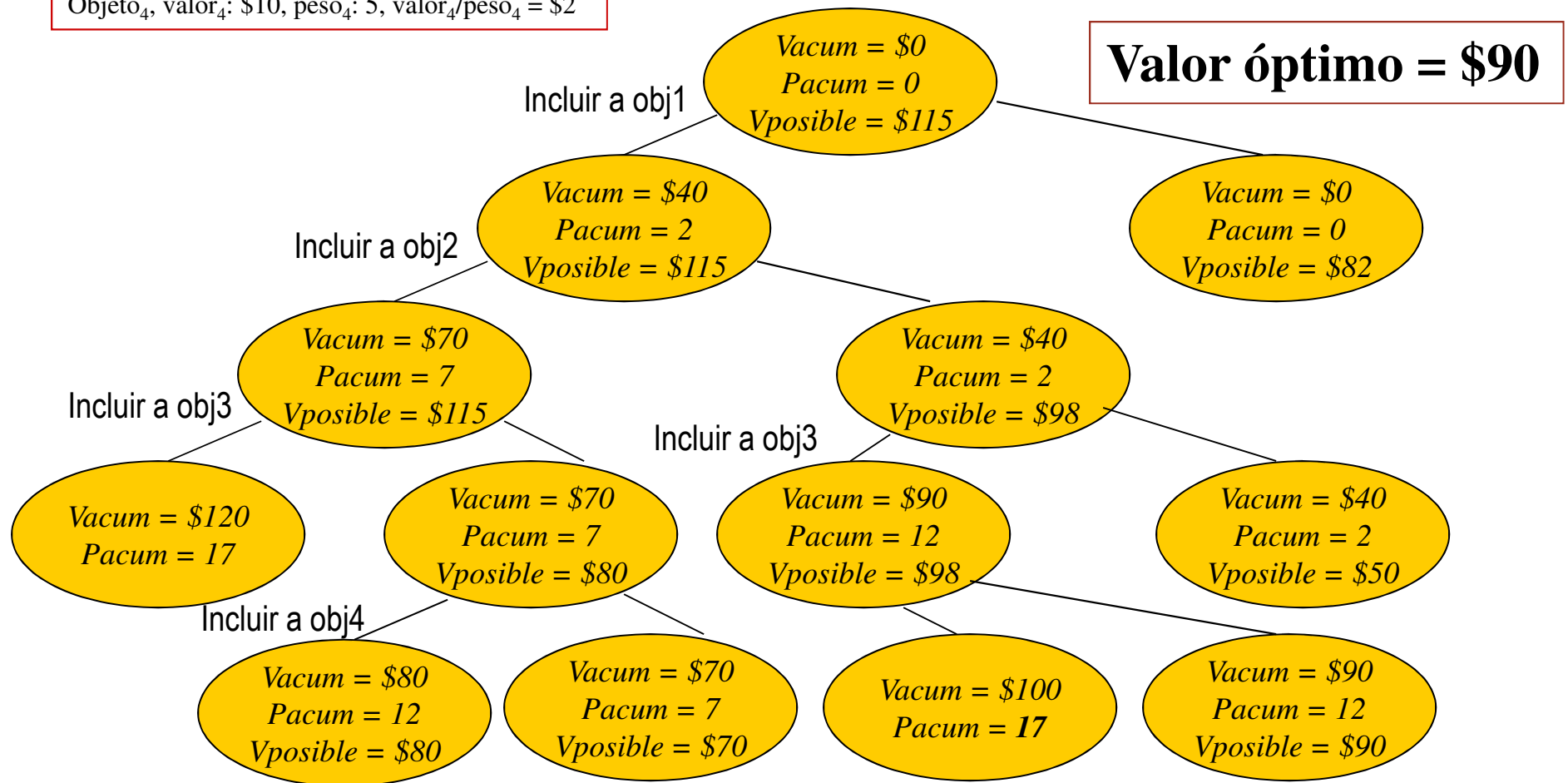
PESO MOCHILA = 16

Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

Objeto₂, valor₂: \$30, peso₂: 5, valor₂/peso₂ = \$6

Objeto₃, valor₃: \$50, peso₃: 10, valor₃/peso₃ = \$5

Objeto₄, valor₄: \$10, peso₄: 5, valor₄/peso₄ = \$2



Ejemplo

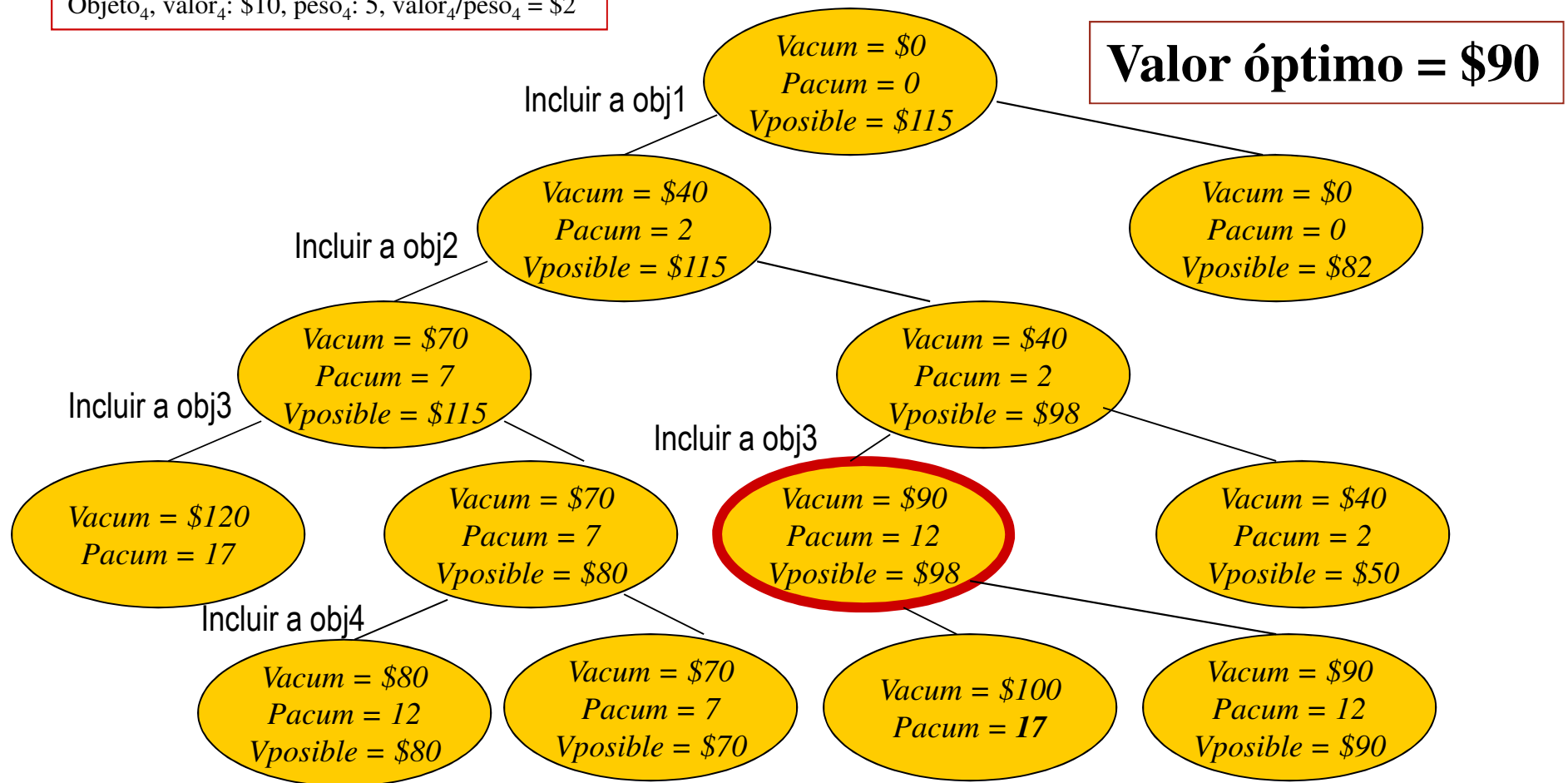
PESO MOCHILA = 16

Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

Objeto₂, valor₂: \$30, peso₂: 5, valor₂/peso₂ = \$6

Objeto₃, valor₃: \$50, peso₃: 10, valor₃/peso₃ = \$5

Objeto₄, valor₄: \$10, peso₄: 5, valor₄/peso₄ = \$2



Eficiencia en el problema de la mochila

- El algoritmo por Programación dinámica, tiene un comportamiento para el peor caso de **$O(\min(2^n, nP))$...**
- El algoritmo con Backtracking tiene un comportamiento para el peor caso de **$O(2^n)$...**
- Horowitz y Sahni en 1978, hicieron pruebas con ambos algoritmos, y en la práctica resultó mejor el algoritmo con backtracking...
- Ellos mismo propusieron una mejora utilizando divide&conquer con programación dinámica, obteniendo un algoritmo con **$O(2^{n/2})$...**