

Análisis y Diseño de Algoritmos



Introducción al
análisis de algoritmos

+ ¿Qué es un algoritmo?

- “(del árabe al-Khowârizmî, sobrenombre del célebre matemático árabe Mohámed ben Musa). Conjunto ordenado y finito de operaciones que permite encontrar la solución a un problema...”
- Un algoritmo, puede expresarse en términos de un lenguaje de programación, para obtener un programa que resuelve el problema por medio de la computadora.

+ ¿Cómo expresaremos los algoritmos en el curso?

- Utilizando lenguaje natural:
 - Puede haber ambigüedades...
 - El grado de especificación es subjetivo...
- Utilizando pseudocódigo en un lenguaje computacional:
 - No hay ambigüedades, pues se trabaja con estructuras de control: secuencia, decisiones, ciclos, recursividad.
 - El grado de especificación es estándar, aunque el diseño puede variar.
 - Facilidad de convertirlos en programas computacionales.

+ Cita...

- “No hay un incremento concebible en el poder de las computadoras que pueda saturar la demanda científica: aún pensando que una computadora posea un ciclo de tiempo subnuclear (10^{-23} seg.) y densidades de almacenamiento subnucleares (10^{39} bits/cm³), ésta no podría manejar la mayoría de los problemas que son importantes en la investigación científica básica y aplicada. Por lo tanto, existirá siempre una fuerte presión para incrementar la **eficiencia** de los programas, para poder incrementar también la cantidad de información útil generada por un programa.”

Ken Wilson, Nóbel de Física 1982

+ Áreas de estudio

- **¿Cómo construir algoritmos?**
 - Técnicas de diseño
- **¿Cómo expresar algoritmos?**
 - Enfoques de los lenguajes de programación
- **¿Cómo validar algoritmos?**
 - Verificación formal
- **¿Cómo analizar algoritmos?**
 - Complejidad computacional, eficiencia, legibilidad, usabilidad, etc...

+ Análisis de algoritmos

- Si se tuvieran 2 programas que hacen lo mismo, ¿cómo se podrían comparar?

1. Eficiencia:

- **Tiempo de ejecución**
- Uso de espacios de memoria

2. Facilidad de lectura, mantenimiento, rapidez para codificarlo.



Medición del tiempo de ejecución

7

■ El tiempo de ejecución depende de:

1. La entrada al programa:

Su tamaño

Sus características

2. La calidad del código generado para el programa por el compilador .

3. La rapidez de las instrucciones de máquina.

4. La **complejidad de tiempo del algoritmo**.

+ ¿Cómo medir?

- Cantidad de instrucciones básicas (o elementales) que se ejecutan.
- Ejemplos de instrucciones básicas:
 - asignación de escalares
 - lectura o escritura de escalares
 - saltos (goto's) implícitos o explícitos.
 - evaluación de condiciones
 - llamada a funciones
 - etc.

+ Ejemplo

9

cont = 1;

→ 1

while (*cont* ≤ *n*) {

→ *n* + 1

x = *x* + *a*[*cont*];

→ *n*

x = *x* + *b*[*cont*];

→ *n*

cont = *cont* + 1;

→ *n*

}

→ *n* (goto implícito)

→ 1 goto en falso.

TOTAL: **5*n* + 3**

+ Ejemplo

10

$z = 0;$

→ 1

for ($x = 1; x \leq n; x++$)

→ 1 asignación + (n+1) comparaciones

→ $(n+2)*n = n^2 + 2n$

for ($y = 1; y \leq n; y++$)

→ $n*n = n^2$

→ $2n^2$ (incremento + goto implícito)

$z = z + a[x,y];$

→ n (goto en falso for y)

→ $2n$ (incremento + goto implícito)

→ 1 (goto en falso for x)

TOTAL: $4n^2 + 6n + 4$

+ Consecuencia...

- Se requiere contar con una notación que permita comparar la eficiencia entre los algoritmos...
- La **NOTACIÓN ASINTÓTICA** es la propuesta de notación aceptada por la comunidad científica para describir el comportamiento en eficiencia (o complejidad) de un algoritmo.
- Describe en forma sintética el comportamiento de la función que con la variable de entrada, determina el número de operaciones que realiza el algoritmo.

+ NOTACIÓN ASINTÓTICA

- **COMPLEJIDAD TEMPORAL (y ESPACIAL)**. Tiempo (o espacio) requerido por un algoritmo, expresado en base a una función que depende del tamaño del problema.
- **COMPLEJIDAD TEMPORAL ASINTÓTICA (y ESPACIAL)**. Comportamiento límite conforme el tamaño del problema se incrementa. Determina el tamaño del problema que puede ser resuelto por un algoritmo.

+ Definición

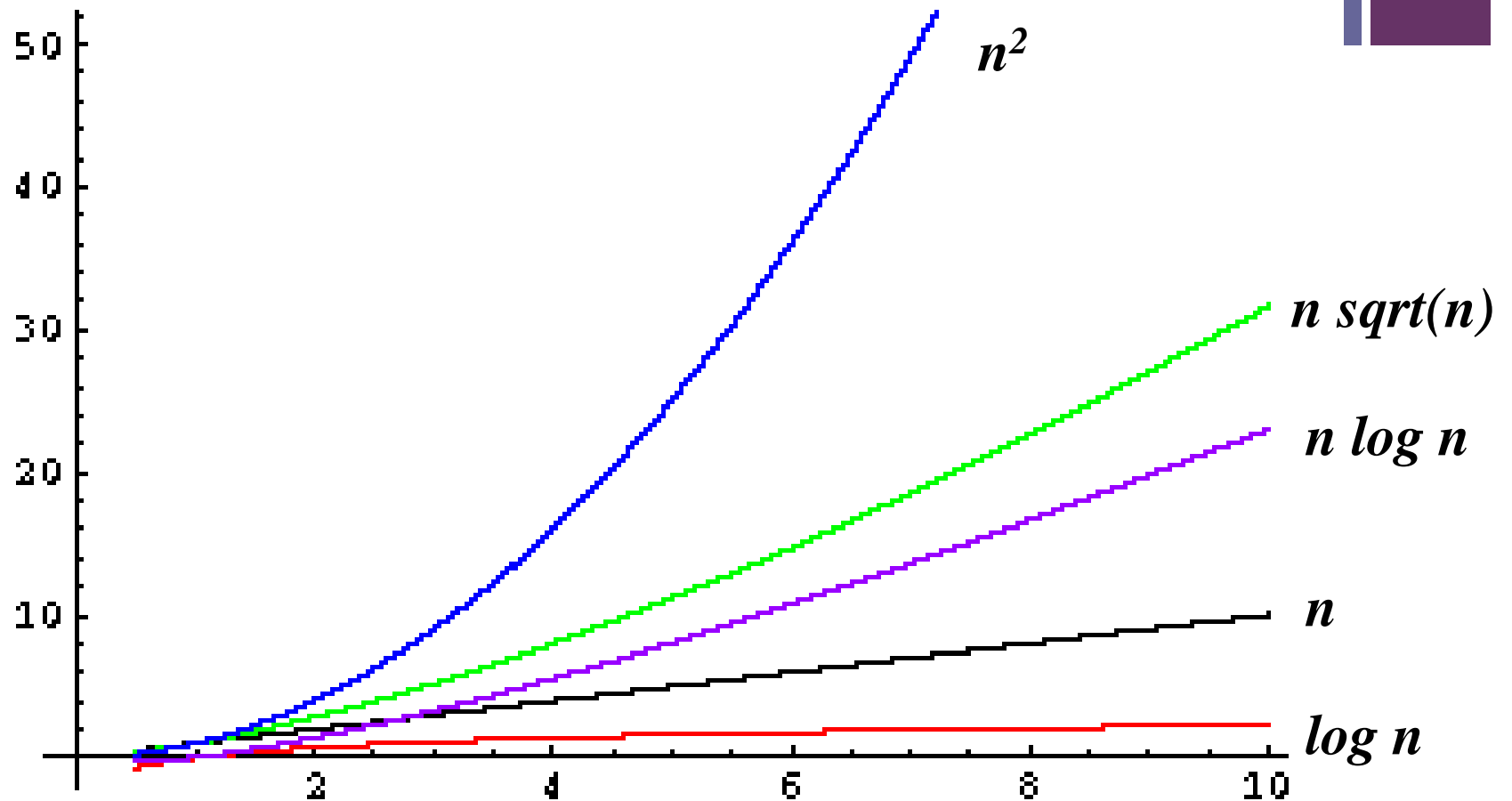
- Se dice que la función $f(n)$ “es de orden $g(n)$ ” [**$O(g(n))$**], si existen constantes positivas c y n_0 tales que $f(n) \leq c g(n)$ cuando $n \geq n_0$
- Ejemplos:
 - $n+5$ es **$O(n)$** pues $n+5 \leq 2n$ para toda $n \geq 5$
 - $(n+1)^2$ es **$O(n^2)$** pues $(n+1)^2 \leq 4n^2$ para $n \geq 1$
 - $(n+1)^2$ **NO** es **$O(n)$** pues para cualquier $c > 1$ no se cumple que $(n+1)^2 \leq c \cdot n$

+ Ordenes más comunes de los algoritmos

- $O(1)$ **Constante**
- $O(n)$ **Lineal**
- $O(n^2)$ **Cuadrático**
- $O(n^3)$ **Cúbico**
- $O(n^m)$ **Polinomial**
- $O(\log(n))$ **Logarítmico**
- $O(n \log(n))$ **$n \log(n)$**
- $O(m^n)$ **exponencial**
- $O(n!)$ **factorial**

+ Comportamiento de las funciones

15





Ejemplo del comportamiento

16

- Considerar que el algoritmo A1 es de orden **$O(n \log n)$** y que procesa 1,000 datos en 1 segundo.
- Su comportamiento al aumentar el tamaño de la entrada sería el siguiente:

$n = 2,000$	2.2 segundos
$n = 5,000$	6.2 segundos
$n = 10,000$	13.3 segundos
$n = 100,000$	2.77 minutos
$n = 1,000,000$	33.3 minutos
$n = 10,000,000$	6.48 horas



Ejemplo del comportamiento

17

- Considerar que el algoritmo A2 es de orden **$O(n^2)$** y que procesa 1,000 datos en 1 segundo.
- Su comportamiento al aumentar el tamaño de la entrada sería el siguiente:

$n = 2,000$	4 segundos
$n = 5,000$	25 segundos
$n = 10,000$	1.66 minutos
$n = 100,000$	2.77 horas
$n = 1,000,000$	11.5 días
$n = 10,000,000$	3.25 años



Ejemplo del comportamiento

18

- Si se contara con un algoritmo A3 de orden **exponencial** y que procesa 1,000 datos en 1 segundo...
- Su comportamiento al aumentar el tamaño de la entrada sería el siguiente:

n = 1,500	32 segundos
n = 2,000	17.1 minutos
n = 2,500	9.1 horas
n = 3,000	12.1 días
n = 3,500	1.09 años
n = 5,000	35,700 años
n = 10,000	4.02 X 10¹⁹ años !!!

+ ¿Cómo afecta la velocidad de la computadora?

- Ejemplo: Suponer que se tienen los siguientes algoritmos con la complejidad de tiempo correspondiente:

A1 con $T(n) = 100n$

A2 con $T(n) = 5n^2$

A3 con $T(n) = n^3/2$

A4 con $T(n) = 2^n$

- ¿Cuál es el orden de cada algoritmo?

+ ¿Cómo afecta la velocidad de la computadora?

- También suponer que se dispone de 1,000 segundos (aprox. 17 minutos) para ejecutar cada algoritmo...
- ¿Cuál es el tamaño máximo de la entrada que se puede procesar?

A1 con $T(n) = 100n$ **10**

A2 con $T(n) = 5n^2$ **14**

A3 con $T(n) = n^3/2$ **12**

A4 con $T(n) = 2^n$ **10**

+ ¿Cómo afecta la velocidad de la computadora?

- Suponer ahora que se cuenta con una computadora 10 veces más rápida... por lo tanto, es posible dedicar 10,000 segundos a la solución de problemas que antes se les dedicaba 1,000 segundos...
- ¿Cuál es el tamaño máximo de la entrada que se puede procesar?

A1 con $T(n) = 100n$

100

A2 con $T(n) = 5n^2$

45

A3 con $T(n) = n^3/2$

27

A4 con $T(n) = 2^n$

13

+ ¿Cómo afecta la velocidad de la computadora?

- ¿Cuál fue el aumento en la capacidad de procesamiento de cada algoritmo con el incremento de 10 veces en la velocidad de la computadora?
- ¿Cuál es el tamaño máximo de la entrada que se puede procesar?

A1 con $T(n) = 100n$	<i>10 --> 100</i>	<i>10</i>
A2 con $T(n) = 5n^2$	<i>14 --> 45</i>	<i>3.2</i>
A3 con $T(n) = n^3/2$	<i>12 --> 27</i>	<i>2.3</i>
A4 con $T(n) = 2^n$	<i>10 --> 13</i>	<i>1.3</i>

+ Ejemplo: ¿Influye la computadora?

- Para una Cray-1 utilizando Fortran, un algoritmo se procesa en $3n^3$ nanosegundos...
 - Para $n = 10$, se tardaría 3 microsegundos...
 - Para $n = 100$, se tardaría 3 milisegundos...
 - Para $n = 1,000$, se tardaría 3 segundos...
 - Para $n = 2,500$, se tardaría 50 segundos...
 - Para $n = 10,000$, se tardaría 49 minutos...
 - Para $n = 1,000,000$, se tardaría 95 años!!

+ Ejemplo: ¿Influye la computadora?

- Para una TRS-80 (computadora personal Tandy de los 80's) utilizando Basic, un algoritmo se procesa en $19,500,000n$ nanosegundos...
- Para $n = 10$, se tardaría .2 segundos...
- Para $n = 100$, se tardaría 2 segundos...
- Para $n = 1,000$, se tardaría 20 segundos...
- Para $n = 2,500$, se tardaría 50 segundos...
- Para $n = 10,000$, se tardaría 3.2 minutos...
- Para $n = 1,000,000$, se tardaría 5.4 horas...

+ Diseño de algoritmos: viéndolo en acción

- Encontrar el n-ésimo elemento de la serie de Fibonacci....
- Algoritmos de solución:
 - Iterativo (técnica de la programación dinámica).
 - Recursivo (técnica de “divide y vencerás”).
- Complejidad de los algoritmos (análisis):
 - Iterativo: **$O(n)$**
 - Recursivo: **$O(2^{n/2})$**

+ Caso Fibonacci...

- Supongamos una máquina que procese una operación básica en 1 nanosegundo...
- Para encontrar el elemento #80 de la serie se tardarían:
 - Iterativo.....: 81 nanosegundos.
 - Recursivo.....: 18 minutos.
- Para encontrar el elemento #121 se tardarían:
 - Iterativo.....: 121 nanosegundos.
 - Recursivo.....: **36 años!!!!**.
- ¿Importa el diseño del algoritmo?

+ Algoritmo Fibonacci

27

```
#include <iostream.h>

using namespace std;

long int fibo1 (int n)

{ long int ant = 1, act = 1, aux;

  for (int i=3; i <= n; i++)

  { aux = ant + act;

    ant = act;

    act = aux; }

  return act;

}

long int fibo2 (int n)

{ if (n <3)

  return 1;

  else

    return (fibo2(n-1) + fibo2(n-2));

}
```

+ Conclusión

- “A medida de que los computadores aumenten su rapidez y disminuyan su precio, como con toda seguridad seguirá sucediendo, también el deseo de resolver problemas más grandes y complejos seguirá creciendo. Así la importancia del descubrimiento y el empleo de algoritmos eficientes irá en aumento, en lugar de disminuir...”

Aho, Hopcroft, Ullman, 1983

+ Por lo tanto...

- Es importante aprender a **analizar** algoritmos...
- Es importante conocer técnicas para **diseñar** algoritmos eficientes...
- Y así, ante un problema, tener capacidad de decidir y aplicar el mejor algoritmo de solución...

+ En resumen...

- ¿Cómo analizar la complejidad de tiempo de un algoritmo?
 - Contando la cantidad de operaciones básicas
 - Las que más se repiten
 - Convirtiendo a Notación Asintótica
 - Tomando el término más representativo

+ Ejemplo: Sumar los datos de un arreglo

```
suma = 0;  
for (int i=1; i<=n; i++)  
    suma += arreglo[ i ];
```

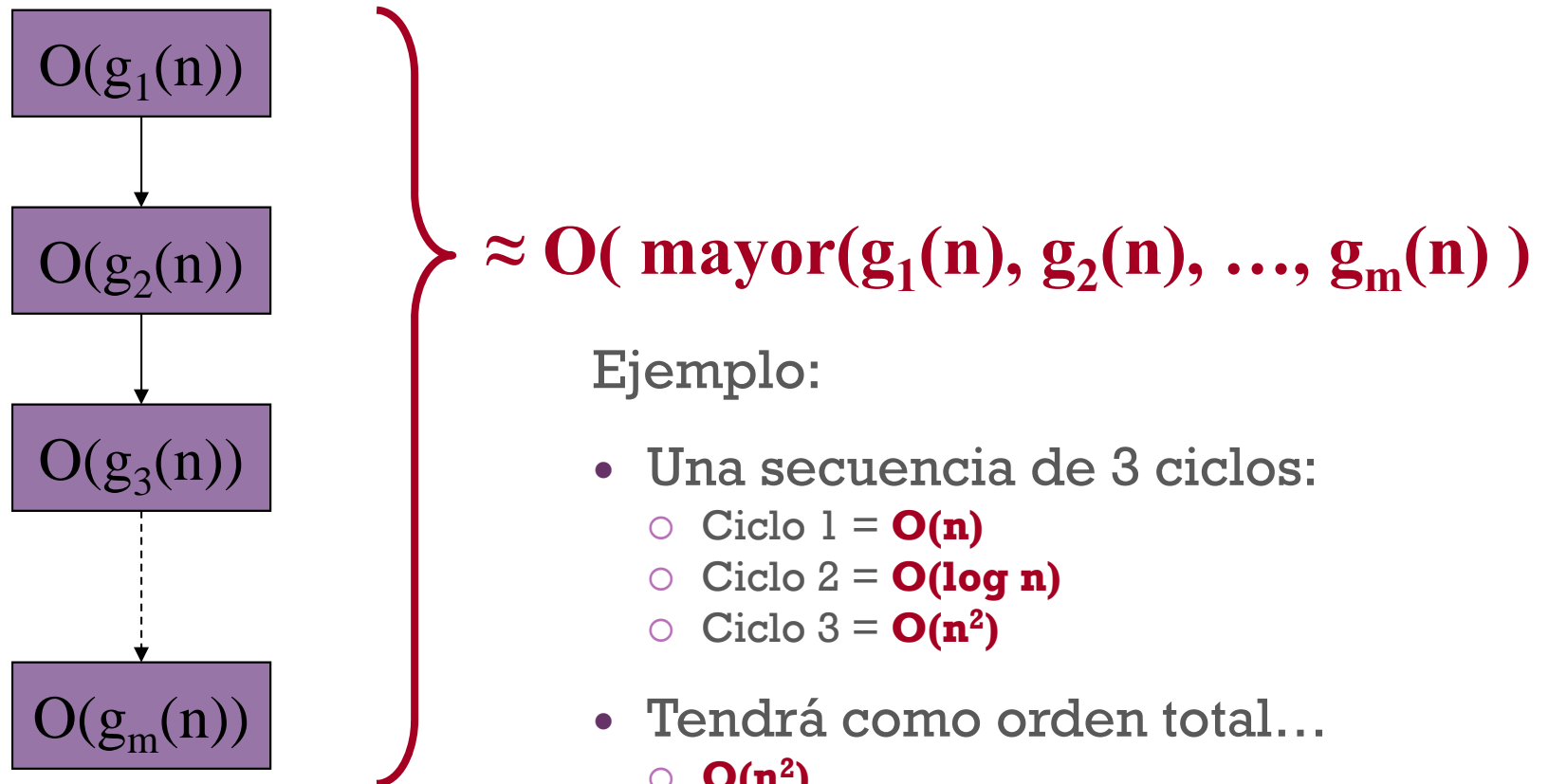
¿Cómo obtener la complejidad de tiempo del algoritmo?

- Operación básica: *suma = suma + arreglo[i]*;
- Se realiza n veces
- Algoritmo de orden n : **$O(n)$**

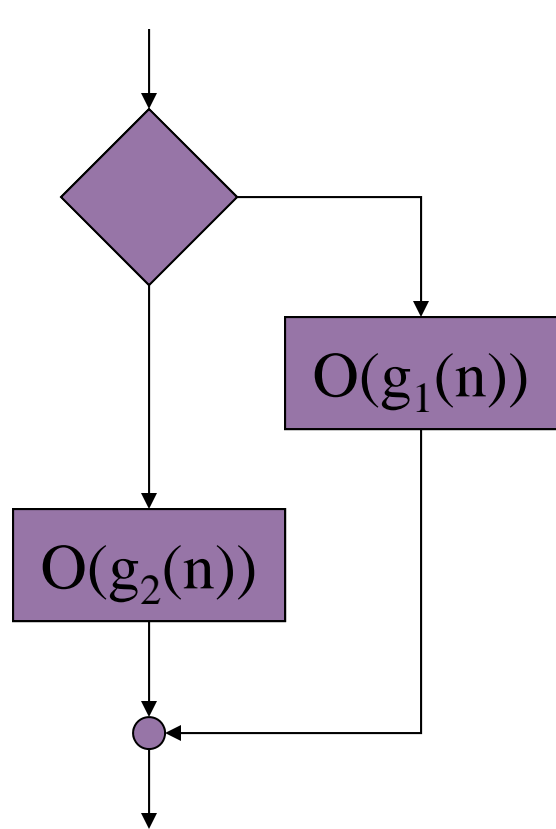
+ Otro método para calcular la complejidad

- Consiste en aplicar reglas a los estatutos estructurados:
 1. Secuencia de instrucciones
 2. Decisiones (ejemplo: if)
 3. Ciclos (ejemplo: while)
 4. Recursividad

+ Regla 1: Secuencia de instrucciones



+ Regla 2: Decisiones

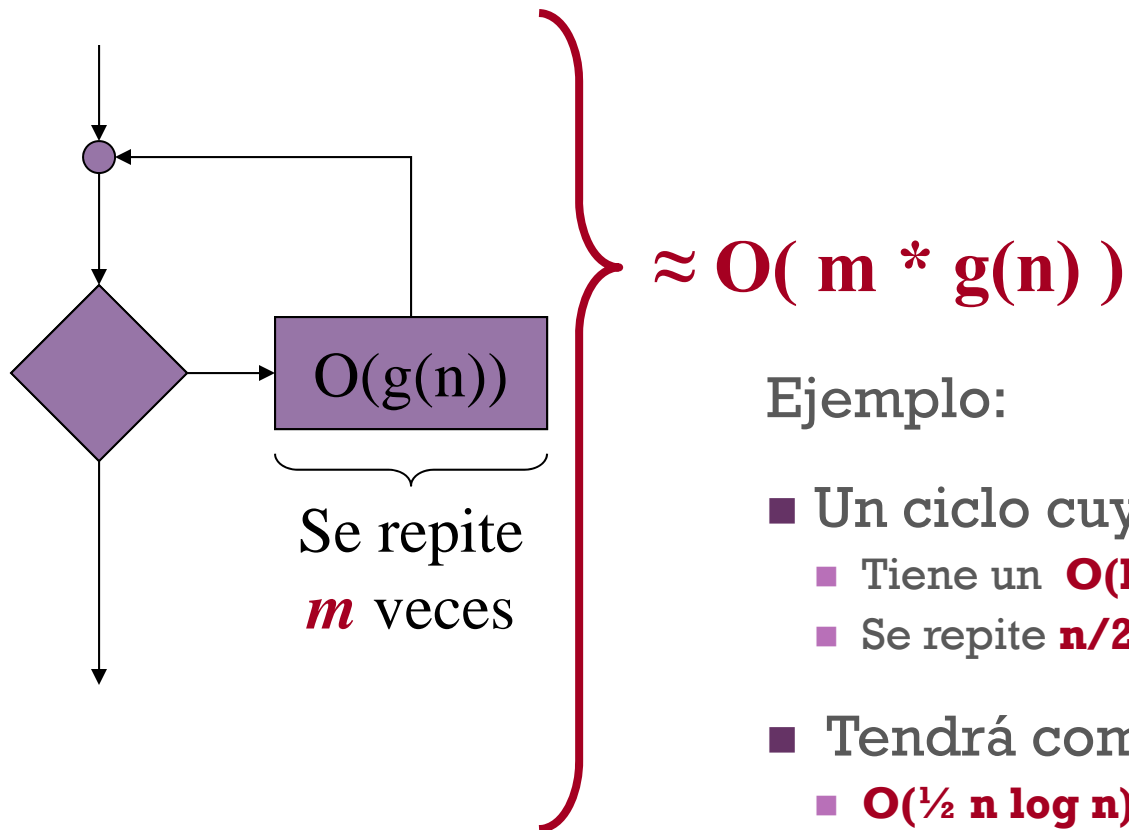


$\approx O(\text{mayor}(g_1(n), g_2(n)))$

Ejemplo:

- Una decisión con:
 - Rama then = $O(n \log n)$
 - Rama else = $O(\log n)$
- Tendrá como orden total...
 - $O(n \log n)$.

+ Regla 3: Ciclos



Ejemplo:

- Un ciclo cuya instrucción:
 - Tiene un $O(\log n)$
 - Se repite $n/2$ veces
- Tendrá como orden total...
 - $O(\frac{1}{2} n \log n) = O(n \log n)$.

+ Consideraciones especiales

- En decisiones y ciclos anidados:
 - Analizar el código desde la instrucción más interna hacia el más externa.
- Tip para los ciclos:
 - ¿“Normalmente” cuál es el orden de la instrucción interna?
 - Si la variable de control se incrementa o decrementa con un valor constante: **Orden LINEAL**.
 - Si la variable de control se multiplica o divide por un valor constante: **Orden LOGARÍTIMICO**.

+ Ejemplo: Sort por intercambio

- Toma la primera posición del arreglo, y compara su contenido contra el resto de los valores del arreglo. Cada vez que se encuentra un elemento menor al de la posición, lo intercambia. Esto asegura que el dato que queda en la primera posición está ordenado.
- El proceso se repite con la segunda posición, la tercera y así sucesivamente hasta la penúltima posición, siempre comparando solamente contra los elementos desordenados.

+ Ejemplo: Sort por intercambio

- 8 3 5 9 2 7 4
- 3 8 5 9 2 7 4
- **2** 8 5 9 3 7 4
- **2** 8 5 9 3 7 4
- **2** 5 8 9 3 7 4
- **2 3** 8 9 5 7 4
- **2 3** 8 9 5 7 4
- **2 3** 5 9 8 7 4
- **2 3 4** 9 8 7 5

- **2 3 4** **9** 8 7 5
- **2 3 4** **8** 9 7 5
- **2 3 4** **7** 9 8 5
- **2 3 4** **5** 9 8 7
- **2 3 4** **5** **9** 8 7
- **2 3 4** **5** **8** 9 7
- **2 3 4** **5** **7** 9 8
- **2 3 4** **5** **7** **9** 8
- **2 3 4** **5** **7** **8** 9

+ Ejemplo: Sort por intercambio

for (i=1; i<n; i++)

for (int j=i+1; j<=n; j++)

if (a[j] < a[i])

$\rightarrow \mathbf{O(1)}$ $\left. \vphantom{\begin{matrix} \rightarrow \mathbf{O(1)} \\ \rightarrow \mathbf{O(1)} \end{matrix}} \right\} \rightarrow \mathbf{O(1)}$

intercambia(a[i], a[j]);

Regla 2: Decisiones = mayor de las 2 ramas

+ Ejemplo: Sort por intercambio

for (i=1; i<n; i++) ← Peor caso: se repite n-1 veces

for (int j=i+1; j<=n; j++) } → **O(1)** } → **O(n)**

if (a[j] < a[i])

intercambia(a[i], a[j]);

Regla 3: Ciclos = # veces * orden de la instrucción interna

+ Ejemplo: Sort por intercambio

Se repite n-1 veces

```
for (i=1; i<n; i++)  
    for (int j=i+1; j<=n; j++)  
        if (a[j] < a[i])  
            intercambia(a[i], a[j]);
```

$O(n)$ $O(n^2)$

Regla 3: Ciclos = # veces * orden de la instrucción interna



Ejemplo: Multiplicación de matrices

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{nm} \end{pmatrix} =$$

$$\begin{pmatrix} c_{11} & c_{12} & \dots & c_{1m} \\ c_{21} & c_{22} & \dots & c_{2m} \\ \dots & \dots & \dots & \dots \\ c_{m1} & c_{m2} & \dots & c_{mm} \end{pmatrix}$$

$$c_{11} = a_{11} * b_{11} + a_{12} * b_{21} + \dots + a_{1n} * b_{n1}$$

$$c_{12} = a_{11} * b_{12} + a_{12} * b_{22} + \dots + a_{1n} * b_{n2}$$

...

$$c_{21} = a_{21} * b_{11} + a_{22} * b_{21} + \dots + a_{2n} * b_{n1}$$

...

$$c_{mm} = a_{m1} * b_{1m} + a_{m2} * b_{2m} + \dots + a_{mn} * b_{nm}$$

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

+ Ejemplo: Multiplicación de matrices

$O(n^3) \leftarrow$ *for (int i=1; i<=n; i++)*

$O(n^2) \leftarrow$ *for (int j=1; j<=n; j++)*

~~$O(1) \leftarrow$~~ *{ C[i,j] = 0;*

$O(n) \leftarrow$ *for (int k=1; k<=n; k++)*

$O(1) \leftarrow$ *C[i,j] = C[i,j] + A[i,k]*B[k,j];*

}

+ Regla 4: Recursividad

- La complejidad de tiempo se obtiene contando la cantidad de veces que se hace la llamada recursiva.
- Casos que “normalmente” se dan:
 - Orden LINEAL si sólo se tiene una llamada recursiva, con incrementos o decrementos en el parámetro de control.
 - Orden LOGARITMICO si sólo se tiene una llamada recursiva, con multiplicaciones o divisiones en el parámetro de control.
 - Si hay más de una llamada recursiva, el orden puede tender a ser EXPONENCIAL.

+ Ejemplo: Fibonacci (Iterativo)

```

ant = 1;           --> 1
act = 1;           --> 1
while (n>2){       --> n-2 + 1
    aux = ant + act;    --> n-2
    ant = act;          --> n-2
    act = aux;          --> n-2
    n = n - 1;         --> n-2
}                  --> n-2+1
printf(“%d”,act); --> 1

```

$$T(n) = 6n - 7$$

*Por lo tanto el orden
del algoritmo es*

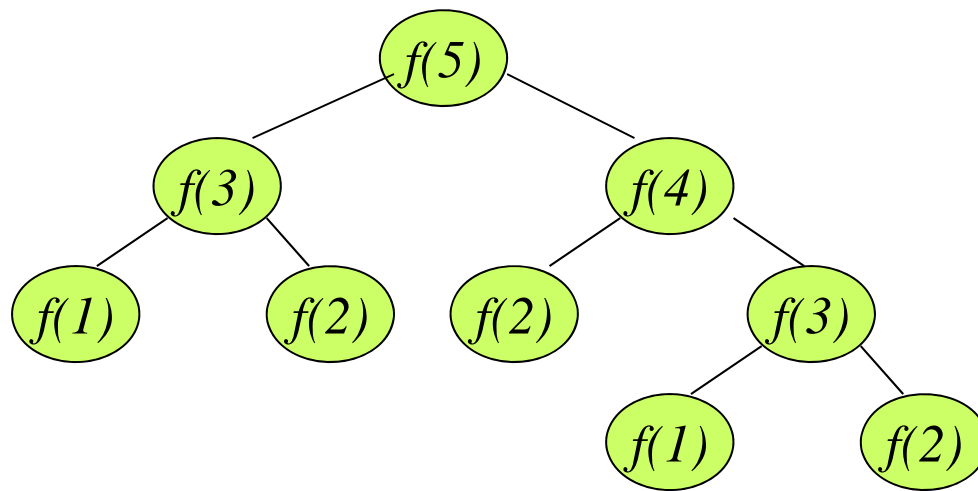
$$O(n)$$

+ Ejemplo: Fibonacci (recursivo)

```
Function fibonacci (n:int): int;  
  if (n < 3) return 1;  
  else return fibonacci(n-1) + fibonacci(n-2);
```

- ¿Cómo obtener la complejidad de tiempo del algoritmo?
- Cantidad de llamadas recursivas: *2 en cada llamada.*
- Algoritmo de orden: **$O(2^{n/2})$**

+ Análisis de Fibonacci (recursivo)



¿Cuántos términos se requieren para calcular:

$f(5)?$

$f(4)?$ --> 9

$f(3)?$ --> 5

$f(2)?$ --> 3

--> 1

$f(6)?$ --> 15

Relación:

El término $T(n)$ requiere $T(n-1)+T(n-2)+1$ términos para calcularse.

+ Análisis de Fibonacci

- Si el término **$T(n)$** requiere **$T(n-1)+T(n-2)+1$** términos para calcularse...
- se puede decir que **$T(n) > 2 * T(n-2)$** ...
- y por lo tanto: **$T(n) > 2 * 2 * T(n-4)$** ...
- y **$T(n) > 2 * 2 * 2 * T(n-6)$** ...
- y así sucesivamente hasta:

$$T(n) > 2 * 2 * 2 * \dots * 2 * T(1)$$

$\underbrace{\hspace{10em}}$
 $n/2$ veces

Por lo tanto:
 $T(n) > 2^{n/2}$
y podemos decir
que el orden del
algoritmo es
 $O(2^{n/2})$

+ Complejidades variables

- Hasta ahora nuestro método funciona sin problemas si la complejidad es igual (constante) para cualquier contenido de la entrada.
 - Ejemplos: Sumar los elementos de un arreglo, Multiplicar 2 matrices, etc.
- ¿Qué hacer si la complejidad varía dependiendo del contenido de la entrada?
 - Ejemplos: Búsqueda secuencial, binaria, etc.
- Debemos utilizar:
 - La complejidad del algoritmo para el peor caso
 - En algunos algoritmos posibles, el caso promedio.

+ Ejemplo: Búsqueda secuencial

```
pos = 1;  
while(pos ≤ n) and (arreglo[pos] < dato) do  
    pos = pos + 1;  
if (pos > n) or (arreglo[pos] <> dato) then  
    pos = 0;
```

- Mejor caso: **1**
- Peor caso: ***n***
- Caso promedio?
 - Depende de probabilidades: **$3n/4 + 1/4$**

Por lo tanto:
 $O(n)$

+ Ejemplo: Búsqueda binaria

inicio = 1; fin = n; pos = 0;

while (inicio ≤ fin) and (pos == 0) do

mitad = (inicio + fin) div 2;

if (x == arreglo[mitad]) then

pos = mitad;

else if (x < arreglo[mitad]) then

fin = mitad - 1

else

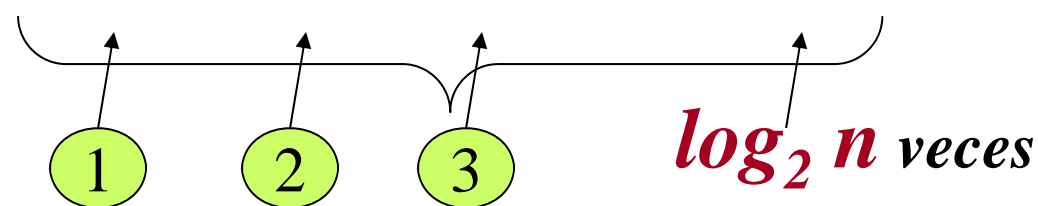
inicio = mitad + 1;

■ Operación Básica: $x == \text{arreglo}[\text{mitad}]$

■ Mejor caso: **1**

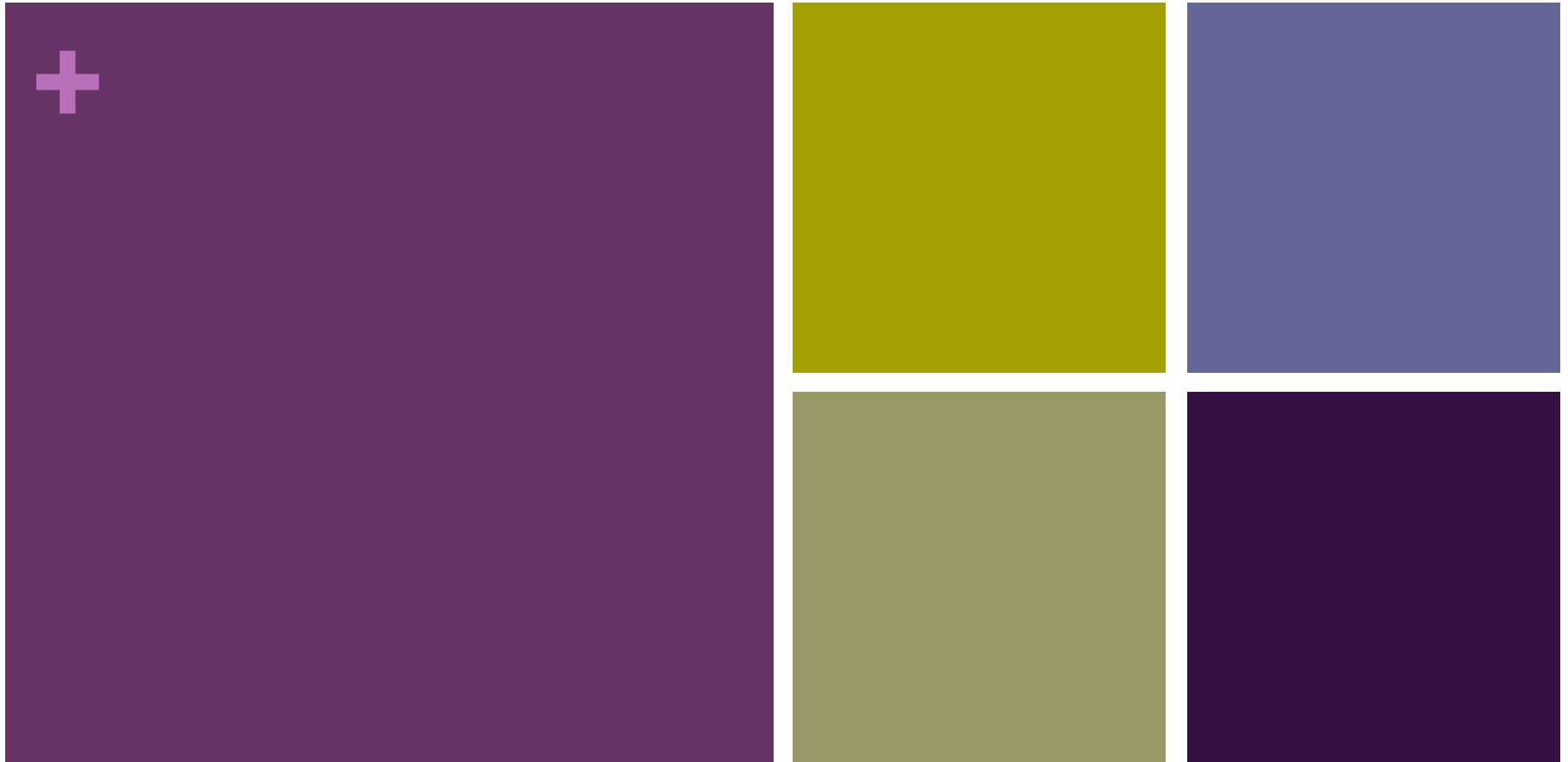
+ Ejemplo: Búsqueda binaria

- Peor caso: No encontrar el dato
- Suponiendo que n es potencia de 2:

$$n/2 + n/4 + n/8 + \dots + n/n$$


$\log_2 n$ veces

- Caso Promedio: Un análisis detallado lleva a encontrar la complejidad de: $\lfloor \log_2 n \rfloor \pm 1/2$
- Por lo tanto, el orden del algoritmo es: **$O(\log n)$**



Ejercicios

+ Varios órdenes

- Si un algoritmo tiene como función de complejidad la siguiente

- $T(n) = 3n^{0.5} + 2 \log(n^2) + 1000n - 1/n$

- ¿Cuál es su orden?

- $O(n)$

+ Ciclos

```
i = n*2;
while (i>2)
{   instruccion1;
    for (j=0; j<n/2; j++)
        instruccion2;
    for (k=n; k>1; k--)
        for (m=1; m<10; m++)
            intruccion3;

    i = i / 3;
}
```

+ Ciclos

```
i = 1;
while (i < n)
{
    instruccion1;
    for (j = 1; j < n; j = j * 2)
    {
        instruccion2;
        for (k = n; k > 1; k = k / 3)
            instruccion3;
    }
    i = i * 3;
}
```


+ Búsqueda Binaria Recursiva

```
int buscarNum (int numero, int inicio, int final)
{
    if (inicio>final)
        return 0;
    else
    {
        int centro=(inicio+final)/2;
        if (numero==arreglo[centro])
            return centro;
        else if (numero<arreglo[centro])
            return buscarNum (numero, inicio, centro-1);
        else
            return buscarNum (numero, centro+1, final);
    }
}
```

+ Búsqueda en un ABB

```
nodo* buscarNodo (int numero, nodo* raiz)
{
    if (raiz == NULL)
        return NULL;
    else if (numero == raiz->info)
        return raiz;
    else if (numero < raiz->info)
        return buscarNodo (numero, raiz->izq);
    else
        return buscarNodo (numero, raiz->der);
}
```

Análisis de Algoritmos

Divide y vencerás
(Divide-and-Conquer)

+ Divide y vencerás

- Técnica para enfrentar la solución de problemas.
- Consiste en dividir un problema en 2 o más instancias del problema más pequeñas, resolver (conquistar) esas instancias, y obtener la solución general del problema, combinando las soluciones de los problemas más pequeños.
- Utiliza un enfoque de solución de tipo ***top-down***.
- Corresponde a una solución natural de tipo **recursivo**.

+ Casos que aplican” Divide y vencerás”

- ⊗ Búsqueda binaria
- ✓ Merge Sort
- ✓ Quick Sort
- ✓ Algoritmo de Strassen’s para multiplicar matrices
- ✓ Aritmética de enteros grandes

+ Búsqueda binaria

- Enfoque con la técnica de “divide y vencerás” : SOLUCIÓN RECURSIVA.

```
int busca (inicio, fin: index){  
  if (inicio > fin) return 0;  
  else  
    mitad = (inicio + fin) / 2;  
    if (x == arreglo[mitad]) return mitad;  
    else if (x < arreglo[mitad]) return(busca(inicio,  
      mitad-1));  
    else return(busca(mitad+1,fin));  
}
```

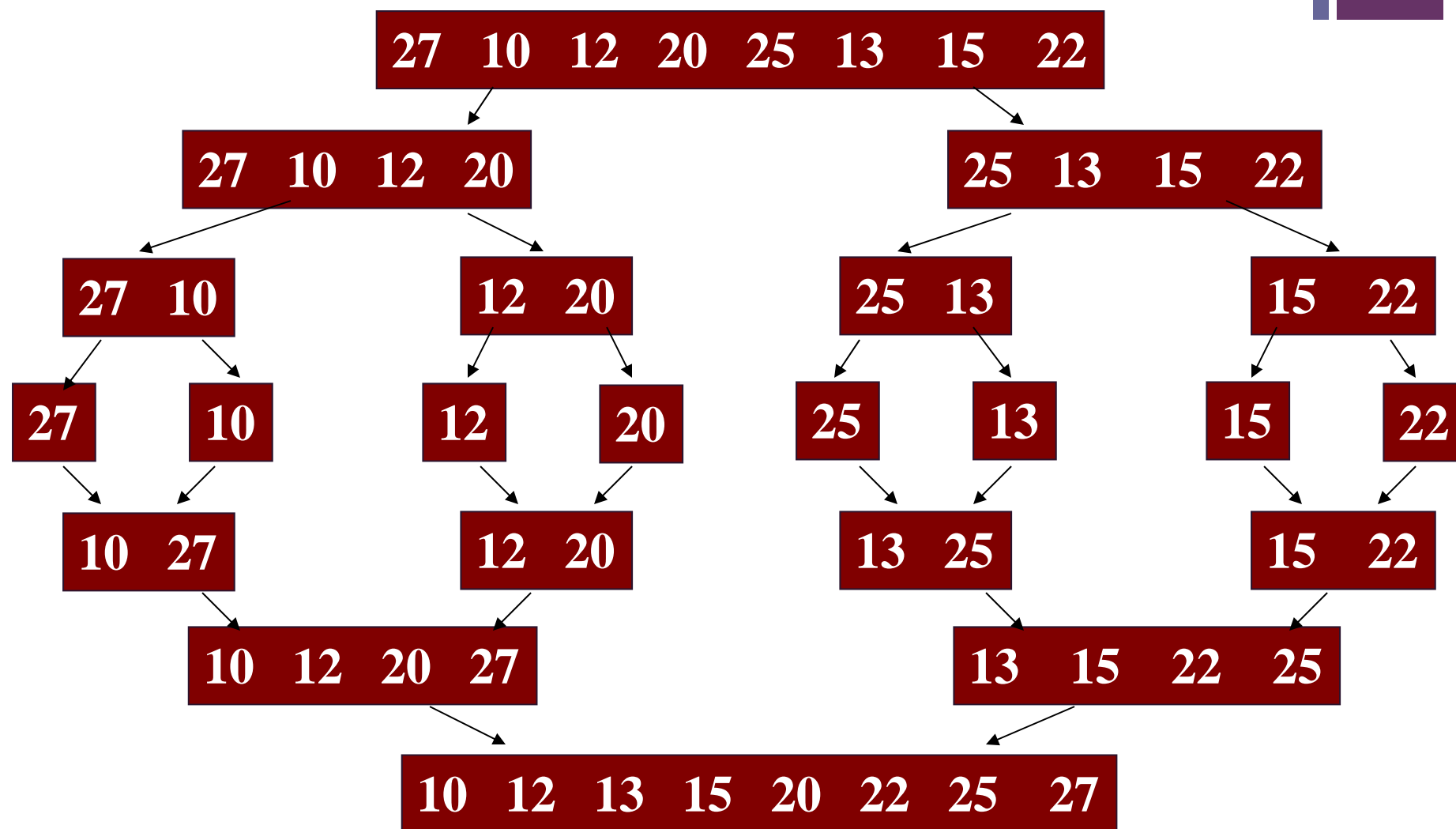
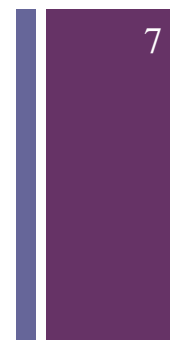
+ Búsqueda binaria

- ¿Es diferente el comportamiento del algoritmo recursivo vs. el iterativo?
- **NO** en el contexto general del tiempo de ejecución...
 - La complejidad de tiempo es diferente, pero por un valor constante...
 - Por lo tanto, el orden es el mismo: **$O(\log n)$**
- **SI** en el contexto de la complejidad de espacio del algoritmo, por el uso del stack en la recursividad.

+ Merge Sort

- Divide el arreglo en 2 subarreglos.
- Se ordenan ambos subarreglos.
- Se forma el arreglo ordenado, considerando que se tienen 2 subarreglos ya ordenados.

+ Merge Sort



+ Algoritmo: Merge Sort

```
void MergeSort (inicio, fin)  
  
if (inicio < fin) {  
  
    mitad = (inicio+fin) /2;  
  
    MergeSort(inicio, mitad);  
  
    MergeSort(mitad+1, fin);  
  
    Une(inicio, mitad, fin);  
  
}
```

+ Algoritmo: Une (Merge)

```
void Une (inicio, mitad, fin) {  
    i = inicio; j = mitad+1; k = inicio;  
    while (i<=mitad) && (j<=fin) {  
        if (arreglo[i] < arreglo[j]) {  
            aux[k] = arreglo[i];    i = i+1; }  
        else {  
            aux[k] = arreglo[j]    j = j+1; }  
        k = k + 1;  
    }  
    if (i>mitad)  
        Mover elementos j a fin del arreglo al arreglo aux de k a fin;  
    else  
        Mover elementos i a mitad del arreglo al arreglo aux de k a fin;  
    Copiar aux a arreglo;  
}
```

+ Análisis del Merge Sort

■ ¿Porqué no es un análisis “every-case”?

- El algoritmo *Une* tiene comportamiento distinto dependiendo del caso.

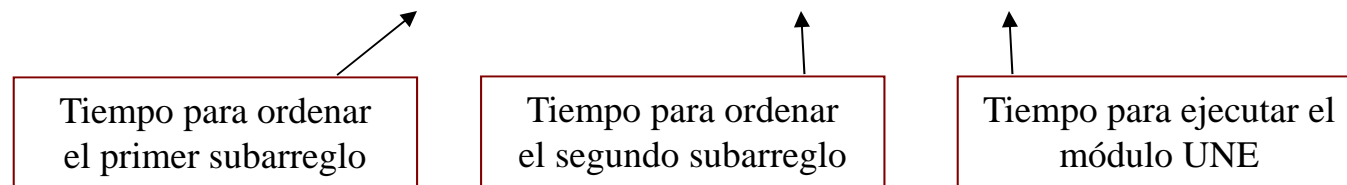
■ ¿Cuál es el peor caso si la operación de comparación es la que determina la complejidad del algoritmo?

- Si $n1$ y $n2$ son los tamaños de los subarreglos...
- Cuando $n1-1$ datos del primer subarreglo son menores a los datos del otro subarreglo...
- Se hacen $n1 - 1 + n2$ comparaciones, que equivalen a $n - 1$

+ Análisis del Merge Sort

- Sea $T(n)$ el peor tiempo para hacer el Merge Sort a un arreglo de n elementos...

$$T(n) = T(n/2) + T(n/2) + n-1$$



$$T(n) = 2T(n/2) + n-1$$

- La recurrencia se resuelve con: $n \log n - (n - 1)$
- Por lo tanto, el orden del peor caso es: $O(n \log n)$

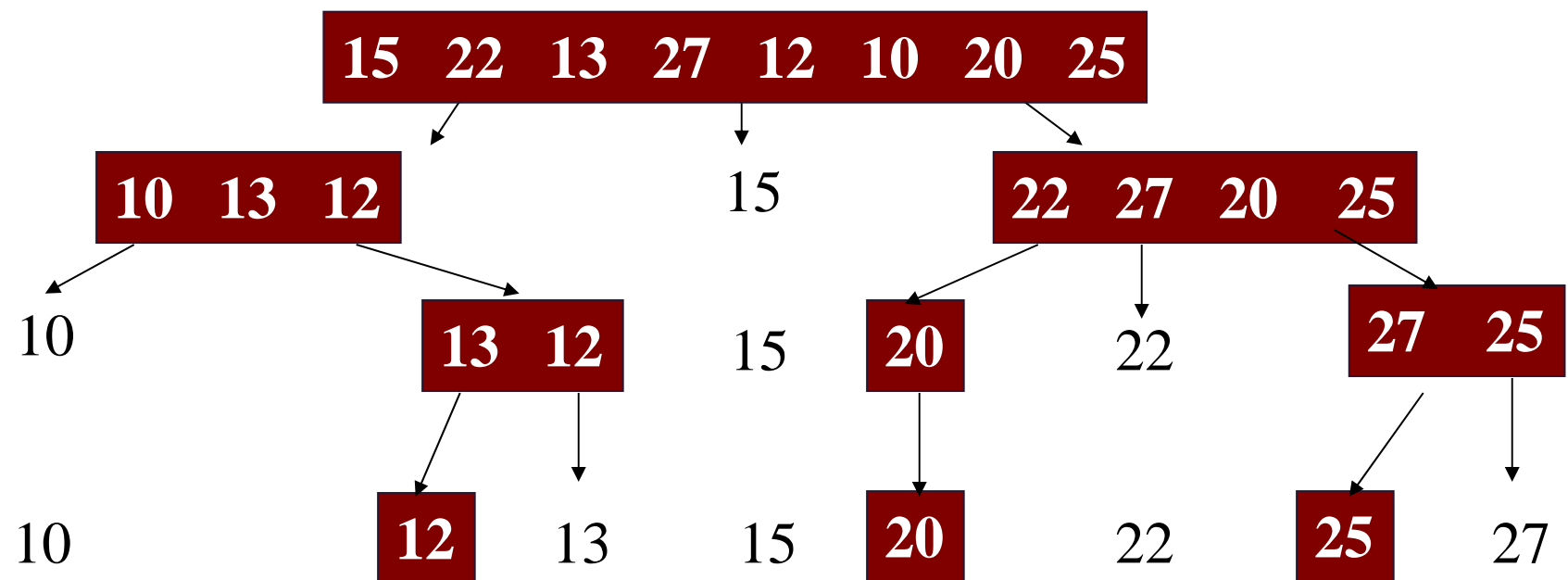
+ Quick Sort

- Divide el arreglo en 2 particiones, una que contiene a los elementos menores a un elemento pivote, y otra que contiene a los elementos mayores al pivote.
- Se ordenan ambas particiones, y automáticamente se tiene todo el arreglo ordenado.
- La elección del elemento pivote es libre (por facilidad, se toma el primer elemento del arreglo).

+ Ejemplo: Quick Sort

13

Considerando que el primer elemento del arreglo es el pivote:



+ Algoritmo: Quick Sort

```
void QuickSort (inicio, fin){
```

```
if (inicio < fin) {
```

```
    Partición(inicio, fin, pivote)
```

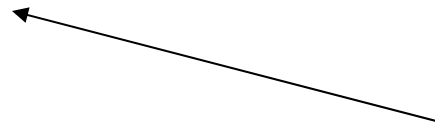
```
    QuickSort(inicio, pivote-1);
```

```
    QuickSort(pivote+1, fin);
```

```
}
```

```
}
```

Valor resultante



+ Algoritmo: Partición

```
void Partición (inicio, fin, pivote){  
    elempivote = arreglo[inicio]; j = inicio;  
    for (int i=inicio+1; i≤fin; i++) {  
        if (arreglo[i] < elempivote) {  
            j = j+1;  
            Intercambia arreglo[i] con arreglo[j]  
        }  
    }  
    pivote = j;  
    Intercambia arreglo[inicio] con arreglo[pivote]  
}
```

EJEMPLO: 15 22 13 27 12 10 20 25

+ Análisis del Quick Sort

- ¿Porqué no es un análisis “every-case”?
 - La partición del arreglo es variable (depende del pivote).
- ¿Cuál es el peor caso si la operación de comparación (al hacer la Partición) es la que determina la complejidad del algoritmo?
 - Cuando la partición genera un subarreglo vacío y el otro con $n-1$ datos
 - Se hacen $n - 1$ comparaciones

+ Análisis del Quick Sort

- Sea $T(n)$ el peor tiempo para hacer el Quick Sort a un arreglo de n elementos...

$$T(n) = T(0) + T(n-1) + n-1$$

Tiempo para ordenar
el subarreglo vacío = 0

Tiempo para ordenar
el segundo subarreglo

Tiempo para ejecutar el
módulo PARTICIÓN

$$T(n) = T(n-1) + n-1$$

- La recurrencia se resuelve con: $n * (n - 1) / 2$
- Por lo tanto, el orden del peor caso es: $O(n^2)$

+ Complejidad de los algoritmos

18

■ MERGE SORT:

- Peor caso: **$O(n \log n)$**
- Caso promedio: **$O(n \log n)$**

■ QUICK SORT:

- Peor caso: **$O(n^2)$**
- Caso promedio: **$O(n \log n)$**

*Demostración
en el libro*



+ Algoritmo de Strassen para Multiplicar Matrices

- El análisis matemático de Strassen, descubrió que para:

$$\begin{bmatrix} c11 & c12 \\ c21 & c22 \end{bmatrix} = \begin{bmatrix} a11 & a12 \\ a21 & a22 \end{bmatrix} \times \begin{bmatrix} b11 & b12 \\ b21 & b22 \end{bmatrix}$$

Existen los valores:

$$m1 = (a11 + a22) * (b11 + b22)$$

$$m2 = (a21 + a22) * b11$$

$$m3 = a11 * (b12 - b22)$$

$$m4 = a22 * (b21 - b11)$$

$$m5 = (a11 + a12) * b22$$

$$m6 = (a21 - a11) * (b11 + b12)$$

$$m7 = (a12 - a22) * (b21 + b22)$$

Tales que:

$$c11 = m1 + m4 - m5 + m7$$

$$c12 = m3 + m5$$

$$c21 = m2 + m4$$

$$c22 = m1 + m3 - m2 + m6$$

+ ¿Qué ventaja tiene el algoritmo de Strassen?

20

- La solución tradicional en matrices 2×2 , requiere de:
 - 8 multiplicaciones y 4 sumas...
- La solución de Strassen requiere de:
 - 7 multiplicaciones y 18 sumas/restas...
- Aparentemente, no es significativo el beneficio...
- Pero ahorrar **una** multiplicación de matrices en el algoritmo de Strassen, a costa de más sumas o restas de matrices, tiene repercusiones significativas...

+ Algoritmo de Strassen

- Dividir cada una de las matrices en 4 submatrices, y resolver por el método de Strassen el problema.

$$\begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix} = \begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} \times \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix}$$

Cada submatriz
es de $n/2 \times n/2$

+ Algoritmo de Strassen

```
void Strassen (int n, matriz A, matriz B, &matriz C)
{ if ( n <= umbral )
    Calcular C = A x B de forma tradicional
  else
  {   Calcular M1 hasta M7 llamando a Strassen
      // ejemplo:  $M1 = (A11 + A22) * (B11 + B22)$ 
      //      Suma_Matriz (n/2, A11, A22, R1)
      //      Suma_Matriz (n/2, B11, B22, R2)
      //      Strassen (n/2, R1, R2, M1)
      Calcular C11 hasta C22 utilizando M1 a M7
  }
}
```


+ Análisis del algoritmo de Strassen

- ¿Cuál es la operación básica a medir?
 - La MULTIPLICACIÓN de 2 datos.
- Sea $T(n)$ el tiempo de una multiplicación de 2 matrices de $n \times n$...
 - ¿Cuál es la fórmula para calcular $T(n)$?
- El algoritmo de Strassen, requiere de 7 multiplicaciones de matrices de $n/2 \times n/2$, entonces...
 - $T(n) = 7 * T(n/2)$
- La recurrencia se resuelve $T(n) = n^{\log 7}$

$$O(n^{2.81})$$

+ Aritmética de enteros grandes

- Un entero grande puede ser almacenado en una estructura en la que se guarde cada dígito del número.
- ¿Qué algoritmos están involucrados en la implementación de este tipo de dato?
 - Sumar 2 enteros grandes
 - Restar 2 enteros grandes
 - Multiplicar 2 enteros grandes
 - Dividir 2 enteros grandes, etc.

+ Orden de los algoritmos

- La suma y la resta, realizadas de manera tradicional, tienen un comportamiento lineal $O(n)$, donde n es la cantidad de dígitos del entero grande.
- La multiplicación en forma tradicional, tiene un comportamiento cuadrático $O(n^2)$ en el **peor caso**...
- Sin embargo, la **multiplicación/división/residuo** por una potencia de 10, tienen un comportamiento lineal $O(n)$.

+ Propuesta de mejora al algoritmo de la multiplicación

- Dividir el número en 2 números de tal manera que
$$e = x * 10^m + y$$
- EJEMPLO: $8,234,127 = 8234 * 10^3 + 127$
- Si los 2 números que se desean multiplicar se expresan de esta manera, se tiene que:
- $$\begin{aligned} e_1 * e_2 &= (x_1 * 10^m + y_1) * (x_2 * 10^m + y_2) \\ &= x_1 x_2 * 10^{2m} + (x_1 y_2 + x_2 y_1) * 10^m + y_1 y_2 \end{aligned}$$
- De esta manera se hacen 4 multiplicaciones con enteros más pequeños...

+ Propuesta de mejora al algoritmo de la multiplicación

- El análisis de la propuesta, indica que el algoritmo sigue teniendo un orden cuadrático...
- Sin embargo, se puede eliminar una multiplicación con el siguiente análisis:
- $r = (x_1 + y_1) * (x_2 + y_2) = x_1x_2 + (x_1y_2 + x_2y_1) + y_1y_2$
- y por lo tanto: $(x_1y_2 + x_2y_1) = r - x_1x_2 - y_1y_2$
- y sustituyendo en: $x_1x_2 * 10^{2m} + (x_1y_2 + x_2y_1) * 10^m + y_1y_2$
- $x_1x_2 * 10^{2m} + (r - x_1x_2 - y_1y_2) * 10^m + y_1y_2$
- $x_1x_2 * 10^{2m} + ((x_1+y_1)*(x_2+y_2) - x_1x_2 - y_1y_2) * 10^m + y_1y_2$

+ Algoritmo de la multiplicación

```

entero_grandeMultiplica (n1, n2: entero_grande){
  n = cantidad de dígitos mayor entre n1 y n2.
  if (n1 = 0) || (n2 = 0) return 0;
  else if (n <= umbral) return n1 * n2 tradicional;
  else{
    m = n / 2;
    x1 = n1 div 10m; y1 = n1 mod 10m;
    x2 = n2 div 10m; y2 = n2 mod 10m;
    r = Multiplica(x1+y1, x2+y2);
    p = Multiplica(x1, x2); q = Multiplica(y1, y2);
    return (p X 102m + (r-p-q) X 10m + q);
  }
}

```

Límite en que resulta mejor
realizar la operación en forma
tradicional

$O(n^{1.58})$

+ Generalización de Divide y vencerás

Función $DV(x)$

if x es suficientemente pequeño o sencillo
 return (solución tradicional al problema)

else{

 Descomponer x en casos más pequeños x_1, x_2, \dots, x_m

for $i = 1$ ***to*** m ***do*** $y_i = DV(x_i)$

 Recombinar las y_i para obtener la solución y de x

return y

}

+ Condiciones para utilizar Divide y Vencerás

- Debe ser posible descomponer el problema en subproblemas.
- Debe ser posible recomponer las soluciones de una manera eficiente.
- Los subproblemas deben de ser, en lo posible, del mismo tamaño.
- **¿Cuándo NO utilizar DyV?**
 - Si el tamaño de los subproblemas es casi el mismo tamaño original.
 - Si la cantidad de subproblemas es casi la misma que el tamaño del problema.

+ Comportamiento general de algoritmos con DyV

■ Sea:

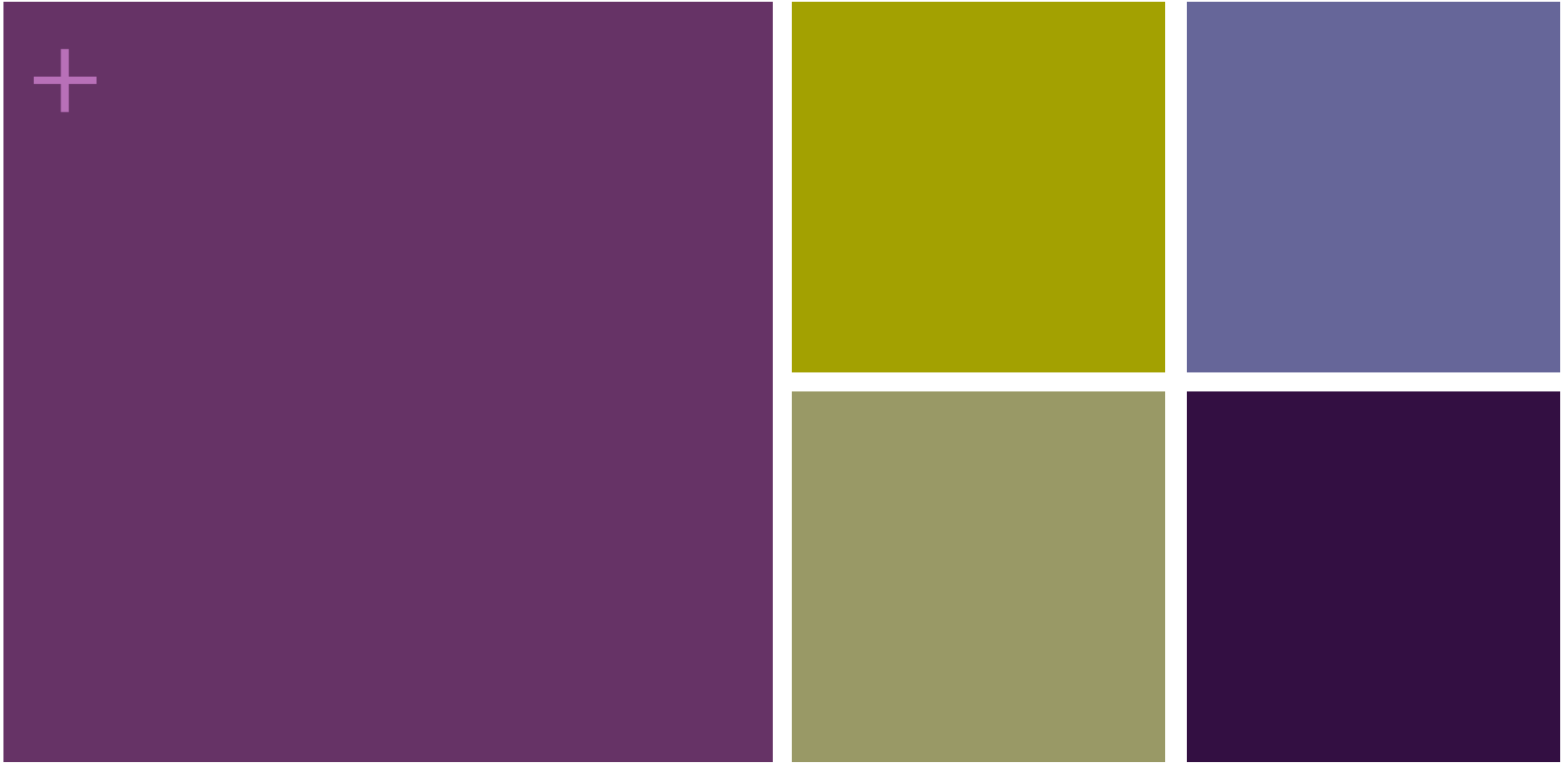
- m la cantidad de subcasos más pequeños que se requieren para la solución (llamadas recursivas).
- b el factor con que se divide el problema en casos más pequeños.
- k un valor cualquiera para el análisis.

■ Entonces el orden es:

$$\mathbf{O(n^k)} \text{ si } m < b^k$$

$$\mathbf{O(n^k \log n)} \text{ si } m = b^k$$

$$\mathbf{O(n^{\log_b m})} \text{ si } m > b^k$$



Ejercicios

+ Diseño de algoritmo

- Diseña un algoritmo utilizando la técnica de Divide y Vencerás que calcule:

$$\sum_{x=1}^n x^2$$

+ Solución

```
function sumatoria (inicio, fin)
  if (inicio == fin)
    return fin*fin
  else
    m = (inicio + fin) div 2
    x1 = sumatoria (inicio, m)
    x2 = sumatoria (m+1, fin)
    return x1 + x2
```

+ Fórmula recursiva

- Calcula la fórmula recursiva de $T(n)$ para el siguiente algoritmo:

Función DVx (x, n)

if (n==1)

return 1

else

m = n div 3

descompone (x, x₁, x₂, x₃, x₄, x₅, m) // $T_d(n) = n^2 + 3$

for i = 1 to 5 do

y_i = DVx (x_i, m)

y = recombina (y₁, y₂, y₃, y₄, y₅, m) // $T_r(n) = n - 1$

return y

+ Recurrencia de $T(n)$

- Resuelve la siguiente recurrencia (encuentra su forma cerrada):
 - Si $n = 1$, entonces $T(n) = 1$
 - Si $n > 1$, entonces $T(n) = 5 T(n/3) + 2n$
- Tomar en cuenta la siguiente fórmula:

$$\sum_{i=1}^n a^i = \frac{a^{n+1} - 1}{a - 1}$$

Análisis y Diseño de Algoritmos

Programación Dinámica



Programación dinámica

- Técnica para enfrentar la solución de problemas.
- Consiste en dividir un problema en instancias del problema más pequeñas
 - Resolver primero las instancias más pequeñas,
 - Almacenar los resultados, y
 - Contar con estos para no recalcularlos cuando la solución de instancias de mayor nivel lo necesiten.
- Utiliza un enfoque de tipo *bottom-up*.
- Transforma a una solución natural de tipo **recursivo**, **en una solución iterativa**, almacenando resultados.

EJEMPLO: Coeficiente binomial

- El coeficiente binomial es un término que sirve para encontrar la cantidad de combinaciones de tamaño 'k' entre 'n' elementos, y se obtiene así:

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

- Evitando el comportamiento de la función factorial, el coeficiente binomial también se puede definir así:

$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0 \text{ o } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \end{cases}$$

Coeficiente binomial

- La definición recursiva, permite plantear un algoritmo de solución con la técnica de divide y vencerás:

function binomial(int n, k): int;

if (k=0) or (k=n) then return 1

else return binomial(n-1, k-1) + binomial(n-1,k);

$$O\left(\begin{matrix} n \\ k \end{matrix}\right)$$

- ¿Cuál es el inconveniente de este algoritmo?
- Se recalculan los mismos términos varias veces..
- Ejemplo: *binomial(n-1, k-1)* y *binomial(n-1,k)* requieren ambos de *binomial(n-2,k-1)*.

Coeficiente binomial

- Enfoque de solución con PROGRAMACIÓN DINÁMICA:
 - Considerar que se tiene un arreglo B en donde se guardan los coeficientes binomiales (i,j).
 - Dada la definición recursiva del problema se tiene que:
$$B[i,j] = 1 \text{ si } j = 0 \text{ o } j = i$$
$$B[i,j] = B[i-1,j-1] + B[i-1,j] \text{ si } i > 0 \text{ e } i < j$$
 - Resolver las instancias de B comenzando por el primer renglón (problema más pequeño), y continuando secuencialmente.

Coeficiente binomial

EJEMPLO: Encontrar **B[4,2]**

- $B[0,0] = 1$
- $B[1,0] = 1$
- $B[1,1] = 1$
- $B[2,0] = 1$
- $B[2,1] = B[1,0] + B[1,1] = 2$
- $B[2,2] = 1$
- $B[3,0] = 1$
- $B[3,1] = B[2,0] + B[2,1] = 3$
- $B[3,2] = B[2,1] + B[2,2] = 3$
- $B[4,0] = 1$
- $B[4,1] = B[3,0] + B[3,1] = 4$
- $B[4,2] = B[3,1] + B[3,2] = 6$

Algoritmo para el coeficiente binomial

```
int binomial (int n; int k){  
  for (int i=0; i<n; i++)  
    for (int j= 0; j<minimo(i,k); j++)  
      if ((j=0) or (j=i)) B[i][j] = 1  
      else  
        B[i][j] = B[i-1][j-1] + B[i-1][ j];  
  return B[n][k];  
}
```

$O(nk)$

Repaso de grafos

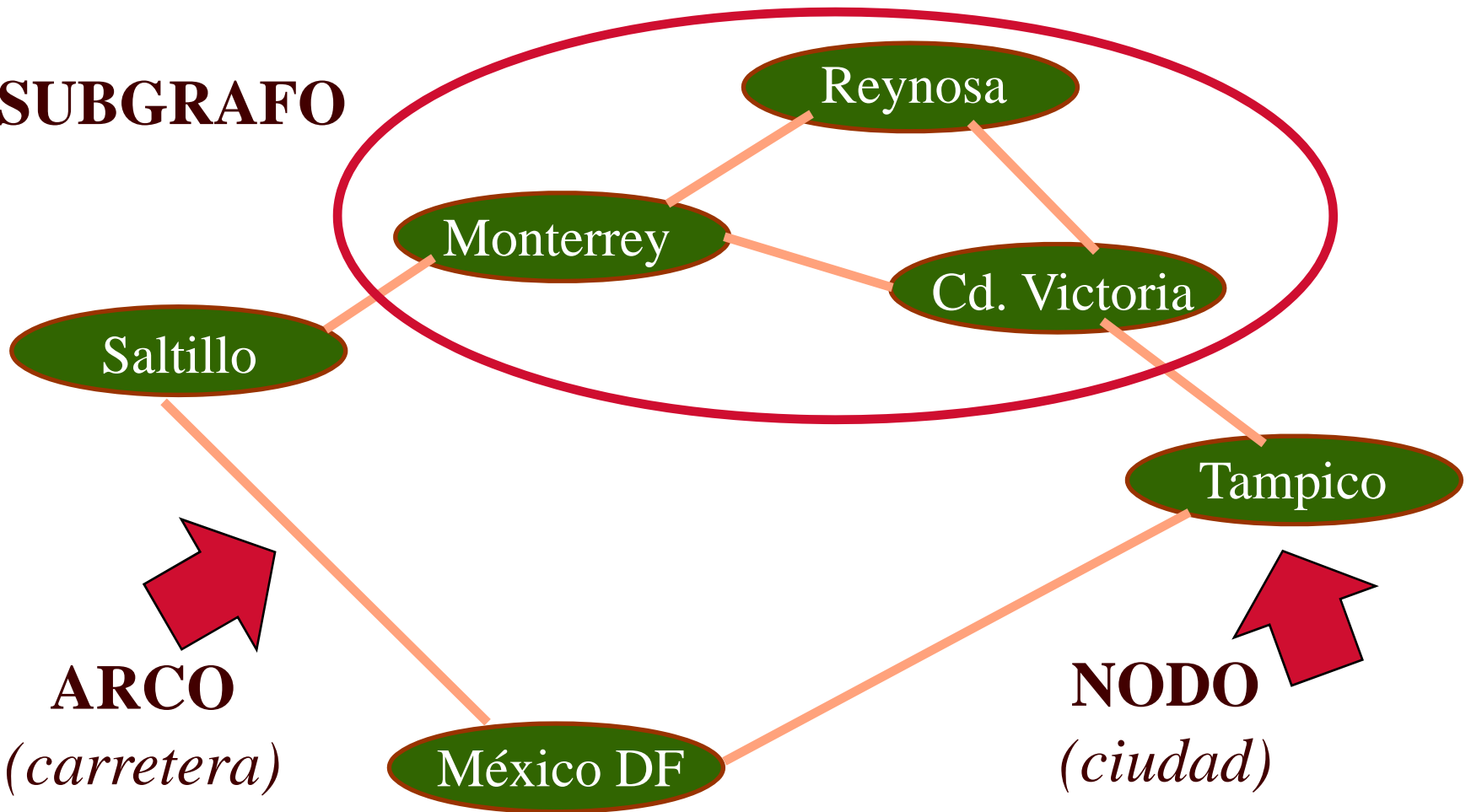


¿Qué es un grafo?

- ❑ Bajo el enfoque de las estructuras de datos, es la estructura de datos más general...
- ❑ La relación entre los elementos es de tipo RED, es decir, de “muchos a muchos”...
- ❑ Es la representación de muchos modelos “gráficos” o abstracciones...
- ❑ Autómatas de estados finitos, Diagramas de estado, Diagramas de flujo, etc.

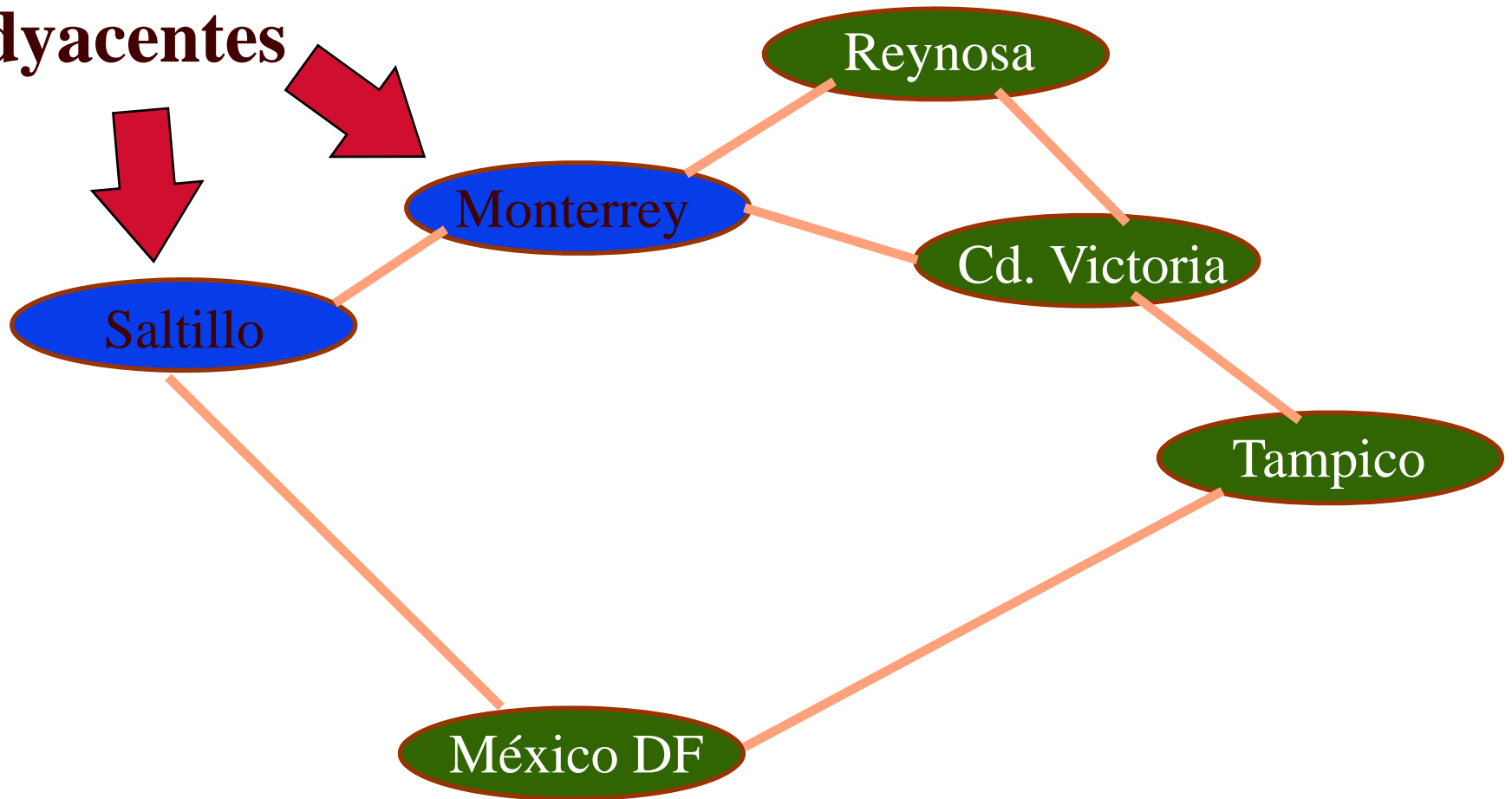
Terminología...

SUBGRAFO

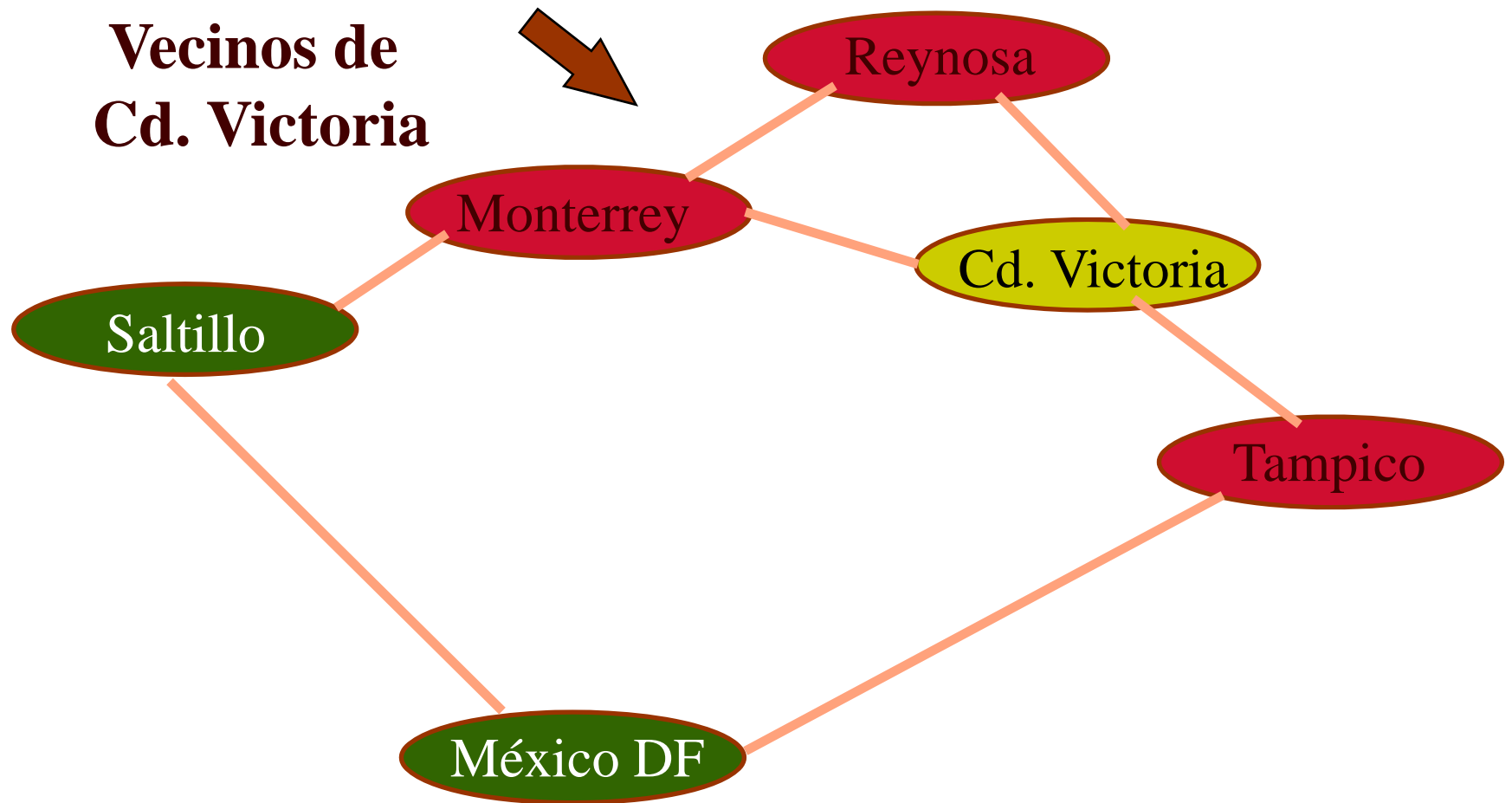


Terminología...

Adyacentes

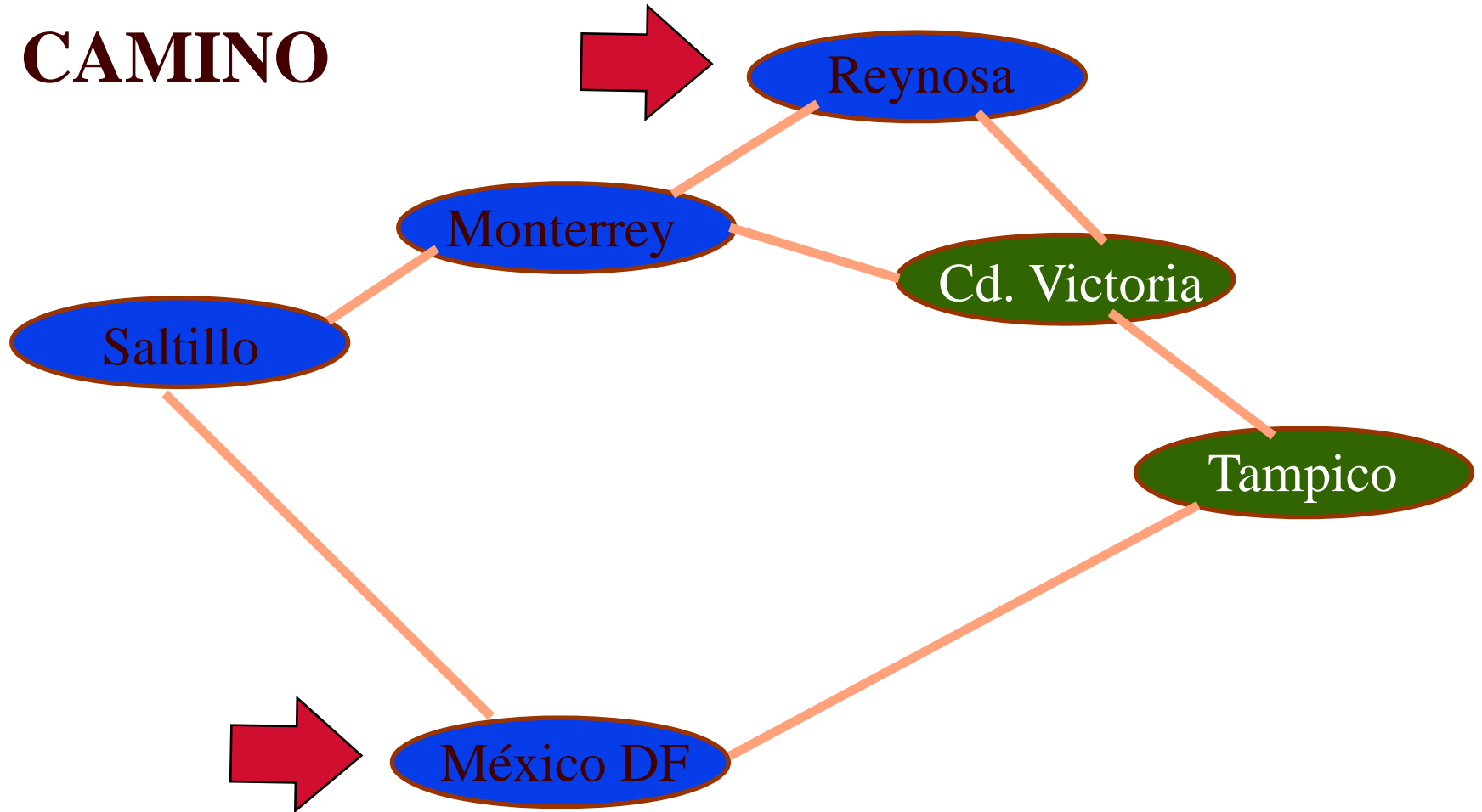


Terminología...

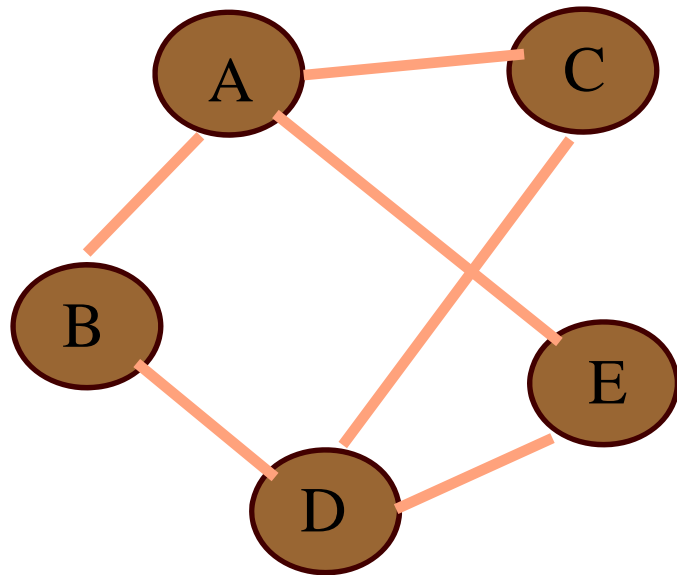


Terminología...

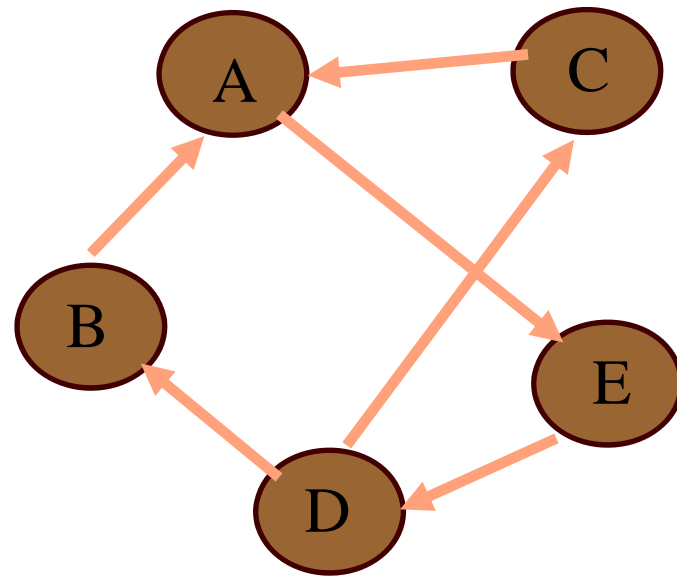
CAMINO



Terminología...

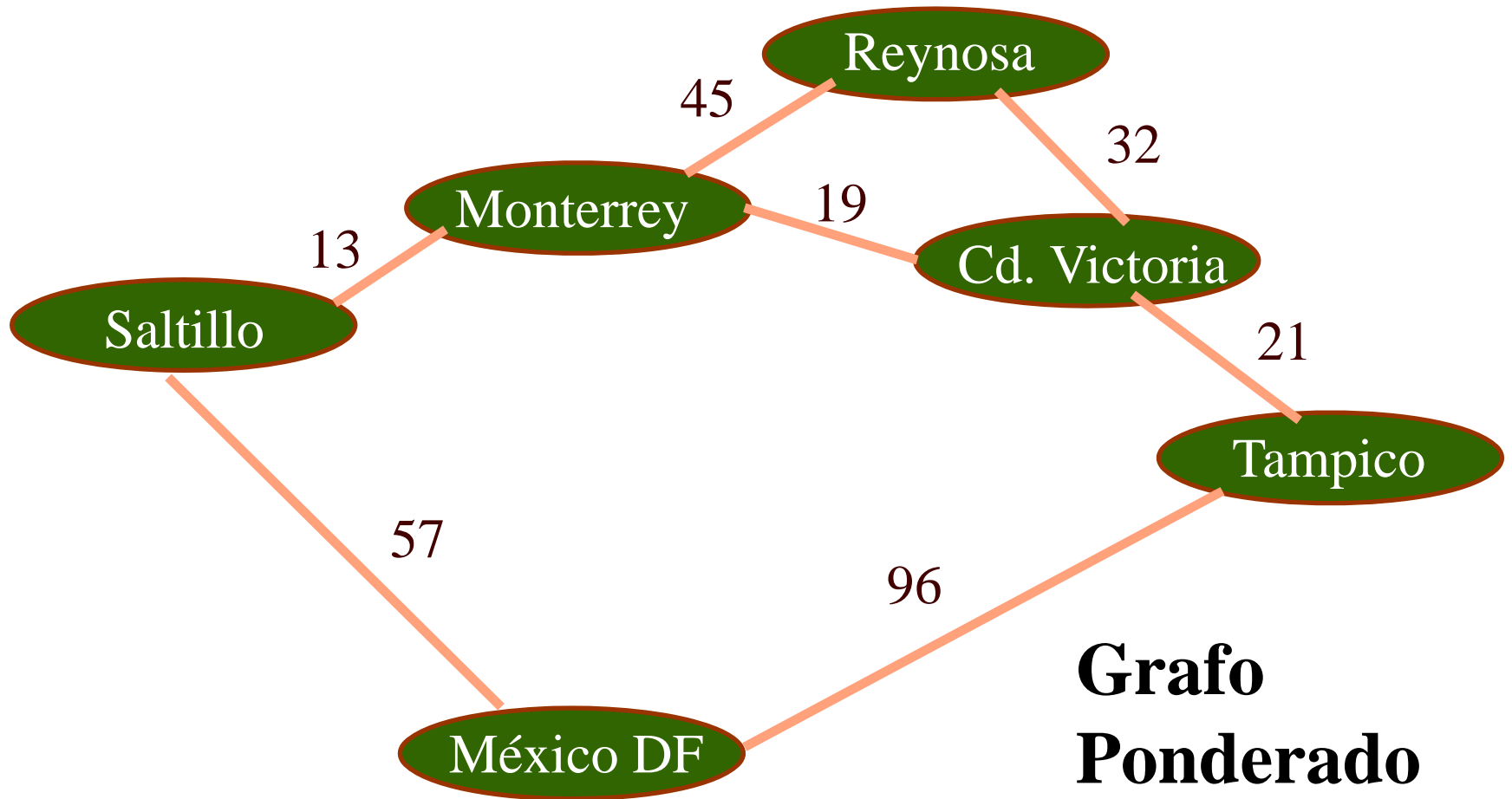


Grafo No-Dirigido

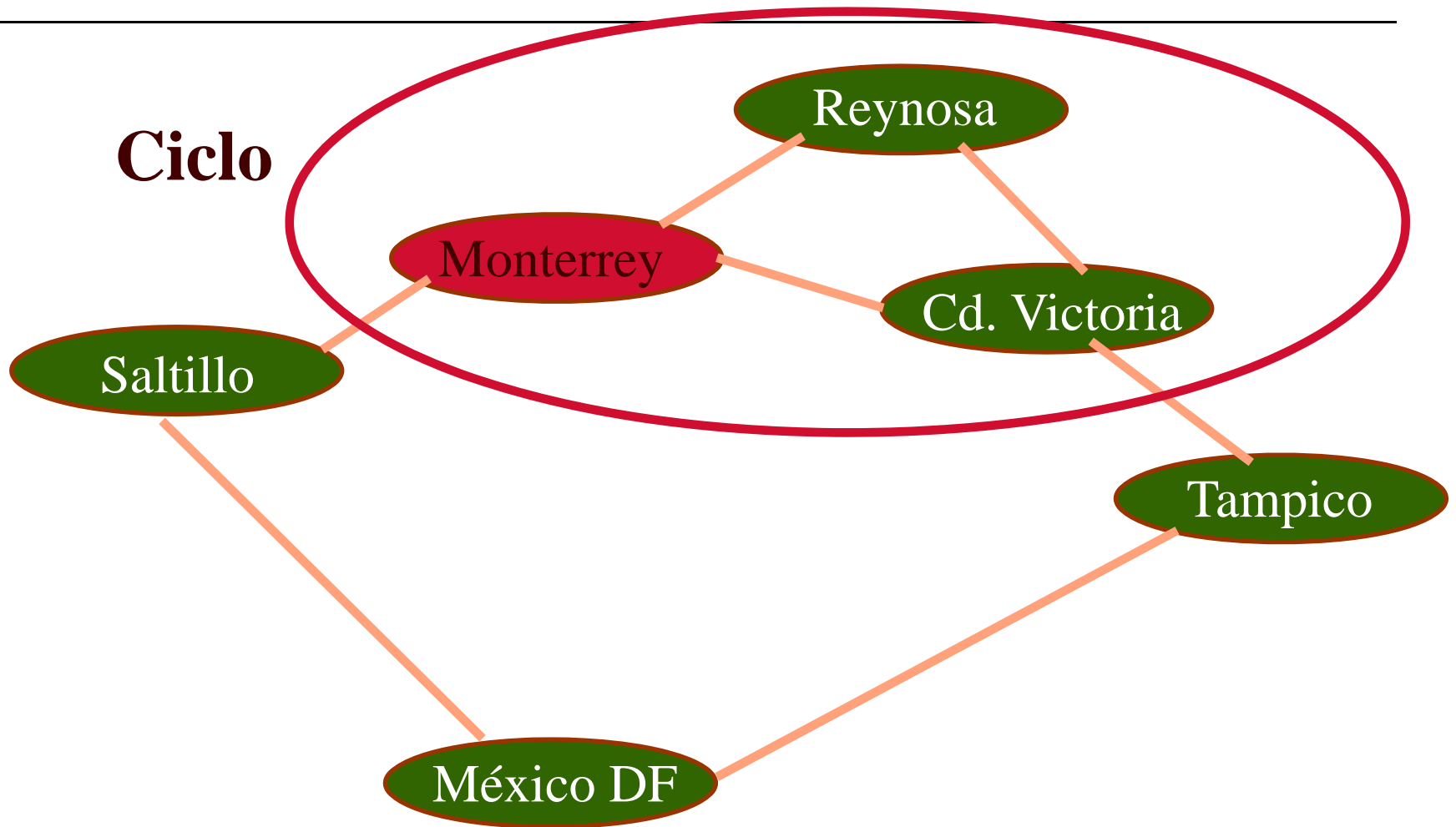


Grafo Dirigido (Digrafo)

Terminología...



Terminología...





Aplicaciones de un Grafo

- Conectividad, Redes de Transporte
 - ¿Existe un camino entre dos Nodos?
 - ¿Cuál es el costo mínimo de conexión para todos los Nodos?
 - *¿Cuál es la ruta óptima para ir de un Nodo a otro?*
- Autómatas o Diagramas de Estado

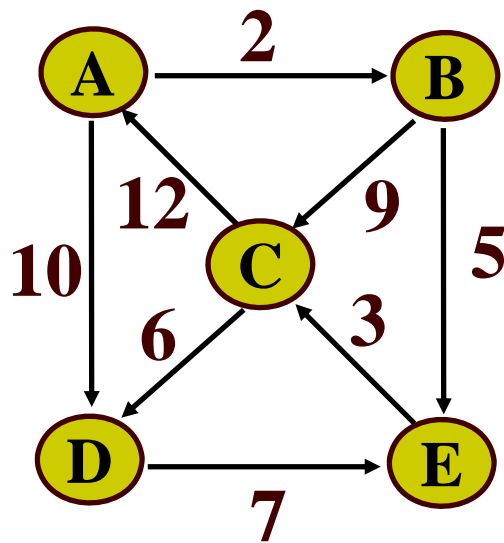


Representación de Grafos

- Existen muchas formas de representar a un grafo, sin embargo, las más comunes son:
 - *Matriz de Adyacencias*
 - *Lista de Adyacencias*
 - *Lista de Arcos*

Algoritmo de Floyd

El problema del camino más corto



Camino más corto de A a C:

- A-B-C = 11
- **A-B-E-C = 10**
- A-D-E-C = 20

- Es un problema de optimización...
- Se busca obtener, para un vértice origen, el costo mínimo para llegar a cada uno del resto de los nodos, sin importar por cuántos nodos se tenga que pasar...



Propuestas de solución

- ❑ **Algoritmo obvio:** Obtener todos los caminos posibles para llegar de un nodo a otro, y elegir al menor... Este proceso se repite para todos los nodos...
 - El análisis de este algoritmo lleva a concluir que tendría un comportamiento de orden factorial...
- ❑ **Algoritmo con colas priorizadas:** Conceptualmente válido, pero complejo de implementar (requiere de HEAPS).

Algoritmo de Floyd

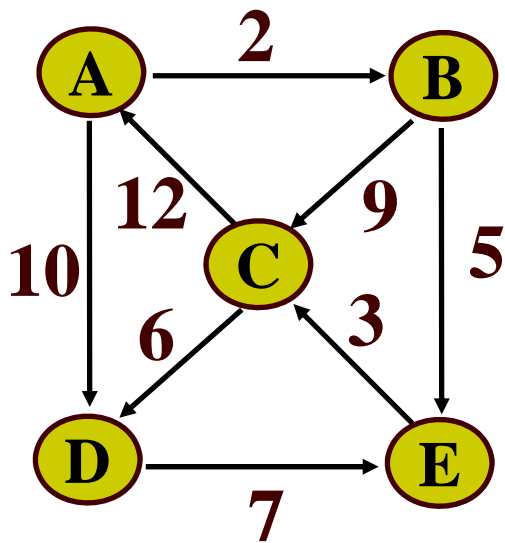
- Propuesta basada en la técnica de la programación dinámica...
- Se basa en la representación del grafo en una matriz de adyacencias bajo las siguientes condiciones:
 - *$M[i,j]$ guarda el peso del arco del nodo i al nodo j .*
 - *$M[i,j]$ guarda el valor 0 cuando $i = j$*
 - *$M[i,j]$ guarda el valor ∞ cuando no hay arco entre el nodo i y el nodo j .*



Algoritmo de Floyd

- Dada la matriz de adyacencias, obtener la matriz del camino más corto en donde el elemento $[i,j]$ es el valor del camino más corto para ir del nodo i al nodo j ...
- El algoritmo también se puede utilizar para mostrar el camino más corto (no sólo el valor)...
- Este algoritmo tiene un comportamiento de orden **$O(n^3)$** .

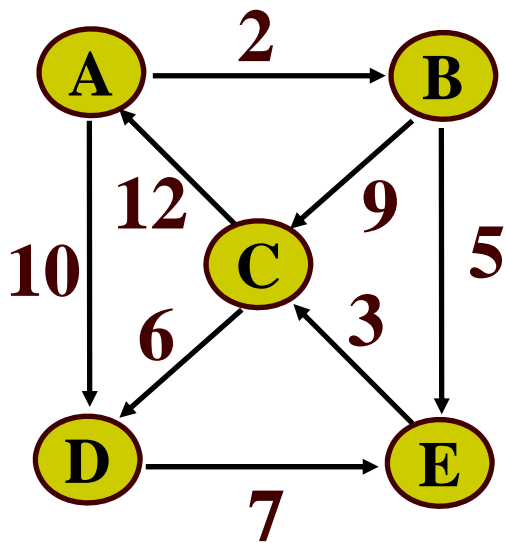
Ejemplo



MATRIZ DE ADYACENCIAS

	A	B	C	D	E
A	0	2	∞	10	∞
B	∞	0	9	∞	5
C	12	∞	0	6	∞
D	∞	∞	∞	0	7
E	∞	∞	3	∞	0

Ejemplo



MATRIZ DEL CAMINO MÁS CORTO

	A	B	C	D	E
A	0	2	10	10	7
B	20	0	8	14	5
C	12	14	0	6	13
D	22	24	10	0	7
E	15	17	3	9	0

Algoritmo de Floyd

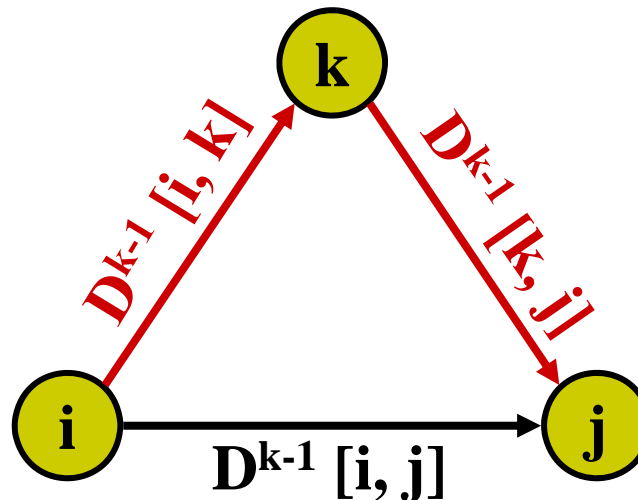
- Sea \mathbf{D}^k una matriz del camino más corto, en donde cada elemento $[i,j]$ indica:
 - Cuál es el camino más corto desde el nodo i hasta el nodo j ,
 - Pasando solamente por los nodos desde 1 hasta k ...
- \mathbf{D}^0 es la matriz con los caminos más cortos sin pasar por otros vértices...
 - Por lo tanto, es la matriz de adyacencias original.
- \mathbf{D}^n será la matriz con los caminos más cortos pasando por TODOS los nodos posibles...
 - ***ESTA ES LA MATRIZ QUE SE BUSCA.***

Algoritmo de Floyd

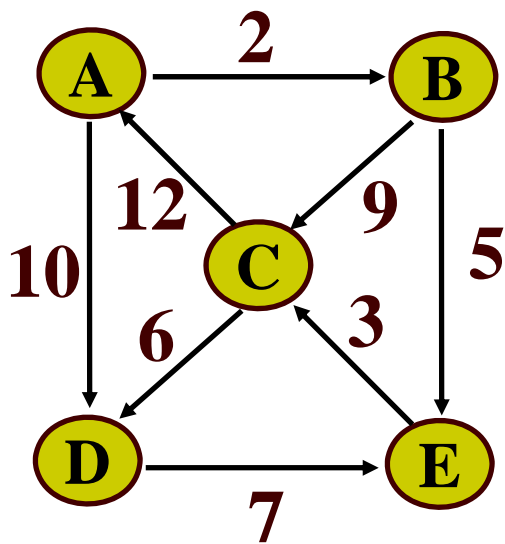
- ¿Cómo obtener a la matriz D^n dada la matriz D^0 ?
- El enfoque de la PROGRAMACIÓN DINÁMICA preguntaría si:
 - A partir de la matriz D^0 se puede obtener D^1 ...
 - Si obteniendo a D^1 , se puede obtener a D^2 ...
 - Y así sucesivamente hasta obtener a D^n , que es lo que buscamos.

Algoritmo de Floyd

- Se debe escoger el mínimo de 2 caminos:
 - El que **no** incluye al nodo k, o
 - El que **sí** incluye al nodo k.



Ejemplo



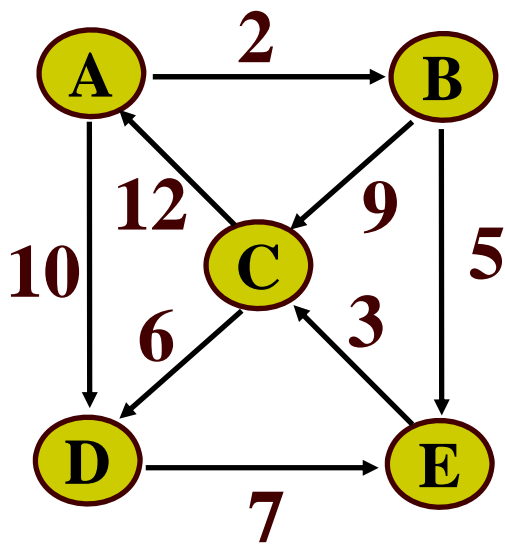
$$C-A + A-B = 12 + 2$$

Mínimo entre:
C-D y C-A + A-D

D^0	A 1	B 2	C 3	D 4	E 5
A 1	0	2	∞	10	∞
B 2	∞	0	9	∞	5
C 3	12	∞	0	6	∞
D 4	∞	∞	∞	0	7
E 5	∞	∞	3	∞	0

D^1	A 1	B 2	C 3	D 4	E 5
A 1	0	2	∞	10	∞
B 2	∞	0	9	∞	5
C 3	12	14	0	6	∞
D 4	∞	∞	∞	0	7
E 5	∞	∞	3	∞	0

Ejemplo



D^1	A 1	B 2	C 3	D 4	E 5
A 1	0	2	∞	10	∞
B 2	∞	0	9	∞	5
C 3	12	14	0	6	∞
D 4	∞	∞	∞	0	7
E 5	∞	∞	3	∞	0

D^2	A 1	B 2	C 3	D 4	E 5
A 1	0	2	11	10	7
B 2	∞	0	9	∞	5
C 3	12	14	0	6	19
D 4	∞	∞	∞	0	7
E 5	∞	∞	3	∞	0

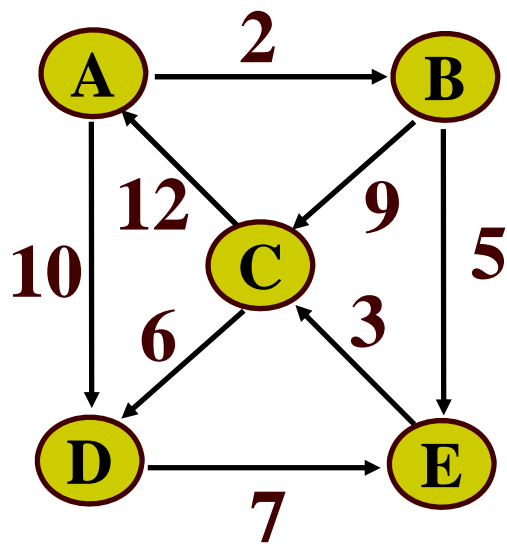
$$A-B + B-C = 2 + 9$$

$$A-B + B-E = 2 + 5$$

Mínimo entre: $C-D$ y $C-B + B-D$

$$C-B + B-E = 14 + 5$$

Ejemplo



D^2	A 1	B 2	C 3	D 4	E 5
A 1	0	2	11	10	7
B 2	∞	0	9	∞	5
C 3	12	14	0	6	19
D 4	∞	∞	∞	0	7
E 5	∞	∞	3	∞	0

D^3	A 1	B 2	C 3	D 4	E 5
A 1	0	2	11	10	7
B 2	21	0	9	15	5
C 3	12	14	0	6	19
D 4	∞	∞	∞	0	7
E 5	15	17	3	9	0

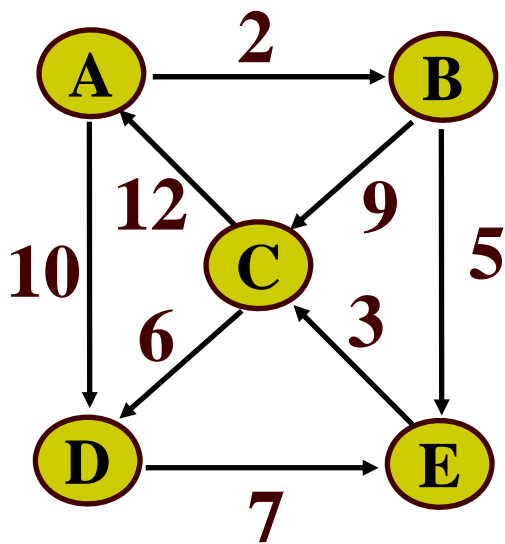
$$B-C + C-A = 9 + 12$$

Mínimo entre: A-D y A-C + C-D

Mínimo entre: A-E y A-C + C-E

$$E-C + C-A = 3 + 12$$

Ejemplo



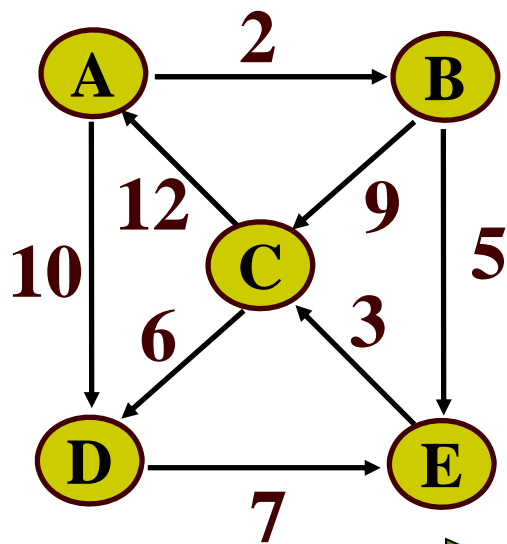
D³	A 1	B 2	C 3	D 4	E 5
A 1	0	2	11	10	7
B 2	21	0	9	15	5
C 3	12	14	0	6	19
D 4	∞	∞	∞	0	7
E 5	15	17	3	9	0

D⁴	A 1	B 2	C 3	D 4	E 5
A 1	0	2	11	10	7
B 2	21	0	9	15	5
C 3	12	14	0	6	13
D 4	∞	∞	∞	0	7
E 5	15	17	3	9	0

Mínimo entre: A-E y A-D + D-E

Mínimo entre: C-E y C-D + D-E

Ejemplo



MATRIZ META

Mínimo entre: $A-C$ y $A-E + E-C$

Mínimo entre: $B-A$ y $B-E + E-A$

$$D-E + E-A = 7 + 15$$

D⁴	A 1	B 2	C 3	D 4	E 5
A 1	0	2	11	10	7
B 2	21	0	9	15	5
C 3	12	14	0	6	13
D 4	∞	∞	∞	0	7
E 5	15	17	3	9	0

D⁵	A 1	B 2	C 3	D 4	E 5
A 1	0	2	10	10	7
B 2	20	0	8	14	5
C 3	12	14	0	6	13
D 4	22	24	10	0	7
E 5	15	17	3	9	0

Algoritmo de Floyd

- Generalizando...

$$D^k[i,j] = \text{minimo}(D^{k-1}[i,j] , D^{k-1}[i,k] + D^{k-1}[k,j])$$

- Esta relación recursiva, es el planteamiento de solución al problema...
- La técnica de la programación dinámica, sugiere que se obtenga el caso más pequeño y a partir de ese, se construyan los casos superiores, almacenando resultados si se requieren...

Algoritmo de Floyd

- Por lo tanto, el algoritmo se puede plantear de la siguiente manera...

D = matriz de adyacencias

for (int k=1; k<=n; k++)

for (int i=1; i<=n; i++)

for (int j=1; j<=n; j++)

D[i,j] = minimo(D[i,j], D[i,k]+D[k,j]);

Algoritmo de Floyd

- ❑ ¿Porqué se puede usar una sola matriz y **NO** se requieren ‘n’ matrices intermedias?
- ❑ Los valores de la k-ésima columna y el k-ésimo renglón **NO** cambian en la k-ésima iteración...
 - $D[i,k] = \text{minimo}(D[i,k], D[i,k]+D[k,k]) = \textcolor{red}{D[i,k]}$
 - $D[k,j] = \text{minimo}(D[k,j], D[k,j]+D[k,k]) = \textcolor{red}{D[k,j]}$
- ❑ Puesto que $D[i,j]$ en la k-ésima iteración se calcula con su propio valor previo y los de la k-ésima columna y el k-ésimo renglón que se mantienen de la iteración anterior, **NO HAY PROBLEMA** en usar la propia matriz...



Algoritmo de Floyd

- ❑ **¿Cómo encontrar los nodos que conforman el camino más corto?**
- ❑ Almacenar en una matriz auxiliar, el índice más grande del nodo intermedio por el que se pasa en el camino más corto del nodo i al nodo j .
- ❑ Utilizando esa matriz, si para ir del nodo i al nodo j se pasa por el nodo k , entonces se pasa también por el nodo con el índice más grande del camino más corto del nodo i al nodo k , y del nodo k al nodo j ...

Algoritmo de Floyd

- Para obtener la matriz con el último nodo visitado en el camino más corto...

D = matriz de adyacencias

P = matriz de último nodo inicializada en ceros.

for (int k = 1; k ≤ n; k++)

for (int i = 1; i ≤ n; i++)

for (int j = 1; j ≤ n; j++)

if (D[i,k] + D[k,j] < D[i,j]) {

P[i,j] = k;

D[i,j] = D[i,k] + D[k,j];}

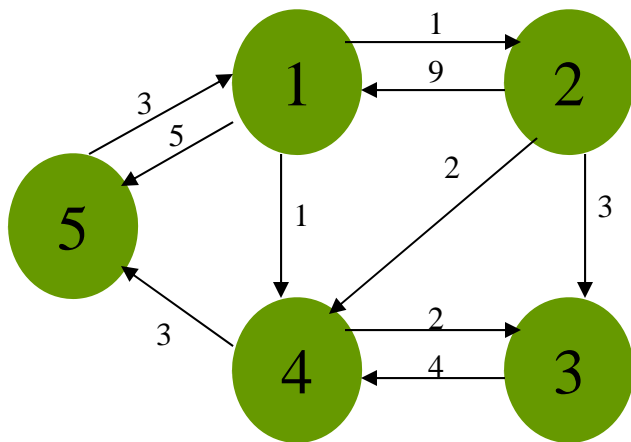
Algoritmo de Floyd

- Para desplegar el camino más corto...

```
void camino (int inicio, int fin){  
  if (P[inicio][fin] <> 0) {  
    camino(inicio, P[inicio][fin]);  
    write(P[inicio][fin]);  
    camino(P[inicio][fin], fin); }  
}
```

- Recordar que ***P[inicio, fin]*** es el nodo intermedio con el índice más grande en el camino más corto de inicio a fin.

Ejemplo



P	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

D ⁵	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

Modulo camino (inicio, fin)
if (P[inicio, fin] <> 0) then
 camino(inicio, P[inicio, fin]);
 write(P[inicio, fin]);
 camino(P[inicio, fin], fin);



El principio de optimalidad

- Todo problema de optimización, se puede resolver con la técnica de la **programación dinámica**, si la solución cumple con el *principio de optimalidad*...
- *La solución óptima de la instancia de un problema siempre contiene las soluciones óptimas de todas las subinstancias que conforman la solución...*
- Ejemplos: Camino más corto vs. más largo...

Multiplicación encadenada de matrices



Un ejemplo más...

Multiplicación encadenada de matrices

- A diferencia del problema resuelto por el algoritmo de Strassen, en que se hace más eficiente la multiplicación de matrices MUY grandes, ahora se buscará eficientizar el proceso de multiplicar una secuencia de matrices de tamaño relativamente pequeño...
- *¿Qué condición se debe cumplir para que dos matrices se puedan multiplicar?*
- La cantidad de columnas de la primer matriz debe de ser igual a la cantidad de renglones de la segunda matriz...



Multiplicación encadenada de matrices

- Sea la matriz M1 una matriz de orden $n \times m$ y la matriz M2 de orden $m \times p$...
- Si se multiplican ambas matrices, la matriz resultante será de orden $n \times p$...
- *¿Cuántas multiplicaciones escalares requiere el cálculo de un elemento de la matriz resultante?*
- m multiplicaciones...
- Por lo tanto, TODA la matriz resultante requiere de $m \times n \times p$ multiplicaciones...



Multiplicación encadenada de matrices

- *¿Cómo influye el orden en que se ejecuten las multiplicaciones de matrices en una secuencia encadenada de multiplicaciones $M_1 \times M_2 \times \dots \times M_n$?*
- Puesto que la cantidad de multiplicaciones escalares a realizar está determinado por el tamaño de la matrices a multiplicar, y...
- puesto que la multiplicación de matrices es ASOCIATIVA...
- Existe un orden OPTIMO para multiplicar las matrices y obtener los resultados más eficientemente...



EJEMPLO:

Multiplicación encadenada de matrices

- Se desea obtener la multiplicación de **A X B X C X D**, donde **A** es una matriz de 20 X 2, **B** una matriz de 2 X 30, **C** una matriz de 30 X 12 y **D** una matriz de 12 X 8...
- *¿De qué tamaño será la matriz resultante?*
- **20 X 8**
- *¿Cuál será la forma más eficiente de multiplicar las matrices?*
- La que requiera menos multiplicaciones escalares...

EJEMPLO:

Multiplicación encadenada de matrices

- Se desea obtener la multiplicación de $\mathbf{A} \times \mathbf{B} \times \mathbf{C} \times \mathbf{D}$, donde \mathbf{A} es una matriz de 20×2 , \mathbf{B} una matriz de 2×30 , \mathbf{C} una matriz de 30×12 y \mathbf{D} una matriz de 12×8 ...
- *¿Cuántas multiplicaciones escalares habría en la secuencia normal de multiplicación?*
- $\mathbf{A} \times \mathbf{B} = 20 \times 2 \times 30 = 1200 + \dots$
- resultado anterior $\times \mathbf{C} = 20 \times 30 \times 12 = 7200 + \dots$
- resultado anterior $\times \mathbf{D} = 20 \times 12 \times 8 = 1920 \qquad \qquad \qquad = \mathbf{10,320}$
- ¿Habría una forma más OPTIMA?
- **SI !!!** ...

EJEMPLO:

Multiplicación encadenada de matrices

- Se desea obtener la multiplicación de $\mathbf{A} \times \mathbf{B} \times \mathbf{C} \times \mathbf{D}$, donde \mathbf{A} es una matriz de 20×2 , \mathbf{B} una matriz de 2×30 , \mathbf{C} una matriz de 30×12 y \mathbf{D} una matriz de 12×8 ...
- *¿Cuál es el orden de multiplicación que MINIMIZA las multiplicaciones escalares?*
- $\mathbf{B} \times \mathbf{C} = 2 \times 30 \times 12 = 720 + \dots$
- resultado anterior $\times \mathbf{D} = 2 \times 12 \times 8 = 192 + \dots$
- $\mathbf{A} \times \text{resultado anterior} = 20 \times 2 \times 8 = 320$ $= \mathbf{1,232}$
- **Encontrar cómo se obtiene el orden más eficiente de multiplicación ES NUESTRO PROBLEMA...**



Multiplicación encadenada de matrices

- Propuestas de solución:
 - **FUERZA BRUTA:** Encontrar **TODAS** las posibles combinaciones de orden para las multiplicaciones, calcular la cantidad de multiplicaciones escalares, y seleccionar la mínima. *Comportamiento de orden exponencial.*
 - **PROGRAMACIÓN DINÁMICA** si se aplica el principio de optimalidad...
 - Algoritmo propuesto por Godbole (1973) con un comportamiento de orden cúbico.

Diseño del Algoritmo para la Multiplicación encadenada de matrices

- El problema de tamaño n significa que se tienen n matrices a multiplicar: $M_1 \times M_2 \times \dots \times M_n \dots$
- Cada matriz tiene dimensiones $d_{i-1} \times d_i \dots$
- Por lo tanto... M_1 es de tamaño $d_0 \times d_1 \dots M_2 : d_1 \times d_2 \dots M_3 : d_2 \times d_3 \dots$ y así hasta $M_n : d_{n-1} \times d_n \dots$
- Los datos d_i son la entrada para el algoritmo del problema, y NO los datos de las matrices...
- La cantidad de multiplicaciones escalares al multiplicar $M_k \times M_{k+1}$ es igual a $d_{k-1} \times d_k \times d_{k+1} \dots$

Diseño del Algoritmo para la Multiplicación encadenada de matrices

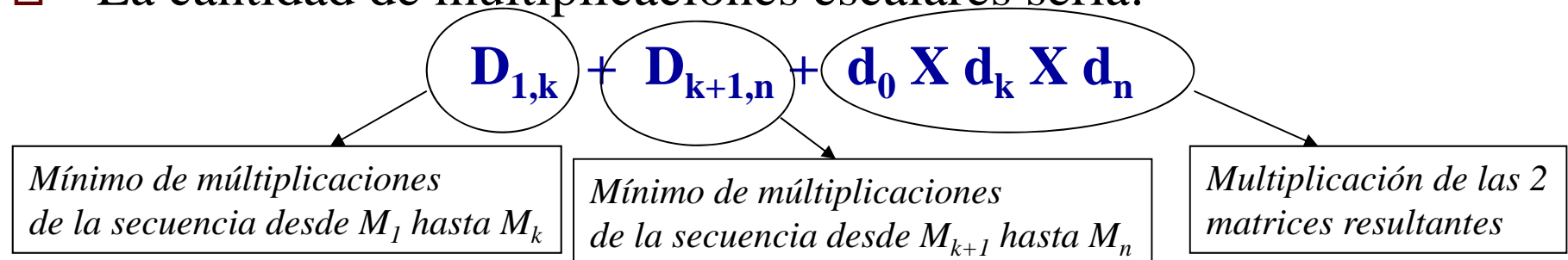
- Sea $D_{i,j}$ la cantidad mínima de multiplicaciones escalares al multiplicar la secuencia de matrices desde M_i hasta M_j ...
- ¿Cómo se puede obtener $D_{i,j}$?
 - $D_{i,j} = 0$ cuando $i = j$...
 - $D_{i,j} = d_{i-1} \times d_i \times d_{i+1}$ cuando $i+1 = j$...
- El resto de los casos se tienen que generalizar “**pensando recursivamente**”, y aplicando el **principio de optimalidad** que corresponde a las diferentes formas de agrupamiento de las matrices...

Diseño del Algoritmo para la Multiplicación encadenada de matrices

- Sea una posible agrupación:

$$(M_1 \times M_2 \times \dots \times M_k) \times (M_{k+1} \times M_{k+2} \times \dots \times M_n)$$

- La cantidad de multiplicaciones escalares sería:



- Puesto que se pueden hacer diversas agrupaciones...
 - $D_{1,n}$ sería el mínimo de $D_{1,k} + D_{k+1,n} + d_0 \times d_k \times d_n$ para los valores de k desde 1 hasta $n-1$...

Diseño del Algoritmo para la Multiplicación encadenada de matrices

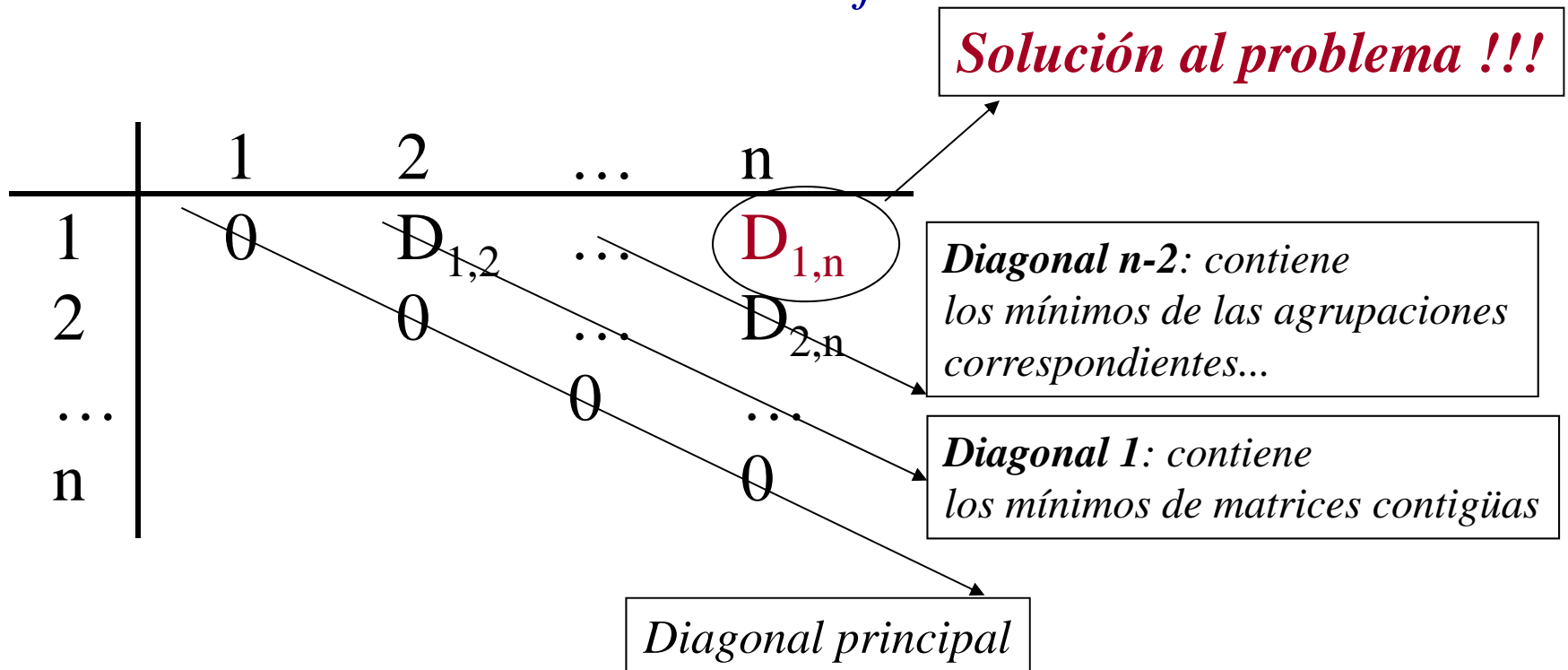
- Por lo tanto, generalizando...

$$D_{i,j} = \text{mínimo} (D_{i,k} + D_{k+1,j} + d_{i-1} \times d_k \times d_j) \text{ para } i \leq k \leq j-1$$

- El valor que resuelve el problema es $D_{1,n} \dots$
- Y según la técnica de la programación dinámica, éste se puede obtener a partir de los valores de base que serían las $D_{i,i} \dots$
- Dado que $D_{i,j}$ es un espacio matricial... algorítmicamente hablando se trabajará con una matriz $D \dots$

Diseño del Algoritmo para la Multiplicación encadenada de matrices

$$D_{i,j} = \text{mínimo} (D_{i,k} + D_{k+1,j} + d_{i-1} \times d_k \times d_j) \text{ para } i \leq k \leq j-1$$



Algoritmo para la Multiplicación encadenada de matrices

```
for (int i=1; i<=n; i++) D[i,i] = 0;
for (int diag=1; i<= n-1 do
    for i = 1 to n-diag do
        j = i + diag;
        D[i,j] = minimo(i, j, D, d); /*Esta función calcula
            el valor mínimo entre los diversos valores de:
            D[i,k] + D[k+1, j] + d[i-1]*d[k]*d[j]
            para k desde i hasta j-1 */
return D[1,n];
```

$O(n^3)$



Algoritmo para la Multiplicación encadenada de matrices

- ❑ *¿Cómo obtener la agrupación más eficiente además del valor de las multiplicaciones escalares mínimas?*
- ❑ Utilizar una matriz auxiliar que guarde la última matriz utilizada en la agrupación más óptima (similar al caso del camino más corto)...
- ❑ Con esta matriz, una rutina recursiva puede desplegar la agrupación más eficiente...
- ❑ *Ver capítulo 3, sección 3.4 para los detalles...*

Árbol Binario de Búsqueda (ABB) Óptimo

Un ejemplo más...

ABB óptimo

Recordando...

- ❑ Un Arbol Binario de Búsqueda (ABB) es una estructura de datos útil para realizar la búsqueda de datos...
- ❑ Los nodos tienen 0, 1 ó 2 hijos...
- ❑ Los descendientes por la izquierda son menores, los de la derecha son mayores...
- ❑ La altura del árbol determina el peor caso en la búsqueda...

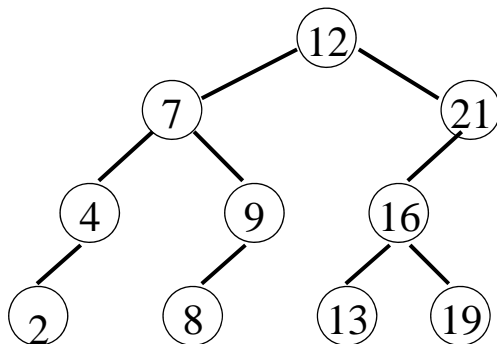
ABB óptimo

PROBLEMA:

- dada una secuencia ordenada de llaves a insertar en un ABB... d_1, d_2, \dots, d_n ($d_1 < d_2 < \dots < d_n$)
- de las cuales se conoce la probabilidad de que sean buscadas en el ABB... p_1, p_2, \dots, p_n
- ¿cuál es la forma del ABB que **minimiza** el tiempo promedio de búsqueda de las llaves?

¿Cómo se calcula el tiempo promedio de búsqueda en un ABB?

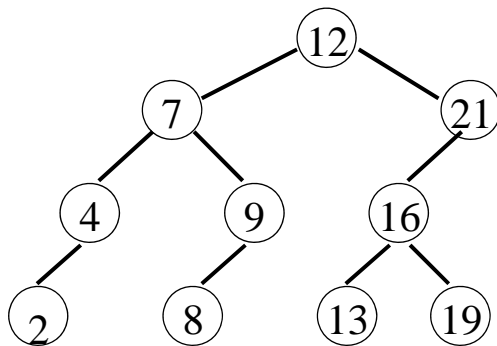
- Si todas las llaves tienen la misma probabilidad de ser buscadas...
- El tiempo promedio de búsqueda sería...
 - La sumatoria de la cantidad de comparaciones que se requieren para encontrar cada llave...
 - entre la cantidad de llaves...



$$\begin{aligned} &1 + \\ &2 + 2 + \\ &3 + 3 + 3 + \\ &4 + 4 + 4 + 4 = 30/10 = \mathbf{3.0} \end{aligned}$$

¿Cómo se calcula el tiempo promedio de búsqueda en un ABB?

- Si cada una de las llaves tiene una probabilidad de ser buscada...
- El tiempo promedio de búsqueda sería...
 - La sumatoria de multiplicar la cantidad de comparaciones que se requieren para encontrar cada llave por su probabilidad de ser buscada...



$$\begin{aligned} &1(p_6) + \\ &2(p_3) + 2(p_{10}) + \\ &3(p_2) + 3(p_5) + 3(p_8) + \\ &4(p_1) + 4(p_4) + 4(p_7) + 4(p_9) \end{aligned}$$

ABB óptimo

¿Qué se busca optimizar?

- Para una secuencia de llaves, con sus probabilidades de ser buscadas, existen diversos ABB que las pueden guardar...
 - *¿Cuál es el ABB que minimiza el tiempo promedio de búsqueda?*
- **Ejemplo:** Para un ABB con 3 llaves, ¿Cuáles son los posibles árboles a formar?

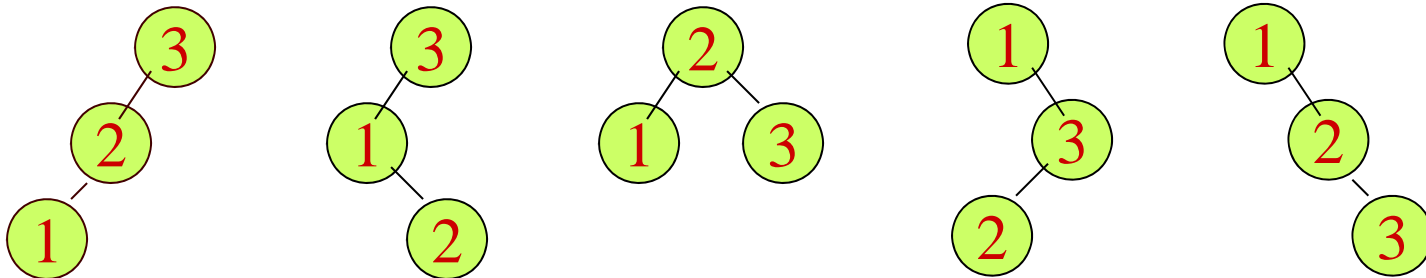
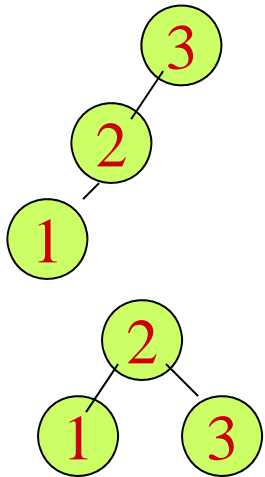


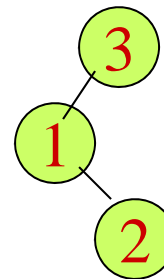
ABB óptimo

¿Qué se busca optimizar?

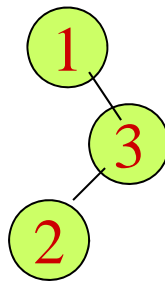
- Si las probabilidades de que se busque cada una de las llaves son: $p_1 = 0.7$, $p_2 = 0.2$ y $p_3 = 0.1$...
- ¿Cuál es el tiempo promedio de búsqueda en cada árbol?



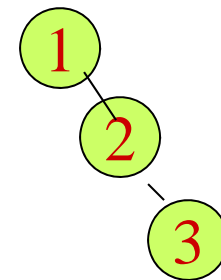
$$1(0.1) + 2(0.2) + 3(0.7) = 2.6$$



$$1(0.1) + 2(0.7) + 3(0.2) = 2.1$$



$$1(0.7) + 2(0.1) + 3(0.2) = 1.5$$



$$1(0.2) + 2(0.7) + 2(0.1) = 1.8$$

$$1(0.7) + 2(0.2) + 3(0.1) = \mathbf{1.4}$$



¿Cómo obtener el ABB óptimo?

- Calcular todas las posibles opciones de formas de ABB para una secuencia de llaves, y después encontrar la del tiempo promedio mínimo...
 - Tiene un comportamiento exponencial...
- ¿Se podrá utilizar la *PROGRAMACIÓN DINÁMICA*?
 - Es un problema de optimización...
 - Y se aplica el principio de optimalidad, pues para el ABB del tiempo mínimo promedio, se tienen subárboles con el tiempo mínimo promedio...

ABB óptimo

Aplicando la programación dinámica

- Sea $A_{i,j}$ el tiempo mínimo promedio para la búsqueda de cualquier llave de la secuencia de d_i a d_j ...
- Se desea encontrar $A_{1,n}$ que es la solución al problema...
- $A_{i,i} = p_i$ ya que estamos hablando de ABB's de un sólo nodo, y por lo tanto se requieren en promedio $1 \times p_i$ comparaciones...
- Busquemos ahora la solución general...

ABB óptimo

Caso general

- Sea el ABB óptimo 'k' aquel cuya raíz es la llave d_k ...
- entonces, los subárboles del nodo raíz d_k , son ABB óptimos que tienen el tiempo mínimo promedio en sus búsquedas...
 - el subárbol izquierdo tiene a las llaves d_1 a d_{k-1} ...
 - y por lo tanto, el valor del tiempo mínimo promedio para el subárbol izquierdo es $A_{1,k-1}$...
 - el subárbol derecho tiene a las llaves d_{k+1} a d_n ...
 - y por lo tanto, el valor del tiempo mínimo promedio para el subárbol derecho es $A_{k+1,n}$...

ABB óptimo

Caso general

- *¿Cuál es el tiempo mínimo óptimo para el ABB óptimo 'k' cuya raíz es la llave d_k ?*
 - La raíz del árbol, requiere una comparación para p_k , más...
 - El tiempo mínimo promedio de los subárboles.
- Pero dado que los subárboles están en un nivel más abajo que la raíz...
 - buscar a cualquier llave que no sea la raíz del árbol, involucra una comparación más...
 - y esto indica que debemos acumular la probabilidad correspondiente...

ABB óptimo

Caso general

Por lo tanto y generalizando...

- ¿Cuál es el tiempo mínimo óptimo para el ABB óptimo 'k' cuya raíz es la llave d_k ?

$$A_{1,k-1} + (p_1 + p_2 + \dots + p_{k-1}) + p_k + A_{k+1,n} + (p_{k+1} + \dots + p_n)$$

- que equivale a: $A_{1,k-1} + A_{k+1,n} + \sum_{m=1}^n p_m$

- y para $A_{1,n} = \text{mínimo} (A_{1,k-1} + A_{k+1,n}^m)^l + \sum_{m=1}^n p_m$

- donde $A_{i,i-1}$ y $A_{j+1,j}$ valen 0
 $1 \leq k \leq n$

Algoritmo para obtener el ABB óptimo

- Propuesto por Gilbert and Moore (1959).
- **ENTRADAS:**
 - cantidad de llaves n
 - probabilidades de que sean buscadas cada una de las llaves p_i
- **SALIDAS:**
 - Tiempo mínimo promedio para la búsqueda de llaves (en el ABB óptimo).
 - Matriz **R** ($1..n+1 \times 0..n$) donde **R**[**i,j**] contiene el índice de la llave que es la raíz en el ABB óptimo que contiene a las llaves desde i hasta j . Esta matriz servirá para construir el ABB óptimo si se requiere.

Algoritmo para obtener el ABB óptimo

```
for (int i=1; i<=n; i++) // inicialización de matrices de resultados
```

```
{ A[i,i-1] = 0; A[i,i] = p[i];
```

```
  R[i,i] = i; R[i,i-1] = 0; }
```

```
A[n+1,n] = 0; R[n+1,n] = 0;
```

```
for (int diag=1; i<= n-1; i++)
```

```
  for (int i = 1; i<=n-diag; i++)
```

```
    { j = i + diag;
```

```
      A[i,j] = minimo(i, j, A) + sumatoria(i, j, p); }
```

```
/*La función mínimo calcula el valor mínimo entre los diversos valores de:
```

```
A[i,k-1] + A[k+1, j] para k desde i hasta j. La función sumatoria calcula la  
suma de las probabilidades de la llave i hasta la llave j.*/ }
```

```
return A[1,n];
```

$O(n^3)$

Construcción del ABB óptimo

□ Dada la matriz R...

función ABB (i,j) : apuntador;

k = R[i,j];

if (k = 0) return NULL;

else

{ q = new nodo(llave[k]);

q->izq = ABB(i,k-1);

q->der = ABB(k+1,j);

return q; }

El problema del viajero



Problema...

- ❑ Imagina que tú y otra persona, han pedido trabajo en cierta empresa...
- ❑ Quién los va a contratar, les dice que el contrato será para quien le indique cuál es la ruta más óptima para visitar 20 ciudades distintas del país, pasando solamente una vez por cada una de ellas, en el menor tiempo posible (y al menor costo)...
- ❑ Las 20 ciudades, están conectadas por vías de transporte con todas las restantes...
- ❑ Este es el **problema del viajero**... ¿cómo solucionarlo?



Problema...

- La otra persona, tiene experiencia programando, y ha decidido poner a su computadora a trabajar para encontrar esta ruta óptima...
- Él implementa un programa que encontrará todos los posibles caminos desde la ciudad inicial, pasando por todas las ciudades, hasta llegar de nuevo a la ciudad inicial y calculará para cada uno de ellos sus costos... de ahí obtendrá el menor, para dar su respuesta...
 - ¿Cuánto se tardará la computadora en darle el resultado?...

Problema...

- ¿Cuántos caminos posibles hay en las 20 ciudades?
 - $19! = 121,645,100,408,832,000$
- Si el cálculo de la longitud de cada camino se tarda 1 microsegundo en la computadora....
 - El resultado tardaría **3,857 años!!!**
- Pero tú, que conoces algunas técnicas de diseño de algoritmos, al observar que es un problema de optimización, decides explorar la posibilidad de utilizar a la **PROGRAMACIÓN DINÁMICA** para resolverlo...

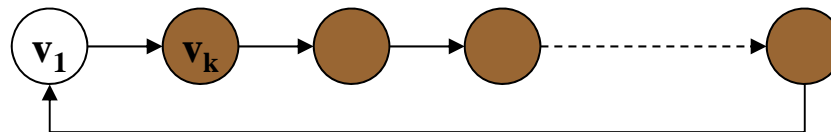


Propuesta de solución...

- El problema se puede modelar con un grafo ponderado en el que cada vértice será una ciudad, y los arcos tendrán los costos que implica viajar de una ciudad a otra...
- El problema generará como resultado la ruta más óptima que incluye a todos los vértices del grafo (Ciclo Hamiltoniano)...
- Puesto que en un ciclo, lo importante es empezar y terminar en el mismo vértice, para el análisis de este problema, es irrelevante el vértice inicial, por lo que se tomará el primer vértice como base (v_1)...

Propuesta de solución...

- Sea v_k el siguiente vértice a visitar después de v_1 en la ruta más óptima...
 - El camino de v_k a v_1 será el más corto que pase una vez por todos los vértices restantes... y por lo tanto el más óptimo...

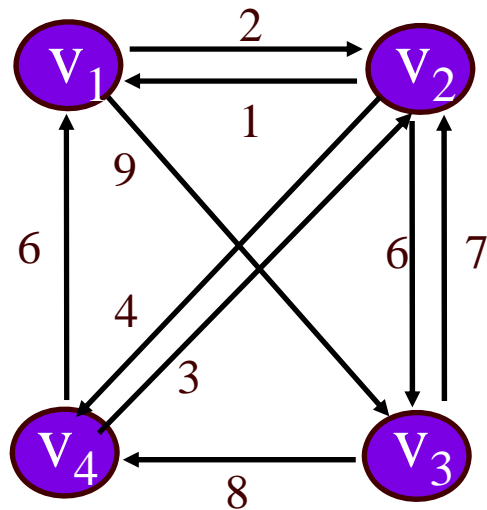


- Esto significa, que para resolver el ciclo completo, se tiene un subproblema, que cumplirá la condición de ser el más óptimo para poder solucionar el caso general...
- Por lo tanto, **SÍ** se puede aplicar la **PROGRAMACIÓN DINÁMICA**....

Propuesta de solución...

- Sea W la matriz de adyacencias del grafo correspondiente...
- Sea V el conjunto de todos los vértices del grafo...
- Sea A un subconjunto de V ...
- Entonces $D[v_i, A]$ es la longitud del camino más corto de v_i hasta v_1 pasando una vez por todos los vértices de A ...

Ejemplo...



□ $V = \{v_1, v_2, v_3, v_4\}$

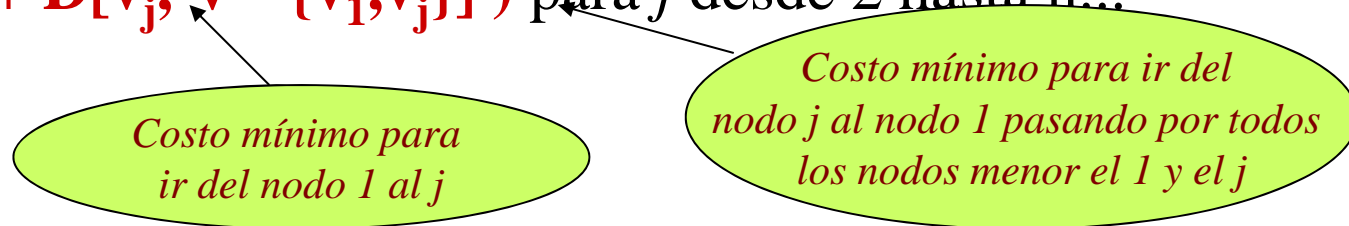
□ Si $A = \{v_3\}...$

□ $D[v_2, A] = \text{longitud}[v_2, v_3, v_1]$
 $= \infty$

- Si $A = \{v_3, v_4\}...$
- $D[v_2, A] = \text{mínimo}(\text{longitud}[v_2, v_3, v_4, v_1], \text{longitud}[v_2, v_4, v_3, v_1])$
 $= \text{mínimo}(20, \infty) = 20$

Continuando con el análisis...

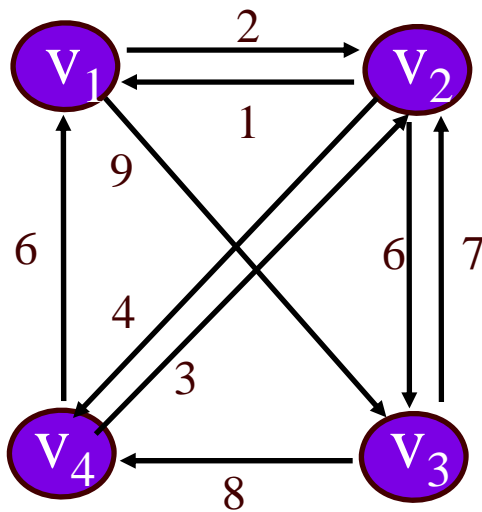
- Para un grafo de n vértices...
 - $D[v_i, \emptyset]$ es la longitud del camino de v_i a v_1 sin pasar por algún vértice...
 - Por lo tanto: $D[v_i, \emptyset] = W[i, 1]$
- Por otro lado, el resultado que se desea obtener es: **mínimo**($W[1, j] + D[v_j, V - \{v_1, v_j\}]$) para j desde 2 hasta n ...



Continuando con el análisis...

- Y generalizando...
- Para todos los vértices excepto el 1, y cuando el vértice_i no se encuentre en A...
 - **$D[v_i, A] = \text{mínimo}(W[i, j] + D[v_j, A - \{v_j\}])$**
para todos los v_j que se encuentre en A...
- Esta es la clave del algoritmo...

En el ejemplo...



□ $D[v_2, \emptyset] = 1$, $D[v_3, \emptyset] = \infty$,
 $D[v_4, \emptyset] = 6$

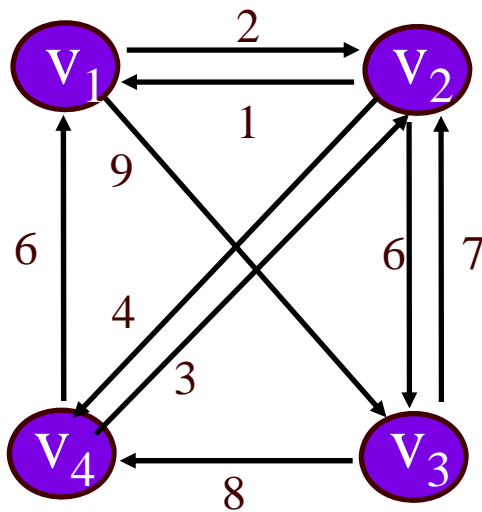
□ Para A de un elemento:

- $D[v_3, \{v_2\}] = \text{minimo}(W[3,j] + D[v_j, \{v_2\} - \{v_j\}])$ para v_j en $\{v_2\}$
- $W[3,2] + D[v_2, \emptyset] = 7 + 1 = 8$

- $D[v_4, \{v_2\}] = 3 + 1 = 4$
- $D[v_2, \{v_3\}] = 6 + \infty = \infty$
- $D[v_4, \{v_3\}] = \infty + \infty = \infty$

- $D[v_2, \{v_4\}] = 4 + 6 = 10$
- $D[v_3, \{v_4\}] = 8 + 6 = 14$

En el ejemplo...

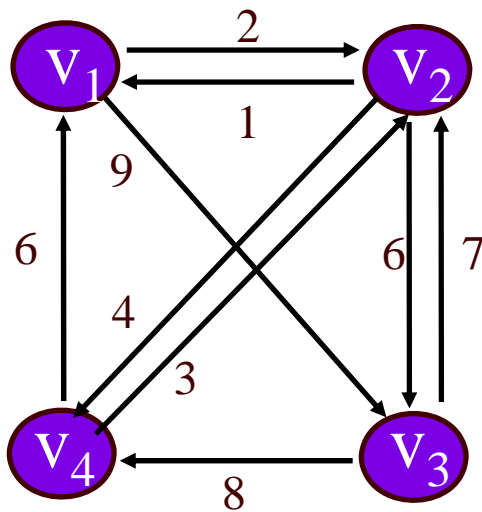


□ Para A de dos elementos:

$$\begin{aligned} D[v_4, \{v_2, v_3\}] &= \text{minimo}(W[4, j] + D[v_j, \{v_2, v_3\} - \{v_j\}]) \text{ para } v_j \text{ en } \{v_2, v_3\} \\ &= \text{mínimo} (W[4, 2] + D[v_2, \{v_3\}], \\ &\quad W[4, 3] + D[v_3, \{v_2\}]) \\ &= \text{minimo}(3 + \infty, \infty + 8) = \infty \end{aligned}$$

- $D[v_3, \{v_2, v_4\}] = \text{minimo}(7 + 10, 8 + 4) = 12$
- $D[v_2, \{v_3, v_4\}] = \text{minimo}(6 + 14, 4 + \infty) = 20$

En el ejemplo...



□ Finalmente, para A de tres elementos:

$$D[v_1, \{v_2, v_3, v_4\}] = \min(W[1, j] + D[v_j, \{v_2, v_3, v_4\} - \{v_j\}]) \text{ para } v_j \text{ en } \{v_2, v_3, v_4\}$$

$$= \min(W[1, 2] + D[v_2, \{v_3, v_4\}], \\ W[1, 3] + D[v_3, \{v_2, v_4\}], \\ W[1, 4] + D[v_4, \{v_2, v_3\}])$$

$$= \min(2 + 20, 9 + 12, \infty + \infty) = 21$$

CAMINO ÓPTIMO: v_1, v_3, v_4, v_2, v_1

Algoritmo del problema del viajero...

for $i=2$ to n do $D[i, \emptyset] = W[i, 1]$;

for $k=1$ to $n-2$ do

*for (todos los subconjuntos A que contengan k
vértices desde v_2 hasta v_n)*

for (todas las $i \neq 1$ y en que v_i no esté en A)

$D[i, A] = \text{minimo}(W[i, j] + D[j, A - \{v_j\}])$;

$D[1, V - \{v_1\}] = \text{minimo}(W[1, j] + D[j, V - \{v_1, v_j\}])$;

return $D[1, V - \{v_1\}]$;

Regresando al inicio...

- Complejidad de tiempo del algoritmo con programación dinámica: $T(n) = (n-1)(n-2)2^{n-3}$
- Que corresponde a un orden: **$O(n^2 2^n)$**
- Para el problema de las 20 ciudades...
 - $19 \times 18 \times 2^{17} = 44,826,624 \dots$
- Que la misma computadora de 1 microsegundo por cálculo, tardaría ...
 - **45 segundos!!!**

*Es un problema
al que no se
le ha encontrado
una solución
más óptima*

Análisis de Algoritmos



Algoritmos voraces
(Greedy algorithms)

Técnicas de diseño de algoritmos

- Divide y vencerás
- Programación dinámica
- **Algoritmos voraces**
- Backtracking
- Branch and bound

Algoritmos voraces

- También conocidos como algoritmos ávidos o miopes...
- Su característica es que toman decisiones basándose en la información que tienen en forma inmediata, sin tener en cuenta los efectos que esto pueda tener en el futuro, es decir, nunca reconsidera su decisión, sea cual fuera la situación que ocurrirá más adelante...
- Se confía que la decisión tomada sea la mejor para la solución general del problema...

Algoritmos voraces

- Utilizados en aplicaciones de optimización...
- Son algoritmos fáciles de diseñar y de implementar...
- pero... **NO** siempre llevan a una solución correcta del problema...
- Cuando si obtienen la solución correcta, lo hacen de una manera eficiente...
- Sin embargo, es difícil demostrar formalmente cuando un algoritmo voraz es correcto o no...

Estructura general de un algoritmo voraz

- Se apoya en un conjunto original de datos (C), y el conjunto resultante que contendrá la solución (S)

$$S = \emptyset$$

Mientras $C \neq \emptyset$ y no se haya encontrado la solución:

$x = \underline{\text{selección}}$ de mejor candidato de C

$$C = C - \{x\}$$

si es factible $S \cup \{x\}$ entonces $S = S \cup \{x\}$

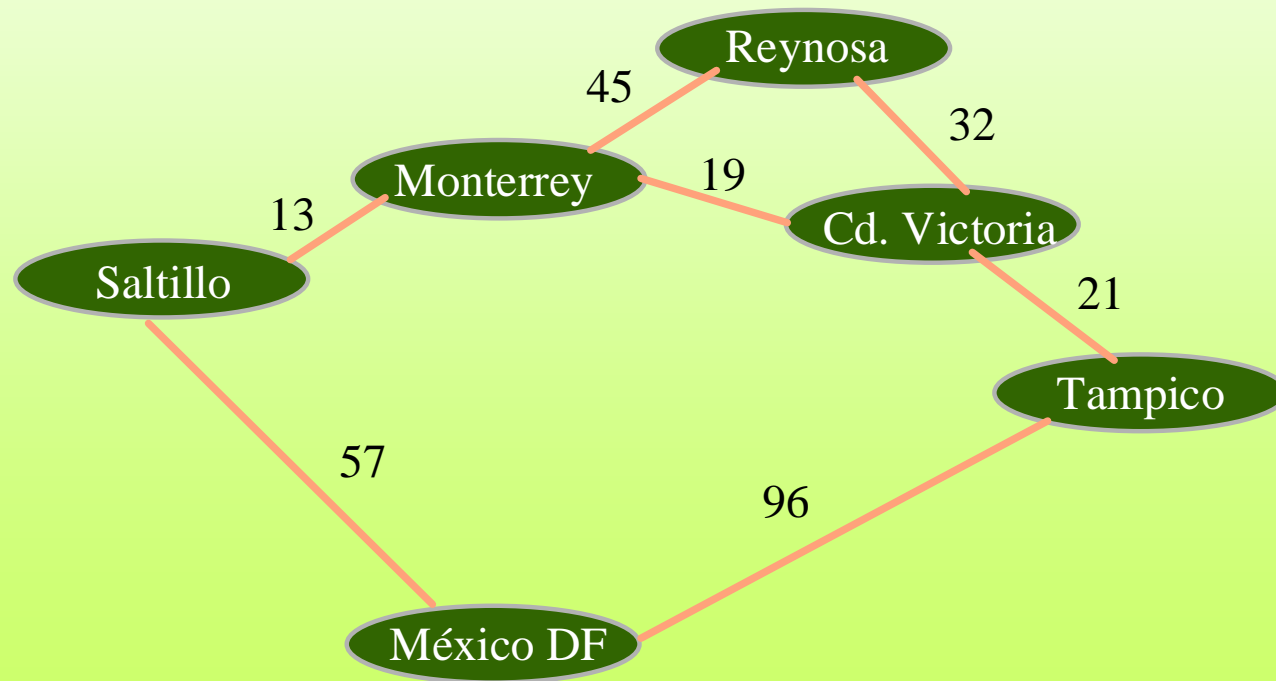
Si S tiene la solución devolver S , sino, no hay solución.

El problema del árbol de extensión mínima

- Dado un grafo no dirigido y ponderado...
- ¿Cuál es el costo MÍNIMO para tener a todos los vértices conectados?
- La solución del problema, implica que se obtenga un subgrafo del grafo original, en el que NO se tengan ciclos...
- por lo tanto, la solución obtiene un ÁRBOL que es llamado de EXTENSIÓN MÍNIMA por la optimización que se realiza.

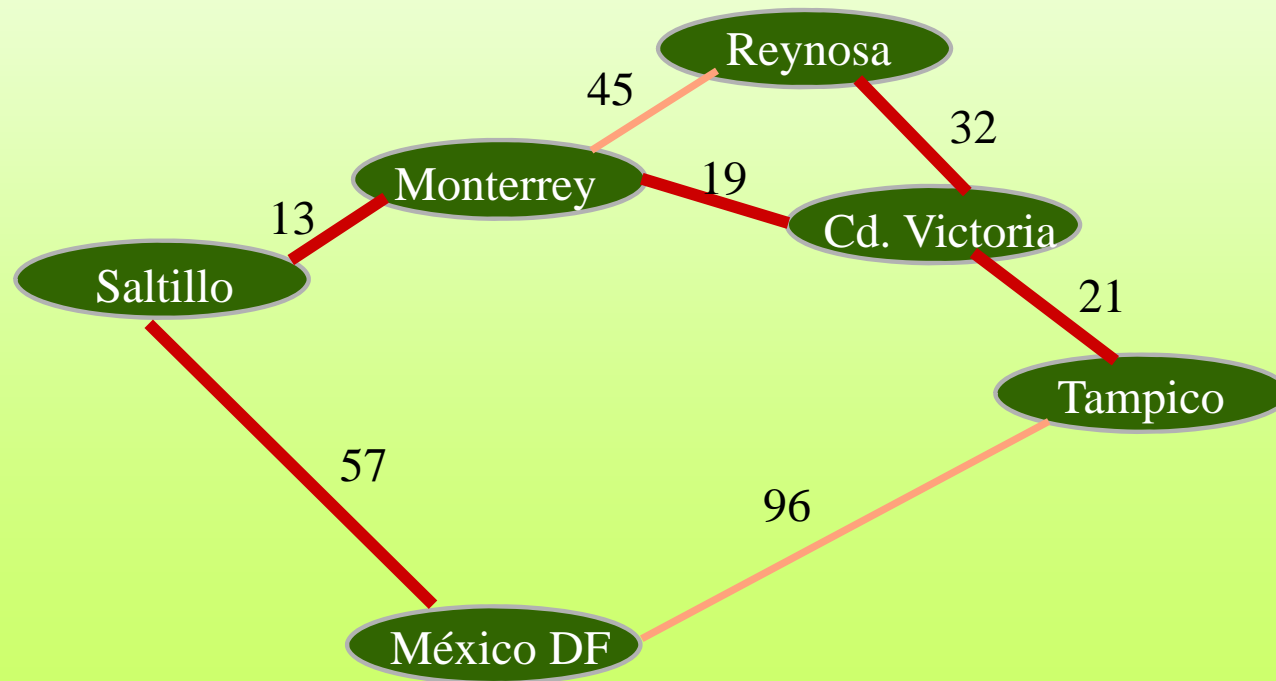
Ejemplo

- ¿Cómo puedo tener conectadas a las siguientes ciudades por un costo mínimo?



Ejemplo

- ¿Cómo puedo tener conectadas a las siguientes ciudades por un costo mínimo?



Algoritmo general para obtener el Árbol de extensión mínima

- Sea el grafo $G = (V, A)$ en donde V es el conjunto de vértices y A el conjunto de arcos de la forma (v_i, v_j) .

$$S = \emptyset$$

Mientras (no se haya resuelto el problema)

Seleccionar un arco de A de acuerdo a cierta política de optimización --> SELECCION

Si al agregar ese arco a S no genera un ciclo en el subgrafo, agregarlo a S --> FACTIBILIDAD

Si (V, S) es el árbol de extensión mínima, el problema se ha resuelto --> VERIFICACIÓN DE LA SOLUCIÓN

*La forma en que se
hace la selección
determina el algoritmo
específico*

Algoritmo de Prim

$$S = \emptyset$$

$$Y = \{v_1\}$$

Mientras (no se haya resuelto el problema)

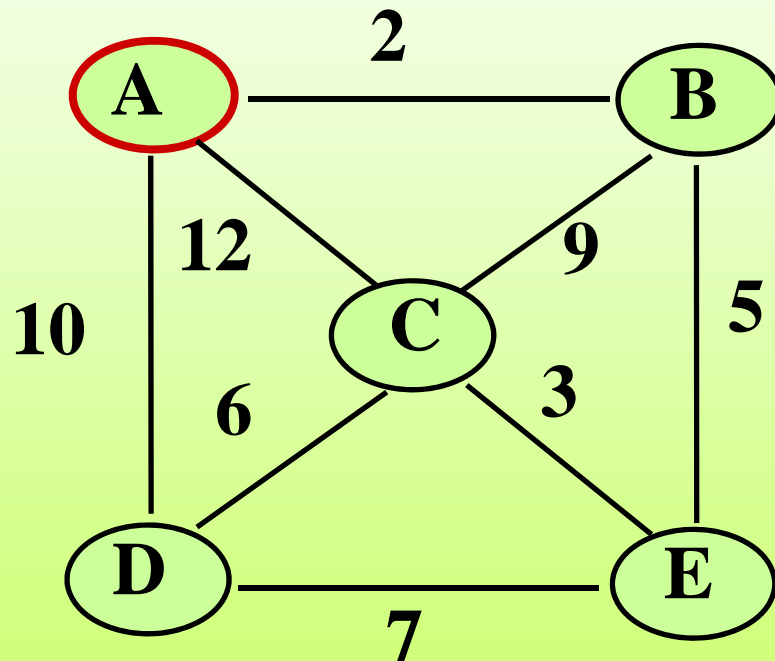
Seleccionar el vértice de Y a $V-Y$ que sea el más cercano (menor peso) a alguno de los vértices en Y .

Agregar el vértice a Y .

Agregar el arco correspondiente a S .

Si $Y = V$ el problema se ha resuelto.

Ejemplo



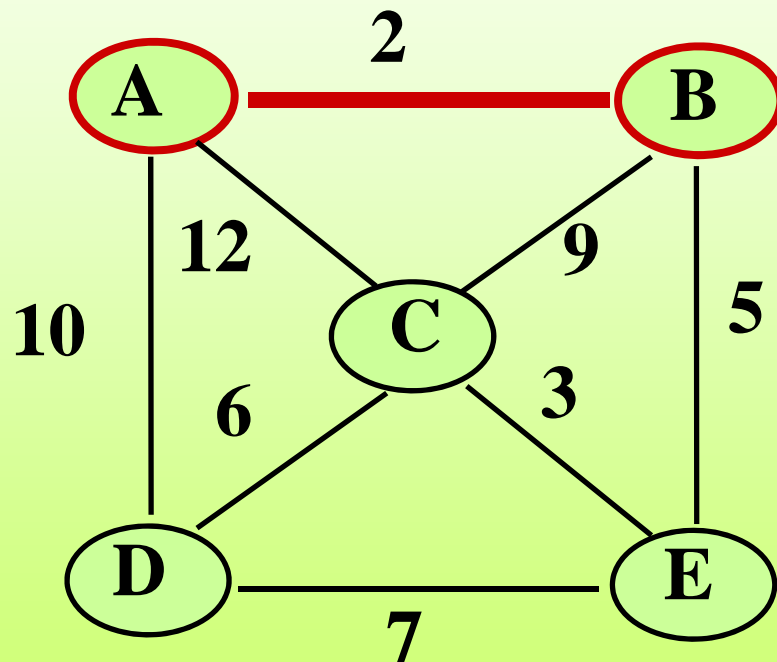
$$S = \emptyset$$

$$Y = \{v_A\}$$

$$V-Y = \{v_B, v_C, v_D, v_E\}$$

*Seleccionar el arco
de menor costo de
 Y a $V-Y$*

Ejemplo



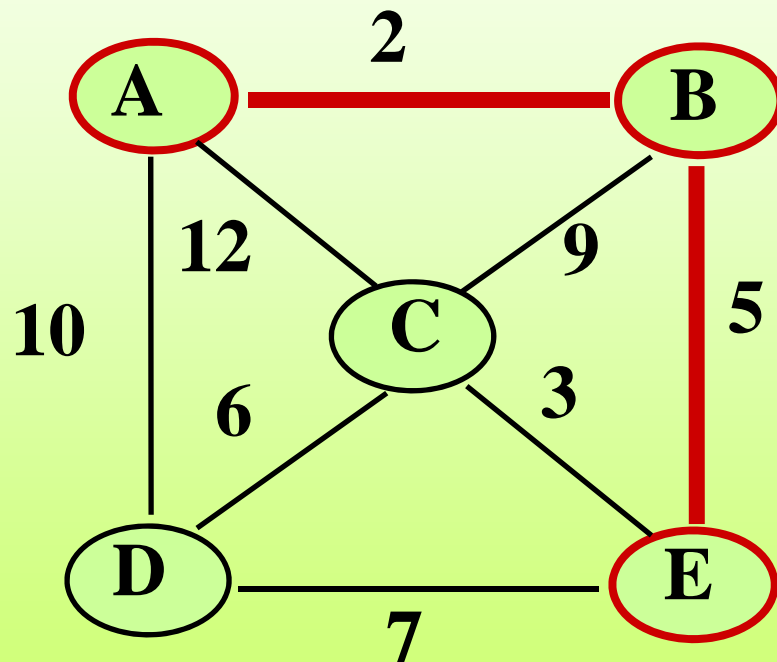
$$S = \{(v_A, v_B)\}$$

$$Y = \{v_A, v_B\}$$

$$V-Y = \{v_C, v_D, v_E\}$$

*Seleccionar el arco
de menor costo de
 Y a $V-Y$*

Ejemplo



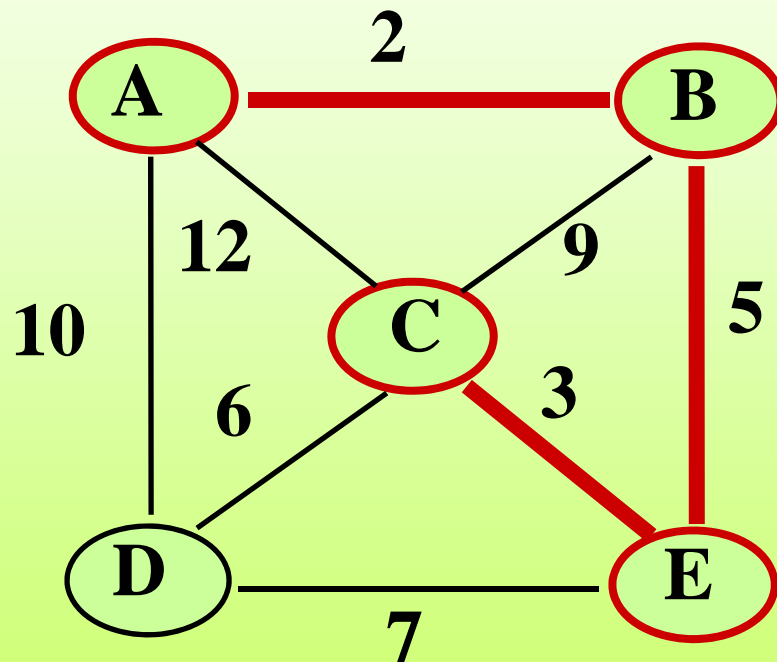
$$S = \{(v_A, v_B), (v_B, v_E)\}$$

$$Y = \{v_A, v_B, v_E\}$$

$$V-Y = \{v_C, v_D\}$$

*Seleccionar el arco de
menor costo de Y a
V-Y*

Ejemplo



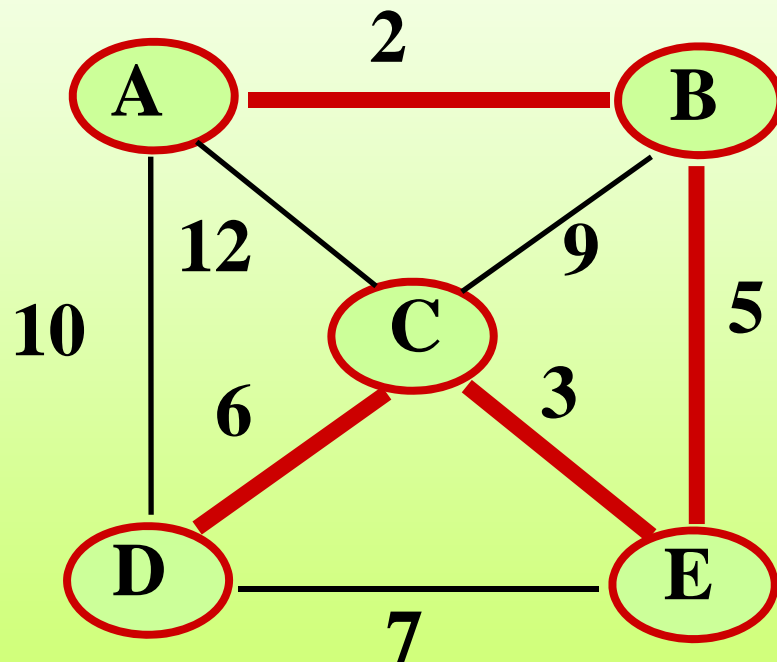
$$S = \{(v_A, v_B), (v_B, v_E), (v_E, v_C)\}$$

$$Y = \{v_A, v_B, v_E, v_C\}$$

$$V - Y = \{v_D\}$$

Seleccionar el arco de menor costo de Y a V-Y

Ejemplo



$$S = \{(v_A, v_B), (v_B, v_E), \\ (v_E, v_C), (v_C, v_D)\}$$

$$Y = \{v_A, v_B, v_C, v_D, v_E\}$$

$$V - Y = \emptyset$$

*Puesto que Y es igual a
 V , se ha encontrado
la solución*

Algoritmo de Kruskal

$S = \emptyset$

$Y =$ subconjuntos disjuntos de V , cada uno con cada uno de los vértices de V

Ordenar los arcos en A en forma ascendente de acuerdo a su peso.

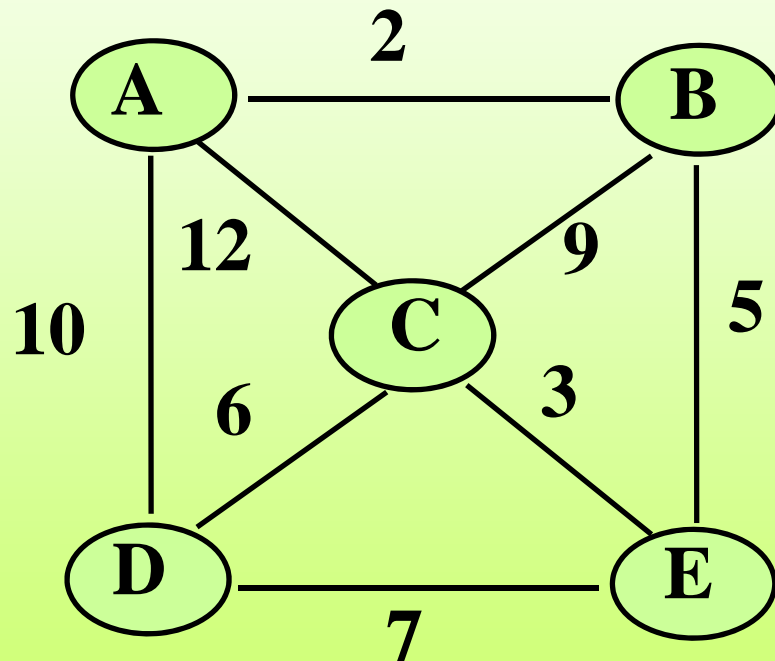
Mientras (no se haya resuelto el problema)

 Seleccionar el siguiente arco de A .

 Si el arco conecta 2 vértices de Y , unir los subconjuntos y añadir el arco a S .

 Si todos los subconjuntos se han unido, el problema se ha resuelto.

Ejemplo



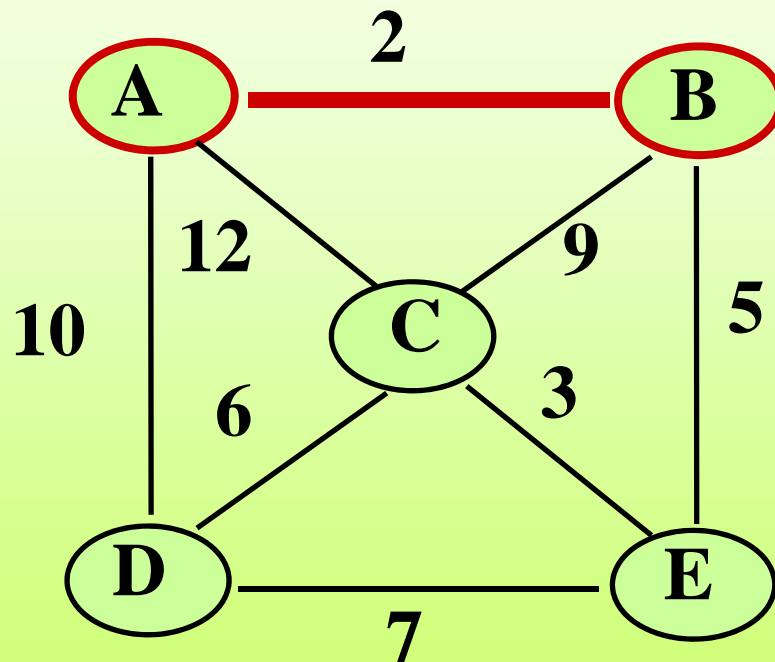
$$S = \emptyset$$

$$Y = \{\{v_A\} \{v_B\} \{v_C\} \{v_D\} \{v_E\}\}$$

$$A = \{(v_A, v_B), (v_C, v_E), (v_B, v_E), (v_C, v_D), (v_D, v_E), (v_B, v_C), (v_A, v_D), (v_A, v_C)\}$$

Seleccionar el arco de menor costo de A y si une a dos subconjuntos de Y, unirlos y agregar el arco a S

Ejemplo



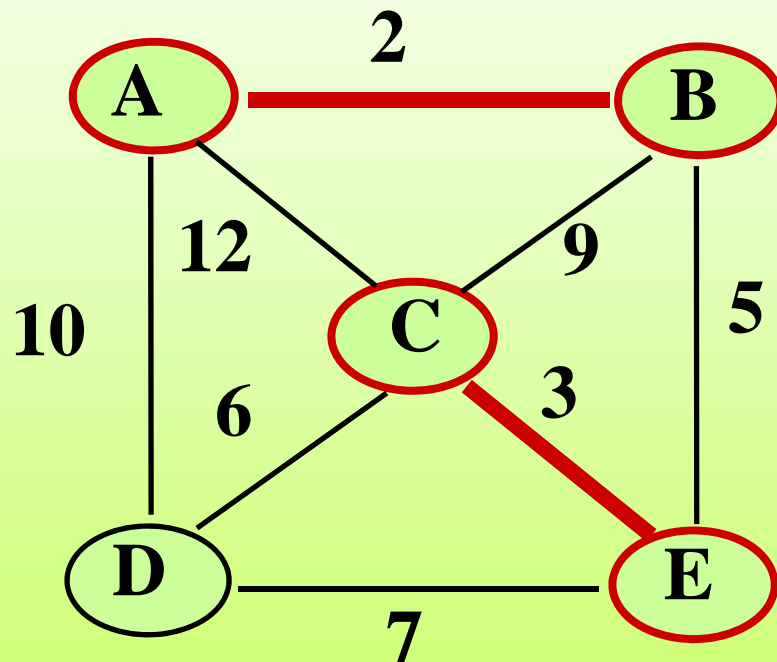
$$S = \{(v_A, v_B)\}$$

$$Y = \{\{v_A, v_B\} \{v_C\} \{v_D\} \{v_E\}\}$$

$$A = \{(v_C, v_E), (v_B, v_E), (v_C, v_D), (v_D, v_E), (v_B, v_C), (v_A, v_D), (v_A, v_C)\}$$

Seleccionar el arco de menor costo de A y si une a dos subconjuntos de Y, unirlos y agregar el arco a S

Ejemplo



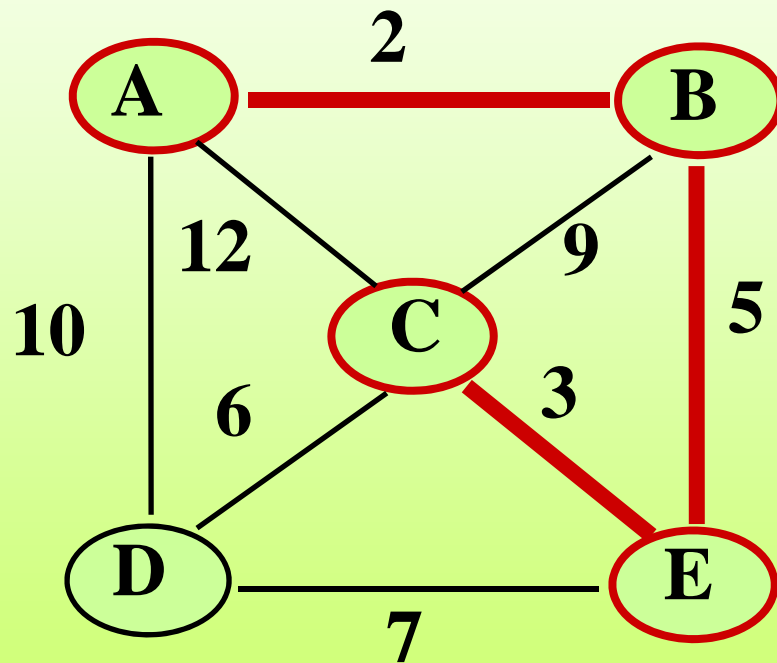
$$S = \{(v_A, v_B), (v_C, v_E)\}$$

$$Y = \{\{v_A, v_B\}, \{v_C, v_E\}, \{v_D\}\}$$

$$A = \{(v_B, v_E), (v_C, v_D), (v_D, v_E), (v_B, v_C), (v_A, v_D), (v_A, v_C)\}$$

Seleccionar el arco de menor costo de A y si une a dos subconjuntos de Y, unirlos y agregar el arco a S

Ejemplo



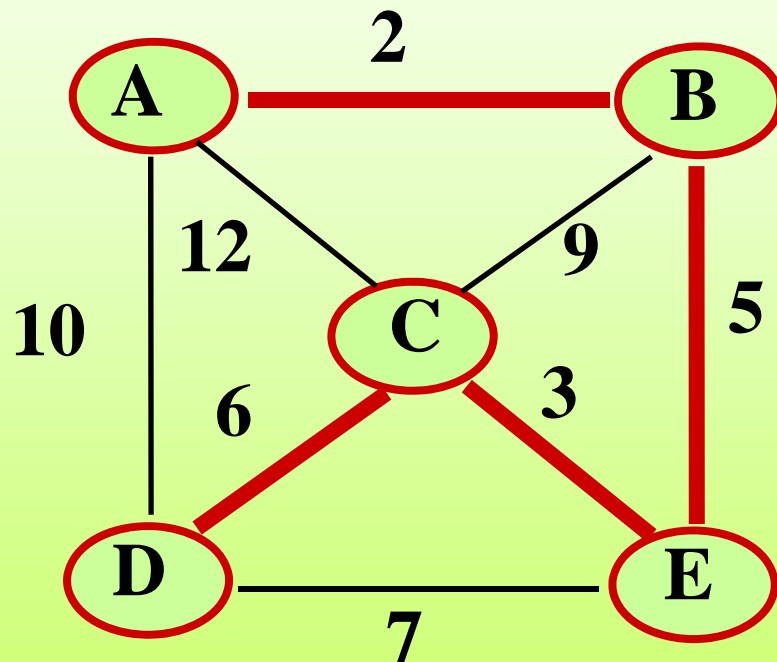
$$S = \{(v_A, v_B), (v_C, v_E), (v_B, v_E)\}$$

$$Y = \{\{v_A, v_B, v_C, v_E\}, \{v_D\}\}$$

$$A = \{(v_C, v_D), (v_D, v_E), (v_B, v_C), (v_A, v_D), (v_A, v_C)\}$$

Seleccionar el arco de menor costo de A y si une a dos subconjuntos de Y, unirlos y agregar el arco a S

Ejemplo



$$S = \{(v_A, v_B), (v_C, v_E), (v_B, v_E), (v_C, v_D)\}$$

$$Y = \{\{v_A, v_B, v_C, v_D, v_E\}\}$$

$$A = \{(v_D, v_E), (v_B, v_C), (v_A, v_D), (v_A, v_C)\}$$

Puesto que Y sólo contiene un subconjunto con todos los vértices, S contiene la solución.

Implementación de los algoritmos

- Requieren de un tipo de dato conjunto, que permita trabajar con las operaciones de conjuntos (unión, diferencia, añadir).
- **Prim** se apoya en la matriz de transiciones, y en arreglos auxiliares (*ver detalle en libro*).
- **Kruskal** requiere de la implementación del tipo de dato conjunto disjunto (*ver detalle en libro*).
- Los **HEAPS** pueden ayudar a obtener otras versiones eficientes de implementación.

¿Cómo se comprueba que los algoritmos son correctos?

- En el caso de la **programación dinámica**, basta comprobar que se cumple el principio de optimalidad para saber que se tiene un algoritmo válido...
- En el caso de **divide y vencerás**, la recursividad está fundamentada, y comprueba la validez del algoritmo...
- En el caso de **algoritmos voraces**, se requiere una demostración matemática específica dependiendo del problema, y normalmente es más compleja (*ver demostración de Prim y Kruskal en libro*).

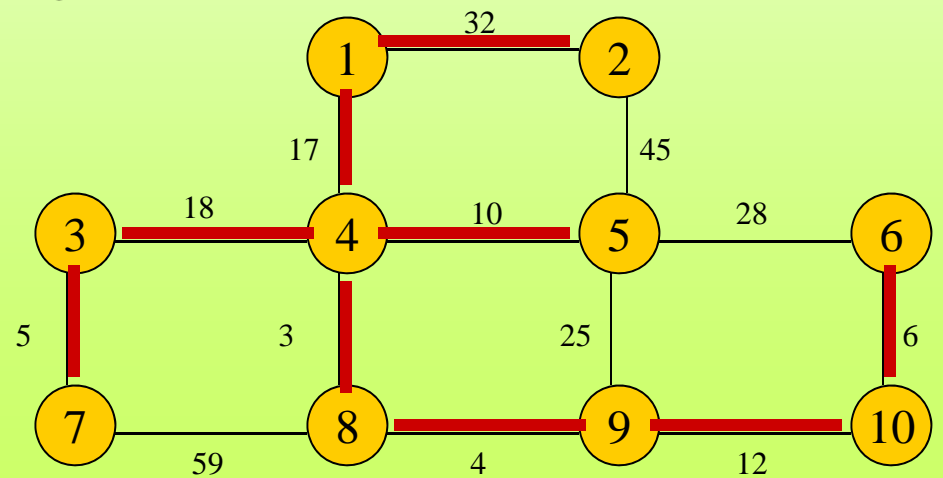
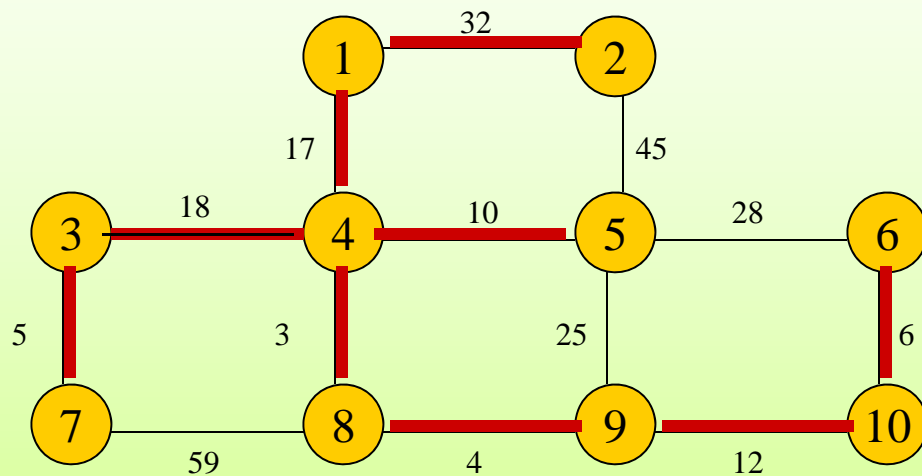
¿Cómo es el comportamiento de los algoritmos?

- Sea n la cantidad de vértices en el grafo y m la cantidad de arcos.
- **Prim:**
 - Comportamiento igual para todos los casos
 - Complejidad de tiempo: $O(n^2)$ *(ver detalle en el libro)*.
- **Kruskal:**
 - Comportamiento distinto, depende de la cantidad de arcos
 - Complejidad de tiempo: $O(m \log_2 m)$ *(ver detalle en el libro)*.

¿Cuál de los dos algoritmos es mejor utilizar?

- ¿Cuántos arcos puede tener un grafo no dirigido conectado de n vértices?
 - MINIMO: $m = n - 1$
 - MAXIMO: $m = n(n - 1) / 2$
- Dado que: ***Prim $O(n^2)$ vs. Kruskal $O(m \log_2 m)$***
 - Grafo con pocos arcos: **Kruskal** es más eficiente ($O(n \log_2 n)$ vs. $O(n^2)$)
 - Grafo muy denso (altamente conectado): **Prim** es mejor ($O(n^2)$ vs. $O(n^2 \log_2 n)$)

EJEMPLO



El problema del camino más corto

- ¿Qué pasaría si en determinada aplicación se requiere conocer el camino más corto de un vértice a otro?
- El *Algoritmo de Floyd* obtiene el camino más corto de TODOS los vértices hacia TODOS los vértices y tiene un comportamiento de $O(n^3)$...
- Si sólo se requiere el análisis para un sólo camino, ¿podría hacerse de una manera más eficiente?...
- SI... la propuesta del *Algoritmo de Dijkstra* resuelve el problema en $O(n^2)$.

Algoritmo de Dijkstra

$$S = \emptyset$$

$$Y = \{v_1\}$$

Mientras (no se haya resuelto el problema)

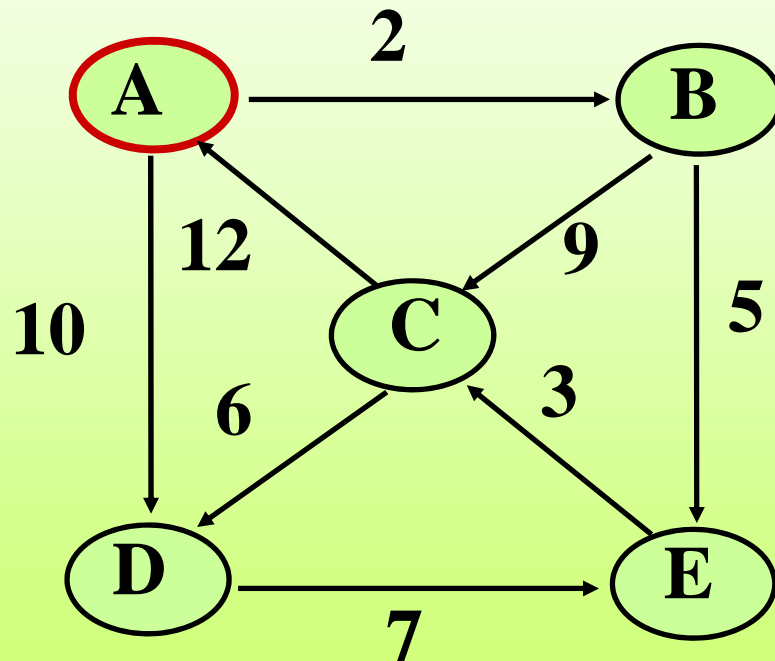
*Seleccionar el vértice de Y a $V-Y$ que tenga **el camino más corto** desde v_1 usando sólo a los vértices en Y como intermediarios.*

Agregar el vértice a Y .

Agregar el arco que llega a al vértice seleccionado a S .

Si $Y = V$ el problema se ha resuelto.

Ejemplo



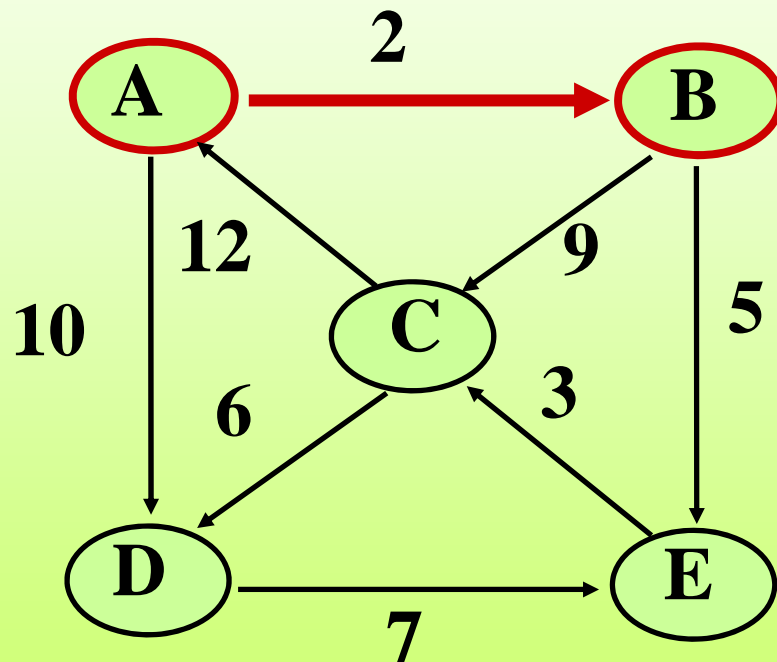
$$S = \emptyset$$

$$Y = \{v_A\}$$

$$V-Y = \{v_B, v_C, v_D, v_E\}$$

Seleccionar el vértice de $V-Y$ que tenga el camino más corto desde v_A , pasando por los vértices de Y

Ejemplo



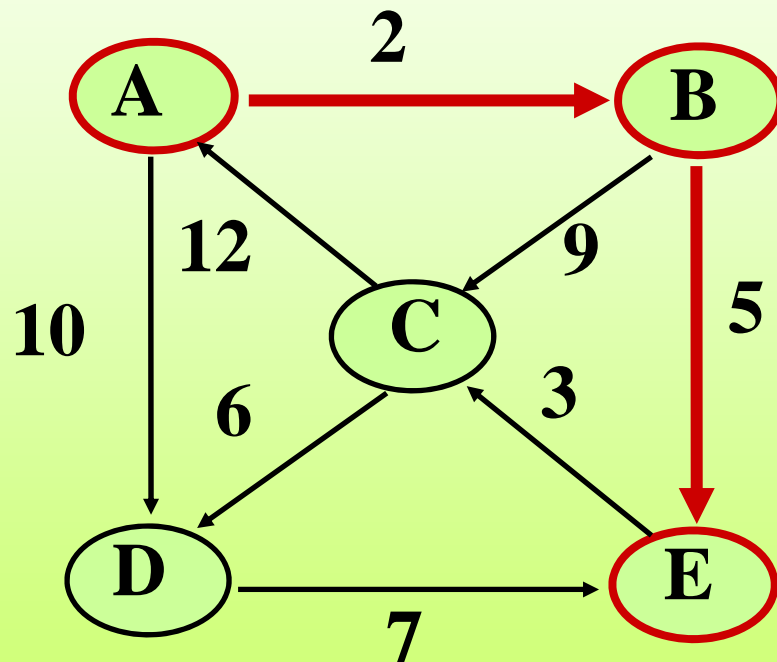
$$S = \{(v_A, v_B)\}$$

$$Y = \{v_A, v_B\}$$

$$V-Y = \{v_C, v_D, v_E\}$$

*Seleccionar el vértice de
 $V-Y$ que tenga el
camino más corto
desde v_A , pasando por
los vértices de Y*

Ejemplo



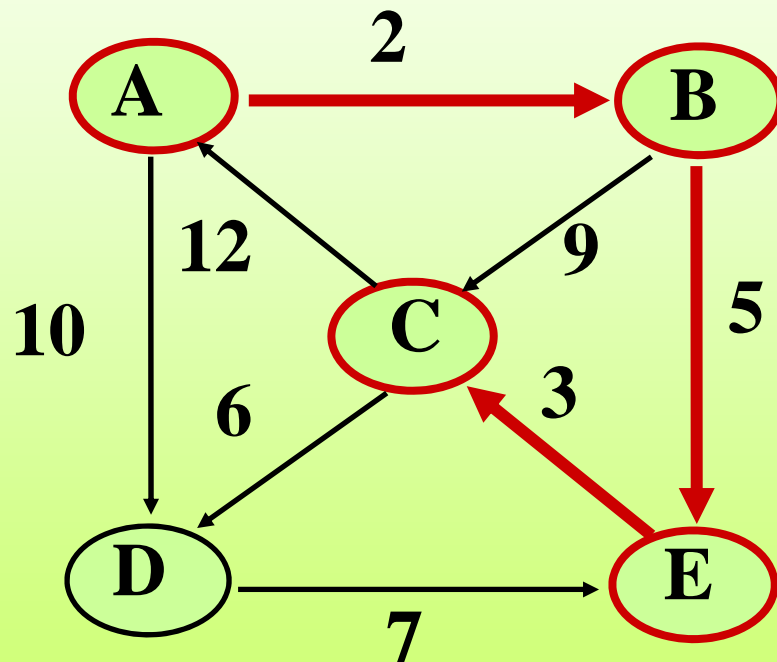
$$S = \{(v_A, v_B), (v_B, v_E)\}$$

$$Y = \{v_A, v_B, v_E\}$$

$$V-Y = \{v_C, v_D\}$$

Seleccionar el vértice de $V-Y$ que tenga el camino más corto desde v_A , pasando por los vértices de Y

Ejemplo



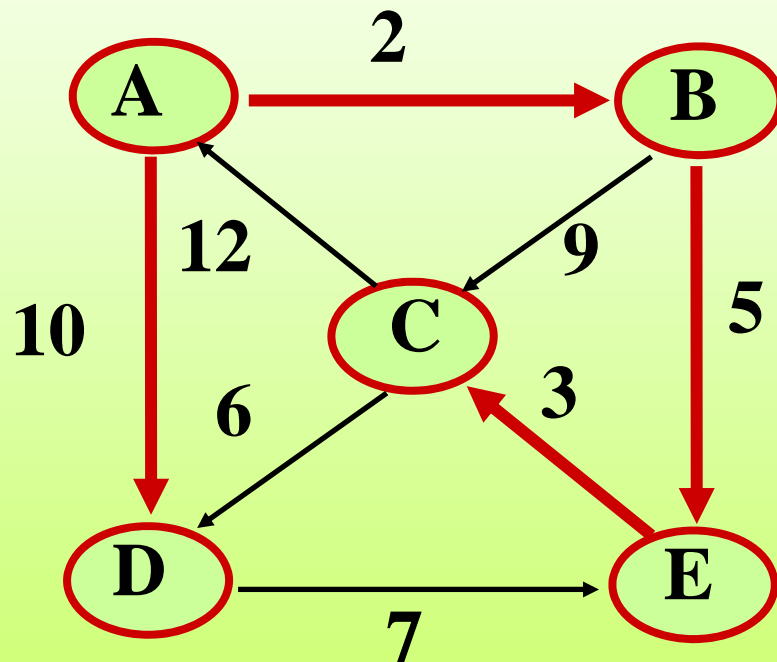
$$S = \{(v_A, v_B), (v_B, v_E), (v_E, v_C)\}$$

$$Y = \{v_A, v_B, v_C, v_E\}$$

$$V-Y = \{v_D\}$$

Seleccionar el vértice de $V-Y$ que tenga el camino más corto desde v_A , pasando por los vértices de Y

Ejemplo



$$S = \{(v_A, v_B), (v_B, v_E), \\ (v_E, v_C), (v_A, v_D)\}$$

$$Y = \{v_A, v_B, v_C, v_D, v_E\}$$

$$V - Y = \emptyset$$

Puesto que Y es igual a V , se ha encontrado la solución

Implementación del Algoritmo de Dijkstra

- Utiliza a la matriz de adyacencias del grafo (W).
- Se auxilia de un arreglo L , indexado de 2 a n , en donde guardará la longitud de los caminos más cortos del vértice v_1 al vértice v_i , usando solamente a los vértices del conjunto Y como intermediarios.
- Se auxilia de un arreglo T , indexado de 2 a n , en donde guardará el índice del vértice v , cuyo arco (v, v_i) es el último arco en el camino más corto de v_1 a v_i usando solamente a los vértices del conjunto Y como intermediarios.

Algoritmo de Dijkstra

$S = \emptyset;$

```
for (i = 2; i <= n; i++)  
{ L[i] = W[1][i];  
  T[i] = 1; }
```

Inicializa los
arreglos auxiliares:
L con los caminos directos
a partir de v_1
T con v_1 pues no se ha
pasado por otro vértice

Repetir n-1 veces: //para incluir los vértices en Y

```
{ min =  $\infty$ ;  
  for (i = 2; i <= n; i++)  
    if ( 0 <= L[i] <= min)  
      { min = L[i]; vmin = i; }
```

Busca el menor
de los caminos
más cortos a
partir de v_1 ,
descartando a los
ya alcanzados

continua...

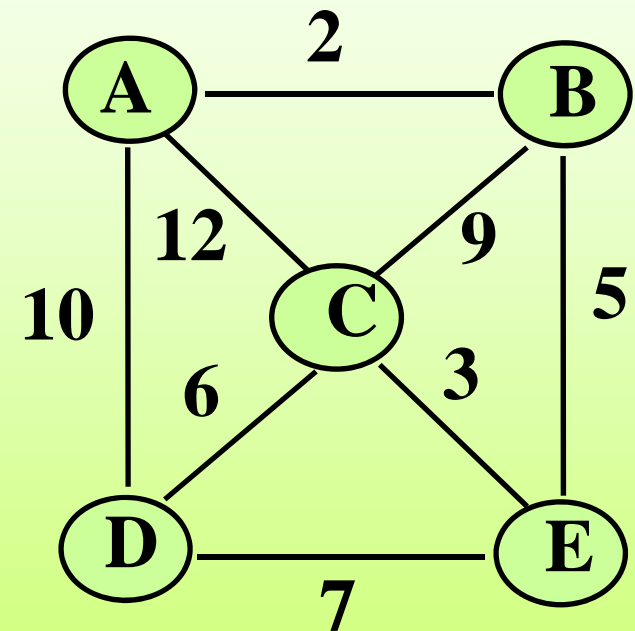
Algoritmo de Dijkstra

```
e = arco formado por T[vmin] y vmin;  
Añadir e a S;  
for (i=2; i<=n; i++)  
    if (L[vmin]+W[vmin][i] < L[i])  
        { L[i] = L[vmin]+W[vmin][i];  
            T[i] = vmin; }  
L[vmin] = -1; //control para que ya no se considere en la búsqueda del menor  
}
```


Ejemplo

Repetir $n-1$ veces:

```
{  min =  $\infty$ ;  
  for (i = 2; i <= n; i++)  
    if ( 0 <= L[i] <= min)  
      { min = L[i]; vmin = i; }  
  e = arco formado por T[vmin] y vmin;  
  Añadir e a S;  
  for (i=2; i <= n; i++)  
    if (L[vmin] + W[vmin][i] < L[i])  
      { L[i] = L[vmin] + W[vmin][i];  
        T[i] = vmin; }  
  L[vmin] = -1;  
}
```





Integración de las 3 técnicas vistas

El problema de la mochila

El problema de la mochila

- Suponer que se tiene un conjunto de objetos, cada uno de los cuales tiene un peso y un valor asociados...
- Suponer también que se tiene una mochila que puede soportar sólo cierto peso en los objetos que guarda, para que no se rompa...
- *¿Qué objetos se pueden guardar en la mochila de tal manera que se guarde el mayor valor posible, sin exceder la capacidad de la mochila?*

Formalmente...

- Sea $S = \{ obj_1, obj_2, \dots, obj_n \}$
- Sea p_i el peso del objeto i y v_i el valor del objeto i .
- Sea P el peso máximo que la mochila soporta.
- El problema de la mochila consiste en encontrar un subconjunto A de S tal que:
 - La sumatoria de los v_i de los objetos de A es el máximo posible,
 - siempre y cuando, la sumatoria de los p_i de los objetos de A es menor o igual a P .

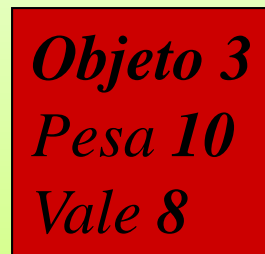
Solución al problema...

- FUERZA BRUTA:
 - Encontrar todos los posibles subconjuntos que se pueden formar de S y seleccionar el que tiene la sumatoria mayor de los valores de los objetos.
 - Comportamiento $O(2^n)$
- Aplicando técnicas de diseño de algoritmos:
 - ¿Divide y vencerás?
 - ¿Programación dinámica?
 - ¿Algoritmo voraz?

Solución con el enfoque de un **algoritmo voraz**...

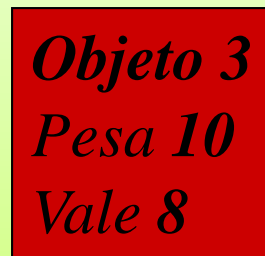
- Intuitivamente, ¿cómo llenarías la mochila?
 - ☐ – Seleccionando primero a los de **mayor peso**?
 - ☐ – Seleccionando primero a los de **menor valor**?
 - ☐ – Seleccionando primero a los de **mayor valor**?
 - ☐ – Seleccionando primero a los de **menor peso**?

Seleccionando el valor mayor...



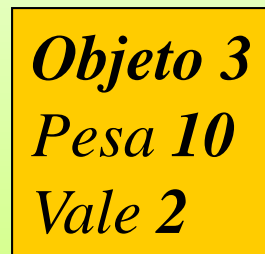
- Solución: *Objeto 1*, acumula 10
- Solución óptima: *Objetos 2 y 3*, acumulan 17

Seleccionando el peso menor...



- Solución: *Objetos 2 y 3*, acumulan 17
- Solución óptima: *Objetos 2 y 3*, acumulan 17

Seleccionando el peso menor...



- Solución: *Objetos 2 y 3*, acumulan 5
- Solución óptima: *Objeto 1*, acumula 10

NUEVA PROPUESTA: Seleccionar el valor mayor por unidad de peso



- Solución: *Objetos 1 y 3*, acumulan \$190
- Solución óptima: *Objetos 2 y 3*, acumulan \$200

Conclusión (Voraz)...

- No hay una solución “greedy” que resuelva el problema... (por contraejemplo se demostró)...
- *¿Qué pasaría si los objetos se pueden fraccionar?*
 - La mochila se puede llenar en forma óptima, utilizando la última propuesta de selección (valor mayor por unidad de peso), y llenando la mochila con la fracción correspondiente al siguiente objeto...
 - En este caso el algoritmo voraz funciona... y tiene un comportamiento de $O(n \log_2 n)$.

Solución con Programación dinámica

- *¿Aplica el principio de optimalidad?*
 - Sea **A** el subconjunto de objetos que maximiza el valor acumulado en la mochila (solución al problema).
 - Si **A** NO contiene al *objeto_n*, **A** es igual a la solución óptima si el problema tuviera sólo a los primeros *n-1* objetos.
- Si **A** contiene al *objeto_n*, el valor total acumulado de **A** es igual a v_n más el valor óptimo del subconjunto formado por los primeros *n-1* objetos, con la restricción de que el peso no exceda a $P-p_n$.

Solución con Programación dinámica

- *¿Cuál es el planteamiento de solución recursiva?*
- Sea $V_{i,p}$ el valor acumulado óptimo para los objetos de 1 a i , sin exceder al peso p .
- La solución a encontrar es $V_{n,P}$... la cual puede obtenerse de la siguiente forma:
$$\text{Si } p_n > P \dots\dots\dots V_{n,P} = V_{n-1,P}$$
$$\text{Si } p_n \leq P \dots\dots\dots V_{n,P} = \text{máximo} (V_{n-1,P} , \quad v_n + V_{n-1,P-p_n})$$
- A su vez esto se generaliza para cualquier i y cualquier p .

Implementación

- Definir una matriz **V** de 0 a n renglones y de 0 a P columnas...
- El renglón 0 y la columna 0 se inicializan con 0...
- El cálculo se realiza renglón por renglón de acuerdo a la siguiente fórmula:
$$\text{Si } p_i > p: \quad V[i,p] = V[i-1,p]$$
$$\text{Si } p_i \leq p: \quad V[i,p] = \text{máximo}(V[i-1,p], v_i + V[i-1,p-p_i])$$
- Comportamiento del algoritmo: **O(nP)**.

Análisis del algoritmo

- Para el caso de 3 objetos y una mochila con capacidad de 30 unidades, se tendría una matriz de 3 X 30 para encontrar el elemento $V[3,30]$...
- Cuando el valor de P sea muy grande, la eficiencia del algoritmo se verá afectada... y pudiera llegar a ser mejor la solución de la fuerza bruta!!
- Analizando la definición recursiva, el cálculo de un valor requiere sólo ciertos valores del renglón anterior, por lo que una mejora al algoritmo, implica sólo calcular los valores que se utilizarán en el siguiente renglón...

Ejemplo



- La solución se encuentra en $V[3,30]$...

$$V[3,30] = V[2,30] \text{ si } 5 > 30$$

$$\underline{V[3,30] = \text{máximo}(V[2,30], \$50 + V[2,25]) \text{ si } 5 \leq 30}$$

Ejemplo



$$V[2,30] = V[1,30] \text{ si } 10 > 30$$

$$\underline{V[2,30] = \text{máximo}(V[1,30], \$60 + V[1,20]) \text{ si } 10 \leq 30}$$

$$V[2,25] = V[1,25] \text{ si } 10 > 25$$

$$\underline{V[2,25] = \text{máximo}(V[1,25], \$60 + V[1,15]) \text{ si } 10 \leq 25}$$

Ejemplo



$$V[1,30] = V[0,30] \text{ si } 20 > 30$$

$$V[1,30] = \text{máximo}(V[0,30], \$140 + V[0,10]) \text{ si } 20 \leq 30$$

$$V[1,25] = V[0,25] \text{ si } 20 > 25$$

$$V[1,25] = \text{máximo}(V[0,25], \$140 + V[0,5]) \text{ si } 20 \leq 25$$

$$V[1,20] = V[0,20] \text{ si } 20 > 20$$

$$V[1,20] = \text{máximo}(V[0,20], \$140 + V[0,0]) \text{ si } 20 \leq 20$$

$$V[1,15] = V[0,15] \text{ si } 20 > 15$$

$$V[1,15] = \text{máximo}(V[0,15], \$140 + V[0,-5]) \text{ si } 20 \leq 15$$

$$V[1,30] = 140$$

$$V[1,25] = 140$$

$$V[1,20] = 140$$

$$V[1,15] = 0$$

Ejemplo



$$V[1,30] = 140$$

$$V[1,25] = 140$$

$$V[1,20] = 140$$

$$V[1,15] = 0$$

$$V[2,30] = V[1,30] \text{ si } 10 > 30$$

$$V[2,30] = \text{máximo}(V[1,30], \$60 + V[1,20]) \text{ si } 10 \leq 30$$

$$V[2,25] = V[1,25] \text{ si } 10 > 25$$

$$V[2,25] = \text{máximo}(V[1,25], \$60 + V[1,15]) \text{ si } 10 \leq 25$$

$$V[2,30] = 200$$

$$V[2,25] = 140$$

Ejemplo



$$V[2,30] = 200$$

$$V[2,25] = 140$$

$$V[3,30] = V[2,30] \text{ si } 5 > 30$$

$$\underline{V[3,30] = \text{máximo}(V[2,30], \$50 + V[2,25]) \text{ si } 5 \leq 30}$$

$$V[3,30] = 200$$

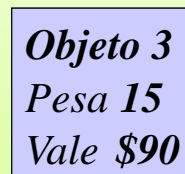
7 cálculos vs. 90 cálculos sin la optimización

Por lo tanto, aplicar la optimización cuando 2^n sea menor a nP

Conclusión final...

- El problema de la mochila (sin objetos fraccionados), se puede resolver por medio de la programación dinámica con un comportamiento de $O(\min(2^n, nP))$...
- Por divide y vencerás tendría un comportamiento de $O(2^n)$...
- Un algoritmo voraz NO resuelve el problema...
- Después se analizará el problema con otras técnicas...
- Pero hasta ahora no se ha demostrado, ni se ha encontrado, un algoritmo que resuelva el problema con un comportamiento mejor a $O(2^n)$ para el peor caso...

Ejercicio



Si $p_i > p$ $V[i,p] = V[i-1,p]$

Si $p_i \leq p$ $V[i,p] = \text{máximo}(V[i-1,p], v_i + V[i-1,p-p_i])$

Análisis de Algoritmos



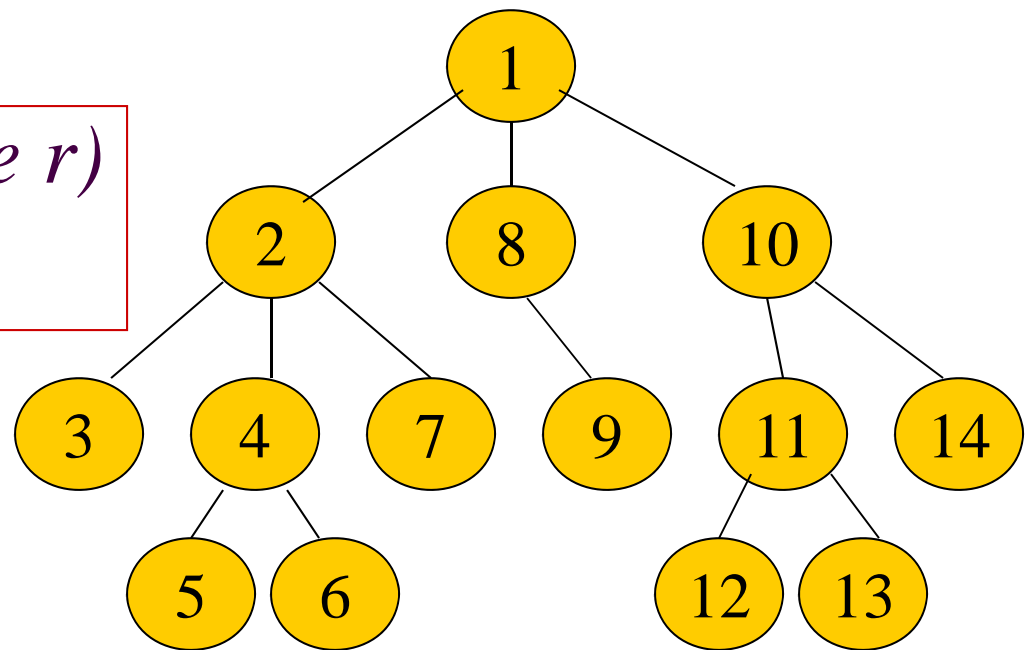
La técnica de Backtracking

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...

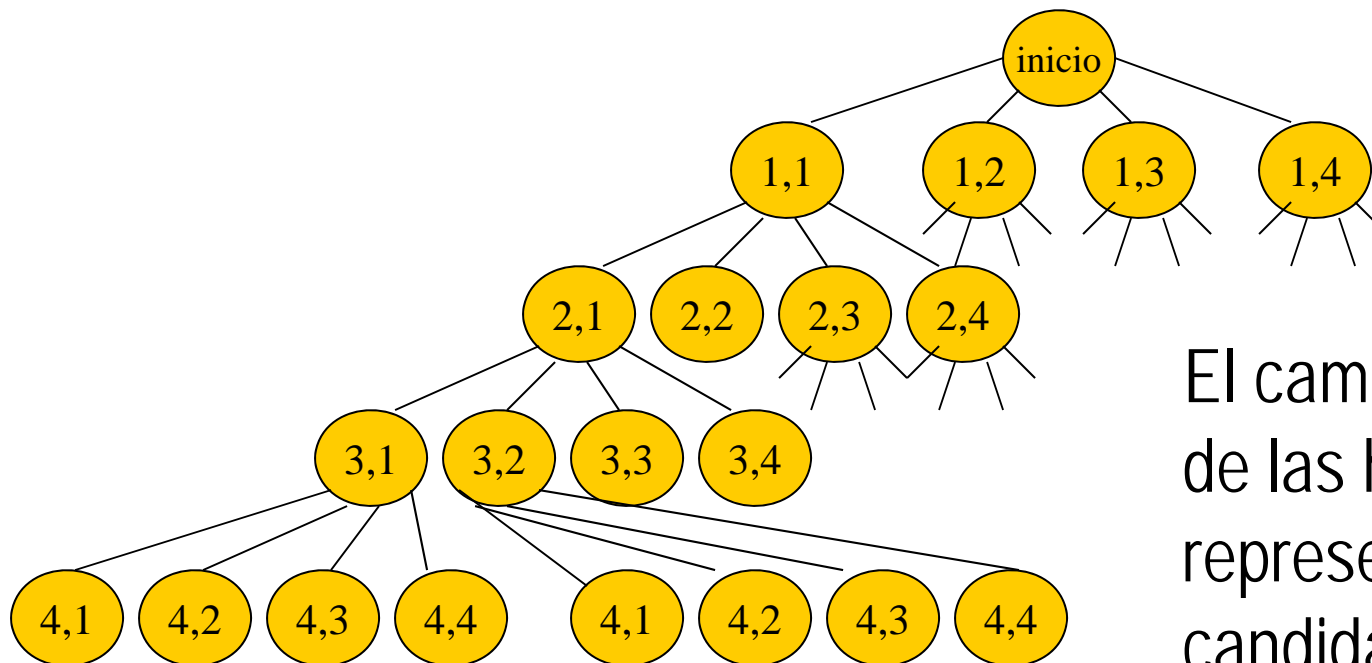
- En un tablero de ajedrez, ¿en qué posiciones se pueden colocar 8 reinas sin que se ataquen una a otra?
 - Una reina ataca a otra, cuando está en el mismo renglón, o en la misma columna, o en la misma diagonal dentro del tablero.
- El problema se puede generalizar como el problema de las n reinas, y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

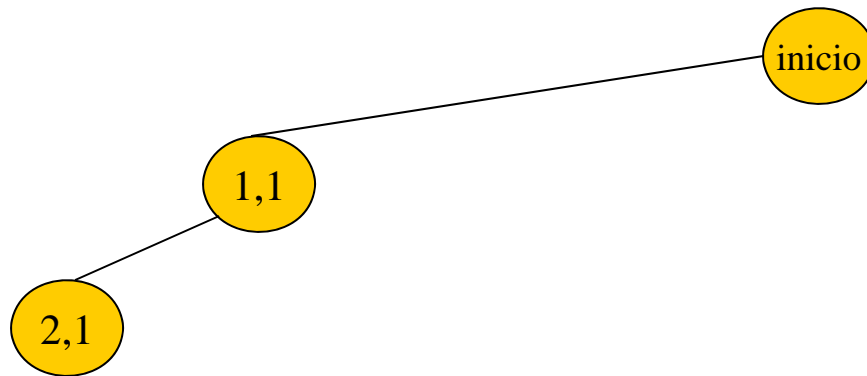


*Restricción
en la búsqueda*



Búsqueda Depth-first

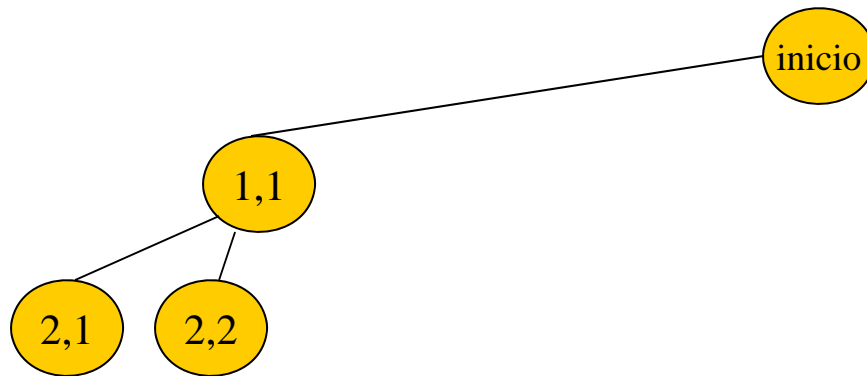
Ejemplo... para $n = 4$



*La estrategia ASEGURA
no ocupar el mismo renglón*

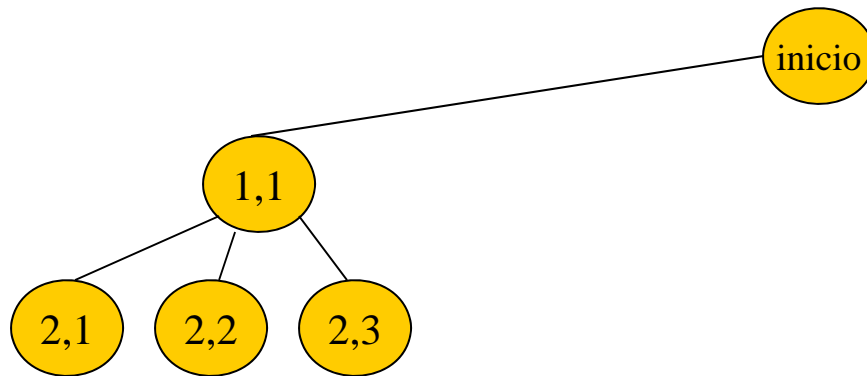
*NO se cumple el criterio
(misma columna)*

Ejemplo... para $n = 4$



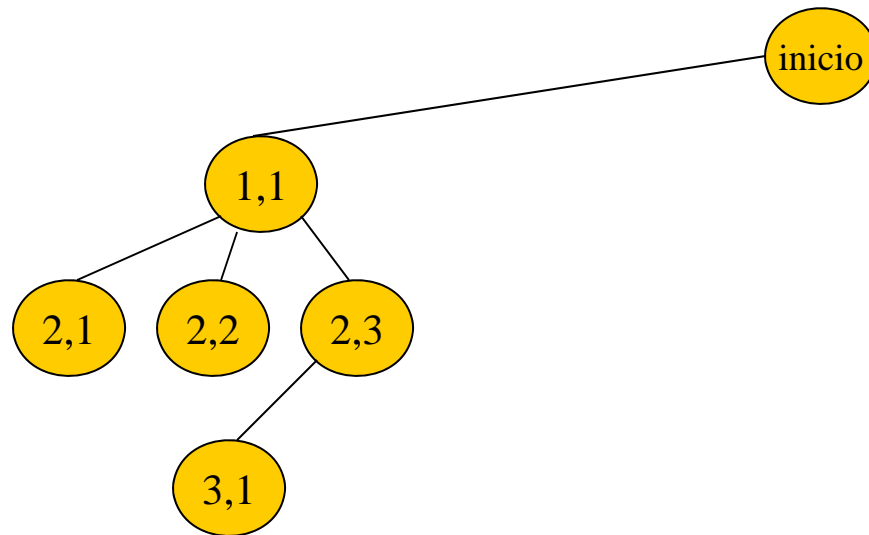
*NO se cumple el criterio
(misma diagonal)*

Ejemplo... para $n = 4$



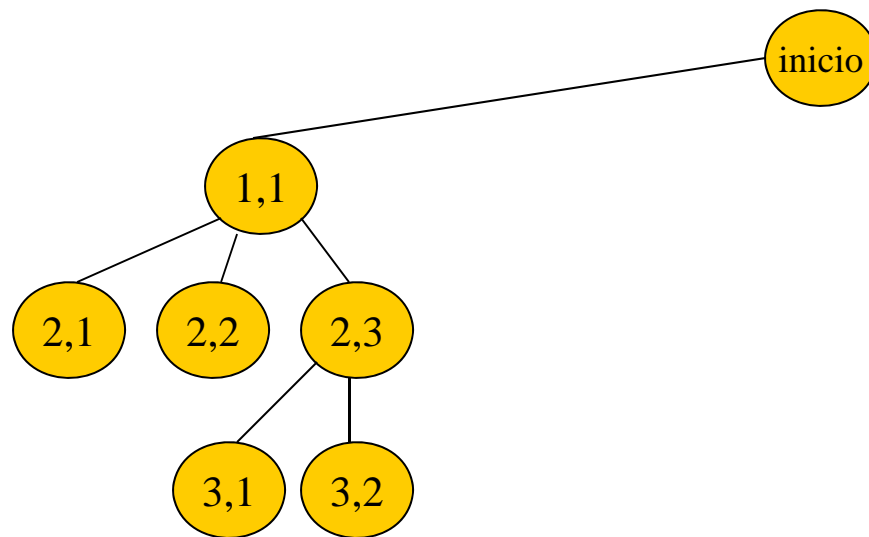
*OK... adelante en la
búsqueda !*

Ejemplo... para $n = 4$



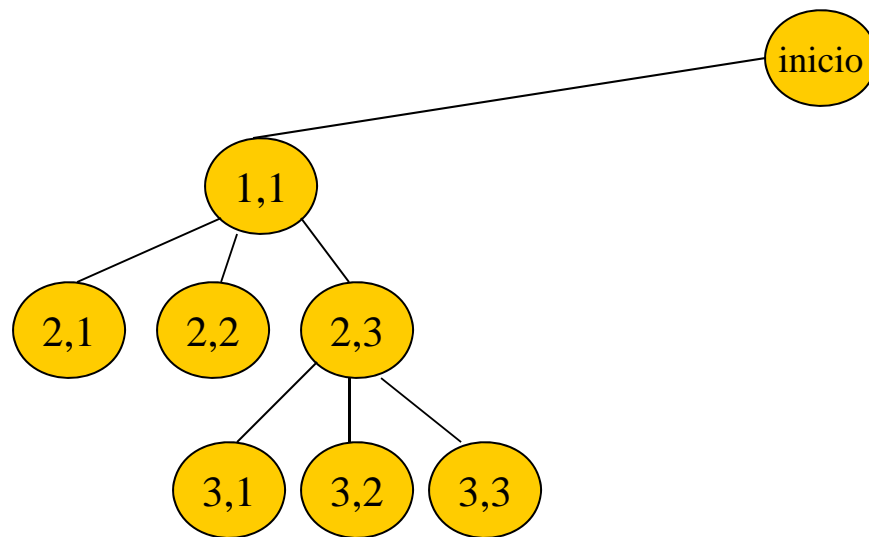
*NO se cumple el criterio
(misma columna)*

Ejemplo... para $n = 4$



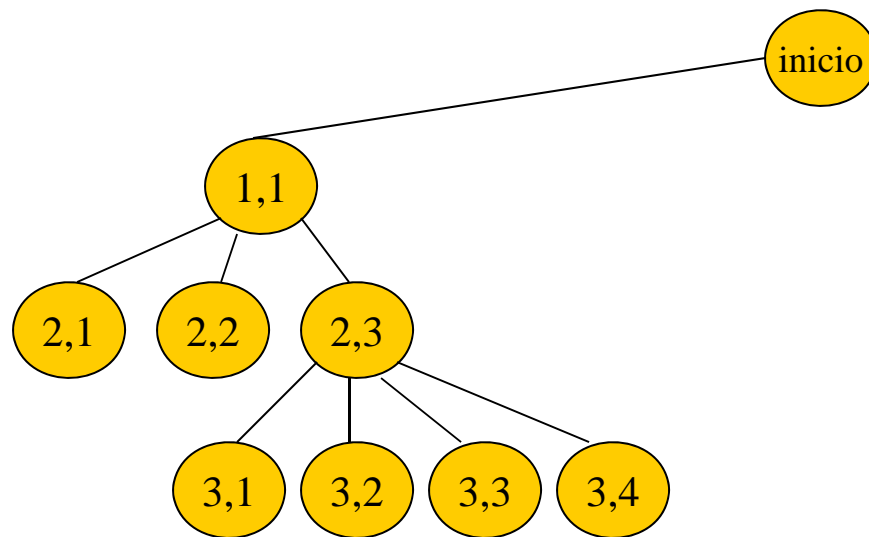
*NO se cumple el criterio
(misma diagonal que 2,3)*₁₃

Ejemplo... para $n = 4$



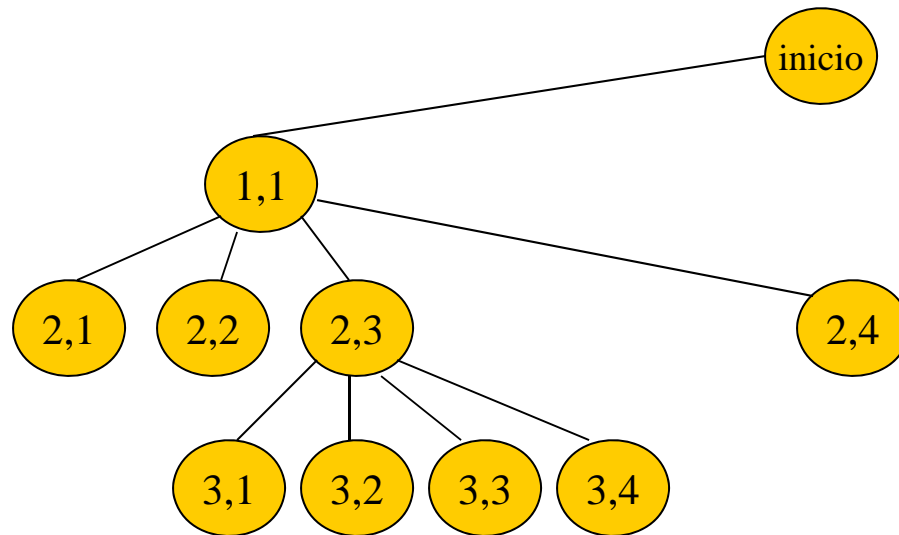
*NO se cumple el criterio
(misma diagonal que 1,1)*

Ejemplo... para $n = 4$



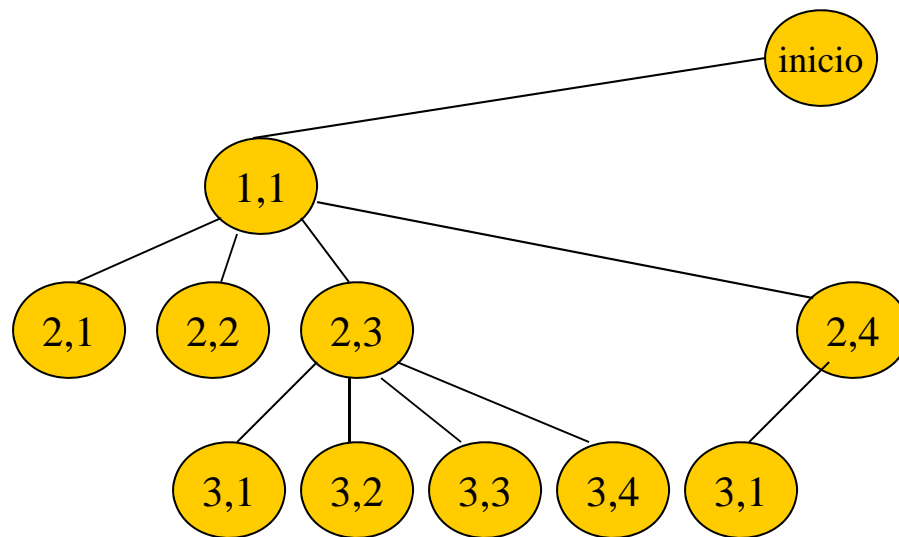
*NO se cumple el criterio
(misma diagonal que 2,3)*

Ejemplo... para $n = 4$



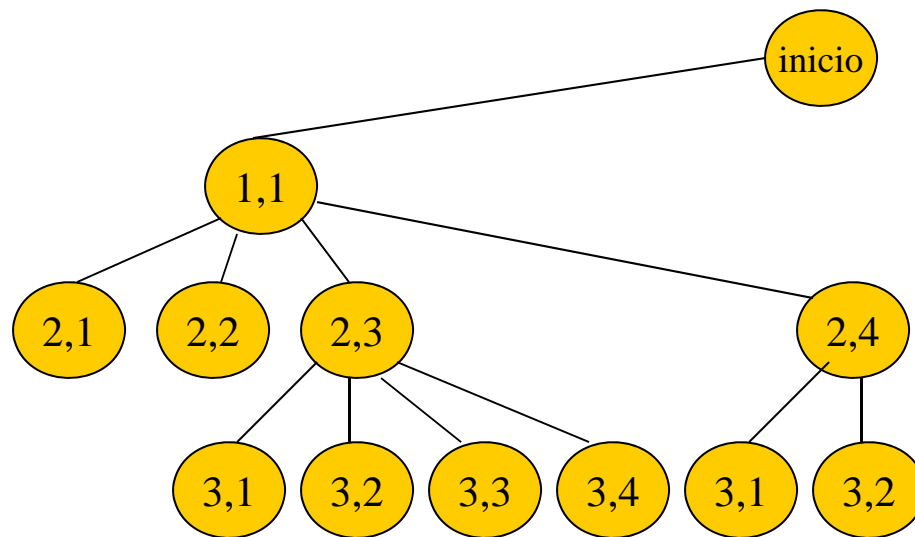
*OK... adelante con la
búsqueda!*

Ejemplo... para $n = 4$



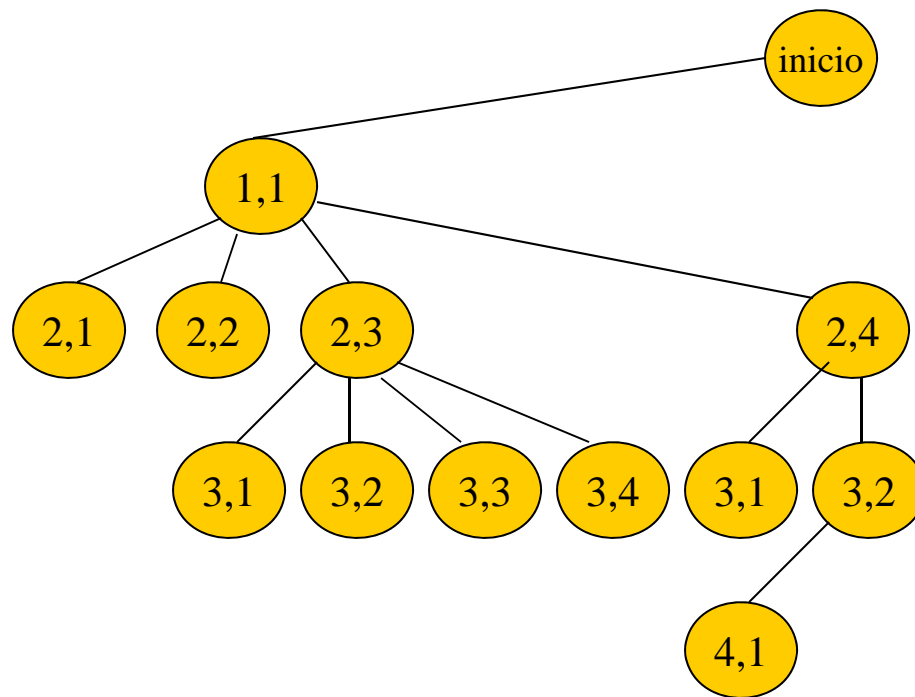
*NO se cumple criterio
(misma columna)*

Ejemplo... para $n = 4$



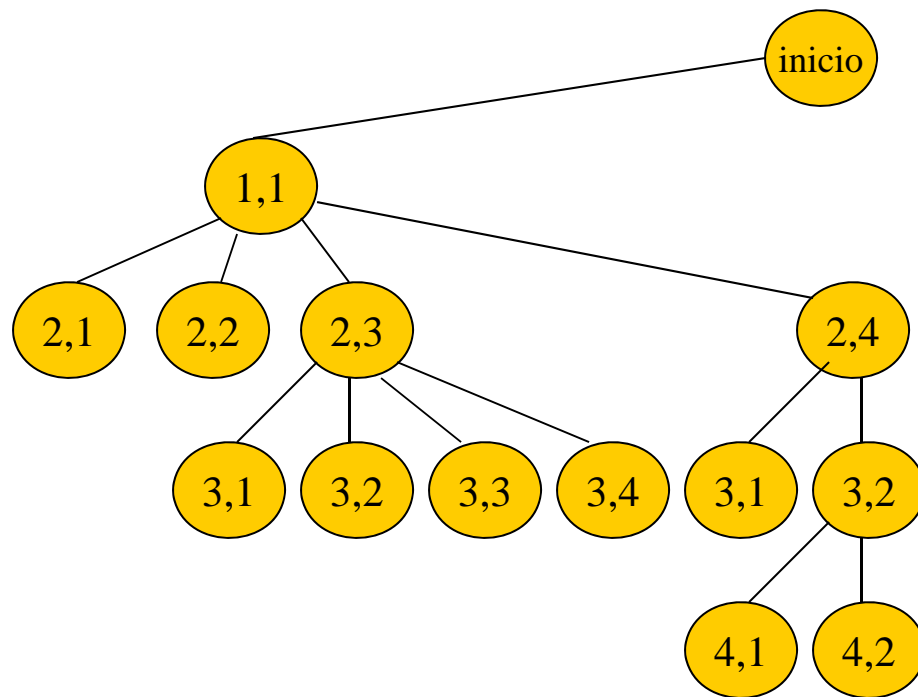
*OK... adelante con la
búsqueda!*

Ejemplo... para $n = 4$



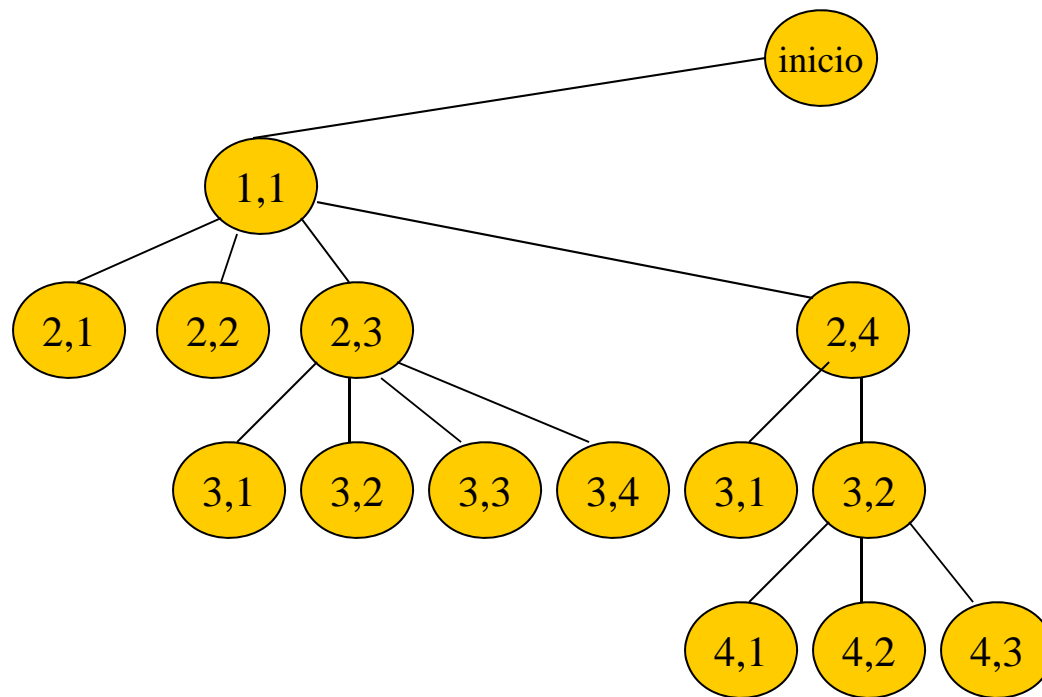
*NO se cumple criterio
(misma columna)*

Ejemplo... para $n = 4$



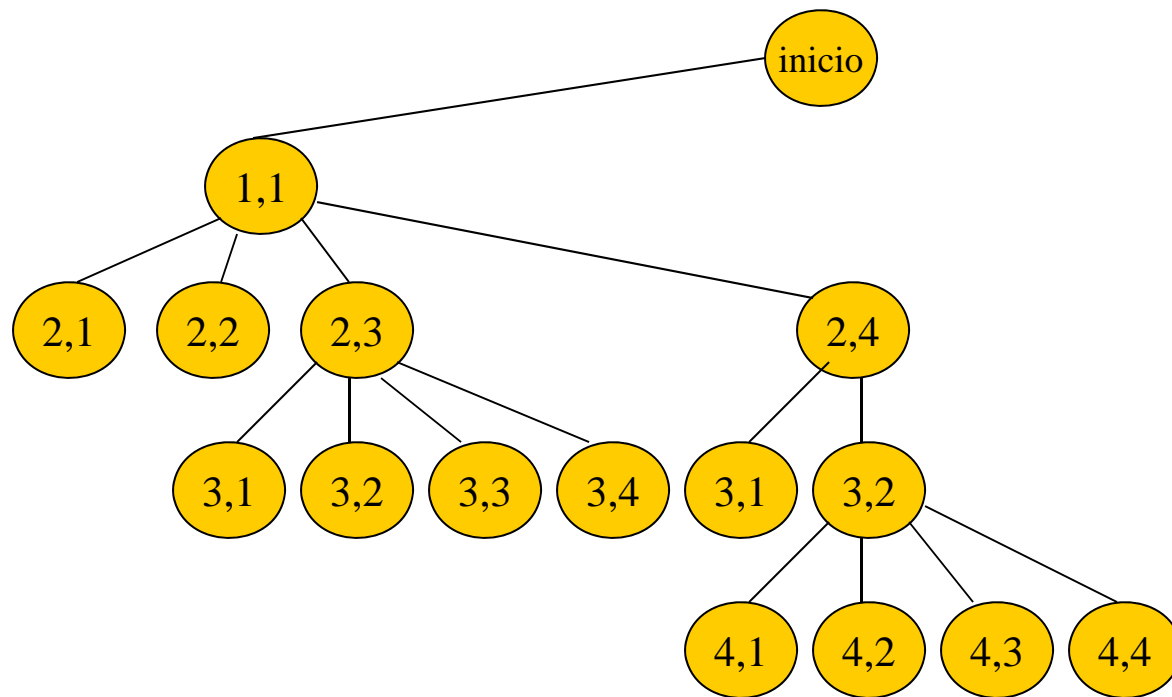
*NO se cumple criterio
(misma columna)*

Ejemplo... para $n = 4$



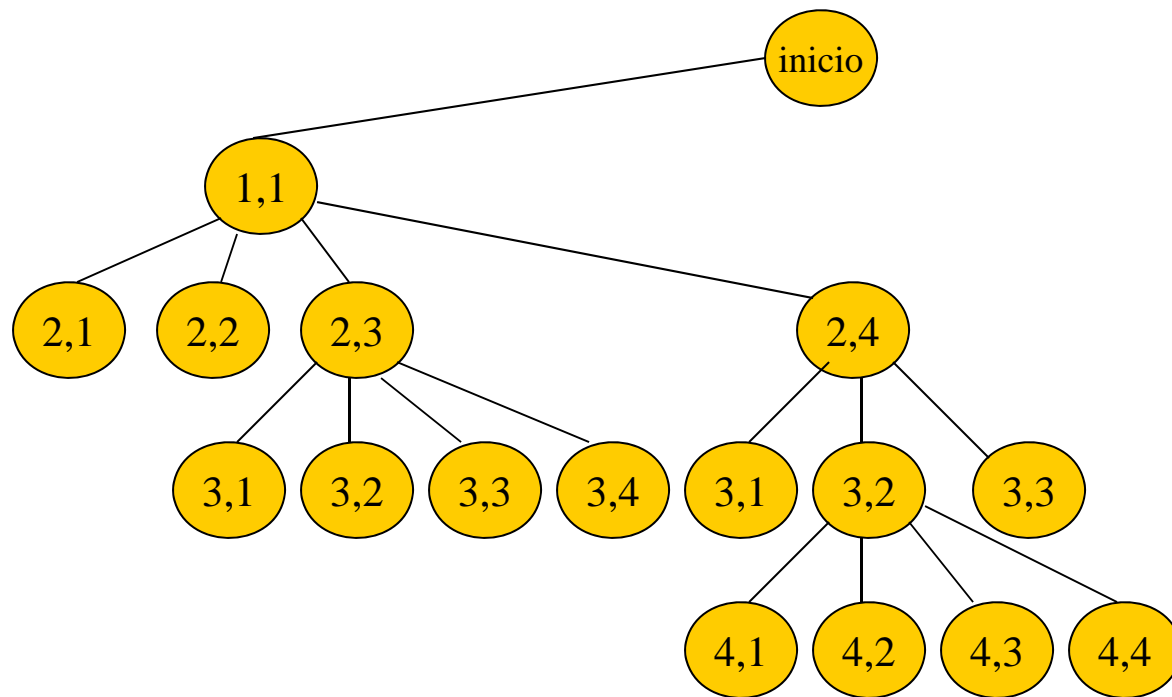
*NO se cumple criterio
(misma diagonal 3,2)*

Ejemplo... para $n = 4$



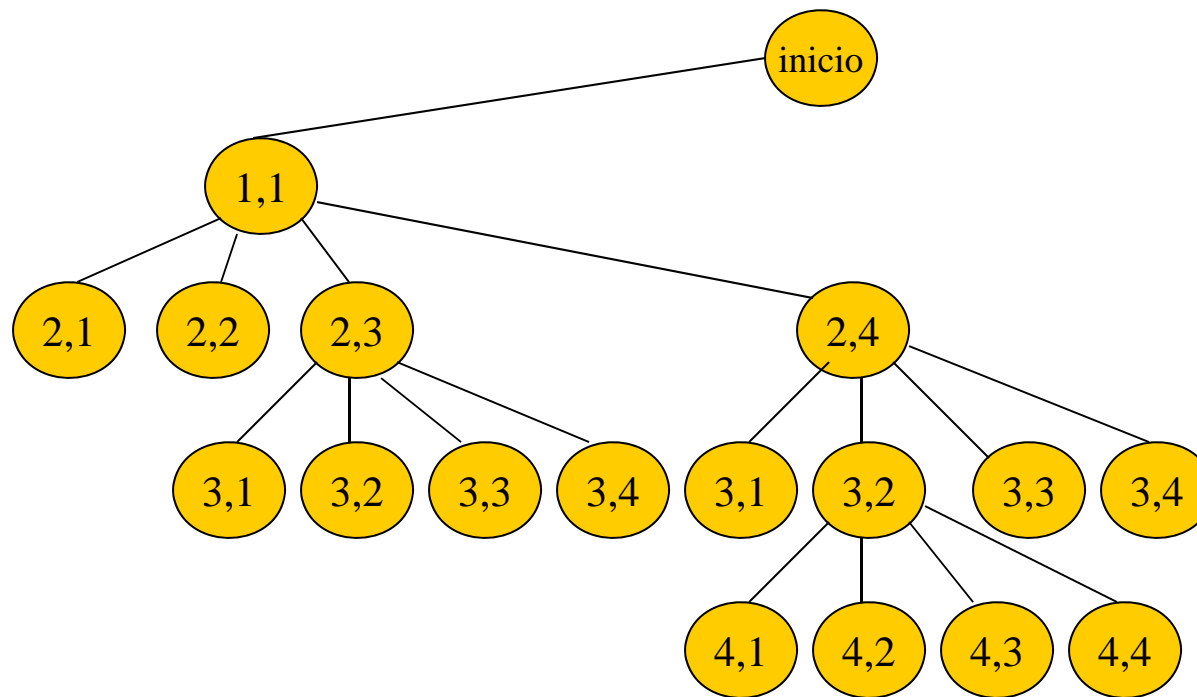
*NO se cumple criterio
(misma columna)*

Ejemplo... para $n = 4$



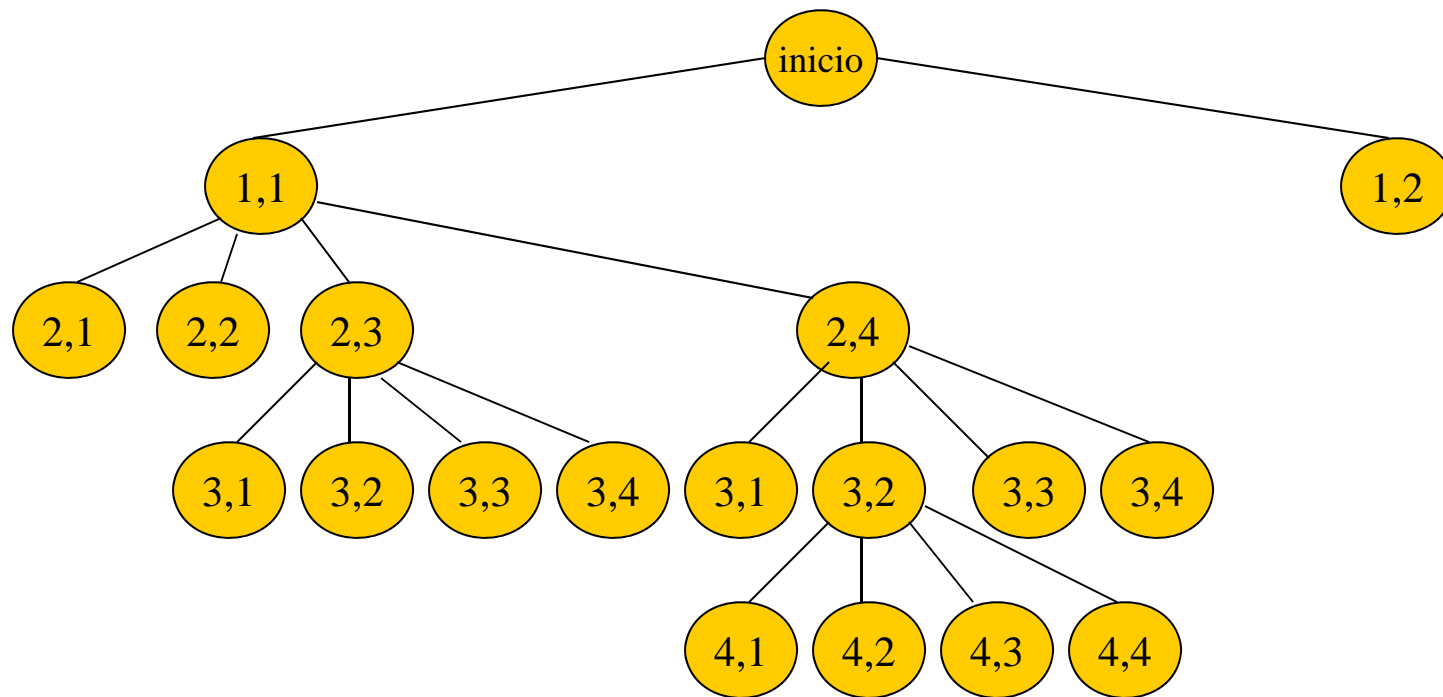
*NO se cumple criterio
(misma diagonal)*

Ejemplo... para $n = 4$



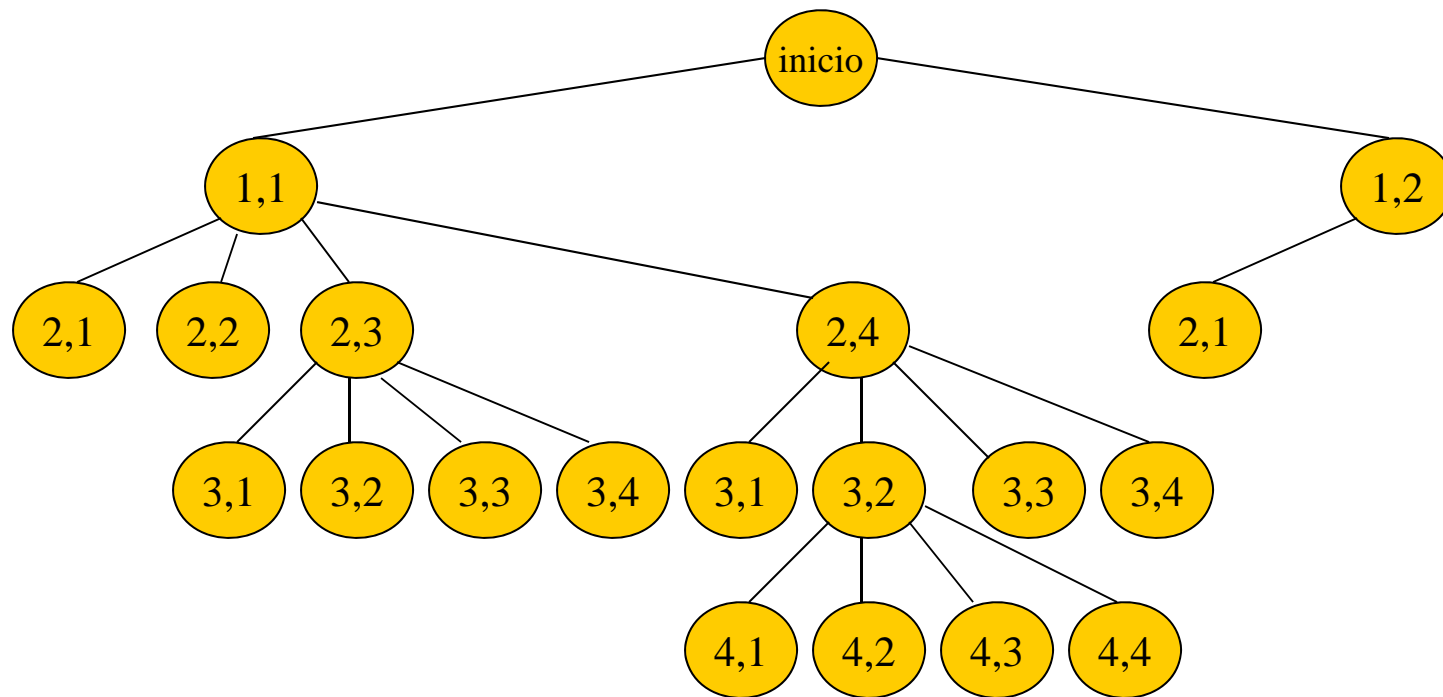
*NO se cumple criterio
(misma columna)*

Ejemplo... para $n = 4$



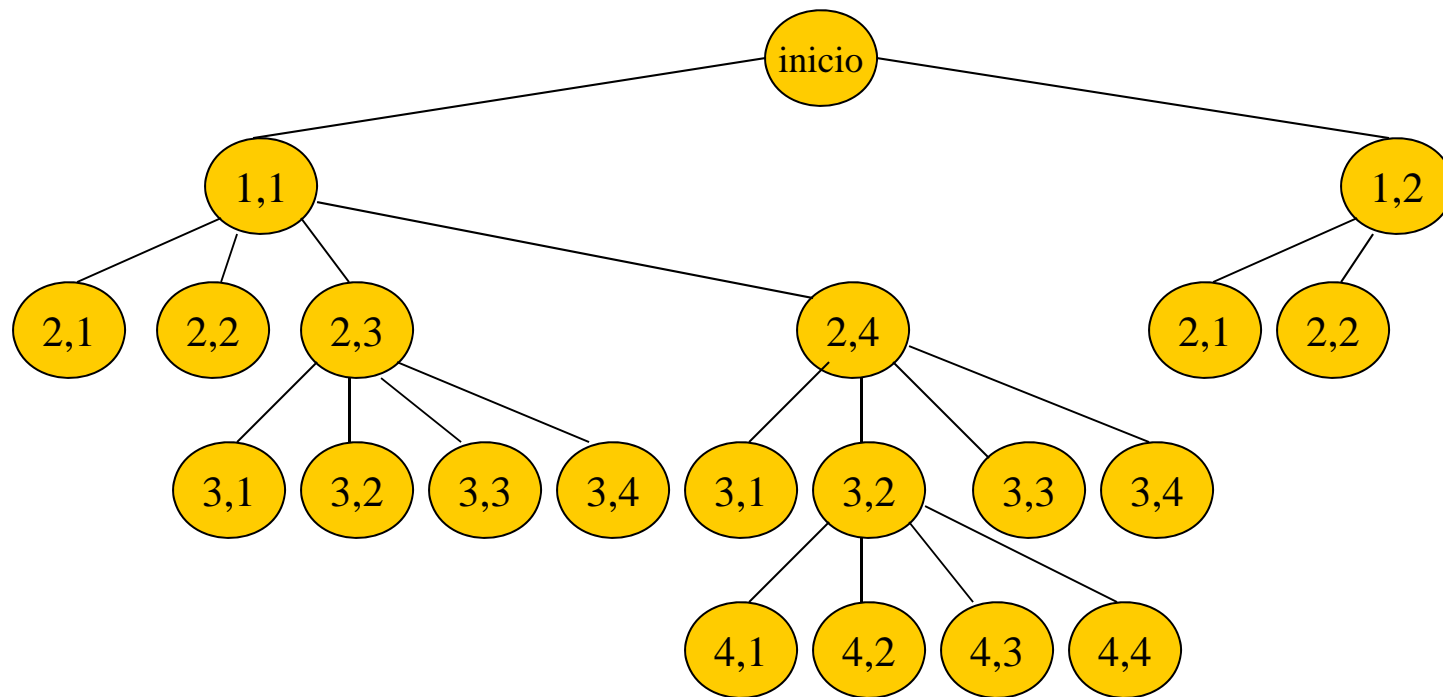
*OK... adelante con la
búsqueda!*

Ejemplo... para $n = 4$



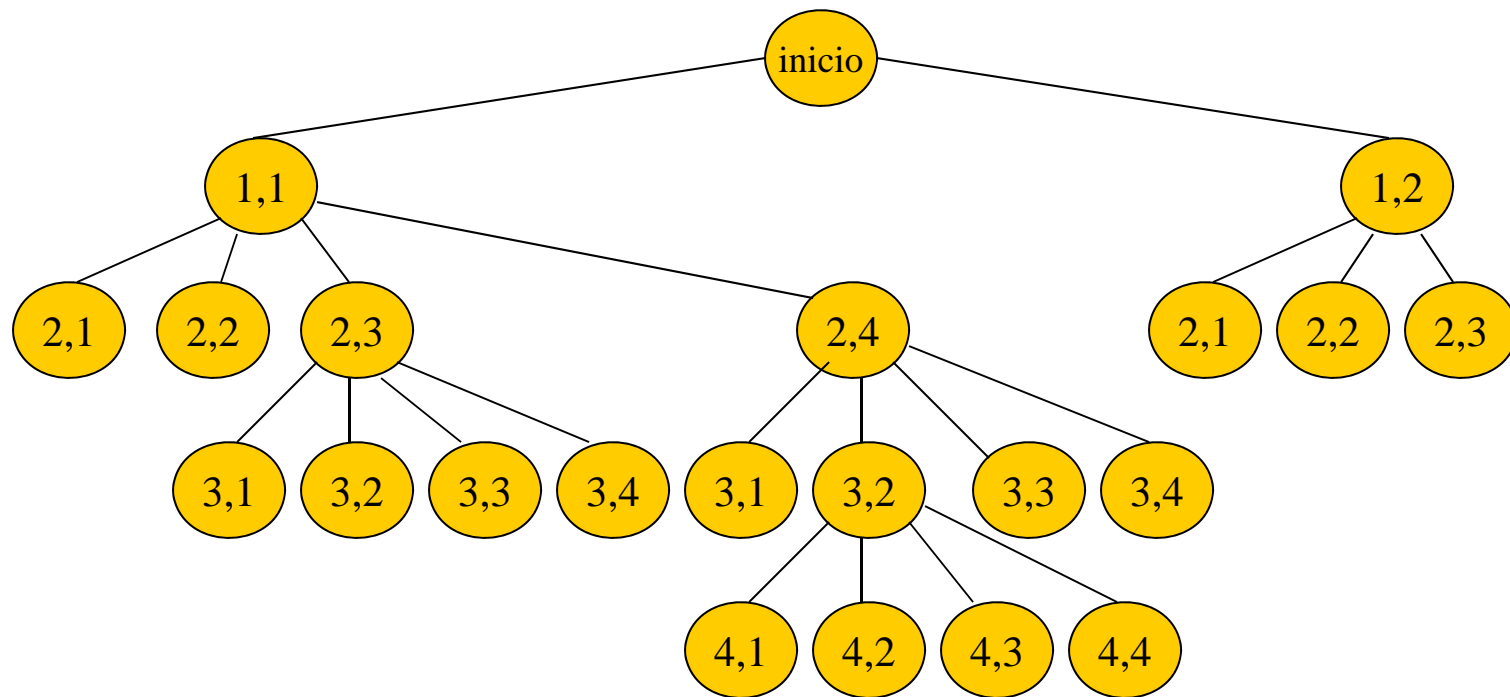
*NO cumple criterio
(misma diagonal)*

Ejemplo... para $n = 4$



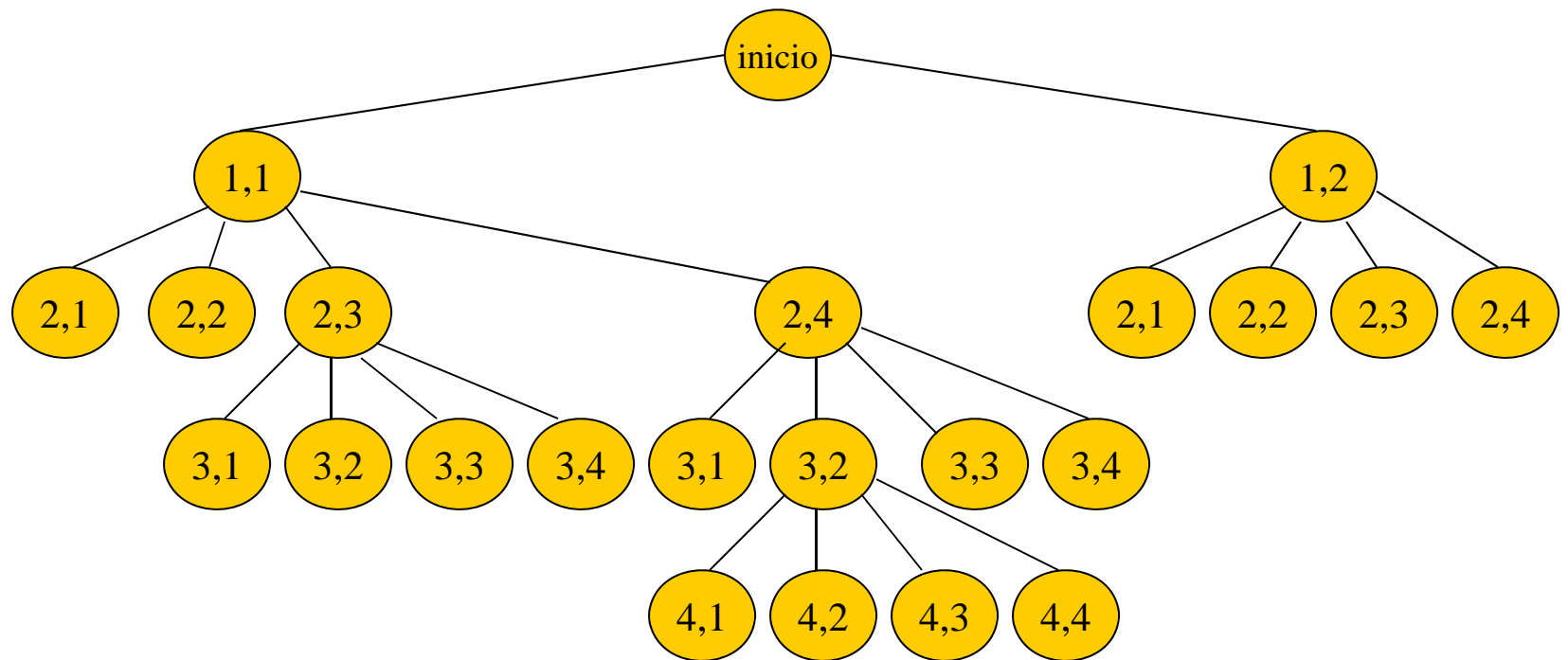
*NO cumple criterio
(misma columna)*

Ejemplo... para $n = 4$



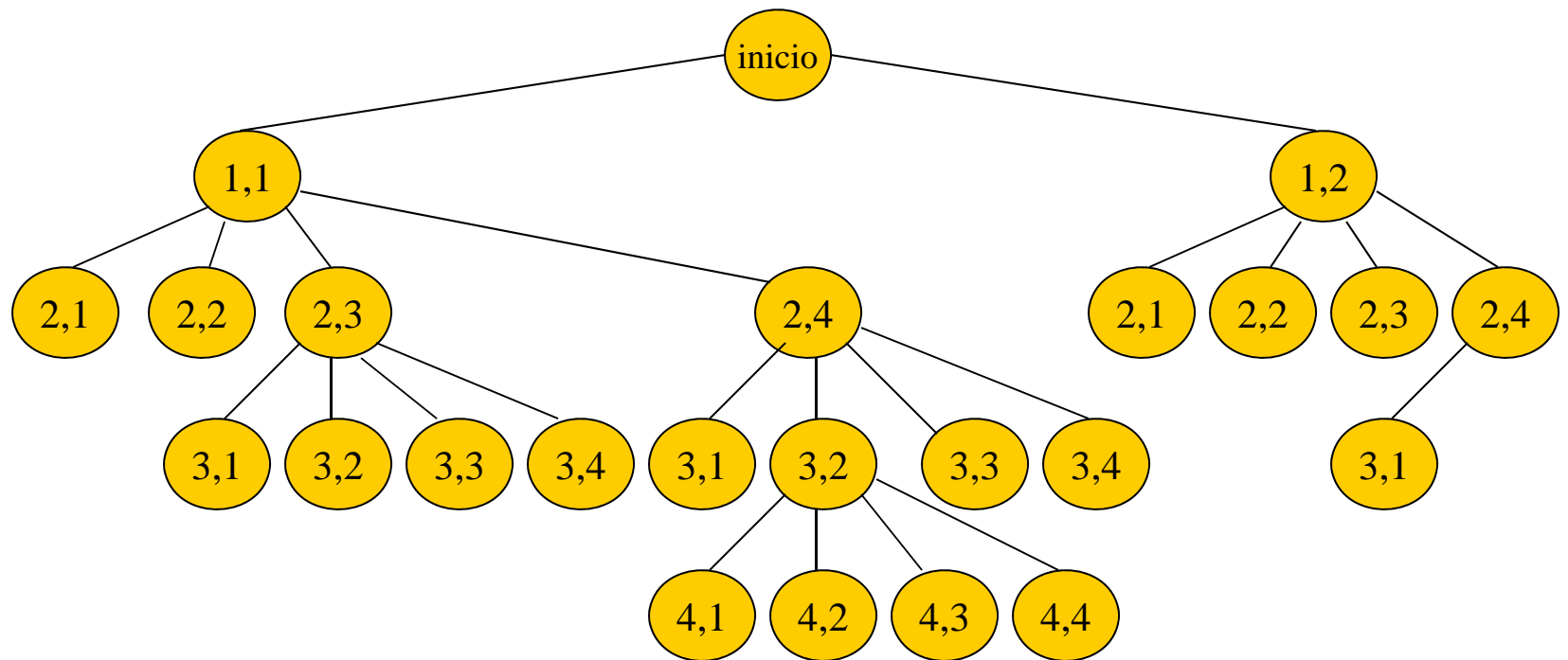
*NO cumple criterio
(misma diagonal)*

Ejemplo... para $n = 4$



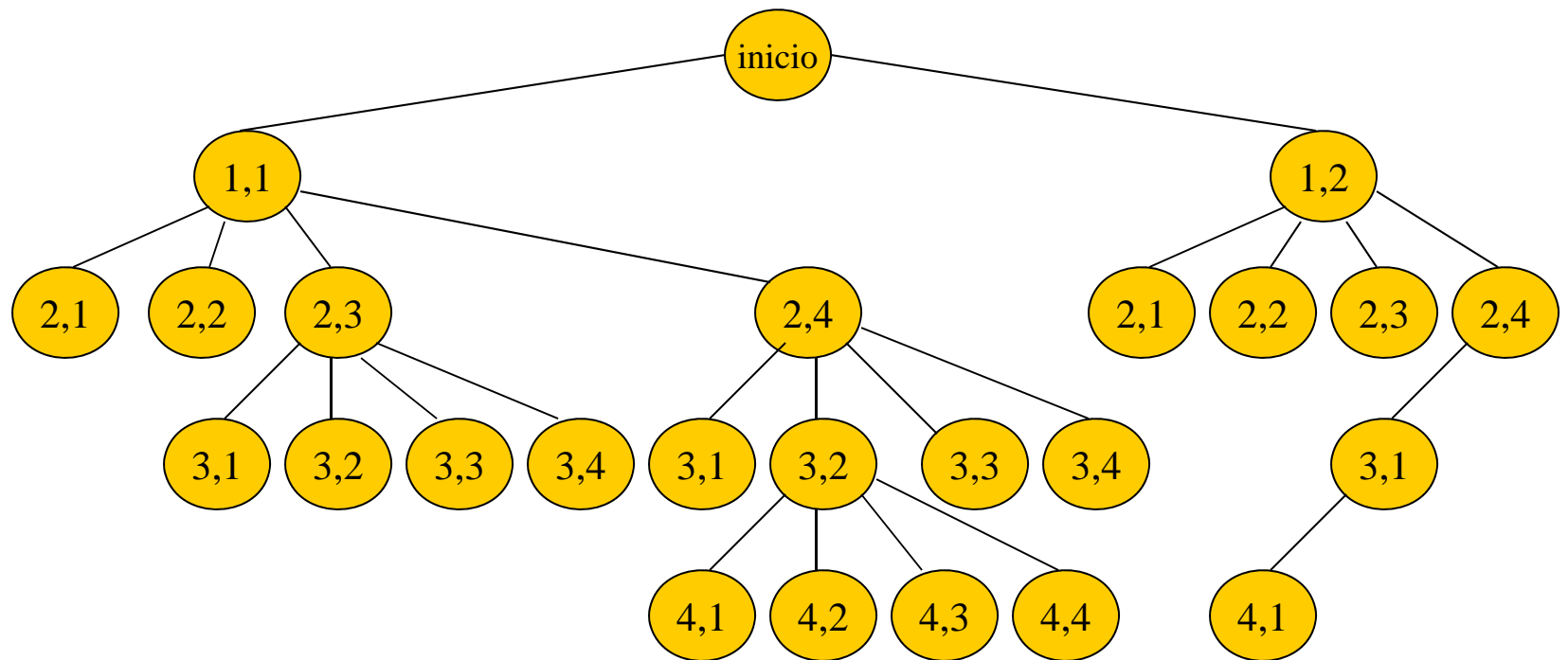
*OK... adelante con la
búsqueda!*

Ejemplo... para $n = 4$



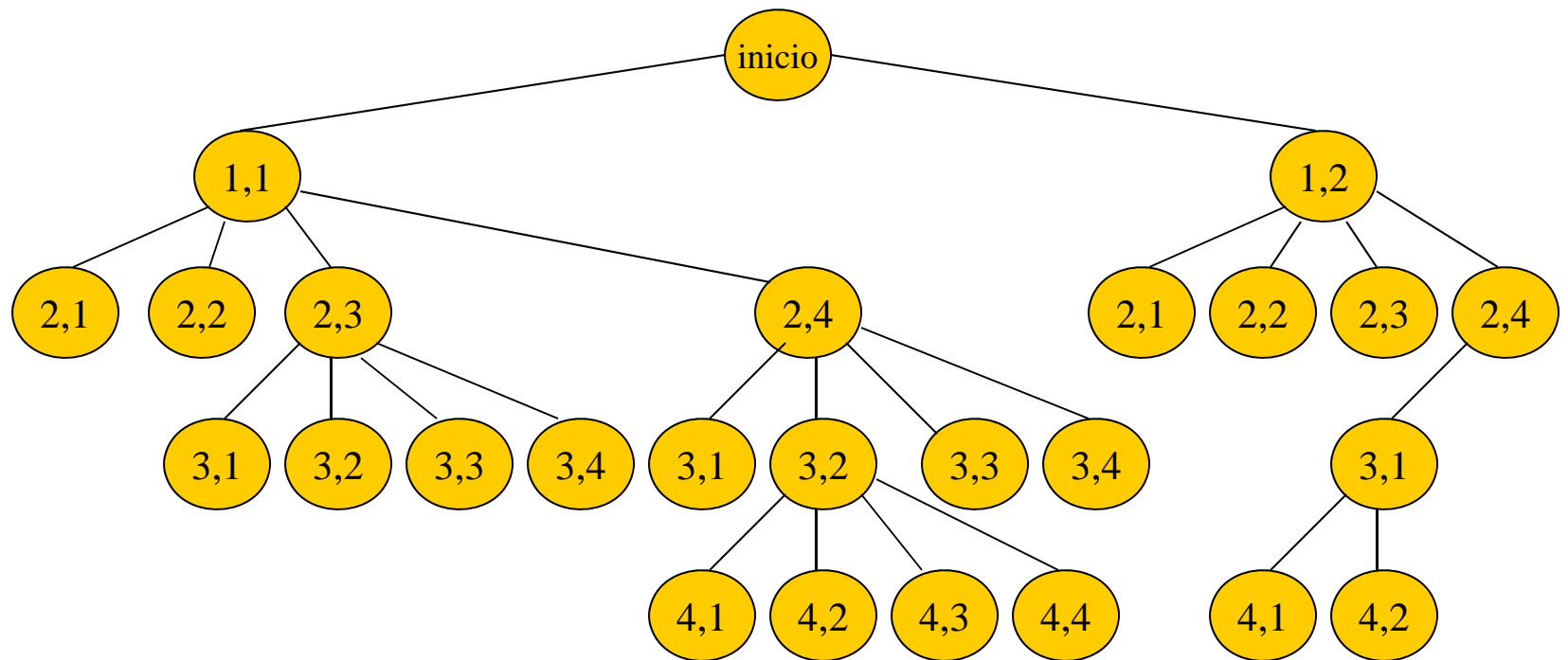
*OK... adelante con la
búsqueda!*

Ejemplo... para $n = 4$



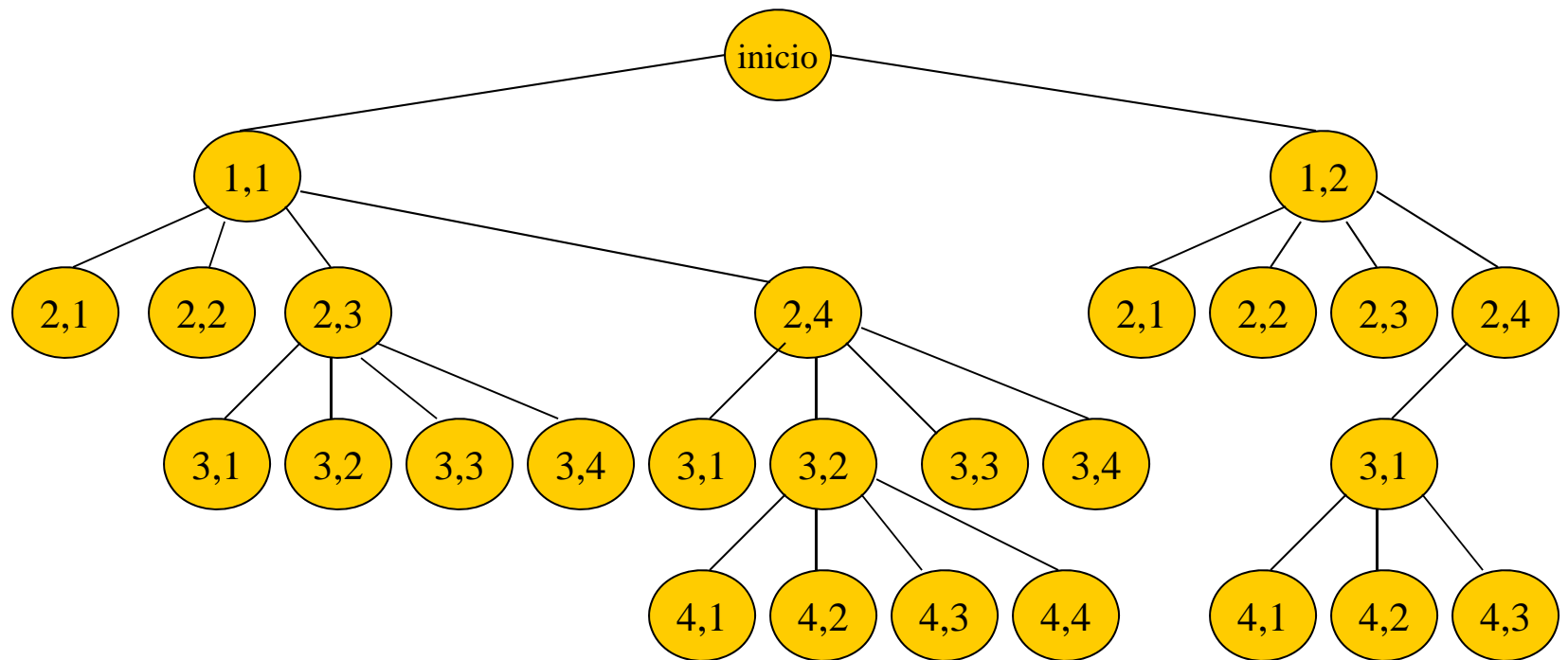
*NO cumple el criterio
(misma columna)*

Ejemplo... para $n = 4$



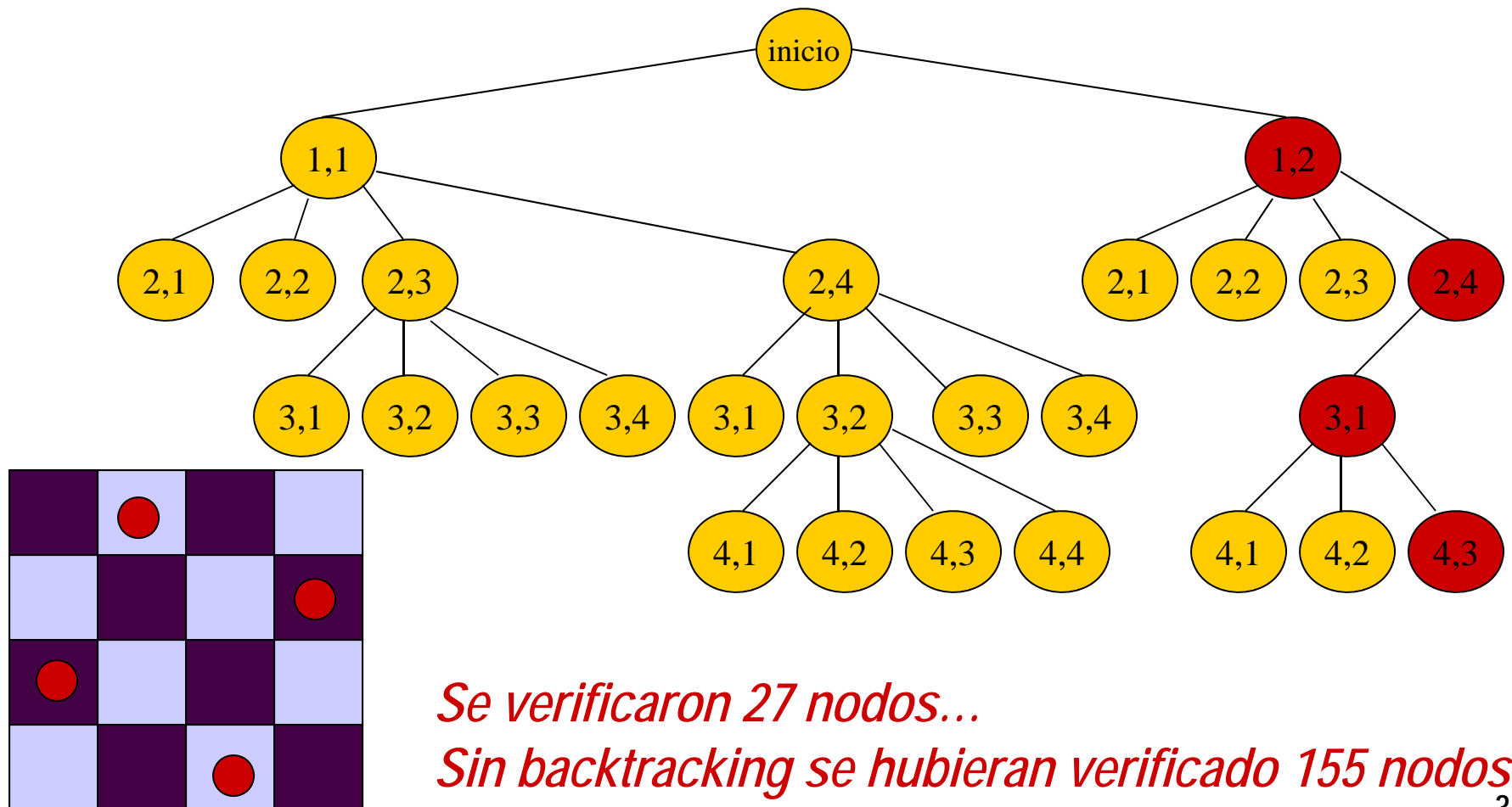
*NO cumple el criterio
(misma columna)*

Ejemplo... para $n = 4$



*OK... se encontró
solución !!*

Ejemplo... para $n = 4$



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

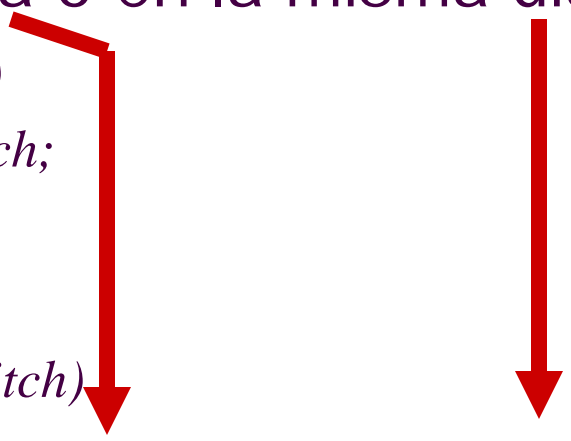
- Se utilizará un arreglo global llamado *col*, donde *col[i]* contiene la posición de la columna para la reina en el renglón *i*.

```
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 los 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 exactamente 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 exactamente 21?

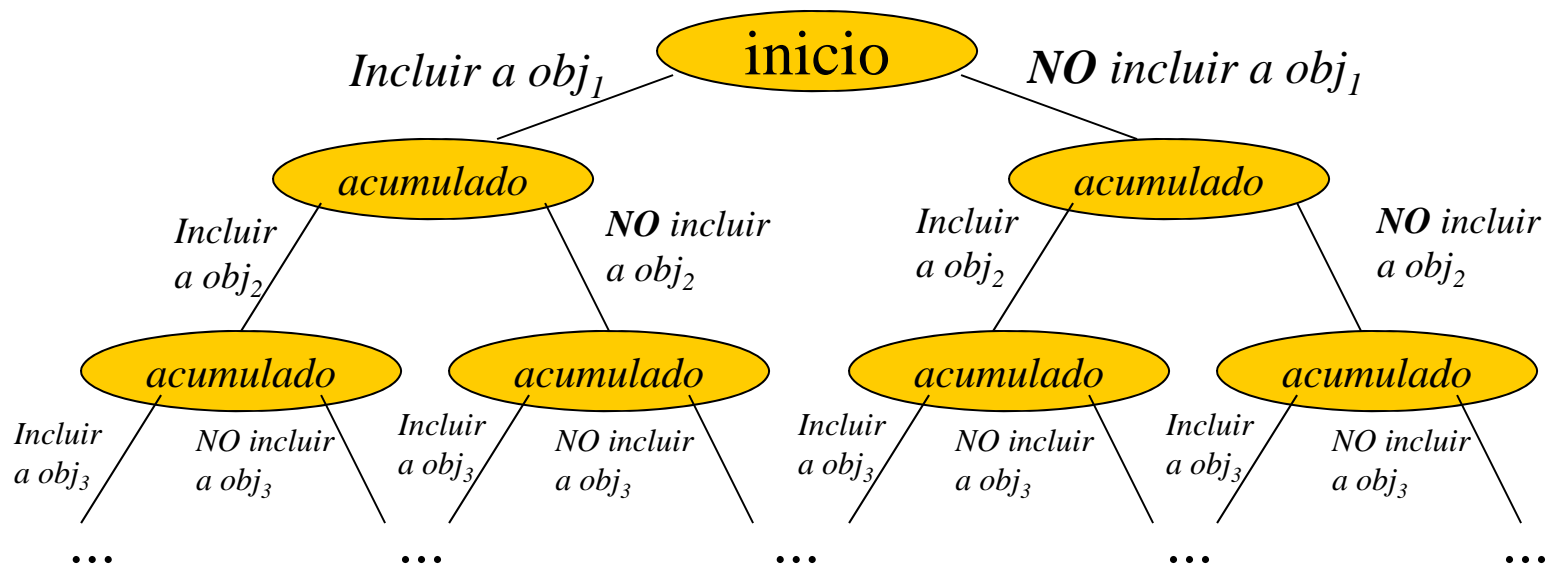
$\{obj_1, obj_2, obj_3\}$

$\{obj_1, obj_5\}$

$\{obj_3, obj_4\}$

Solución con backtracking

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



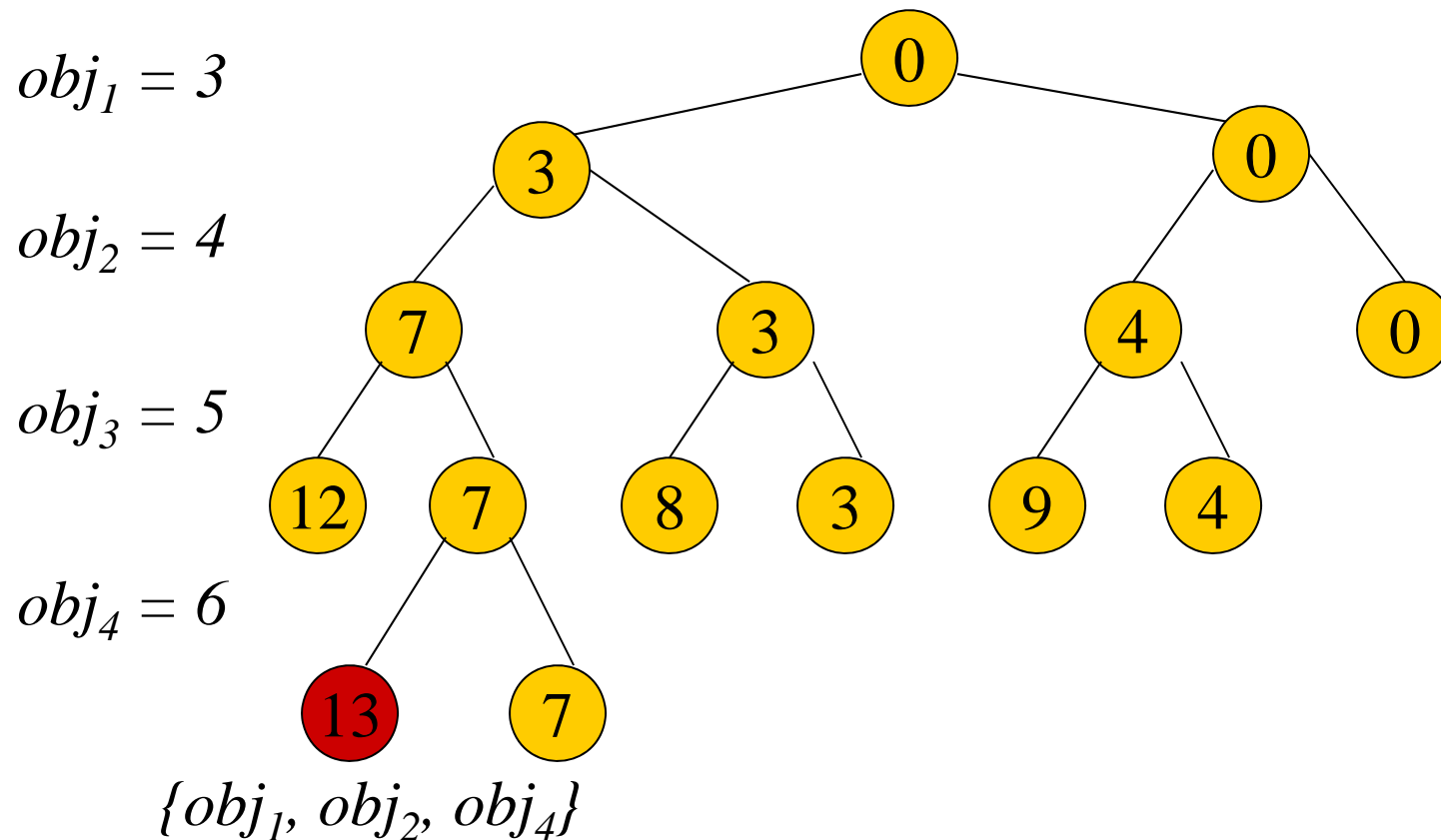
- Las hojas del árbol que tengan el acumulado buscado, indican cuáles son los subconjuntos solución.

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.

Backtracking




Otras aplicaciones

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

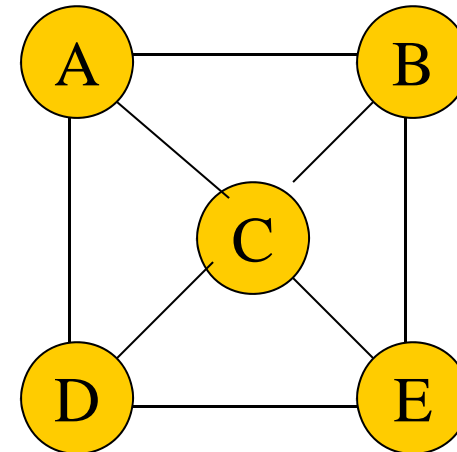
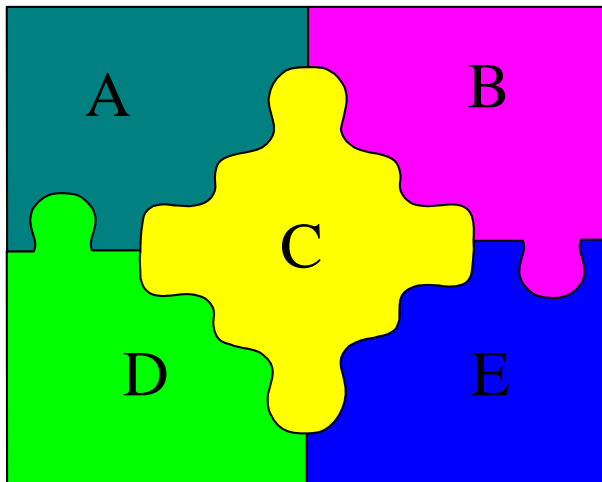
Coloreado de grafos



- Árbol 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.

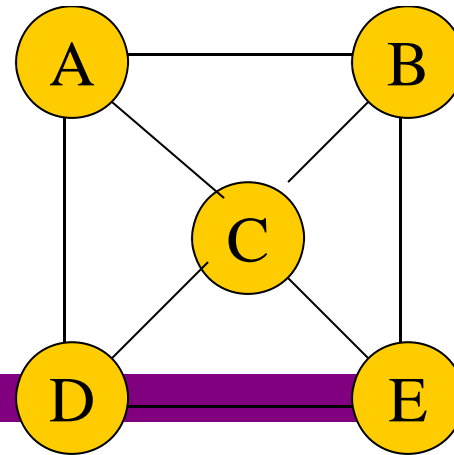
Coloreado de grafos

- Modelación de un mapa en un grafo...



- Árbol de 5 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...*

Análisis de
Algoritmos



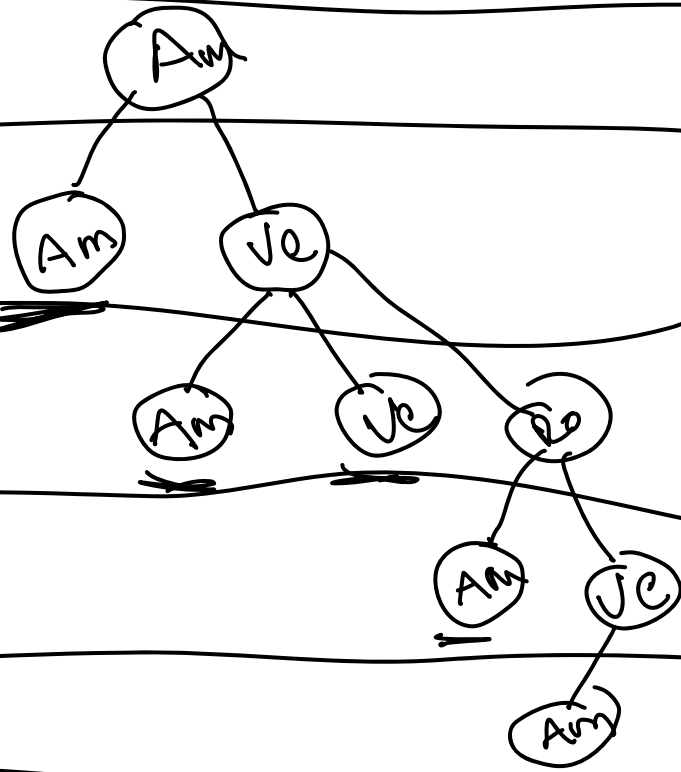
1° - A

2° - B

3° - C

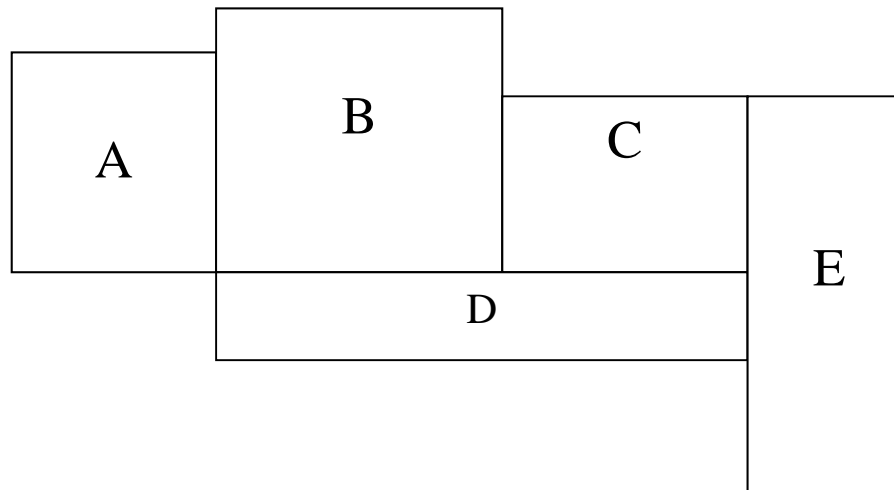
4° - D

5° - E



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

- Árbol 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

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 $[T(n) = (n-1)(n-2)2^{n-3}] \dots$
- 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).

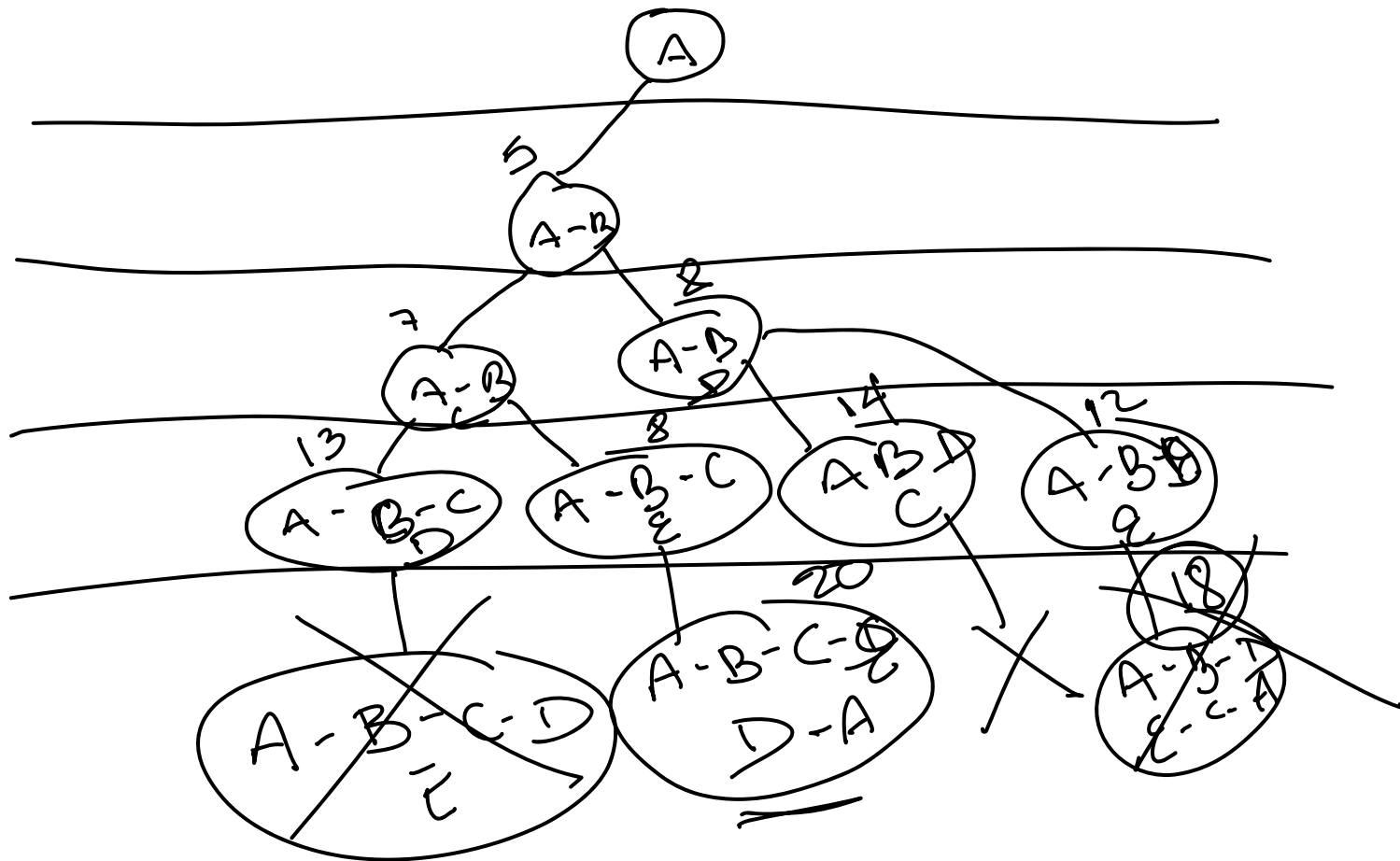
EJERCICIO

- 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.

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

Análisis de Algoritmos

0	5	∞	$\frac{8}{3}$	$\frac{8}{3}$
<u>5</u>	<u>0</u>	<u>2</u>	<u>3</u>	<u>3</u>
8	2	<u>0</u>	<u>6</u>	<u>1</u>
8	3	6	0	<u>4</u>
8	3	1	<u>4</u>	<u>0</u>



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



- Árbol 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, 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

Vacum = \$0

Pacum = 0

Vposible = \$115

Valor óptimo = \$0

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

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

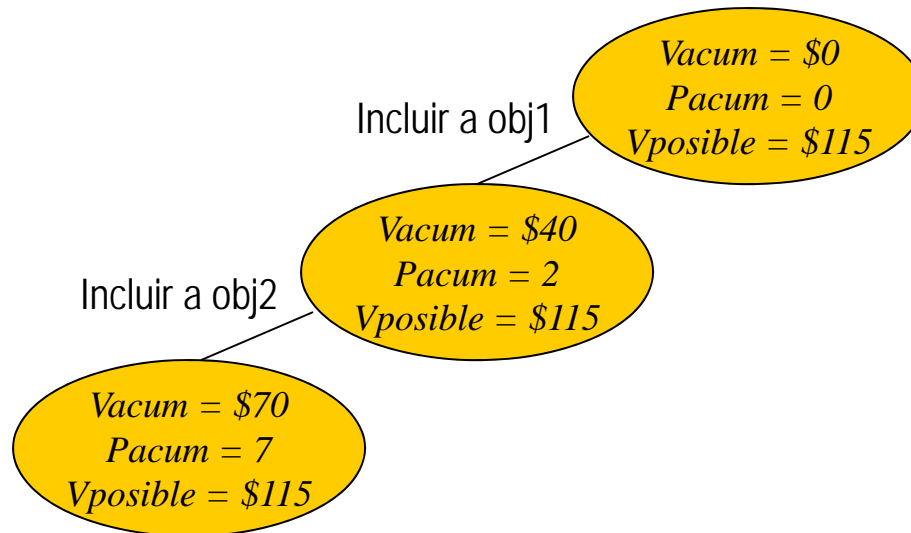
PESO MOCHILA = 16

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

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

Valor posible a acumular:

NO se pueden acumular más objetos sin exceder el peso.

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

Ejemplo

PESO MOCHILA = 16

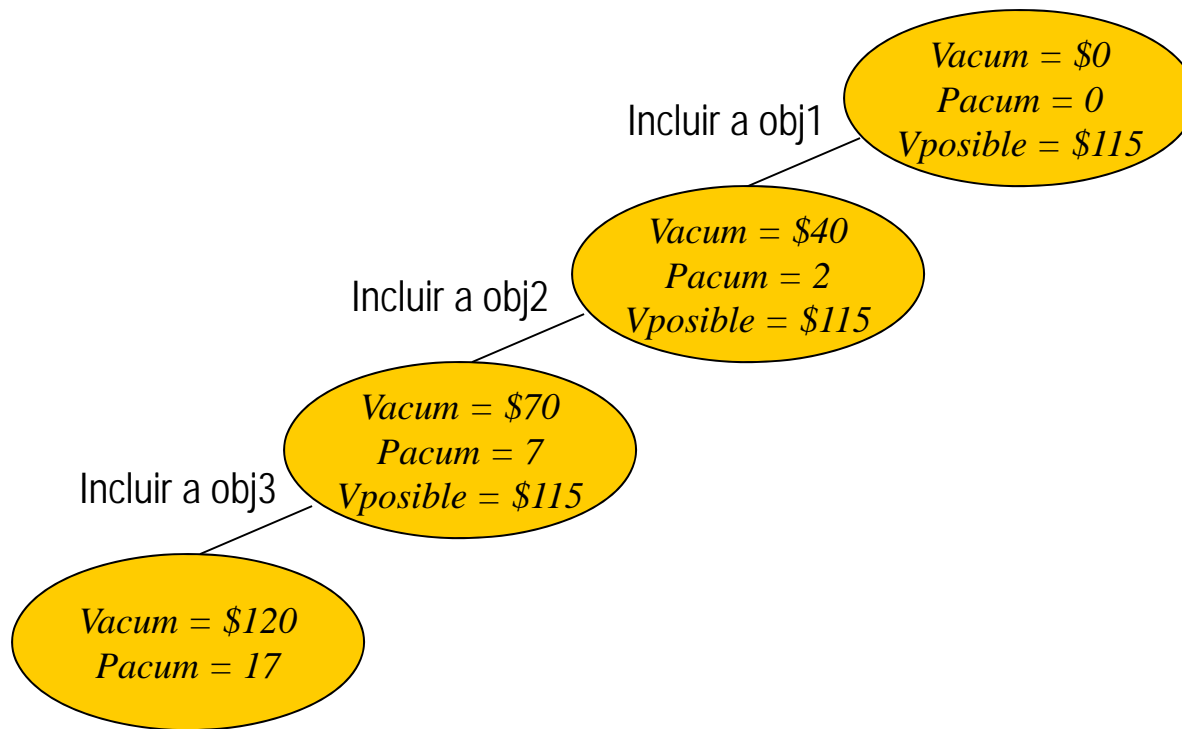
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

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

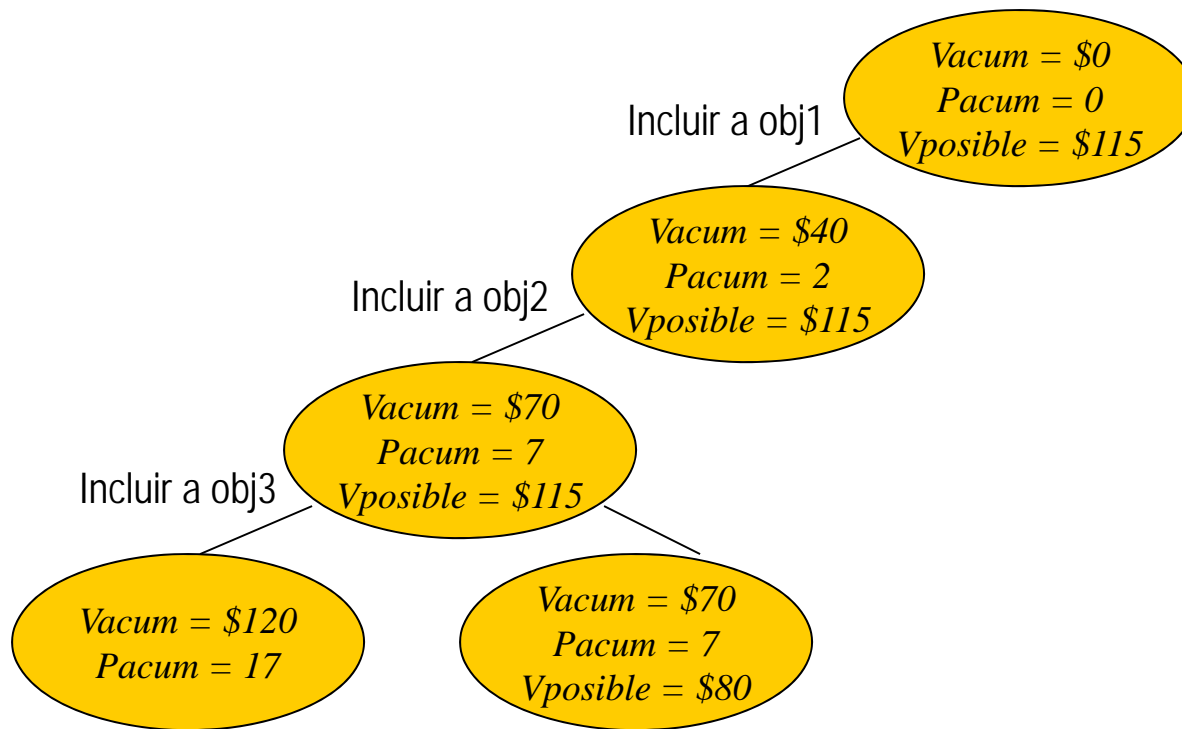
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

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



✓ $Pacum < 16$

✓ $Vposible > \text{Valor óptimo}$

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

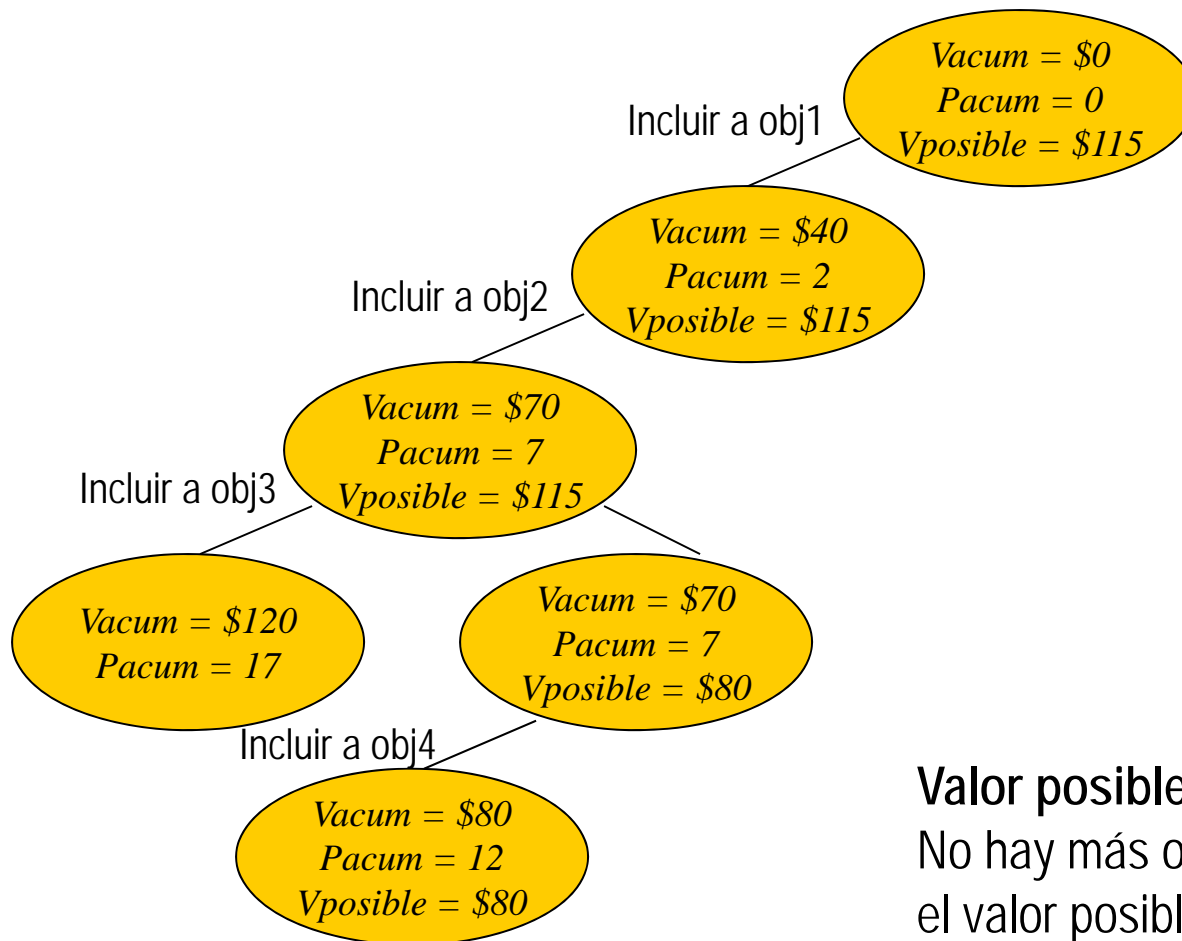
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

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* > 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

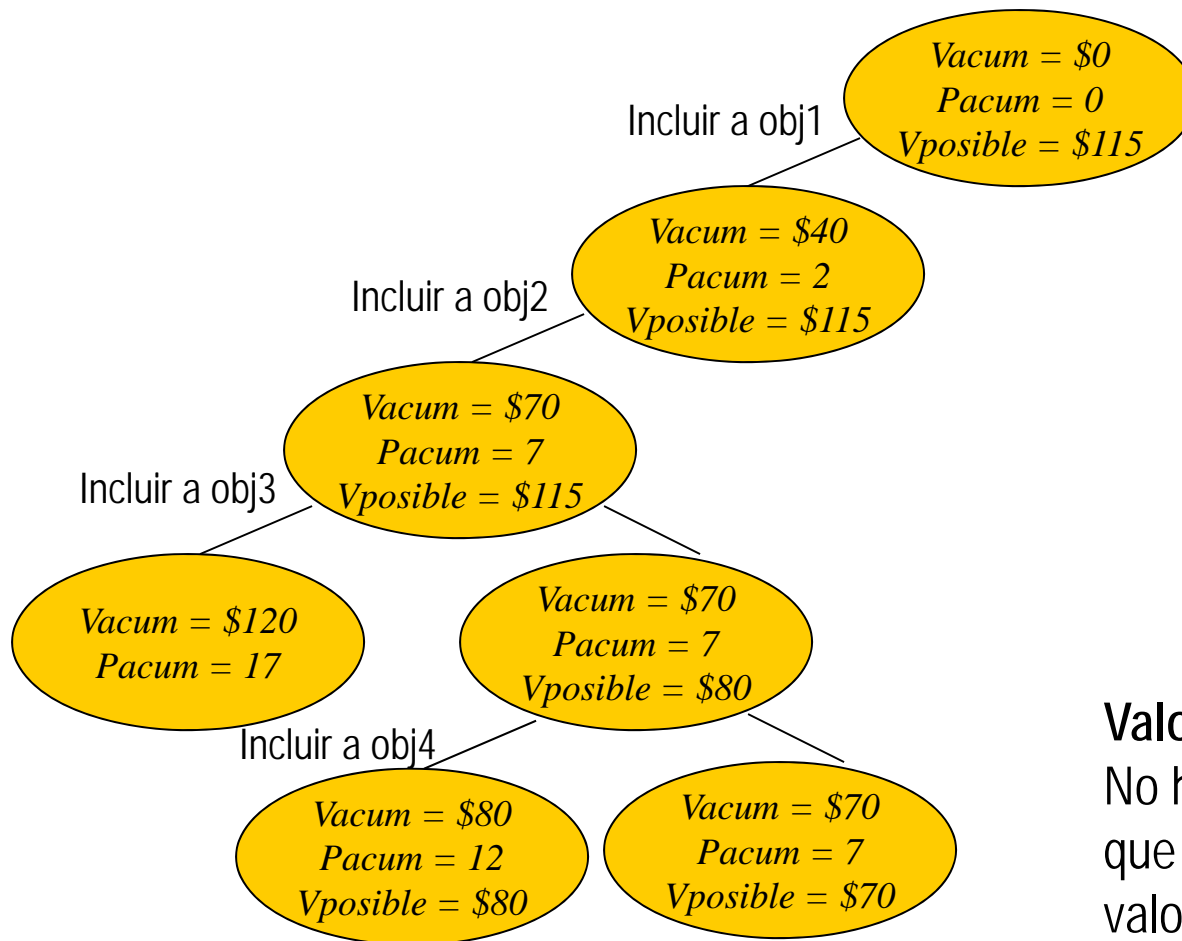
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

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

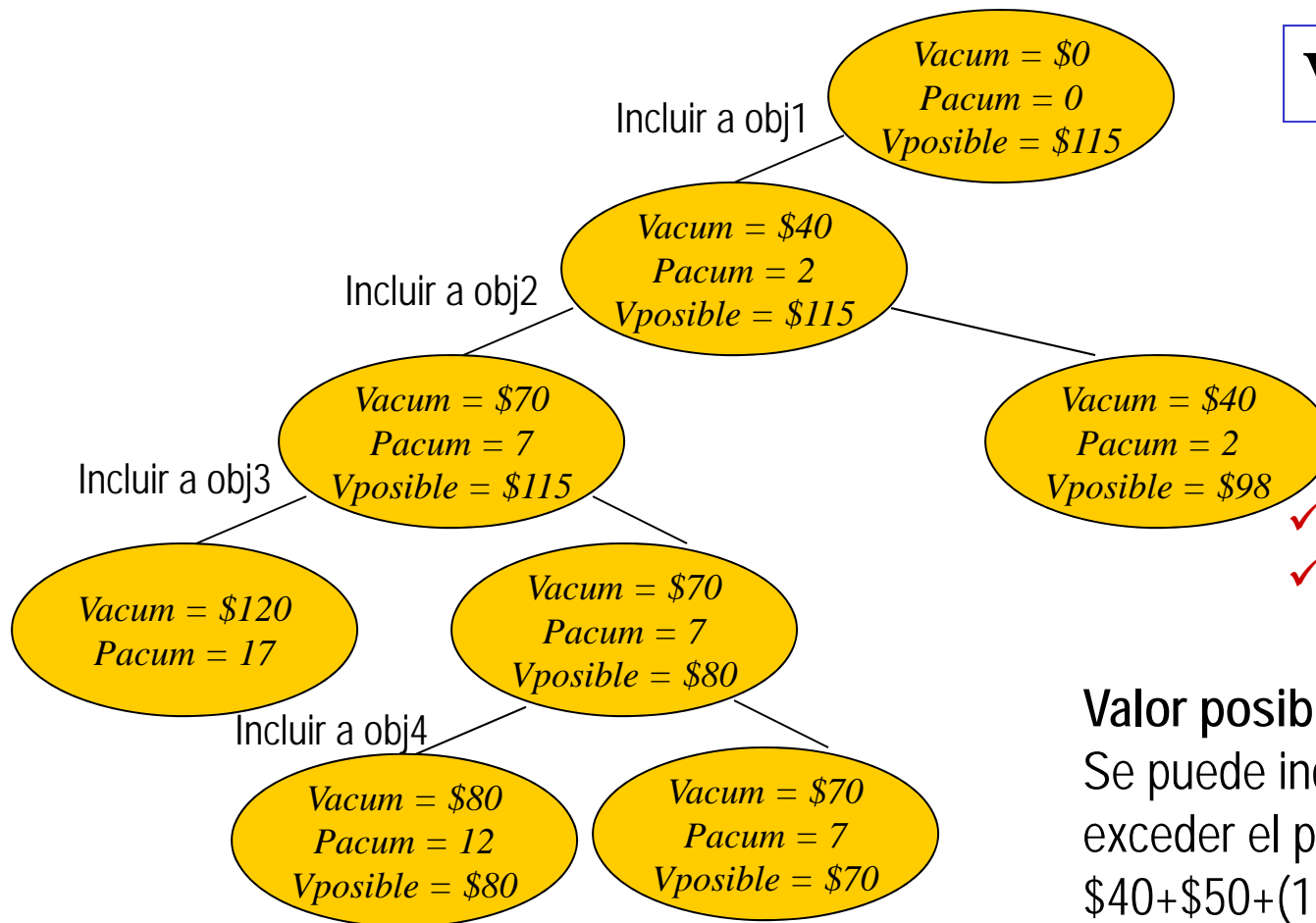
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

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



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

Valor posible a acumular:

Se puede incluir el objeto 3 sin exceder el peso

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

Ejemplo

PESO MOCHILA = 16

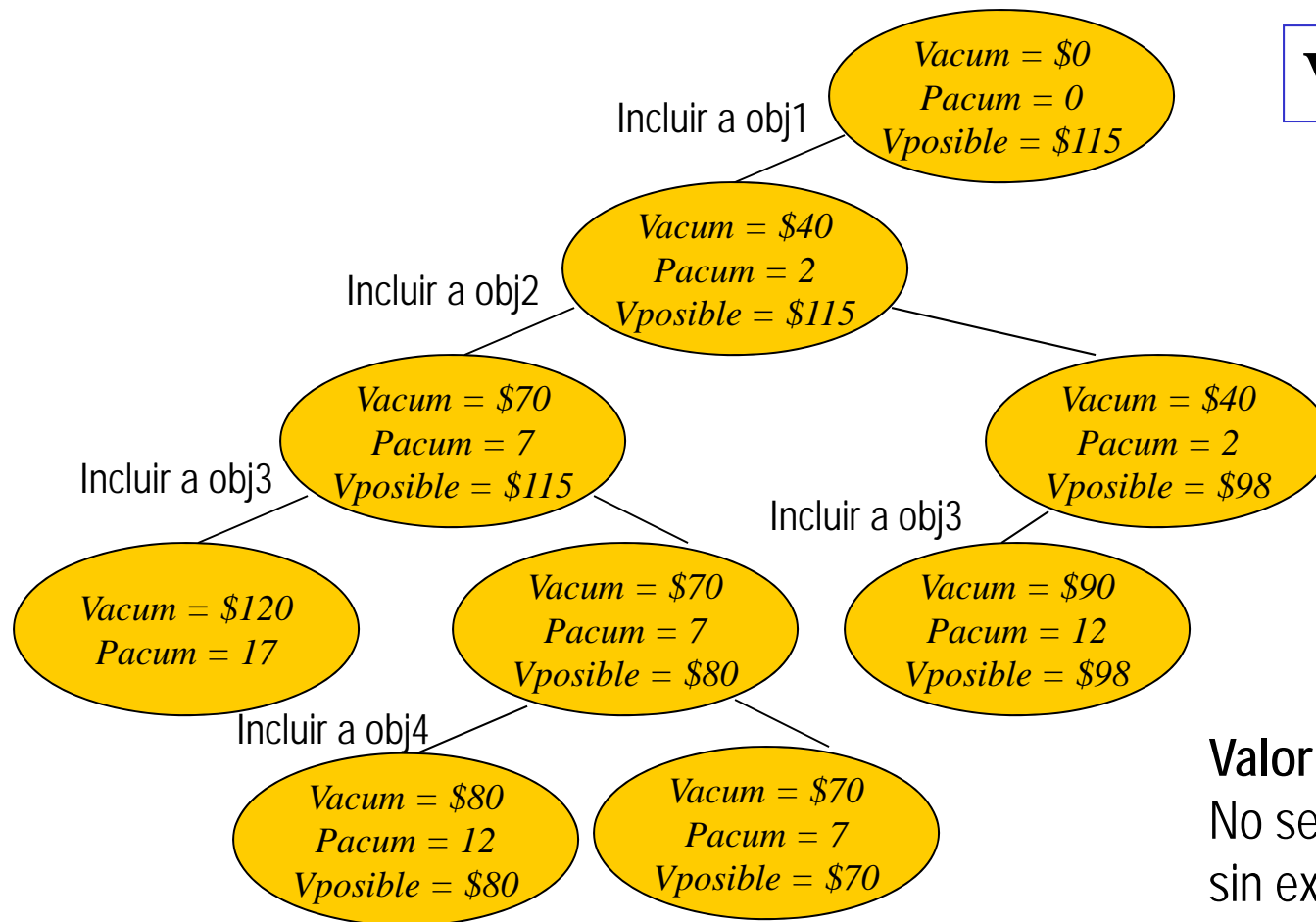
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

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

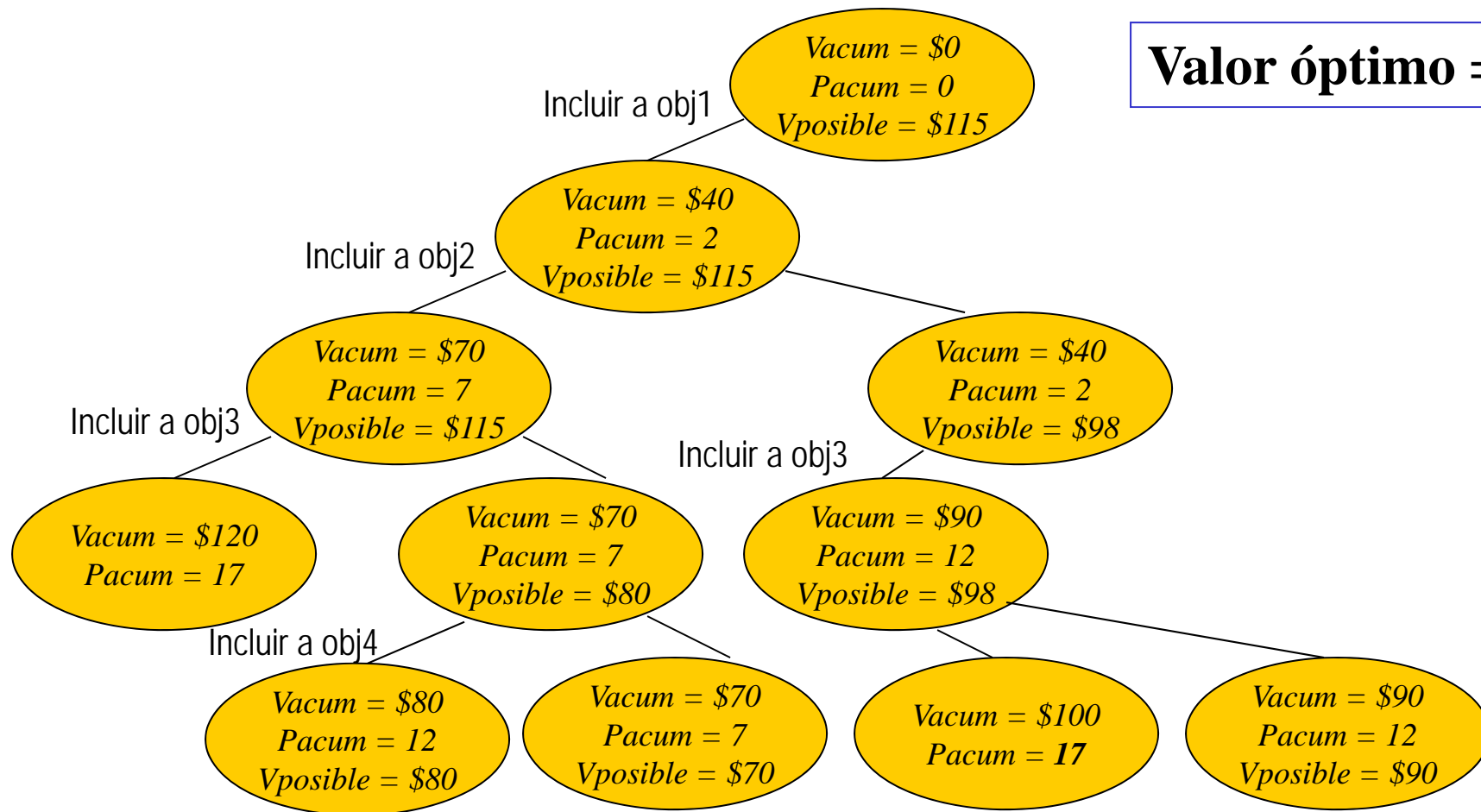
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

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

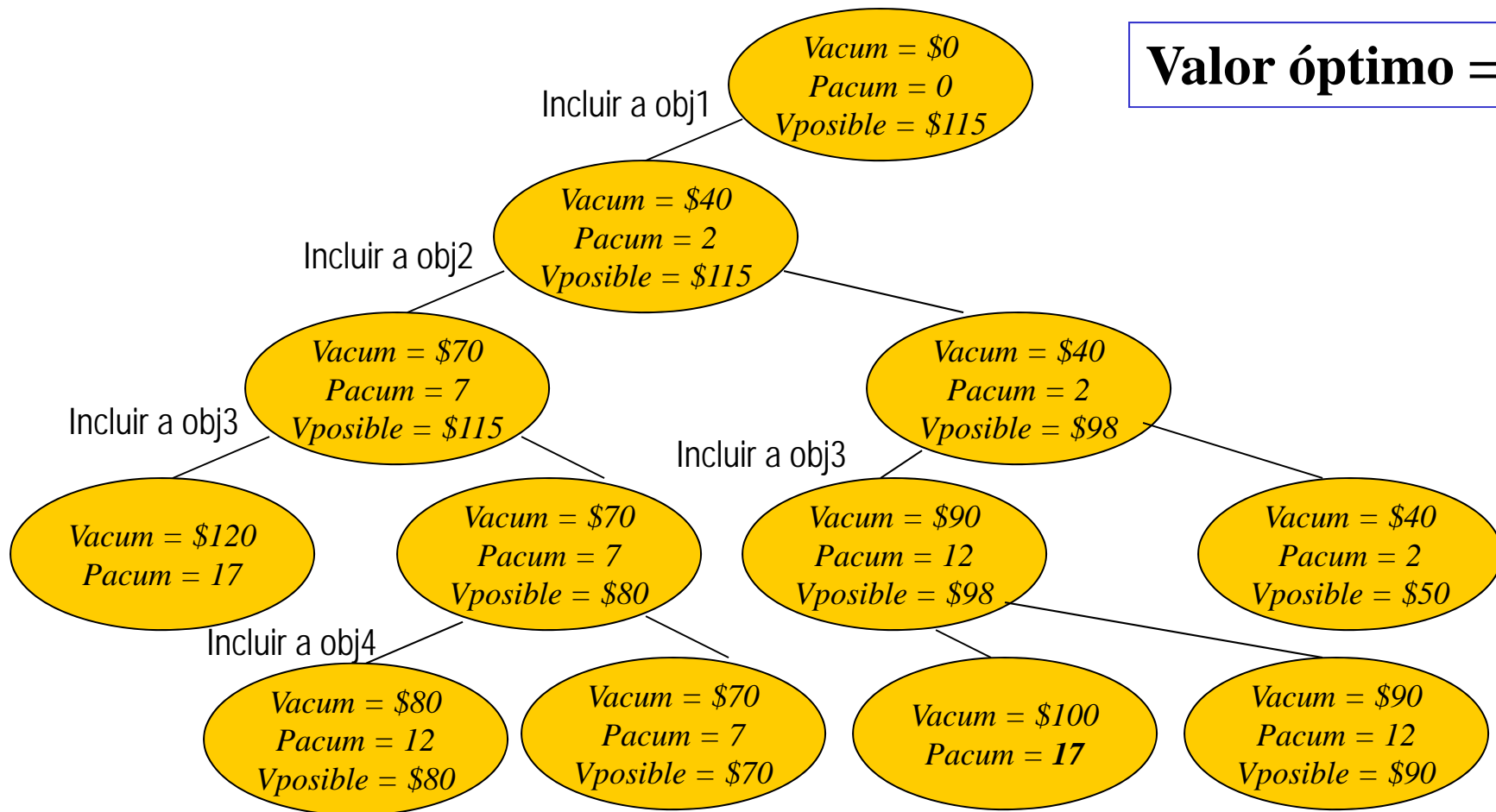
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

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

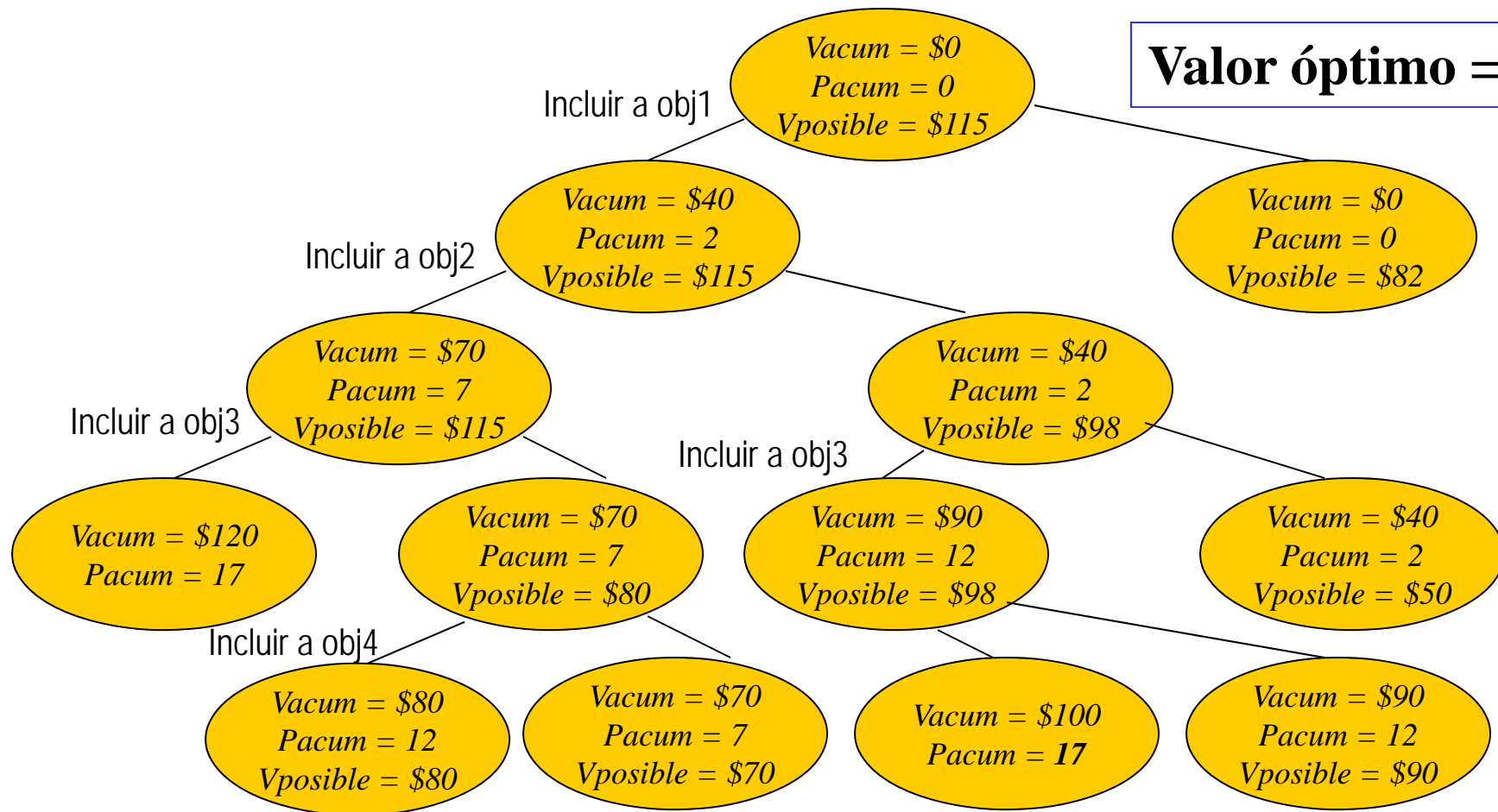
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

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

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

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

Valor óptimo = \$90



Ejemplo

PESO MOCHILA = 16

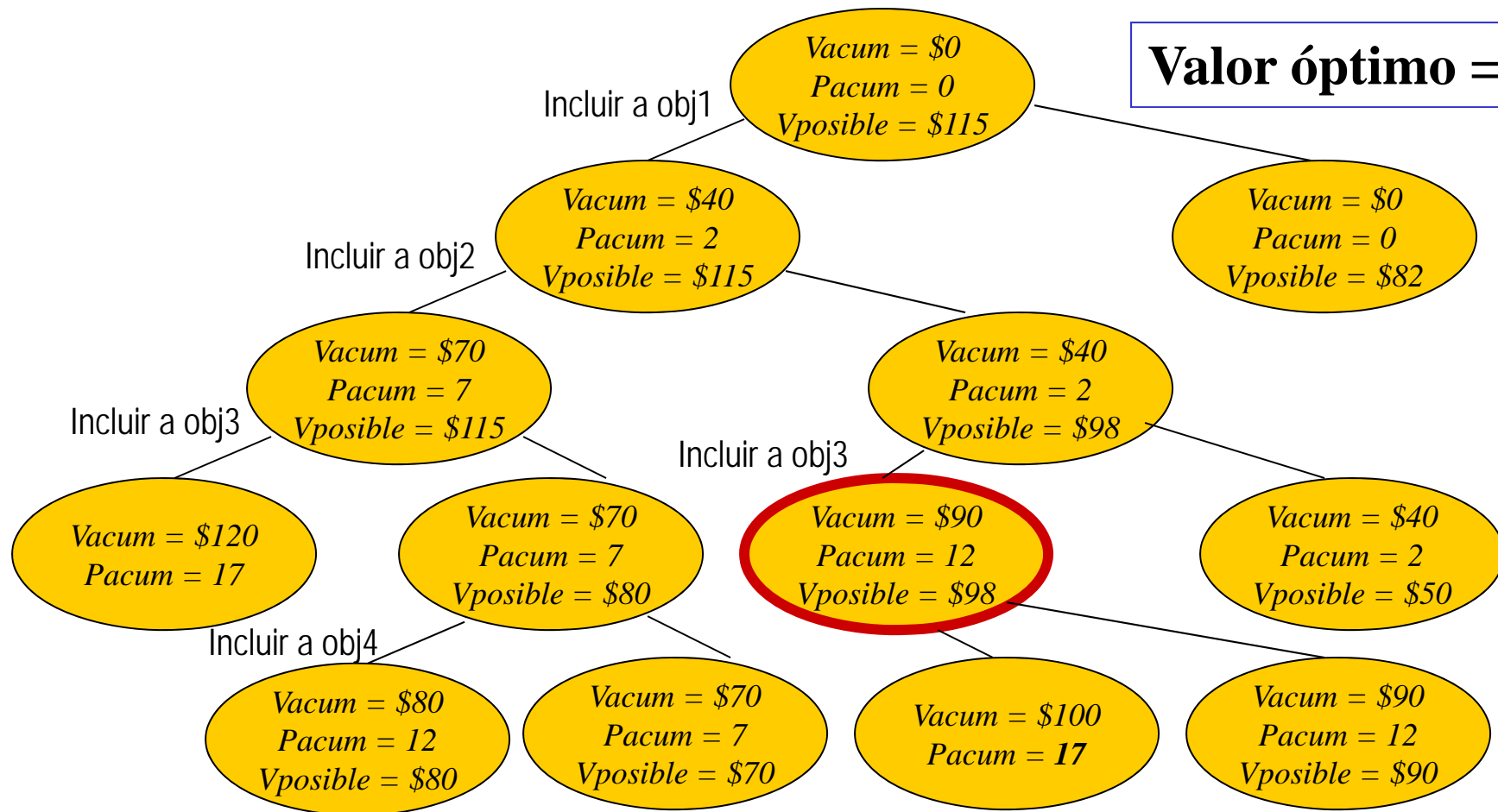
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

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



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 mismos propusieron una mejora utilizando divide&conquer con programación dinámica, obteniendo un algoritmo con $O(2^{n/2})...$

Ejercicio

- Mochila con capacidad de soportar 20 unidades de peso, se tienen los siguientes 3 objetos:
 - Objeto₁, valor₁: \$70, peso₁: 10, valor₁/peso₁ = \$7
 - Objeto₂, valor₂: \$48, peso₂: 8, valor₂/peso₂ = \$6
 - Objeto₃, valor₃: \$55, peso₃: 11, valor₃/peso₃ = \$5

Análisis de Algoritmos



La técnica de
Branch and Bound

Branch and bound

- **Branch and bound** es una técnica muy similar a la de **Backtracking**, pues basa su diseño en el análisis del árbol de búsqueda de soluciones a un problema.
- Sin embargo, no utiliza la búsqueda en profundidad (depth first), y solamente se aplica en problemas de OPTIMIZACIÓN.
- Los algoritmos generados por esta técnica son normalmente de orden exponencial o peor en su peor caso, pero su aplicación ante instancias muy grandes, ha demostrado ser eficiente (incluso más que backtracking).

Diseño de algoritmos con Branch and bound

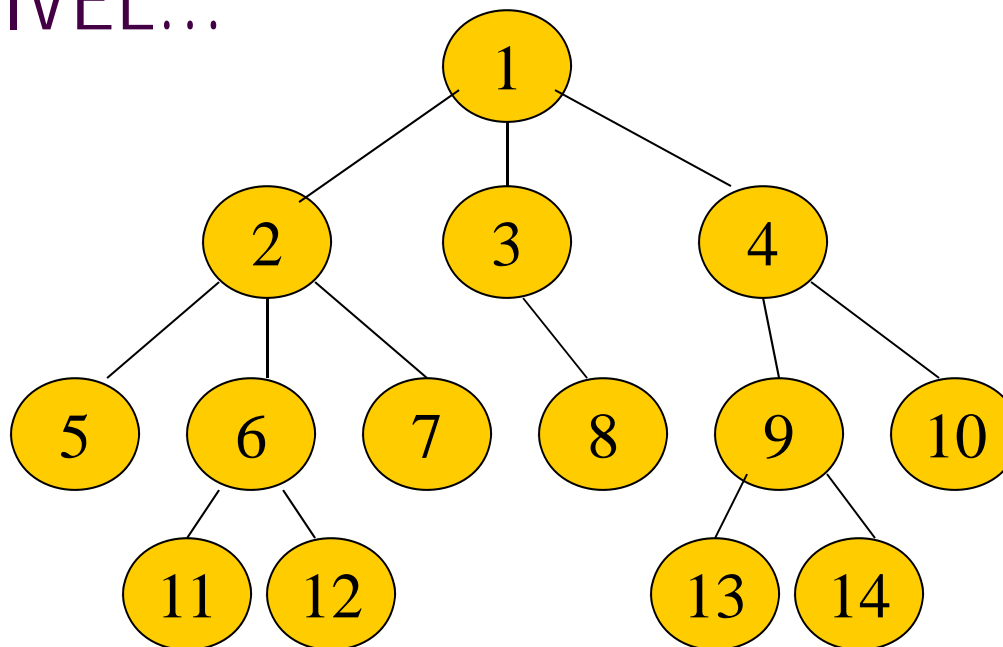


- Decidir de qué manera se conforma el árbol de búsqueda de soluciones.
- Sobre el árbol, se aplicará una búsqueda en anchura (*breadth-first*), pero considerando prioridades en los nodos que se visitan (*best-first*).
- El criterio de selección de nodos, se basa en un **valor óptimo posible** (bound), con el que se toman decisiones para hacer las podas en el árbol.

Recordando...

recorrido *breadth first*

- En un árbol, corresponde al recorrido NIVEL POR NIVEL...



- Y, ¿cómo se implementa el algoritmo?

Algoritmo del recorrido NIVEL x NIVEL

```
void nivelXnivel (Nodo r)  
{  Nodo n, h; Fila f;  
  Meter en f al apuntador r;  
  while (f no esté vacía)  
  { Sacar de f hacia n;  
    Visitar n;  
    for (cada hijo h de n)  
      Meter en f al apuntador h; }  
}
```

Este será el
esqueleto general
de los algoritmos
con Branch and bound

El problema de la mochila

- El problema ya fue resuelto con Programación dinámica, con Backtracking, y ahora se hará con Branch and bound.
- La solución con Backtracking, ya consideró que era un problema de optimización, e incluyó en su criterio de selección de nodos, el proceso de calcular un valor óptimo posible (valor posible a acumular), para tomar las decisiones de podas...
- Esta estrategia servirá pero ahora en el contexto de Branch and bound, lo cual implica que se realizará la búsqueda nivel por nivel en vez de primero en profundidad.

Problema de la mochila



- Árbol 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...

Valor óptimo posible (valor posible a acumular)

- *Análisis idéntico a lo que se hizo con backtracking:*
 - 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, 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

Análisis de Algoritmos

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

$V_a = \$0$
 $P_a = 0$
 $V_p = \$115$

Valor óptimo = \$0

✓ $P_{acum} < 16$

✓ $V_{posible} > \text{Valor óptimo}$

Valor posible a acumular:

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

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

Análisis de Algoritmos

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

$Va = \$0$

$Pa = 0$

$Vp = \$115$

$Va = \$40$

$Pa = 2$

$Vp = \$115$

Valor óptimo = \$40

Valor posible a acumular:

Se pueden acumular el objeto 2 sin exceder el peso.

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

✓ $P_{acum} < 16$

✓ $V_{posible} > \text{Valor óptimo}$

Ejemplo

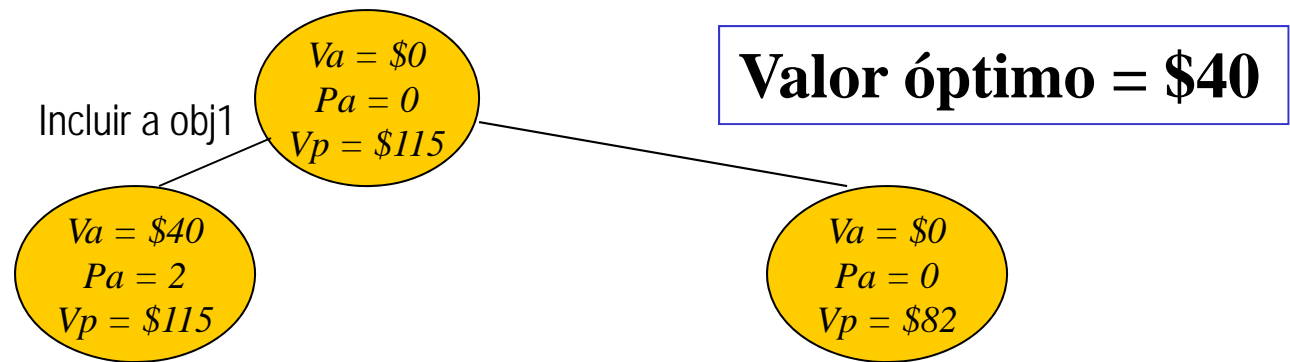
PESO MOCHILA = 16

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 posible a acumular:

Se pueden acumular el objeto 2 y 3 sin exceder el peso.

$$\$30 + \$50 + (16 - 5 - 10) * \$10/5 = \$82$$

✓ Pacum < 16

✓ Vposible > Valor óptimo

Análisis de Algoritmos

Ejemplo

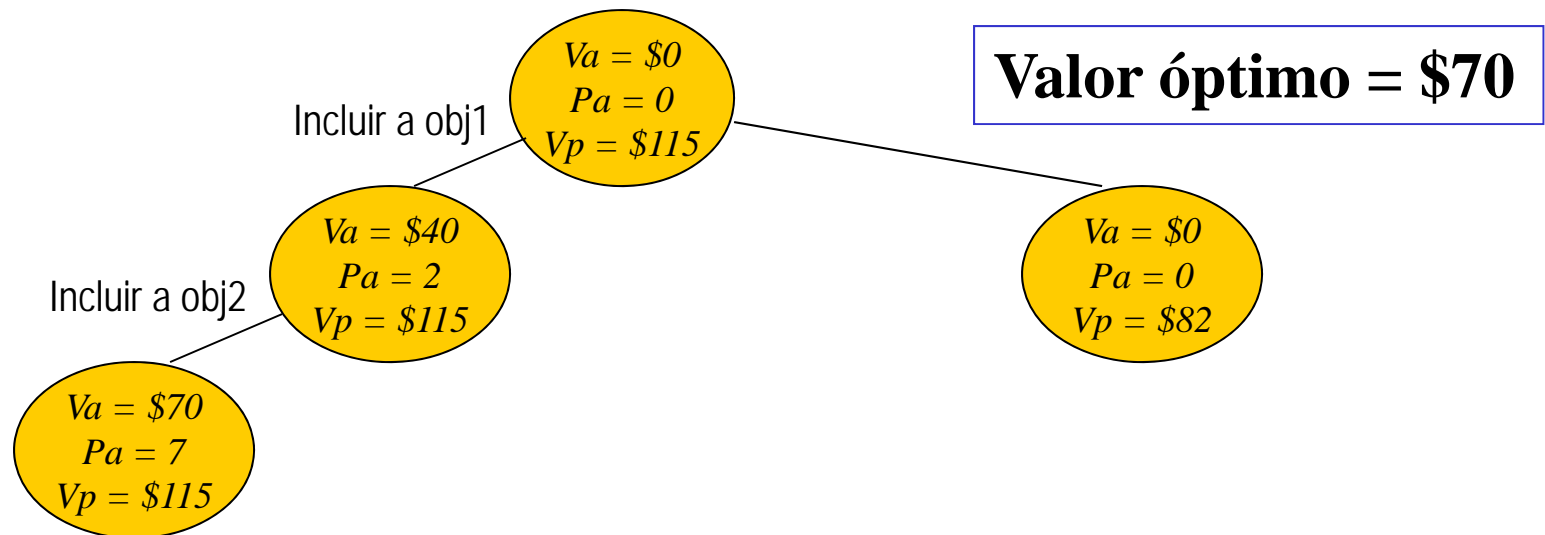
PESO MOCHILA = 16

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



✓ Pacum < 16

✓ Vposible > Valor óptimo

Valor posible a acumular:

NO se pueden acumular más objetos sin exceder el peso.

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

Ejemplo

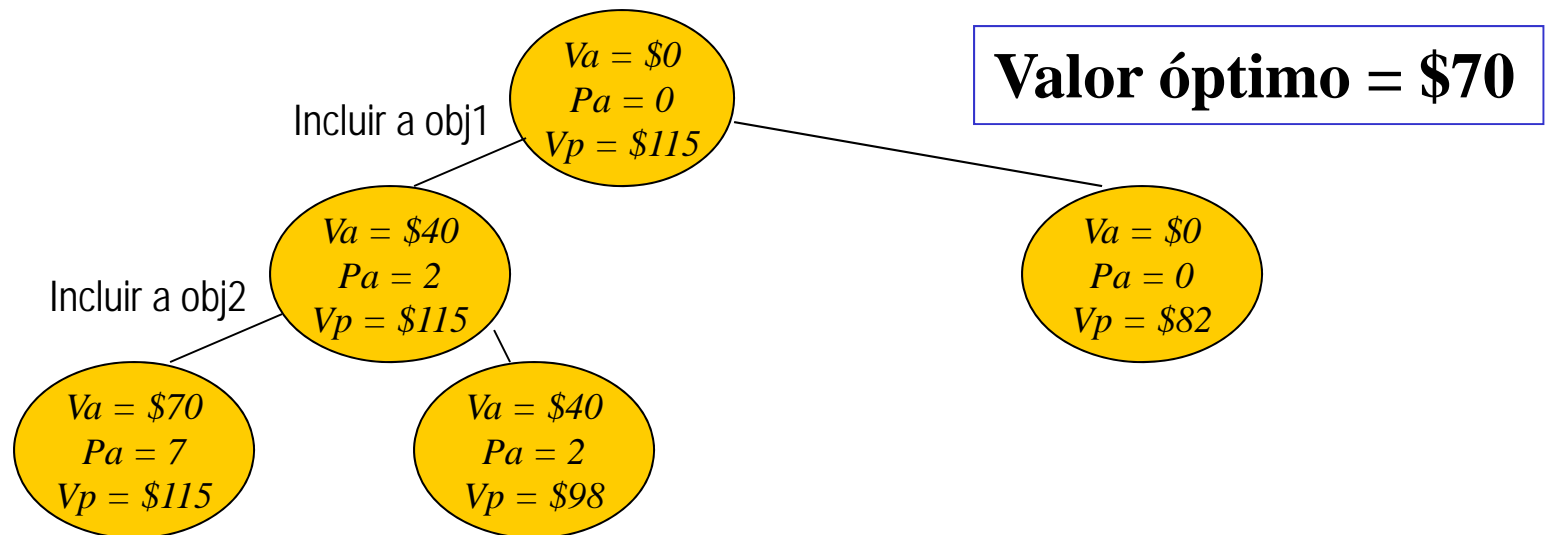
PESO MOCHILA = 16

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



- ✓ 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

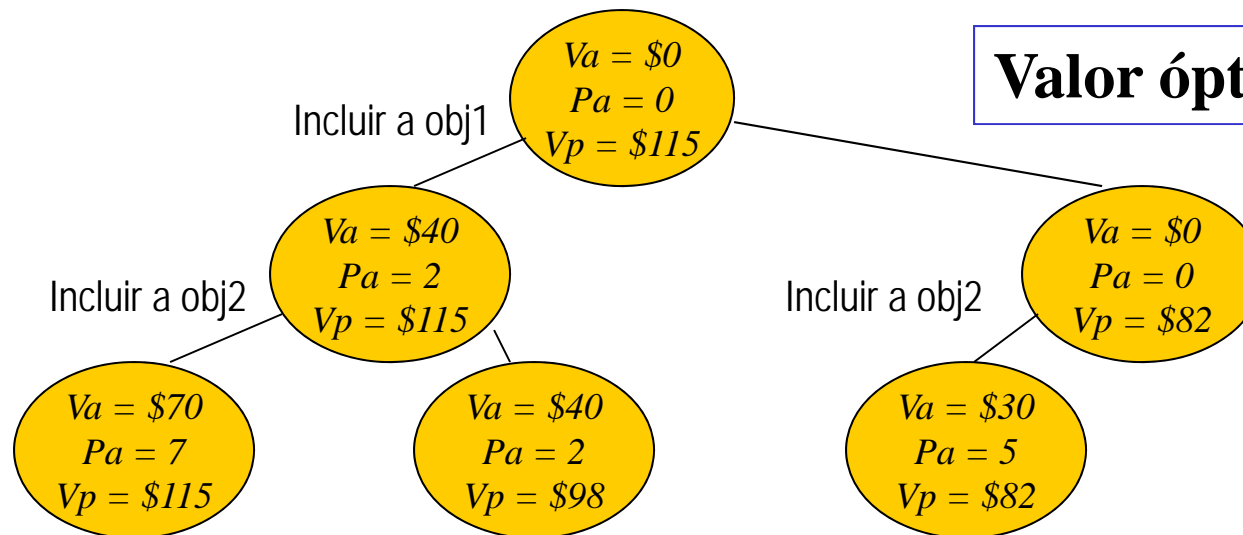
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

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 = \$70



- ✓ $P_{acum} < 16$
- ✓ $V_{posible} > \text{Valor óptimo}$

Valor posible a acumular:

Se puede incluir el objeto 3 sin exceder el peso

$$\$30 + \$50 + (16 - 5 - 10) \cdot \$10/5 = \$82$$

Análisis de Algoritmos

Ejemplo

PESO MOCHILA = 16

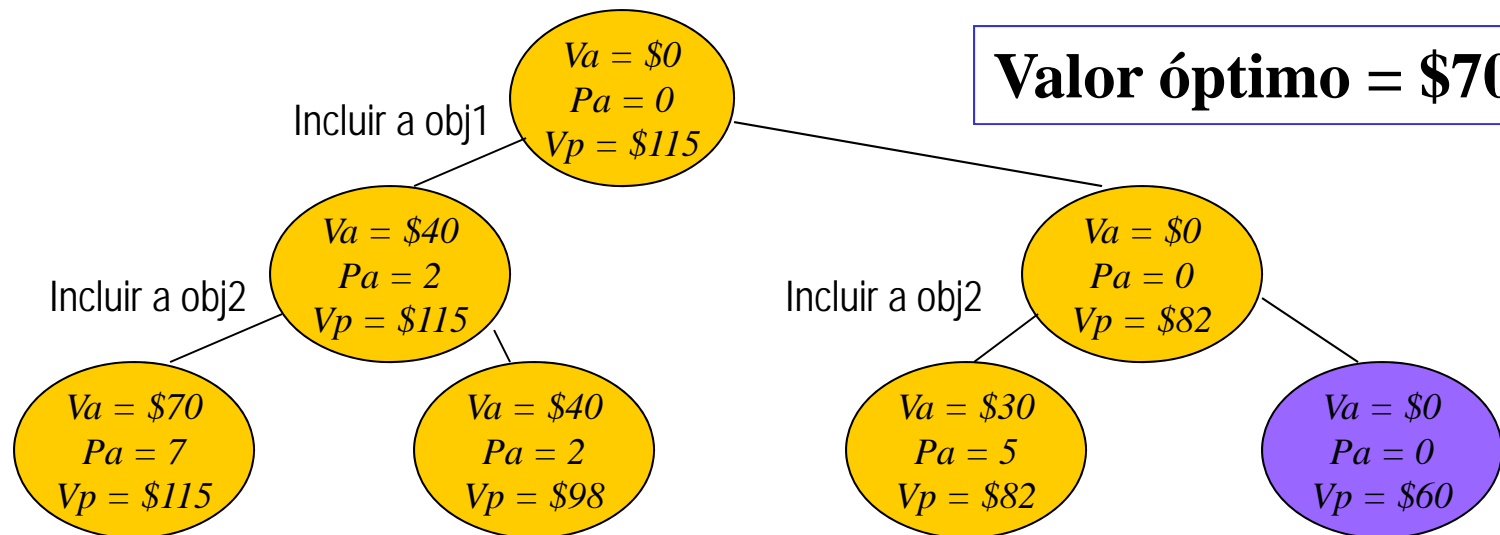
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

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



✓ $Pacum < 16$

✗ $Vposible > \text{Valor óptimo}$

Valor posible a acumular:

Se pueden incluir los objeto 3 y 4 sin exceder el peso

$$\$50 + \$10 + (16 - 5 - 10) * \$0 = \$60$$

Análisis de Algoritmos

Ejemplo

PESO MOCHILA = 16

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

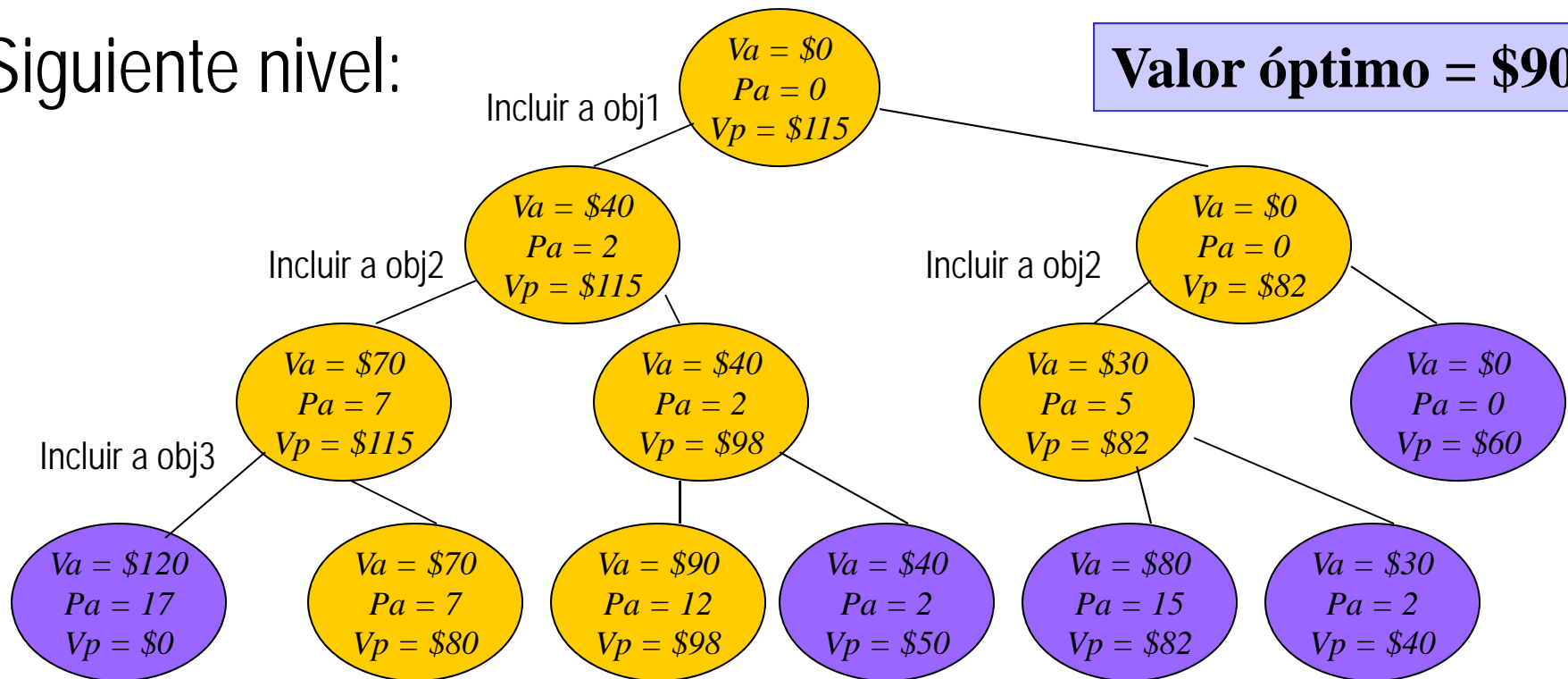
Objeto₂, valor₂: \$30, peso₂: 5, valor₂/peso₂ = \$6

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

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

Siguiente nivel:

Valor óptimo = \$90



Análisis de Algoritmos

Ejemplo

PESO MOCHILA = 16

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

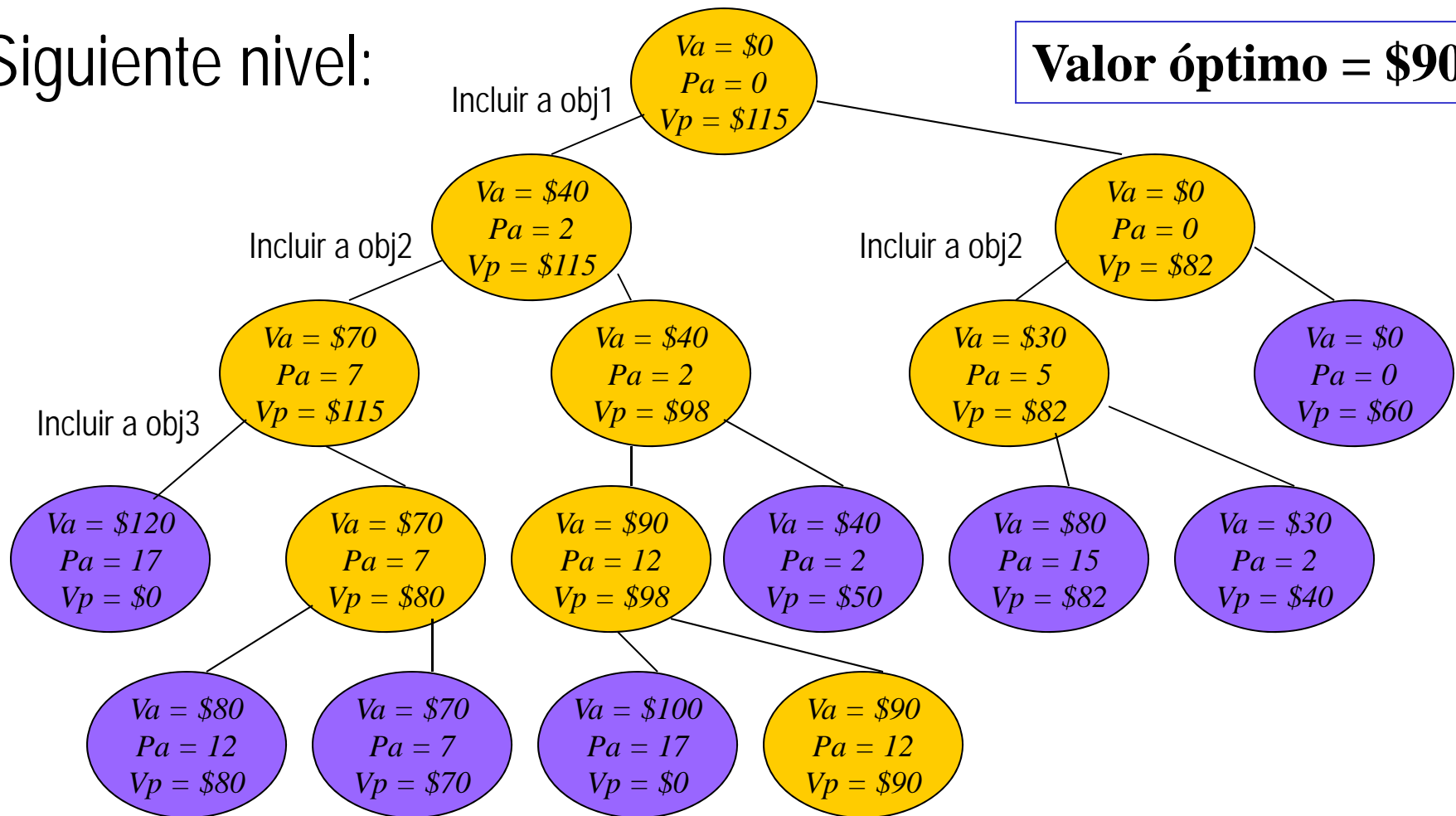
Objeto₂, valor₂: \$30, peso₂: 5, valor₂/peso₂ = \$6

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

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

Siguiente nivel:

Valor óptimo = \$90



Análisis de Algoritmos

Ejemplo

PESO MOCHILA = 16

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

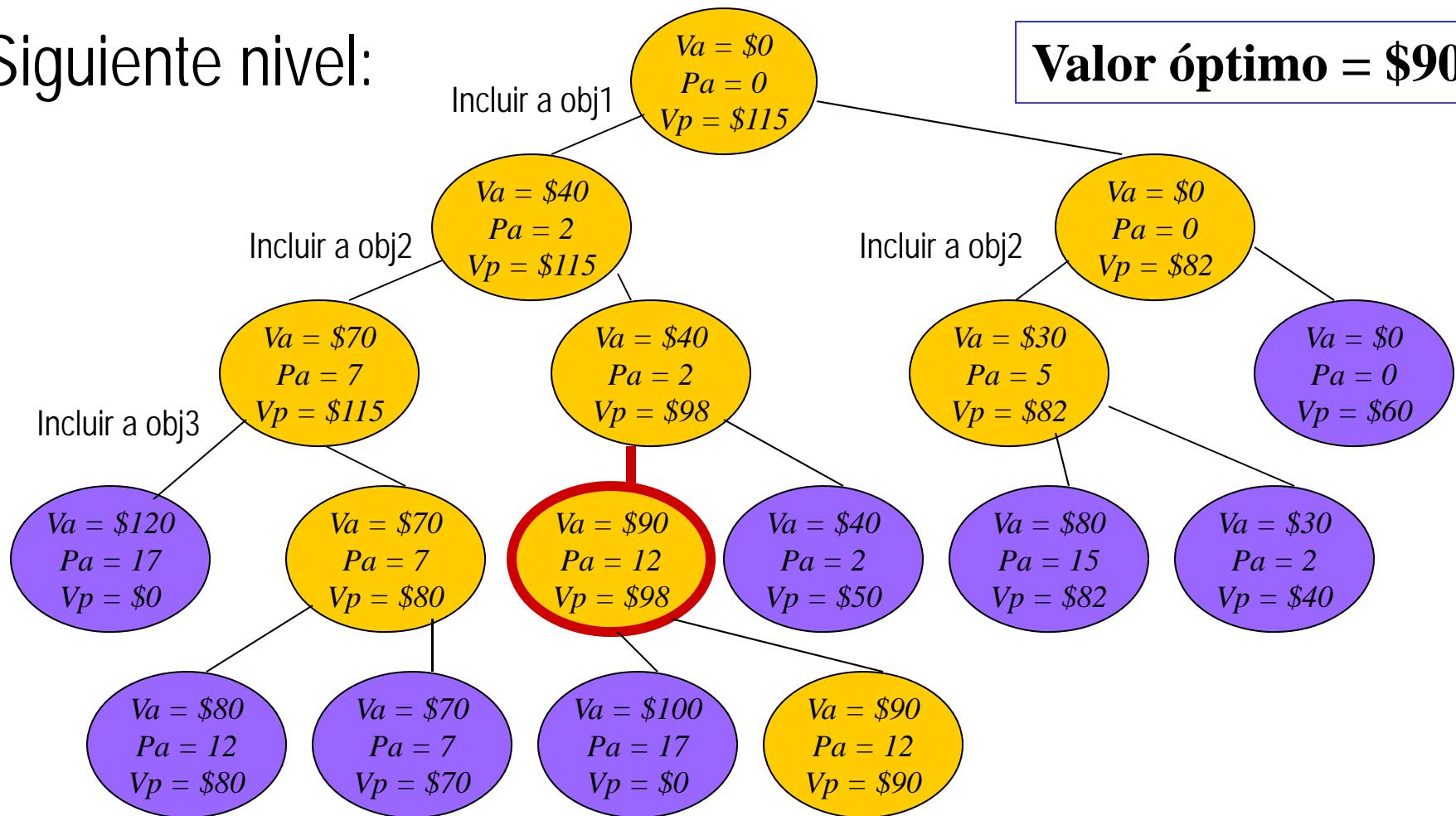
Objeto₂, valor₂: \$30, peso₂: 5, valor₂/peso₂ = \$6

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

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

Siguiente nivel:

Valor óptimo = \$90



Análisis del ejemplo

- El árbol de búsqueda de solución por Backtacking analizó 13 nodos...
- Por Branch and bound analizó 17 nodos...
- Aparentemente NO hay beneficio...
- Por lo tanto, se tiene que mejorar la técnica...
- ¿Cómo?
- Aprovechando la visita de todo un nivel, escoger la expansión del "mejor nodo" (best first)...

Análisis de Algoritmos

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

$Va = \$0$
 $Pa = 0$
 $Vp = \$115$

Valor óptimo = \$0

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

$Va = \$0$
 $Pa = 0$
 $Vp = \$115$

$Va = \$40$
 $Pa = 2$
 $Vp = \$115$

Valor óptimo = \$40

Análisis de Algoritmos

Ejemplo

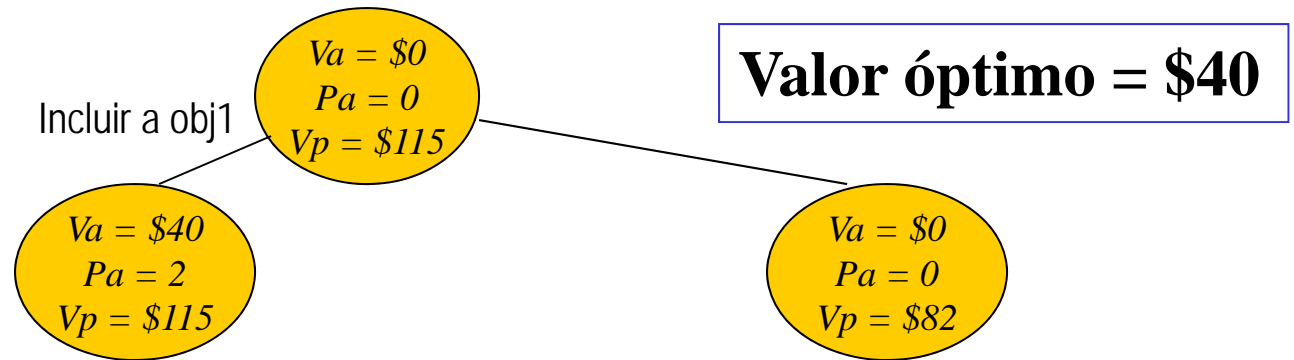
PESO MOCHILA = 16

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



Análisis de Algoritmos

Ejemplo

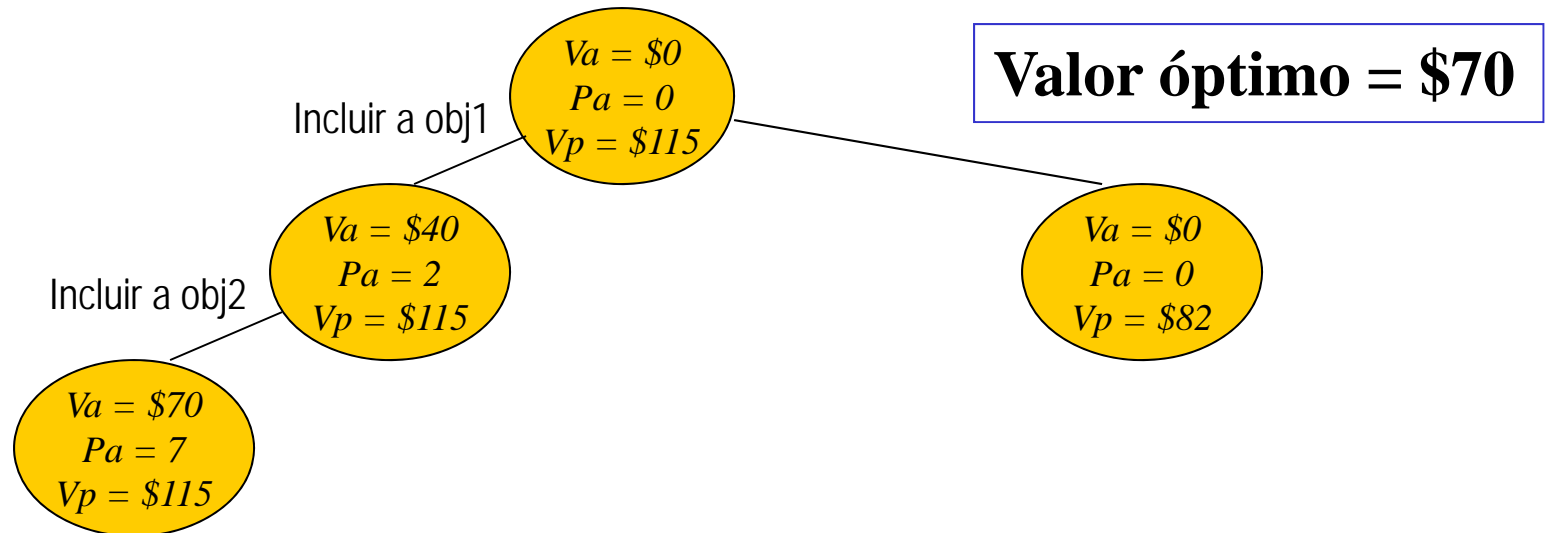
PESO MOCHILA = 16

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

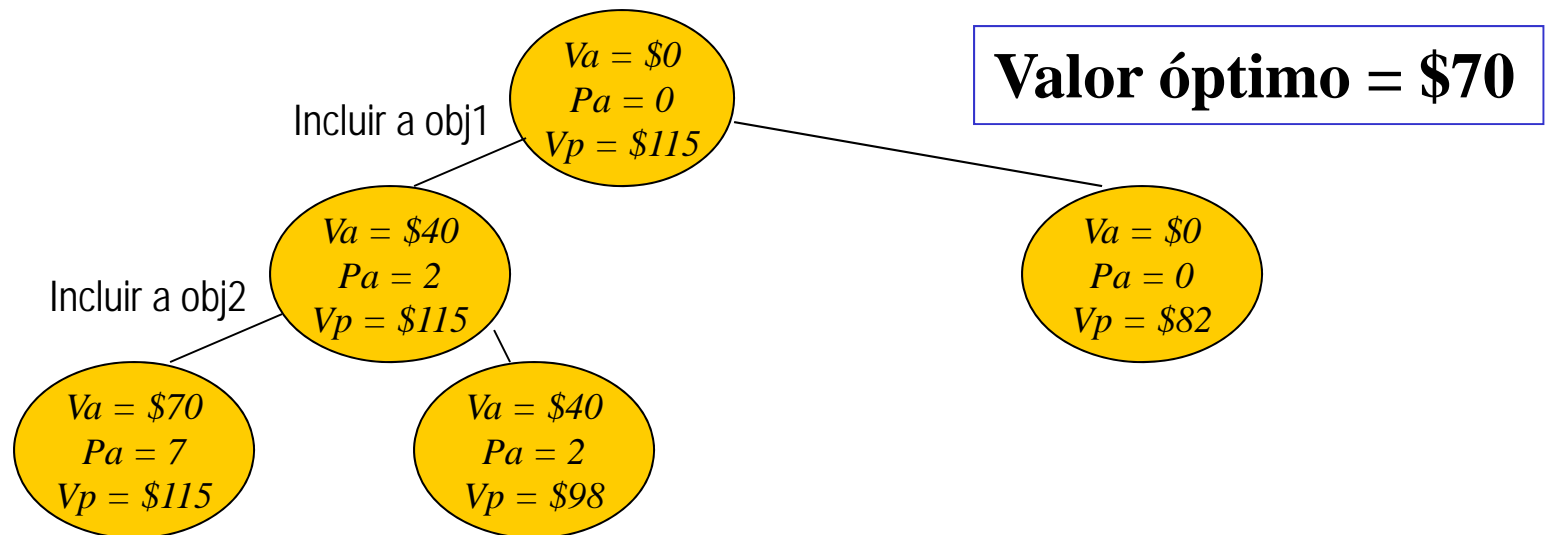
PESO MOCHILA = 16

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



De los nodos a expandir,
¿cuál tiene el mejor valor posible?

Ejemplo

PESO MOCHILA = 16

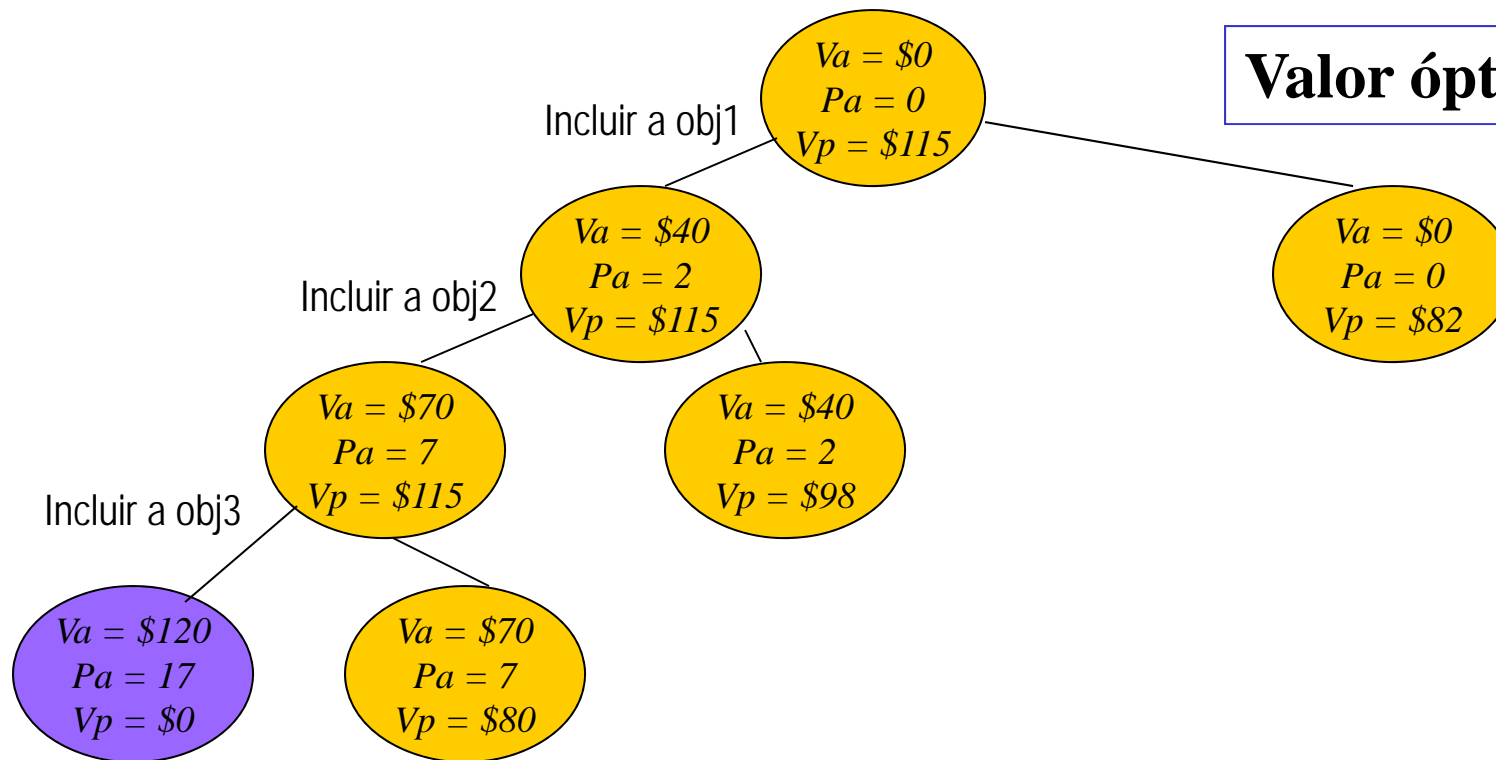
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

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



De los nodos a expandir,
¿cuál tiene el mejor valor posible?

Análisis de Algoritmos

Ejemplo

PESO MOCHILA = 16

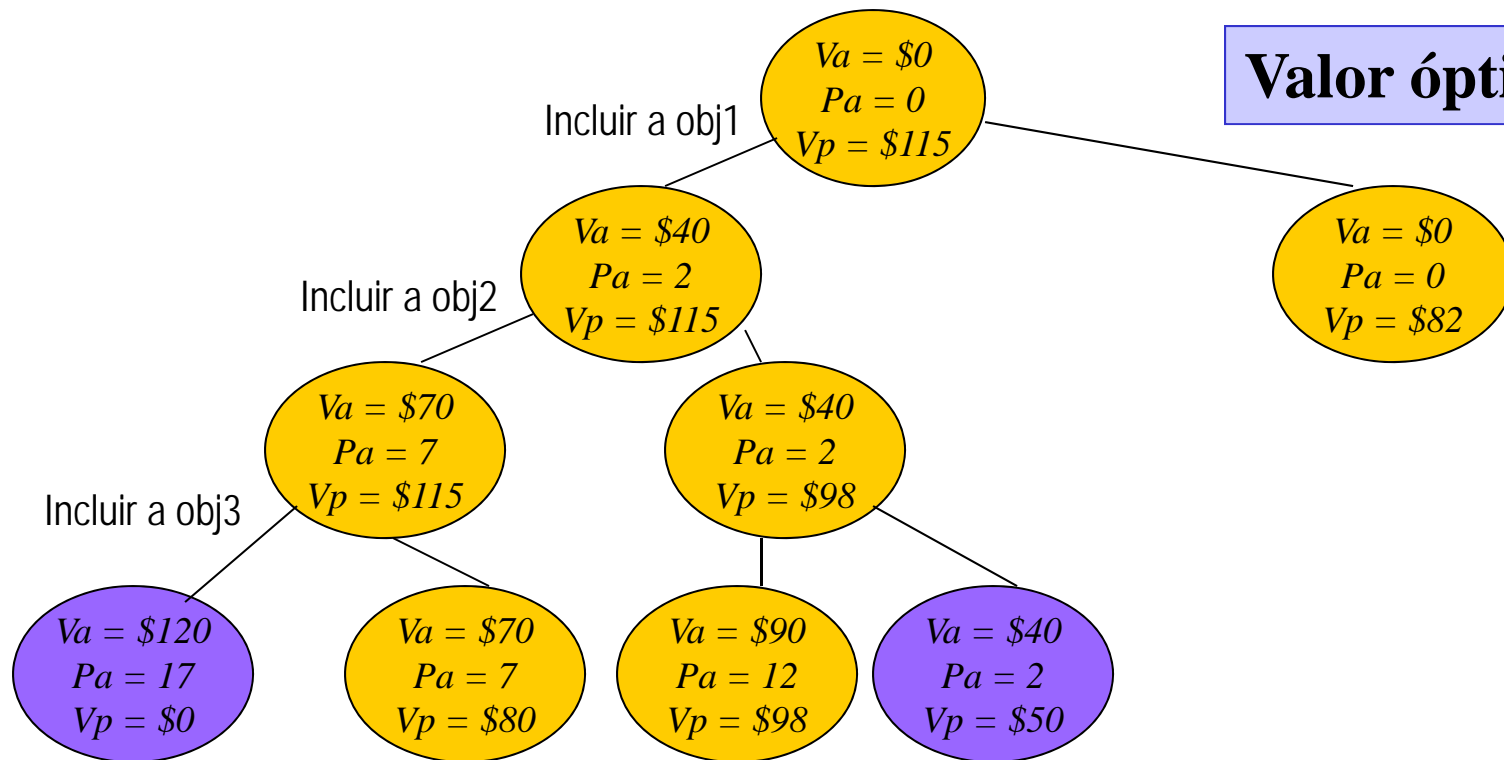
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

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



De los nodos a expandir,
¿cuál tiene el mejor valor posible?

Análisis de Algoritmos

Ejemplo

PESO MOCHILA = 16

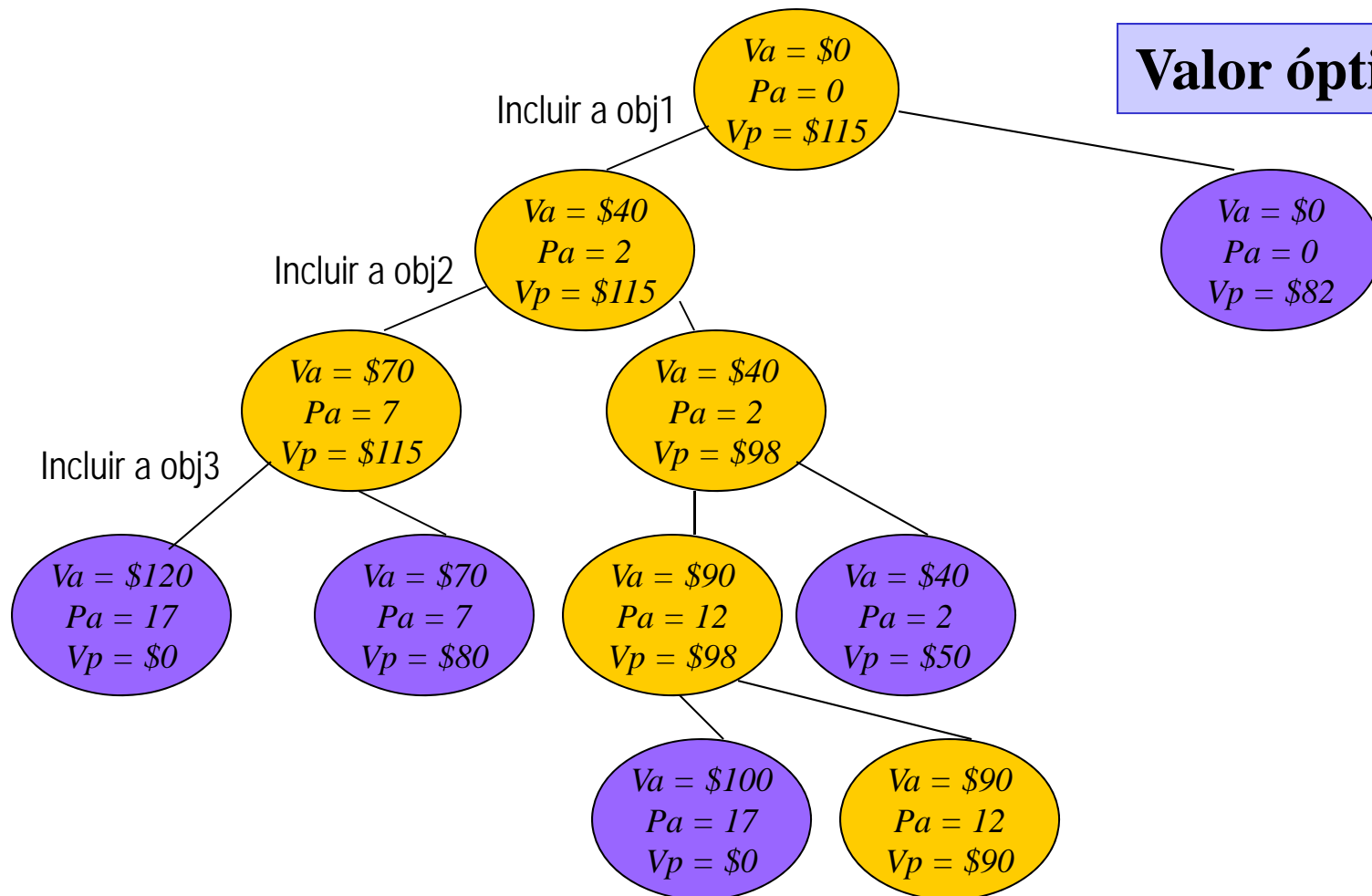
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

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



Análisis de Algoritmos

Ejemplo

PESO MOCHILA = 16

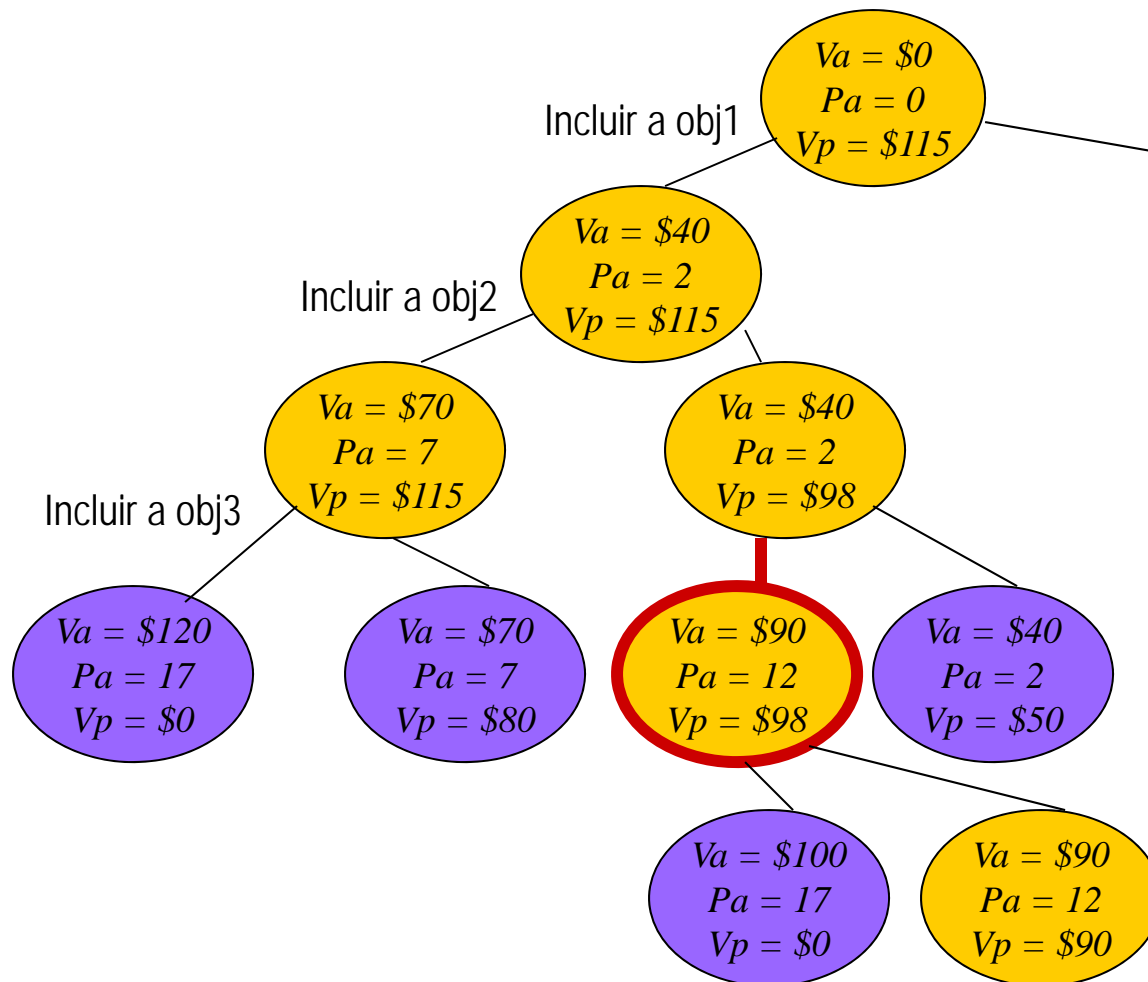
Objeto₁, valor₁: \$40, peso₁: 2, valor₁/peso₁ = \$20

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



Solución que requirió sólo 11 nodos

Implementación del algoritmo



- Tomando como base el algoritmo del recorrido nivel por nivel, se utilizará una **cola priorizada** en vez de una fila...
- La prioridad la tendrá el nodo con mayor valor posible a acumular...
- El algoritmo se adapta al contexto del problema, considerando la formación de nodos, y su inserción y eliminación de la cola priorizada.
- Los nodos sólo guardan los siguientes valores de información: nivel, valor acumulado, peso acumulado y valor posible a acumular... (no hay necesidad de tener apuntadores).

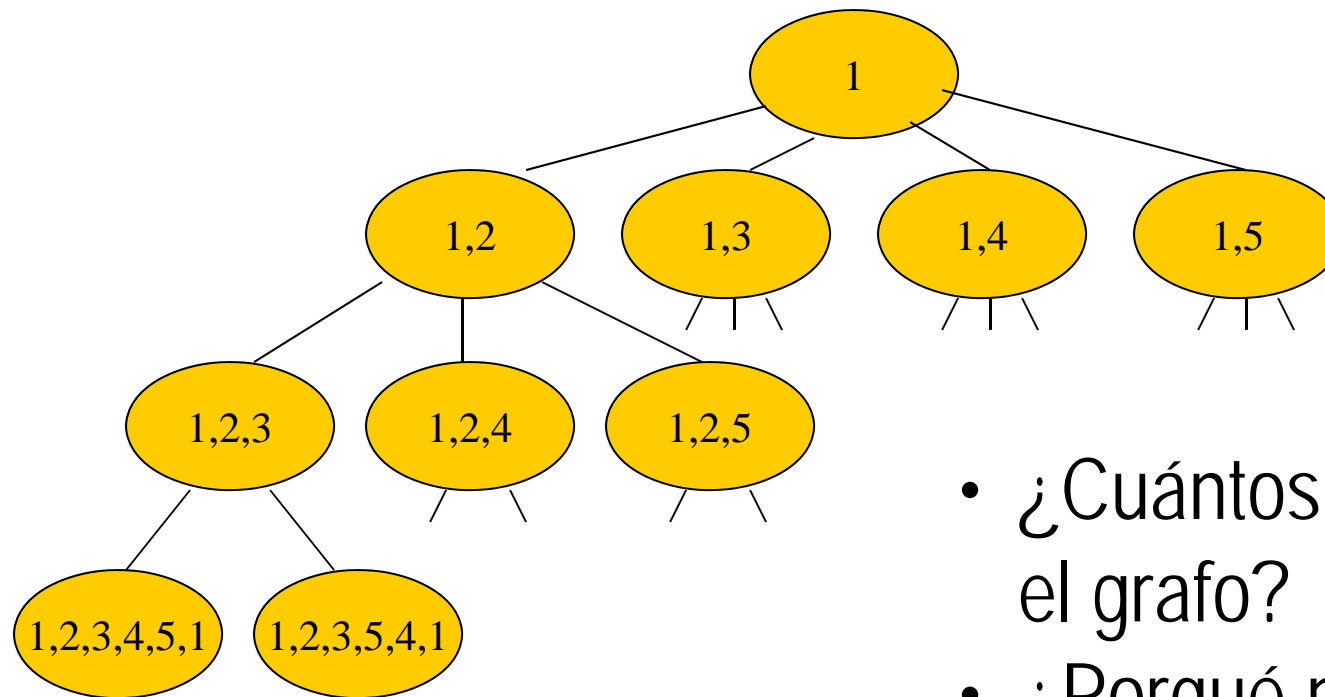
El problema del viajero

- Este problema fue resuelto con la Programación dinámica, obteniendo un algoritmo de orden $O(n^2 2^n)$...
- Para una 'n' grande, el algoritmo es ineficiente...
- Con Backtracking, se resolvió el problema de los ciclos Hamiltonianos, que indirectamente, resuelve el problema del viajero...
- Sin embargo, también puede llegar a tener un comportamiento exponencial o peor...
- Branch and bound se adapta para solucionarlo...

El problema del viajero

- *Recordando:* Ciclo en el grafo en el que TODOS los vértices del grafo se visitan sólo una vez al menor costo.
- Árbol 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.

Ejemplo



- ¿Cuántos vértices tiene el grafo?
- ¿Porqué no se requiere el último nivel en el árbol?

Análisis del problema con Branch and bound

- Criterio de selección para expandir un nodo del árbol de búsqueda de soluciones:
 - Un vértice en el nivel i del árbol, debe ser adyacente al vértice en el nivel $i-1$ del camino correspondiente en el árbol.
 - Puesto que es un problema de MINIMIZACIÓN, si el costo posible a acumular al expandir el nodo i , es **menor** al mejor costo acumulado hasta ese momento, vale la pena expandir el nodo, si no, el camino ahí se deja de explorar...

Estimación del costo posible a acumular

- Si se sabe cuáles son los vértices que faltan por visitar...
- Cada vértice faltante, tiene arcos de salida hacia otros vértices...
- El mejor costo, será el del arco que tenga el valor menor...
- Esta información se puede obtener del renglón correspondiente al vértice en la matriz de adyacencias (excluyendo a los valores de cero)...
- La sumatoria de los mejores arcos de cada vértice faltante, más el costo del camino ya acumulado, es un estimado válido para tomar decisiones respecto a las podas en el árbol..

- | | | | | | | | |
|----|----|----|----|----|---|----------|-------------------|
| 0 | 14 | 4 | 10 | 20 | → | Mínimo = | 4 |
| 14 | 0 | 7 | 8 | 7 | → | Mínimo = | 7 |
| 4 | 5 | 0 | 7 | 16 | → | Mínimo = | 4 |
| 11 | 7 | 9 | 0 | 2 | → | Mínimo = | 2 |
| 18 | 7 | 17 | 4 | 0 | → | Mínimo = | 4 |
| | | | | | | | <hr/> |
| | | | | | | | TOTAL = 21 |

Ejemplo

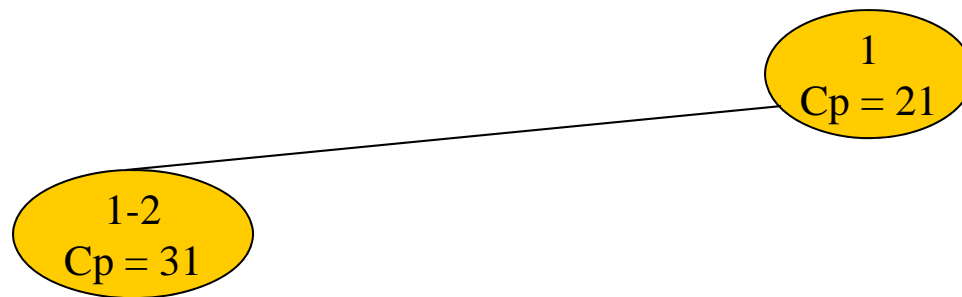
0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0

1
 $C_p = 21$

Costo mínimo = ∞

Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



Costo mínimo = ∞

Cálculo del Costo posible:

Acumulado de 1-2 : 14

Más mínimo de 2-3, 2-4 y 2-5: 7

Más mínimo de 3-1, 3-4 y 3-5: 4

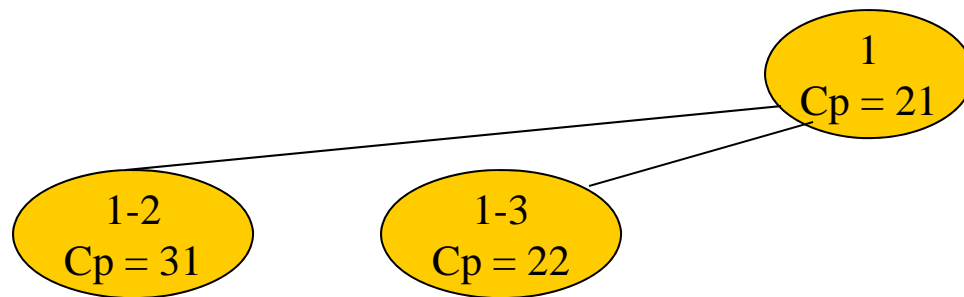
Más mínimo de 4-1, 4-3 y 4-5: 2

Más mínimo de 5-1, 5-3 y 5-4: 4

TOTAL = 31

Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



Costo mínimo = ∞

Cálculo del Costo posible:

Acumulado de 1-3 : 4

Más mínimo de 3-2, 3-4 y 3-5: 5

Más mínimo de 2-1, 2-4 y 2-5: 7

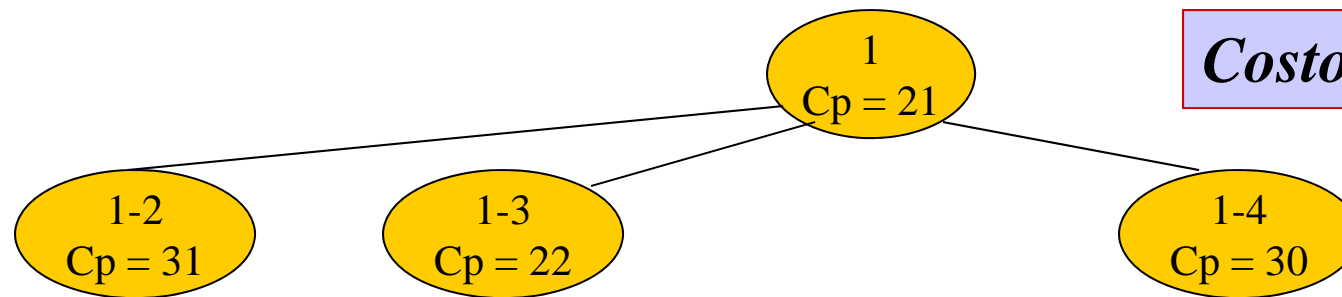
Más mínimo de 4-1, 4-2 y 4-5: 2

Más mínimo de 5-1, 5-2 y 5-4: 4

TOTAL = 22

Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



Costo mínimo = ∞

Cálculo del Costo posible:

Acumulado de 1-4 : 10

Más mínimo de 4-2, 4-3 y 4-5: 2

Más mínimo de 3-1, 3-2 y 3-5: 4

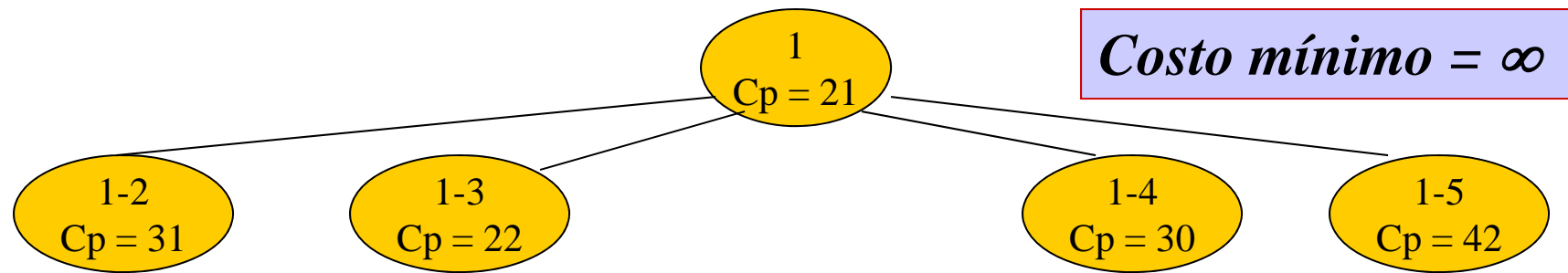
Más mínimo de 2-1, 2-3 y 2-5: 7

Más mínimo de 5-1, 5-2 y 5-3: 7

TOTAL = 30

Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



¿Cuál es el mejor nodo
para expandir?

Cálculo del Costo posible:

Acumulado de 1-5 : 20

Más mínimo de 5-2, 5-3 y 5-4: 4

Más mínimo de 4-1, 4-2 y 4-3: 7

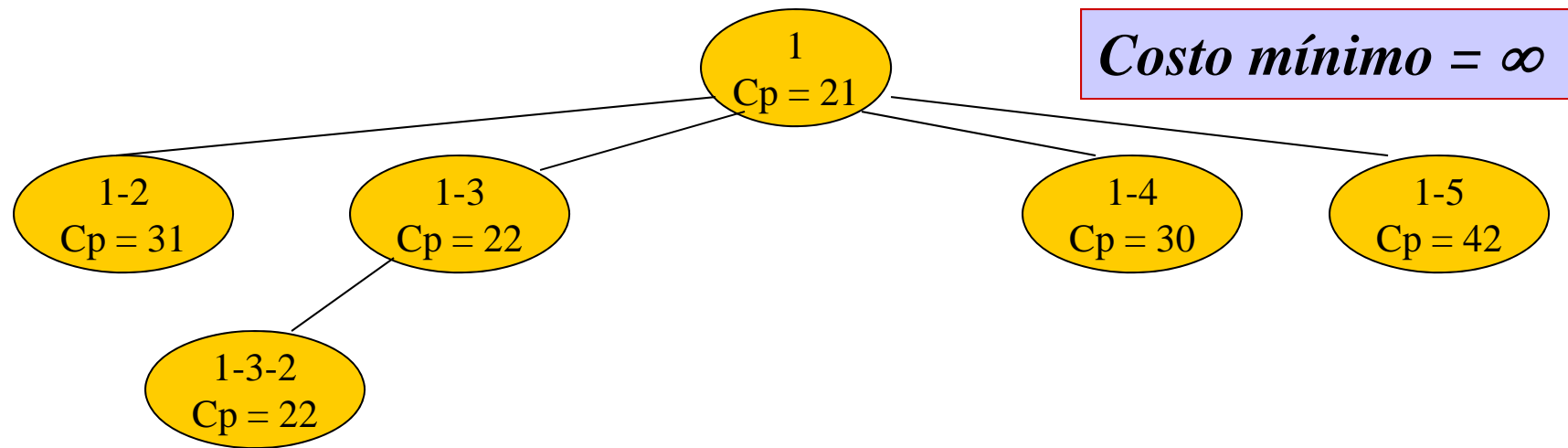
Más mínimo de 3-1, 3-2 y 3-4: 4

Más mínimo de 2-1, 2-3 y 2-4: 7

TOTAL = 42

Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



Cálculo del Costo posible:

Acumulado de 1-3-2 : 9

Más mínimo de 2-4 y 2-5: 7

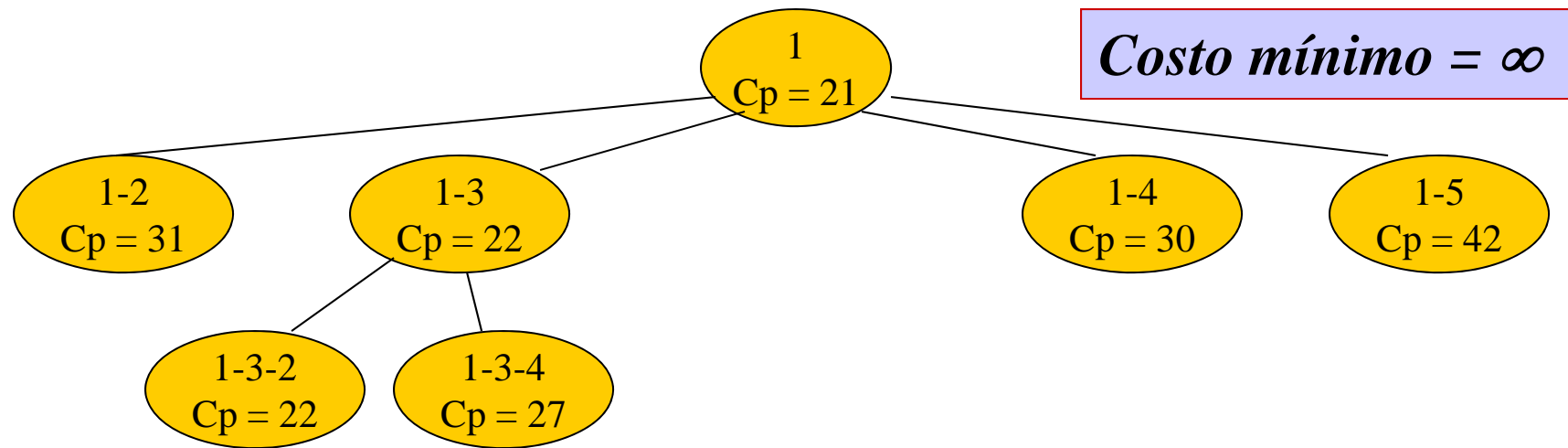
Más mínimo de 4-1 y 4-5: 2

Más mínimo de 5-1 y 5-4: 4

TOTAL = 22

Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



Cálculo del Costo posible:

Acumulado de 1-3-4 : 11

Más mínimo de 4-2 y 4-5: 2

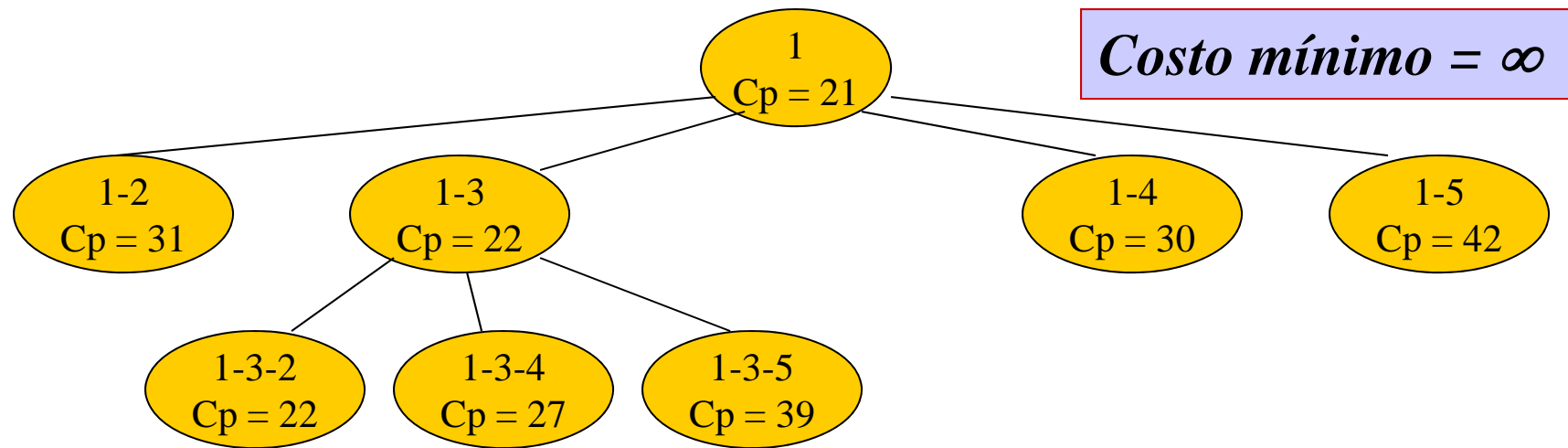
Más mínimo de 2-1 y 2-5: 7

Más mínimo de 5-1 y 5-2: 7

TOTAL = 27

Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



¿Cuál es el mejor nodo
para expandir?

Cálculo del Costo posible:

Acumulado de 1-3-5 : 20

Más mínimo de 5-2 y 5-4: 4

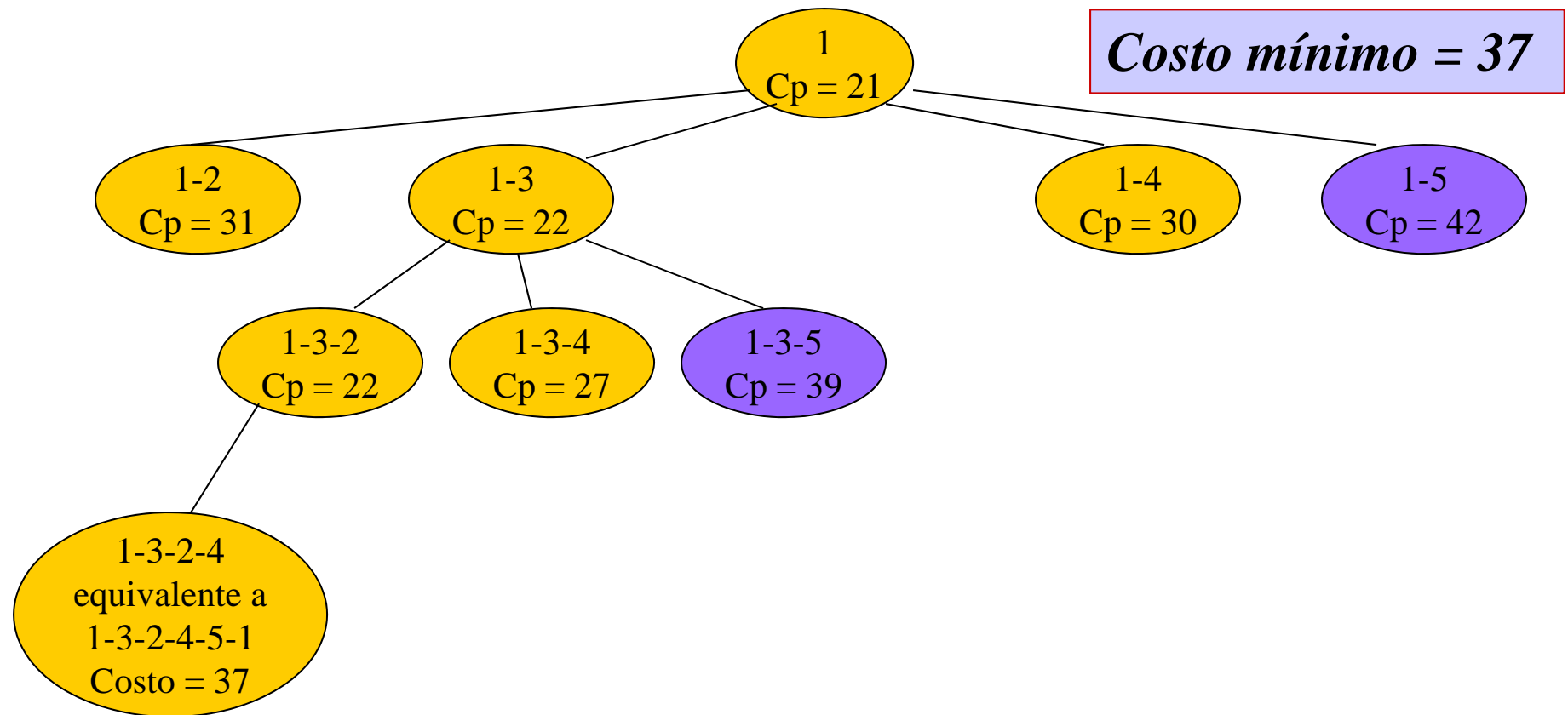
Más mínimo de 2-1 y 2-4: 8

Más mínimo de 4-1 y 4-2: 7

TOTAL = 39

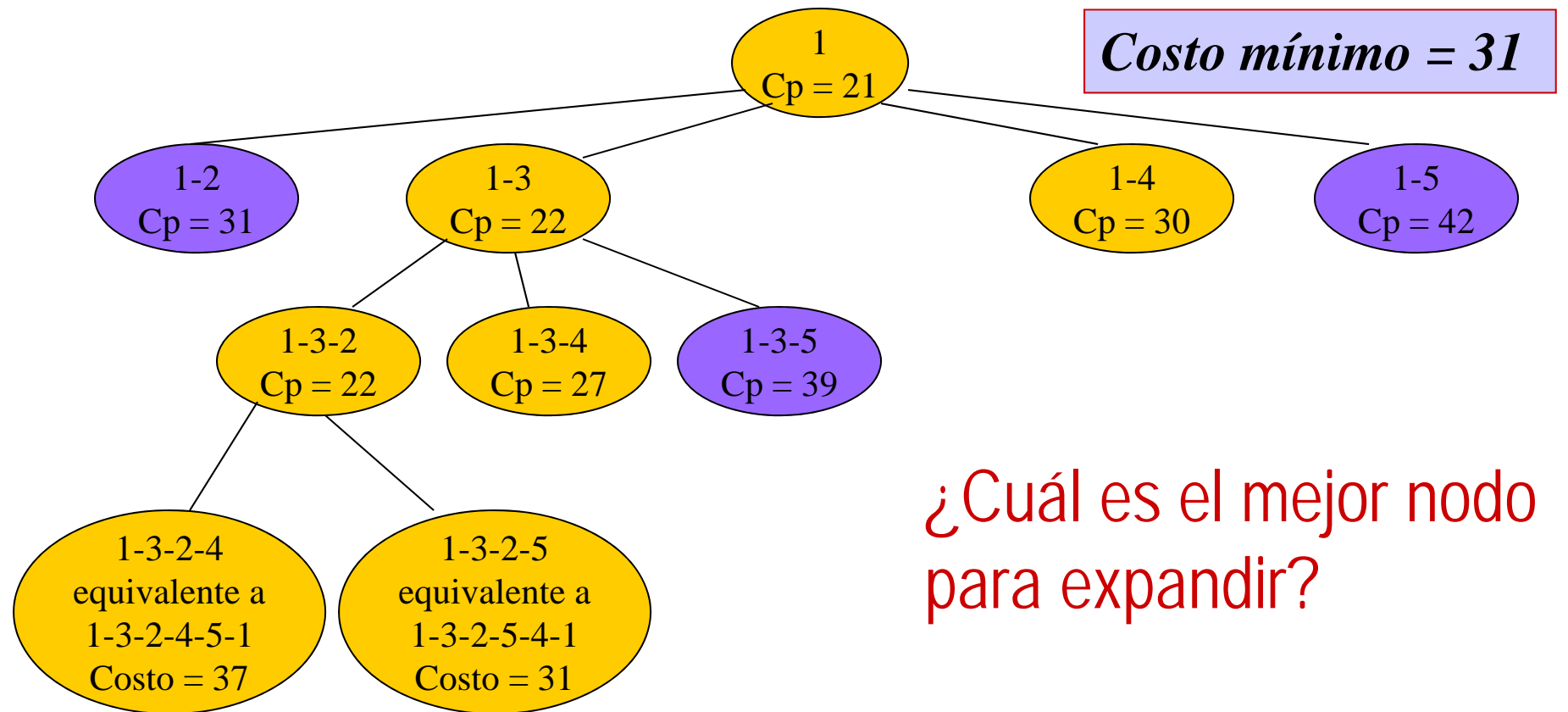
Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



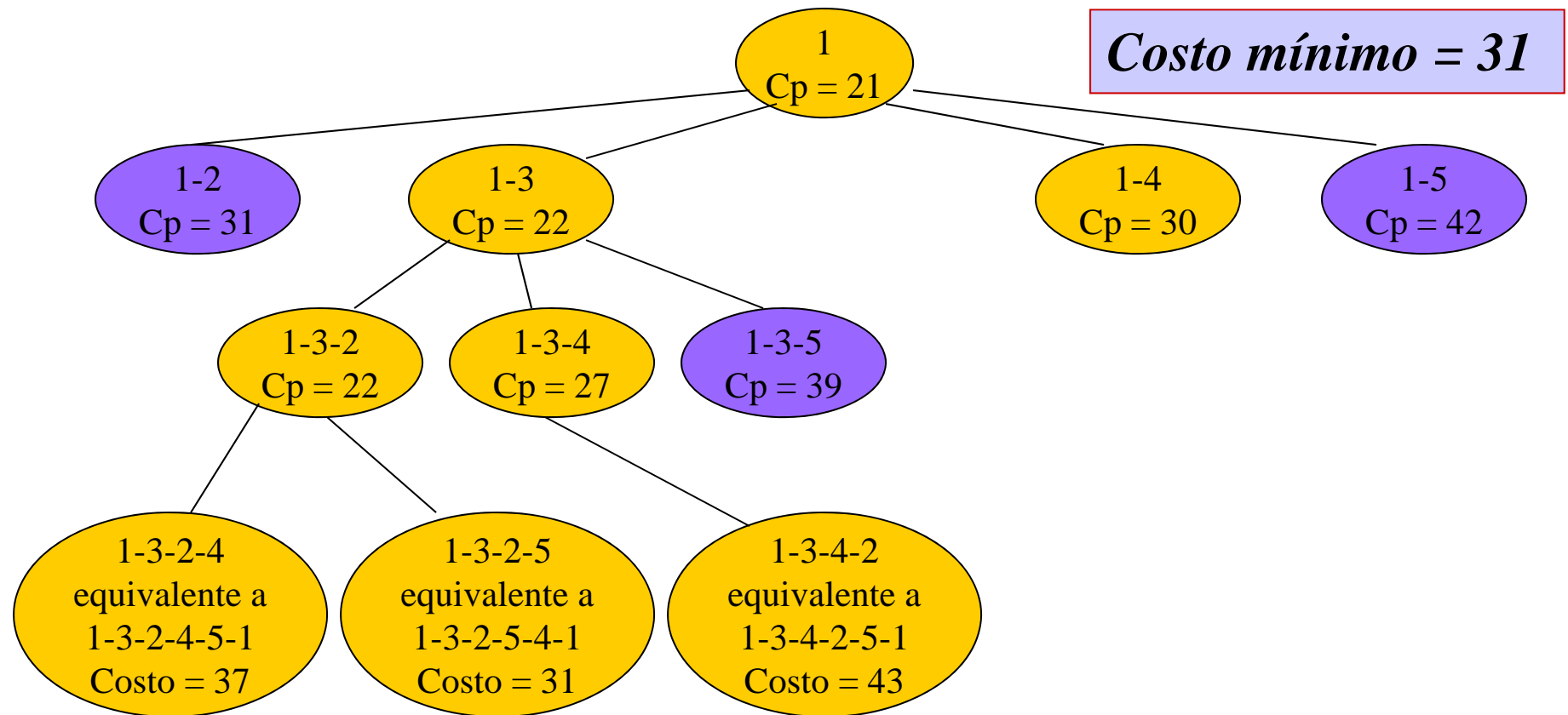
Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



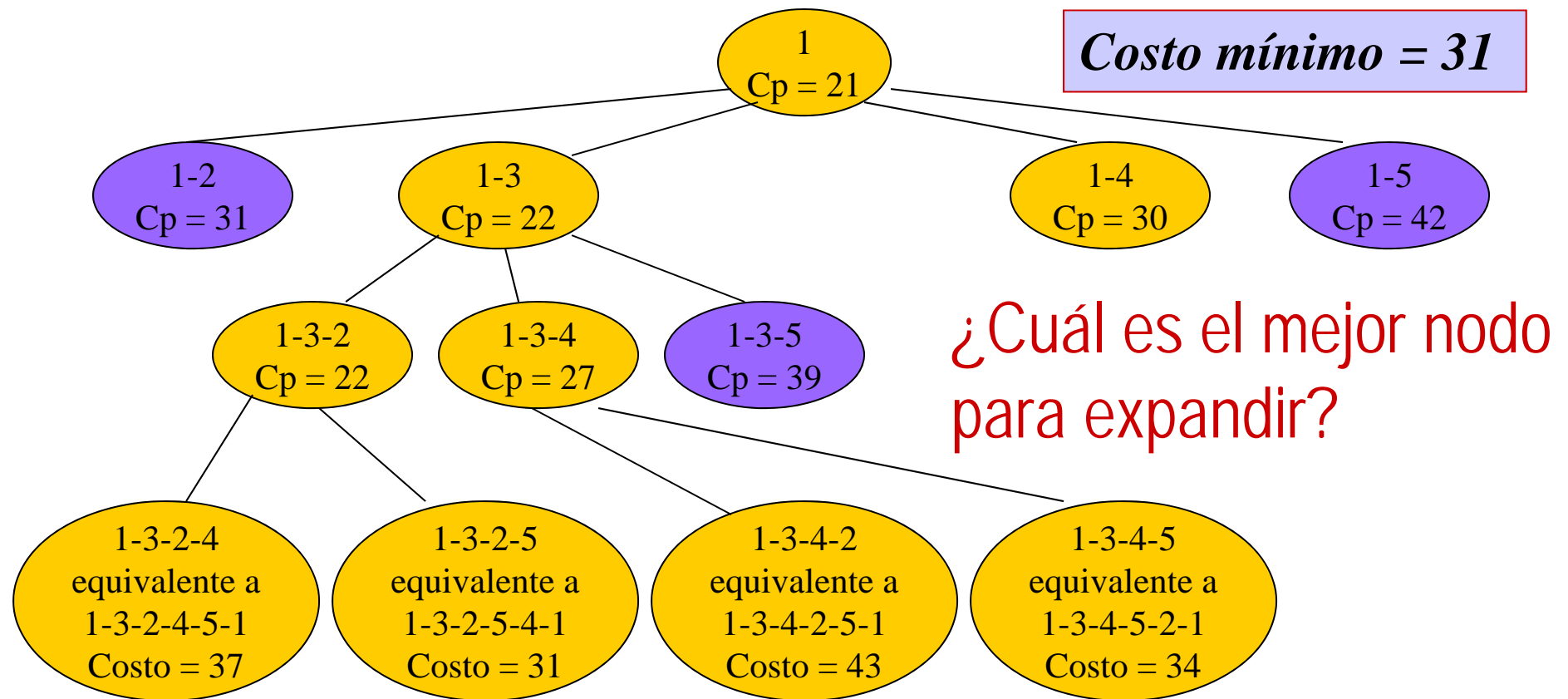
Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



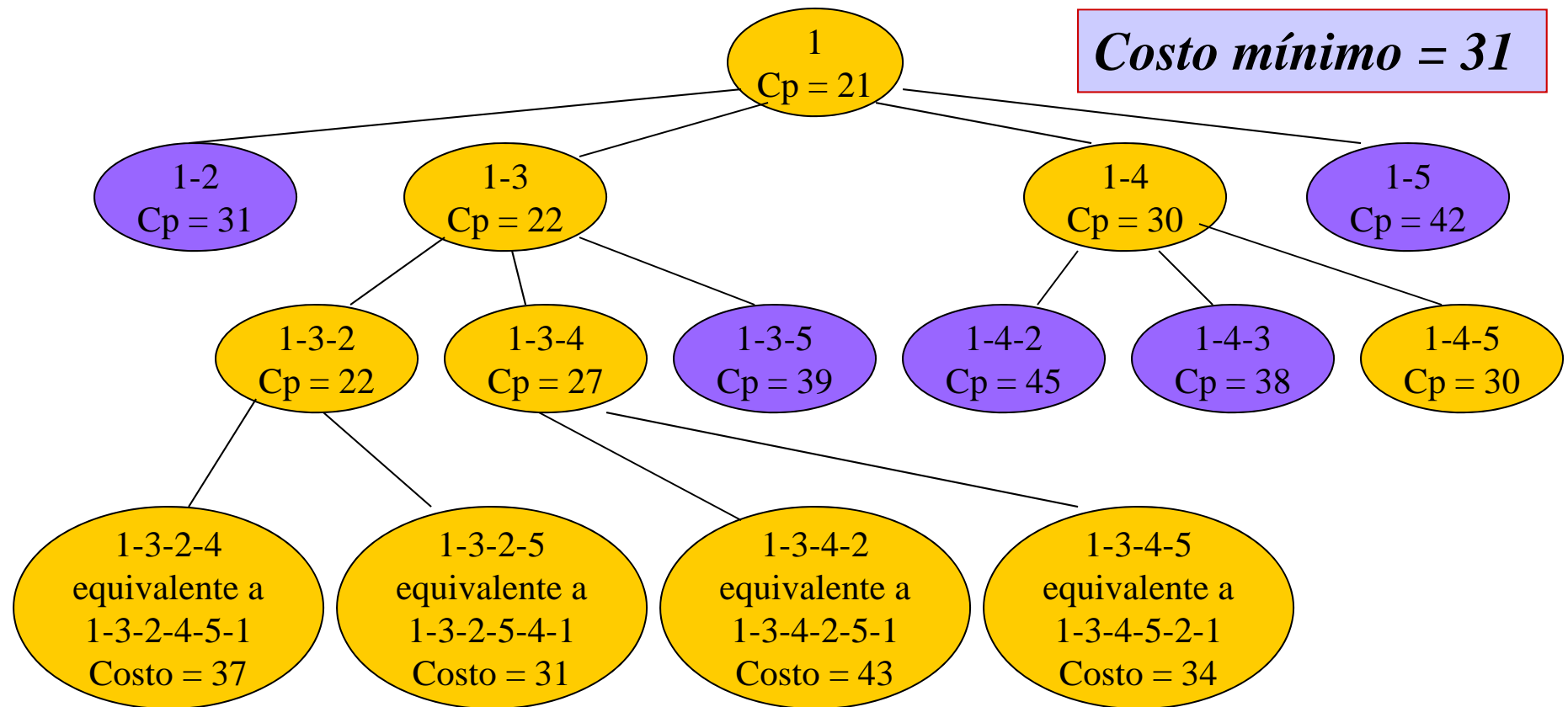
Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0

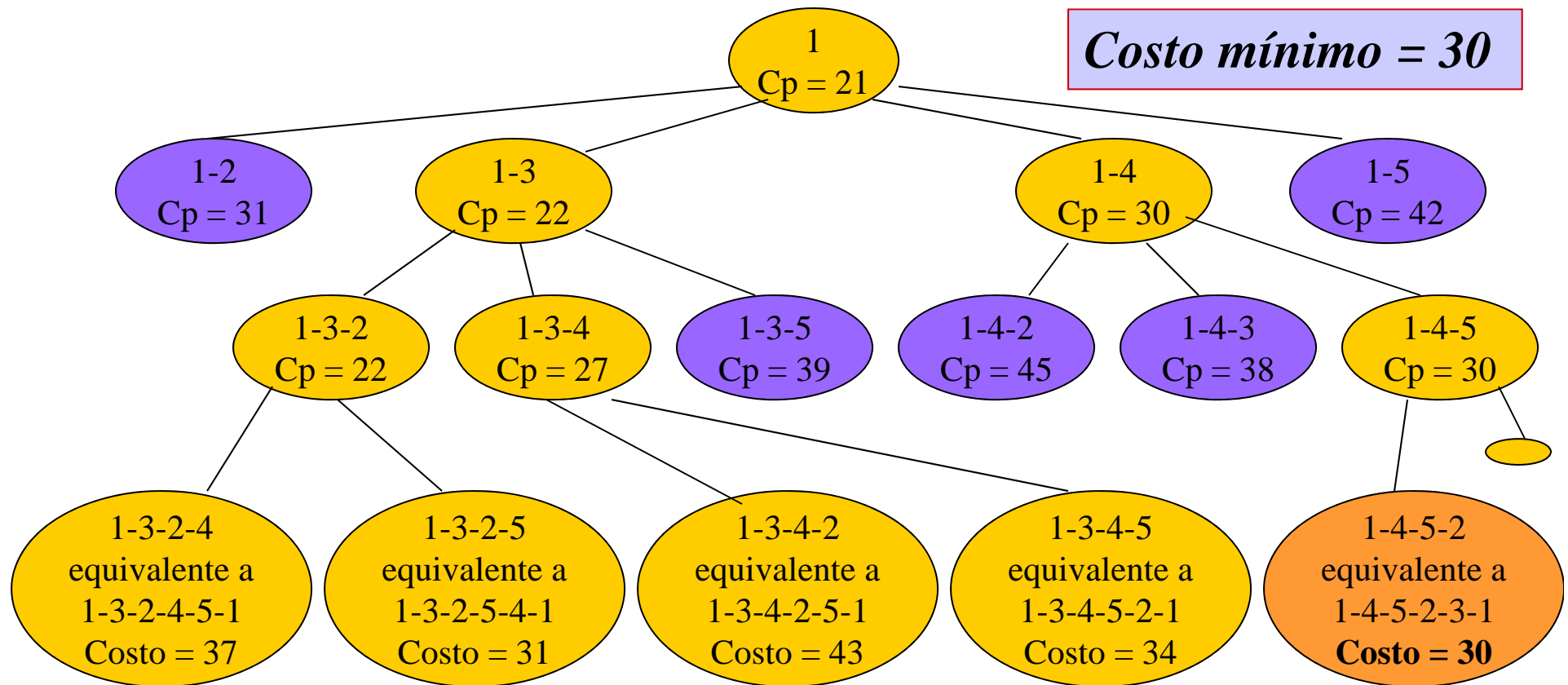


¿Cuál es el mejor nodo para expandir?

Análisis de Algoritmos


Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



Conclusión final

El problema del viajero



- Branch and bound ofrece una opción más a la solución del problema del viajero...
- Sin embargo, NO asegura tener un buen comportamiento en cuanto a eficiencia, pues en el peor caso, tiene un comportamiento exponencial...
- El problema puede ser resuelto con la técnica de los "algoritmos aproximados"...

Análisis y Diseño de Algoritmos



Clasificación de Algoritmos

Teoría de la complejidad



La teoría de la clasificación de problemas esta basada en que tan difícil se pueden resolver.

- Problemas de Clase P
- Problemas de Clase NP
- Problemas de Clase NP – Complete.

Problemas P

- Un problema es asignado a la clase P (Tiempo Polinomial) cuando el tiempo de solución está en una potencia constante del tamaño de la entrada (n^k).
- Ejemplos:
 - Strassen $\rightarrow n^{2.81}$
 - Floyd $\rightarrow n^3$
 - ABB Óptimo $\rightarrow n^3$

Problemas NP

Problema Polinomial No-Determinístico.

- Un algoritmo No-Determinístico es un algoritmo que consta de dos fases: suponer y comprobar.
- Si la complejidad temporal en la etapa de comprobación de un algoritmo no-determinístico es polinomial, entonces este algoritmo se denomina algoritmo polinomial no-determinístico.

Problemas P vs. Problemas NP

- La clase P es un subconjunto de la clase NP ya que podríamos construir un algoritmo que resolviera los problemas de la clase P con las mismas dos etapas que se usan en los problemas de la clase NP.
- La diferencia es que tenemos soluciones en tiempo polinomial para *todos* los problemas de la clase P, pero no los tenemos para *todos* los de la clase NP.
- Preguntarnos si “ $P = NP$ ” hoy por hoy es un problema abierto.

Problemas NP – Completos

- Los problemas NP-completos son los problemas mas difíciles de la clase de problemas NP.
 - La creencia de que $P \neq NP$ viene dada por la existencia de problemas NP-Completos.
- Un problema B es llamado NP – Completo, si los dos siguientes puntos son verdaderos:
 - B es NP
 - Para otro problema A en NP, A se reduce en B.

Problemas NP – Completos



- Ejemplos:
 - Camino Máximo: Dados dos vértices de un grafo encontrar el camino (simple) máximo.
 - TSP: Ciclo simple de menor costo que contiene cada vértice del grafo.

Grafo Isomorfico

TSP

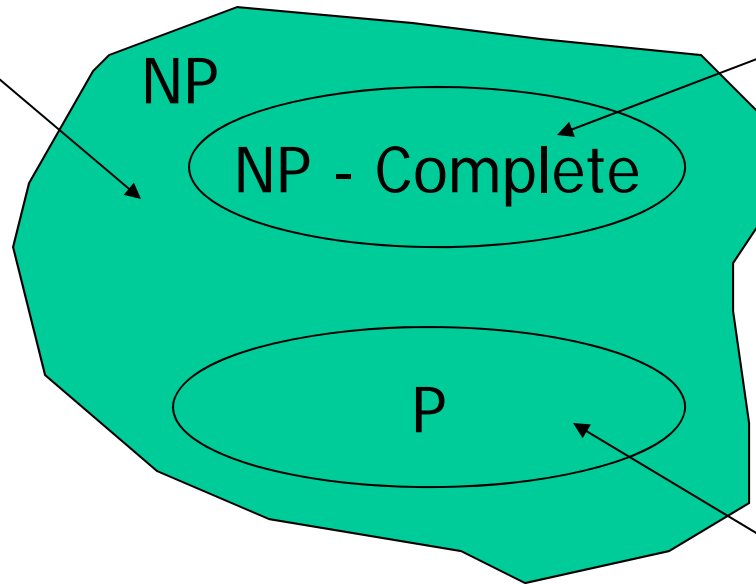


ABB Óptimo