

# Algorítmica

---

**Tema 1. Planteamiento General**

**Tema 2. La Eficiencia de los Algoritmos**

**Tema 3. Algoritmos “Divide y vencerás”**

**Tema 4. Algoritmos Voraces (“Greedy”)**

**Tema 5. Algoritmos basados en Programación Dinámica**

**Tema 6. Algoritmos para la Exploración de Grafos**

**(“Backtracking”, “Branch and Bound”)**

**Tema 7. Otras metodologías algorítmicas**

# Algorítmica

## TEMA 1: PLANTEAMIENTO GENERAL

---

1. CONCEPTO DE ALGORITMO
2. RESOLUCIÓN DE PROBLEMAS
3. CLASIFICACIÓN DE PROBLEMAS
4. ALGORÍTMICA
5. INTRODUCCION A LA EFICIENCIA DE ALGORITMOS
6. DISEÑO DE ALGORITMOS

Bibliografía:

- G. BRASSARD, P. BRATLEY. Fundamentos de Algoritmia. Prentice Hall (1997)

# CONCEPTO DE ALGORITMO

---

## Definición de Algoritmo

Secuencia ordenada de pasos exentos de ambigüedad y determinísticos tal que al llevarse a cabo con fidelidad dará como resultado que se realice la tarea para la que se ha diseñado en un tiempo finito (se obtiene la solución del problema planteado)

# Propiedades de un ALGORITMO

---

- a) **Finitud:** terminar después de un número finito de etapas
- b) **Precisión:** cada etapa debe estar definida de forma precisa; las acciones que hay que llevar a cabo deben estar rigurosamente especificadas para cada caso
- c) **Entrada:**
- d) **Salida:**
- e) **Efectividad:** todas las operaciones que hay que realizar deben ser tan básicas como para que se puedan hacer exactamente y en un periodo finito de tiempo

# **RESOLUCIÓN DE PROBLEMAS**

---

## **Resolución de un problema con un ordenador**

- **Diseñar un algoritmo para el problema**
- **Expresar el algoritmo como un programa**
- **Ejecutar el programa correctamente**

# ESTUDIO/CLASIFICACIÓN DE PROBLEMAS

---

**Década 30:** Problemas Computables y No Computables

**Década 50:** Complejidad de los problemas computables (búsqueda de algoritmos más eficaces)

**Década 70:** Clasificación de los problemas computables: P y NP

# ESTUDIO/CLASIFICACIÓN DE PROBLEMAS

---

## Problemas P y NP

- ¿Qué es la clase P?
- ¿Qué es la clase NP?
- ¿Qué hace que un problema sea NP?

# Clase P de problemas

---

- Se consideran problemas con tiempo polinomial porque sus tiempos de ejecución son funciones polinomiales
- Se dice que todos esos problemas están en la Clase P
- Se asume que un tiempo polinómico es un tiempo *eficiente*

# Clase NP de problemas

- Hay problemas para los que la única forma de resolverlos en tiempo polinomial es realizando una etapa aleatoria (incluyendo el azar de alguna manera).
- Típicamente, la solución incluye una primera etapa que de forma no determinista elige una posible solución, y entonces en etapas posteriores comprueba si esa solución es correcta.

*Para esto, haría falta una máquina de turing no determinista. Podemos suponer que una máquina no determinista es una máquina "adivina" o aquella que podría clonarse y realizar tantas operaciones en paralelo como se quiera. Es lo que se busca con la computación cuántica.*

*La empresa canadiense D-Wave System había supuestamente presentado el 13 de febrero de 2007 en Silicon Valley, una primera computadora cuántica comercial de 16-qubits de propósito general; luego la misma compañía admitió que tal máquina llamada Orion no es realmente una Computadora Cuántica, sino una clase de máquina de propósito general que usa algo de mecánica cuántica para resolver problemas. Aún seguimos esperando...*

## Relación entre las clases P y NP

---

### Clase P $\subseteq$ Clase 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.

# Reducción de problemas y complejidad

---

- Si reducimos un problema (A) a otro (B) y obtenemos una solución con un algoritmo para el problema B, podemos transformar esa solución para convertirla en una solución para el problema A
- Si podemos realizar las transformaciones de un problema en tiempo polinomial ( $A \rightarrow B$  y  $B \rightarrow A$ ), y también podemos resolverlo en tiempo polinomial, sabemos que el problema original es resoluble en tiempo polinomial

$$(A \rightarrow B + \text{Res}(B) + B \rightarrow A) \Rightarrow O(n^i) + O(n^j) + O(n^k) \Rightarrow O(n^f)$$

tq.  $f = \max(i, j, k)$

# Problemas NP-Completos

---

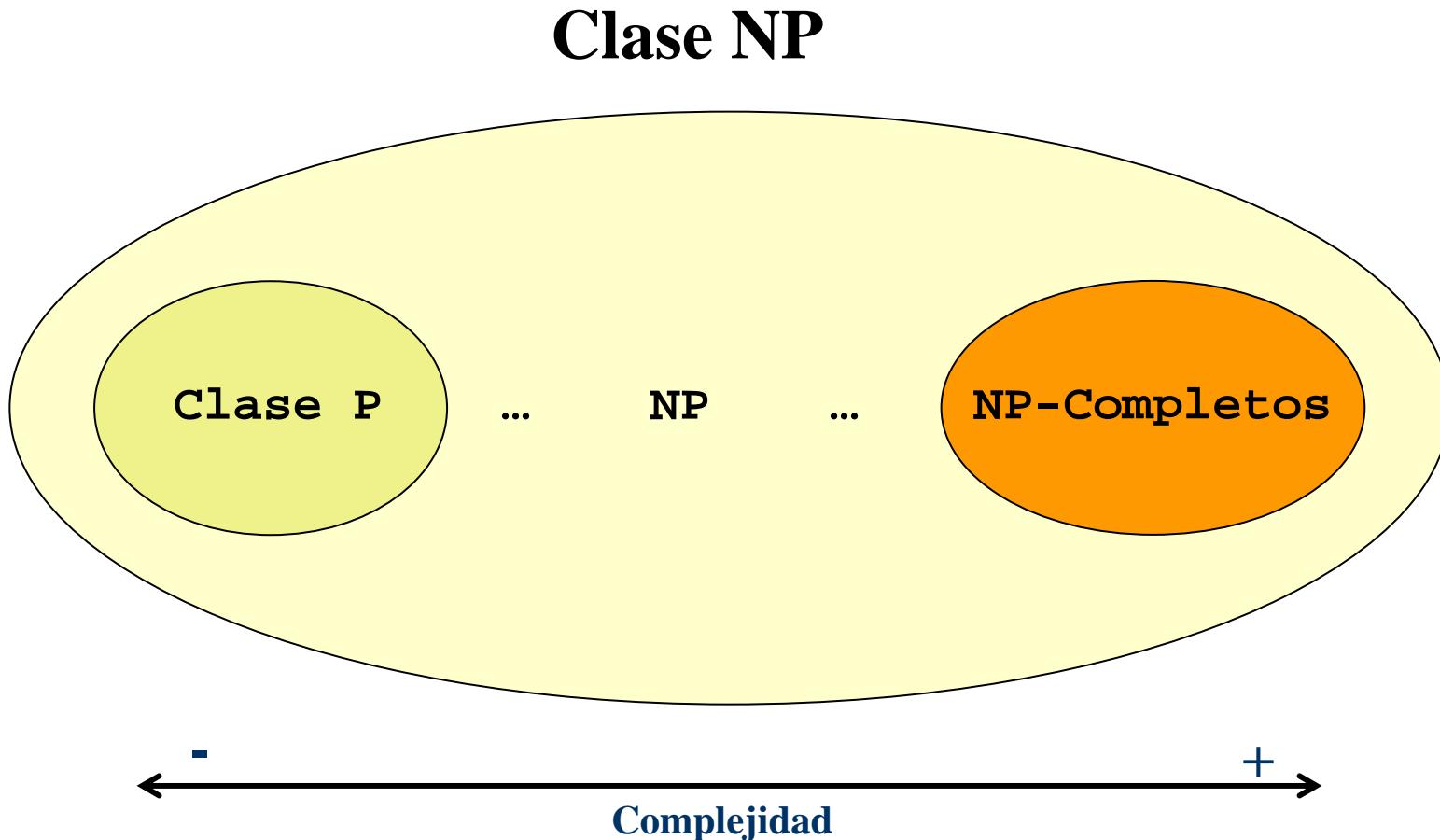
- Se dice que un problema es NP-completo si todos los problemas de la clase NP pueden reducirse a él
- Los problemas NP-completos son los problemas mas difíciles de la clase de problemas NP
- Debido a la reducción, si encontráramos un algoritmo en tiempo polinomial para un problema NP-completo, sabríamos que todos los problemas de la clase NP requieren un tiempo polinómico

# ¿Es P = NP?

---

- La clase P es un subconjunto de la clase NP
- Para que ambas clases sean iguales, todos los problemas en la clase NP debería tener algoritmos en tiempo polinomial
- Hasta ahora nadie ha sido capaz de encontrar un algoritmo en tiempo polinomial ni de demostrar lo contrario
- Por tanto, la cuestión de si “P = NP” es actualmente un problema abierto

# ¿Es P = NP?



# **ALGORÍTMICA**

---

El estudio de los algoritmos incluye el de diferentes e importantes áreas de investigación y docencia, y se suele llamar Algorítmica

La Algorítmica considera:

- 1. La construcción**
- 2. La expresión**
- 3. La validación**
- 4. El análisis, y**
- 5. Estudio empírico**

# 1. La construcción de algoritmos

---

- Creación de algoritmos: parte creativa y sistemática
- El acto de crear un algoritmo no se puede dominar si no se conocen a la perfección las técnicas de diseño de los algoritmos
- El área de la construcción de algoritmos engloba el estudio de los métodos que, facilitando esa tarea, se han demostrado en la práctica más útiles

## 2. La expresión de algoritmos

---

- Los algoritmos deben expresarse de forma precisa, clara, e independientes de un lenguaje de programación en concreto

### 3. Validación de algoritmos

---

- Validación: Demostrar que las respuestas que da son correctas para todas las posibles entradas
- Único método válido: demostración formal
- En muchos casos no es factible: pruebas empíricas o parciales

## 4. Análisis de algoritmos

---

- Análisis de algoritmos: Determinar los recursos (tiempo, espacio) que consume un algoritmo en la resolución de un problema
- Permite comparar algoritmos con criterios cuantitativos
- Elección de un algoritmo entre varios para resolver un problema

## 5. Estudio empírico

---

- Se trata de generar e implementar el algoritmo mediante un programa: codificación, prueba y validación (con depuración en su caso)
- Evaluación empírica del tiempo y espacio que requiere el algoritmo implementado para resolver problemas de distinto tamaño.

# INTRODUCCIÓN A LA EFICIENCIA DE LOS ALGORITMOS

---

## Elección de un Algoritmo

En la práctica no solo queremos algoritmos, sino que queremos buenos algoritmos en algún sentido propio de cada usuario.

# INTRODUCCIÓN A LA EFICIENCIA DE LOS ALGORITMOS

---

## Elección de un Algoritmo

A menudo tendremos varios algoritmos para un mismo problema, y deberemos decidir cual es el mejor, o cual es el que tenemos que escoger según algún criterio, para resolverlo.

Esto nos centra el estudio en el campo del **Análisis de los Algoritmos**.

### Problema Central:

**Determinar ciertas características que sirvan para evaluar su rendimiento.**

# INTRODUCCIÓN A LA EFICIENCIA DE LOS ALGORITMOS

---

## Elección de un Algoritmo

Un criterio de bondad es la longitud del tiempo consumido para ejecutar el algoritmo; esto puede expresarse en términos del número de veces que se ejecuta cada etapa.

# INTRODUCCIÓN A LA EFICIENCIA DE LOS ALGORITMOS

---

**Modelos para el análisis de la eficiencia:**

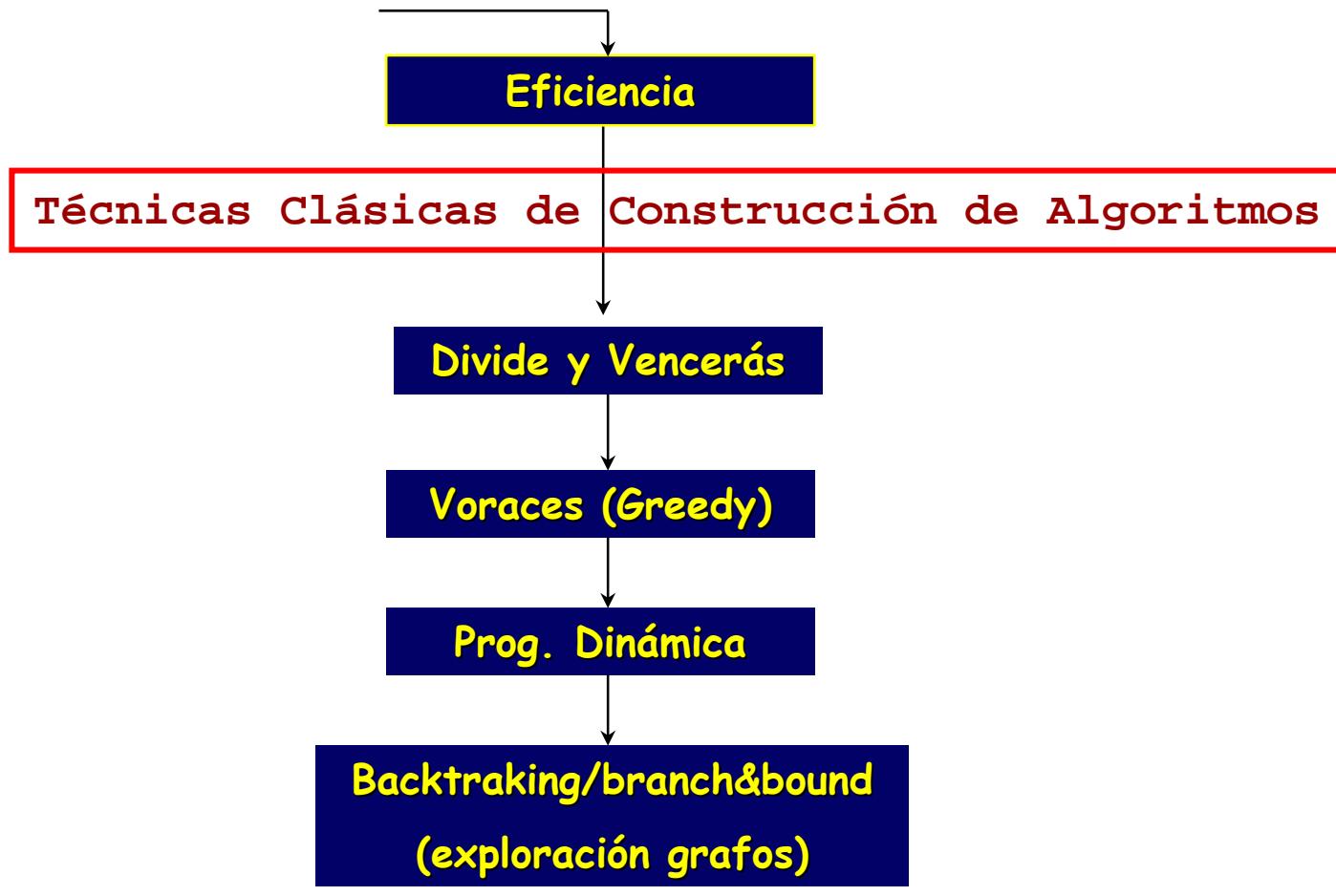
- Enfoque Empírico.
- Enfoque teórico.
- Enfoque Híbrido.

# **SOBRE EL DISEÑO DE ALGORITMOS**

---

**Se necesitan técnicas para la Construcción  
de Algoritmos Eficaces y Eficientes.**

# SOBRE EL DISEÑO DE ALGORITMOS



# Resumen

---

## ■ **Algoritmo:**

Conjunto de reglas para resolver un problema.

## ■ **Propiedades**

- **Definibilidad:** El conjunto debe estar bien definido, sin dejar dudas en su interpretación.
- **Finitud:** Debe tener un número finito de pasos que se ejecuten en un tiempo finito.



# Resumen

---

- **Algoritmos deterministas:** Para los mismos datos de entrada se producen los mismos datos de salida.
- **Algoritmos no deterministas:** Para los mismos datos de entrada pueden producirse diferentes de salida.
- **ALGORÍTMICA:** Ciencia que estudia técnicas para construir algoritmos eficientes y técnicas para medir la eficacia de los algoritmos.
- **Objetivo:** Dado un problema concreto encontrar la mejor forma de resolverlo.

# Objetivo de la asignatura

---

Ser capaz de **analizar**, **comprender** y **resolver** una amplia variedad de **problemas** de programación, diseñando soluciones **eficientes** y de **calidad**.

Pero **ojo**, los algoritmos no son el único componente en la resolución de un problema de programación.

# Resumen



Algoritmos  
+  
Estructuras  
de datos

MES	CITAS	CTS./DIA	COMPLETOS	DIAS VACIOS
ENERO	5	0.16	1	28
FEBRERO	5	0.17	1	24
MARZO	2	0.06	14	29
ABRIL	8	0.00	4	30
MAYO	12	0.33	3	31
JUNIO	10	0.33	6	30
JULIO	1	0.03	0	30
AGOSTO	1	0.03	0	30
SEPTIEMBRE	1	0.03	12	29
OCTUBRE	4	0.13	12	29
NOVIEMBRE	36	1.20	5	25
DICIEMBRE	48	1.55	6	24

Algoritmos + Estructuras de Datos = Programas

- **Estructura de datos:** Parte estática, almacenada.
- **Algoritmo:** Parte dinámica, manipulador.

# Resumen

---

## **ALGORITMICA = ANÁLISIS + DISEÑO**

- **Análisis de algoritmos:** Estudio de los recursos que necesita la ejecución de un algoritmo.
- No confundir con análisis de un problema.
  
- **Diseño de algoritmos:** Técnicas generales para la construcción de algoritmos.
- Por ejemplo, divide y vencerás: dado un problema, divídalo, resuelve los subproblemas y luego junta las soluciones.

# Resumen

---

- **Análisis de algoritmos.** Normalmente estamos interesados en el estudio del tiempo de ejecución.
- Dado un algoritmo, usaremos las siguientes notaciones:
  - $t(\dots)$ : Tiempo de ejecución del algoritmo.
  - $O(\dots)$ : Orden de complejidad.
  - $o(\dots)$ : O pequeña del tiempo de ejecución.
  - $\Omega(\dots)$ : Cota inferior de complejidad.
  - $\Theta(\dots)$ : Orden exacto de complejidad.

# Resumen: Análisis de algoritmos.

---

- **Ejemplo.** Analizar el tiempo de ejecución y el orden de complejidad del siguiente algoritmo.

**Hanoi (N, A, B, C: integer)**

```
if N=1 then
    Mover (A, C)
else begin
    Hanoi (N-1, A, C, B)
    Mover (A, C)
    Hanoi (N-1, B, A, C)
end
```

- Mecanismos:
  - Conteo de instrucciones.
  - Uso de ecuaciones de recurrencia.
  - Medida del trabajo total realizado.

# Resumen: Diseño de algoritmos.

- **Diseño de Algoritmos.** Técnicas generales, aplicables a muchas situaciones.
- **Esquemas algorítmicos.** Ejemplo:

```
ALGORITMO Voraz (C: ConjuntoCandidatos; var S: ConjuntoSolución )  
    S:= Ø  
    mientras (C ≠ Ø) Y NO SOLUCION(S) hacer  
        x:= SELECCIONAR(C)  
        C:= C - {x}  
        si FACTIBLE(S, x) entonces  
            INSERTAR(S, x)  
    finsi  
    finmientras
```

The diagram illustrates the structure of the Voraz algorithm. It features several text boxes containing pseudocode and two callout boxes on the right side. The callout boxes are labeled "Insertar tipos AQUÍ" (in green) and "Insertar código AQUÍ" (in yellow). Arrows point from these callout boxes to specific parts of the pseudocode: one arrow from each callout box points to the "SOLUCION(S)" box, and another arrow from the yellow callout box points to the "INSERTAR(S, x)" box.

Y para terminar..algunos ejemplos...

---

¿Qué clase de problemas  
pueden estudiarse?

# EI Sudoku

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

**Se puede diseñar un algoritmo que lo resuelva  
de forma sencilla**

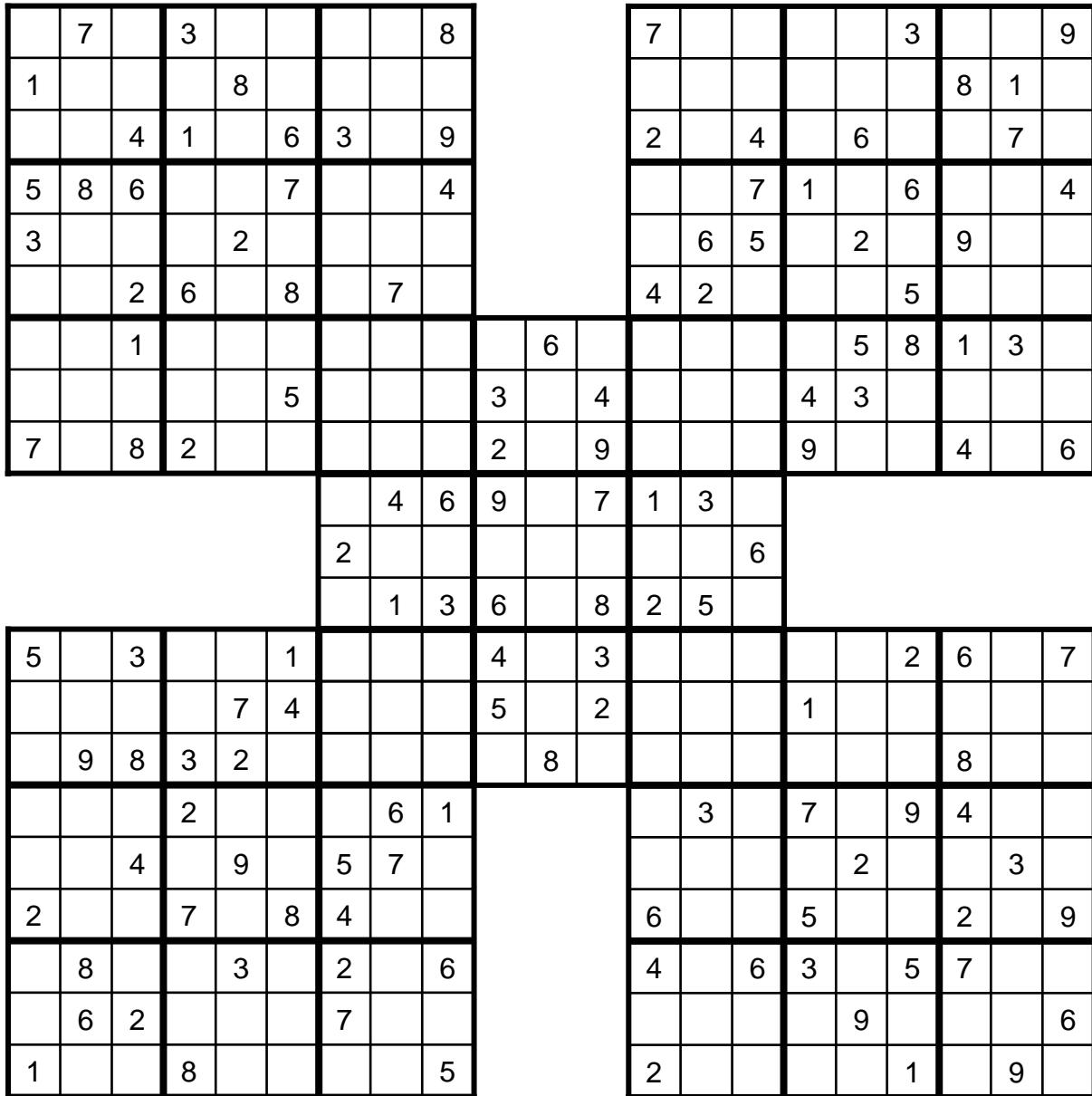
# EI Sudoku

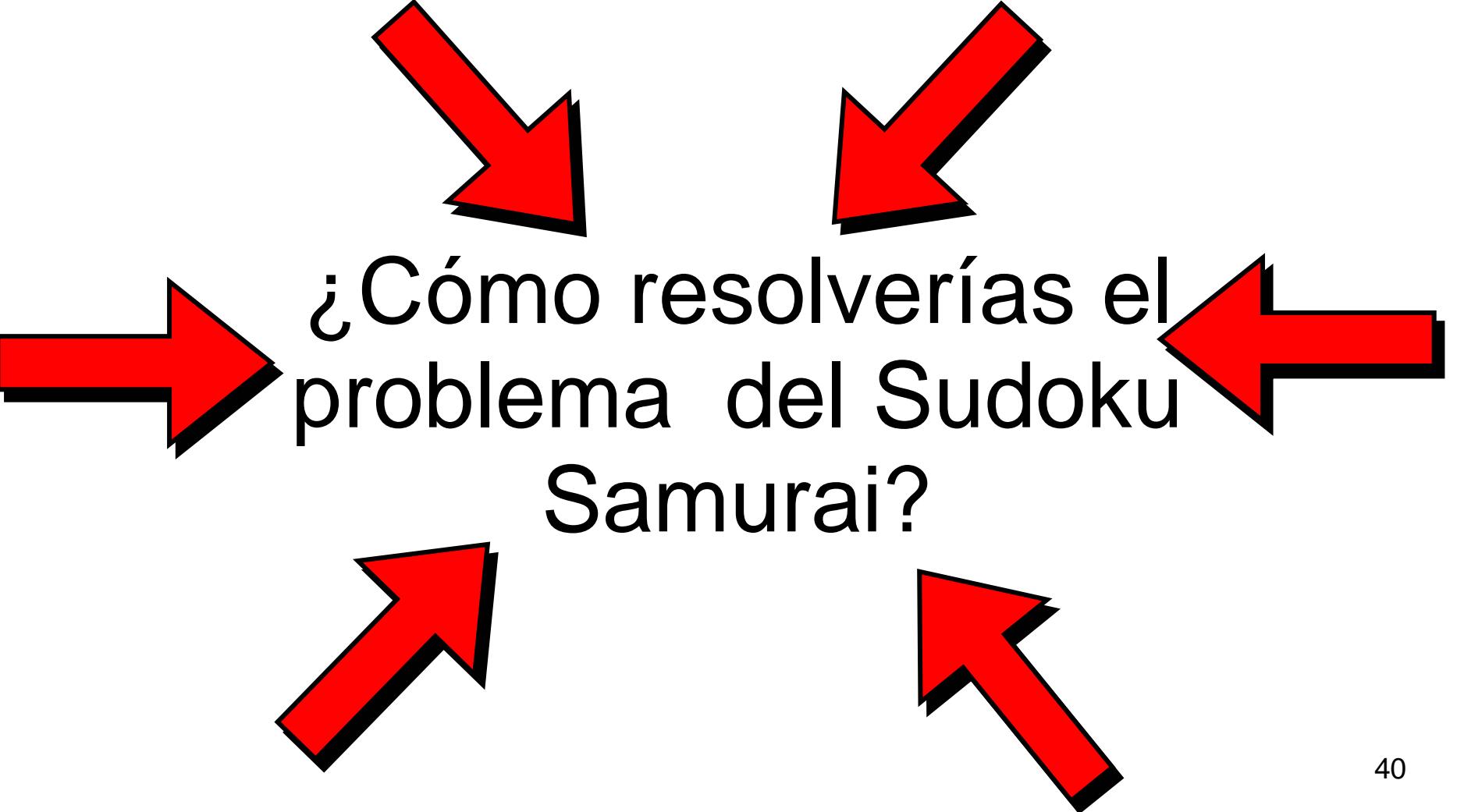
5	8	1	9	2	4	3	7	6
2	3	4	7	6	8	9	1	5
7	6	9	1	3	5	2	4	8
9	2	7	8	4	6	5	3	1
4	1	8	5	9	3	6	2	7
6	5	3	2	1	7	8	9	4
3	4	5	6	7	9	1	8	2
1	7	6	3	8	2	4	5	9
8	9	2	4	5	1	7	6	3

# EI

# Sudoku

# Samurai





¿Cómo resolverías el  
problema del Sudoku  
Samurai?

# Ejemplos de problemas



# Ejemplos de problemas

The screenshot shows a Microsoft Internet Explorer window with the title bar "Google - Microsoft Internet Explorer". The menu bar includes "Archivo", "Edición", "Ver", "Favoritos", "Herramientas", and "Ayuda". The toolbar contains icons for Back, Forward, Stop, Refresh, Home, Search, Favorites, Multimedia, and Print. The address bar shows the URL "http://www.google.es/".

The main content is the Google España homepage. A search bar at the top contains the query "problemas computacionales". Below it are two buttons: "Búsqueda en Google" and "Voy a Tener Suerte". To the right of the search bar are links for "Búsqueda Avanzada", "Preferencias", and "Herramientas del idioma".

Below the search bar, there are three radio button options: "Búsqueda en la Web" (selected), "páginas en español", and "páginas de España". At the bottom of the page is a link "Crea tu propia página de Google".

The footer contains the copyright notice "©2004 Google - Buscando 4.285.199.774 páginas web".

# Ejemplos de problemas

Búsqueda en Google: problemas computacionales - Microsoft Internet Explorer

Archivo Edición Ver Favoritos Herramientas Ayuda

Atrás Último Buscar Navegar Búsqueda Favoritos Multimedia Correo Imprimir Navegar Archivo

Dirección: http://www.google.es/search?q=problemas+computacionales&hl=es&lr=&ie=UTF-8&start=70&sa=N Ir

**Google** La Web Imágenes Grupos Directorio News

problemas computacionales Búsqueda Búsqueda Avanzada Preferencias

Búsqueda:  la Web  páginas en español  páginas de España

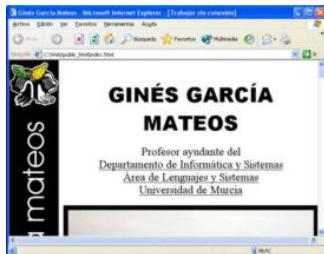
**La Web** Resultados 71 - 80 de aproximadamente 31.400 de **problemas computacionales**. (0,23 segundos)

[\[PDF\] UNIVERSIDAD DEL PEDREGAL Ingeniería en Sistemas Computacionales y ...](#)  
Formato de archivo: PDF/Adobe Acrobat - [Versión en HTML](#)  
... enfocadas a reconocer y suavizar los **problemas** de comunicación en todos sus niveles.  
Page 3. Escuela de Ingeniería en Sistemas **Computacionales** y Telemática ...  
[www.upedregal.edu.mx/Licenciaturas/ ProgIng/DesarrolloProy.pdf](#) - [Páginas similares](#)

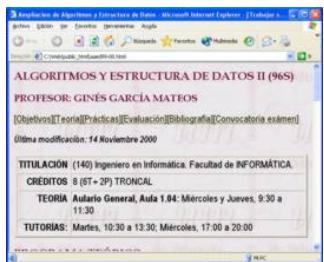
[DISEÑO DE UN GENERADOR DE SISTEMAS COMPUTACIONALES PARA LA ...](#)  
... objetivo del proyecto es construir un generador de sistemas **computacionales** para  
la ... lenguajes de alto nivel para la representación de **problemas**, a algoritmos ...  
[www.fondef.cl/bases/fondef/PROYECTO/92/I/D92I1028.HTML](#) - 26k - [En caché](#) - [Páginas similares](#)

**Ofrecemos hacernos cargo de:**  
... Nosotros nos hacemos cargo de sus **problemas computacionales** para que usted se dedique,  
integralmente, a cuidar del gerenciamiento de su negocio, para esto hay ...  
[usuarios.vtr.net/~xasports/consultoria/body\\_home.htm](#) - 5k - [En caché](#) - [Páginas similares](#)

# Buscador de Internet



algoritmos, ayudante, curso,  
datos, estructuras, garcía,  
mateos, ...



algoritmos, cosa, curso,  
datos, estructuras,  
evaluación, prácticas, ...



agua, botavara, barco,  
confeccionar, las, velas, ...

# Buscador de Internet



# Buscador de Internet

Búsqueda en Google: problemas computacionales - Microsoft Internet Explorer

Archivo Edición Ver Favoritos Herramientas Ayuda

Atrás Último Buscar Página Imprimir Volver Ir Dirección http://www.google.es/search?q=problemas+computacionales&hl=es&lr=&ie=UTF-8&start=70&sa=N

Google La Web Imágenes Grupos Directorio News

problemas computacionales Búsqueda Búsqueda Avanzada Preferencias

Búsqueda:  la Web  páginas en español  páginas de España

**La Web** Resultados 71 - 80 de aproximadamente 31.400 de **problemas computacionales**. (0,23 segundos)

[PDF] [UNIVERSIDAD DEL PEDREGAL Ingeniería en Sistemas Computacionales y ...](#)  
Formato de archivo: PDF/Acrobat Acrobat - Versión en HTML  
... enfocadas a n  
Page 3. Escuela  
[www.upedregal.e](http://www.upedregal.e)

**(0,23 segundos)**

[DISEÑO DE U](#) A ...  
... objetivo del proyecto es construir un generador de sistemas **computacionales** para  
la ... lenguajes de alto nivel para la representación de **problemas**, a algoritmos ...  
[www.fondef.cl/bases/fondef/PROYECTO/92/I/D92I1028.HTML](http://www.fondef.cl/bases/fondef/PROYECTO/92/I/D92I1028.HTML) - 26k - En caché - Páginas similares

**Ofrecemos hacernos cargo de:**  
... Nosotros nos hacemos cargo de sus **problemas computacionales** para que usted se dedique,  
integralmente, a cuidar del gerenciamiento de su negocio, para esto hay ...  
[usuarios.vtr.net/~xasports/consultoria/body\\_home.htm](http://usuarios.vtr.net/~xasports/consultoria/body_home.htm) - 5k - En caché - Páginas similares

Internet

# Buscador de Internet

- ¡¡¡Cuatro mil millones de páginas en un cuarto de segundo!!!
- **Problema:** ¿cómo estructurar la información necesaria para realizar las consultas rápidamente? ¿Qué algoritmos de búsqueda utilizar?

# Buscador de Internet

- Supongamos una red de 1024 ordenadores a 3 GHz.
- Supongamos que cada página tiene 200 palabras, de 8 letras cada una y en cada letra se tarda 2 ciclos de reloj.
- ¡¡El recorrido de todas las páginas tardaría 4,5 segundos!!

# Buscador de Internet

- Solución: Darle la vuelta al problema...

agua →



ayudante →



cosa →



las →



...

Búsqueda en Google: problemas computacionales - Microsoft Internet Explorer

Archivo Edición Ver Favoritos Herramientas Ayuda

Atrás Último Buscador FAVORITOS Multimedias Imprimir Página Recortar Copiar Pegar Búsqueda Favoritos

Dirección http://www.google.es/search?q=problemas+computacionales&hl=es&lr=&ie=UTF-8&start=70&sa=N Ir

Google™ La Web Imágenes Grupos Directorio News

problemas computacionales Búsqueda Búsqueda avanzada Preferencias

Búsqueda:  la Web  páginas en español  páginas de España

**La Web** Resultados 71 - 80 de aproximadamente 31.400 de **problemas computacionales**. (0,23 segundos)

[PDF] UNIVERSIDAD DEL PEDREGAL Ingeniería en Sistemas Computacionales y ...  
Formato de archivo: PDF/Adobe Acrobat - Versión en PDF  
... enfocadas a reconocer y suavizar los problemas de comunicación entre todos sus niveles.  
Page 3. Escuela de Ingeniería en Sistemas Computacionales y Telemática  
[www.upedregal.edu.mx/veratura/ProyectoIndividuoProy.pdf](http://www.upedregal.edu.mx/veratura/ProyectoIndividuoProy.pdf) Páginas similares

**DISEÑO DE UN GENERADOR DE SISTEMAS COMPUTACIONALES PARA LA ...**  
... objetivo del proyecto es construir un generador de sistemas **computacionales** para  
la ... lenguajes de alto nivel, la representación de **problemas**, a algoritmos ...  
[www.fondef.cl/~fondef/PROYECTO/92/I/D92I1028.HTML](http://www.fondef.cl/~fondef/PROYECTO/92/I/D92I1028.HTML) - 26k - En caché - Páginas similares

**Ofrecemos hacernos cargo de:**  
... Nosotros nos hacemos cargo de sus **problemas computacionales** para que usted se dedique,  
integralmente, a cuidar del gerenciamiento de su negocio, para esto hay ...  
[usuarios.vtr.net/~xaspors/consultoria/body\\_home.htm](http://usuarios.vtr.net/~xaspors/consultoria/body_home.htm) - 5k - En caché - Páginas similares

Internet

# Planificador de rutas

 Guía Campsa España 2000, **INTERACTIVA**

Población Cagitan

**Cagitan**

NUCLEO

Población: 32 habitantes  
Superficie: 45097 m<sup>2</sup>  
Altitud: 460 m  
Provincia: MURCIA

Itinerario

Cagitan CORUÑA, LA A CORUÑA

Calcular ruta

Más Rápida  
Más Corta  
Incidencias

Mapa de España

61



# Planificador de rutas

Guía Campsa España 2000, **INTERACTIVA**

Población Cagitan

**Cagitan**

NUCLEO

Población: 32 habitantes  
Superficie: 45097 m<sup>2</sup>  
Altitud: 460 m  
Provincia: MURCIA

**Informe del Itinerario**

Origen: Cagitan  
Destino: CORUÑA, LA/ A CORUÑA

Distancia: 959.2 Km  
Tiempo: 9 h 49 m

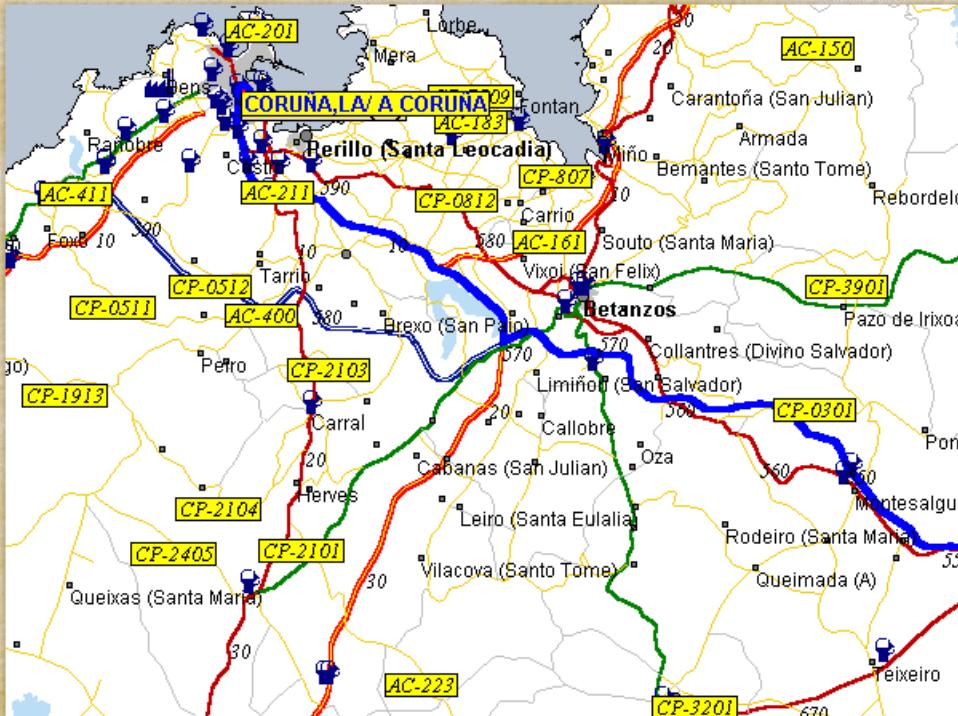
Hito

- Cagitan por la C-330
- C-330→N-301a
- N-301a-Km S.N. - CIEZA
- CIEZA
- N-301a→N-301
- N-301 - Km 316,1 - HELLIN
- N-301 - Km 233 - TOBARRA

62

# Planificador de rutas

 Guía Campsa España 2000, **INTERACTIVA**



Map icons:

Población Cagitan

**Cagitan**

**NUCLEO**

**Población:** 32 habitantes  
**Superficie:** 45097 m<sup>2</sup>  
**Altitud:** 460 m  
**Provincia:** MURCIA

**Tráfico**

Autopistas
Autovías
Nacionales
1er Orden
2º Orden
Locales
Otras

**Poblaciones**

Capitales de Provincia	<input checked="" type="checkbox"/> TOLEDO
Más de 20.000 habitantes	<input checked="" type="radio"/> COSLADA
De 5.000 a 20.000 habitantes	<input checked="" type="radio"/> Aguadulce
De 1.000 a 5.000 habitantes	<input checked="" type="radio"/> Leitza
Menos de 1.000 habitantes	<input checked="" type="radio"/> Rabanera del Pinar

**Informe del Itinerario**

**Origen:** Cagitan  
**Destino:** CORUÑA, LA/ A CORUÑA

**Distancia:** 959.2 Km  
**Tiempo:** 9 h 49 m

Hito

- Cagitan por la C-330
- C-330 → N-301a
- N-301a - Km S.N. - CIEZA
- CIEZA
- N-301a → N-301
- N-301 - Km 316,1 - HELLIN
- N-301 - Km 233 - TOBARRA

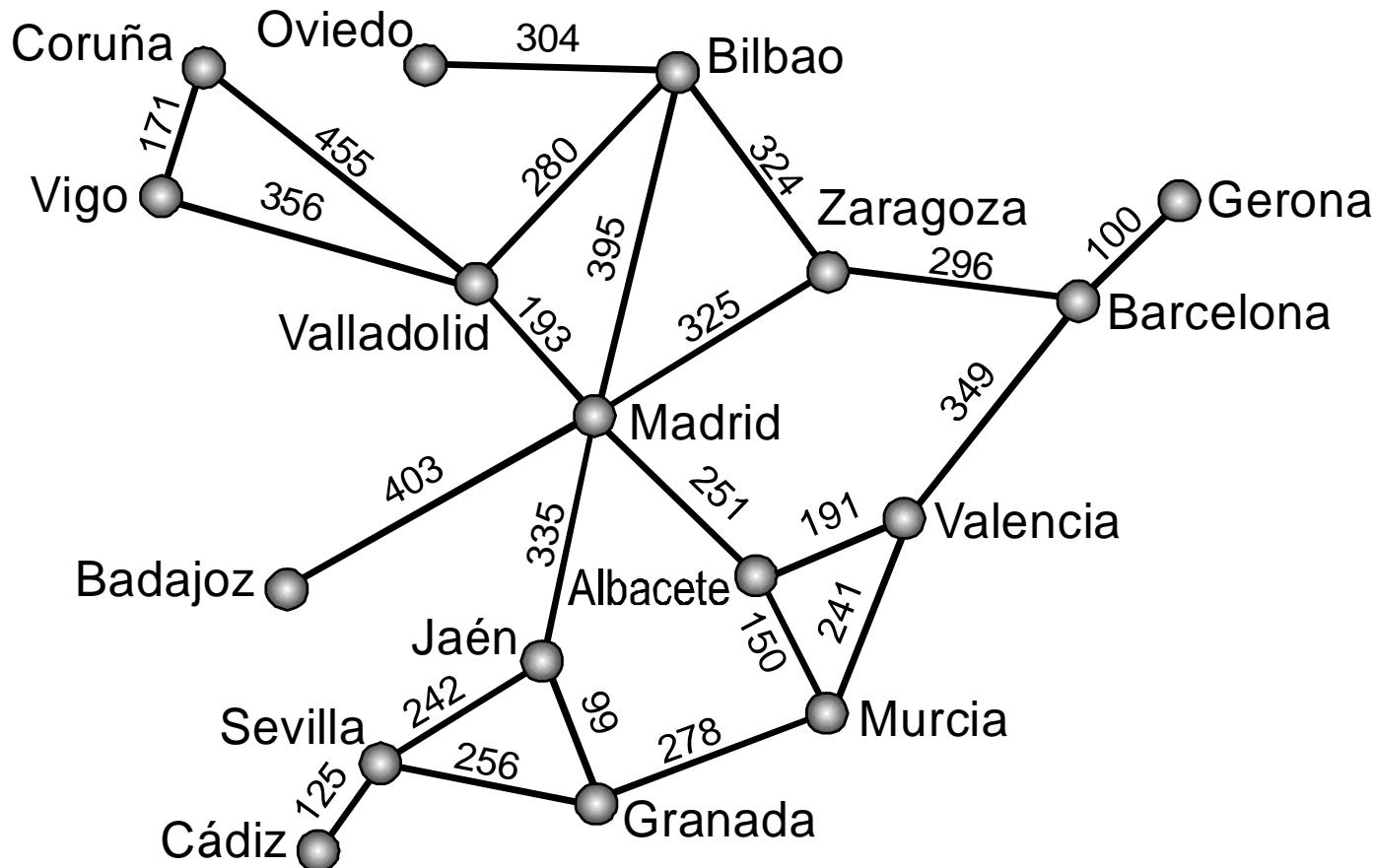
Distancia

# Planificador de rutas

- ¿Cómo representar la información (lugares y carreteras)?
- ¿Cómo calcular el camino más corto entre dos lugares?

# Planificador de rutas

- Representación mediante un **grafo**:
  - Lugares = nodos.
  - Carreteras = arcos entre nodos.



# Planificador de rutas

- ¿Cómo calcular los caminos mínimos en el mapa?
- Fuerza bruta: empezar por un sitio y probar todos los caminos hasta llegar a otro. NO!!!!
- Algorítmos de camino mínimo

# Planificador de rutas

**RESUELTO**

**Guía Campsa España 2000, INTERACTIVA**

Población Cagitan

**Cagitan**

NUCLEO

Population: inhabitants

Surface area: 097 m<sup>2</sup>

Elevation: 460 m

Province: ASTURIAS

**Itinerario**

Cagitan → CORUÑA/LA A CORUÑA

Calcular ruta

Más Rápida  
Más Corta  
Alternativas

Map of Spain showing provincial boundaries.

# Jugador de Ajedrez

Game 6, black  
19...Kasparov  
resigns!



# Jugador de Ajedrez

- ¿Cómo representar el problema?
- ¿Cómo decidir el siguiente movimiento de forma “inteligente”?    ¿?

# Jugador de Ajedrez

Situación  
Inicial



Movimien-  
tos de A



Movimien-  
tos de B



Movimien-  
tos de A



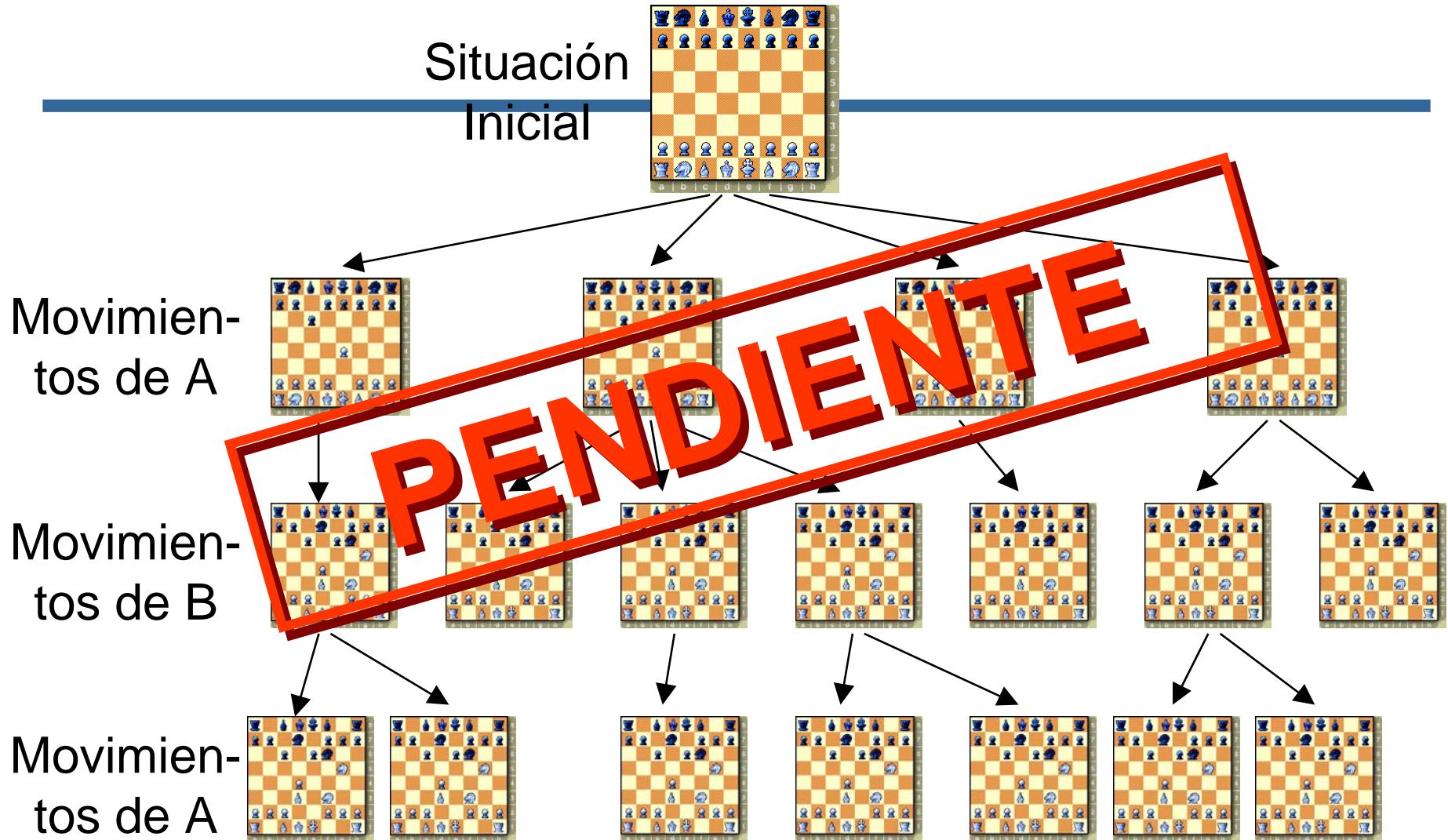
# Jugador de Ajedrez

- El **árbol de juego** del ajedrez representa todas las posibles partidas del juego.
- **Solución:** encontrar un camino en el árbol que llegue hasta la victoria.
- ¿Qué tamaño tiene el árbol de juego del ajedrez?

# Jugador de Ajedrez

- Suponiendo que cada jugador hace unos 50 movimientos, el factor de ramificación medio es de 35 posibles movimientos.
- Tamaño del árbol:  $35^{100} = 2,5 \cdot 10^{154}$
- ¡¡Sólo existen  $10^{87}$  partículas subatómicas en el universo!!

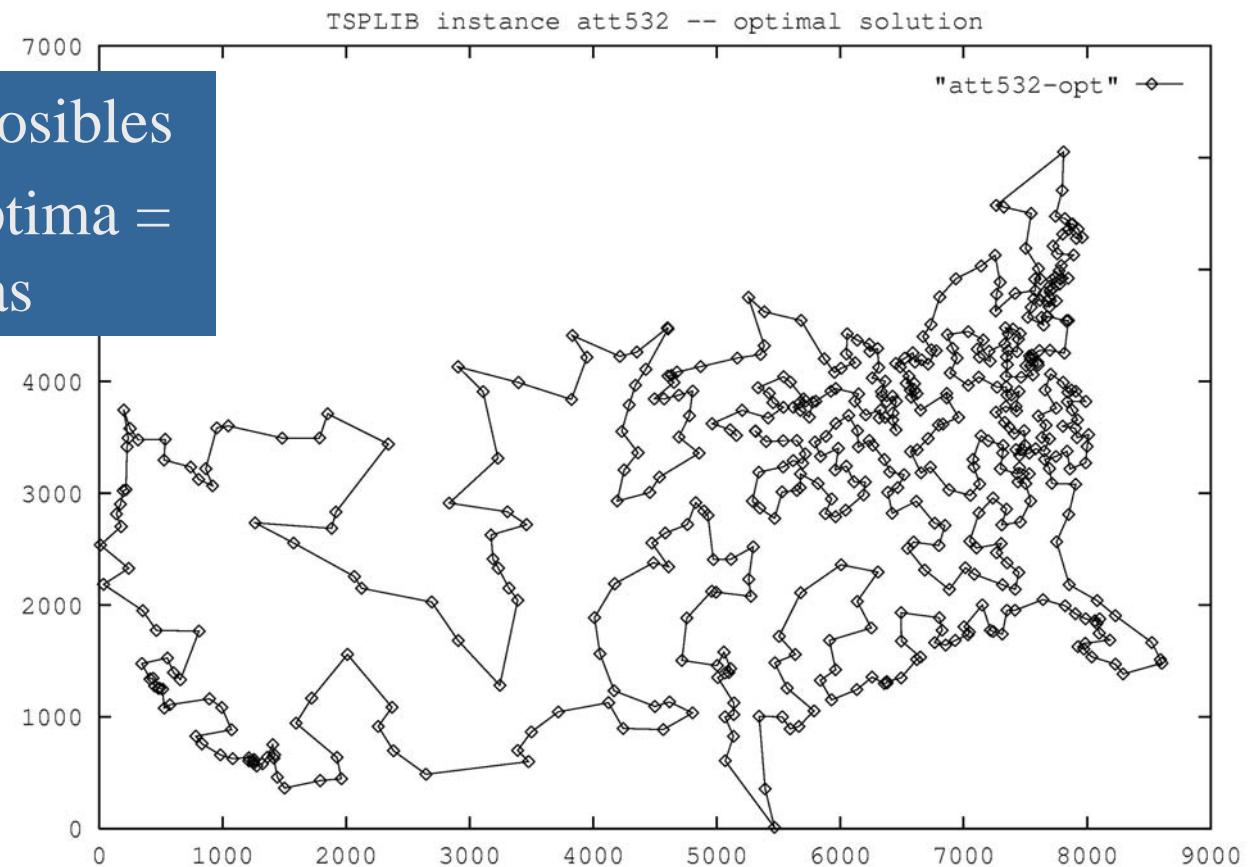
# Jugador de Ajedrez



# El problema del viajante

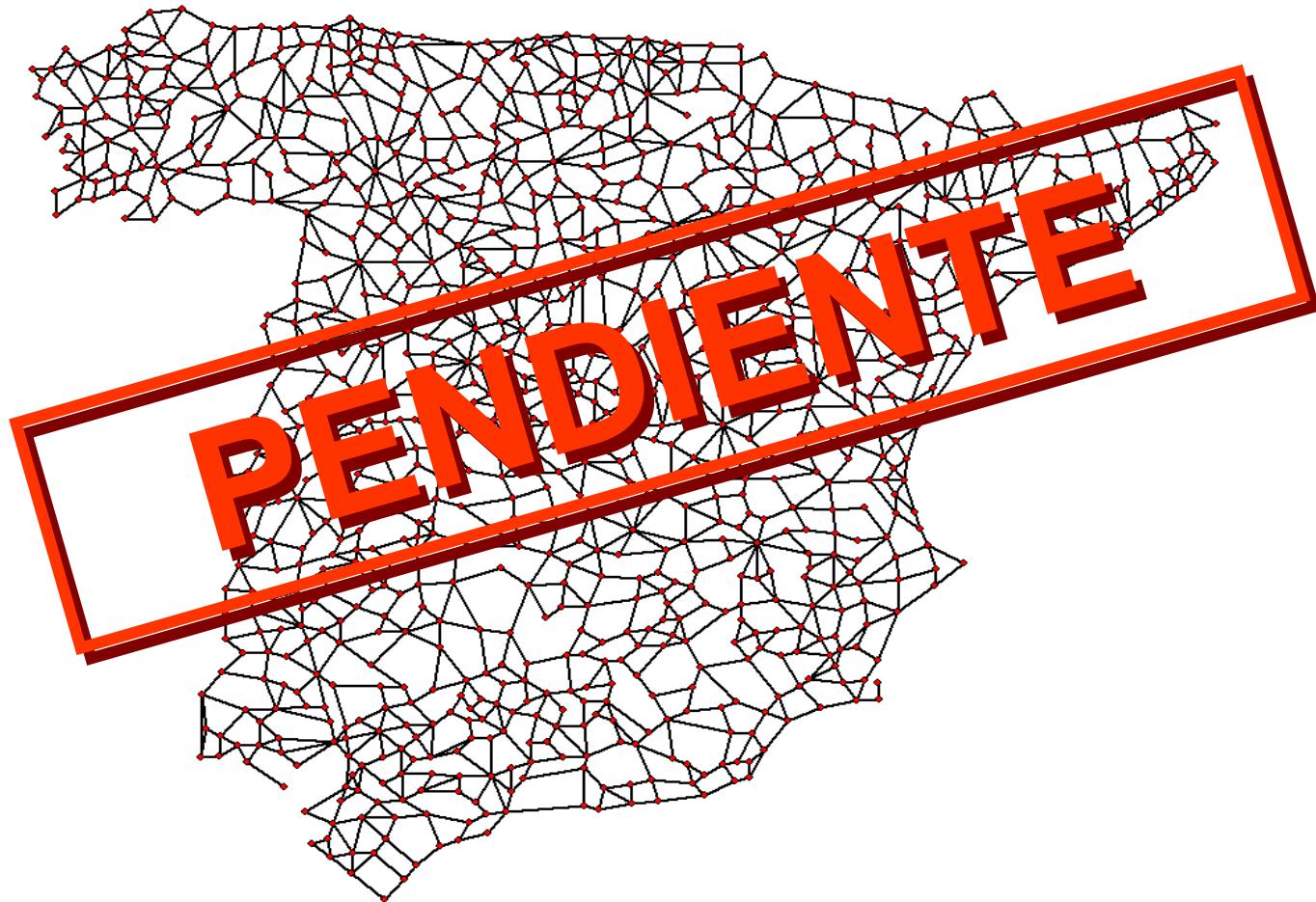
*Ejemplo de Viajante de Comercio con 532 ciudades*

532! soluciones posibles  
Coste solución óptima =  
27.686 millas



# Problema del viajante

---



# Resumen

---

- **Técnicas de diseño** de algoritmos:
  1. Divide y vencerás.
  2. Algoritmos voraces.
  3. Programación dinámica.
  4. Backtracking. Ramificación y poda.
- **Dado un problema:** seleccionar la técnica, seguir el proceso/esquema algorítmico, obtener el algoritmo y comprobarlo.
- **Recordar:** No empezar tecleando código como locos.

# Algorítmica

---

**Tema 1. Planteamiento General**

**Tema 2. La Eficiencia de los Algoritmos**

**Tema 3. Algoritmos “Divide y vencerás”**

**Tema 4. Algoritmos Voraces (“Greedy”)**

**Tema 5. Algoritmos basados en Programación Dinámica**

**Tema 6. Algoritmos para la Exploración de Grafos**

**(“Backtracking”, “Branch and Bound”)**

**Tema 7. Otras metodologías algorítmicas**

# Algorítmica

---

**Tema 1. Planteamiento General**

**Tema 2. La Eficiencia de los Algoritmos**

**Tema 3. Algoritmos “Divide y vencerás”**

**Tema 4. Algoritmos Voraces (“Greedy”)**

**Tema 5. Algoritmos basados en Programación Dinámica**

**Tema 6. Algoritmos para la Exploración de Grafos**

**(“Backtracking”, “Branch and Bound”)**

**Tema 7. Otras metodologías algorítmicas**

# Tema 2: La Eficiencia de los Algoritmos

---

**Bibliografía:**

**G. BRASSARD, P. BRATLEY. Fundamentos de Algoritmia. Prentice Hall (1997).**

# Objetivos

---

- Principio de Invarianza
- Concepto de eficiencia
- Concepto de operación elemental
- Notaciones asintóticas
- Saber calcular la eficiencia de un algoritmo
- Dominar la técnica de la ecuación recurrente

# Comparación de algoritmos

---

- Es frecuente disponer de más de un algoritmo para resolver un problema dado:
- ¿Con cuál nos quedamos?
- Estudio de los recursos:
  - **Tiempo**
  - Espacio

# Uso de recursos

---

- Los recursos empleados se distribuyen en:
  - Diseño
  - Implementación
  - Explotación

# Factores que influyen en el uso de los recursos

---

- Hardware: arquitectura, velocidad, etc.
- Codificador: Calidad del código creado
- Compilador: Calidad del código generado
- Diseñador: algoritmo
- Tamaño de las entradas

# Principio de Invarianza

---

- Dos implementaciones de un mismo algoritmo no diferirán más que en una constante multiplicativa.
- Si  $t_1(n)$  y  $t_2(n)$  son los tiempos de dos implementaciones de un mismo algoritmo, se verifica:

$$\exists c, d \in \mathbb{R}, \quad t_1(n) \leq ct_2(n); \quad t_2(n) \leq dt_1(n)$$

# Eficiencia

---

- Medida del uso de los recursos en función del tamaño de las entradas
- $T(n)$ : tiempo empleado para una entrada de tamaño  $n$

# Eficiencia (II)

---

- Dos algoritmos para un mismo problema:
- Algoritmo 1:  $T(n) = 10^{-4} 2^n$  segs. (la constante  $10^{-4}$  representa la velocidad de la máquina.)  
 $n=38, T(n) = 1$  año
- Algoritmo 2:  $T(n) = 10^{-2} n^3$  segs. (la constante  $10^{-2}$  representa la velocidad de la máquina.)  
 $n=1000, T(n) = 1$  año
- Se precisa **análisis asintótico**

- **Criterio empresarial:** Maximizar la eficiencia.
- **Eficiencia:** Relación entre los recursos consumidos y los productos conseguidos.
- **Recursos consumidos:**
  - Tiempo de ejecución.
  - Memoria principal.
  - Entradas/salidas a disco.
  - Comunicaciones, procesadores,...
- **Lo que se consigue:**
  - Resolver un problema de forma exacta.
  - Resolverlo de forma aproximada.
  - Resolver algunos casos...

- **Recursos consumidos.**

**Ejemplo.** ¿Cuántos recursos de tiempo y memoria consume el siguiente algoritmo sencillo?

```
i:= 0  
a[n+1]:= x  
repetir  
    i:= i + 1  
hasta a[i] == x
```

- **Respuesta:** Depende.

- ¿De qué depende?

- De lo que valga **n** y **x**, de lo que haya en **a**, de los tipos de datos, de la máquina...

- Factores que influyen en el consumo de recursos:
  - **Factores externos.**
    - El ordenador donde se ejecute.
    - El lenguaje de programación y el compilador usado.
    - La implementación que haga el programador del algoritmo.  
En particular, de las estructuras de datos utilizadas.
  - **Tamaño de los datos de entrada.**
    - Ejemplo. Procesar un fichero de blog: número de mensajes.
  - **Contenido de los datos de entrada.**
    - **Mejor caso ( $t_m$ ).** El contenido favorece una rápida ejecución.
    - **Peor caso ( $t_M$ ).** La ejecución más lenta posible.
    - **Caso promedio ( $t_p$ ).** Media de todos los posibles contenidos.

- Los factores externos no aportan información sobre el algoritmo.
- **Conclusión:** Estudiar la variación del tiempo y la memoria necesitada por un algoritmo respecto al tamaño de la entrada y a los posibles casos, de forma aproximada (y parametrizada).
- **Ejemplo.** Algoritmo de búsqueda secuencial.
  - Mejor caso. Se encuentra  $x$  en la 1<sup>a</sup> posición:  
$$t_m(N) = a$$
  - Peor caso. No se encuentra  $x$ :  
$$t_M(N) = b \cdot N + c$$
- **Ojo:** El mejor caso no significa tamaño pequeño.

Normalmente usaremos la notación  $t(N)=\dots$ , pero  
¿qué significa  $t(N)$ ?

- Tiempo de ejecución en segundos.  $t(N) = bN + c$ .
  - Suponiendo que  $b$  y  $c$  son constantes, con los segundos que tardan las operaciones básicas correspondientes.
- Instrucciones ejecutadas por el algoritmo.  
 $t(N) = 2N + 4$ .
  - ¿Tardarán todas lo mismo?
- Ejecuciones del bucle principal.  $t(N) = N+1$ .
  - ¿Cuánto tiempo, cuántas instrucciones,...?
  - Sabemos que cada ejecución lleva un tiempo constante, luego se diferencia en una constante con los anteriores.

- El proceso básico de análisis de la eficiencia algorítmica es el conocido como **conteo de instrucciones** (o de memoria).
- **Conteo de instrucciones:** Seguir la ejecución del algoritmo, sumando las instrucciones que se ejecutan.
- **Conteo de memoria:** Lo mismo. Normalmente interesa el máximo uso de memoria requerido.

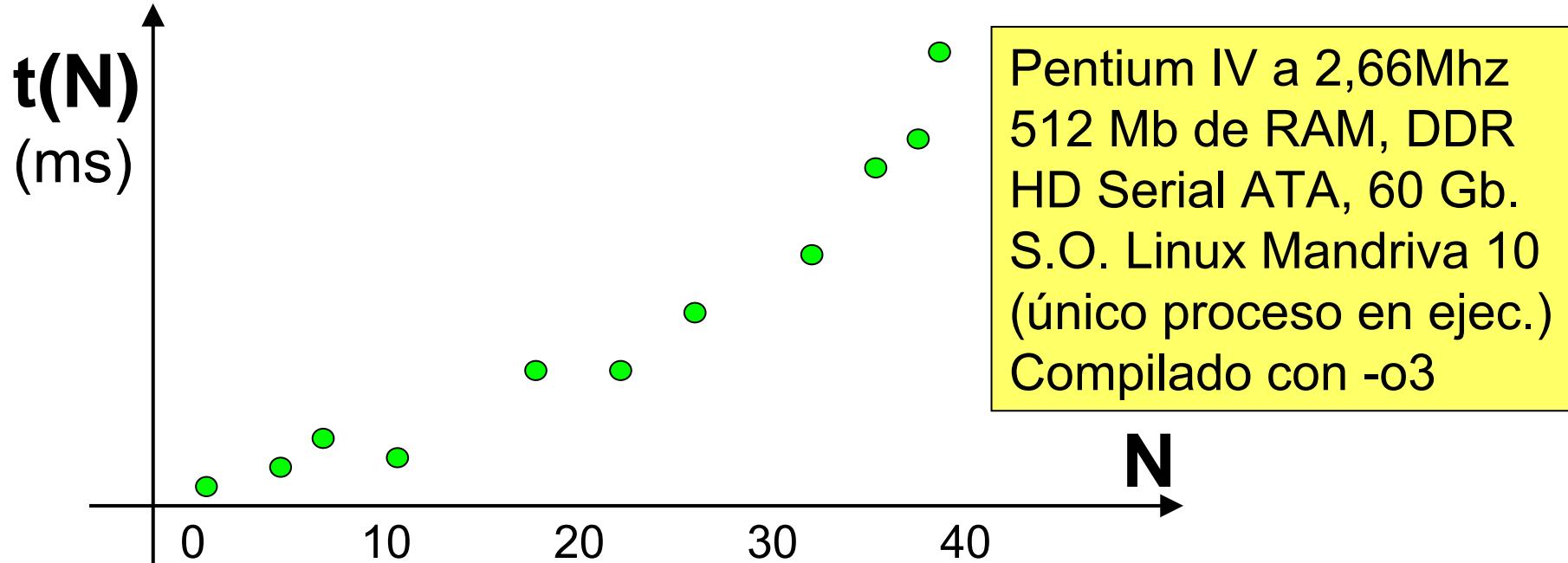
# Medidas de la eficiencia

---

- Enfoques de medida:
  - Enfoque práctico (*a posteriori*)
  - Enfoque teórico (*a priori*)
  - Enfoque híbrido
- Tipos de Análisis:
  - Peor caso
  - Mejor caso
  - Caso promedio
  - Análisis probabilístico
  - Análisis amortizado

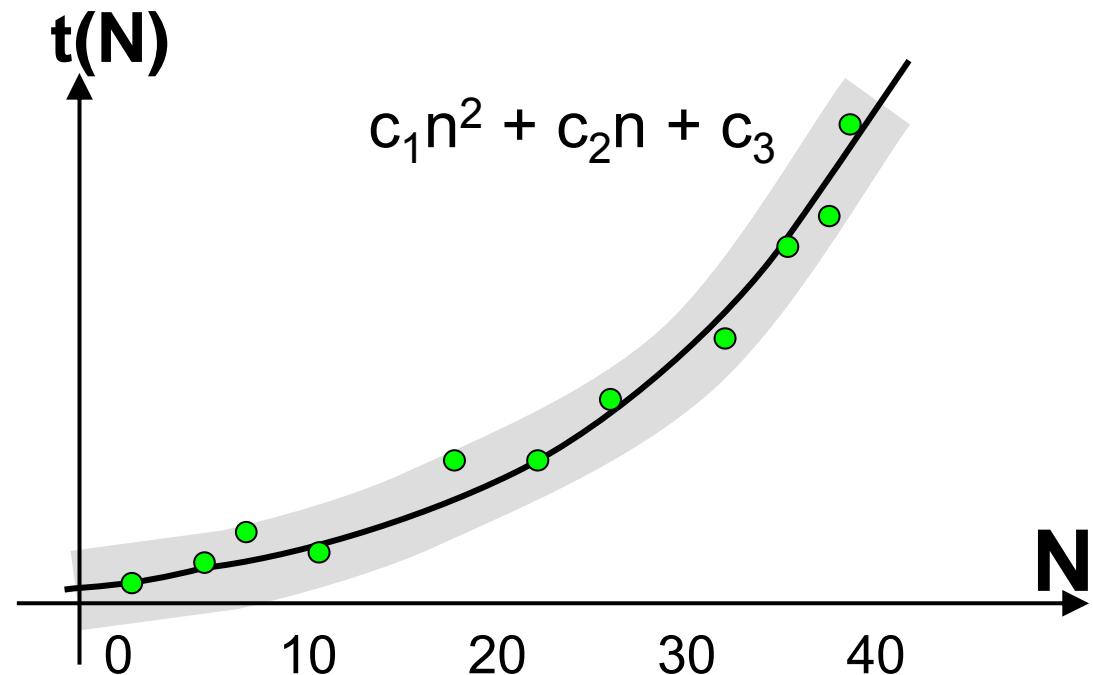
# Enfoque a posteriori

- El análisis de algoritmos puede hacerse **a posteriori**: implementar el algoritmo y contar lo que tarda para distintas entradas.
- En este caso, cobran especial importancia las **herramientas de la estadística**: representaciones gráficas, técnicas de muestreo, regresiones, tests de hipótesis, etc.
- Hay que ser muy **específicos**, indicar: ordenador, S.O., condiciones de ejecución, opciones de compilación, etc.



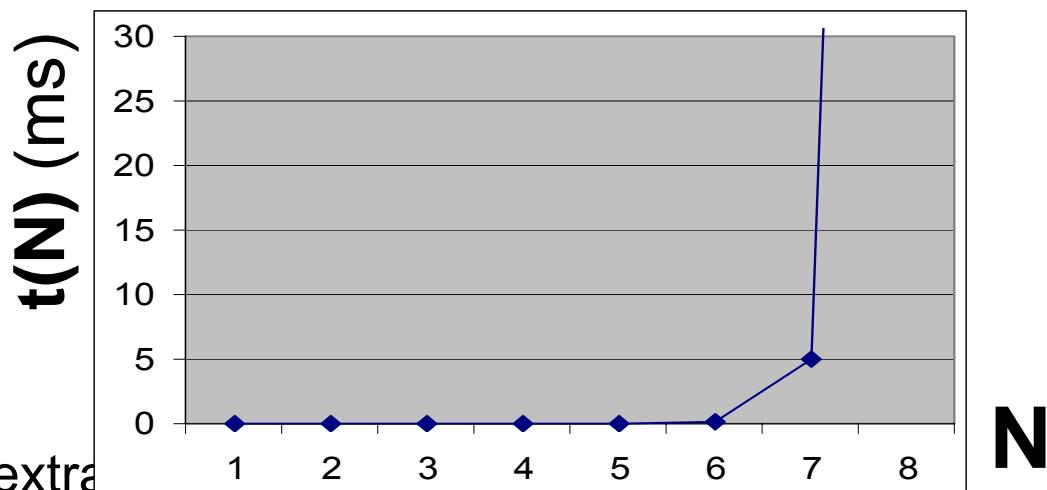
# Enfoque a posteriori

- Indicamos los **factores externos**, porque influyen en los tiempos (multiplicativamente), y son útiles para comparar tiempos tomados bajo condiciones distintas.
- La medición de los tiempos es un **estudio experimental**.
- El análisis a posteriori suele complementarse con un **estudio teórico** y un **contraste teórico/experimental**.
- **Ejemplo.** Haciendo el estudio teórico de un programa, deducimos que su tiempo es de la forma:  $c_1n^2 + c_2 n + c_3$
- Podemos hacer una regresión. → ¿Se ajusta bien? ¿Es correcto el estudio teórico?



# Enfoque a posteriori

- El contraste teórico/experimental **permite**: detectar posibles errores de implementación, hacer previsiones para tamaños inalcanzables, comparar implementaciones.
- Sin el estudio teórico, extraer conclusiones relevantes del tiempo de ejecución puede ser complejo.
- **Ejemplo.** Para un cierto programa:
  - $N= 4, T(4)= 0.1 \text{ ms}$
  - $N= 5, T(5)= 5 \text{ ms}$
  - $N= 6, T(6)= 0.2 \text{ s}$
  - $N= 7, T(7)= 10 \text{ s}$
  - $N= 8, T(8)= 3.5 \text{ min}$
- ¿Qué conclusiones podemos extraer?
- El **análisis a priori** es siempre un estudio teórico previo a la implementación. Puede servir para evitar la implementación, si el algoritmo es poco eficiente.



# Notación asintótica

---

- Estudia el comportamiento del algoritmo cuando el tamaño de las entradas,  $n$ , es lo suficientemente grande, sin tener en cuenta lo que ocurre para entradas pequeñas y obviando factores constantes
- Notaciones:  $O$ ,  $\Omega$ ,  $\Theta$  ( $o$ , omega y tita)

# Órdenes de eficiencia

---

Un algoritmo tiene un *tiempo de ejecución de orden*  $T(n)$ , para una función dada  $T$ , si existe una constante positiva  $c$ , y una implementación del algoritmo capaz de resolver cada caso del problema en un tiempo acotado superiormente por  $cT(n)$ , donde  $n$  es el tamaño del problema considerado

# Órdenes más habituales

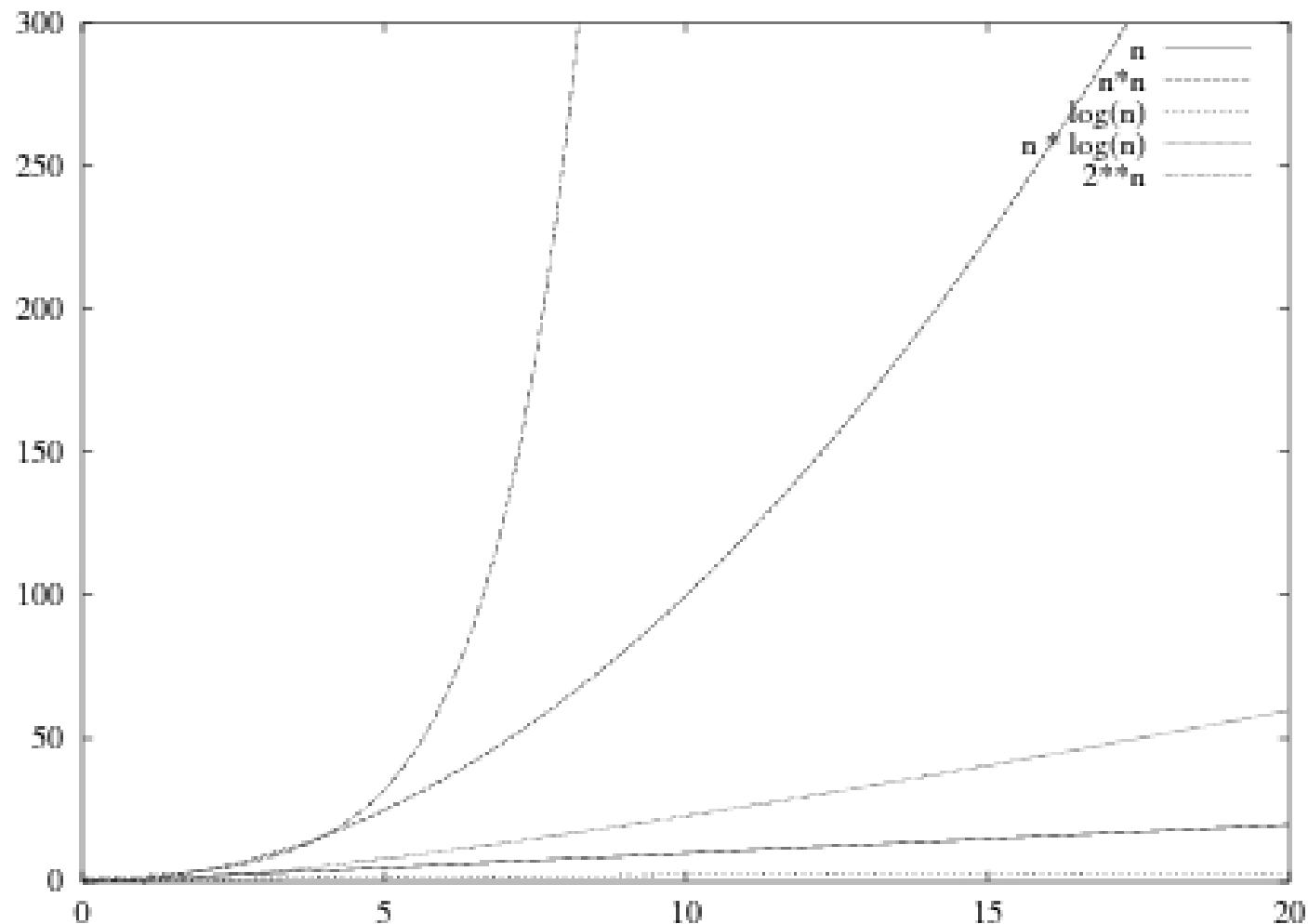
---

- Lineal:  $n$
- Cuadrático:  $n^2$
- Polinómico:  $n^k$ , con  $k \in \mathbb{N}$
- Logarítmico:  $\log(n)$
- Exponencial:  $c^n$

$\log(n)$	$n$	$n^2$	$n^5$	$2^n$
1	2	4	32	4
2	4	16	1024	16
3	8	64	32768	256
4	16	256	1048576	65536
5	32	1024	33554432	4.29E+09
6	64	4096	1.07E+09	1.84E+19
7	128	16384	3.44E+10	3.4E+38
8	256	65536	1.1E+12	1.16E+77
9	512	262144	3.52E+13	1.3E+154
10	1024	1048576	1.13E+15	#NUM!

$$1 < \log(n) < \log^2(n) < \sqrt{n} < n < n * \log(n) < n^2 < 2^n$$

# Órdenes más habituales (gráfico)



- El tiempo de ejecución  $T(n)$  está dado en base a unas constantes que dependen de factores externos.
- Nos interesa un análisis que sea independiente de esos factores.
- **Notaciones asintóticas:** Indican como crece  $T$ , para valores suficientemente grandes (asintóticamente) sin considerar constantes.
- $O(T)$ : Orden de complejidad de  $T$ .
- $\Omega(T)$ : Orden inferior de  $T$ , u omega de  $T$ .
- $\Theta(T)$ : Orden exacto de  $T$ .

# Notación O-Mayúscula

---

- Decimos que una función  $T(n)$  es  $O(f(n))$  si existen constantes  $n_0$  y  $c$  tales que  $T(n) \leq cf(n)$  para  $n \geq n_0$ :
- $T(n)$  es  $O(f(n)) \Leftrightarrow \exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N}$ , tal que  $\forall n \geq n_0, T(n) \leq cf(n)$

# Ejemplos en O(.)

---

- $T(n) = (n + 1)^2$  es  $O(n^2)$
- $T(n) = 3n^3 + 2n^2$  es  $O(n^3)$
- $T(n) = 3^n$  no es  $O(2^n)$

*Flexibilidad en la notación:* Emplearemos la notación  $O(f(n))$  aun cuando en un número finito de valores de  $n$ ,  $f(n)$  sea negativa o no esté definida. Ej.:  $n/\log(n)$

# Interpretación de la definición de O

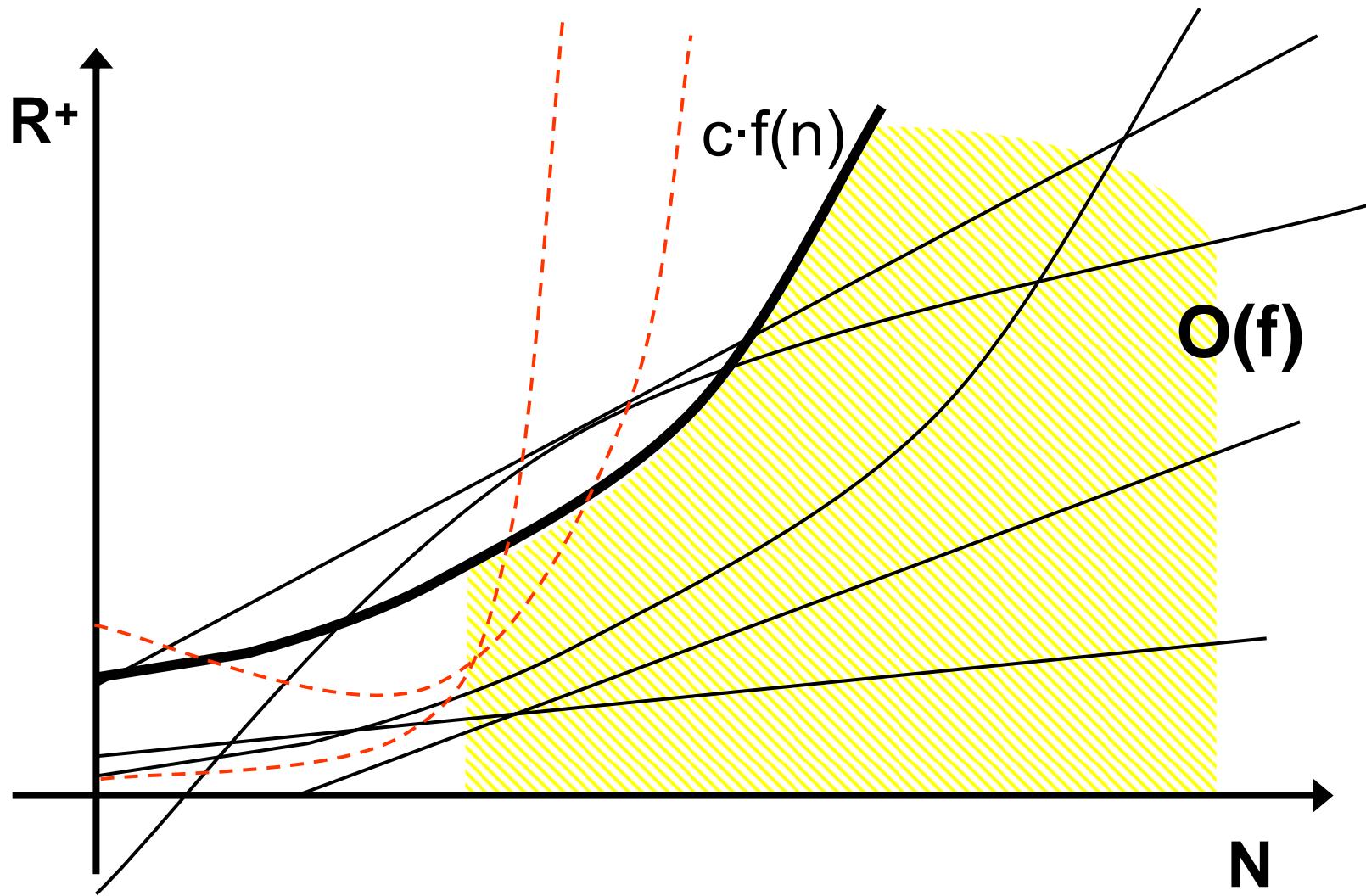
## Orden de complejidad de $f(n)$ : $O(f)$

- Dada una función  $f: \mathbb{N} \rightarrow \mathbb{R}^+$ , llamamos **orden de f** al conjunto de todas las funciones de  $\mathbb{N}$  en  $\mathbb{R}^+$  acotadas superiormente por un múltiplo real positivo de  $f$ , para valores de  $n$  suficientemente grandes.

$$O(f) = \{ t: \mathbb{N} \rightarrow \mathbb{R}^+ / \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0; t(n) \leq c \cdot f(n) \}$$

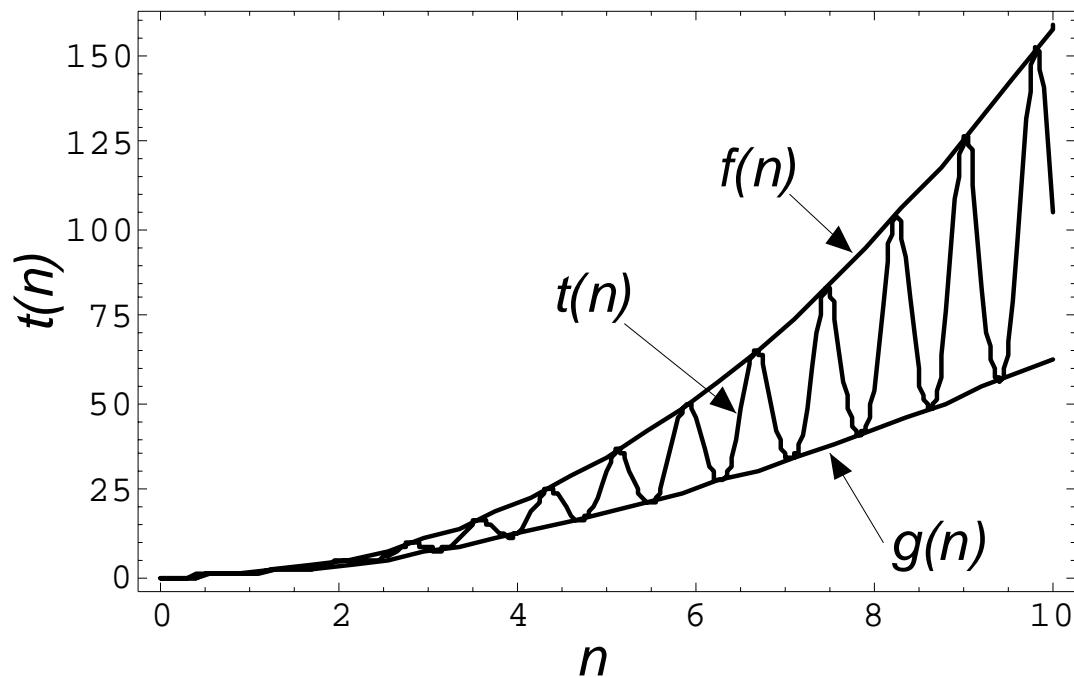
## Observaciones:

- $O(f)$  es un **conjunto de funciones**, no una función.
- “Valores de  $n$  suficientemente grandes...”: no nos importa lo que pase para valores pequeños.
- “Funciones acotadas superiormente por un múltiplo de  $f\dots$ ”: nos quitamos las constantes multiplicativas.
- La definición es aplicable a cualquier función de  $N$  en  $R$ , no sólo tiempos de ejecución.



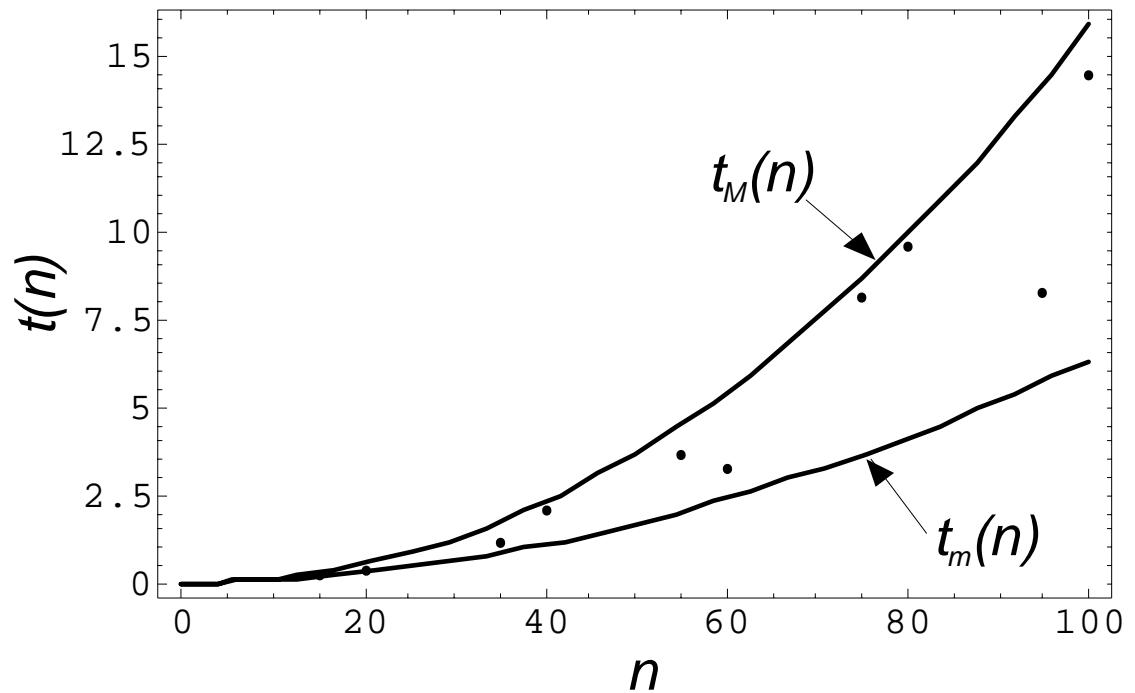
# Uso de los órdenes de complejidad

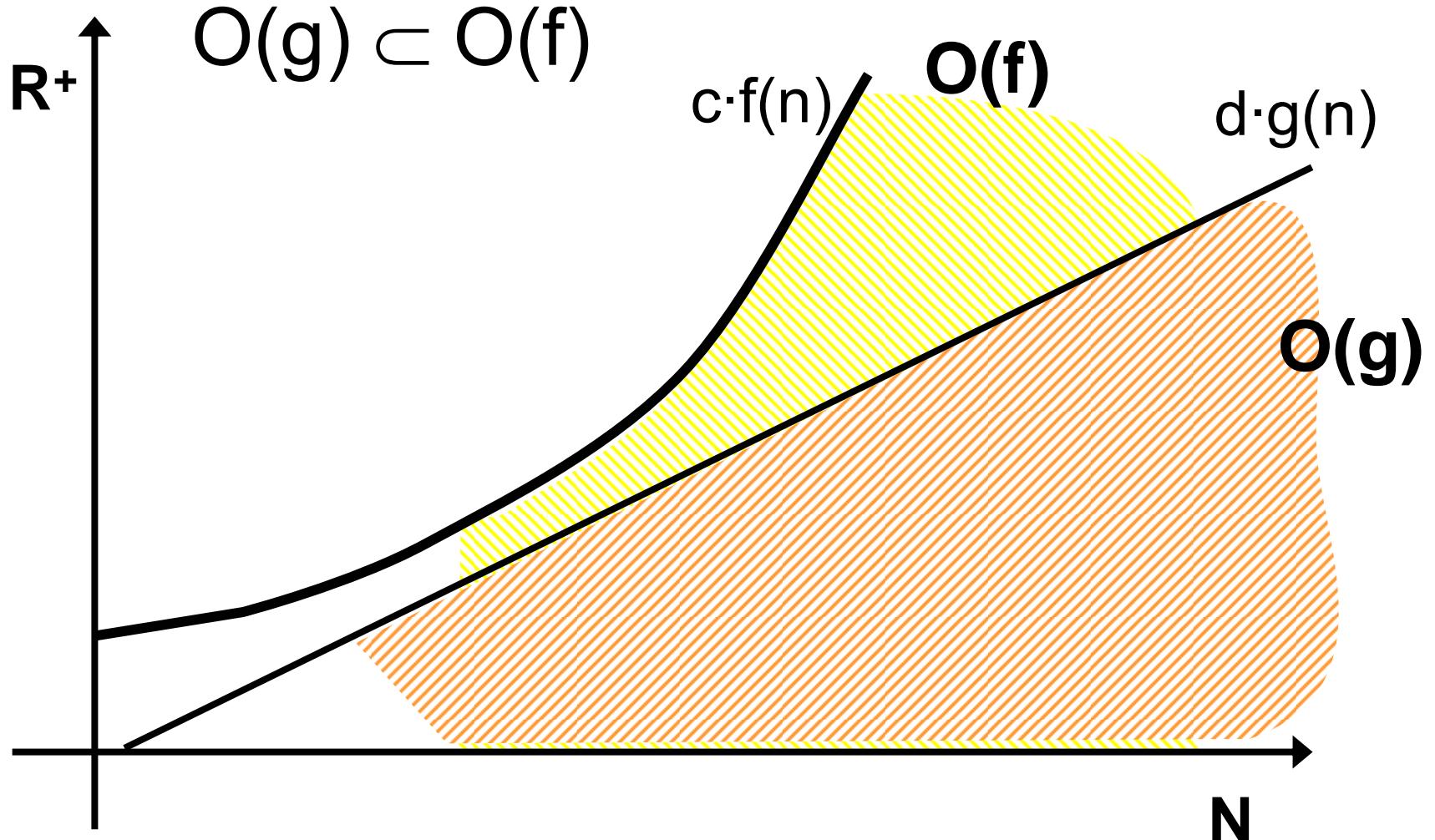
- 1) Dado un tiempo  $t(n)$ , encontrar la función  $f$  más simple tal que  $t \in O(f)$ , y que más se aproxime asintóticamente.
- **Ejemplo.**  $t(n) = 2n^2/5 + 6n + 3\pi \cdot \log_2 n + 2 \Rightarrow t(n) \in O(n^2)$
- 2) Acotar una función difícil de calcular con precisión.
- **Ejemplo.**  $t(n) \in O(f(n))$



# Uso de los órdenes de complejidad

- 3) Acotar una función que no tarda lo mismo para el mismo tamaño de entrada (distintos casos, mejor y peor).
- **Ejemplo.**  
 $t(n) \in O(t_M(n))$





Relación de orden entre  $O(\dots)$  = Relación de inclusión entre conjuntos.

$O(g) \leq O(f) \Leftrightarrow O(g) \subseteq O(f) \Leftrightarrow$  Para toda  $t \in O(g)$ ,  $t \in O(f)$

# Notaciones $\Omega$ y $\Theta$

---

Notación  $\Omega$ : cota inferior:

$T(n)$  es  $\Omega(f(n))$  cuando  $\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}: \forall n \geq n_0 \Rightarrow T(n) \geq cf(n)$

Notación  $\Theta$ : Orden exacto

$T(n)$  es  $\Theta(f(n))$  cuando

$T(n)$  es  $O(f(n))$  y  $T(n)$  es  $\Omega(f(n))$

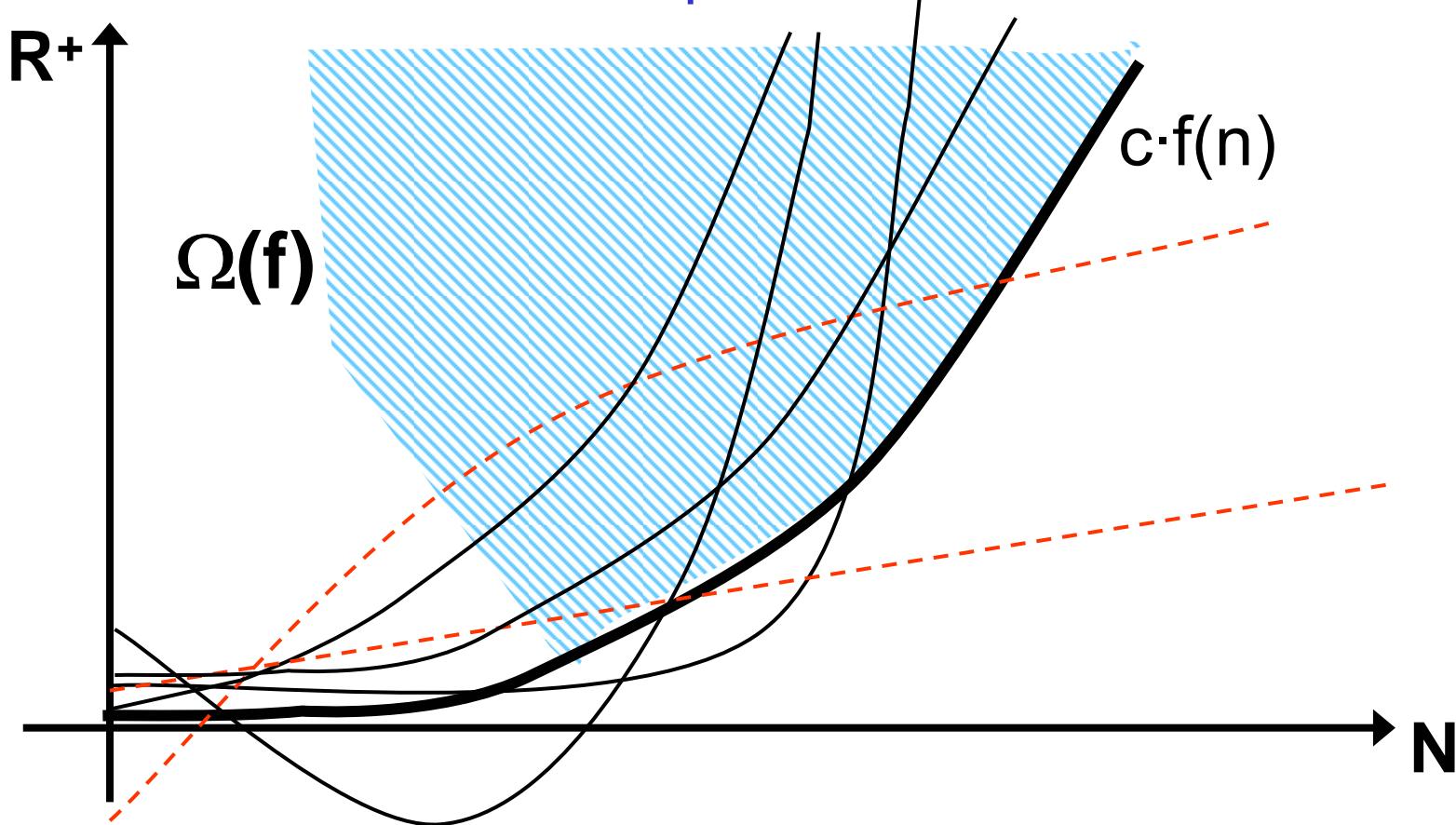
## Interpretación

### Orden inferior u omega de $f(n)$ : $\Omega(f)$

- Dada una función  $f: \mathbb{N} \rightarrow \mathbb{R}^+$ , llamamos **omega de  $f$**  al conjunto de todas las funciones de  $\mathbb{N}$  en  $\mathbb{R}^+$  acotadas **inferiormente** por un múltiplo real positivo de  $f$ , para valores de  $n$  suficientemente grandes.

$$\Omega(f) = \{ t: \mathbb{N} \rightarrow \mathbb{R}^+ / \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0; t(n) \geq c \cdot f(n) \}$$

## Interpretación



- La notación omega se usa para establecer cotas inferiores del tiempo de ejecución.
- **Relación de orden:** igual que antes, basada en la inclusión.

# Interpretación

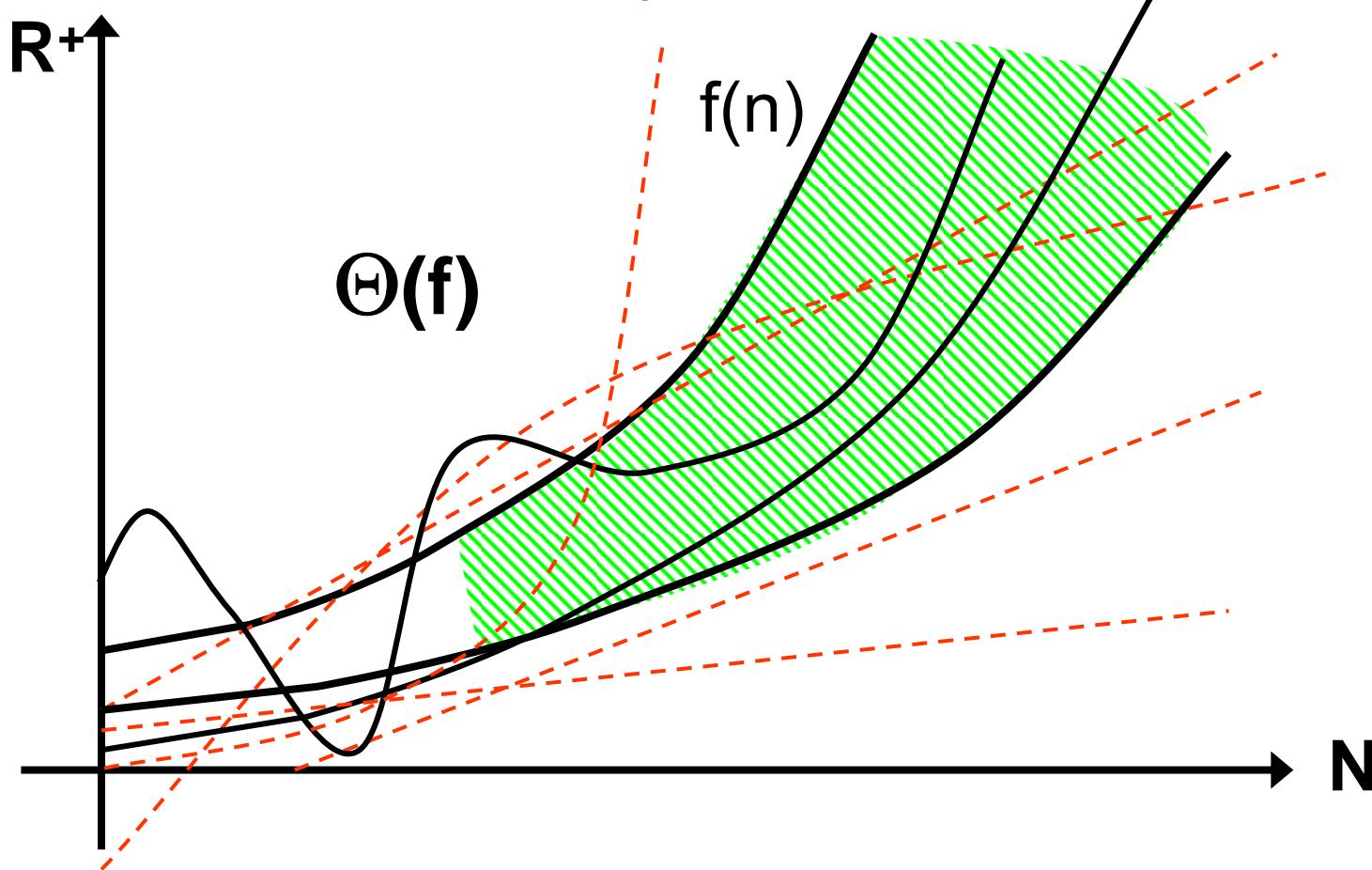
## Orden exacto de $f(n)$ : $\Theta(f)$

- Dada una función  $f: \mathbb{N} \rightarrow \mathbb{R}^+$ , llamamos orden **exacto de  $f$**  al conjunto de todas las funciones de  $\mathbb{N}$  en  $\mathbb{R}^+$  que crecen igual que  $f$ , asintóticamente y salvo constantes.

$$\Theta(f) = O(f) \cap \Omega(f) =$$

$$= \{ t: \mathbb{N} \rightarrow \mathbb{R}^+ / \exists c, d \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0; c \cdot f(n) \geq t(n) \geq d \cdot f(n) \}$$

## Interpretación



- Si un algoritmo tiene un  $t$  tal que  $t \in O(f)$  y  $t \in \Omega(f)$ , entonces  $t \in \Theta(f)$ .

- **Ejemplos. ¿Cuáles son ciertas y cuáles no?**

$$3n^2 \in O(n^2)$$

$$n^2 \in O(n^3)$$

$$n^3 \in O(n^2)$$

$$3n^2 \in \Omega(n^2)$$

$$n^2 \in \Omega(n^3)$$

$$n^3 \in \Omega(n^2)$$

$$3n^2 \in \Theta(n^2)$$

$$n^2 \in \Theta(n^3)$$

$$n^3 \in \Theta(n^2)$$

$$2^{n+1} \in O(2^n)$$

$$(2+1)^n \in O(2^n)$$

$$(2+1)^n \in \Omega(2^n)$$

$$O(n) \in O(n^2)$$

$$(n+1)! \in O(n!)$$

$$n^2 \in O(n!!)$$

# Propiedades de las notaciones asintóticas.

- **P1. Transitividad.**
  - Si  $f \in O(g)$  y  $g \in O(h)$  entonces  $f \in O(h)$ .
    - Si  $f \in \Omega(g)$  y  $g \in \Omega(h)$  entonces  $f \in \Omega(h)$
    - Ej.  $2n+1 \in O(n)$ ,  $n \in O(n^2) \Rightarrow 2n+1 \in O(n^2)$
- **P2. Si  $f \in O(g)$  entonces  $O(f) \subseteq O(g)$ .**
- **P3. Relación pertenencia/contenido.**

Dadas  $f$  y  $g$  de  $N$  en  $R^+$ , se cumple:

  - i)  $O(f) = O(g) \Leftrightarrow f \in O(g)$  y  $g \in O(f)$
  - ii)  $O(f) \subseteq O(g) \Leftrightarrow f \in O(g)$

# Propiedades de las notaciones asintóticas.

- **P4. Propiedad del máximo.**

Dadas  $f$  y  $g$ , de  $N$  en  $\mathbb{R}^+$ ,  $O(f+g) = O(\max(f, g))$ .

- Con omegas:  $\Omega(f+g) = \Omega(\max(f, g))$

- **P5. Relación límites/órdenes.**

Dadas  $f$  y  $g$  de  $N$  en  $\mathbb{R}^+$ , se cumple:

- i)  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+ \Rightarrow O(f) = O(g)$

- ii)  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow O(f) \subset O(g)$

- iii)  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow O(f) \supset O(g)$

## Notaciones con varios parámetros.

- En general, el tiempo y la memoria consumidos pueden depender de muchos parámetros.
- $f: \mathbf{N}^m \rightarrow \mathbf{R}^+$       ( $f: N \times \dots \times N \rightarrow \mathbf{R}^+$ )  
**Orden de complejidad de  $f(n_1, n_2, \dots, n_m)$ :  $O(f)$**
- Dada una función  $f: \mathbf{N}^m \rightarrow \mathbf{R}^+$ , llamamos **orden de f** al conjunto de todas las funciones de  $\mathbf{N}^m$  en  $\mathbf{R}^+$  acotadas superiormente por un múltiplo real positivo de  $f$ , para valores de  $(n_1, \dots, n_m)$  suficientemente grandes.

$$O(f) = \{ t: \mathbf{N}^m \rightarrow \mathbf{R}^+ / \exists c \in \mathbf{R}^+, \exists n_1, n_2, \dots, n_m \in \mathbf{N}, \forall k_1 \geq n_1, \\ \forall k_2 \geq n_2, \dots, \forall k_m \geq n_m; t(k_1, k_2, \dots, k_m) \leq c \cdot f(k_1, k_2, \dots, k_m) \}$$

## Notaciones con varios parámetros.

- **Ejemplo.** Tiempo de ejecución de la BPP con listas de adyacencia:  $O(n+a)$ .

Memoria usada en una tabla hash: depende del número de cubetas, elementos, tamaño de celda...

- Podemos extender los conceptos de  $\Omega(f)$  y  $\Theta(f)$ , para funciones con varios parámetros.
- Las propiedades se siguen cumpliendo
- Ejemplos:  
 $O(n+m)$ ,  $O(n^m)$ ,  $O(n+2^m)$

## Notaciones condicionales.

- En algunos casos interesa estudiar el tiempo sólo para ciertos tamaños de entrada.
- **Ejemplo.** Algoritmo de búsqueda binaria: Si  $N$  es potencia de 2 el estudio se simplifica.

### Orden condicionado de $f(n)$ : $O(f | P)$

- Dada una función  $f: \mathbb{N} \rightarrow \mathbb{R}^+$ , y  $P: \mathbb{N} \rightarrow \mathbb{B}$ , llamamos **orden de f según P** (o condicionado a P) al conjunto:  
$$O(f | P) = \{ t: \mathbb{N} \rightarrow \mathbb{R}^+ / \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0;$$
  
$$P(n) \Rightarrow t(n) \leq c \cdot f(n) \}$$

# Notaciones condicionales.

- De igual forma, tenemos  $\Omega(f | P)$  y  $\Theta(f | P)$ .
- **Ejemplo.**
  - Si estudiamos el tiempo para tamaños de entrada que sean potencia de 2:  
 $t(n) \in O(f | n = 2^k)$
  - Para tamaños que sean múltiplos de 2:  
 $t(n) \in O(f | n = 2k)$

## Cotas de complejidad frecuentes.

- **Algunas relaciones entre órdenes frecuentes.**

$$\begin{aligned} O(1) &\subset O(\log n) \subset O(n) \subset O(n \cdot \log n) \subset \\ O(n \cdot (\log n)^2) &\subset O(n^{1.001\dots}) \subset O(n^2) \subset O(n^3) \subset \dots \\ &\subset O(2^n) \subset O(n!) \subset O(n^n) \end{aligned}$$

- Se establece una jerarquía de órdenes

# Cotas de complejidad frecuentes.

- El orden de un polinomio  $a_nx^n + \dots + a_1x + a_0$  es  $O(x^n)$ .
- $\sum_{i=1}^n 1 \in O(n)$ ;  $\sum_{i=1}^n i \in O(n^2)$ ;  $\sum_{i=1}^n i^m \in O(n^{m+1})$
- Si hacemos una operación para  $n$ , otra para  $n/2$ ,  $n/4$ , ..., aparecerá un orden logarítmico  $O(\log_2 n)$ .
- Los **logaritmos** son del mismo orden, independientemente de la base. Por eso, se omite normalmente.
- **Sumatorios:** se pueden aproximar con integrales, una acotando superior y otra inferiormente.
- **Casos promedios:** usar probabilidades.

---

# Cálculo del Orden de Eficiencia

# Operación elemental

---

- Operación de un algoritmo cuyo tiempo de ejecución se puede acotar superiormente por una constante.

En nuestro análisis sólo contará el número de operaciones elementales y no el tiempo exacto necesario para cada una de ellas.

# Consideraciones sobre operaciones elementales

---

- En la descripción de un algoritmo puede ocurrir que una línea de código corresponda a un número variable de operaciones elementales.

Por ejemplo, si  $A$  es un vector con  $n$  elementos, y queremos calcular

$$x = \max\{A[k], 0 \leq k < n\}$$

el tiempo para hacerlo depende de  $n$ , no es constante

# Consideraciones sobre operaciones elementales

---

- Algunas operaciones matemáticas no deben ser tratadas como tales operaciones elementales. Por ejemplo, el tiempo necesario para realizar sumas y productos crece con la longitud de los operandos. Sin embargo, en la práctica se consideran elementales siempre que los datos que se usen tengan un tamaño razonable.
- No obstante, consideraremos las operaciones suma, diferencia, producto, cociente, módulo, operaciones booleanas, comparaciones y asignaciones como elementales, salvo que explícitamente se establezca otra cosa.

# Reglas del cálculo del tiempo de ejecución

---

1. Sentencias simples
2. Bucles
3. Sentencias condicionales
4. Bloques de sentencias
5. Llamadas a funciones
6. Funciones recursivas

# Sentencias simples

---

- Consideraremos que cualquier sentencia simple (lectura, escritura, asignación, etc.) va a consumir un tiempo constante,  $O(1)$ , salvo que contenga una llamada a función
- Número de instrucciones  $t(n) \rightarrow$  sumar 1 por cada instrucción o línea de código de ejecución constante.
- Tiempo de ejecución  $t(n) \rightarrow$  sumar una constante ( $c_1, c_2, \dots$ ) por cada tipo de instrucción o grupo de instrucciones secuenciales.

# Bucles

---

- El tiempo invertido en un bucle es la suma del tiempo invertido en cada iteración. Este tiempo debería incluir: el tiempo del cuerpo y el asociado a la evaluación de la condición y actualización.
- En un bucle donde todas las iteraciones son iguales, el tiempo total será el producto del número de iteraciones por el tiempo que requiere cada una.

# Introducción.

- **Bucles FOR:** Se pueden expresar como un sumatorio, con los límites del FOR como límites del sumatorio.

$$\sum_{i=1}^n k = kn$$

$$\sum_{i=a}^b k = k(b-a+1)$$

$$\sum_{i=1}^n i = n(n+1)/2$$

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

$$\sum_{i=1}^n i^2 \approx \int_0^n i^2 di = [i^3/3]_0^n = (n^3)/3$$

- **Bucles WHILE y REPEAT:** cota inferior y superior del número de ejecuciones ¿Se puede convertir en un FOR?

# Sentencias condicionales

---

- El tiempo de ejecución es el máximo tiempo de la parte if y de la parte else, de forma que si son, respectivamente,  $O(f(n))$  y  $O(g(n))$ , será  $O(\max(f(n), g(n)))$ .

# Bloques de sentencias

---

- Se aplica la regla de la suma, de forma que se calcula el tiempo de ejecución tomando el máximo de los tiempos de ejecución de cada una de las partes (sentencias individuales, bucles o condicionales) en que puede dividirse.

# Llamadas a funciones

---

- Si una determinada función P tiene una eficiencia de  $O(f(n))$  con n la medida del tamaño de los argumentos, cualquier función que llame a P tiene en la llamada una cota superior de eficiencia de  $O(f(n))$ 
  - Las asignaciones con diversas llamadas a función deben sumar las cotas de tiempo de ejecución de cada llamada
  - La misma consideración es válida para las condiciones de bucles y sentencias condicionales

# Ejemplos

- **Ejemplos.** Estudiar  $t(n)$ .

```
for i:= 1 to N
    for j:= 1 to N
        suma:= 0
        for k:= 1 to N
            suma:=suma+a[i,k]*a[k,j]
        end
        c[i, j]:= suma
    end
end
```

```
Funcion Fibonacci (N: int): int;
if N<0 then
    error('No válido')
case N of
    0, 1: return N
else
    fnm2:= 0
    fnm1:= 1
    for i:= 2 to N
        fn:= fnm1 + fnm2
        fnm2:= fnm1
        fnm1:= fn
    end
    return fn
end
```

# Introducción.

- **Ejemplos.** Estudiar  $t(n)$ .

```
for i:= 1 to N do
    if Impar(i) then
        for j:= i to n do
            x:= x + 1
    else
        for j:= 1 to i do
            y:= y + 1
    end
end
end
```

# Funciones recursivas

---

- Las funciones de tiempo asociadas son también recursivas.
- Ejemplo:

$$t(n) = t(n-1) + f(n)$$

# Ecuaciones de recurrencia.

- Es normal que un algoritmo se base en procedimientos auxiliares, haga llamadas recursivas para tamaños menores o reduzca el tamaño del problema progresivamente.
- En el análisis, el tiempo  $t(n)$  se expresa en función del tiempo para  $t(n-1)$ ,  $t(n-2)$ ... → **Ecuaciones de recurrencia**.
- **Ejemplo.** ¿Cuántas operaciones **mover** se ejecutan?

**Hanoi (n, i, j, k)**

**if**  $n > 0$  **then**

    Hanoi ( $n-1$ , i, k, j)

    mover (i, j)

    Hanoi ( $n-1$ , k, j, i)

**else**

    mover (i, j)

## Expansión de recurrencias

- Aplicar varias veces la fórmula recurrente hasta encontrar alguna “regularidad”.
- **Ejemplo.** Calcular el número de **mover**, para el problema de las torres de Hanoi.

$$t(0) = 1$$

$$t(n) = 2 t(n-1) + 1.$$

- Expansión de la ecuación recurrente:

$$\begin{aligned} t(n) &= 2 t(n-1) + 1 = 2^2 t(n-2) + 2 + 1 = 2^3 t(n-3) + 4 + 2 + 1 = \\ &= \dots \dots \dots = 2^n t(n-n) + \sum_{i=0}^{n-1} 2^i = \sum_{i=0}^n 2^i = 2^{n+1} - 1 \end{aligned}$$

# Solución de ecuaciones recursivas

## Resolución de recurrencias asintóticas

---

- Ecuación característica
- Recurrencias homogéneas
- Recurrencias no homogéneas
- Cambios de variable
- Transformación de rangos

# Funciones recursivas (I)

---

- Para analizar el tiempo de ejecución de un procedimiento recursivo, le asociamos una función de eficiencia desconocida,  $T(n)$ , y la estimamos a partir de  $T(k)$  para distintos valores de  $k$ .

# Función factorial

---

```
1: int fact(int n) {  
2:   if (n <= 1)  
3:     return 1;  
4:   else  
5:     return (n * fact(n - 1));  
6: }
```

Llamamos  $T(n)$  al tiempo de ejecución de  $\text{fact}(n)$ .

Las líneas 2 y 3 son operaciones elementales, sean  $c$  y  $d$  sus tiempos de ejecución

# Función factorial (II)

---

$$\begin{aligned} T(n) &= c + T(n - 1) \\ &= c + (c + T(n-2)) = 2c + T(n-2) \\ &= 2c + (c + T(n-3)) = 3c + T(n-3) \\ &\dots \\ &= ic + T(n-i) \\ &\dots \\ &= (n-1)c + T(n - (n-1)) = (n-1)c + c + d \\ &= nc + d \end{aligned}$$

De donde  $T(n)$  es  $O(n)$

# Ejemplo

---

```
1:     int E(int n) {  
2:         if (n == 1)  
3:             return 0;  
4:         else  
5:             return E(n/2) + 1;  
6:     }
```

$$T(n) = \begin{cases} 1, & n = 1 \\ 1 + T\left(\frac{n}{2}\right) & n > 1 \end{cases}$$

$T(n)$  es  $O(\log_2(n))$

# Ejemplo de recurrencia

---

$$T(n) = T\left(\frac{n}{2}\right) + n^2, \quad n \geq 2, \quad T(1) = 1$$

- Cambio de variable:  $n=2^m$ ;  $n^2 = (2^m)^2 = (2^2)^m = 4^m$

$$\begin{aligned} T(2^m) &= T(2^{m-1}) + 4^m = \\ &= T(2^{m-2}) + 4^{m-1} + 4^m \\ &\dots \\ &= T(2^{m-i}) + [4^{m-(i-1)} + \dots + 4^{m-1} + 4^m] \end{aligned}$$

# Ejemplo

---

$$\begin{aligned}T(2^m) &= T(1) + [4^1 + \cdots + 4^{m-2} + 4^{m-1} + 4^m] \\&= \sum_{i=0}^m 4^i = \frac{4^{m+1} - 1}{4 - 1} = \frac{4}{3}4^m - \frac{1}{3} \\&= [4^m = n^2] = \frac{4}{3}n^2 - \frac{1}{3}\end{aligned}$$

$T(n)$  es  $O(n^2)$

# Resolución de recurrencias

---

- Desarrollar; intentar encontrar la expresión general; resolver.
- Método de la ecuación característica

# Resolución de recurrencias

- En general, las ecuaciones de recurrencia tienen la forma:

$$t(n) = b \quad \text{Para } 0 \leq n \leq n_0 \quad \textbf{Caso base}$$

$$t(n) = f(t(n), t(n-1), \dots, t(n-k), n) \quad \text{En otro caso}$$

- Tipos de ecuaciones de recurrencia:**

- Lineales y homogéneas:

$$a_0 t(n) + a_1 t(n-1) + \dots + a_k t(n-k) = 0$$

- Lineales y no homogéneas:

$$a_0 t(n) + a_1 t(n-1) + \dots + a_k t(n-k) = p(n) + \dots$$

- No lineales:

$$\text{Ejemplo: } a_0 t^2(n) + t(n-1)*t(n-k) + \sqrt{t(n-2) + 1} = p(n)$$

# Recurrencias homogéneas

---

- Consideramos recurrencias *homogéneas lineales con coeficientes constantes*:

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0$$

- Lineales: no hay términos  $t_{n-1}t_{n-j}, t^2_{n-i}$
- Homogénea: igualada a 0
- Con coeficientes constantes:  $a_i$  son ctes
- Ejemplo (sucesión de Fibonacci)

$$f_n = f_{n-1} + f_{n-2} \Rightarrow f_n - f_{n-1} - f_{n-2} = 0$$

- *Observación:* las combinaciones lineales de las soluciones de la recurrencia también son soluciones

# Ecuaciones lineales homogéneas.

- La ecuación de recurrencia es de la forma:  
$$a_0 t(n) + a_1 t(n-1) + \dots + a_k t(n-k) = 0; \quad a_i \text{ constante}$$

- **Caso sencillo:**

$$t(n) = \begin{cases} 1 & \text{Si } n = 0 \\ x \cdot t(n-1) & \text{Si } n > 0 \end{cases}$$

- **Solución:**  $t(n) = x^n$

## Ecuaciones lineales homogéneas.

- Suponiendo que las soluciones son de la forma  $t(n) = x^n$ , la ecuación de recurrencia homogénea:

$$a_0 t(n) + a_1 t(n-1) + \dots + a_k t(n-k) = 0$$

- Se transforma en:

$$\begin{aligned} a_0 x^n + a_1 x^{n-1} + \dots + a_k x^{n-k} &= 0 \Rightarrow /x^{n-k} \Rightarrow \\ a_0 x^k + a_1 x^{k-1} + \dots + a_k &= 0 \end{aligned}$$

**Ecuación característica de la ecuación recurrente  
lineal homogénea**

# Ecuaciones lineales homogéneas.

$$a_0x^k + a_1x^{k-1} + \dots + a_k = 0$$

## Ecuación característica de la ecuación recurrente lineal homogénea

- $k$ : conocida.  $a_i$ : conocidas.  $x$ : desconocida.
- Resolver el sistema para la incógnita  $x$ . El resultado es:

$$t(n) = x^n$$

- Pero... Un polinomio de grado  $k$  tendrá  $k$  soluciones...

## Ecuaciones lineales homogéneas.

- Sean las soluciones  $x = (s_1, s_2, \dots, s_k)$ , todas distintas.
- La solución será:

$$t(n) = c_1 \cdot s_1^n + c_2 \cdot s_2^n + \dots + c_k \cdot s_k^n = \sum_{i=1}^k c_i \cdot s_i^n$$

- Siendo  $c_i$  constantes, cuyos valores dependen de los casos base (condiciones iniciales).
- Son constantes que añadimos nosotros. Debemos resolverlas, usando los casos base de la ecuación recurrente.

# En resumen:

---

Suposición  $t_n = x^n$

$$a_0x^n + a_1x^{n-1} + \cdots + a_kx^{n-k} = 0$$

satisfecha si  $x=0$  o (ecuación característica)

$$a_0x^k + a_1x^{k-1} + \cdots + a_k = 0$$

## Polinomio característico

$$p(x) = a_0x^k + a_1x^{k-1} + \cdots + a_k$$

# En resumen:

---

(Teorema fundamental del Álgebra)

$$p(x) = \prod_{i=1}^k (x - r_i)$$

Consideremos una raíz del polinomio característico,  $r_i$ ,  $p(r_i) = 0$ ,  $r_i^n$  es solución de la recurrencia.

Además,

$$t_n = \sum_{i=1}^k c_i r_i^n$$

Cuando todos los  $r_i$  son distintos, éstas son las únicas soluciones

# Ejemplo Fibonacci

---

$$f_n = \begin{cases} n, & \text{si } n = 0 \text{ ó } n = 1 \\ f_{n-1} + f_{n-2} & \text{en otro caso} \end{cases}$$

$$f_n - f_{n-1} - f_{n-2} = 0$$

$$p(x) = x^2 - x - 1$$

$$r_1 = \frac{1 + \sqrt{5}}{2} \quad y \quad r_2 = \frac{1 - \sqrt{5}}{2}$$

# Solución Fibonacci

---

Solución general:

$$f_n = c_1 r_1^n + c_2 r_2^n$$

$$\begin{array}{rcl} c_1 & + & c_2 = 0 \\ r_1 c_1 & + & r_2 c_2 = 1 \end{array} \quad c_1 = \frac{1}{\sqrt{5}} \quad y \quad c_2 = -\frac{1}{\sqrt{5}}$$

$$f_n = \frac{1}{\sqrt{5}} \left[ \left( \frac{1+\sqrt{5}}{2} \right)^n - \left( \frac{1-\sqrt{5}}{2} \right)^n \right]$$

# Ejemplo

---

$$t_n = \begin{cases} 0, & n = 0 \\ 5, & n = 1 \\ 3t_{n-1} + 4t_{n-2}, & \text{en otro caso} \end{cases}$$

$$t_n - 3t_{n-1} - 4t_{n-2} = 0$$

$$x^2 - 3x - 4 = (x + 1)(x - 4)$$

A partir de las condiciones iniciales:  $c_1 = -1$  y  $c_2 = 1$

$$t_n = c_1(-1)^n + c_2 4^n$$

## Ecuaciones lineales homogéneas.

- Si no todas las soluciones  $x = (s_1, s_2, \dots, s_k)$  son distintas, entonces el polinomio característico será:  
$$a_0x^n + a_1x^{n-1} + \dots + a_kx^{n-k} = (x - s_1)^m \cdot (x - s_2) \cdot \dots \cdot (x - s_p) \cdot x^{n-k}$$
- ¿Cuál es la solución para  $t(n)$ ?
- Las derivadas valen 0 en  $s_1$ , hasta la  $m-1$ -ésima.

$$\begin{aligned} a_0 n \cdot x^{n-1} + a_1 (n-1) \cdot x^{n-2} + \dots + a_k (n-k) \cdot x^{n-k-1} &= 0 \Rightarrow \cdot x \Rightarrow \\ a_0 n \cdot x^n + a_1 (n-1) \cdot x^{n-1} + \dots + a_k (n-k) x^{n-k} &= 0 \end{aligned}$$

## Ecuaciones lineales homogéneas.

- Las derivadas valen 0 en  $s_1$ , hasta la  $m-1$ -ésima.
- **Conclusión:**  $t(n) = n \cdot s_1^n$  también será solución de la ecuación característica.
- Para la segunda derivada:  $t(n) = n^2 s_1^n$  será solución...
- Si  $s_i$  tiene multiplicidad  $m$ , entonces tendremos:  
$$s_i^n \quad n \cdot s_i^n \quad n^2 \cdot s_i^n \quad \dots \quad n^{m-1} \cdot s_i^n$$
- Dadas las soluciones  $x = (s_1, s_2, \dots, s_k)$  siendo  $s_k$  de multiplicidad  $m$ , la solución será:

$$\begin{aligned} t(n) &= c_1 \cdot s_1^n + c_2 \cdot s_2^n + \dots + c_k \cdot s_k^n + c_{k+1} \cdot n \cdot s_k^n + \\ &+ c_{k+2} \cdot n^2 \cdot s_k^n + \dots + c_{k+1+m} \cdot n^{m-1} \cdot s_k^n \end{aligned}$$

## En resumen: Si hay raíces múltiples....

---

Si las raíces del polinomio característico NO son todas distintas.  
Sean  $r_i$  con multiplicidad  $m_i$ ,  $i=1, \dots, l$ , las soluciones de  $p(x)$ .  
Entonces

$$t_n = \sum_{i=1}^l \sum_{j=0}^{m_i-1} c_{ij} n^j r_i^n$$

# Ejemplo

---

$$t_n = \begin{cases} n, & \text{si } n = 0, 1 \text{ ó } 2 \\ 5t_{n-1} - 8t_{n-2} + 4t_{n-3} & \text{en otro caso} \end{cases}$$

$$t_n - 5t_{n-1} + 8t_{n-2} - 4t_{n-3} = 0$$

$$x^3 - 5x^2 + 8x - 4 = (x-1)(x-2)^2$$

$$t_n = c_1(1)^n + c_2 2^n + c_3 n 2^n$$

A partir de las condiciones iniciales:

$$c1 = -2, c2 = 2, c3 = -1/2$$

$$t_n = 2^{n+1} - n 2^{n-1} - 2$$

# Recurrencias no homogéneas

---

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = b^n p(n)$$

- $b$  es una constante
- $p(n)$  es un polinomio de  $n$  de grado  $d$
  
- Ejemplo:  $t_n - 2t_{n-1} = 3^n$
  
- El polinomio característico es:

$$p(x) = (a_0 x^k + a_1 x^{k-1} + \cdots + a_k)(x - b)^{d+1}$$

- Se procede igual que en el caso homogéneo, salvo que las condiciones iniciales se obtienen de la propia recurrencia

# Ejemplo

---

$$t_n = \begin{cases} 0, & \text{si } n = 0 \\ 2t_{n-1} + 1 & \text{en otro caso} \end{cases}$$

$$t_n - 2t_{n-1} = 1 \quad p(x) = (x-2)(x-1)$$

Solución general :  $t_n = c_1 2^n + c_2$

$$\begin{array}{rcl} c_1 + c_2 & = & 0 \\ 2c_1 + c_2 & = & 1 \end{array} \Rightarrow \begin{array}{l} c_1 = 1 \\ c_2 = -1 \end{array}$$

$$t_n = 2^n - 1$$

# Ejemplo

---

$$t_n = \begin{cases} 0, & \text{si } n = 0 \\ 2t_{n-1} + n & \text{en otro caso} \end{cases} \quad p(x) = (x - 2)(x - 1)^2$$

Solución general :  $t_n = c_1 2^n + c_2 1^n + c_3 n 1^n$

$$c_1 + c_2 = 0 \quad c_1 = 2$$

$$2c_1 + c_2 + c_3 = 1 \Rightarrow c_2 = -2$$

$$4c_1 + c_2 + 2c_3 = 4 \quad c_3 = -1$$

$$t_n = 2^{n+1} - n - 2$$

# Cambio de variable

---

- **Calcular el orden de  $T(n)$  si  $n$  es potencia de 2, y**

$$T(n) = 4T(n/2) + n, n > 1$$

- Reemplazamos  $n$  por  $2^k$  (de modo que  $k = \lg n$ ) para obtener  $T(2^k) = 4T(2^{k-1}) + 2^k$ . Esto puede escribirse,

$$t_k = 4t_{k-1} + 2^k$$

- si  $t_k = T(2^k) = T(n)$ .
- La ecuación característica es  $(x-4)(x-2) = 0$   
y entonces  $t_k = c_1 4^k + c_2 2^k$ .
- Poniendo  $n$  en lugar de  $k$ , tenemos  $T(n) = c_1 n^2 + c_2 n$
- y  $T(n)$  es por tanto  $O(n^2)$  tq.  $n$  es una potencia de 2 y suponiendo que  $c_1$  es positivo)

# Cambio de variable

---

- Encontrar el orden de  $T(n)$  si  $n$  es una potencia de 2 y si

$$T(n) = 4T(n/2) + n^2, \quad n > 1$$

- Obtenemos sucesivamente

$$T(2^k) = 4T(2^{k-1}) + 4^k, \text{ y}$$

$$t_k = 4t_{k-1} + 4^k$$

- Ecuación característica:  $(x-4)^2 = 0$ , y así

$$t_k = c_1 4^k + c_2 k 4^k, \quad T(n) = c_1 n^2 + c_2 n^2 \lg n$$

- y  $T(n)$  es  $O(n^2 \log n)$  tq.  $n$  es potencia de 2).

# Cambio de variable

---

- Calcular el orden de  $T(n)$  si  $n$  es una potencia de 2

$$T(n) = 2T(n/2) + n \lg n, n > 1$$

- Obtenemos

$$T(2^k) = 2T(2^{k-1}) + k2^k$$

$$t_k = 2t_{k-1} + k2^k$$

- La ecuación característica es  $(x-2)^3 = 0$ , y así,

$$t_k = c_1 2^k + c_2 k2^k + c_3 k^2 2^k$$

$$T(n) = c_1 n + c_2 n \lg n + c_3 n \lg^2 n$$

- Así  $T(n)$  es  $O(n \log^2 n)$  tq.  $n$  es potencia de 2).

# Cambio de variable

---

- Calcular el orden de  $T(n)$  si  $n$  es potencia de 2 y  $T(n) = 3T(n/2) + cn$  ( $c$  es constante,  $n \geq 1$ ).
- Obtenemos sucesivamente,

$$\begin{aligned} T(2^k) &= 3T(2^{k-1}) + c2^k \\ t_k &= 3t_{k-1} + c2^k \end{aligned}$$

- Ecuación característica:  $(x-3)(x-2) = 0$ , y así,

$$\begin{aligned} t_k &= c_1 3^k + c_2 2^k \\ T(n) &= c_1 3^{\lg n} + c_2 n \end{aligned}$$

- y como  $a^{\lg b} = b^{\lg a}$ ,  $T(n) = c_1 n^{\lg 3} + c_2 n$   
y finalmente  $T(n)$  es  $O(n^{\lg 3})$  tq.  $n$  es potencia de 2).

## Transformaciones del rango

- Se utiliza en algunos casos, donde las ecuaciones recurrentes son no lineales. **Ejemplo.**
$$t(1) = 6; \quad t(n) = n t^2(n/2)$$
- Suponiendo  $n$  potencia de 2, hacemos el cambio  $n=2^k$ :
$$t(2^0) = 6; \quad t(2^k) = 2^k t^2(2^{k-1})$$
- Tomando logaritmos (en base 2):
$$\log t(2^0) = \log 6; \quad \log t(2^k) = k + 2 \cdot \log t(2^{k-1})$$
- Se hace una transformación de la imagen:
$$v(x) = \log t(2^x) \Rightarrow$$
$$v(0) = \log 6; \quad v(k) = k + 2 \cdot v(k-1)$$

## Transformaciones del rango

- Resolver la ecuación recurrente:

$$v(0) = \log 6; \quad v(k) = k + 2 \cdot v(k-1)$$

- Resultado:

$$v(k) = c_1 \cdot 2^k + c_2 + c_3 \cdot k \Rightarrow v(k) = (3 + \log 3) \cdot 2^k - k - 2$$

- Ahora deshacer el cambio  $v(x) = \log t(2^x)$ :

$$\log t(2^k) = \log t(n) = (3 + \log 3) \cdot 2^k - k - 2$$

- Y quitar los logaritmos, elevando a 2:

$$\begin{aligned} t(n) &= 2^{(3+\log 3)n - \log n - 2} = 2^{3n} \cdot 2^{\log 3 \cdot n} \cdot 2^{-\log n} \cdot 2^{-2} = \\ &= (2^{3n-2} \cdot 3^n)/n = 24^n/4n \end{aligned}$$

- Quitar la condición de que  $n$  sea potencia de 2.

# Transformaciones del rango

---

- $T(n) = nT^2(n/2)$ ,  $n > 1$  ,  $T(1) = 6$  y  $n$  potencia de 2.

- Cambiamos la variable:  $t_k = T(2^k)$ , y asi

$$t_k = 2^k t_{k-1}^2, k > 0; t_0 = 6.$$

- Esta recurrencia no es lineal, y uno de los coeficientes no es constante.

- Para transformar el rango, creamos una nueva recurrencia tomando  $V_k = \lg t_k$  , lo que da,

$$V_k = k + 2 V_{k-1}, k > 0; V_0 = \lg 6.$$

- Ecuación característica:  $(x-2)(x-1)^2 = 0$  y asi,

$$V_k = c_1 2^k + c_2 1^k + c_3 k 1^k$$

# Algo más sobre las condiciones iniciales

- ¿Cuál es el significado de las condiciones iniciales?
- **Condición inicial:** caso base de una ecuación recurrente.
- ¿Cuántas aplicar?
  - Tantas como constantes indeterminadas.
  - $n$  incógnitas,  $n$  ecuaciones: sistema determinado.  
Aplicamos el método de Cramer.
- ¿Cuáles aplicar?
  - Las condiciones aplicadas se deben poder alcanzar desde el caso general.
  - Si se ha aplicado un cambio de variable, deben cumplir las condiciones del cambio.

# Algo más sobre las condiciones iniciales

- **Ejemplo.**

$$t(n) = n \quad \text{Si } n \leq 10$$

$$t(n) = 5 \cdot t(n-1) - 8 \cdot t(n-2) + 4 \cdot t(n-3) \quad \text{Si } n > 10$$

- Resultado:  $t(n) = c_1 + c_2 2^n + c_3 n \cdot 2^n$

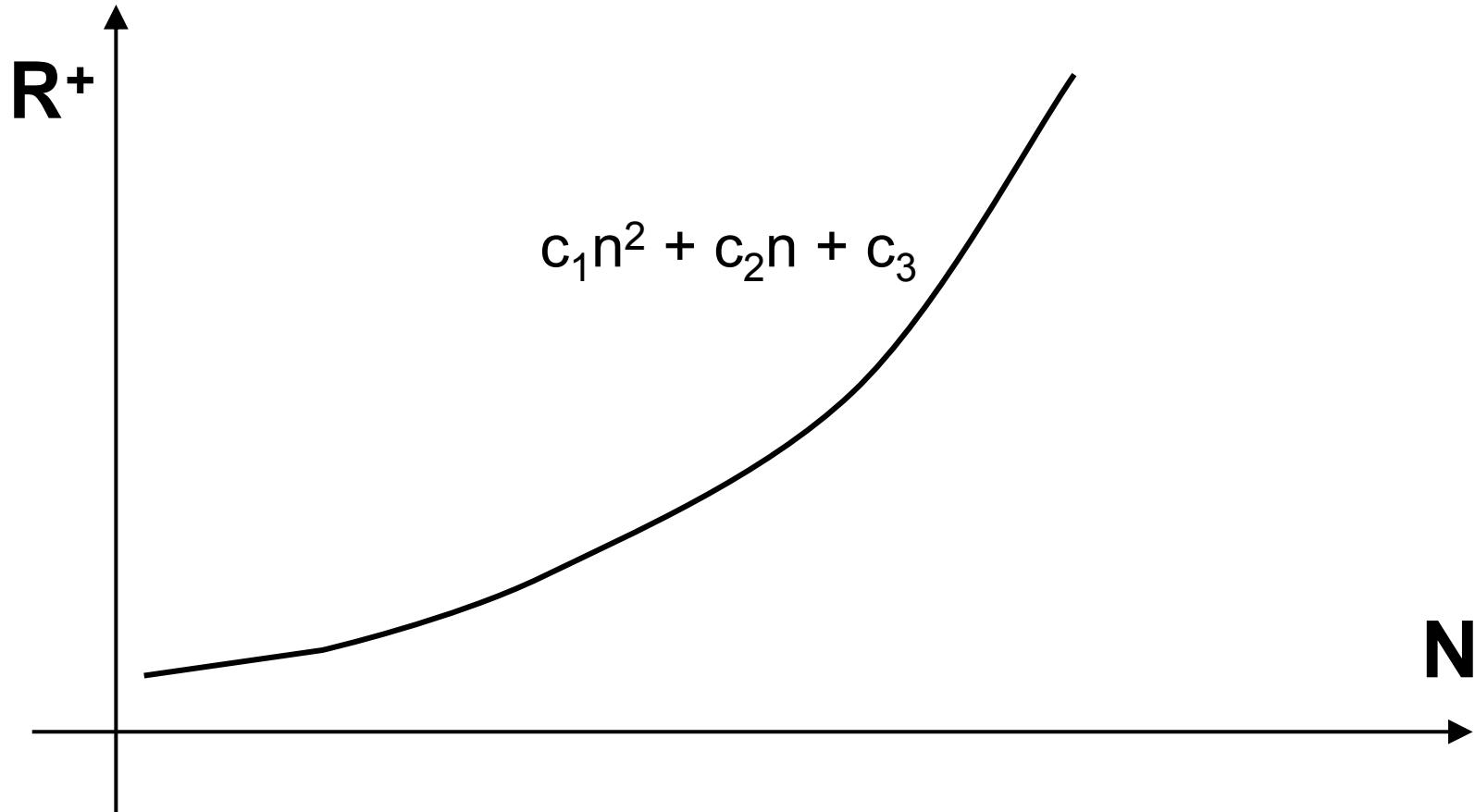
- Aplicar las condiciones iniciales para despejar  $c_1$ ,  $c_2$ ,  $c_3$ .

# Algo más sobre las condiciones iniciales

- El cálculo de constantes también se puede aplicar en el estudio experimental de algoritmos.
- **Proceso**
  1. Hacer una estimación teórica del tiempo de ejecución.
  2. Expresar el tiempo en función de constantes indefinidas.
  3. Tomar medidas del tiempo de ejecución para distintos tamaños de entrada.
  4. Resolver las constantes.

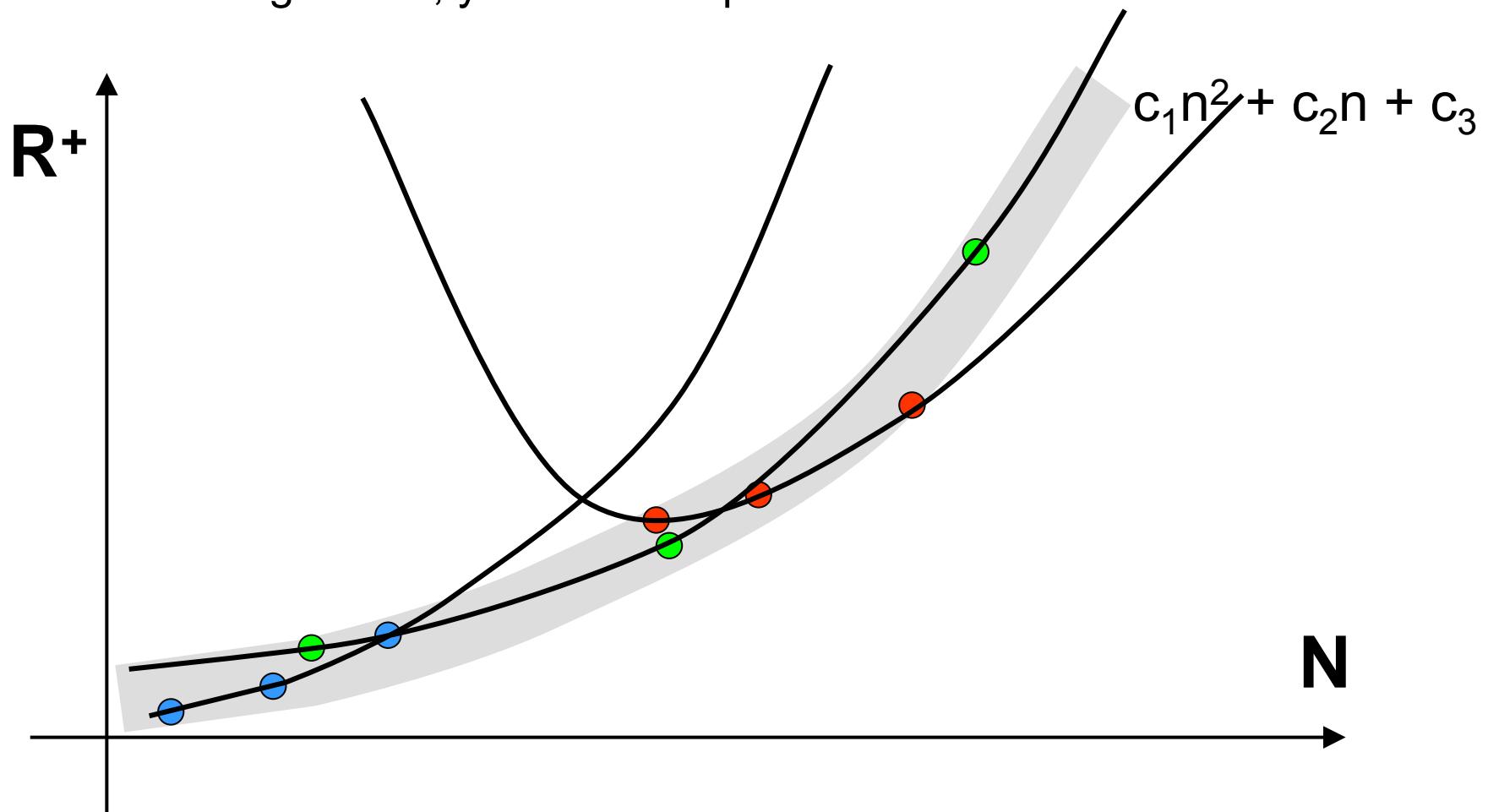
## Algo más sobre las condiciones iniciales

- **Ejemplo:**  $t(n) = a(n+1)^2 + (b+c)n + d$
- Simplificamos constantes:  $t(n) = c_1n^2 + c_2n + c_3$



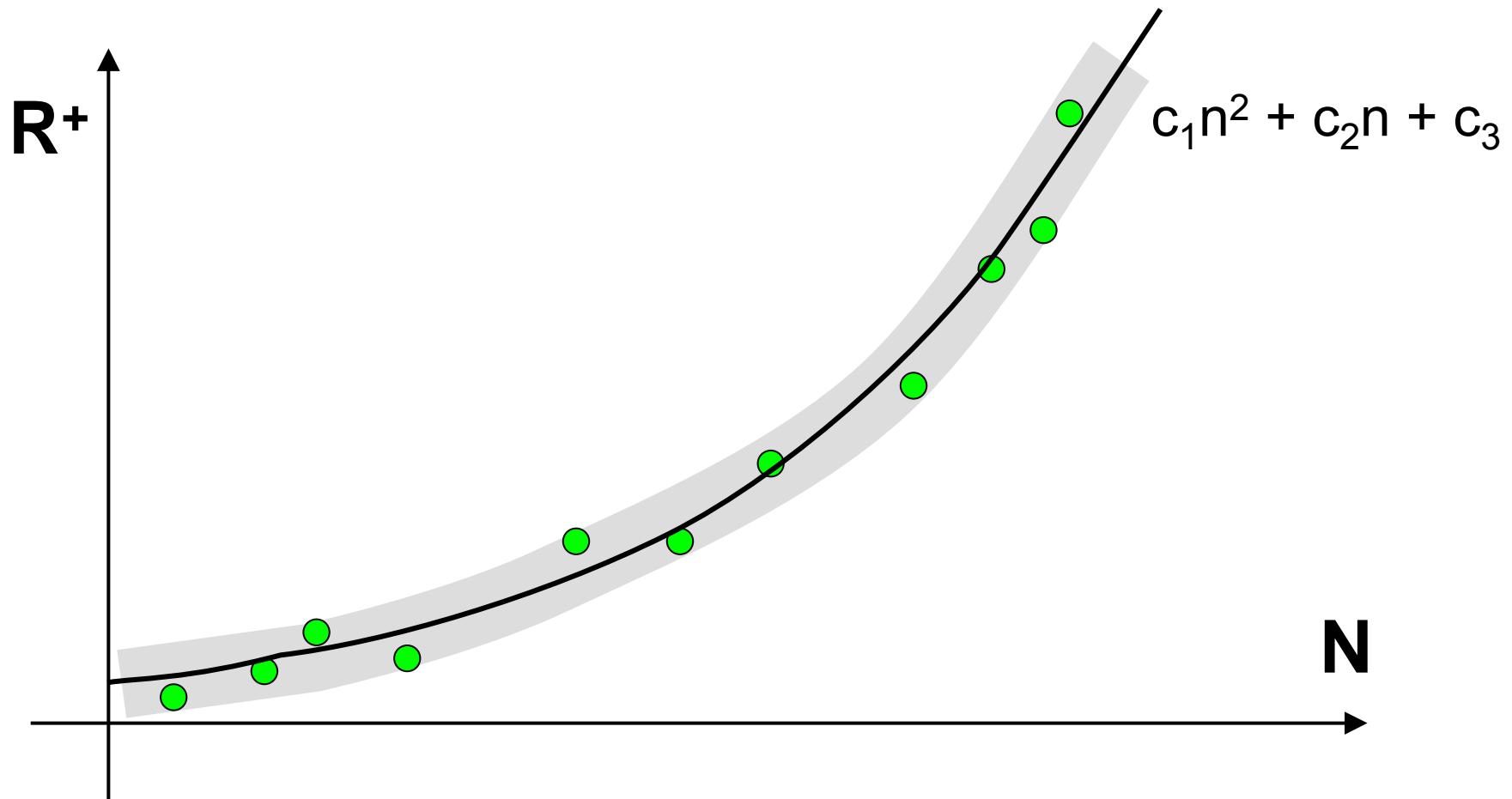
# Algo más sobre las condiciones iniciales

- **Ajuste sencillo:** Tomar 3 medidas de tiempo.  
3 incógnitas y 3 ecuaciones: resolvemos  $c_1, c_2, c_3$ .
- Tamaños grandes, y medidas separadas.



# Algo más sobre las condiciones iniciales

- **Ajuste preciso:** Tomar muchas medidas de tiempo.
- Hacer un ajuste de regresión.



# Análisis de algoritmos.

## Conclusiones:

- **Eficiencia:** consumo de recursos en función de los resultados obtenidos.
- **Recursos consumidos** por un algoritmo: fundamentalmente tiempo de ejecución y memoria.
- La estimación del tiempo,  $t(n)$ , es aproximada, parametrizada según el tamaño y el caso ( $t_m$ ,  $t_M$ ,  $t_p$ ).
- **Conteo de instrucciones:** obtenemos como resultado la función  $t(n)$
- Para simplificar se usan las notaciones asintóticas:  $O(t)$ ,  $\Omega(t)$ ,  $\Theta(t)$ ,  $o(t)$ .

# Análisis de algoritmos.

## Conclusiones:

- **Ecuaciones recurrentes:** surgen normalmente del conteo (de tiempo o memoria) de algoritmos recursivos.
- Tipos de ecuaciones recurrentes:
  - Lineales, homogéneas o no homogéneas.
  - No lineales (menos comunes en el análisis de algoritmos).
- **Resolución de ecuaciones recurrentes:**
  - Método de expansión de recurrencias (el más sencillo).
  - Método de la ecuación característica (lineales).
  - Cambio de variable (previo a la ec. característica) o transformación de la imagen.
  - Inducción constructiva (general pero difícil de aplicar).

# Algorítmica

---

**Tema 1. Planteamiento General**

**Tema 2. La Eficiencia de los Algoritmos**

**Tema 3. Algoritmos “Divide y vencerás”**

**Tema 4. Algoritmos Voraces (“Greedy”)**

**Tema 5. Algoritmos basados en Programación Dinámica**

**Tema 6. Algoritmos para la Exploración de Grafos  
 (“Backtracking”, “Branch and Bound”)**

**Tema 7. Otras metodologías algorítmicas**

# Algorítmica

---

**Tema 1. Planteamiento General**

**Tema 2. La Eficiencia de los Algoritmos**

**Tema 3. Algoritmos “Divide y vencerás”**

**Tema 4. Algoritmos Voraces (“Greedy”)**

**Tema 5. Algoritmos basados en Programación Dinámica**

**Tema 6. Algoritmos para la Exploración de Grafos**

**(“Backtracking”, “Branch and Bound”)**

**Tema 7. Otras metodologías algorítmicas**

# Tema 3: Algoritmos Divide y Venceras

---

**Bibliografía:**

**G. BRASSARD, P. BRATLEY. Fundamentos de Algoritmia. Prentice Hall (1997).**

# Objetivos

---

- Comprender el principio de Divide y Vencerás
- Conocer las características de un problema resoluble con DV
- Saber calcular el umbral
- Conocer los principales algoritmos de ordenación
- Resolución de diversos subproblemas

# Índice

---

## **EL ENFOQUE DIVIDE Y VENCERÁS**

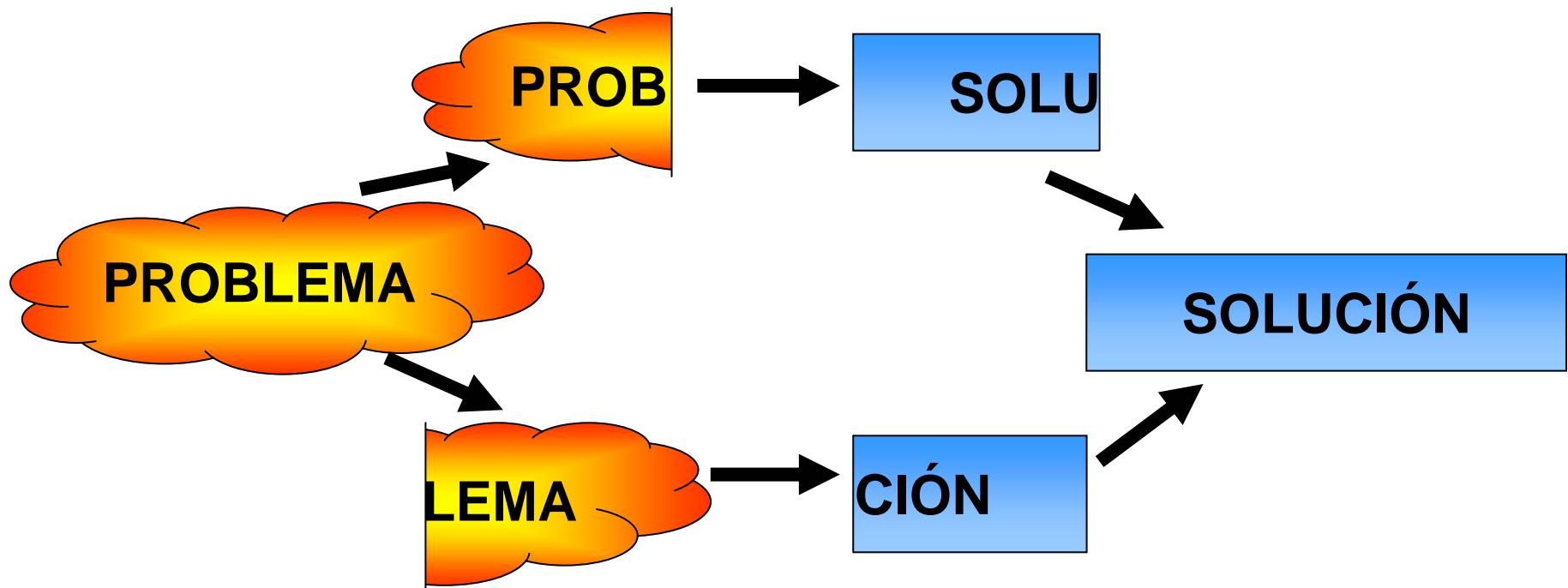
- 1. Enfoque Divide y Vencerás para el Diseño de Algoritmos**
  - 1.1. Introducción**
  - 1.2. Ejemplo: Multiplicación de Enteros Muy Grandes**
- 2. Método General DV**
  - 2.1. Procedimiento General**
  - 2.2. Condiciones para que DV sea ventajoso**
  - 2.3. Análisis del Orden de los Algoritmos DV**
- 3. La Determinación del Umbral**

## **APLICACIONES DE LA TÉCNICA DIVIDE Y VENCERÁS**

- **Algoritmos de Ordenación**
- **Multiplicación de Matrices**

# Intuitivamente...

- La técnica **divide y vencerás** consiste en:
  - Descomponer un problema en un conjunto de subproblemas más pequeños.
  - Se resuelven estos subproblemas.
  - Se combinan las soluciones para obtener la solución para el problema original.



# Introducción

- **Esquema general:**

**DivideVencerás (p: problema)**

*Dividir (p, p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>k</sub>)*

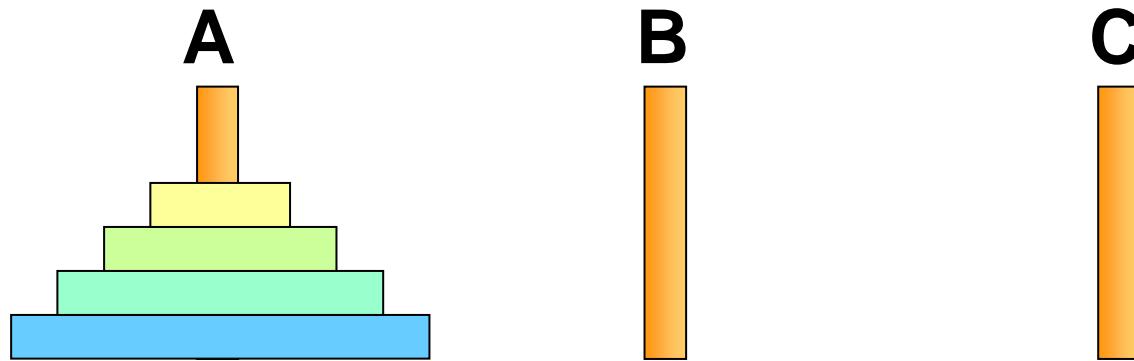
**para** i := 1, 2, ..., k

*s<sub>i</sub> := Resolver (p<sub>i</sub>)*

*solución := Combinar (s<sub>1</sub>, s<sub>2</sub>, ..., s<sub>k</sub>)*

- Normalmente para resolver los subproblemas se utilizan llamadas recursivas al mismo algoritmo (aunque no necesariamente).
- **Ejemplo.** Problema de las Torres de Hanoi.

# Introducción



- **Ejemplo.** Problema de las torres de Hanoi.  
Mover  $n$  discos del poste A al C:
  - Mover  $n-1$  discos de A a B
  - Mover 1 disco de A a C
  - Mover  $n-1$  discos de B a C

# Introducción

**Hanoi (n, A, B, C: entero)**

**si** n==1 **entonces**

        mover (A, C)

**sino**

        Hanoi (n-1, A, C, B)

        mover (A, C)

        Hanoi (n-1, B, A, C)

**finsi**

- Si el problema es “pequeño”, entonces se puede resolver de forma directa.
- **Otro ejemplo.** Cálculo de los números de Fibonacci:  
 $F(n) = F(n-1) + F(n-2)$
- $F(0) = F(1) = 1$

# Introducción

- **Ejemplo.** Cálculo de los números de Fibonacci.
  - El cálculo del  $n$ -ésimo número de Fibonacci se descompone en calcular los números de Fibonacci  $n-1$  y  $n-2$ .
  - *Combinar:* sumar los resultados de los subproblemas.
- La idea de la técnica divide y vencerás es aplicada en muchos campos:
  - Estrategias militares.
  - Demostraciones lógicas y matemáticas.
  - Diseño modular de programas.
  - Diseño de circuitos.
  - Etc.

# Introducción

- **Esquema recursivo.** Con división en 2 subproblemas y datos almacenados en una tabla entre las posiciones p y q:

**DivideVencerás (p, q: índice)**

**var** m: índice

**si** Pequeño (p, q) **entonces**

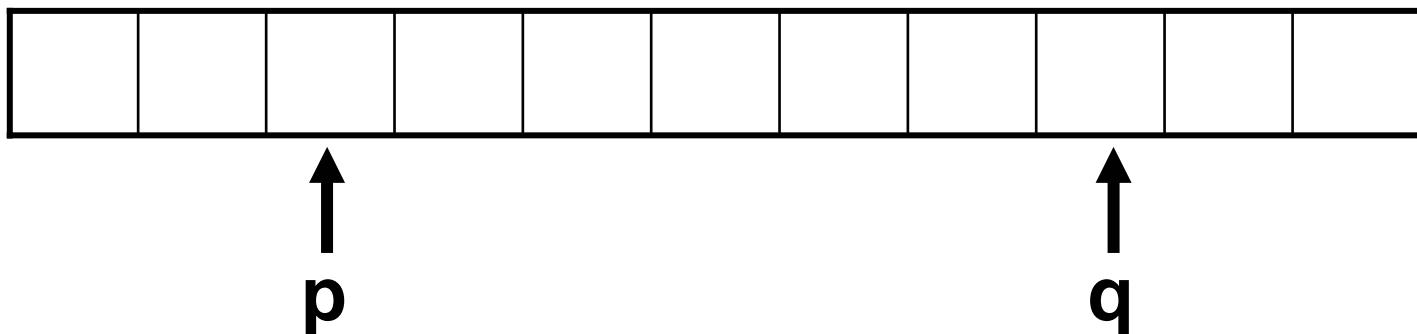
solucion:= SoluciónDirecta (p, q)

**sino**

m:= Dividir (p, q)

solucion:= Combinar (DivideVencerás (p, m),  
DivideVencerás (m+1, q))

**finsi**



# Introducción

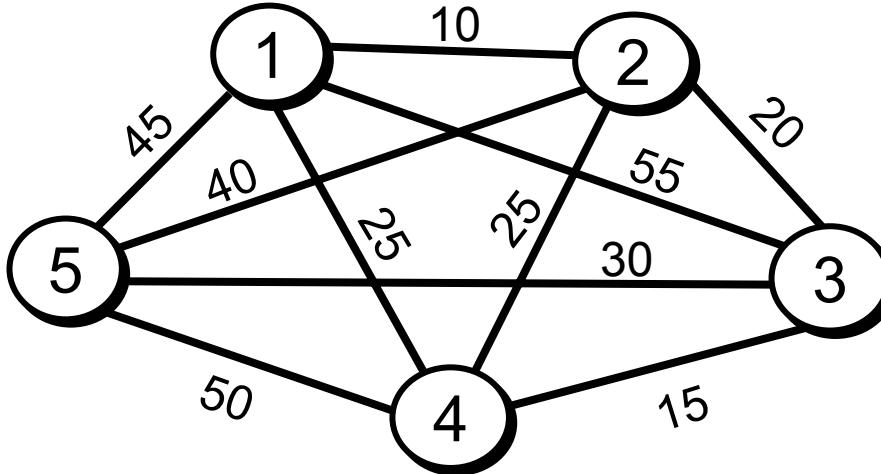
- Aplicación de divide y vencerás: encontrar la forma de definir las **funciones genéricas**:
  - *Pequeño*: Determina cuándo el problema es pequeño para aplicar la resolución directa.
  - *Solución Directa*: Método alternativo de resolución para tamaños pequeños.
  - *Dividir*: Función para descomponer un problema grande en subproblemas.
  - *Combinar*: Método para obtener la solución al problema original, a partir de las soluciones de los subproblemas.
- Para que pueda aplicarse la técnica divide y vencerás debe existir una forma de definirlas → Aplicar un razonamiento inductivo...

# Introducción

- Requisitos para aplicar divide y vencerás:
  - Necesitamos un **método** (más o menos **directo**) de resolver los problemas de tamaño pequeño.
  - El problema original debe poder dividirse fácilmente en un conjunto de subproblemas, del **mismo tipo** que el problema original pero con una resolución **más sencilla** (menos costosa).
  - Los subproblemas deben ser **disjuntos**: la solución de un subproblema debe obtenerse independientemente de los otros.
  - Es necesario tener un método de **combinar** los resultados de los subproblemas.

# Introducción

- **Ejemplo.**  
Problema  
del viajante.



- Método directo de resolver el problema:  
Trivial con 3 nodos.
- Descomponer el problema en subproblemas más pequeños:  
¿Por dónde?
- Los subproblemas deben ser disjuntos:  
...parece que no
- Combinar los resultados de los subproblemas:  
¡¡Imposible aplicar divide y vencerás!!

# Introducción

- Normalmente los subproblemas deben ser de tamaños parecidos.
- Como mínimo necesitamos que hayan dos subproblemas.
- Si sólo tenemos un subproblema entonces hablamos de técnicas de **reducción** (o **simplificación**).
- **Ejemplo sencillo:** Cálculo del factorial.  
 $\text{Fact}(n) := n * \text{Fact}(n-1)$

# Introducción

- Para el esquema recursivo, con división en dos subproblemas con la mitad de tamaño:

$$t(n) = \begin{cases} g(n) & \text{Si } n \leq n_0 \text{ (caso base)} \\ 2*t(n/2) + f(n) & \text{En otro caso} \end{cases}$$

- t(n)**: tiempo de ejecución del algoritmo DV.
- g(n)**: tiempo de calcular la solución para el caso base, algoritmo directo.
- f(n)**: tiempo de dividir el problema y combinar los resultados.

# Introducción

- **Ejemplo 1.** La resolución directa se puede hacer en un tiempo constante y la división y combinación de resultados también.

$$g(n) = c; \quad f(n) = d$$

$$\Rightarrow t(n) \in \Theta(n)$$

- **Ejemplo 2.** La solución directa se calcula en  $O(n^2)$  y la combinación en  $O(n)$ .

$$g(n) = c \cdot n^2; \quad f(n) = d \cdot n$$

$$\Rightarrow t(n) \in \Theta(n \log n)$$

# Introducción

- En general, si se realizan **a** llamadas recursivas de tamaño **n/b**, y la división y combinación requieren  $f(n) = d \cdot n^p \in O(n^p)$ , entonces:

$$t(n) = a \cdot t(n/b) + d \cdot n^p$$

Suponiendo  $n = b^k \Rightarrow k = \log_b n$

$$t(b^k) = a \cdot t(b^{k-1}) + d \cdot b^{pk}$$

Podemos deducir que:

$$t(n) \in \begin{cases} O(n^{\log_b a}) & \text{Si } a > b^p \\ O(n^p \cdot \log n) & \text{Si } a = b^p \\ O(n^p) & \text{Si } a < b^p \end{cases}$$

Fórmula  
maestra

# Introducción

- **Ejemplo 3.** Dividimos en 2 trozos de tamaño  $n/2$ , con  $f(n) \in O(n)$ :  
 $a = b = 2$   
 $t(n) \in O(n \cdot \log n)$
- **Ejemplo 4.** Realizamos 4 llamadas recursivas con trozos de tamaño  $n/2$ , con  $f(n) \in O(n)$ :  
 $a = 4; b = 2$   
 $t(n) \in O(n^{\log_2 4}) = O(n^2)$

# 1. El Enfoque Divide y Vencerás

---

## **Resumen de la Introducción:**

Por tanto...la técnica Divide y Vencerás (DV) consiste en:

- Descomponer el caso a resolver en un cierto número de subcasos más pequeños del mismo problema.
- Resolver sucesiva e independientemente todos estos subcasos.
- Combinar las soluciones obtenidas para obtener la solución del caso original.

## **Cuestiones:**

**¿Por qué hacer esto?**

**¿Cómo se resuelven los subcasos?**

# Ejemplo: Multiplicación de enteros muy grandes

---

Algoritmo clásico de multiplicación de enteros de  $n$  cifras:  $\Theta(n^2)$

Algoritmo basado en la división de enteros de tamaño  $n$ :

$$981 \qquad \qquad 1234$$

$$0981$$

$$w = 09, x = 81$$

$$1234 \ (n = 4)$$

$$y = 12, z = 34 \ (n = 2)$$

$$981 = 10^2w + x$$

$$1234 = 10^2y + z$$

# Ejemplo: Multiplicación de enteros muy grandes

---

$$981 = 10^2w + x$$

$$1234 = 10^2y + z$$

$$\begin{aligned}981 \times 1234 &= (10^2w + x) \times (10^2y + z) \\&= 10^4wy + 10^2(wz + xy) + xz \\&= 1.080.000 + 127.800 + 2.754 \\&= 1.210.554\end{aligned}$$

Se precisan 4 operaciones productos de tamaño  $n/2$ :  $wy, wz, xy, xz$

# Ejemplo: Multiplicación de enteros muy grandes

---

¿Es posible reducir el número multiplicaciones de tamaño  $n/2$ ?

Consideremos:

$$\begin{aligned} r &= (w + x) \times (y + z) = wy + (wz + xy) + xz \\ &= (09 + 81) \times (12 + 34) = 90 \times 46 = 4140 \\ p &= wy = 09 \times 12 = 108 \\ q &= xz = 81 \times 34 = 2754 \end{aligned}$$

Finalmente:

$$\begin{aligned} 981 \times 1234 &= 10^4p + 10^2(r - p - q) + q = \\ &= 1.080.000 + 127.800 + 2.754 = 1.210.554 \end{aligned}$$

1 multiplicación de tamaño  $n \rightarrow 3$  de tamaño  $n/2$  más operaciones sumas y restas.

Reducción de 4 multiplicaciones de tamaño  $n/2$  a 3 multiplicaciones de tamaño  $n/2$ : **25%**

# Ejemplo: Multiplicación de enteros muy grandes

---

Ecuaciones de tiempo:

Algoritmo básico (AB):  $h(n) = cn^2$

Consideremos  $g(n)$  operaciones en el algoritmo DV excepto las 3 multiplicaciones de tamaño  $n/2$ .

Ecuación DV con el Algoritmo básico (AB) para tamaño  $n/2$ :

$$3h(n/2) + g(n) = 3c(n/2)^2 + g(n) = \frac{3}{4}cn^2 + g(n) = \frac{3}{4}h(n) + g(n)$$

$h(n) \in \Theta(n^2)$ ,  $g(n) \in \Theta(n) \rightarrow$  ganancia aproximada 25%.

- Ganancia de tiempo
- ¿Cómo resolver los subcasos?

# Ejemplo: Multiplicación de enteros muy grandes

---

¿Qué ocurre si resolvemos los subcasos de forma recursiva?

$$t(n) = 3t(n/2) + g(n), \quad g(n) \in \Theta(n)$$

$$t(n) \in \Theta(n^{\log 3}), \quad t(n) \in \Theta(n^{1.585}),$$

Algunos estudios pendientes:

- ¿Cómo se tratan los números de longitud impar?
- ¿Cómo multiplicamos dos números de diferente tamaño?

Brassard-Bratley, 1997.

## 2. Método General DV

---

### Procedimiento General

Función DV(x)

**si** x es suficientemente pequeño entonces

**devolver** ad hoc(x)

descomponer x en casos más pequeños  $x_1, x_2, \dots, x_l$

**para**  $i=1$  hasta  $l$   $y_i = DV(x_i)$

recombinar los  $y_i$  para obtener una solución y de x

**devolver** y

- l es el número de subcasos
- si l=1 hablamos de reducción
- ad hoc(x) es un algoritmo básico

## 2. Método General DV

---

### Características:

- Subproblemas del mismo tipo que el original.
- Los subproblemas se resuelven independientemente.
- No existe solapamiento entre subproblemas.

## 2. Método General DV

---

### **Condiciones para que DV sea ventajoso:**

- ➔ Selección de cuando utilizar el algoritmo ad hoc, calcular el umbral de recursividad.
- ➔ Poder descomponer el problema en subproblemas y recombinar de forma eficiente a partir de las soluciones parciales.
- ➔ Los subcasos deben tener aproximadamente el mismo tamaño.

## 2. Método General DV

---

### Análisis del Orden de los Algoritmos DV

Para I subcasos con tamaño  $n/b$

$$t(n) = a t(n/b) + g(n)$$

si  $g(n) \in \Theta(n^k)$ , entonces  $t(n)$  es de orden:

$$\Theta(n^k) \text{ si } a < b^k$$

$$\Theta(n^k \log n) \text{ si } a = b^k$$

$$\Theta(n^{\log_b a}) \text{ si } a > b^k$$

# 3. La Determinación del Umbral

---

- Es difícil hablar del umbral  $n_0$  si no tratamos con implementaciones, ya que gracias a ellas conocemos las constantes ocultas que nos permitirán afinar el cálculo de dicho valor.
- El umbral no es único, pero si lo es en cada implementación.
- De partida no hay restricciones sobre el valor que puede tomar  $n_0$ , por tanto variará entre cero e infinito.
  - Un umbral de valor infinito supone no aplicar nunca DV de forma efectiva, porque siempre estaríamos resolviendo con el algoritmo básico.
  - Si  $n_0 = 1$ , entonces estaríamos en el caso opuesto, ya que el algoritmo básico sólo actúa una vez, y se aplica la recursividad continuamente.

### 3. La Determinación del Umbral

---

Ejemplo: Multiplicación de grandes números.

$$t(n) = \begin{cases} h(n) & \text{si } n \leq n_0 \\ 3t(n/2) + g(n) & \text{otro caso} \end{cases}$$

con  $h(n) \in \Theta(n^2)$ ,  $g(n) \in \Theta(n)$

¿Cuál es el valor óptimo para  $n_0$ ?

### 3. La Determinación del Umbral

---

Ejemplo: Multiplicación de grandes números.

Una implementación concreta  $h(n) = n^2$  y  $g(n) = 16n$  ( $\mu$ s), y un caso de tamaño  $n = 5000$ .

Las dos posibilidades extremas nos llevan a

Si  $n_0 = 1$ ,  $t(n) = 41$  sg

Si  $n_0 = \infty$ ,  $t(n) = h(n) = 25$  sg

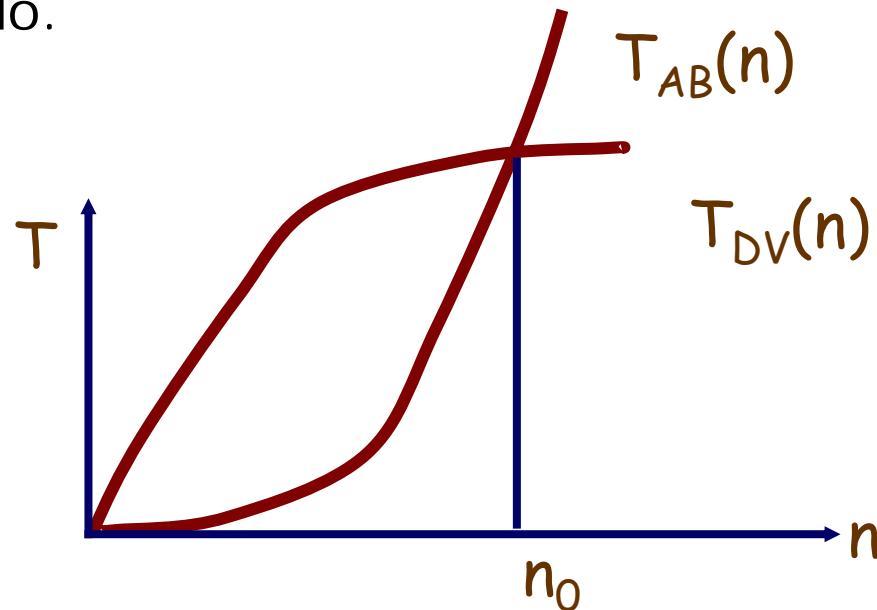
Si puede haber tan grandes diferencias, **¿como podremos determinar el valor óptimo del umbral?**

Tres métodos: **Experimental, Teórico e Híbrido**

# 3. La Determinación del Umbral

## Método experimental

- Implementamos el algoritmo básico (AB) y el algoritmo DV
- Resolvemos para distintos valores de  $n$  con ambos algoritmos
- Hay que esperar que conforme  $n$  aumente, el tiempo del algoritmo básico vaya aumentando asintóticamente, y el del DV disminuyendo.



*iMUY COSTOSO  
EN TIEMPO  
Y RECURSOS!*

# 3. La Determinación del Umbral

---

## Método teórico

- La idea del enfoque experimental se traduce teóricamente a lo siguiente

$$\begin{aligned} t(n) &= h(n) && \text{si } n \leq n_0 \\ &= 3 t(n/2) + g(n) && \text{si } n > n_0 \end{aligned}$$

- Cuando coinciden los tiempos de los dos algoritmos

$$h(n) = t(n) = 3 h(n/2) + g(n); n = n_0$$

## Método híbrido

- Para una implementación concreta (por ejemplo, la anterior,  $h(n) = n^2$  y  $g(n) = 16n$  (ms))

$$n^2 = \frac{3}{4} n^2 + 16n \rightarrow n = \frac{3}{4} n + 16$$

$$n_0 = 64$$

# 3. La Determinación del Umbral

---

## Método híbrido

- Calculamos las constantes ocultas utilizando un enfoque empírico.
- Calculamos el umbral, utilizando el criterio seguido para el umbral teórico.
- Probamos valores alrededor del umbral teórico (umbrales de tanteo) para determinar el umbral óptimo.
- Inconveniente: las constantes ocultas (son poco importantes con  $n$  grandes)

# Indice

---

## **EL ENFOQUE DIVIDE Y VENCERÁS**

- 1. Enfoque Divide y Vencerás para el Diseño deAlgoritmos**
- 2. Método General DV**
- 3. La Determinación del Umbral**

## **APLICACIONES DE LA TÉCNICA DIVIDE Y VENCERÁS**

■ **Algoritmos de Ordenación**

■ **Multiplicación de Matrices**

■ **Viajante de Comercio**

# Algoritmos de Ordenación

---

- La ordenación es *una de las tareas más frecuentemente realizadas.*
- Los algoritmos de ordenación recibirán una colección de registros a ordenar. Cada registro contendrá un campo **clave** por el que se ordenarán los registros.
- La clave puede ser de cualquier tipo (numérica, alfanumérica, ...) para el que exista una función de comparación.
- La clave debe ser de un tipo lo suficientemente grande como para que haya una relación de orden lineal entre las claves.
- Supondremos que todos los registros tienen una función **clave ()** que devuelve la clave.
- También supondremos que está definida una función **swap ()** que intercambia la posición de dos registros cualesquiera.

# Algoritmos de Ordenación

---

- **El problema de la ordenación:** Dados un conjunto de registros  $r_1, r_2, \dots, r_n$  con valores clave  $k_1, k_2, \dots, k_n$  respectivamente, fijar los registros con algún orden  $s$  tal que los registros  $r_{s1}, r_{s2}, \dots, r_{sn}$  tengan claves que obedezcan la propiedad  $k_{s1} \leq k_{s2} \leq \dots \leq k_{sn}$ .  
En otras palabras, el problema de la ordenación es fijar un conjunto de registros de forma que los valores de sus claves estén *en orden no decreciente*.
- Esta definición permite la existencia de valores clave repetidos.
- Cuando existen valores clave repetidos puede ser interesante mantener el orden relativo en que ocurren en la colección de entrada.
- Se denomina **estable** el algoritmo de ordenación que *mantiene el orden relativo en que ocurren los registros con clave repetida en la entrada*.

# Algoritmos de Ordenación

---

## ■ Lentos $\Theta(n^2)$ (*ordenación por cambio*)

- Ordenación de la burbuja
- Ordenación por inserción
- Ordenación por selección
- ✓ son algoritmos sencillos
  - ✗ se comportan mal cuando la entrada es muy grande

## ■ Rápidos $\Theta(n \log n)$

- Ordenación de Shell (Shellsort)
- Ordenación rápida (Quicksort)
- Ordenación por fusión (Mergesort)
- Ordenación por montículo (Heapsort)
- ✗ son algoritmos más complejos
  - ✓ se comportan muy bien cuando la entrada es muy grande.

# Algoritmos de Ordenación

---

## Ordenación por inserción I

- La ordenación por inserción procesa secuencialmente la lista de registros.
- El algoritmo mantiene una lista ordenada con aquellos registros ya procesados.
- Cada registro se inserta en su posición correcta dentro de la lista ordenada cuando le toca.

# Algoritmos de Ordenación

---

## Ordenación por inserción II

```
public void insercion (Elemento [] vector) {  
    // método de ordenación por inserción  
    int i, j;  
    for (i = 1; i < vector.length; i++)  
        for (j = i; (j > 0) && (vector[j].clave () < vector[j - 1].clave ()); j--)  
            swap (vector, j, j - 1);  
}
```

```
public void swap (Elemento [] vector, int i, int j) {  
    // método que intercambia dos posiciones del vector  
    Elemento aux;  
  
    aux = vector [i];  
    vector [i] = vector [j];  
    vector [j] = aux;  
}
```

# Algoritmos de Ordenación

## Ordenación por inserción III

**pasada 0:** 42 20 17 13 28 14 23 15



**pasada 1:** 20 42 17 13 28 14 23 15



**pasada 2:** 17 20 42 13 28 14 23 15



**pasada 3:** 13 17 20 42 28 14 23 15



**pasada 4:** 13 17 20 28 42 14 23 15



**pasada 5:** 13 14 17 20 28 42 23 15



**pasada 6:** 13 14 17 20 23 28 42 15



**pasada 7:** 13 14 15 17 20 23 28 42

# Algoritmos de Ordenación

---

## Ordenación por inserción IV

### Análisis del algoritmo

- El cuerpo del algoritmo está formado por dos bucles *for* anidados.
- El bucle *for* externo se ejecuta  $n - 1$  veces.
- El bucle interno depende del número de claves en la parte ordenada que son menores (o mayores) que la clave a la que buscamos acomodo.

# Algoritmos de Ordenación

---

## Ordenación por inserción IV

- En el peor caso, cada registro se debe mover hasta el principio del vector (*i comparaciones* por pasada).

En el peor caso, el coste será:

$$\sum_{i=1}^n i = \Theta(n^2)$$

- En el mejor caso, las claves estarán ordenadas de menor a mayor y sólo habrá que realizar 1 comparación por pasada.

$$\sum_{i=1}^n 1 = \Theta(n)$$

# Algoritmos de Ordenación

---

## Ordenación por inserción IV

- El *mejor caso* es significativamente más rápido que el *peor caso*.
- El **peor caso** suele ser una **indicación de mayor confianza** del tiempo “típico” que tarda el proceso en realizarse.
- Hay situaciones en la que es muy posible que se comporte como en el mejor caso:
  - e.g.: una lista ordenada se desordena ligeramente.
- Algunos algoritmos *aprovechan este mejor caso* del algoritmo de ordenación por inserción para mejorar su rendimiento:
  - Shellsort
  - Quicksortcuando van a ordenar listas pequeñas (9 elementos o menos) utilizan inserción.

# Algoritmos de Ordenación

---

## Ordenación por inserción IV

- ¿Cuál es el **coste del caso medio**?
- Cuando se ordena el registro *i*-ésimo, el número de iteraciones en el bucle interno depende de lo “desordenado” que estuviera este registro.
- Se darán tantas pasadas al bucle interno como registros haya con clave mayor que la del registro *i*-ésimo.
- Cada vez que se intercambian dos posiciones se denomina **inversión**.
- Contar el número de inversiones determina el número de comparaciones e intercambios a realizar.
- Supongamos que, en media, la mitad de las primeras *i*-1 claves tienen clave mayor que la del registro *i*-ésimo.
- El caso medio debería tener cerca de la mitad del coste del peor caso,  $\Theta(n^2)$ .

# Algoritmos de Ordenación

---

## Ordenación por burbuja

Es de los métodos más simples.

Idea: Los elementos más ligeros ascienden.

```
void burbuja(int T[], int tam)
{
    int i, j;
    int aux;
    for (i = 0; i < tam - 1; i++)
        for (j = tam - 1; j > i; j--)
            if (T[j] < T[j-1]) {
                aux = T[i];
                T[i] = T[j];
                T[j] = aux;
            }
}
```

# Algoritmos de Ordenación

---

## Algoritmo por montículos (heapsort)

Se basa en la simulación de la inserción y borrado en un árbol parcialmente ordenado, simulado sobre un vector:

```
template <typename tipo>
void heapsort(vector<tipo> &T, int inicial, int final)
{
    APO<tipo> a;
    for (int i = inicial; i < final; i++)
        a.insertar(T[i]);
    for (i = inicial; i < final; i++)
        T[i] = a.BorrarMinimo();
}
```

# Algoritmos de Ordenación

---

```
template <typename tipo>
void heapsort(vector<tipo> &T)
{
    int N = T.size()-1;
    for (int i= N/2 -1; i>= 0; i--)
        reajustar(T, N, i);
    for (i= N-1; i>=1; i--) {
        swap(T[0], T[i]);
        reajustar(T, i, 0);
    }
}
```

donde **reajustar(T, n, j)** coloca el elemento **j** en la posición que le corresponde viendo el vector **T** como el recorrido por niveles de un APO con **n** elementos

# Algoritmos de Ordenación

---

El esquema general de ordenación

Divide y Vencerás es el siguiente

## **Algoritmo de Ordenacion con Divide y Vencerás**

### **Begin Algoritmo**

**Iniciar Ordenar(L)**

**Si L tiene longitud mayor de 1 Entonces**

**Begin**

**Partir la lista en dos listas, izquierda y  
derecha**

**Iniciar Ordenar(izquierda)**

**Iniciar Ordenar(derecha)**

**Combinar izquierda y derecha**

**End**

**End Algoritmo**

# Algoritmos de Ordenación

---

## Ordenación por mezcla

- Divide y Vencerás:

- Si  $n=1$  terminar (toda lista de 1 elemento esta ordenada)
- Si  $n>1$ , partir la lista de elementos en dos o mas subcolecciones; ordenar cada una de ellas; combinar en una sola lista.

J\_li&t=-g i b []\_lf[.j [ln]ch9

# Algoritmos de Ordenación

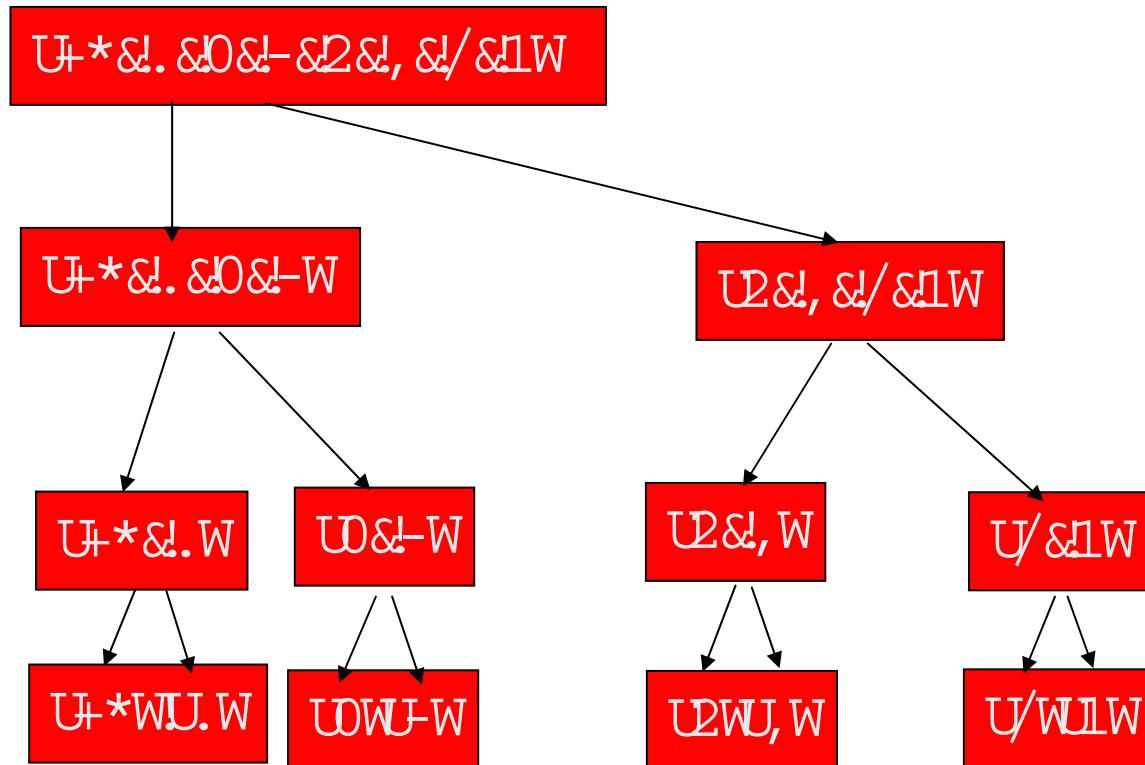
---

## Ordenación por mezcla

- Buscamos hacer una partición equilibrada de la lista en dos partes A y B
- En A habrá  $n/k$  elementos, y en B el resto
- Ordenamos entonces A y B recursivamente
- Combinamos las listas ordenadas A y B usando un procedimiento llamado **mezcla**, que combina las dos listas en una sola
- Las diferentes posibilidades nos las va a dar el valor k que escojamos

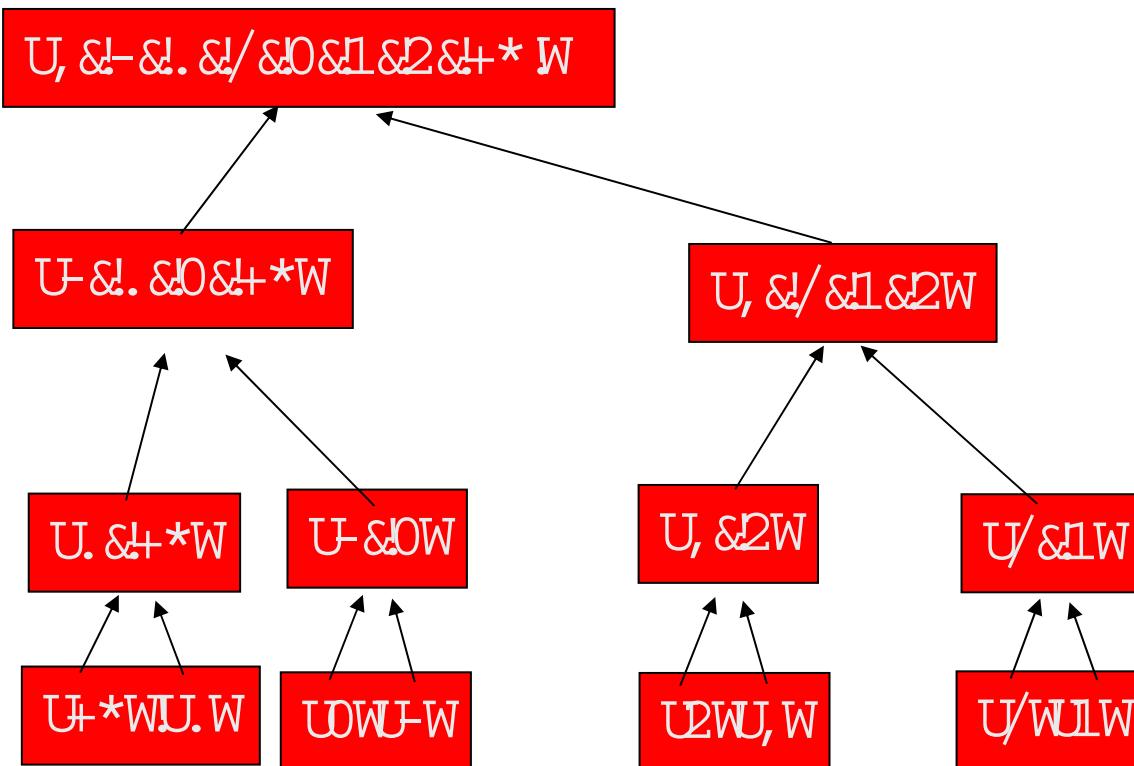
# Algoritmos de Ordenación

Ejemplo:  $k=2$  (Partimos la lista en otras dos de tamaño  $n/2$ )



# Algoritmos de Ordenación

Ejemplo: La operación de mezcla para  $k=2$



# Algoritmos de Ordenación

---

Código de ordenación por mezcla

```
void mergeSort(Comparable []a, int left, int right)
{
    // sort a[left:right]
    if (left < right)
    { // al menos dos elementos
        int mid = (left+right)/2; //punto medio
        mergeSort(a, left, mid);
        mergeSort(a, mid + 1, right);
        merge(a, b, left, mid, right); // combina de "a" a
        "b"
        copy(b, a, left, right); //copy el resultado en a
    }
}
```

# Algoritmos de Ordenación

---

## Cálculo de la eficiencia

### Ecuación recurrente

- Suponemos que  $n$  es potencia de 2

$$T(n) = \begin{cases} c_1 & \text{si } n=1 \\ 2T(n/2) + c_2n & \text{si } n>1, n=2^k \end{cases}$$

- Tenemos

$$T(n) = c_1n + c_2n\log n$$

- Por tanto el tiempo para el algoritmo de ordenacion por mezcla es  $O(n\log n)$

# Algoritmos de Ordenación

---

## Quicksort

- Es el algoritmo (general) de ordenación mas eficiente
  - ordena el array A eligiendo un valor clave v entre sus elementos, que actua como pivote
  - organiza tres secciones: izquierda, pivote, derecha
  - todos los elementos en la izquierda son menores que el pivote, todos los elementos en la derecha son mayores o iguales que el pivote
  - ordena los elementos en la izquierda y en la derecha, sin requerir ninguna mezcla para combinarlos.
  - lo ideal sería que el pivote se colocara en la mediana para que la parte izquierda y la derecha tuvieran el mismo tamaño

# Algoritmos de Ordenación

---

Pseudo Código para quicksort

**Algoritmo QUICKSORT(S,T)**

**IF TAMAÑO(S) ≤ q (umbral) THEN INSERCIÓN(S,T)  
ELSE**

Elegir cualquier elemento p del array como pivote

Partir S en (S1,S2,S3) de modo que

1.  $\forall x \in S1, y \in S2, z \in S3$  se verifique  $x < p < z$  e  $y = p$

2.  $TAMAÑO(S1) < TAMAÑO(S)$  y  $TAMAÑO(S3) < TAMAÑO(S)$

QUICKSORT(S1,T1) // ordena recursivamente particion izquierda

QUICKSORT(S3,T3) // ordena recursivamente particion derecha

Combinacion:  $T = T1 || S2 || T3$  // S2 es el elemento intermedio entre cada mitad ordenada

**End Algoritmo**

# Algoritmos de Ordenación

---

Quicksort: La elección del pivote

- **La elección condiciona el tiempo de ejecución**
- El pivote puede ser cualquier elemento en el dominio, pero no necesariamente tiene que estar en S
  - Podria ser la media de los elementos seleccionados en S
  - Podria elegirse aleatoriamente, pero la función RAND() consume tiempo, que habria que añadirselo al tiempo total del algoritmo
- Pivotes usuales son la mediana de un mínimo de tres elementos, o el elemento medio de S.

# Algoritmos de Ordenación

---

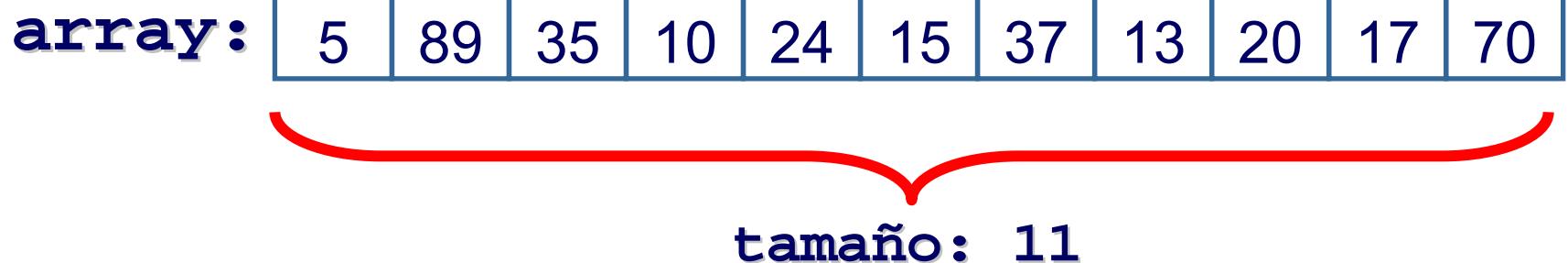
## Quicksort: La elección del pivote

- El empleo de la mediana de tres elementos no tiene justificación teórica.
- Si queremos usar el concepto de mediana, deberíamos escoger como pivote la mediana del array porque lo divide en dos sub-arrays de igual tamaño
  - mediana =  $(n/2)^{\text{o}}$  mayor elemento
  - elegir tres elementos al azar y escoger su mediana; esto suele reducir el tiempo de ejecución aproximadamente en un 5%
- La elección más rápida es escoger como pivote, entre los dos primeros elementos del array, el mayor de ellos

# Algoritmos de Ordenación

---

Quicksort: Ejemplo escogiendo el elemento medio



Con este ejemplo vamos a ilustrar su funcionamiento

# Algoritmos de Ordenación

---

Quicksort: Ejemplo

**array:**  5 89 35 14 24 15 37 13 20 7 70

“*elemento  
pivot*”

# Algoritmos de Ordenación

---

Quicksort: Ejemplo

**array:**

5	89	35	14	24	15	37	13	20	7	70
---	----	----	----	----	----	----	----	----	---	----

**partición:**



5	89	35	14	24	15	37	13	20	7	70
---	----	----	----	----	----	----	----	----	---	----



7	14	5	13	15	35	37	89	20	24	70
---	----	---	----	----	----	----	----	----	----	----



**índice:** 4

# Algoritmos de Ordenación

---

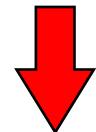
Quicksort: Ejemplo

**array[0]**



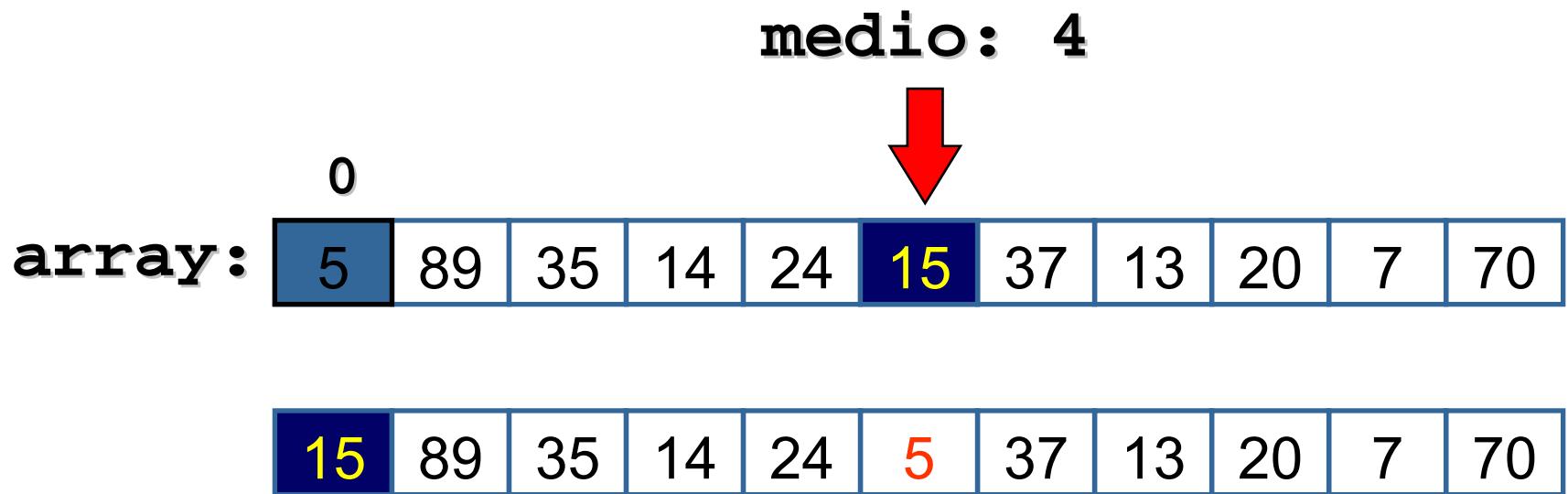
**indice**

**array[indice + 1]**



**(tamaño - indice - 1)**

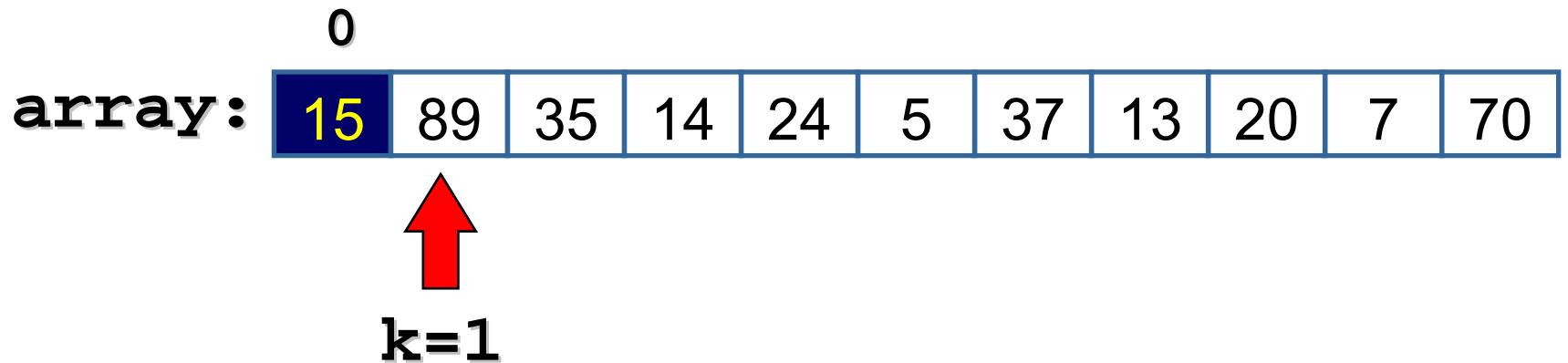
# Algoritmos de Ordenación



Quicksort: Ejemplo

# Algoritmos de Ordenación

---

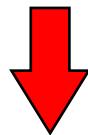


Quicksort: Ejemplo

# Algoritmos de Ordenación

---

**índice: 0**



**array:**

15	89	35	14	24	5	37	13	20	7	70
----	----	----	----	----	---	----	----	----	---	----



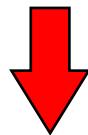
**k: 1**

Quicksort: Ejemplo

# Algoritmos de Ordenación

---

**índice: 0**



**array:**

15	89	35	14	24	5	37	13	20	7	70
----	----	----	----	----	---	----	----	----	---	----



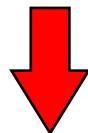
**k: 2**

Quicksort: Ejemplo

# Algoritmos de Ordenación

---

**indice:** 0



**array:**

15	89	35	14	24	5	37	13	20	7	70
----	----	----	----	----	---	----	----	----	---	----



**k:** 3

Quicksort: Ejemplo

# Algoritmos de Ordenación

---

**indice: 1**



**array:**

15	14	35	89	24	5	37	13	20	7	70
----	----	----	----	----	---	----	----	----	---	----



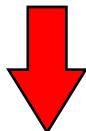
**k: 3**

Quicksort: Ejemplo

# Algoritmos de Ordenación

---

**indice: 1**



**array:**

15	14	35	89	24	5	37	13	20	7	70
----	----	----	----	----	---	----	----	----	---	----



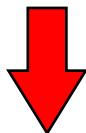
**k: 4**

Quicksort: Ejemplo

# Algoritmos de Ordenación

---

**indice: 1**



**array:**

15	14	35	89	24	5	37	13	20	7	70
----	----	----	----	----	---	----	----	----	---	----



**k: 5**

Quicksort: Ejemplo

# Algoritmos de Ordenación

---

**índice: 2**



**array:**

15	14	5	89	24	35	37	13	20	7	70
----	----	---	----	----	----	----	----	----	---	----



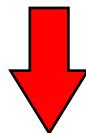
**k: 5**

Quicksort: Ejemplo

# Algoritmos de Ordenación

---

**índice: 2**



**array:**



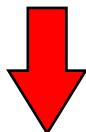
**k: 6**

Quicksort: Ejemplo

# Algoritmos de Ordenación

---

**índice: 2**



**array:**

15	14	5	89	24	35	37	13	20	7	70
----	----	---	----	----	----	----	----	----	---	----



**k: 7      etc...**

Quicksort: Ejemplo

# Algoritmos de Ordenación

índice: 4



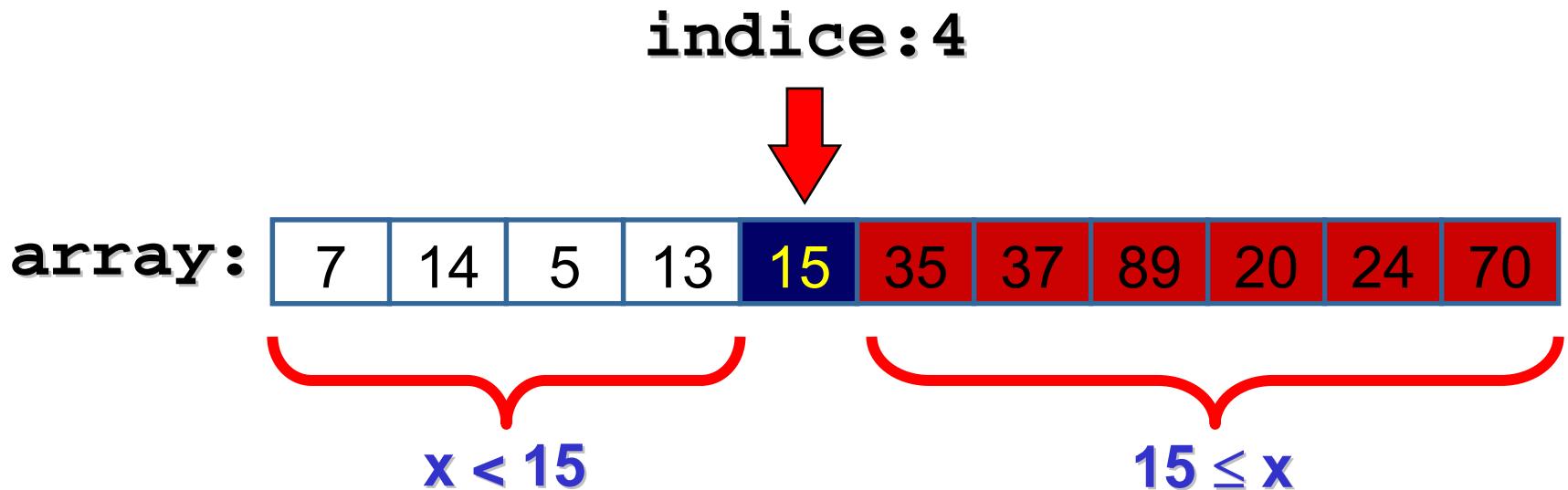
array:



k: 11

Quicksort: Ejemplo

# Algoritmos de Ordenación



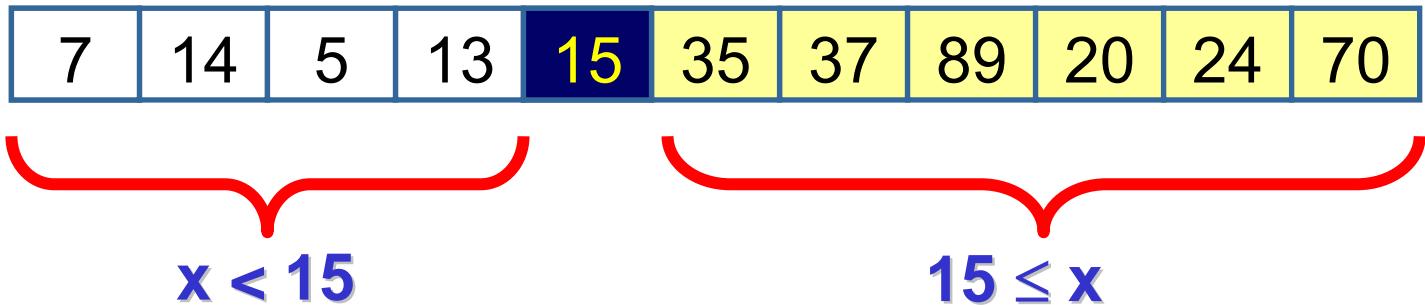
Quicksort: Ejemplo

# Algoritmos de Ordenación

---

array:

*El pivote ahora está en la posición correcta*



Quicksort: Ejemplo

# Algoritmos de Ordenación

---



Ordena



Ordena



Quicksort: Ejemplo

# Algoritmos de Ordenación

---

## Procedimiento Quicksort

Procedimiento quicksort ( $T[i..j]$ )

{ordena un array  $T[i..j]$  en orden creciente}

Si  $j-i$  es pequeño Entonces Insercion ( $T[i..j]$ )

Caso contrario

Escoger un pivote adecuado (sea  $I$  la posición del mismo)

Pivoteo\_lineal ( $T[i..j]$ ,  $I$ )

{tras el pivoteo,  $i \leq k < I \Rightarrow T[k] \leq T[I]$  y,

$I < k \leq j \Rightarrow T[k] > T[I]$ }

quicksort ( $T[i..I-1]$ )

quicksort ( $T[I+1..j]$ )

*El pivote sigue quedándose fuera de ambas partes*

# Algoritmos de Ordenación

---

## Quicksort: Pivoteo\_lineal ( $T[i..j]$ , var pos)

- Intercambiar  $T[pos]$  y  $T[i]$
- Sea  $p = T[i]$  el pivote.
- Una buena forma de pivotear consiste en explorar el array  $T[i..j]$  solo una vez, pero comenzando desde ambos extremos.
- Los punteros  $k$  y  $l$  se inicializan en  $i$  y  $j+1$  respectivamente.
- El puntero  $k$  se incrementa entonces hasta que  $T[k] > p$ , y el puntero  $l$  se disminuye hasta que  $T[l] \leq p$ . Ahora  $T[k]$  y  $T[l]$  estan intercambiados. Este proceso continua mientras que  $k < l$ .
- Finalmente,  $T[i]$  y  $T[l]$  se intercambian para poner el pivote en su posicion correcta, y se devuelve  $pos = l$

# Algoritmos de Ordenación

---

## Quicksort: Algoritmo de Pivoteo

Procedimiento Pivoteo\_lineal ( $T[i..j]$ , var pos)

{ permuta los elementos en el array  $T[i..j]$  de tal forma que al final  $i \leq l \leq j$ , los elementos de  $T[i..l-1]$  no son mayores que p,  $T[l] = p$ , y los elementos de  $T[l+1..j]$  son mayores que p, donde p es el valor inicial de  $T[i]$ }

intercambiar  $T[pos]$  y  $T[i]$ ;

$p = T[i]$

$k = i; l = j + 1$ ;

repetir  $k = k + 1$  hasta  $T[k] > p$  o  $k \geq j$

repetir  $l = l - 1$  hasta  $T[l] \leq p$

Mientras  $k < l$  hacer

{ intercambiar  $T[k]$  y  $T[l]$

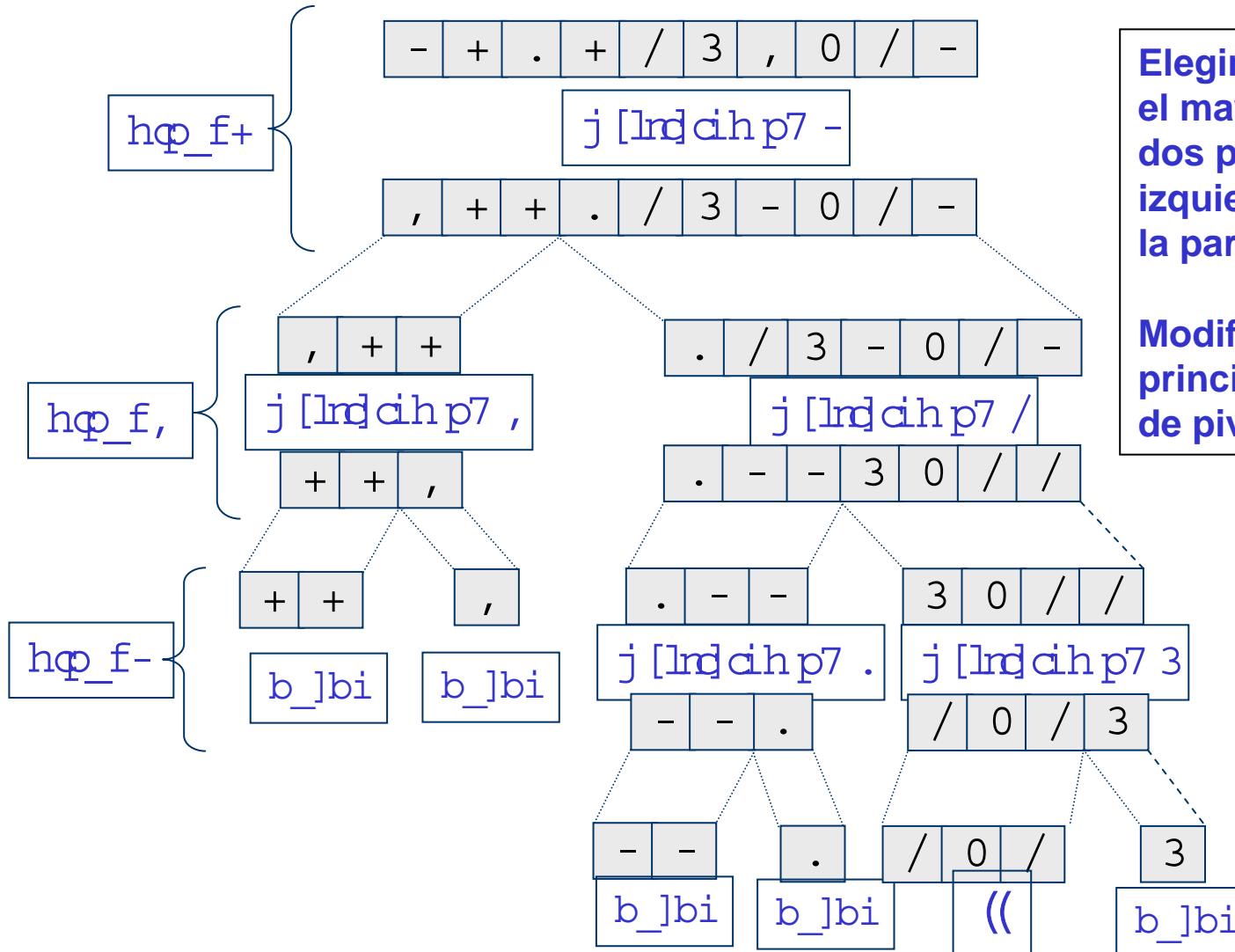
repetir  $k = k + 1$  hasta  $T[k] > p$

repetir  $l = l - 1$  hasta  $T[l] \leq p$ }

intercambiar  $T[i]$  y  $T[l]$

$pos = l$ ;

# Algoritmos de Ordenación



**Elegimos el pivote como el mayor elemento de los dos primeros por la izquierda y se incluye en la parte de la derecha**

## Modificación del cuerpo principal y de la función de pivoteo lineal

# Quicksort: Otro Ejemplo

# Algoritmos de Ordenación

---

## Eficiencia de quicksort

- Si admitimos que
  - El procedimiento de pivoteo es lineal,
  - Quicksort lo llamamos para  $T[1..n]$ , y
  - Elegimos como peor caso que el pivote es el primer elemento del array,
- Entonces el tiempo del anterior algoritmo es
$$T(n) = T(1) + T(n-1) + an$$
- Que evidentemente proporciona un tiempo cuadrático

# Algoritmos de Ordenación

---

## Analisis de Quicksort

- Recordemos que el algoritmo de ordenación por Inserción hacia aproximadamente  $n^2/2 - n/2$  comparaciones, es decir es  $O(n^2)$  en el peor caso.
- En el peor caso quicksort es tan malo como el peor caso del método de inserción (y tambien de selección).
- Es que el número de intercambios que hace quicksort es unas 3 veces el número de intercambios que hace el de inserción en el peor de los casos.
- Sin embargo, en la práctica quicksort es el mejor algoritmo de ordenacion que se conoce...
- ¿Qué pasará con el tiempo del caso promedio?

# Algoritmos de Ordenación

---

## Análisis del caso promedio

- Suponemos que la lista esta dada en orden aleatorio
- Suponemos que todos los posibles ordenes del array son igualmente probables
- El pivote puede ser cualquier elemento
- Puede demostrarse que en el caso promedio quicksort tiene un tiempo  $T(n) = 2n \ln n + O(n)$ , que se debe al numero de comparaciones que hace en promedio en una lista de  $n$  elementos
- Quicksort, tiene un tiempo promedio  $O(n \log n)$

# Multiplicación de Matrices

---

- Si tenemos dos matrices A y B cuadradas en donde A tiene el mismo número de filas que columnas de B, se trata de multiplicar A y B para obtener una nueva matriz C.
- La multiplicación de matrices se realiza conforme a

$$C_{ij} = \sum_{k=1}^n A_{ik} \cdot B_{kj}$$

- Esta fórmula corresponde a la multiplicación normal de matrices, que consiste en tres bucles anidados, por lo que es  $O(n^3)$ .
- Para aplicar la técnica DV, vamos a proceder como con la multiplicación de enteros, con la intención de obtener un algoritmo más eficiente para multiplicar matrices.

# Multiplicación de matrices

- Supongamos el problema de multiplicar dos matrices cuadradas A, B de tamaños  $n \times n$ .  $C = AxB$

$$C(i, j) = \sum_{k=1..n} A(i, k) \cdot B(k, j); \text{ Para todo } i, j = 1..n$$

- Método clásico de multiplicación:

```
for i:= 1 to N do
    for j:= 1 to N do
        suma:= 0
        for k:= 1 to N do
            suma:= suma + a[i,k]*b[k,j]
        end
        c[i, j]:= suma
    end
end
```

- El método clásico de multiplicación requiere  $\Theta(n^3)$ .

# Multiplicación de matrices

- Aplicamos **divide y vencerás**:

Cada matriz de  $n \times n$  es dividida en cuatro submatrices de tamaño  $(n/2) \times (n/2)$ :  $A_{ij}$ ,  $B_{ij}$  y  $C_{ij}$ .

$$\begin{array}{|c|c|} \hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline \end{array} \times \begin{array}{|c|c|} \hline B_{11} & B_{12} \\ \hline B_{21} & B_{22} \\ \hline \end{array} = \begin{array}{|c|c|} \hline C_{11} & C_{12} \\ \hline C_{21} & C_{22} \\ \hline \end{array}$$
$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

- Es necesario resolver 8 problemas de tamaño  $n/2$ .
- La combinación de los resultados requiere un  $O(n^2)$ .

$$t(n) = 8 \cdot t(n/2) + a \cdot n^2$$

- Resolviéndolo obtenemos que  $t(n)$  es  $O(n^3)$ .
- Podríamos obtener una mejora si hicierámos 7 multiplicaciones (o menos)...

# Multiplicación de matrices

- Multiplicación rápida de matrices (Strassen):

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$Q = (A_{12} + A_{22})B_{11}$$

$$R = A_{11}(B_{12} - B_{22})$$

$$S = A_{22}(B_{21} - B_{11})$$

$$T = (A_{11} + A_{12})B_{22}$$

$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P + S - T + U$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U$$

- Tenemos 7 subproblemas de la mitad de tamaño.
- ¿Cuánto es el tiempo de ejecución?

# Multiplicación de Matrices

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} = \begin{pmatrix} ae + bf & ag + bh \\ ce + df & cg + dh \end{pmatrix}$$

$\uparrow$        $\uparrow$        $\uparrow$   
 $C$        $A$        $B$

El método de Strassen

$$P = (a+d)(e+h)$$

$$Q = (c+d)e$$

$$R = a(g-h)$$

$$S = d(f-e)$$

$$T = (a+b)h$$

$$U = (c-a)(e+g)$$

$$V = (b-d)(f+h)$$

$$r = P+S-T+V$$

$$s = R+T$$

$$t = Q+S$$

$$u = P+R-Q+U$$

- Es evidente que solo se necesitan 7 multiplicaciones en lugar de las anteriores 8, más adiciones/substracciones  $O(n^2)$ .

# Multiplicación de matrices

- El tiempo de ejecución será:

$$t(n) = 7 \cdot t(n/2) + a \cdot n^2$$

- Resolviéndolo, tenemos que:

$$t(n) \in O(n^{\log_2 7}) \approx O(n^{2.807}).$$

- Las constantes que multiplican al polinomio son mucho mayores (tenemos muchas sumas y restas), por lo que sólo es mejor cuando la entrada es muy grande (empíricamente, para valores en torno a  $n>120$ ).

# Multiplicación de matrices

- Aunque el algoritmo es más complejo e inadecuado para tamaños pequeños, se demuestra que la **cota de complejidad del problema** es menor que  $O(n^3)$ .
- **Cota de complejidad de un problema:** tiempo del algoritmo más rápido posible que resuelve el problema.
- Algoritmo clásico  $\rightarrow O(n^3)$
- V. Strassen (1969)  $\rightarrow O(n^{2.807})$
- V. Pan (1984)  $\rightarrow O(n^{2.795})$
- D. Coppersmith y S. Winograd (1990)  $\rightarrow O(n^{2.376})$
- ...

# Divide y vencerás.

## Conclusiones:

- **Idea básica Divide y Vencerás:** dado un problema, descomponerlo en partes, resolver las partes y juntar las soluciones.
- Idea muy sencilla e intuitiva, pero...
- ¿Qué pasa con los problemas reales de interés?
  - Pueden existir muchas formas de descomponer el problema en subproblemas → Quedarse con la mejor.
  - Puede que no exista ninguna forma viable, los subproblemas no son independientes → Descartar la técnica.
- Divide y vencerás requiere la existencia de un **método directo** de resolución:
  - Tamaños pequeños: solución directa.
  - Tamaños grandes: descomposición y combinación.
  - ¿Dónde establecer el límite pequeño/grande?

# Algorítmica

---

**Tema 1. Planteamiento General**

**Tema 2. La Eficiencia de los Algoritmos**

**Tema 3. Algoritmos “Divide y vencerás”**

**Tema 4. Algoritmos Voraces (“Greedy”)**

**Tema 5. Algoritmos basados en Programación Dinámica**

**Tema 6. Algoritmos para la Exploración de Grafos**

**(“Backtracking”, “Branch and Bound”)**

**Tema 7. Otras metodologías algorítmicas**

# Algorítmica

---

**Tema 1. Planteamiento General**

**Tema 2. La Eficiencia de los Algoritmos**

**Tema 3. Algoritmos “Divide y vencerás”**

**Tema 4. Algoritmos Voraces (“Greedy”)**

**Tema 5. Algoritmos basados en Programación Dinámica**

**Tema 6. Algoritmos para la Exploración de Grafos  
 (“Backtracking”, “Branch and Bound”)**

**Tema 7. Otras metodologías algorítmicas**

# Tema 4: Algoritmos Voraces ("Greedy")

---

## Bibliografía:

- G. BRASSARD, P. BRATLEY. Fundamentos de Algoritmia. Prentice Hall (1997).
- T.H. CORMEN, C.E. LEISERSON, R.L. RIVEST. Introduction to Algorithms. The MIT Press (1992)
- E. HOROWITZ, S. SAHNI, S. RAJASEKARAN. Computer Algorithms. Computer Science Press (1998).

# Objetivos

---

- Comprender la filosofía de diseño de algoritmos voraces
- Conocer las características de un problema resoluble mediante un algoritmo voraz
- Resolución de diversos problemas
- Heurísticas voraces: Soluciones aproximadas a problemas

# Índice

---

- EL ENFOQUE GREEDY
- ALGORITMOS GREEDY EN GRAFOS
- HEURÍSTICA GREEDY

# Índice

---

## ■ EL ENFOQUE GREEDY

- Características Generales
- Elementos de un Algoritmo Voraz
- Esquema Voraz
- Ejemplo: Problema de Selección de Actividades
- Ejemplo: Almacenamiento Optimal en Cintas
- Ejemplo: Problema de la Mochila Fraccional

## ■ ALGORITMOS GREEDY EN GRAFOS

## ■ HEURÍSTICA GREEDY

# Características generales de los algoritmos voraces

---

- Se utilizan generalmente para resolver problemas de optimización: máximo o mínimo
- Un algoritmo “greedy” toma las decisiones en función de la información que está disponible en cada momento.
- Una vez tomada la decisión no vuelve a replantearse en el futuro.
- Suelen ser rápidos y fáciles de implementar.
- No siempre garantizan alcanzar la solución óptima.

# Características generales de los algoritmos voraces

---



*¡Comete siempre todo  
lo que tengas a mano!*

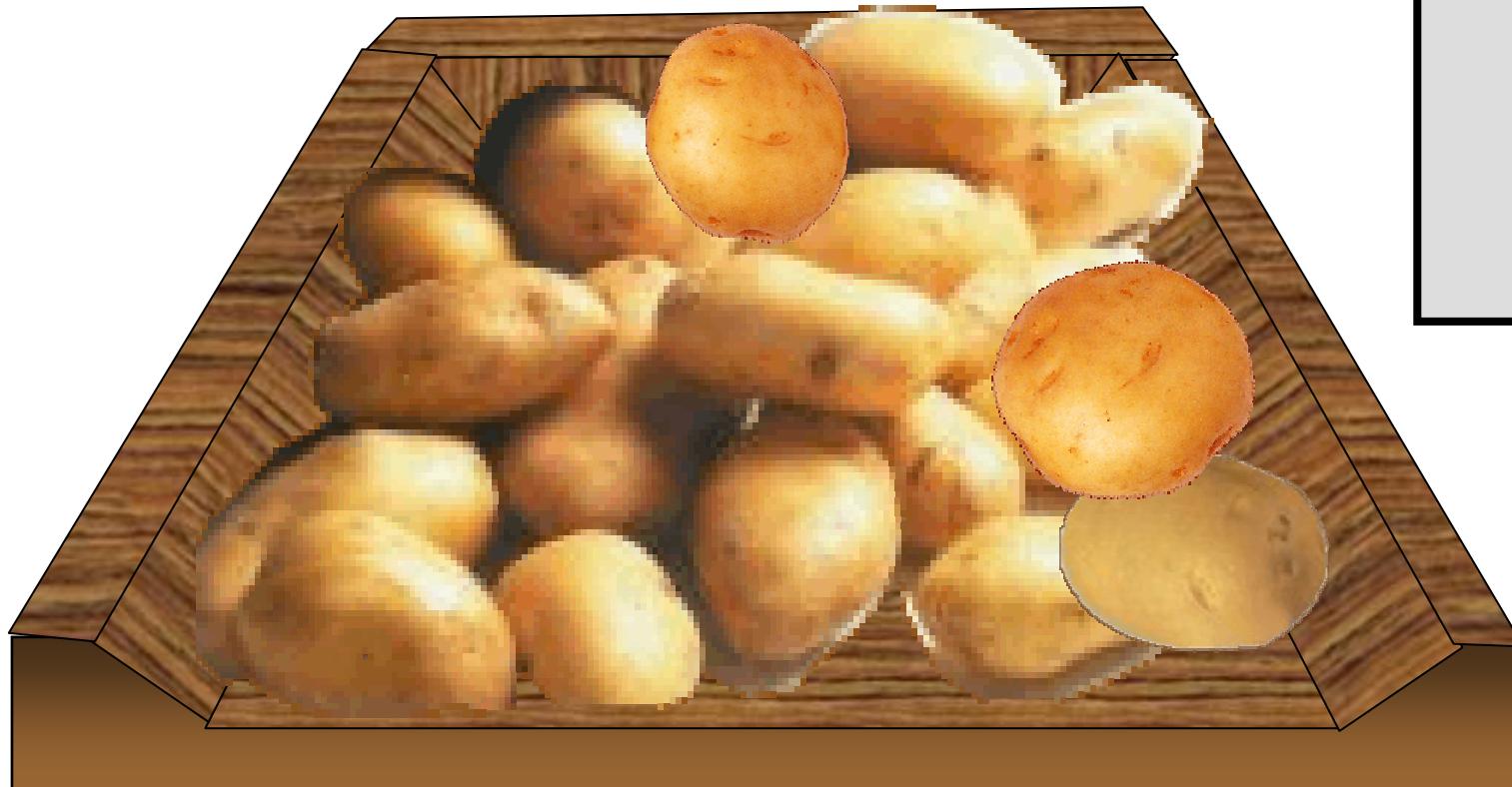
El término **greedy** es  
sinónimo de voraz,  
ávido, glotón, ...

# Características generales

- Como decíamos, los algoritmos **voraces**, **ávidos** o de **avance rápido** (en inglés **greedy**) se utilizan normalmente en problemas de optimización.
  - El problema se *interpreta* como: “tomar algunos elementos de entre un conjunto de candidatos”.
  - El **orden** el que se cogen puede ser importante o no.
- Un **algoritmo voraz** funciona por pasos:
  - Inicialmente partimos de una solución vacía.
  - En cada paso se escoge el siguiente elemento para añadir a la solución, entre los candidatos.
  - Una vez tomada esta decisión no se podrá deshacer.
  - El algoritmo acabará cuando el conjunto de elementos seleccionados constituya una solución.

# Características generales

- **Ejemplo:** “el viejo algoritmo de comprar patatas en el mercado”. ¿Sí o no?



2 Kg.

# Características generales

## **Características básicas del algoritmo:**

- Inicialmente empezamos con una solución “vacía”, sin patatas.
- Función de selección: seleccionar la mejor patata del montón o la que “parezca” que es la mejor.
- Examinar la patata detenidamente y decidir si se coge o no.
- Si no se coge, se aparta del montón.
- Si se coge, se mete a la bolsa (y ya no se saca).
- Una vez que tenemos 2 kilos paramos.

# Características generales

- Se puede generalizar el proceso intuitivo a un esquema algorítmico general.
- El esquema trabaja con los siguientes conjuntos de elementos:
  - **C**: Conjunto de elementos **candidatos**, **pendientes** de seleccionar (inicialmente todos).
  - **S**: Candidatos seleccionados para la **solución**.
  - **R**: Candidatos seleccionados pero **rechazados** después.
- ¿Qué o cuáles son los candidatos? Depende de cada problema.

# Características generales

- **Esquema general de un algoritmo voraz:**  
**voraz (C: CjtoCandidatos; var S: CjtoSolución)**

S:=  $\emptyset$

mientras ( $C \neq \emptyset$ ) Y NO **solución(S)** hacer

    x:= **seleccionar(C)**

    C:= C - {x}

    si **factible(S, x)** entonces

**insertar(S, x)**

    finsi

finmientras

si NO **solución(S)** entonces

    devolver “No se puede encontrar solución”

    finsi

# Características generales

## Funciones genéricas

- **solución (S).** Comprueba si un conjunto de candidatos es una solución (independientemente de que sea óptima o no).
- **seleccionar (C).** Devuelve el elemento más “prometedor” del conjunto de candidatos pendientes (no seleccionados ni rechazados).
- **factible (S, x).** Indica si a partir del conjunto **S** y añadiendo **x**, es posible construir una solución (posiblemente añadiendo otros elementos).
- **insertar (S, x).** Añade el elemento **x** al conjunto solución. Además, puede ser necesario hacer otras cosas.
- Función **objetivo (S)**. Dada una solución devuelve el coste asociado a la misma (resultado del problema de optimización).

# Características generales

- El orden de complejidad depende del número de candidatos, de las funciones básicas a utilizar, del número de elementos de la solución.
- $n$ : número de elementos de una solución.
- Repetir, como mínimo  $m$ :
  - Comprobar si el valor es factible:  $f(m)$ .
  - Normalmente  $O(1)$ .
  - Selección de un elemento entre los candidatos:  $g(n)$ . Entre  $O(1)$  y  $O(n)$ .
- La función **factible** es parecida a **solucion**, pero con una solución parcial  $h(n)$ .
- La unión de un nuevo elemento a la solución puede requerir otras operaciones de cálculo,  $j(n, m)$ .

¡....!

# Características generales

SON 1,11 EUROS

AHÍ VAN 5 EUROS

TOMA EL CAMBIO,  
3,89 EUROS

- **Problema del cambio de monedas.**

Construir un algoritmo que dada una cantidad  $P$  devuelva esa cantidad usando el menor número posible de monedas.

Disponemos de monedas con valores de 1, 2, 5, 10, 20 y 50 céntimos de euro, 1 y 2 euros (€).



# Características generales

- **Caso 1.** Devolver 3,89 Euros.

1 moneda de 2€, 1 moneda de 1€, 1 moneda de 50 c€, 1 moneda de 20 c€, 1 moneda de 10 c€ , 1 moneda de 5 c€ y 2 monedas de 2 c€. Total: 8 monedas.



- El método intuitivo se puede entender como un **algoritmo voraz**: en cada paso añadir una moneda nueva a la solución actual, hasta llegar a P.

# Características generales

## Problema del cambio de monedas

- **Conjunto de candidatos:** todos los tipos de monedas disponibles. Supondremos una cantidad ilimitada de cada tipo.
- **Solución:** conjunto de monedas que sumen  $P$ .
- **Función objetivo:** minimizar el número de monedas.

## Representación de la solución:

- $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ , donde  $x_i$  es el número de monedas usadas de tipo  $i$ .
- Suponemos que la moneda  $i$  vale  $c_i$ .
- **Formulación:** Minimizar  $\sum_{i=1..8} x_i$ , sujeto a  $\sum_{i=1..8} x_i \cdot c_i = P$ ,  $x_i \geq 0$

# Características generales

## Funciones del esquema:

- **inicialización.** Inicialmente  $x_i = 0$ , para todo  $i = 1..8$
- **solución.** El valor actual es solución si  $\sum x_i \cdot c_i = P$
- **seleccionar.** ¿Qué moneda se elige en cada paso de entre los candidatos?
- **Respuesta:** elegir en cada paso la moneda de valor más alto posible, pero sin sobrepasar la cantidad que queda por devolver.
- **factible.** Valdrá siempre verdad.
- En lugar de seleccionar monedas de una en una, usamos la división entera y cogemos todas las monedas posibles de mayor valor.

# Características generales

- **Implementación.** Usamos una variable local **act** para acumular la cantidad devuelta hasta este punto.
- Suponemos que las monedas están ordenadas de menor a mayor valor.

**DevolverCambio (P: entero; C: array [1..n] de entero;  
var X: array [1..n] de entero)**

act:= 0

j:= n

**para i:= 1,...,n hacer**

X[i]:= 0

**mientras act ≠ P hacer**

**mientras (C[i] > (P - act)) AND (j>0) hacer j:= j - 1**

**si j==0 entonces devolver "No existe solución"**

**X[j]:= ⌊(P - act) / C[j]⌋**

**act:= act + C[j]\*X[j]**

**finmientras**

**inicialización**

**no solución(X)**

**seleccionar(C,P,X)**

**no factible(j)**

**insertar(X,j)**

## Características generales

- ¿Cuál es el orden de complejidad del algoritmo?
- ¿Garantiza siempre la solución óptima?
- Para este sistema monetario sí. Pero no siempre...
- **Ejemplo.** Supongamos que tenemos monedas de 100, 90 y 1. Queremos devolver 180.
  - **Algoritmo voraz.** 1 moneda de 100 y 80 monedas de 1: total 81 monedas.
  - **Solución óptima.** 2 monedas de 90: total 2 monedas.



# Características generales

- El orden de complejidad depende de:
  - El número de candidatos existentes.
  - Los tiempos de las funciones básicas utilizadas.
  - El número de elementos de la solución.
  - ...
- **Ejemplo.**  $n$ : número de elementos de  $C$ .  $m$ : número de elementos de una solución.
- Repetir, como máximo  $n$  veces y como mínimo  $m$ :
  - Función solución:  $f(m)$ . Normalmente  $O(1)$  ó  $O(m)$ .
  - Función de selección:  $g(n)$ . Entre  $O(1)$  y  $O(n)$ .
  - Función **factible** (parecida a **solución**, pero con una solución parcial):  $h(m)$ .
  - Inserción de un elemento:  $j(n, m)$ .

# Características generales

- Tiempo de ejecución **genérico**:  
 $t(n,m) \in O(n^*(f(m)+g(n)+h(m)) + m*j(n, m))$
- Ejemplos:
  - Algoritmo de Dijkstra: **n** candidatos, la función de selección e inserción son  $O(n)$ :  $O(n^2)$ .
  - Devolución de monedas: podemos encontrar el siguiente elemento en un tiempo constante (ordenando las monedas):  $O(n)$ .
- El análisis depende de cada algoritmo concreto.
- En la práctica los algoritmos voraces **suelen ser** bastante **rápidos**, encontrándose dentro de órdenes de complejidad **polinomiales**.

# Elementos de un algoritmo voraz

---

Para poder resolver un problema con un enfoque greedy, ha de reunir las 6 siguientes características, que no son necesarias, pero si suficientes:

1. **Conjunto de *candidatos a seleccionar*.**
2. **Conjunto de *candidatos seleccionados***
3. **Función Solución:** determina si los candidatos seleccionados han alcanzado una solución.

# Elementos de un algoritmo voraz

---

4. ***Función de Factibilidad***: determina si es posible completar el conjunto de candidatos seleccionados con el siguiente elemento seleccionado para alcanzar una solución al problema.
5. ***Función Selección***: determina el mejor candidato del conjunto a seleccionar.
6. ***Función Objetivo***: da el valor de la solución alcanzada.

# Esquema de un algoritmo voraz

---

Un algoritmo Greedy procede siempre de la siguiente manera:

- **Se parte de un conjunto de candidatos seleccionados vacío:  $S = \emptyset$**
- **De la lista de candidatos C que hemos podido identificar, con la función de selección, se coge el mejor candidato posible,**
- **Vemos si con ese elemento podríamos llegar a constituir una solución: Si se verifican las condiciones de factibilidad en S se añade y se borra de C**
- **Si el candidato anterior no es válido, lo borramos de la lista de candidatos posibles, y nunca mas es considerado**
- **Utilizamos la función solución. Si todavía no tenemos una solución seleccionamos con la función de selección otro candidato y repetimos el proceso anterior hasta alcanzar la solución.**

# Esquema de un algoritmo voraz

---

Voraz( $C$  : conjunto de candidatos) : conjunto solución

$S = \emptyset$

**mientras**  $C \neq \emptyset$  y no Solución( $S$ ) **hacer**

$x = \text{Seleccion}(C)$

$C = C - \{x\}$

**si** factible( $S \cup \{x\}$ ) **entonces**

$S = S \cup \{x\}$

**fin si**

**fin mientras**

**si** Solución( $S$ ) **entonces**

        Devolver  $S$

**en otro caso**

        Devolver "No se encontró una solución"

**fin si**

El enfoque Greedy suele proporcionar soluciones óptimas, pero no hay garantía de ello. Por tanto, siempre habrá que estudiar la **corrección del algoritmo** para verificar esas soluciones

# Problema Selección de Actividades

---

Tenemos la entrada de una Exposición que organiza un conjunto de actividades

- Para cada actividad conocemos su horario de comienzo y fin.
- Con la entrada podemos asistir a todas las actividades.
- Hay actividades que se solapan en el tiempo.

**Objetivo:** Asistir al mayor número de actividades =>  
Problema de selección de actividades.

**Otra alternativa:** Minimizar el tiempo que estamos ociosos.

# Problema Selección de Actividades

---

Dado un conjunto  $C$  de  $n$  actividades

$s_i$  = tiempo de comienzo de la actividad  $i$

$f_i$  = tiempo de finalización de la actividad  $i$

- Encontrar el subconjunto de actividades compatibles  $A$  de tamaño máximo

# Problema Selección de Actividades

---

*Fijemos los elementos de la técnica*

- *Candidatos a seleccionar:* Conj. Actividades,  $C$
- *Candidatos seleccionados:* Conjunto  $S$ , inic.  $S=\{\emptyset\}$
- *Función Solución:*  $C=\{\emptyset\}$ .
- *Función Selección:* determina el mejor candidato,  $x$ 
  - Mayor, menor duración.
  - Menor solapamiento.
  - Termina antes..
- *Función de Factibilidad:*  $x$  es factible si es compatible con las actividades en  $S$ .
- *Función Objetivo:* Tamaño de  $S$ .

# Problema Selección de Actividades

---

## *Algoritmo Greedy*

- *SelecciónActividades(Activ C, S)*
  - Ordenar  $C$  en orden creciente de tiempo de finalización
  - Seleccionar la primera actividad.
  - Repetir
    - Seleccionar la siguiente actividad en el orden que comience despues de que la actividad previa termine.
  - Hasta que  $C$  este vacio.;

# Problema Selección de Actividades

---

*SelecciónActividadesGreedy(C, S)*

```
{qsort(C,n); // según tiempo de finalización
S[0]= C[0] // Seleccionar primera actividad
i=1; prev = 1;
while (!C.empty()) { // es solucion(C)
    x = C[i] // seleccionar;
    if (x.inicio > S[prev-1].fin) //factible x
        S[prev++] = x; // insertamos en solucion
    i++; // miramos siguiente
}
}
```

# Problema Selección de Actividades

---

- *¿Optimalidad?*

*T.H. CORMEN, C.E. LEISERSON, R.L. RIVEST.  
Introduction to Algorithms. The MIT Press (1992)*

# Ejemplo: Almacenamiento Optimal en Cintas

---

- Tenemos  $n$  programas que hay que almacenar en una cinta de longitud  $L$ .
- Cada programa  $i$  tiene una longitud  $l_i$ ,  $1 \leq i \leq n$
- Todos los programas se recuperan del mismo modo, siendo el tiempo medio de recuperación (TMR),

$$(1/n) \sum_{1 \leq j \leq n} t_j \quad t_j = \sum_{1 \leq k \leq j} l_{i_k}$$

- Nos piden encontrar una permutación de los  $n$  programas tal que cuando esten almacenados en la cinta el TMR sea mínimo.
- Minimizar el TMR es equivalente a minimizar

$$D(I) = \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq j} l_{i_k}$$

# Ejemplo: Almacenamiento Optimal en Cintas

---

El problema se puede resolver mediante la técnica greedy

Sea  $n = 3$  y  $(l_1, l_2, l_3) = (5, 10, 3)$

$((espera1) + (espera1 + espera2) + (espera1 + espera2 + espera3))$

Orden I	D(I)	
■ 1,2,3	$(5) + (5 + 10) + (5 + 10 + 3)$	= 38
■ 1,3,2	$(5) + (5 + 3) + (5 + 3 + 10)$	= 31
■ 2,1,3	$(10) + (10 + 5) + (10 + 5 + 3)$	= 43
■ 2,3,1	$(10) + (10 + 3) + (10 + 3 + 5)$	= 41
■ 3,1,2	$(3) + (3 + 5) + (3 + 5 + 10)$	= 29
■ 3,2,1	$(3) + (3 + 10) + (3 + 10 + 5)$	= 34

# Ejemplo: Almacenamiento Optimal en Cintas

---

Partiendo de la cinta vacía

**for i := 1 to n do**

**grabar el siguiente programa mas corto  
    ponerlo a continuacion en la cinta**

El algoritmo escoge lo más inmediato y mejor sin tener en cuenta si esa decisión será la mejor a largo plazo.

# Ejemplo: Almacenamiento Optimal en Cintas

---

## Teorema

Si  $I_1 \leq I_2 \leq \dots \leq I_n$  entonces el orden de colocación  $i_j = j$ ,  $1 \leq j \leq n$  minimiza

$$\sum_{k=1}^n \sum_{j=1}^k l_{i_j}$$

sobre todas las posibles permutaciones de  $i_j$

### ■ *Optimalidad?*

Ver la demostración en Horowitz-Sahni

# Ejemplo: Problema de la mochila fraccional

---

- Consiste en llenar una mochila:
  - Puede llevar como máximo un peso  $P$
  - *Hay n distintos posibles* objetos i fraccionables cuyos pesos son  $p_i$  y el beneficio por cada uno de esos objetos es de  $b_i$
- El *objetivo* es maximizar el beneficio de los objetos transportados.

$$\text{maximizar} \sum_{1 \leq i \leq n} x_i b_i \quad \text{sujeto a} \sum_{1 \leq i \leq n} x_i p_i \leq P$$

$x_i \in [0,1]$  representa la porción del objeto incluida en la mochila

# Ejemplo: Mochila 0/1

---



Es un claro problema de tipo greedy

Sus aplicaciones son innumerables

La técnica greedy produce soluciones óptimas para este tipo de problemas cuando se permite fraccionar los objetos

# Ejemplo: Mochila 0/1

---



¿Cómo seleccionamos los ítems?

# Ejemplo: Problema de la mochila fraccional

---

## Ejemplo

$p_i$	10	20	30	40	50
$b_i$	20	30	65	40	60

**n= 5, P=100**

Opciones:

¿Poner primero los más pesados?

¿Primero los que tienen más valor?

# Ejemplo: Problema de la mochila fraccional

---

- Supongamos 5 objetos de peso y precios dados por la tabla, la capacidad de la mochila es 100.

Precio (euros)	20	30	65	40	60	
Peso (kilos)	10	20	30	40	50	

- **Metodo 1 elegir primero el menos pesado**
  - Peso total =  $10 + 20 + 30 + 40 = 100$
  - Beneficio total =  $20 + 30 + 65 + 40 = 155$
- **Metodo 2 elegir primero el mas caro**
  - Peso Total =  $30 + 50 + 20 = 100$
  - Beneficio Total =  $65 + 60 + 20 = 145$

# Solucion Greedy

---

- Definimos la densidad del objeto  $A_i$  por  $b_i/p_i$ .
- Se usan objetos de tan alta densidad como sea posible, es decir, los seleccionaremos en orden decreciente de densidad.
- Si es posible se coge todo lo que se pueda de  $A_i$ , pero si no se rellena el espacio disponible de la mochila con una fraccion del objeto en curso, hasta completar la capacidad, y se desprecia el resto.
- Se ordenan los objetos por densidad no creciente, i.e.:  
$$b_i/p_i \geq b_{i+1}/p_{i+1} \text{ para } 1 \leq i \leq n.$$

# Ejemplo: Problema de la mochila fraccional

- Metodo 3 elegir primero el que tenga mayor valor por unidad de peso (razon costo/ peso)

Precio (euros)	20	30	65	40	60
Peso (Kilos)	10	20	30	40	50
Precio/Peso	2	1,5	2,1	1	1,2



- Peso Total =  $30 + 10 + 20 + 40 = 100$
- Costo Total =  $65 + 20 + 30 + 48 = 163$

# Índice

---

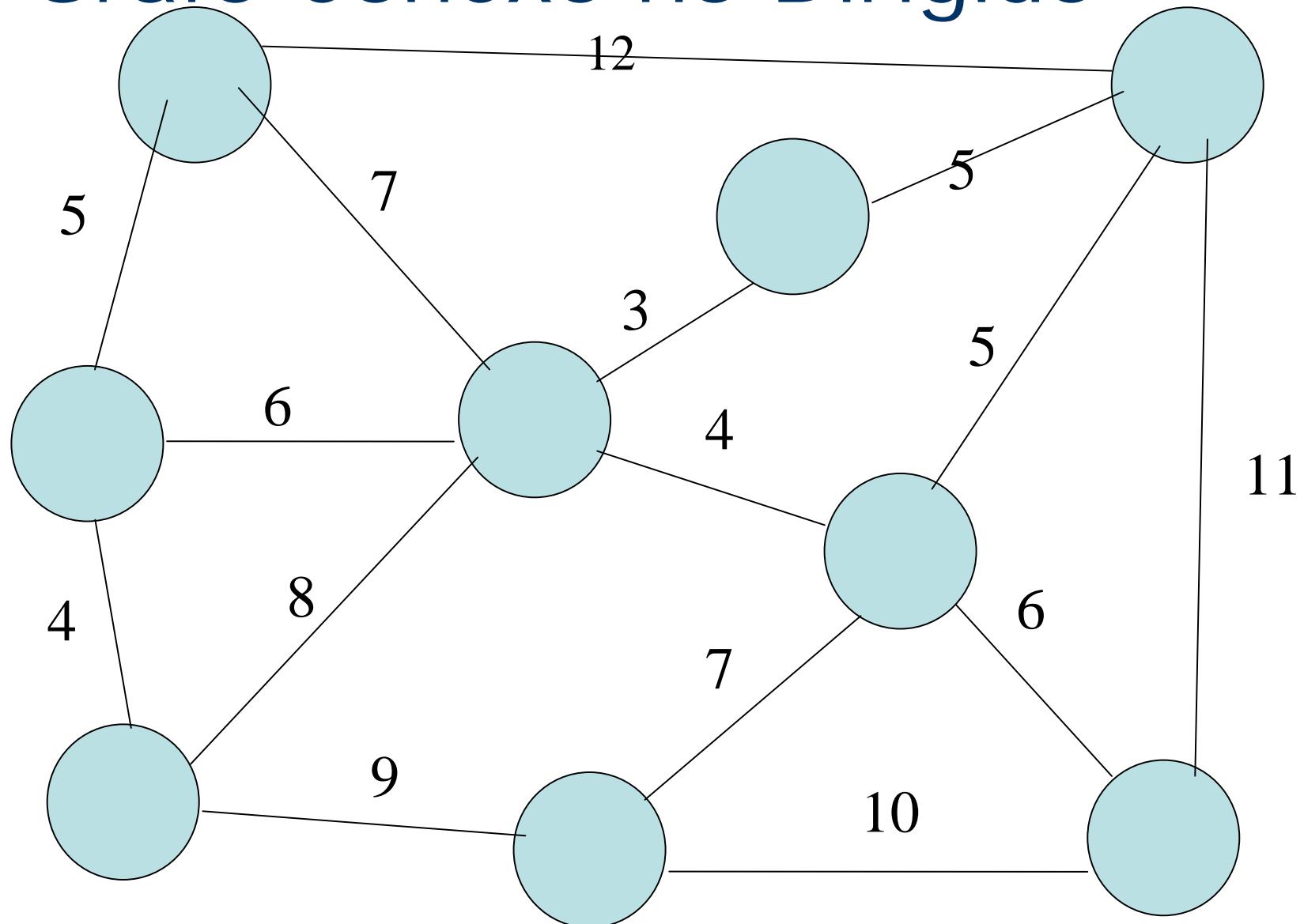
- **EL ENFOQUE GREEDY**
- **ALGORITMOS GREEDY EN GRAFOS**
  - Arboles de Recubrimiento Mínimo (Generadores Minimales)
    - Algoritmo de Kruskal
    - Algoritmo de Prim
  - Caminos Mínimos
    - Algoritmo de Dijkstra
- **HEURÍSTICA GREEDY**

# Arbol generador minimal

---

- Sea  $G = (V, A)$  un grafo conexo no dirigido, ponderado con pesos positivos. Calcular un subgrafo conexo tal que la suma de las aristas seleccionadas sea mínima.
- Este subgrafo es necesariamente un árbol: **árbol generador minimal o árbol de recubrimiento mínimo (ARM)** (en inglés, minimum spanning tree)
- Aplicaciones:
  - Red de comunicaciones de mínimo coste
  - Refuerzo de líneas críticas con mínimo coste
- Dos enfoques para la solución:
  - Basado en aristas: algoritmo de Kruskal
  - Basado en vértices: algoritmo de Prim

# Grafo Conexo no Dirigido



# Algoritmo de Kruskal

---

- Conjunto de candidatos: aristas
- **Función Solución:** un conjunto de aristas que conecta todos los vértices

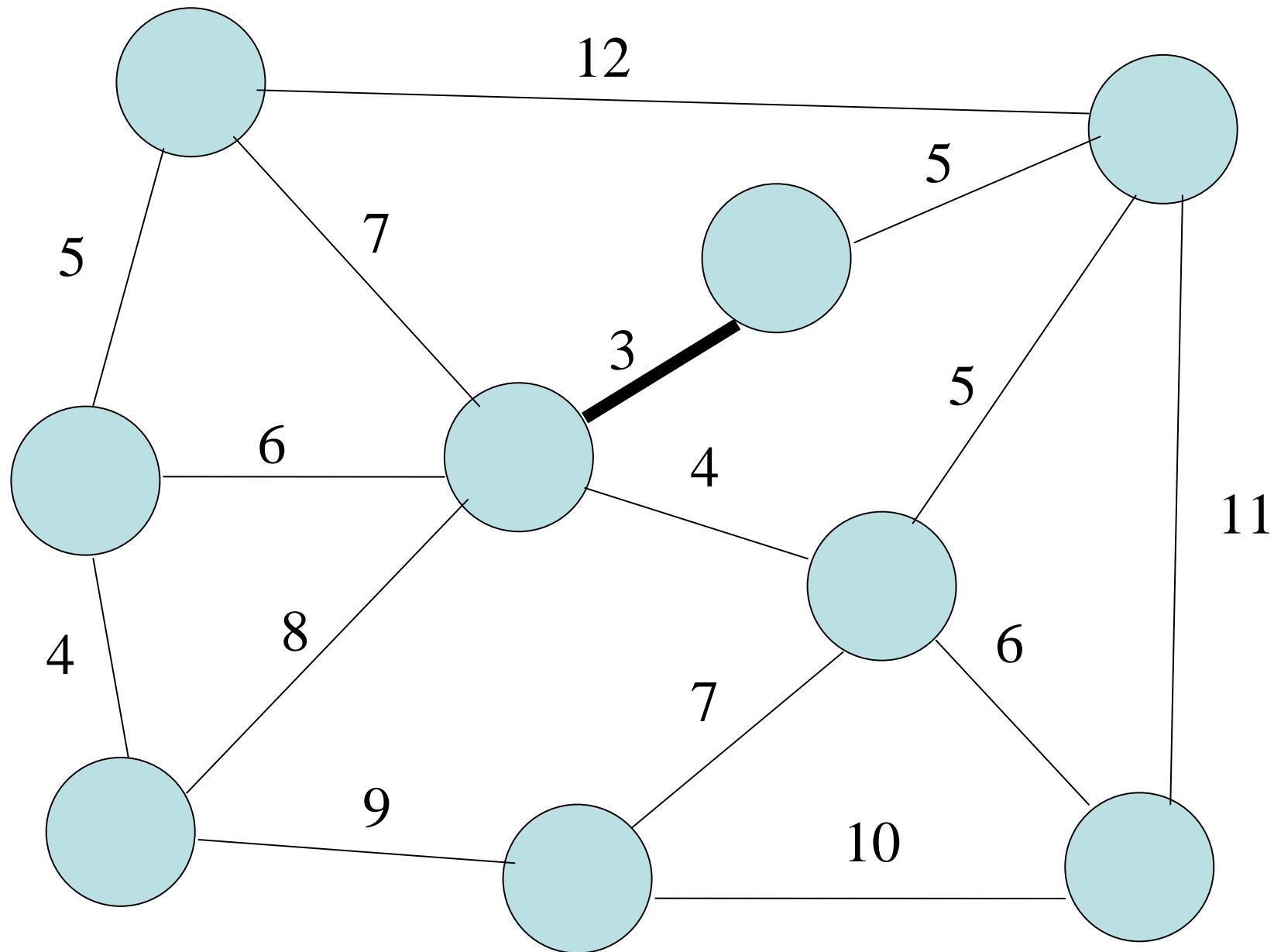
Se ha construido un árbol de recubrimiento ( $n-1$  aristas seleccionadas).

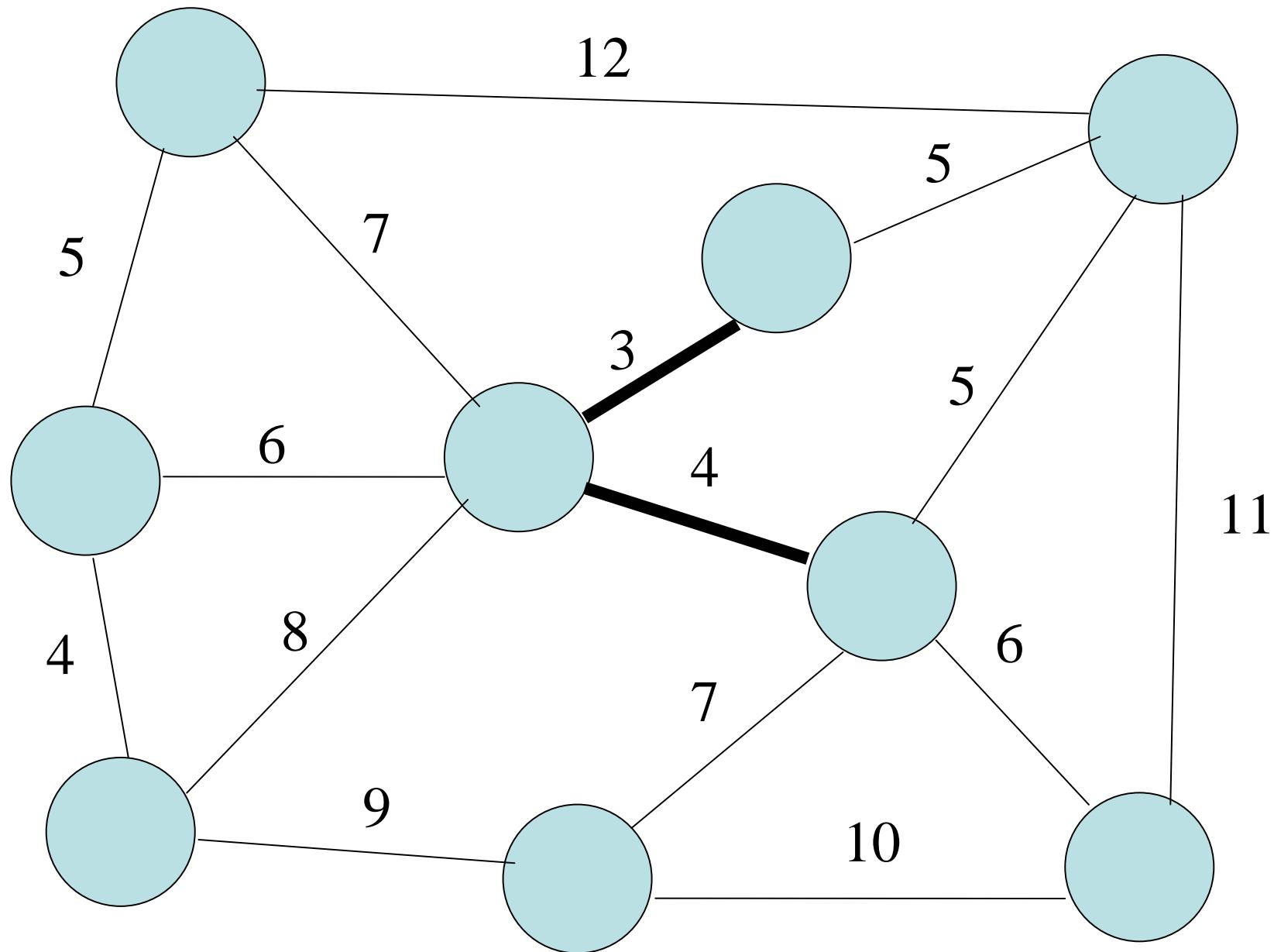
- **Función factible:** el conjunto de aristas no forma ciclos
- **Función selección:** la arista de menor coste
- **Función objetivo:** suma de los costes de las aristas
- ***Conjunto prometedor:*** se puede extender para producir una solución óptima

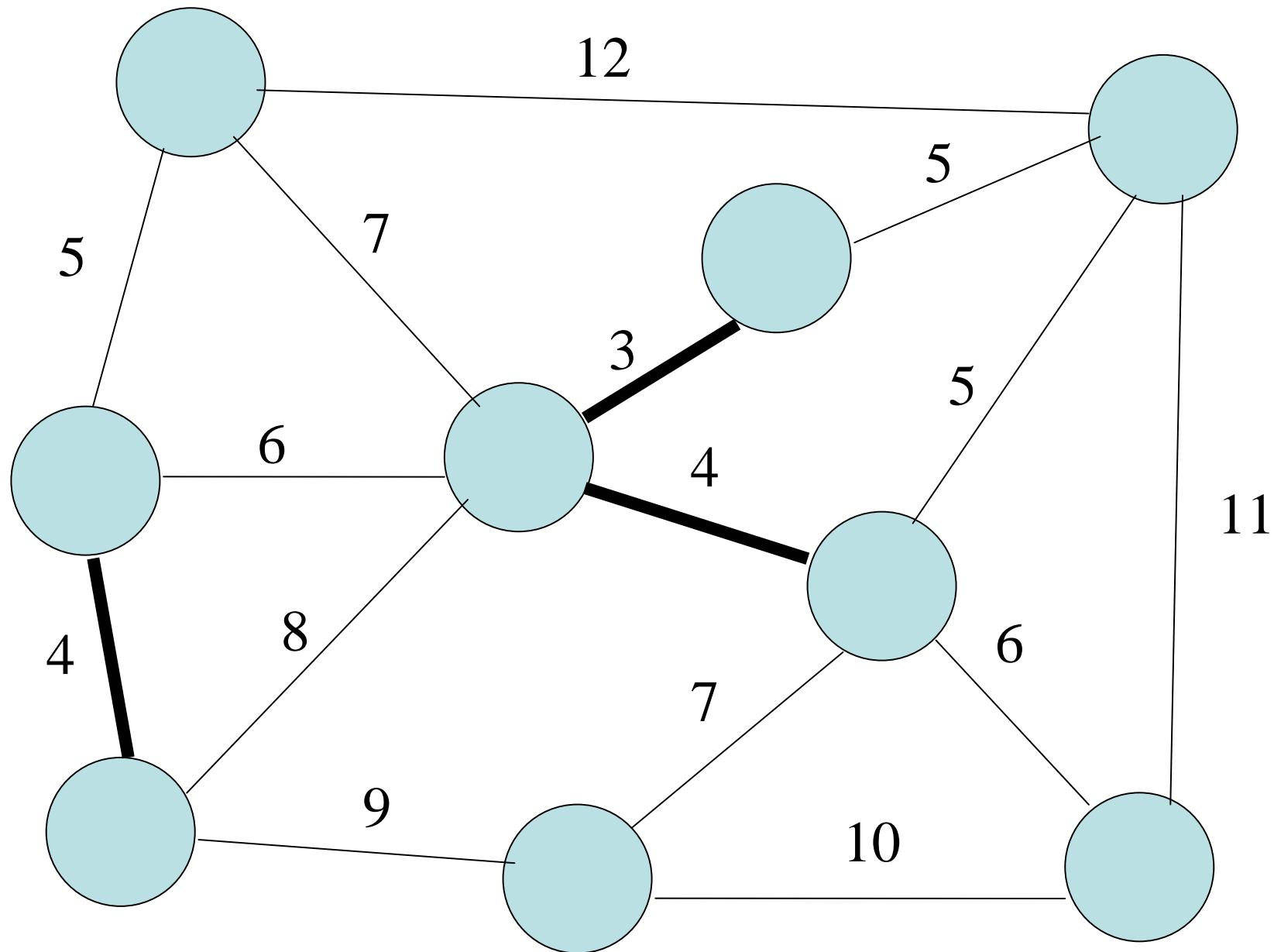
# Algoritmo de Kruskal

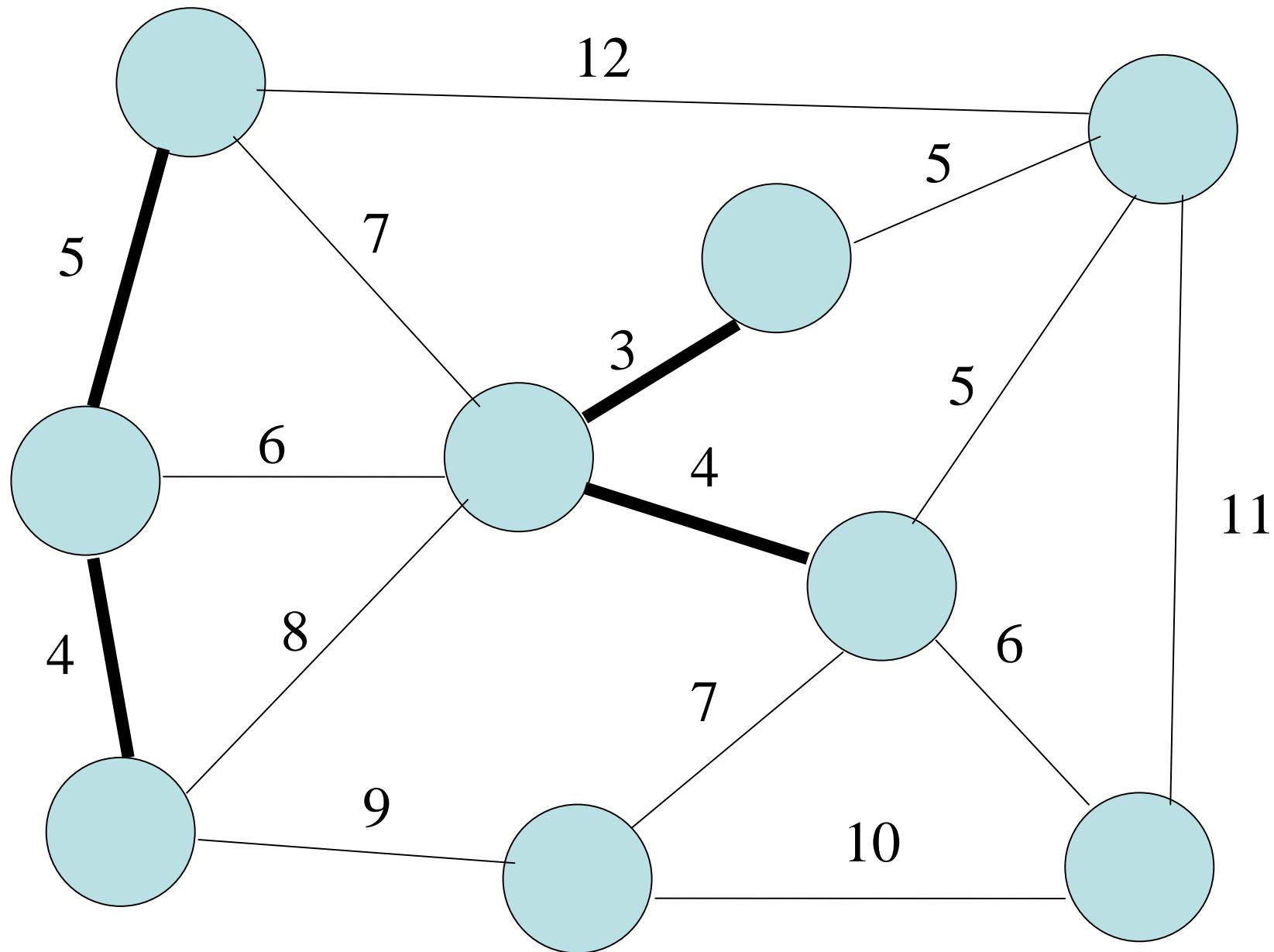
---

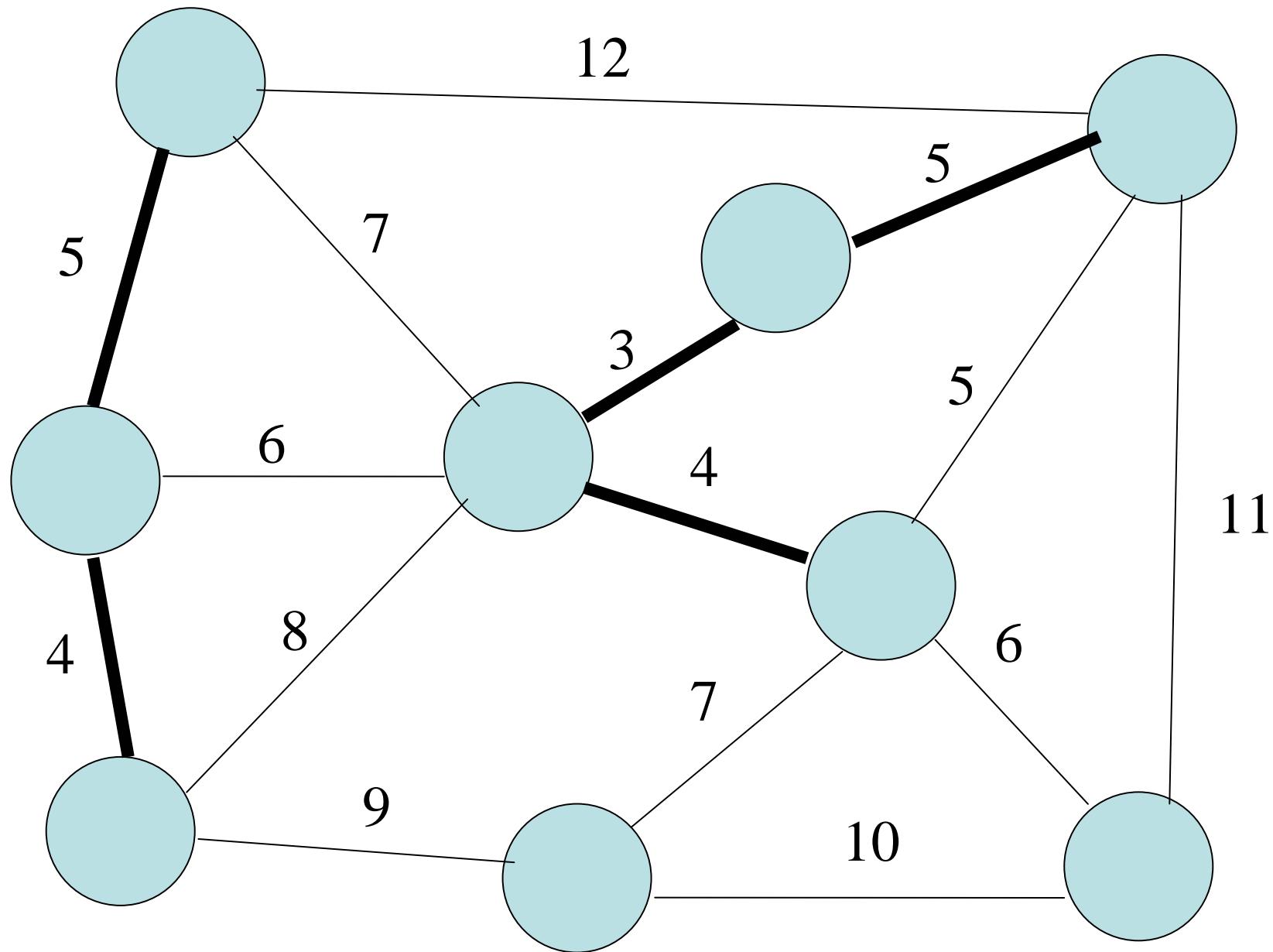
```
Funcion Kruskal_I(Grafo G(V,A))
{set<arcos> C(A);
 set<arcos> S;           // Solución inic. Vacía
 Ordenar(C);
while (!C.empty() && S.size()!=V.size()-1) { //No solución
    x = C.first(); //seleccionar el menor
    C.erase(x);
    if (!HayCiclo(S,x)) //Factible
        S.insert(x);
}
if (S.size()==V.size()-1) return S; // Hay solución (para grafos
else return "No_hay_solucion"; // dirigidos)
}
```

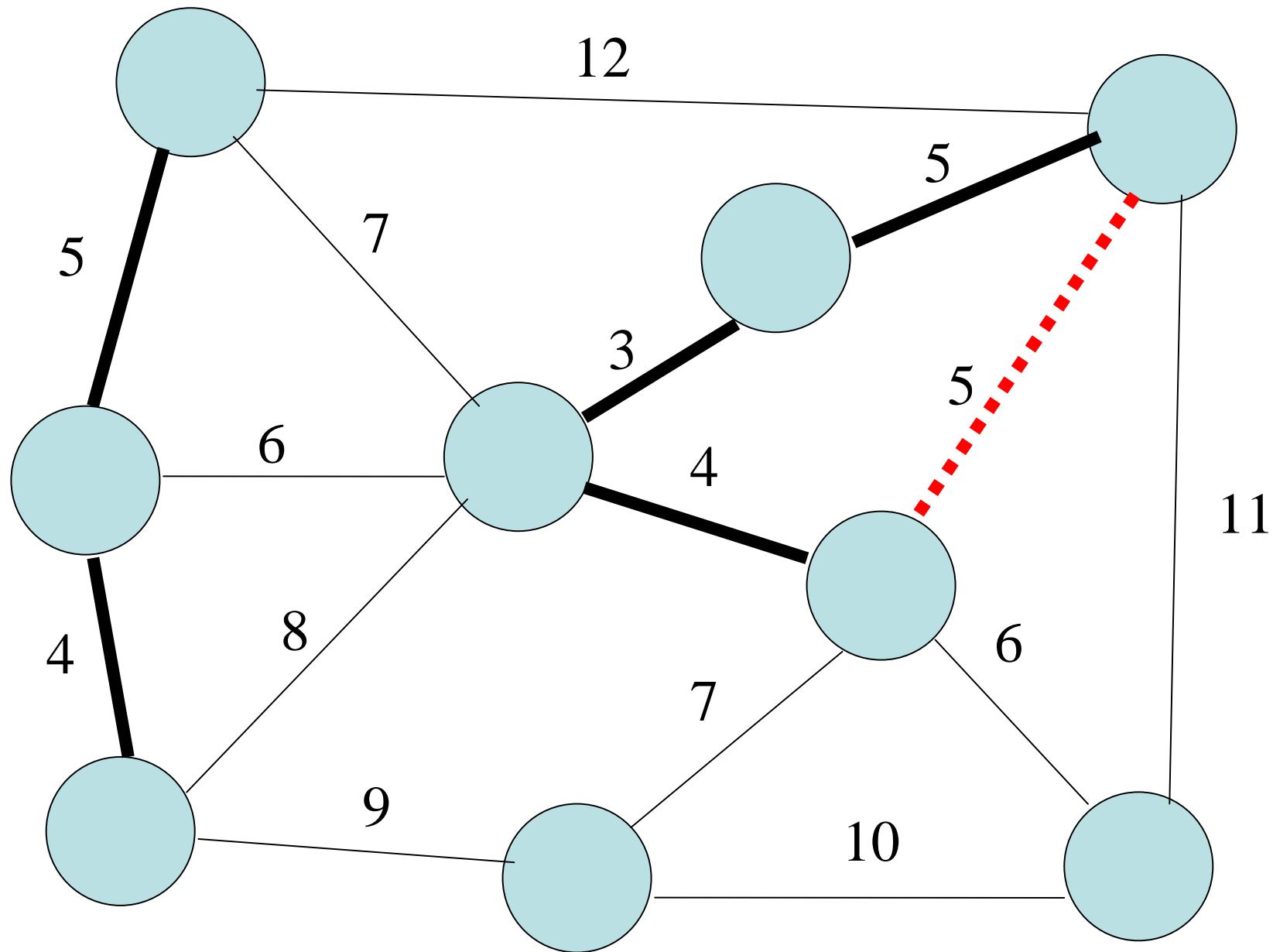


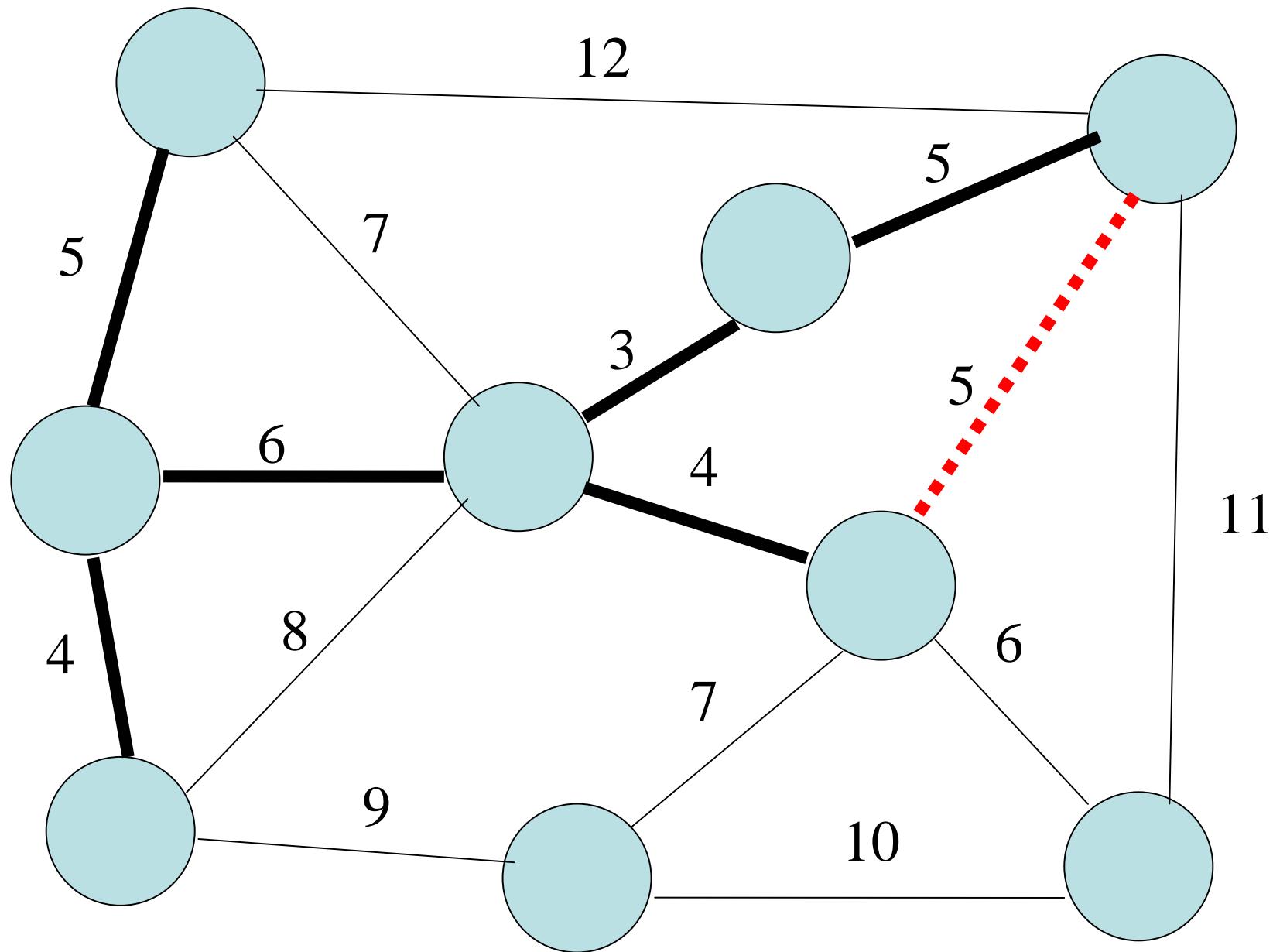


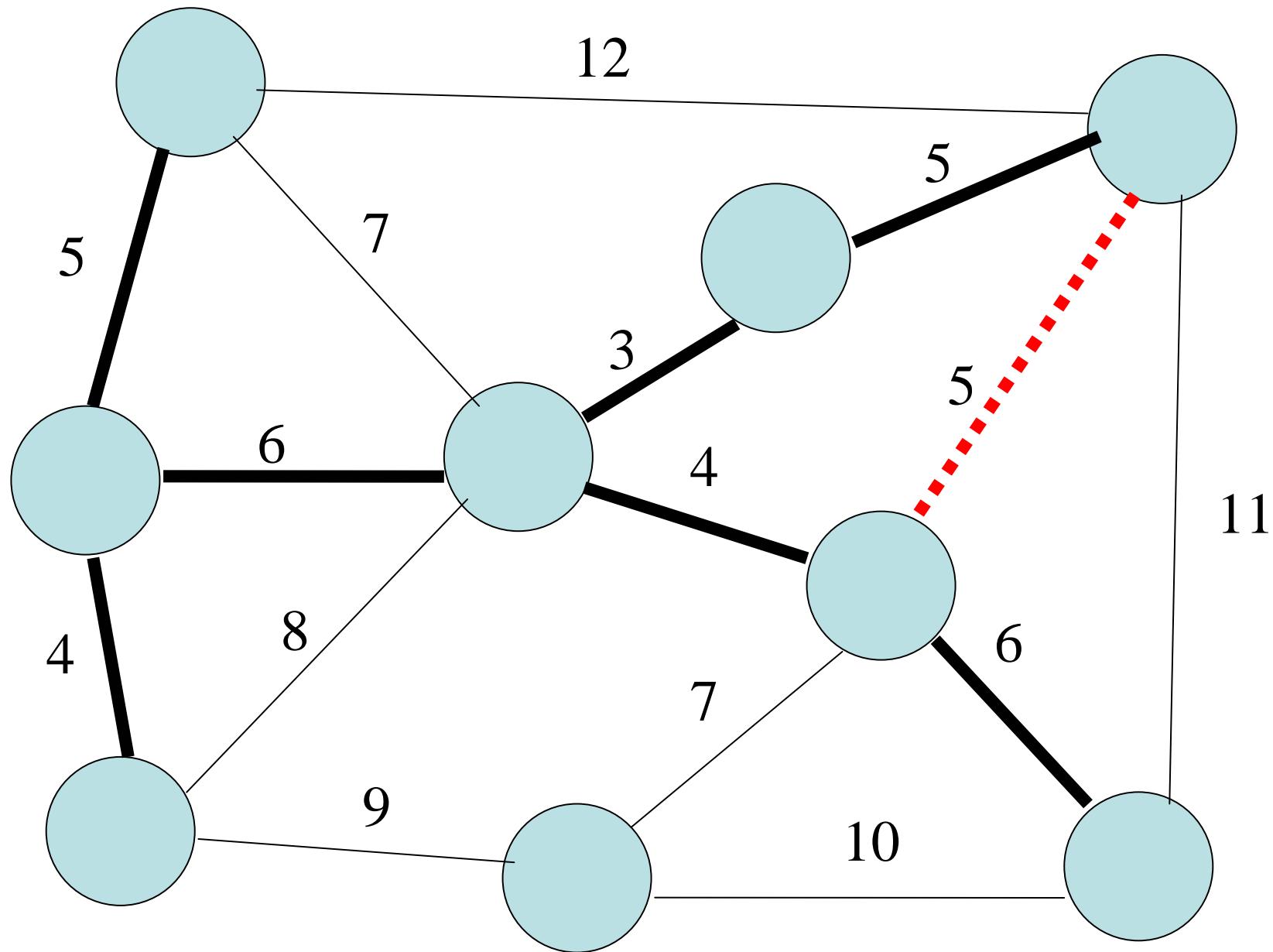


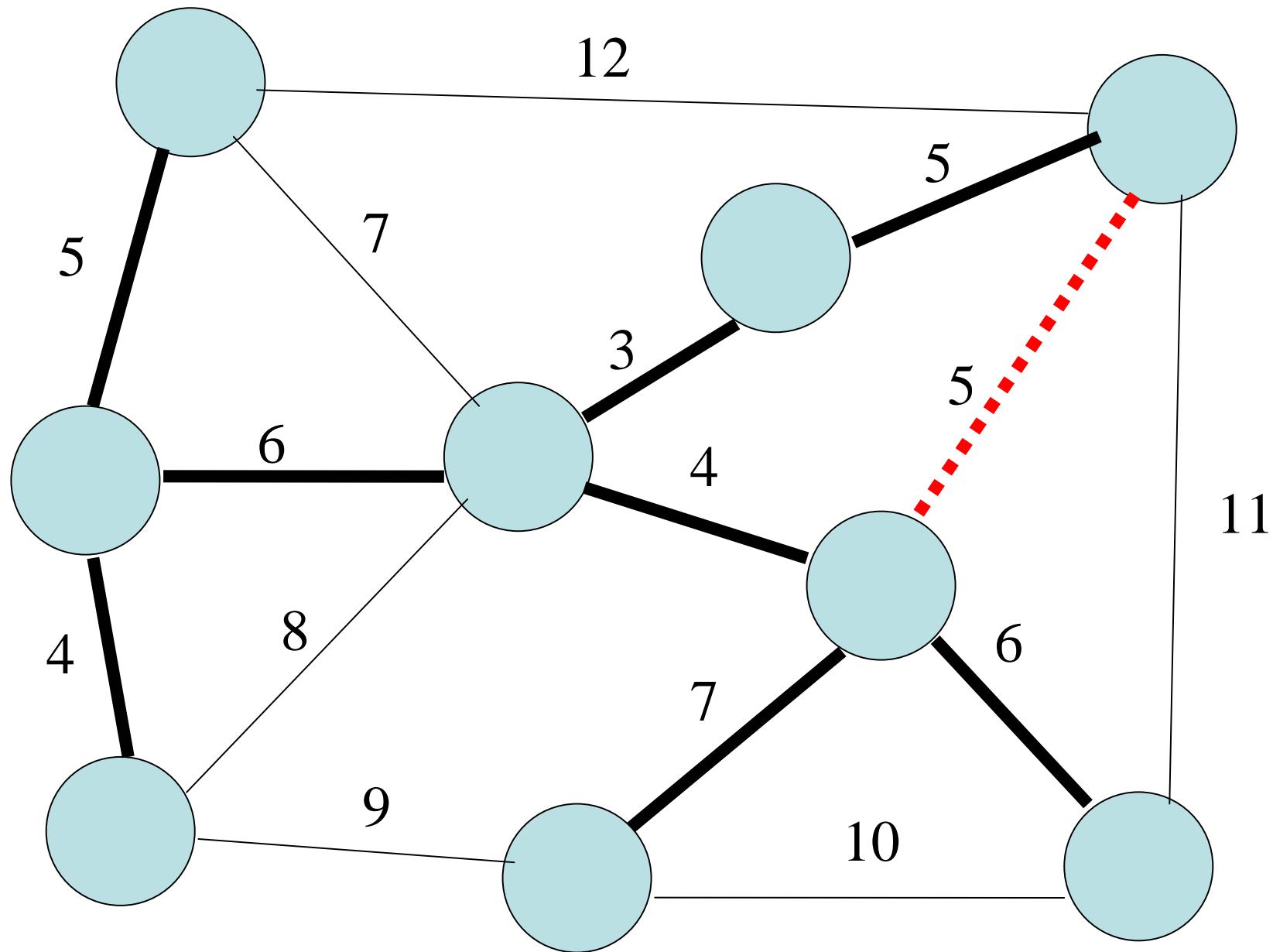


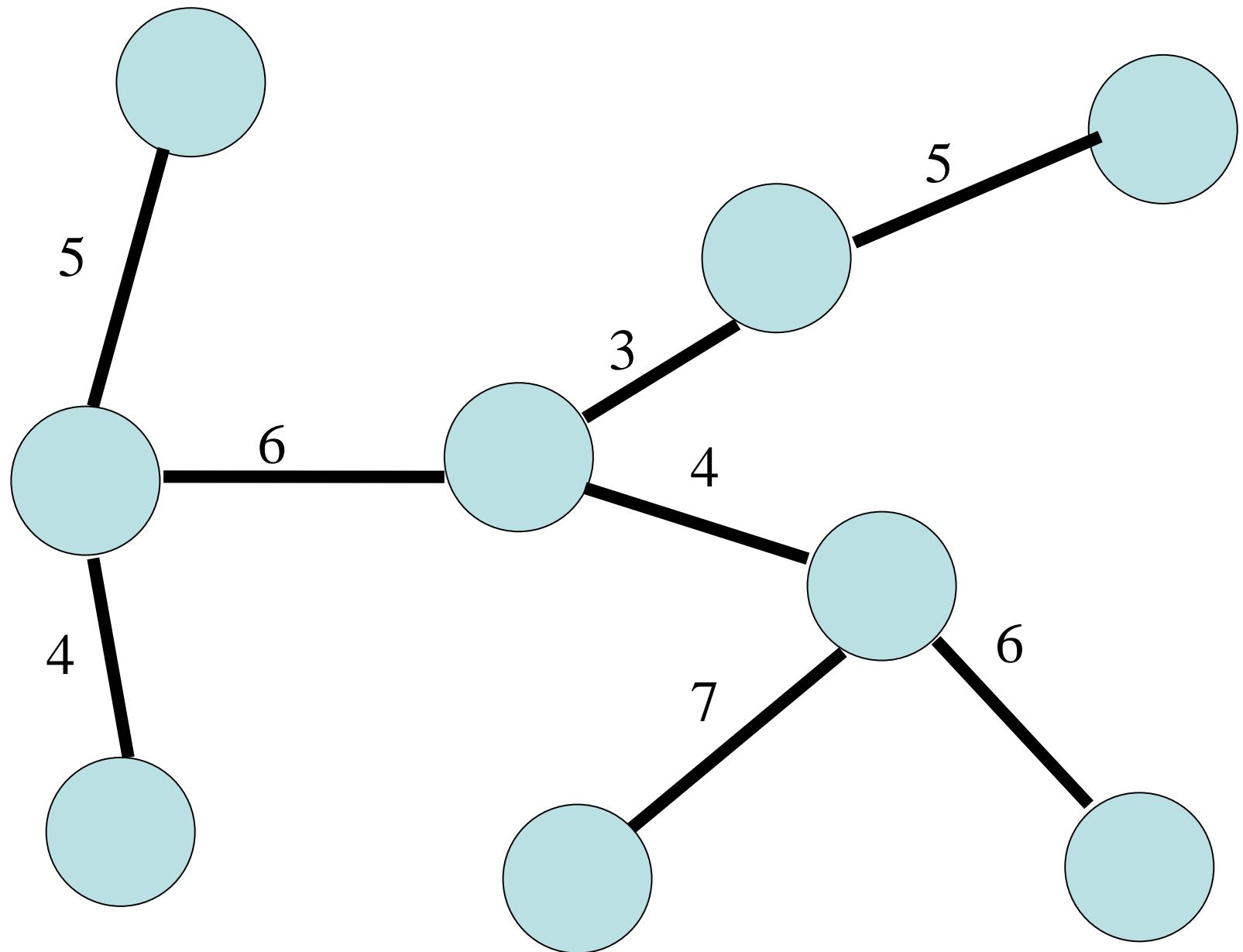












# Eficiencia Kruskal (Directa)

---

```
Funcion Kruskal_I(Grafo G(V,A))
{ vector<arcos> C(A);
  Arbol<arcos> S;           // Solución inic. Vacía
  Ordenar(C);                O(A log A) = O(A log V) xq?
  while (!C.empty() && S.size()!=V.size()-1) { O(1)
    x = C.first(); //seleccionar el menor      O(1)
    C.erase(x);          O(1)
    if (!HayCiclo(S,x)) //Factible        O(1)
      S.insert(x);        O(1)
    }
    if (S.size()==V.size()-1) return S; // Hay solución
    else return "No_hay_solucion";
}
```

$O(AV)$

# Optimalidad de Kruskal (Directa)

---

## Teorema:

*El algoritmo de Kruskal halla un árbol de recubrimiento mínimo*

## Demostración: Inducción sobre el número de aristas que se han incluido en el ARM.

Base. Sea  $k$  la arista de menor peso en  $A$  (la primera propuesta por Kruskal), entonces  $\exists$  un ARM óptimo  $T$  tal que  $\{k\} \in T$ .

*Suponemos cierto para  $i-1$ .*

*La arista  $(i-1)$ -ésima en incluirse por el algoritmo de Kruskal pertenece a un ARM  $T$  óptimo.  $T'$  con las  $i-1$  aristas es un conjunto prometedor*

*Demostramos que es cierto para  $i$*

*La arista  $(i)$ -ésima incluida por el algoritmo de Kruskal tb. pertenece al ARM  $T$  óptimo. Llámemos a esta arista  $i$ . Segundo Kruskal es la arista de menor peso que une dos componentes conexas*

# Optimalidad de Kruskal (Directa)

---

## Demostración

Caso base: Red. Absurdo, supongamos un ARM  $T$  que no incluye a  $k$ . Consideremos  $T \cup k$  con

$$\text{peso}(T \cup k) = \text{peso}(T) + \text{peso}(k).$$

En este caso aparece un ciclo (*¿por qué?*). Eliminemos cualquier arista del ciclo,  $(x)$ , distinta de  $k$ . Al eliminar la arista obtenemos un árbol  $T^* = T + k - x$  con peso

$$\text{peso}(T^*) = \text{peso}(T) + \text{peso}(k) - \text{peso}(x).$$

Si tenemos  $\text{peso}(k) \leq \text{peso}(x)$  entonces deducimos que  $\text{peso}(T^*) \leq \text{peso}(T)$ . !!! Contr. o  $T^*$  tb. es óptimo

# Optimalidad de Kruskal (Directa)

---

Paso Inducción:

Supongamos un ARM  $T$  tal que  $T' \subseteq T$  y no incluye a  $i$ .

Consideremos  $T \cup i$  con

$$\text{peso}(T \cup i) = \text{peso}(T) + \text{peso}(i).$$

En este caso aparece un ciclo, que incluirá al menos una arista  $x$  que NO pertenece al conjunto de aristas  $T'$  seleccionadas por Kruskal (*¿por qué?*). Eliminando dicha arista del ciclo. obtenemos un nuevo árbol  $T^* = T + i - x$  con peso

$$\text{peso}(T^*) = \text{peso}(T) + \text{peso}(i) - \text{peso}(x).$$

Si sabemos  $\text{peso}(i) \leq \text{peso}(x)$  (*porqué?*) entonces deducimos que  $\text{peso}(T^*) \leq \text{peso}(T)$ . Puesto que  $T' \subseteq T$  y  $x \notin T'$  entonces  $T' \subseteq T^*$ , lo que implica que  $T' \cup i \subseteq T^*$  (que es óptimo)

# Algoritmo de Kruskal

---

¿Cómo determino que una arista no forma un ciclo?

Comienzo con un conjunto de componentes conexas de tamaño n (cada nodo es una componente conexa).

La función factible me acepta una arista (la de menor costo) si ésta permite unir dos componentes conexas. Así garantizamos que no hay ciclos.

En cada paso hay una componente conexa menos, finalmente termino con una única componente conexa que forma el arbol generador minimal.

# Algoritmo de Kruskal

---

Voraz( $G = (V, A)$ ;  $L: A \rightarrow R$ ) : conjunto aristas

Ordenar  $A$  por longitudes crecientes

$n = |V|$ ;  $T = \emptyset$

Iniciar  $n$  conjuntos  $\text{comp}_i$ , cada uno con un elemento  $v$  de  $V$

**repite**

$e = \{u, v\}$

$\text{comp}_u, \text{comp}_v$  los conjuntos en los que están  $u, v$

**si**  $\text{comp}_u \neq \text{comp}_v$  **entonces**

**fusionar**( $\text{comp}_u, \text{comp}_v$ )

$T = T \cup \{e\}$

**fin si**

**hasta**  $|T| = n - 1$

**Devolver**  $T$

# Implementación Kruskal (Eficiente)

---

```
Funcion Kruskal(G(V,A)){  
    S = 0; // Inicializamos S con el conjunto vacio  
    for (i=0; i< V.size()-1; i++)  
        MakeSet(V[i]); // Conjuntos con el vértice v[i]  
    Ordenar(A) //Orden creciente de pesos  
    while (!A.empty() && S.size()!=V.size()-1) { //No solución  
        (u,v) =A.first(); //seleccionar el menor arco (y eliminar)  
        if (FindSet(u) != FindSet(v))  
            S = S U {{u,v}};  
            Union(u, v); // Unimos los dos conjuntos  
    }  
}
```

# Análisis de Eficiencia

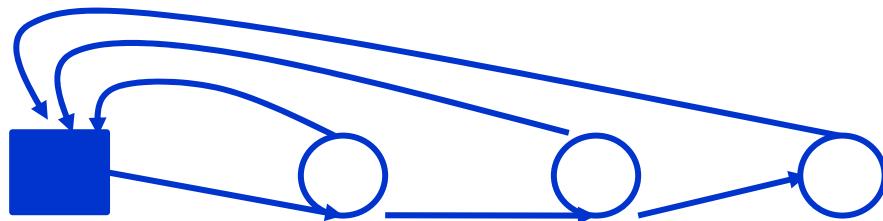
---

```
Funcion Kruskal(G(V,A)){  
    S = 0; // Inicializamos S con el conjunto vacio  
    for (i=0; i< V.size()-1; i++)  
        MakeSet(V[i]);      O(1)  
    Ordenar(A) //Orden creciente de pesos  O(A log V)  
    while (!A.empty() && S.size()!=V.size()-1) { //No solución  
        (u,v) =A.first(); O(1)  
        if (FindSet(u) != FindSet(v)) O(1)  
            S = S U {{u,v}}; O(1)  
            Union(u, v); // O(V) Es el cuello de botella, !!!!  
    }  
}
```

# Eficiencia del Algoritmo de Kruskal

---

- Representación para conjuntos disjuntos
  - Listas enlazadas de elementos con punteros hacia el conjunto al que pertenecen



- MakeSet(): O(1) FindSet(): O(1)
- Union(A,B): “Copia” elementos de A a B haciendo que los elementos de A tambien apunten a B: O(A)
- *¿Cuanto tardan en realizarse las n uniones?*

# Unión de conjuntos disjuntos

---

- Análisis del peor caso:  $O(n^2)$

$\text{Union}(S_1, S_2)$

"copia" 1 elemento

$\text{Union}(S_2, S_3)$

"copia" 2 elementos

$\text{Union}(S_{n-1}, S_n)$

"copia" n-1 elementos

$O(n^2)$

- Mejora: Siempre copiar el menor en el mayor

- *Peor caso* un elemento es copiado como máximo  $\log(n)$  veces => n Uniones es  $O(n \lg n)$
- Por tanto, el análisis amortizado dice que una unión es de  $O(\log n)$

# Análisis de Eficiencia

---

```
Funcion Kruskal(G(V,A)){  
    S = 0; // Inicializamos S con el conjunto vacio  
    for (i=0; i< V.size()-1; i++)  
        MakeSet(V[i]); O(1)  
    Ordenar(A) //Orden creciente de pesos O(A log V)  
    while (!A.empty() && S.size()!=V.size()-1) { //No solución  
        (u,v) = A.first(); O(1)  
        if (FindSet(u) != FindSet(v)) O(1)  
            S = S U {{u,v}}; O(1)  
            Union(u, v); // O(log V)  
    }  
}  
Kruskal es de O(A log V)
```

# Algoritmo de Prim

---

- Primero veremos que el problema ARM tiene subestructuras optimales
- Teorema : *Sea T un ARM y sea  $(u,v)$  una arista de T. Sean  $T_1$  y  $T_2$  los dos árboles que se obtienen al eliminar la arista  $(u,v)$  de T. Entonces  $T_1$  es un ARM de  $G_1 = (V_1, E_1)$ , y  $T_2$  es un ARM de  $G_2 = (V_2, E_2)$* 
  - Demostración: Simple, por red. absurdo  
Idea, partir de que
$$\text{peso}(T) = \text{peso}(u,v) + \text{peso}(T_1) + \text{peso}(T_2)$$
Por tanto, no puede haber arboles de recubrimiento mejores que  $T_1$  o  $T_2$ , pues si los hubiese T no sería solución óptima.

# Algoritmo de Prim

---

Se basa en el siguiente

## Teorema

- Sea  $T$  un ARM óptimal de  $G$ , con  $A \subseteq T$  un subárbol de  $T$  y sea  $(u, v)$  la arista de menor peso conectando los vértices de  $A$  con los de  $V-A$ . Entonces,  $(u, v) \in T$

Demostración: Se deja como ejercicio.

# Algoritmo de Prim

---

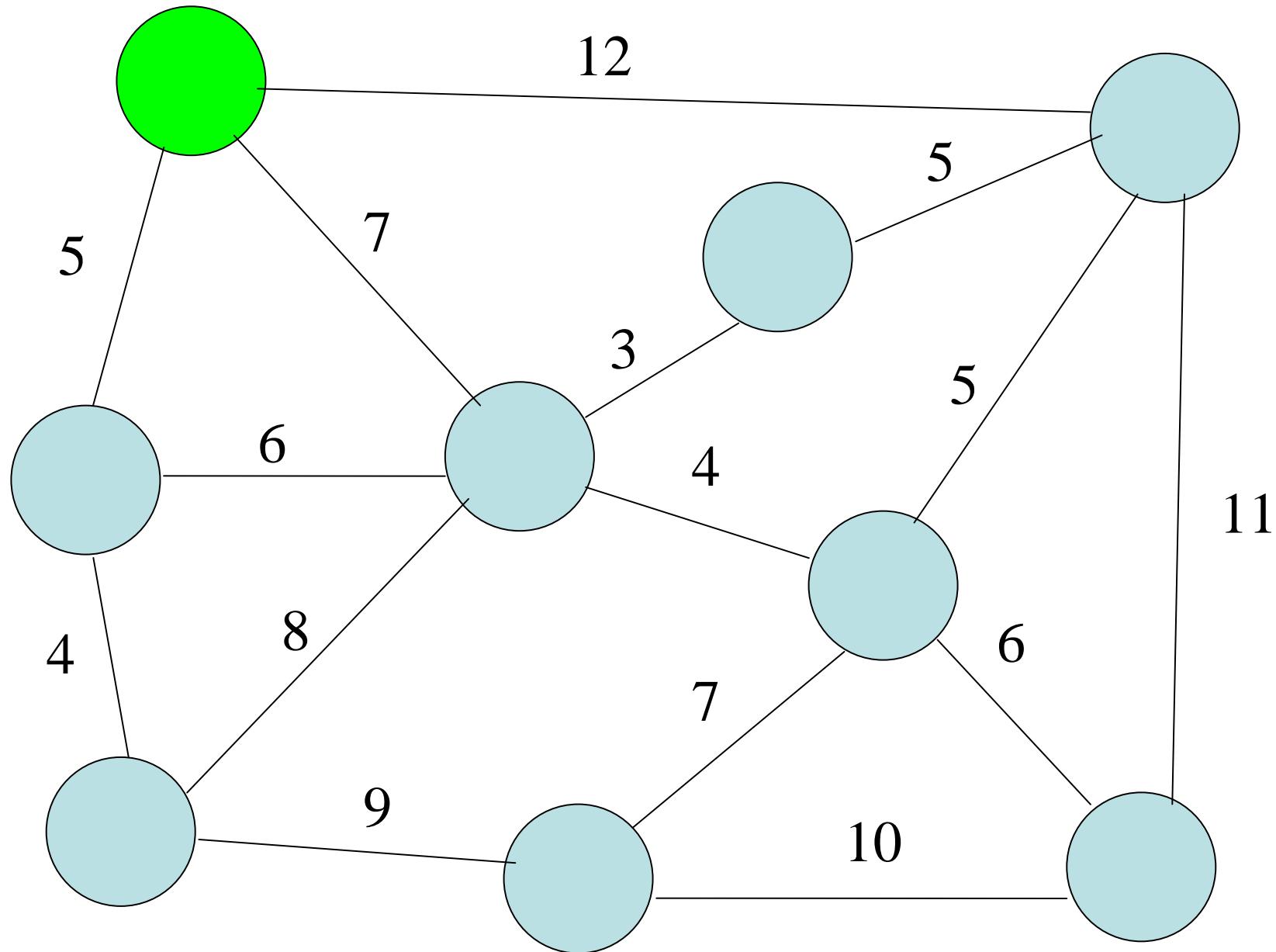
*Candidatos:* Vértices

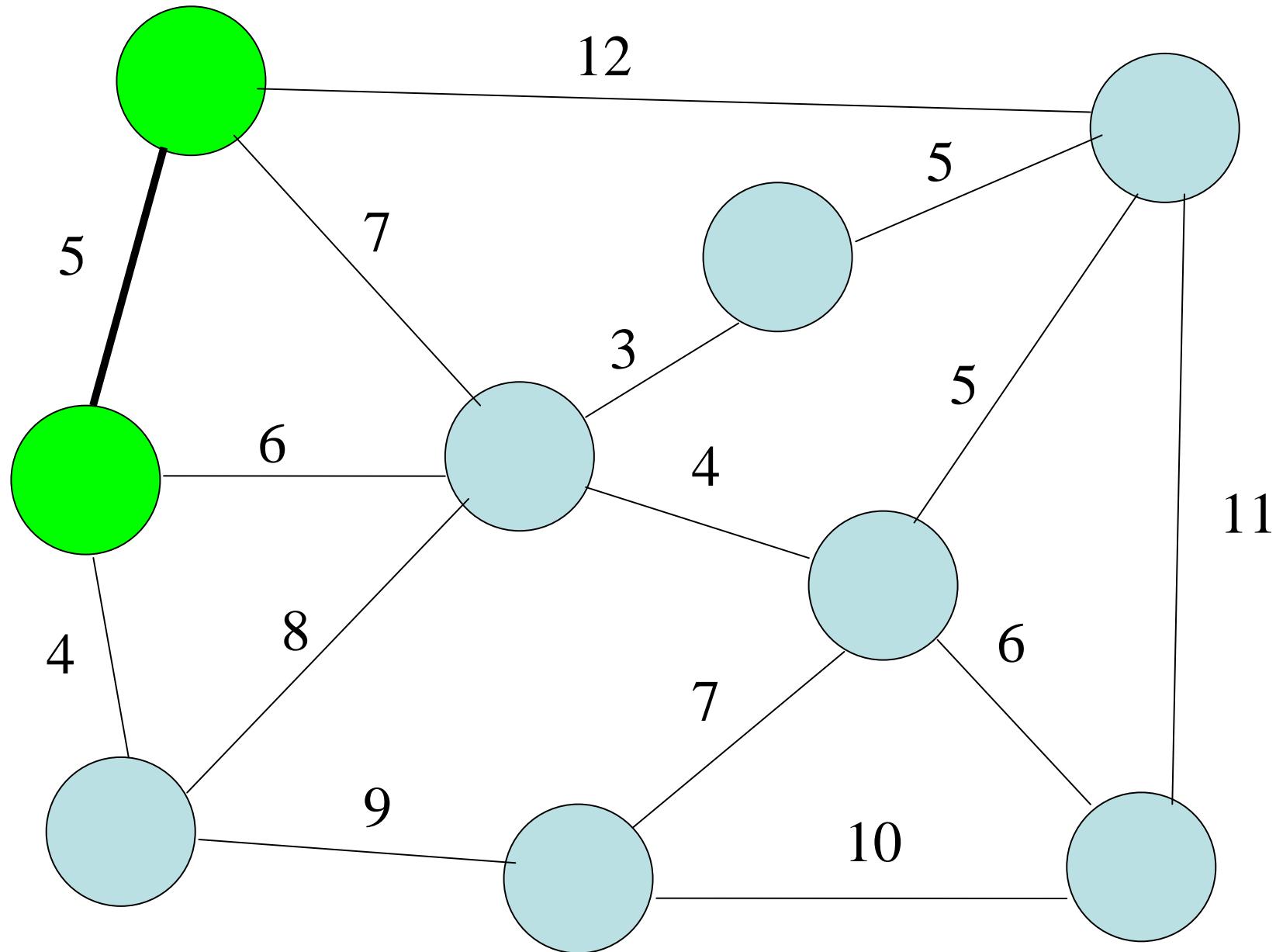
*Función Solución:* Se ha construido un árbol de recubrimiento (n vértices seleccionadas).

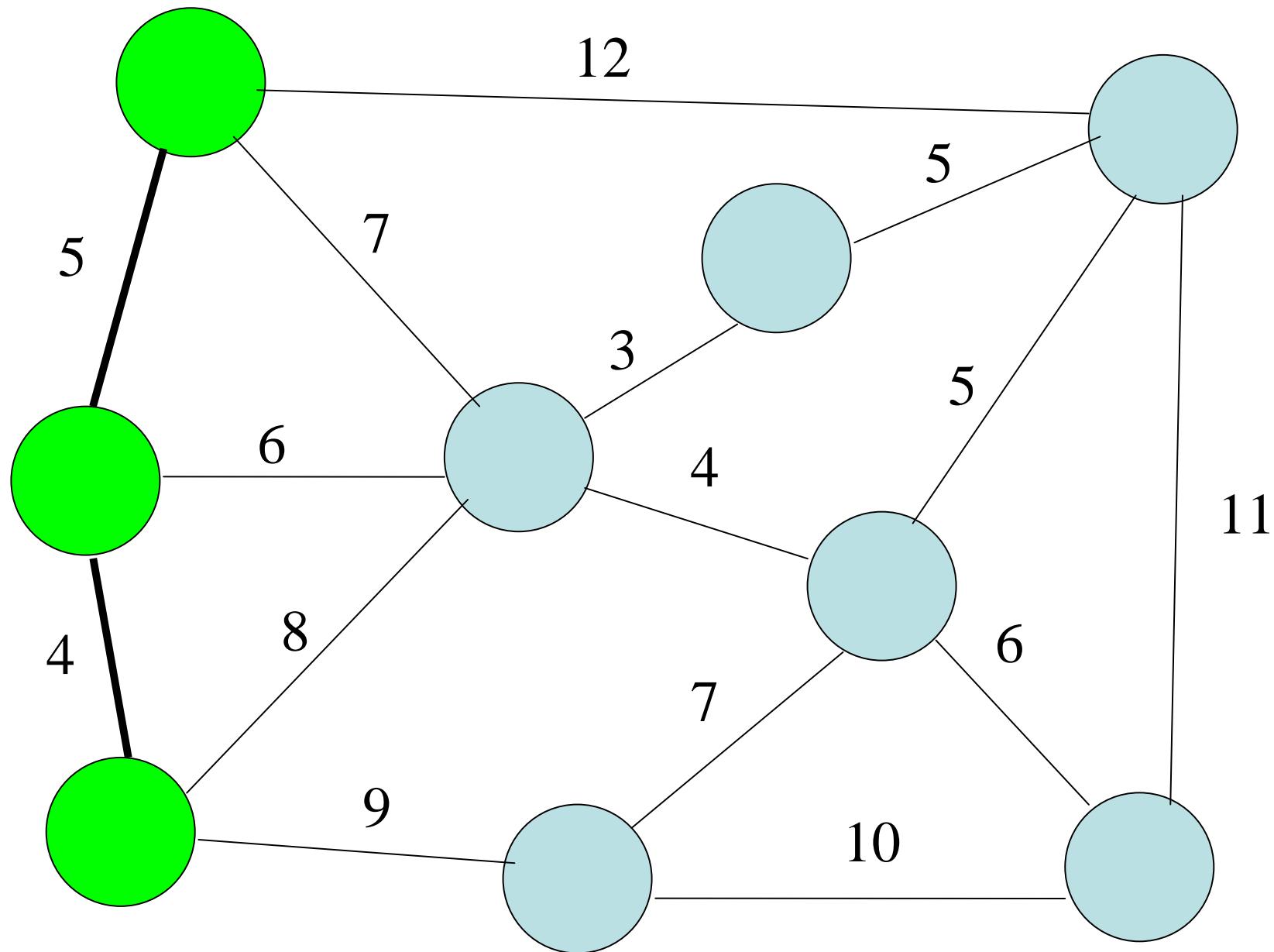
*Función Selección:* Seleccionar el vértice  $u$  del conjunto de no seleccionados que se conecte mediante la arista de menor peso a un vértice  $v$  del conjunto de vértices seleccionados. La arista  $(u,v)$  está en  $T$ .

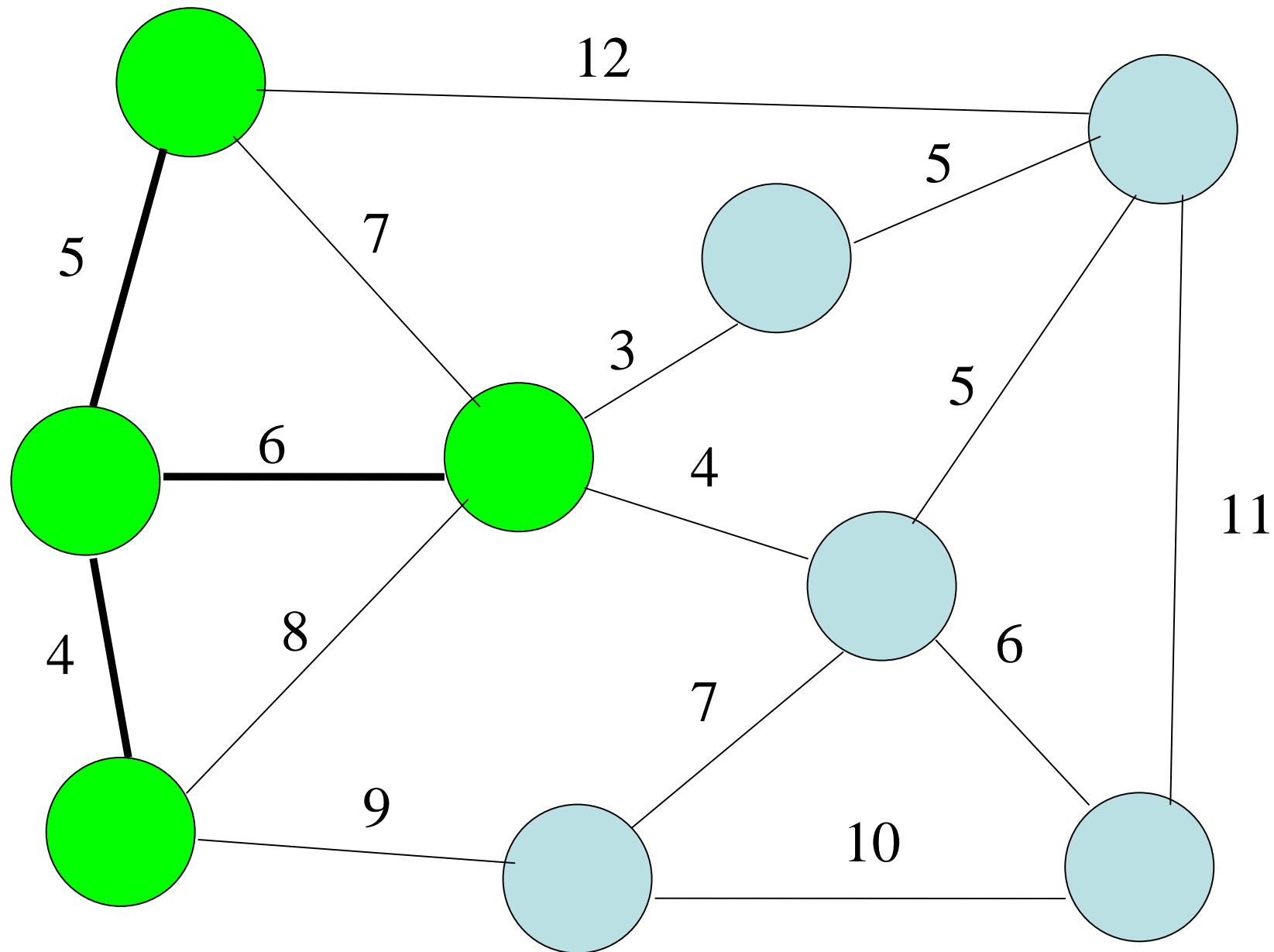
*Función de Factibilidad:* El conjunto de aristas no contiene ningún ciclo. Está implícita en el proceso

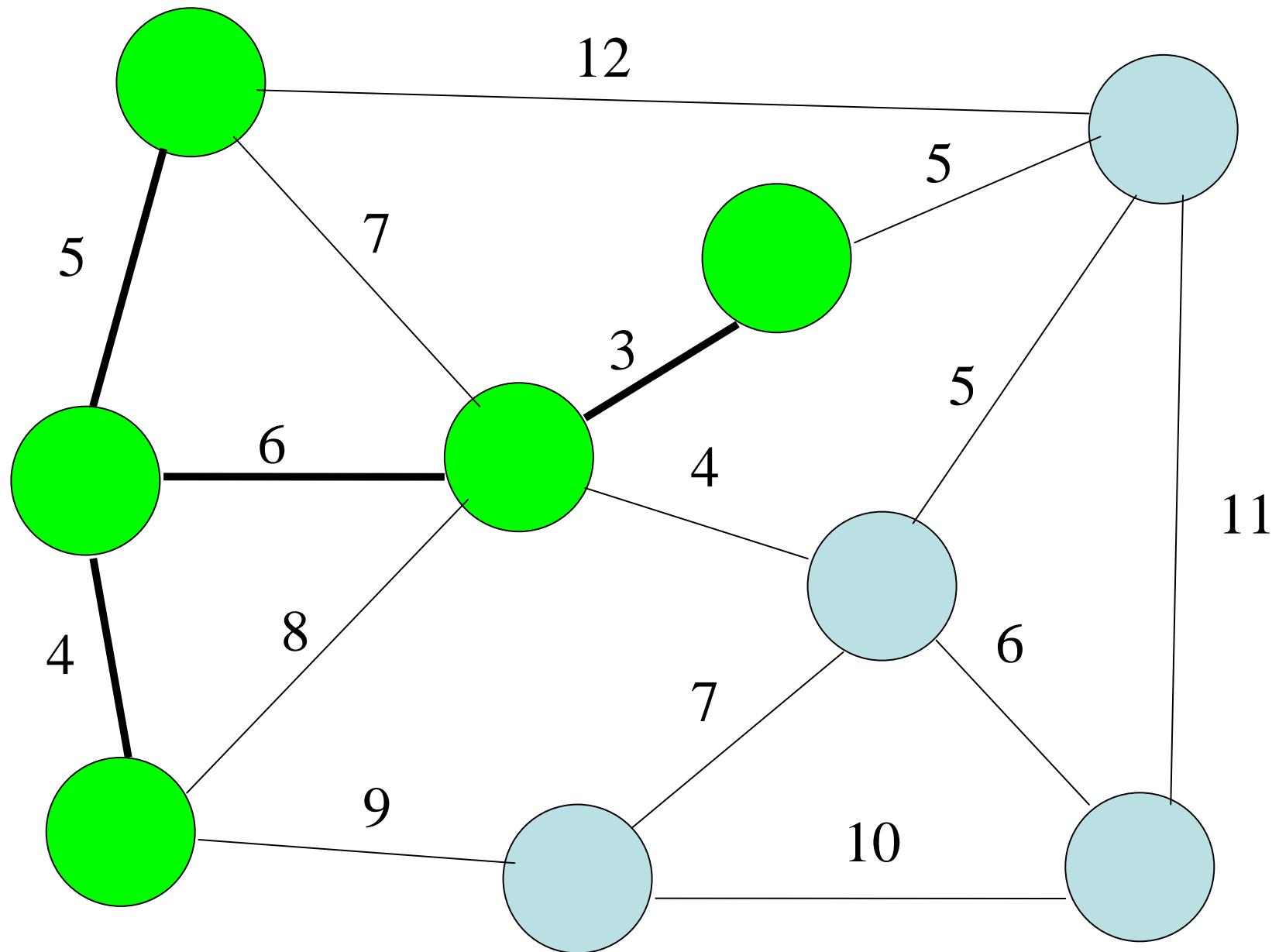
*Función Objetivo:* determina la longitud total de las aristas seleccionadas.

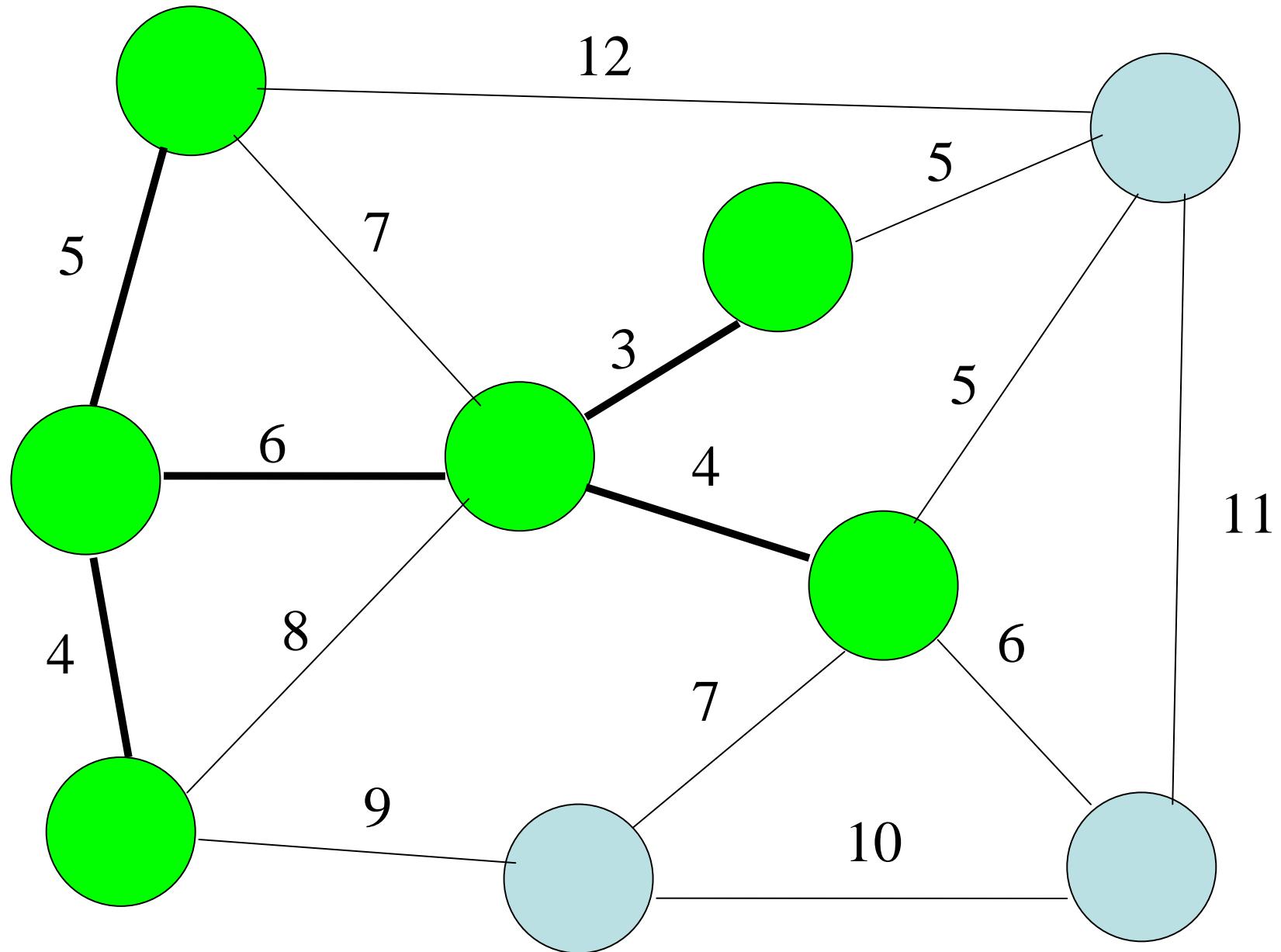


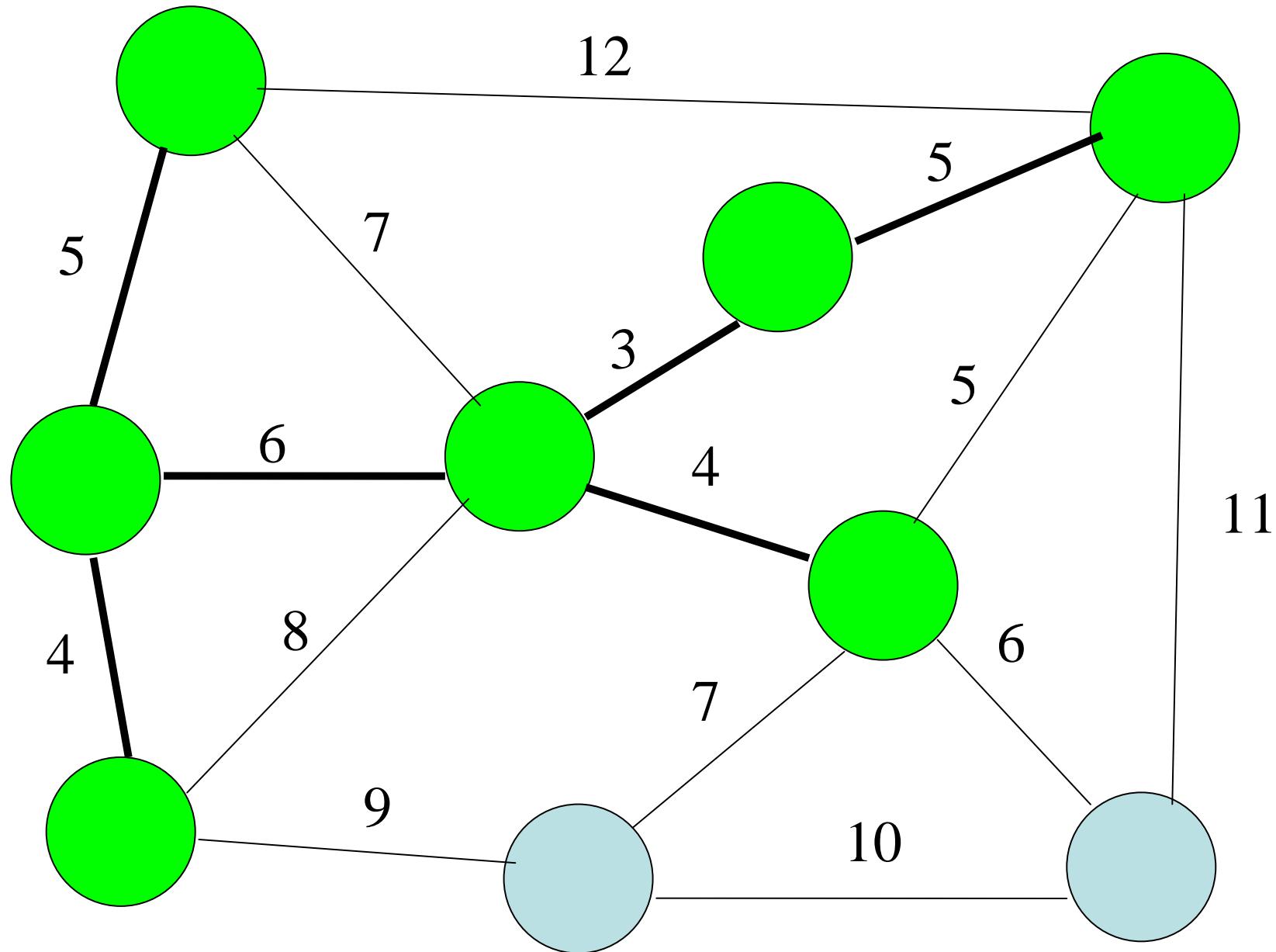


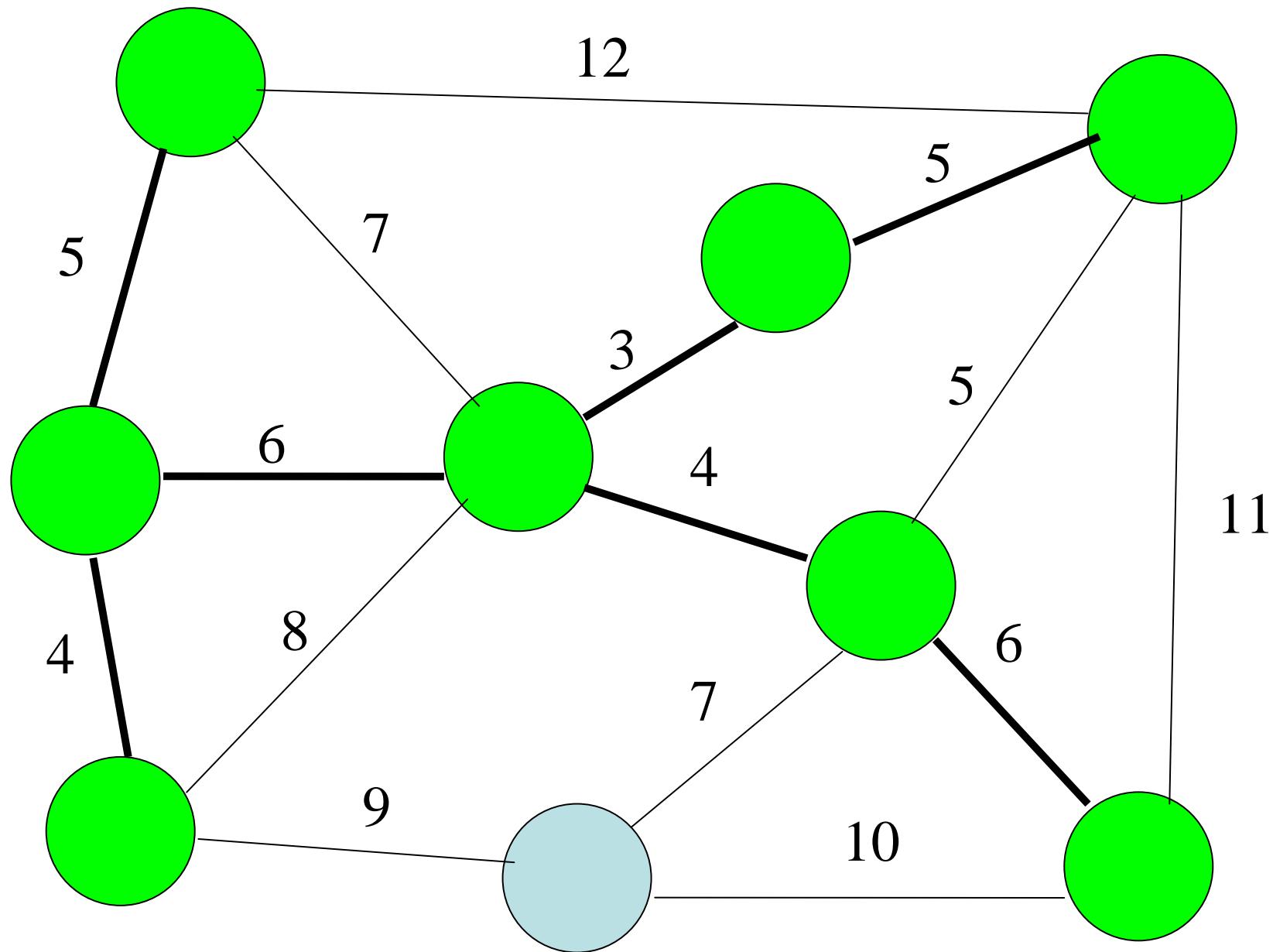


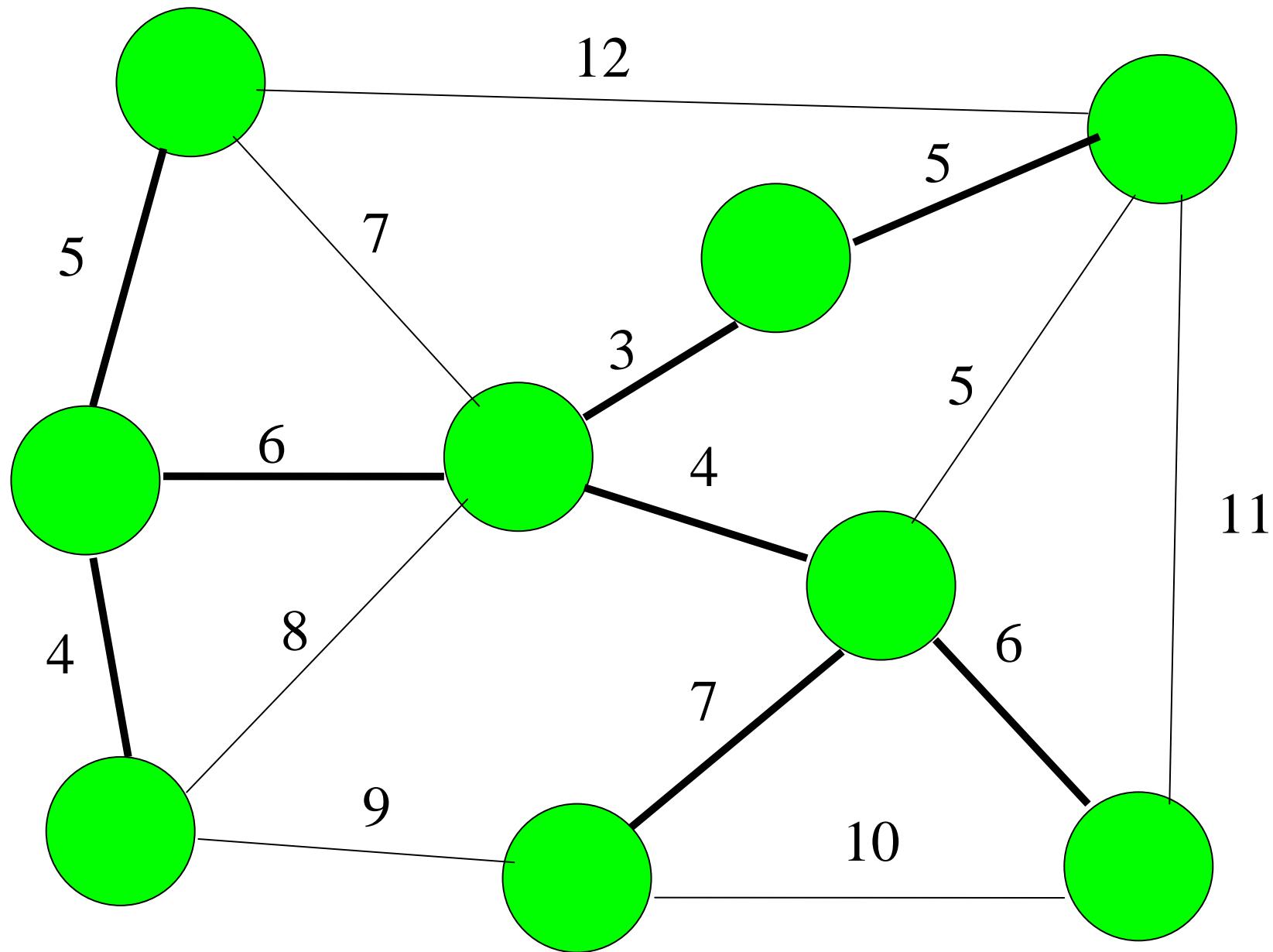












# Implementación

---

- Clave: Seleccionar eficientemente el nuevo arco para añadir al ARM, T.
- Utilizaremos una cola con prioridad Q de vértices con dos campos:
  - $\text{key}[v]$  = menor peso de un arco que conecte  $v$  con un vértice en el conjunto de arcos seleccionados. Toma el valor infinito si no existe dicho arco
  - $p[v]$  = padre del  $v$  en el árbol.

# Implementación

---

**Función Prim(G(V,A), r ) // r es el vertice\_inicio.**

**for each  $u$  en V**

**key[u] = infinito;**

**p[u] = NULL**

**key[r] = 0;**

**PriorityQueue Q(V) // con todos los vértices de V,**

**while (!Q.empty())**

**u = Q.ExtractMin();**

**for each  $v$  en Adj[u]**

**if ( $v$  en Q and peso( $u,v$ ) < key[v])**

**p[v] = u;**

**key[v] = peso(u,v);**

**El ARM está almacenado en la matriz de padres P**

# Eficiencia algoritmo Prim

---

Función Prim( $G(V,A)$ ,  $r$ ) // .

for each  $u$  en  $V$

    key[ $u$ ] = infinito;

    p[ $u$ ] = NULL

    key[ $r$ ] = 0;

PriorityQueue  $Q(V)$  //  $O(V \log V)$ , se puede bajar a  $O(V)$ .

while ( $!Q.empty()$ )

$u = Q.ExtractMin()$ ;  $O(\log V)$ , en total  $O(V \log V)$

    for each  $v$  en  $Adj[u]$

        if ( $v$  en  $Q$  and  $peso(u,v) < key[v]$ )

$p[v] = u$ ;

$key[v] = peso(u,v)$ ;

# Eficiencia algoritmo Prim

---

Función Prim( $G(V,A)$ ,  $r$ ) // .

for each  $u$  en  $V$

    key[ $u$ ] = infinito;

    p[ $u$ ] = NULL

    key[ $r$ ] = 0;

PriorityQueue  $Q(V)$  //  $O(V \log V)$ , se puede bajar a  $O(V)$ .

while ( $!Q.empty()$ )

$u = Q.ExtractMin()$ ;  $O(\log V)$ , en total  $O(V \log V)$

**for each  $v$  en  $Adj[u]$**

**if ( $v$  en  $Q$  and  $peso(u,v) < key[v]$ )**  $O(1)$

            p[ $v$ ] =  $u$ ;

            key[ $v$ ] =  $peso(u,v)$ ; => Modificar  $Q \Rightarrow O(\log V)$

Este cálculo se realiza una vez para cada arco del grafo, por tanto en total tenemos un tiempo  $O(A \log V)$

# Eficiencia algoritmo Prim

---

- Finalmente, podemos concluir que la eficiencia del algoritmo de Prim es del orden

$$O(A \log V + V \log V) = O(A \log V)$$

# Problema de Caminos Minimos

---

Dado un grafo ponderado se quiere calcular el camino con menor peso entre un vértice  $v$  y otro  $w$ .

# Problema de Caminos Minimos

Supongamos que tenemos un mapa de carreteras de España y estamos interesados en conocer el camino más corto que hay para ir desde Granada a Guevejar.



# Problema de Caminos Minimos

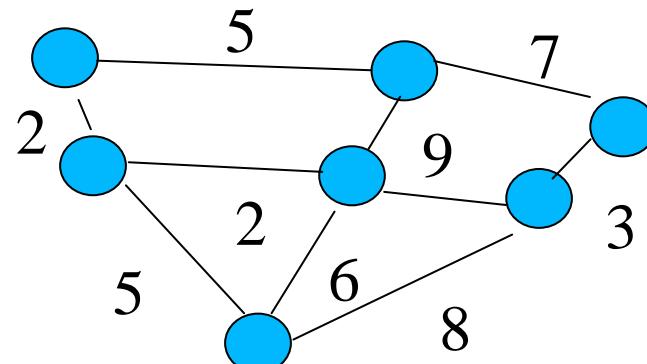
Modelamos el mapa de carreteras como un grafo: los vértices representan las intersecciones y los arcos representan las carreteras. El peso de un arco  $\leq$  distancia



# Problema de Caminos Minimos

---

- Dado un grafo  $G(V,A)$  y dado un vértice  $s$



Encontrar el  
Camino de costo mínimo para llegar  
desde  $s$  al resto de los vértices en el grafo

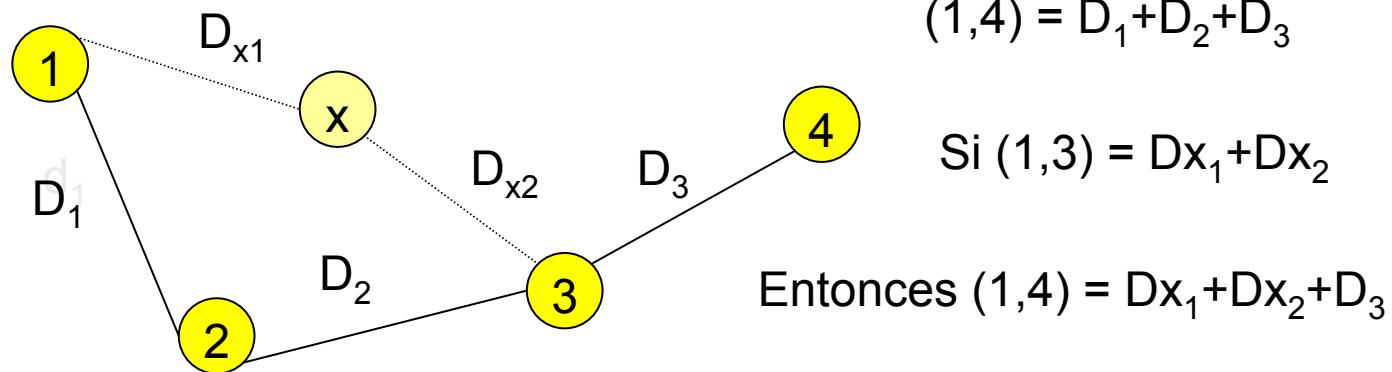
El costo del camino se define como la suma  
de los pesos de los arcos

# Propiedades de Caminos Mínimos

---

Asumimos que no hay arcos con costo negativo.

P1.- Tiene subestructuras óptimas. Dado un camino óptimo, todos los subcaminos son óptimos.  
(xq?)



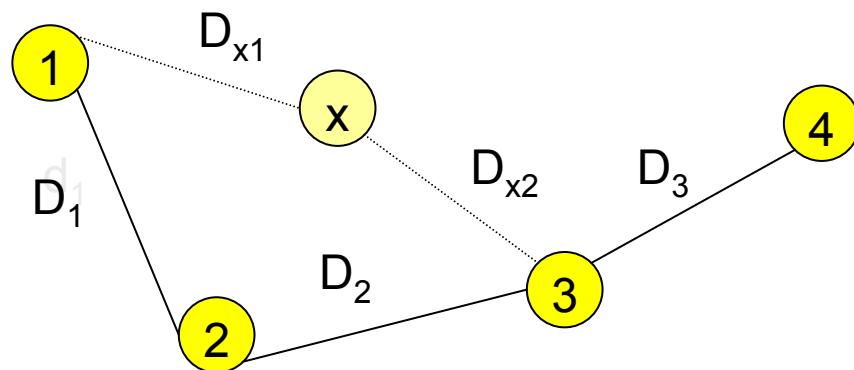
# Propiedades de Caminos Mínimos

---

P2.- Si  $M(s,v)$  es la longitud del camino mínimo para ir de  $s$  a  $v$ , entonces se satisface que

$$M(s,v) \leq M(s,u) + M(u,v)$$

(xq?)



# Implementación: Algoritmo Dijkstra

---

*Candidatos: Vértices*

*Función Selección:* Seleccionar el vértice  $u$  del conjunto de no seleccionados ( $V \setminus S$ ) que tenga menor distancia al vértice origen ( $s$ ).

Uso de cola con prioridad  $Q$  de vértices con dos campos:

- $d[v]$  = longitud del camino de menor distancia del vértice  $s$  a el vértice  $v$  pasando por vértices en  $S$  a Toma el valor infinito si no existe dicho camino
- $p[v]$  = padre del  $v$  en el camino. Toma Null si no existe dicho padre.

# Implementación: Alg. Dijkstra.

---

- Al incluirse un vértice  $v$  en  $S$  puede ocurrir que sea necesario actualizar  $d[x]$ .
  - $Xq?$
  - Qué vértices son susceptibles de sufrir dicha actualización?
  - Como se ve afectado  $d[x]$  y  $p[x]$ ?

# Implementación: Alg. Dijkstra.

---

## ALGORITMO DE DIJKSTRA

Para cada  $v$  en  $V$

$d[v] = \text{infinito}$

$\text{pred}(v) = \text{null}$

$d[s] = 0$

Priorityqueue\_Q ( $V$ ); // se crea la cola respecto a  $d[x]$

while (!Q.empty())

$v = Q.\text{ExtractMin}()$

Para cada  $w$  en  $\text{Adj}[v]$

if  $d[w] > d[v] + c(v,w)$

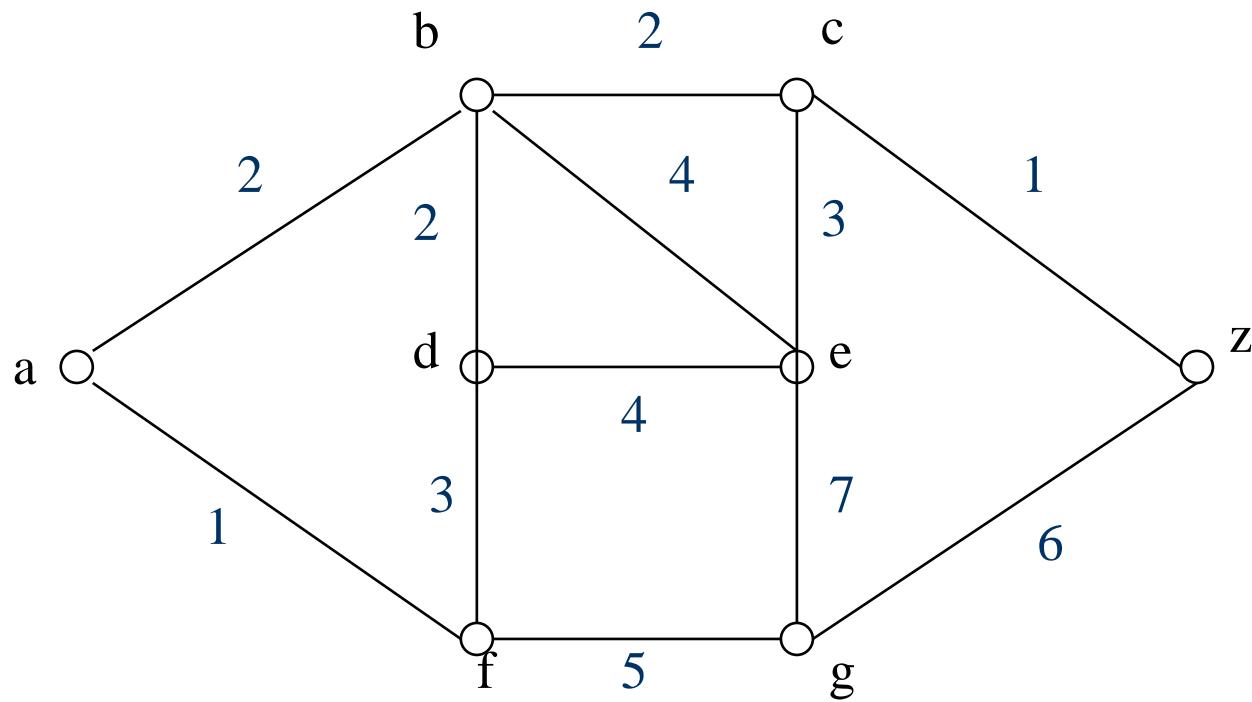
$d[w] = d[v] + c(v,w)$  // se reubica en la cola

$\text{pred}(w) = v$  // se van almacenando los caminos

(Sol.: Se recorren los padres hacia atrás desde el vértice destino hasta el origen)

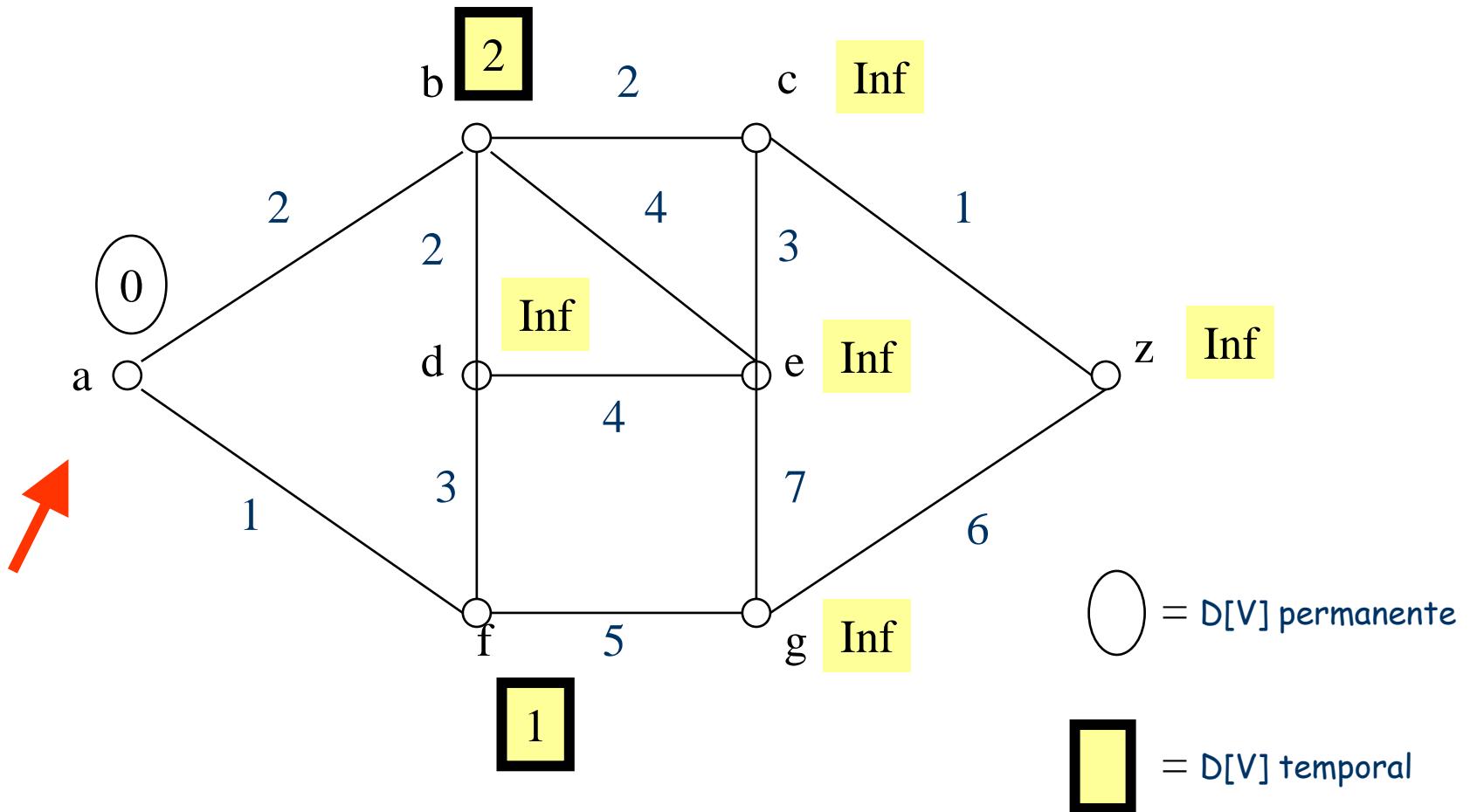
# Ejemplo: Algoritmo Dijkstra

---



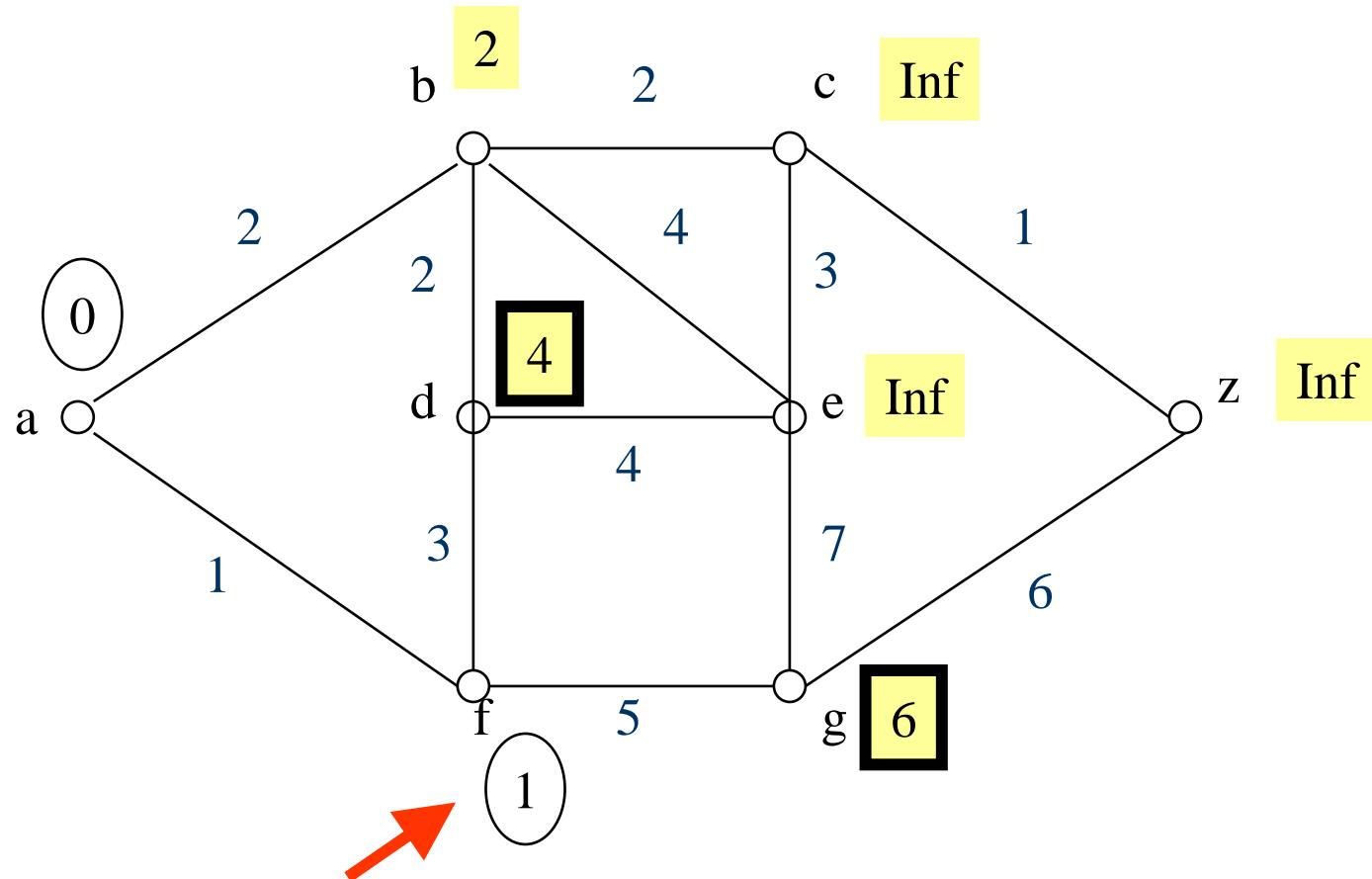
Ejemplo (Solo entre a y z)

# Ejemplo: Algoritmo Dijkstra



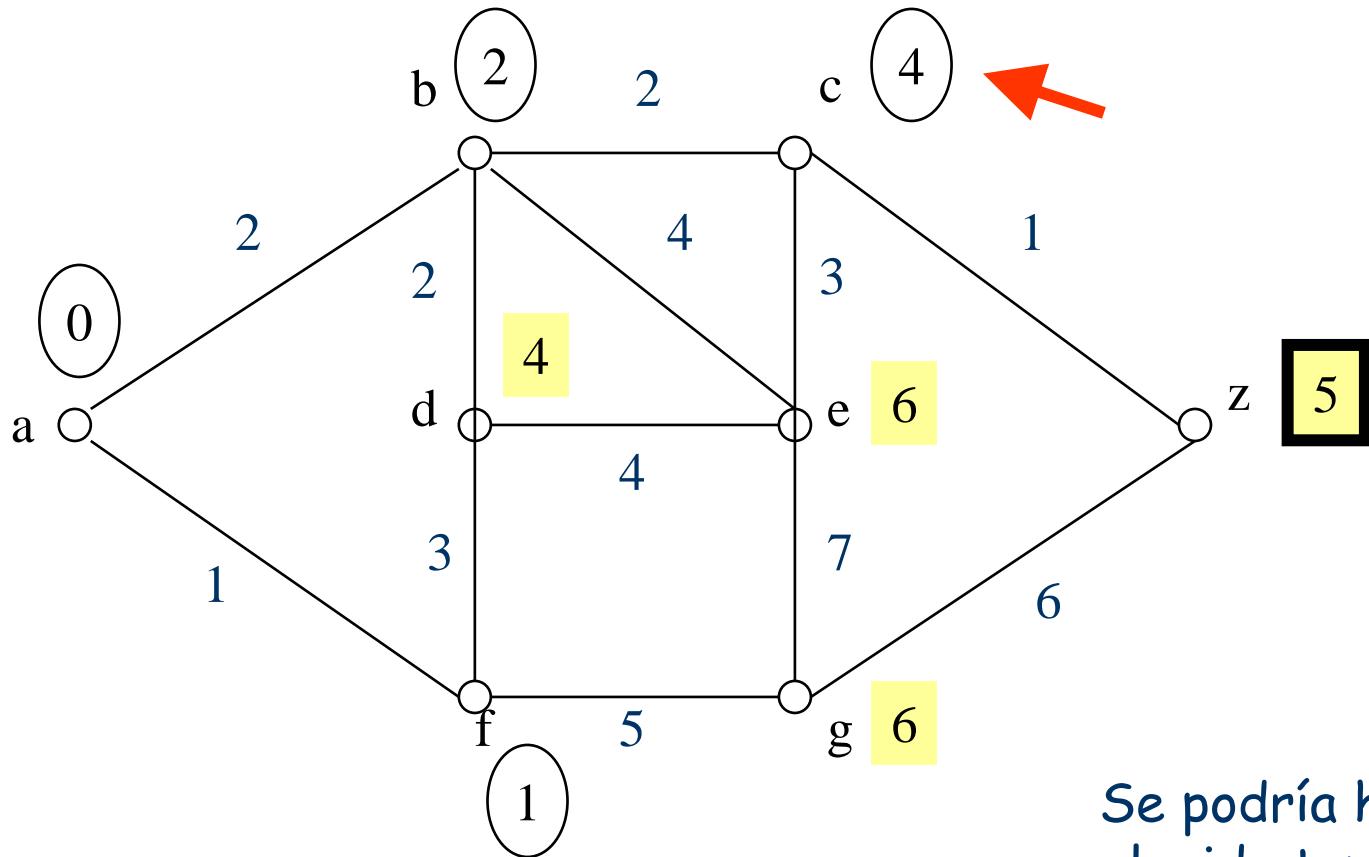
Primera iteración

# Ejemplo: Algoritmo Dijkstra



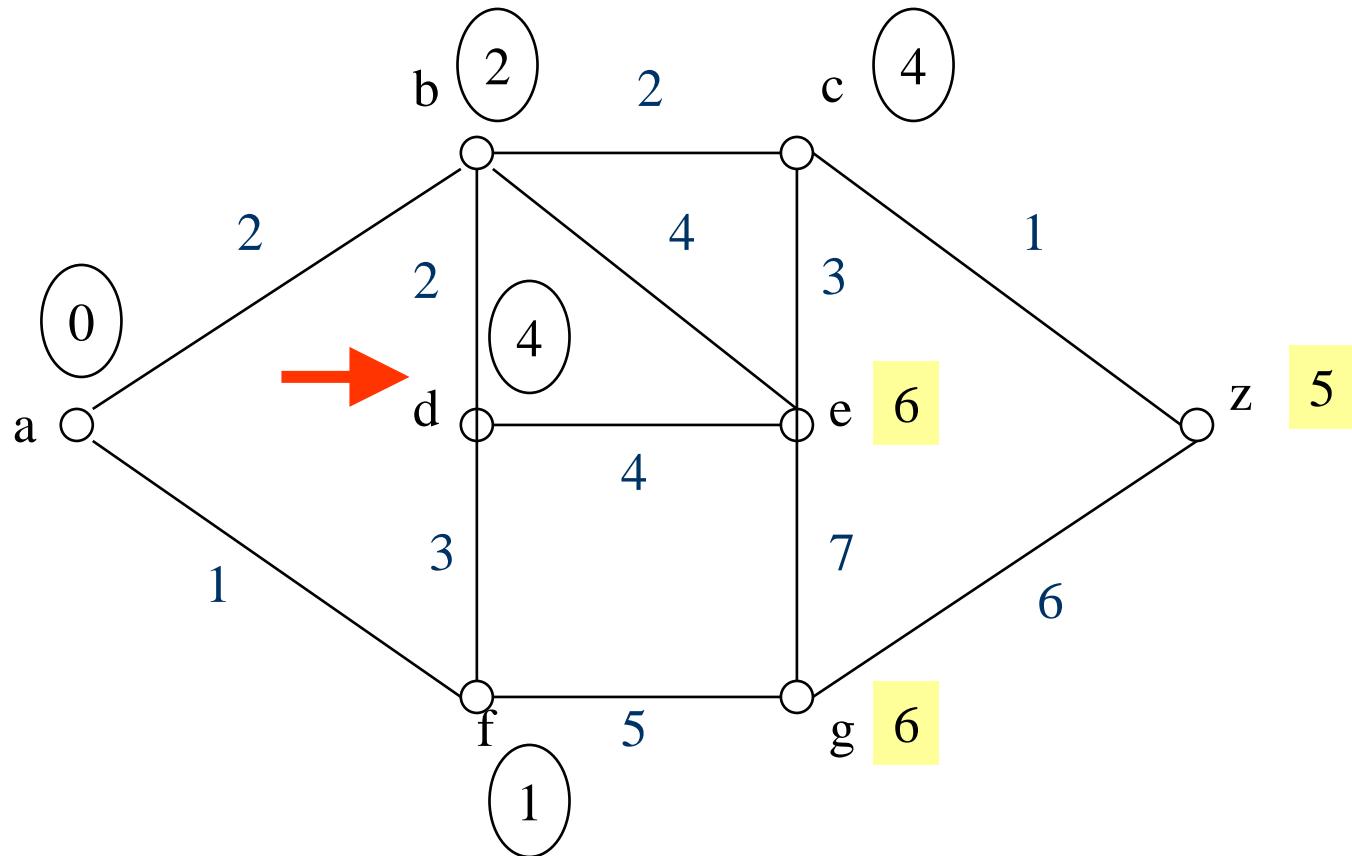
Segunda Iteración

# Ejemplo: Algoritmo Dijkstra



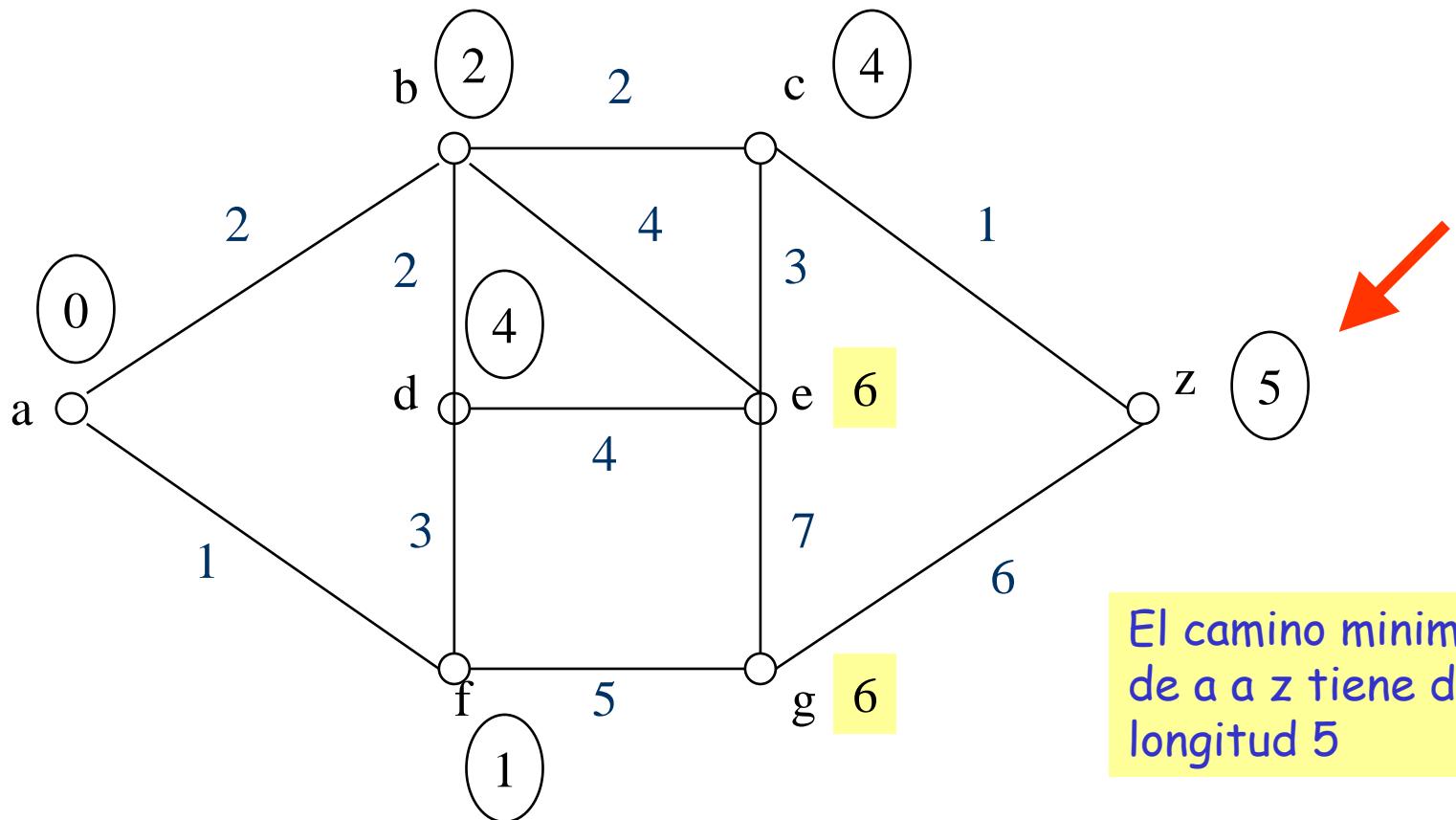
Cuarta Iteración

# Ejemplo: Algoritmo Dijkstra



Quinta Iteración

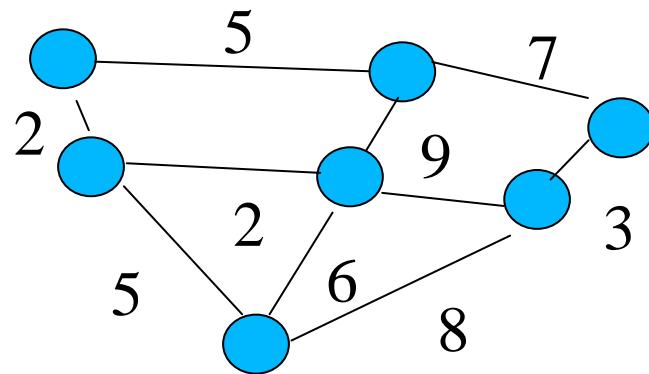
# Ejemplo: Algoritmo Dijkstra



Sexta (y ultima) Iteración

# Ejemplo: Algoritmo Dijkstra

---



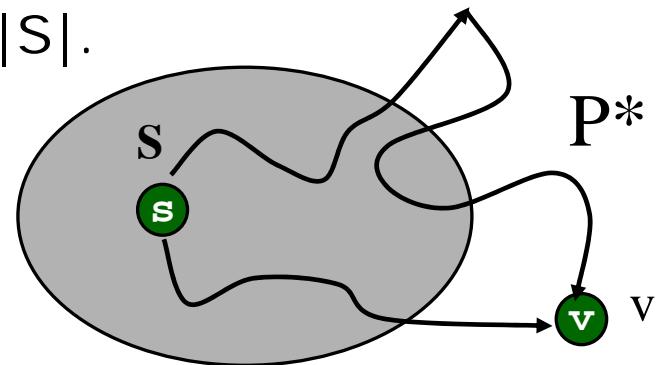
Queda como ejercicio.

# Algoritmo Dijkstra: Demostración

---

**Invariante.** Para cada  $v \in S$ ,  $d(v) = M(s, v)$ .

- Demostr. Por inducción sobre  $|S|$ .
- Caso base:  $|S| = 0$  es trivial.



- Paso inducción:
  - Supongamos que el algoritmo de Dijkstra añade el vértice  $v$  a  $S$ .  $d(v)$  representa la longitud de algún camino de  $s$  a  $v$
  - Si  $d(v)$  no es la longitud del camino mínimo de  $s$  a  $v$ , entonces sea  $P^*$  el camino mínimo de  $s-v$
  - Sea  $x$  el último vértice en  $S$  en dicho camino

# Algoritmo Dijkstra: Demostración

En este caso  $P^*$  debe utilizar algún arco que parta de  $x$ , por ejemplo  $(x, y)$

- Entonces tenemos que

$$d(v) > M(s, v)$$

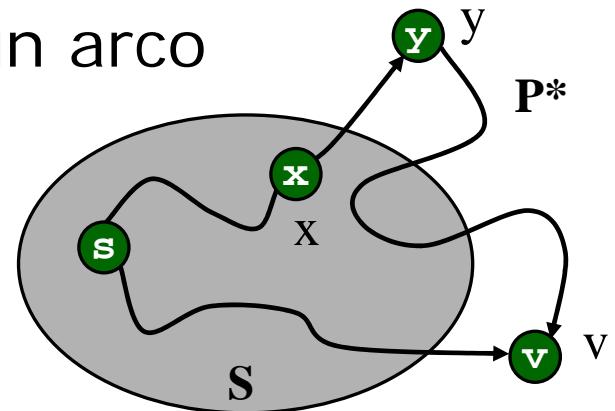
$$= M(s, x) + c(x, y) + M(y, v)$$

$$\geq M(s, x) + c(x, y)$$

$$= d[x] + c(x, y)$$

$$\geq d(y)$$

asumimos  
subestr. optimal  
arcos positivos  
hipótesis inducción



Por tanto el algoritmos de Dijkstra hubiese seleccionado  $y$  en lugar de  $v$ .

# Análisis de Eficiencia

---

- EL análisis del algoritmo es parecido al realizado para el algoritmo de Prim, deduciendo que el algoritmo es del orden  $O(A \log V)$ .
- Realmente es  $O((A+V)\log V)$ , pero se supone que A es bastante mayor que V. Además, si el grafo es muy denso  $A \approx V^2$ .
  - Demostrarlo queda como ejercicio.

# Índice

---

- **EL ENFOQUE GREEDY**
- **ALGORITMOS GREEDY EN GRAFOS**
- 
- **HEURÍSTICA GREEDY**
  - Introducción a la Heurística Greedy
  - El Problema de Coloreo de un Grafo
  - Problema del Viajante de Comercio
  - Problema de la Mochila

# Heurísticas Greedy

---

## SITUACIÓN QUE NOS PODEMOS ENCONTRAR

- Hay casos en los cuales no se puede conseguir una algoritmo voraz para el que se pueda demostrar que encuentra la solución óptima
- Sin embargo, en muchos casos se pueden llegar plantear para distintos problemas NP-completos

# Heurísticas

---

- **Heurística:** Son procedimientos que, basados en la experiencia, proporcionan buenas soluciones a problemas concretos
- **Metaheurísticas de propósito general:**
  - Enfriamiento (Recocido) Simulado, Busqueda Tabu,
  - GRASP (Greedy Randomized Adaptive Search Procedures), Busqueda Dispersa, Busqueda por Entornos Variables, Busqueda Local Guiada, Busqueda Local Iterativa
  - Computación Evolutiva (Algoritmos Genéticos, ...), Algoritmos Memeticos, Colonias de Hormigas, Redes de Neuronas, Algoritmos Basados en Sistemas Inmunológicos, ...

# Heurísticas Greedy

---

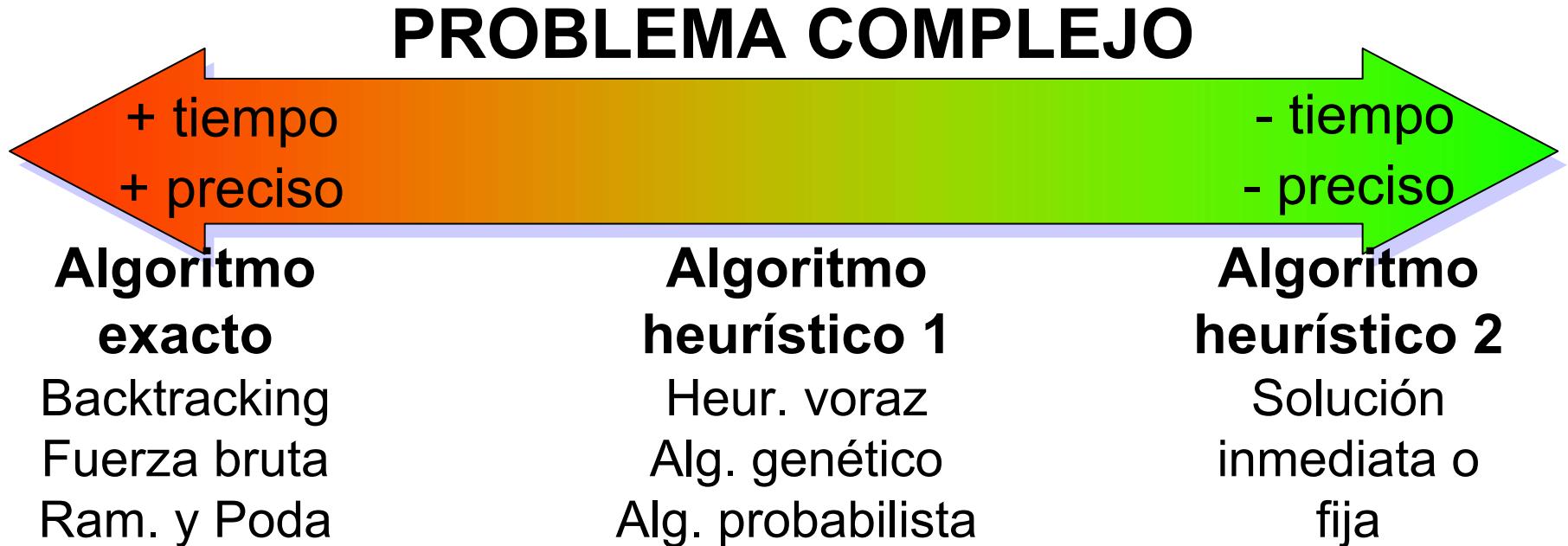
- **¿Satisfacer /optimizar?**
- El tiempo efectivo que se tarda en resolver un problema es un factor clave
- Los algoritmos greedy pueden actuar como heurísticas
  - El problema del coloreo de un grafo
  - El problema del Viajante de Comercio
  - El problema de la Mochila
  - ...
- Suelen usarse también para encontrar una primera solución (como inicio de otra heurística)

# Heurísticas greedy: Resumen

- **Problemas NP-completos:** la solución exacta puede requerir órdenes factoriales o exponenciales (el problema de la *explosión combinatoria*).
- **Objetivo:** obtener “buenas” soluciones en un tiempo de ejecución corto (*razonable*).
- **Algoritmos de aproximación:** garantizan una solución más o menos buena (o una cierta aproximación respecto al óptimo).
- Un tipo son los **algoritmos heurísticos**<sup>1</sup>: algoritmo basado en el conocimiento “intuitivo” o “experto” del programador sobre determinado problema.

<sup>1</sup>DRAE. *Heurística*: Arte de inventar.

# Heurísticas greedy: Resumen



- La estructura de algoritmos voraces se puede utilizar para construir procedimientos heurísticos: hablamos de **heurísticas voraces**.
- **La clave:** diseñar buenas funciones de selección.

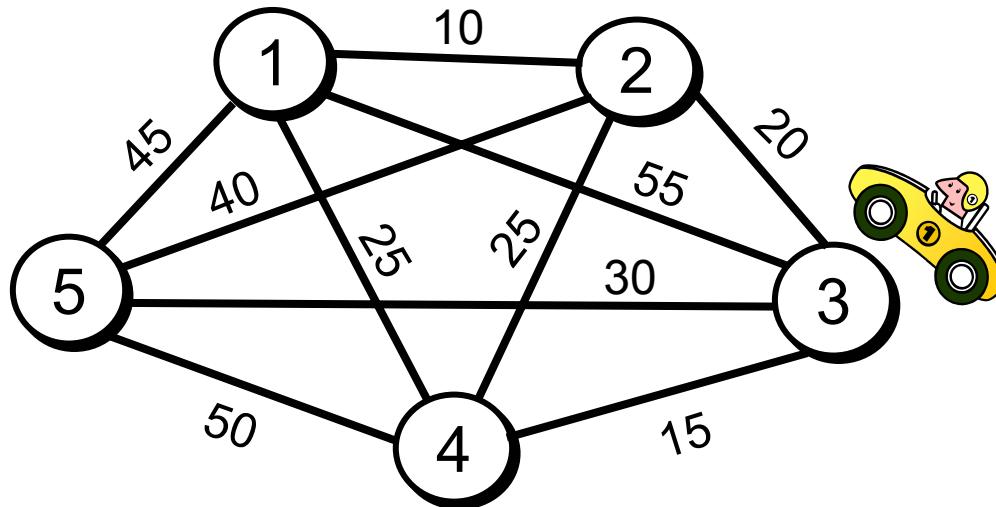
# El Problema del Viajante de Comercio

---

- Un viajante de comercio que reside en una ciudad, tiene que trazar una ruta que, partiendo de su ciudad, visite todas las ciudades a las que tiene que ir una y sólo una vez, volviendo al origen y con un recorrido mínimo
- Es un problema NP, no existen algoritmos en tiempo polinomial, aunque si los hay exactos que lo resuelven para grafos con 40 vértices aproximadamente.
- Para más de 40, es necesario utilizar heurísticas, ya que el problema se hace intratable en el tiempo.

# El problema del viajante.

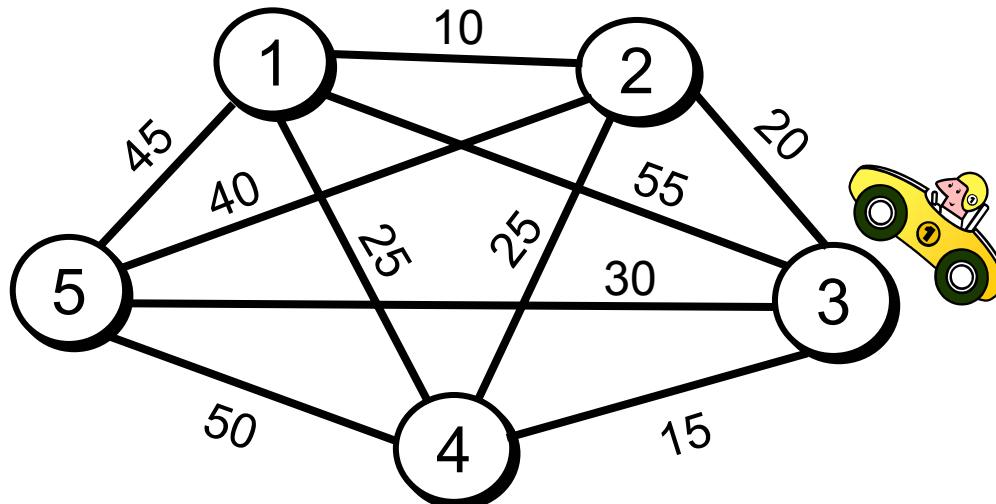
- **Problema:** Dado un grafo no dirigido, completo y ponderado  $G = (V, A)$ , encontrar un ciclo de coste mínimo que pase por todos los nodos.



- Es un problema NP-completo, pero necesitamos una solución eficiente.
- Problema de optimización, la solución está formada por un conjunto de elementos en cierto orden: podemos aplicar el esquema voraz.

# El problema del viajante.

- Primera cuestión: ¿Cuáles son los candidatos?
- **Dos posibilidades:**
  - 1) Los nodos son los candidatos. Empezar en un nodo cualquiera. En cada paso moverse al nodo no visitado más próximo al último nodo seleccionado.
  - 2) Las aristas son los candidatos. Hacer igual que en el algoritmo de Kruskal, pero garantizando que se forme un ciclo.

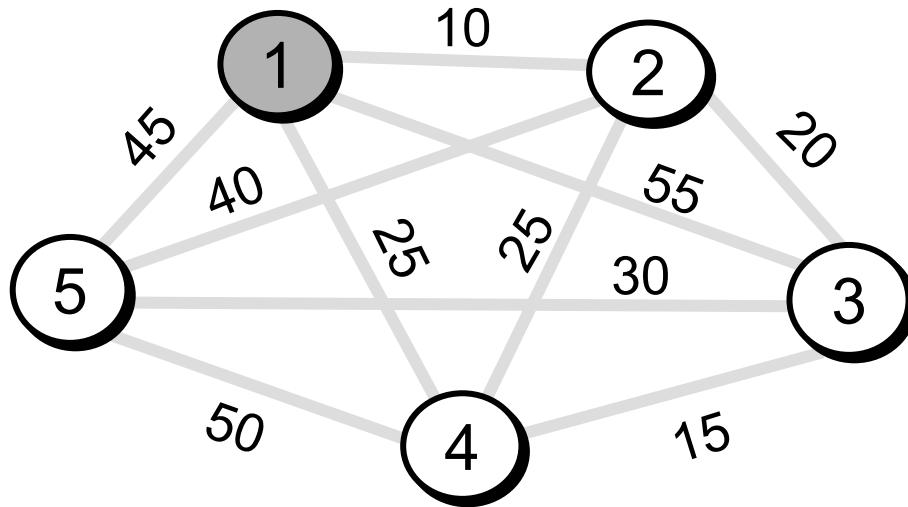


# El problema del viajante.

- **Heurística voraz 1) Candidatos = V**
  - Una solución será un cierto orden en el conjunto de nodos.
  - **Representación de la solución:**  $s = (c_1, c_2, \dots, c_n)$ , donde  $c_i$  es el nodo visitado en el lugar  $i$ -ésimo.
  - **Inicialización:** empezar en un nodo cualquiera.
  - Función de **selección**: de los nodos candidatos seleccionar el más próximo al último (o al primero) de la secuencia actual ( $c_1, c_2, \dots, c_a$ ).
  - Acabamos cuando tengamos  $n$  nodos.

# El problema del viajante.

- **Ejemplo 1.** Empezando en el nodo 1.

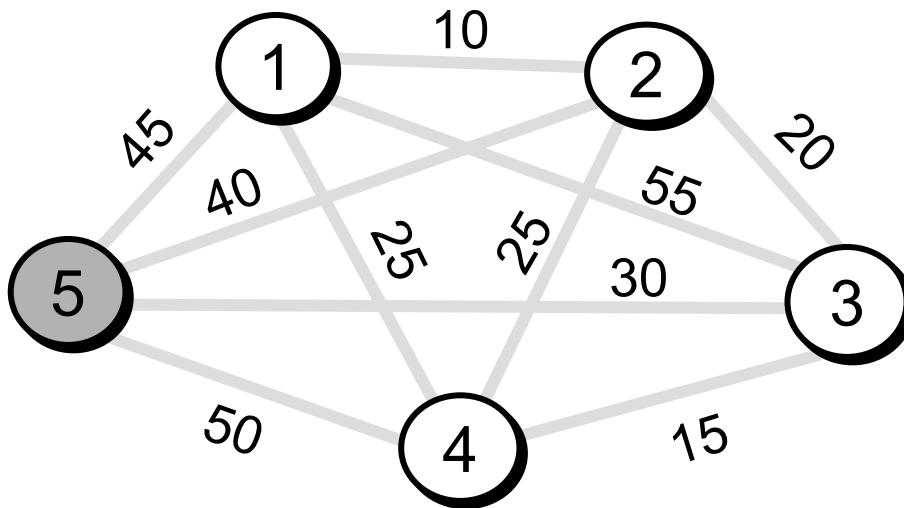


Solución: (5, 1, 2, 3, 4)

Coste total:  $45+10+20+15+50=140$

# El problema del viajante.

- **Ejemplo 2.** Empezando en el nodo 5.



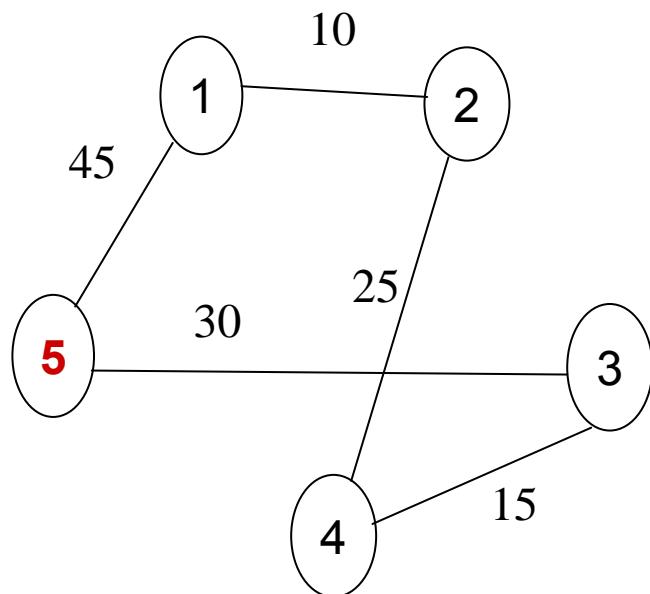
Solución: (5, 3, 4, 2, 1)

Coste total:  $30+15+25+10+45 = 125$

- **Conclusión:** el algoritmo no es óptimo.

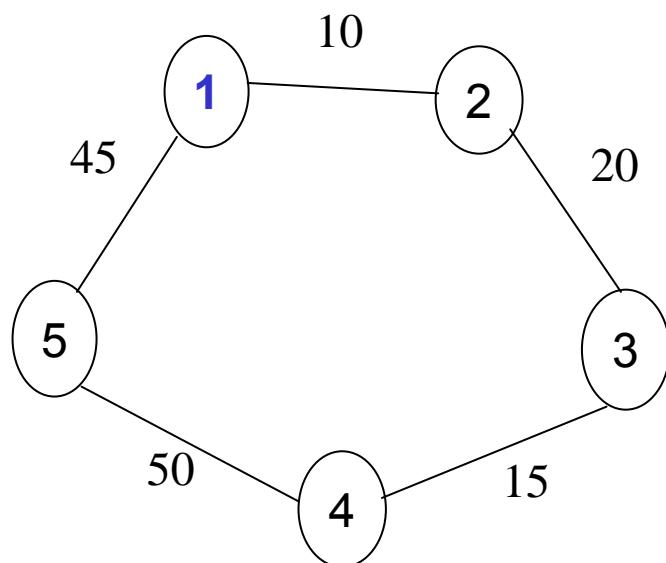
# El Problema del Viajante

- Solución con la primera heurística
- Solución empezando en el nodo 5



Solución:  $(5, 3, 4, 2, 1)$ , 125

Solución empezando en el nodo 1



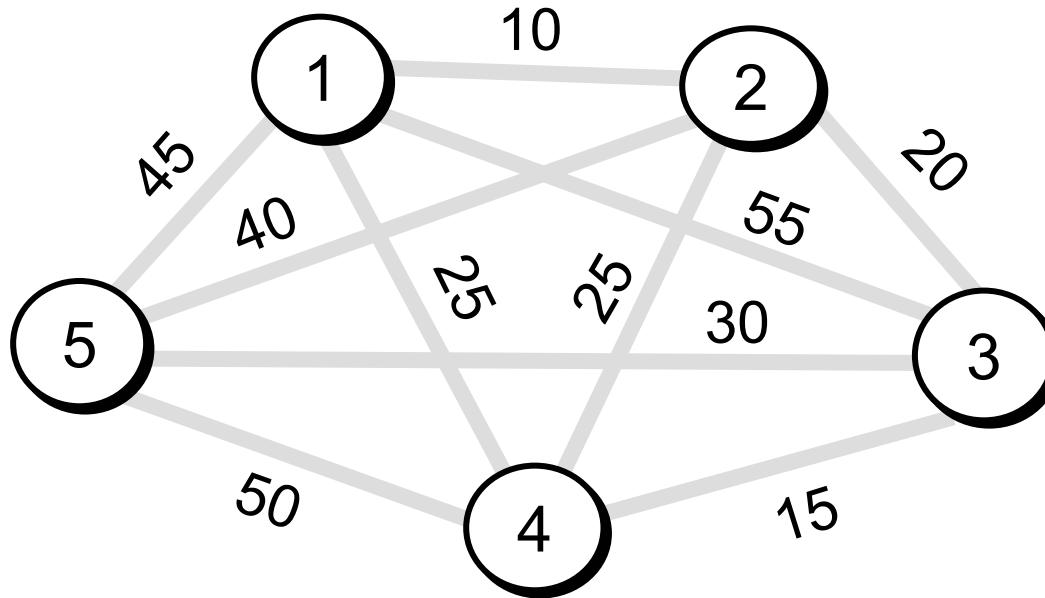
Solución:  $(5, 1, 2, 3, 4)$ , 140

# El problema del viajante.

- **Heurística voraz 2) Candidatos = A**
  - Una solución será un conjunto de aristas ( $a_1, a_2, \dots, a_n$ ) que formen un ciclo hamiltoniano, sin importar el orden.
  - **Representación de la solución:**  $s = (a_1, a_2, \dots, a_n)$ , donde cada  $a_i$  es una arista, de la forma  $a_i = (v_i, w_i)$ .
  - **Inicialización:** empezar con un grafo sin aristas.
  - **Selección:** seleccionar la arista candidata de menor coste.
  - **Factible:** una arista se puede añadir a la solución actual si no se forma un ciclo (excepto para la última arista añadida) y si los nodos unidos no tienen grado mayor que 2.

# El problema del viajante.

- Ejemplo 3.



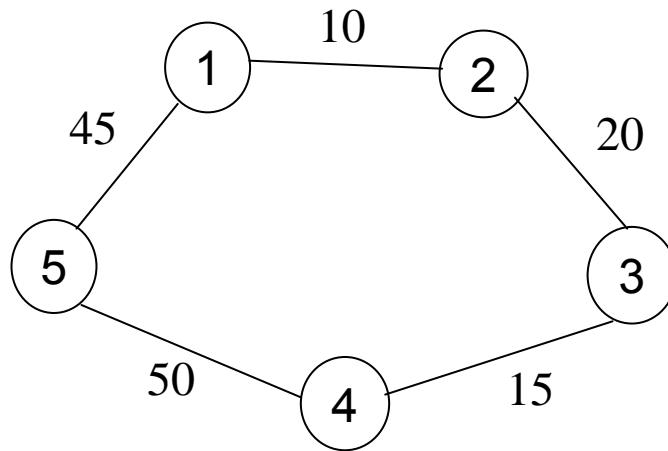
Solución: ((1, 2), (4, 3), (2, 3), (1, 5), (4, 5))

Coste total =  $10+15+20+45+50 = 140$

# El Problema del Viajante

---

- Solucion con la segunda heurística



- Solución:  $((1, 2), (4, 3), (2, 3), (1, 5), (4, 5))$   
Coste =  $10 + 15 + 20 + 45 + 50 = 140$
- En todos los casos la eficiencia es la del algoritmo de ordenación que se use

# El problema del viajante.

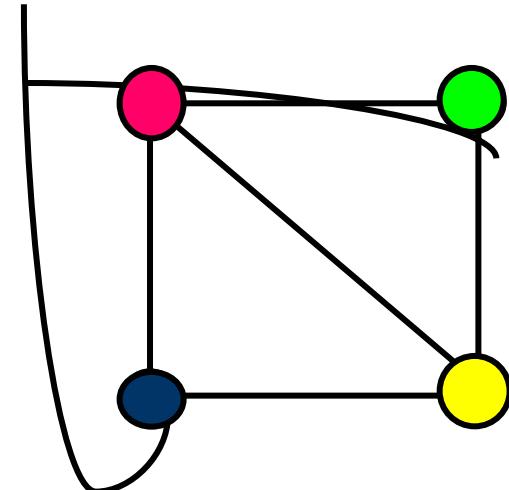
## Conclusiones:

- Ninguno de los dos algoritmos garantiza la solución óptima.
- Sin embargo, “normalmente” ambos dan soluciones buenas, próximas a la óptima.
- Posibles mejoras:
  - Buscar heurísticas mejores, más complejas.
  - Repetir la heurística 1 con varios orígenes.
  - A partir de la solución del algoritmo intentar hacer **modificaciones locales** para mejorar esa solución.

# El Problema del Coloreo de un Grafo

---

- Planteamiento
  - Dado un grafo plano  $G=(V, E)$ , determinar el minimo numero de colores que se necesitan para colorear todos sus vertices, y que no haya dos de ellos adyacentes pintados con el mismo color
- Si el grafo no es plano puede requerir tantos colores como vertices haya
- Las aplicaciones son muchas
  - Representación de mapas
  - Diseño de paginas webs
  - Diseño de carreteras



# El Problema del Coloreo de un Grafo

---

- El problema es NP y por ello se necesitan heurísticas para resolverlo rápido
- El problema reune todos los requisitos para ser resuelto con un algoritmo greedy
- Del esquema general greedy se deduce un algoritmo inmediatamente.
- Teorema de Appel-Hanke (1976): Un grafo plano requiere a lo sumo 4 colores para pintar sus nodos de modo que no haya vértices adyacentes con el mismo color

# El Problema del Coloreo de un Grafo

---

- Suponemos que tenemos una paleta de colores (con mas colores que vértices)
- Elegimos un vértice no coloreado y un color. Pintamos ese vértice de ese color
- Lazo greedy: Seleccionamos un vértice no coloreado  $v$ . Si no es adyacente (por medio de una arista) a un vértice ya coloreado con el nuevo color, entonces coloreamos  $v$  con el nuevo color
- Se itera hasta pintar todos los vértices

# El problema del coloreo de un grafo

- Podemos usar una **heurística voraz** para obtener una solución:
  - Inicialmente ningún nodo tiene color asignado.
  - Tomamos un color  $colorActual := 1$ .
  - Para cada uno de los nodos sin colorear:
    - Comprobar si es posible asignarle el color actual.
    - Si se puede, se asigna. En otro caso, se deja sin colorear.
  - Si quedan nodos sin colorear, escoger otro color ( $colorActual := colorActual + 1$ ) y volver al paso anterior.

# El problema del coloreo de un grafo

- La estructura básica del esquema voraz se repite varias veces, una por cada color, hasta que todos los nodos estén coloreados.
- Función de **selección**: cualquier candidato restante.
- **Factible(x)**: se puede asignar un color a **x** si ninguno de sus adyacentes tiene ese mismo color.

**para todo** nodo y adyacente a **x hacer**

**si**  $c_y == \text{colorActual}$  **entonces**  
**devolver** false

**finpara**

**devolver** true

- ¿Garantiza el algoritmo la solución óptima?

# Implementacion del algoritmo

---

Función **COLOREO**

{**COLOREO** pone en **NuevoColor** los vértices de **G** que  
pueden tener el mismo color}

**Begin**

**NuevoColor** =  $\emptyset$

**Para cada vértice no coloreado v de G Hacer**

**Si v no es adyacente a ningún vértice en NuevoColor**

**Entonces**

**Marcar v como coloreado**

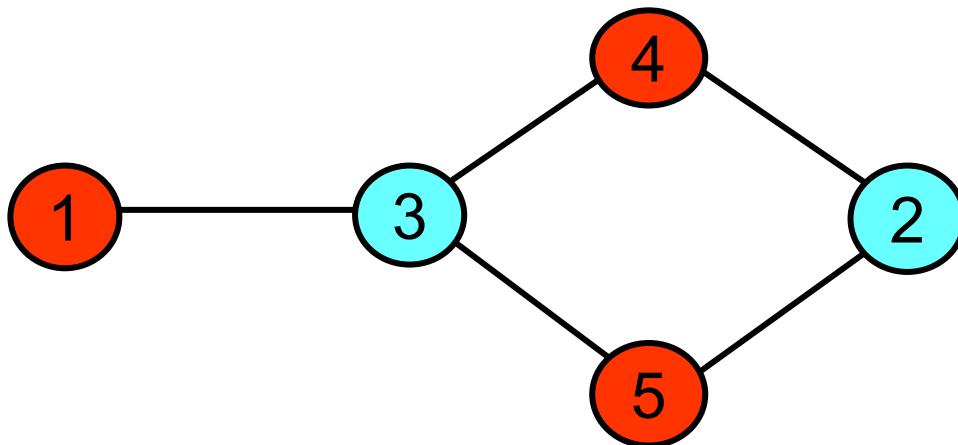
**Añadir v a NuevoColor**

**End**

Se trata de un algoritmo que funciona en **O(n)**, pero que  
no siempre da la solución óptima

# Ejemplo

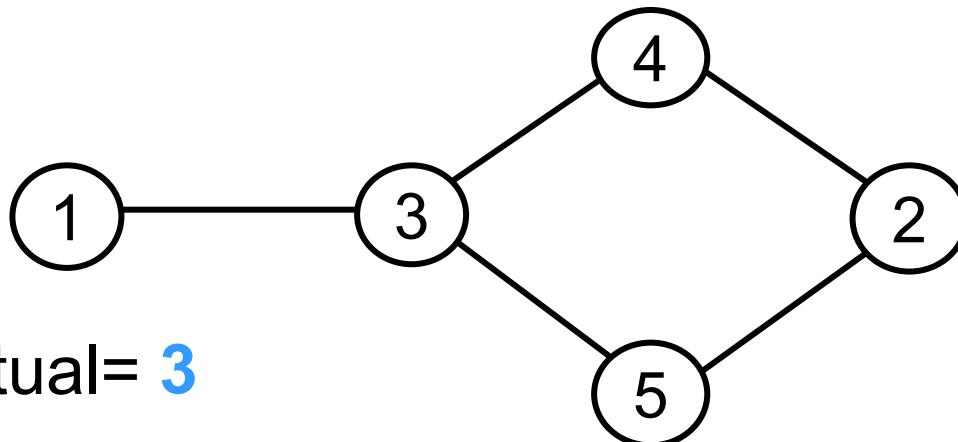
- **Representación de la solución:** una solución tiene la forma  $(c_1, c_2, \dots, c_n)$ , donde  $c_i$  es el color asignado al nodo  $i$ .
- La solución es válida si para toda arista  $(v, w) \in A$ , se cumple que  $c_v \neq c_w$ .



- $S = (1, 2, 2, 1, 1)$ , Total: 2 colores

# Ejemplo

Pero...



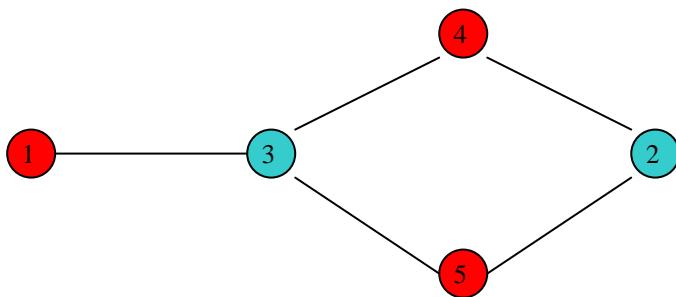
colorActual= 3

$c_1$	$c_2$	$c_3$	$c_4$	$c_5$

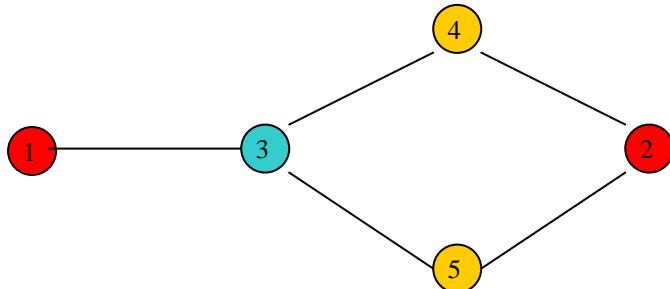
- **Resultado:** se necesitan 3 colores. Recordar que el óptimo es 2 colores.
- **Conclusión:** el algoritmo no es óptimo.

# Ejemplo

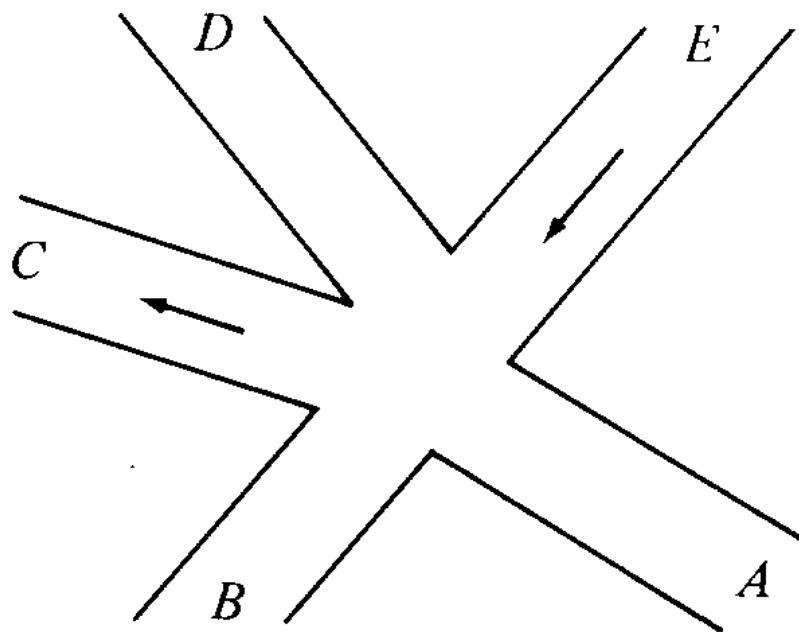
---



El orden en el que se escogen los vértices para colorearlos puede ser decisivo: el algoritmo da la solución óptima en el grafo de arriba, pero no en el de abajo



# Ejemplo: Diseño de cruces de semáforos



- A la izquierda tenemos un cruce de calles
- Se señalan los sentidos de circulación.
- La falta de flechas, significa que podamos ir en las dos direcciones.
- Queremos diseñar un patrón de semáforos con el mínimo número de semáforos, lo que
  - Ahorrara tiempo (de espera) y dinero
  - Suponemos un grafo cuyos vértices representan turnos, y cuyas aristas

unen esos turnos que no pueden realizarse simultáneamente sin que haya colisiones, y el problema del cruce con semáforos se convierte en un problema de coloreo de los vértices de un grafo

# El problema de la Mochila

---

- Tenemos  $n$  objetos y una mochila. El objeto  $i$  tiene un peso  $w_i$  y la mochila tiene una capacidad  $M$ .
- Si metemos en la mochila la fraccion  $x_i$ ,  $0 \leq x_i \leq 1$ , del objeto  $i$ , generamos un beneficio de valor  $p_i x_i$
- El objetivo es llenar la mochila de tal manera que se maximice el beneficio que produce el peso total de los objetos que se transportan, con la limitación de la capacidad de valor  $M$

$$\text{maximizar} \quad \sum_{1 \leq i \leq n} p_i x_i$$

$$\text{sujeto a} \quad \sum_{1 \leq i \leq n} w_i x_i \leq M$$

$$\text{con } 0 \leq x_i \leq 1, \quad 1 \leq i \leq n$$

# Ejemplo: Mochila 0/1

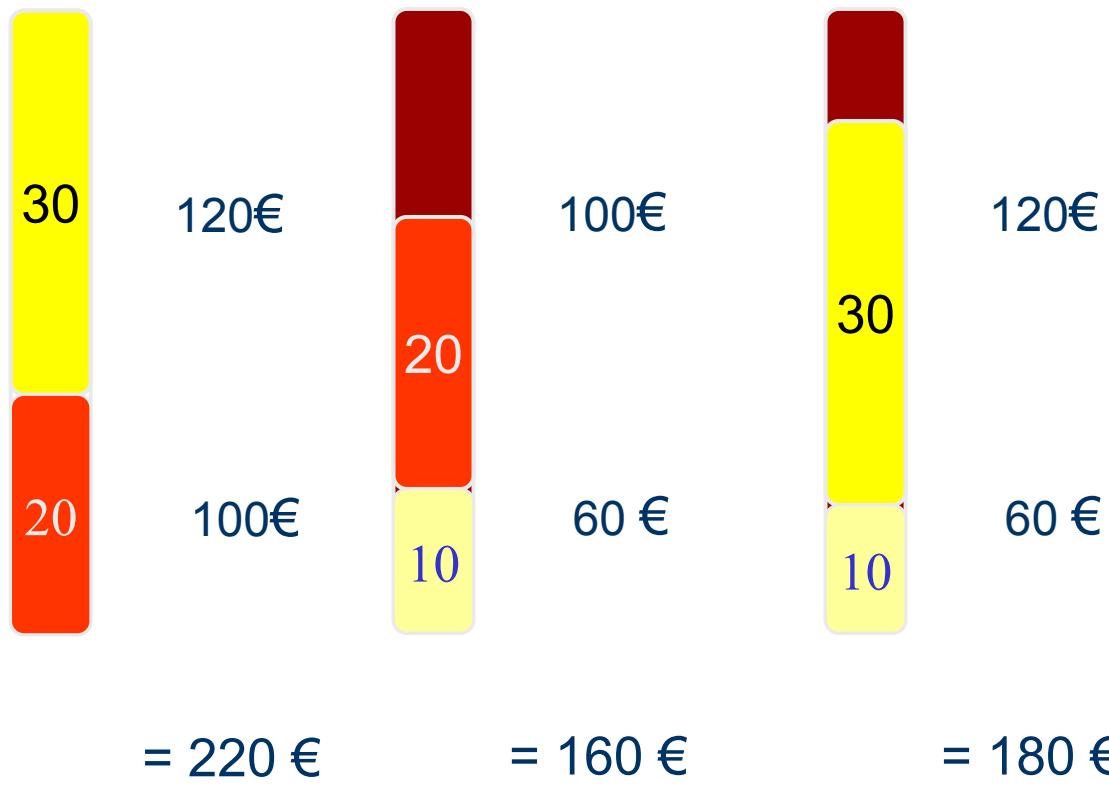
---



Es un claro problema de tipo greedy  
Sus aplicaciones son innumerables  
La tecnica greedy produce soluciones optimales para este tipo de problemas cuando se permite fraccionar los objetos

# Ejemplo: Mochila 0/1

---



Total = 220 €      = 160 €      = 180 €

¿Cómo seleccionamos los ítems?

# Solucion Greedy

---

- Definimos la densidad del objeto  $A_i$  por  $p_i/w_i$ .
- Se usan objetos de tan alta densidad como sea posible, es decir, los seleccionaremos en orden decreciente de densidad.
- Primero, se ordenan los objetos por densidad no creciente, i.e.:

$$p_i/w_i \geq p_{i+1}/w_{i+1} \text{ para } 1 \leq i < n.$$

# PseudoCodigo

---

```
Procedimiento MOCHILA_GREEDY(P,W,M,X,n)
/*P(1:n) y W(1:n) contienen los costos y pesos
 respectivos de los n objetos ordenados como P(I)/W(I)
 ≥ P(I+1)/W(I+1). M es la capacidad de la mochila y
 X(1:n) es el vector solucion*/
real P(1:n), W(1:n), X(1:n), M, cr;
integer I,n;
x = 0; //inicializa la solucion en cero //
cr = M; // cr = capacidad restante de la mochila //
Para i = 1 hasta n Hacer
    Si W(i) > cr Entonces exit endif
    X(I) = 1;
    cr = c - W(i);
repetir
End MOCHILA_GREEDY
```

# Algoritmos voraces.

## Conclusiones:

- Avance rápido se basa en una **idea intuitiva**:
  - Empezamos con una solución “vacía”, y la construimos paso a paso.
  - En cada paso se selecciona un candidato (el más prometedor) y se decide si se mete o no (función factible).
  - Una vez tomada una decisión no se vuelve a deshacer. Acabamos cuando tenemos una solución o nos quedamos sin candidatos.
- **Ojo**: en algunos problemas los algoritmos voraces sí garantizan la **solución óptima**.

# Algoritmos voraces.

## Conclusiones:

- **Primera cuestión:** ¿cuáles son los candidatos?, ¿cómo se representa una solución al problema?
- **Cuestión clave:** diseñar una función de selección adecuada.
  - Algunas pueden garantizar la solución óptima.
  - Otras pueden ser más heurísticas...
- **Función factible:** garantizar las **BACKTRACKING** problemas.
- En general los algoritmos voraces son la solución rápida a muchos problemas (a veces óptimas, otras no).
- ¿Y si podemos deshacer decisiones...? 

# Algorítmica

---

**Tema 1. Planteamiento General**

**Tema 2. La Eficiencia de los Algoritmos**

**Tema 3. Algoritmos “Divide y vencerás”**

**Tema 4. Algoritmos Voraces (“Greedy”)**

**Tema 5. Algoritmos basados en Programación Dinámica**

**Tema 6. Algoritmos para la Exploración de Grafos**

**(“Backtracking”, “Branch and Bound”)**

**Tema 7. Otras metodologías algorítmicas**

# Algorítmica

---

**Tema 1. Planteamiento General**

**Tema 2. La Eficiencia de los Algoritmos**

**Tema 3. Algoritmos “Divide y vencerás”**

**Tema 4. Algoritmos Voraces (“Greedy”)**

**Tema 5. Algoritmos basados en Programación Dinámica**

**Tema 6. Algoritmos para la Exploración de Grafos**

**(“Backtracking”, “Branch and Bound”)**

**Tema 7. Otras metodologías algorítmicas**

# Programación Dinámica

---

- Introducción
- Elementos de la programación dinámica
  - Principio de optimalidad de Bellman
  - Definición recursiva de la solución óptima
  - Enfoque Ascendente
  - Cálculo de la solución óptima
- Ejemplos
  - Multiplicación encadenada de matrices
  - Problema de la mochila
  - Subsecuencia de mayor longitud (LCS)
  - Selección de actividades con pesos
  - Distancia de edición
- Caminos mínimos: Algoritmo de Floyd
- Aplicaciones

# Introducción a la PD

---

- Esta técnica se aplica sobre problemas que a simple vista necesitan un alto coste computacional (posiblemente exponencial) donde:
  - **Subproblemas óptimos:** La solución óptima a un problema puede ser definida en función de soluciones óptimas a subproblemas de tamaño menor, generalmente de forma recursiva.
  - **Solapamiento entre subproblemas:** Al plantear la solución recursiva, un mismo problema se resuelve más de una vez

# Estrategias de diseño

---

## Algoritmos greedy:

Se construye la solución incrementalmente, utilizando un criterio de optimización local.

## Divide y vencerás:

Se descompone el problema en subproblemas **independientes** y se combinan las soluciones de esos subproblemas.

## Programación dinámica:

Se descompone el problema en subproblemas **solapados** y se va construyendo la solución con las soluciones de esos subproblemas.

# Introducción a la PD

---

Se puede seguir un enfoque ascendente (bottom-up):

- Primero se calculan las soluciones óptimas para problemas de tamaño pequeño.
- Luego, utilizando dichas soluciones, encuentra soluciones a problemas de mayor tamaño.

## Clave: Memorización

Almacenar las soluciones de los subproblemas en alguna estructura de datos para reutilizarlas posteriormente. De esa forma, se consigue un algoritmo más eficiente que la fuerza bruta, que resuelve el mismo subproblema una y otra vez.

# Introducción a la PD

---

## Memorización

Para evitar calcular lo mismo varias veces:

- Cuando se calcula una solución, ésta se almacena.
- Antes de realizar una llamada recursiva para un subproblema Q, se comprueba si la solución ha sido obtenida previamente:
  - Si no ha sido obtenida, se hace la llamada recursiva y, antes de devolver la solución, ésta se almacena.
  - Si ya ha sido previamente calculada, se recupera la solución directamente (no hace falta calcularla).
- Usualmente, se utiliza una matriz que se rellena conforme las soluciones a los subproblemas son calculados (espacio vs. tiempo).

# Introducción a la PD

- **Ejemplo. Cálculo de los números de Fibonacci.**

$$F(n) = \begin{cases} 1 & \text{Si } n \leq 2 \\ F(n-1) + F(n-2) & \text{Si } n > 2 \end{cases}$$

- **Con divide y vencerás.**

**operación Fibonacci (n: entero): entero**

**si  $n \leq 2$  entonces devolver 1**

**sino devolver Fibonacci(n-1) + Fibonacci(n-2)**

- **Con programación dinámica.**

**operación Fibonacci (n: entero): entero**

**T[1]:= 1; T[2]:= 1**

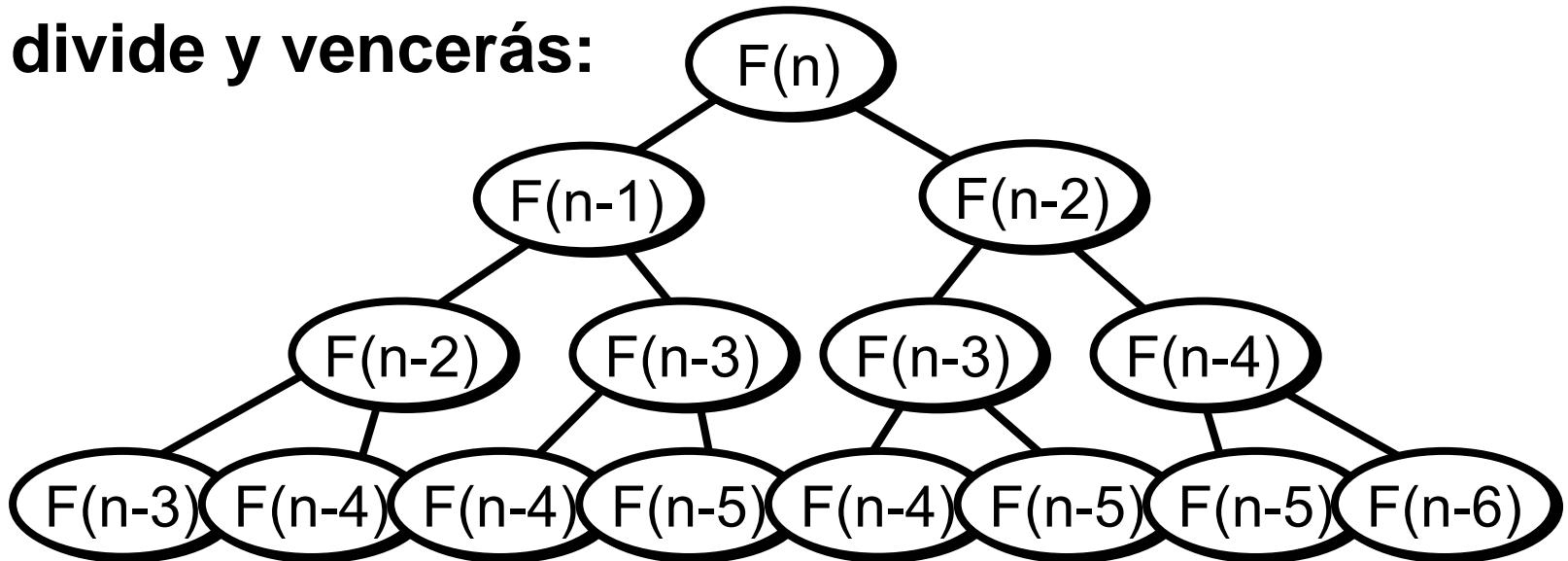
**para  $i := 3, \dots, n$  hacer**

**$T[i]:= T[i-1] + T[i-2]$**

**devolver  $T[n]$**

# Introducción a la PD

- Los dos usan la misma fórmula recursiva, aunque de forma distinta.
- ¿Cuál es más eficiente?
- **Con programación dinámica:**  $\Theta(n)$
- **Con divide y vencerás:**



- **Problema:** Muchos cálculos están repetidos  
(Solapamiento de los subproblemas)
- El tiempo de ejecución es exponencial:  $\Theta(1,62^n)$

# Introducción a la PD

- La base de la **programación dinámica** es el **razonamiento inductivo**: ¿cómo resolver un problema combinando soluciones para problemas más pequeños?
- La idea es la misma que en **divide y vencerás...** pero aplicando una estrategia distinta.
- **Similitud:**
  - Descomposición recursiva del problema.
  - Se obtiene aplicando un razonamiento inductivo.
- **Diferencia:**
  - Divide y vencerás: aplicar directamente la fórmula recursiva (programa recursivo).
  - Programación dinámica: resolver primero los problemas más pequeños, guardando los resultados en una tabla (programa iterativo).

# Introducción a la PD

## Métodos ascendentes y descendentes

- **Métodos descendentes (divide y vencerás)**
  - Empezar con el problema original y descomponer recursivamente en problemas de menor tamaño.
  - Partiendo del problema grande, descendemos hacia problemas más sencillos.
- **Métodos ascendentes (programación dinámica)**
  - Resolvemos primero los problemas pequeños (guardando las soluciones en una tabla). Después los vamos combinando para resolver los problemas más grandes.
  - Partiendo de los problemas pequeños avanzamos hacia los más grandes.

# Introducción a la PD

## Pasos para aplicar programación dinámica:

- 1) Obtener una **descomposición recurrente** del problema:
    - Ecuación recurrente.
    - Casos base.
  - 2) Definir la **estrategia** de aplicación de la fórmula:
    - Tablas utilizadas por el algoritmo.
    - Orden y forma de rellenarlas.
  - 3) Especificar cómo se **recompone la solución** final a partir de los valores de las tablas.
- 
- **Punto clave:** obtener la descomposición recurrente.
  - Requiere mucha “creatividad”...

# Introducción a la PD

- Cuestiones a resolver en el razonamiento inductivo:
  - ¿Cómo reducir un problema a subproblemas más simples?
  - ¿Qué parámetros determinan el **tamaño del problema** (es decir, cuándo el problema es “más simple”)?
- **Idea:** ver lo que ocurre al tomar una decisión concreta  
→ interpretar el problema como un proceso de toma de decisiones.
- **Ejemplo: Mochila 0/1.** Decisiones: coger o no coger un objeto dado.

# Introducción a la PD

- La programación dinámica se basa en el uso de **tablas** donde se almacenan los resultados parciales.
- En general, el **tiempo** será de la forma:  
Tamaño de la tabla\*Tiempo de llenar cada elemento de la tabla.
- Un aspecto **importante** es la memoria puede llegar a ocupar la tabla.
- Además, algunos de estos cálculos pueden ser innecesarios.

# Programación Dinámica

---

Uso de la programación dinámica:

1. Caracterizar la estructura de una solución óptima.
2. Definir de forma recursiva la solución óptima.
3. Calcular la solución óptima de forma ascendente.
4. Construir la solución óptima a partir de los datos almacenados al obtener soluciones parciales.

# Principio de Optimalidad

---

Para poder emplear programación dinámica, una secuencia óptima debe cumplir la condición de que cada una de sus subsecuencias también sea óptima:

Dado un problema P con n elementos,  
si la secuencia óptima es  $e_1e_2\dots e_k\dots e_n$  entonces:

- $e_1e_2\dots e_k$  es solución al problema P considerando los k primeros elementos.
- $e_{k+1}\dots e_n$  es solución al problema P considerando los elementos desde  $k+1$  hasta  $n$ .

# Principio de Optimalidad

---

En otras palabras:

La solución óptima de cualquier instancia no trivial de un problema es una combinación de las soluciones óptimas de sus subproblemas.

- Se busca la solución óptima a un problema como un proceso de decisión “multietápico”.
- Se toma una decisión en cada paso, pero ésta depende de las soluciones a los subproblemas que lo componen.

# Principio de Optimalidad

---

## Principio de Optimalidad de Bellman

[Bellman, R.E.: "Dynamic Programming". Princeton University Press, 1957]

"Una política óptima tiene la propiedad de que, sean cuales sea el estado inicial y la decisión inicial, las decisiones restantes deben constituir una solución óptima con respecto al estado resultante de la primera decisión".

En Informática, un problema que puede descomponerse de esta forma se dice que presenta subestructuras optimales (la base de los algoritmos greedy y de la programación dinámica).

# Principio de Optimalidad

---

## Principio de Optimalidad de Bellman

[Bellman, R.E.: "Dynamic Programming". Princeton University Press, 1957]

El principio de optimalidad se verifica si toda solución óptima a un problema está compuesta por soluciones óptimas de sus subproblemas.

**iOjo!**

El principio de optimalidad no nos dice que, si tenemos las soluciones óptimas de los subproblemas, entonces podamos combinarlas para obtener la solución óptima del problema original...

# Principio de Optimalidad

---

## Principio de Optimalidad de Bellman

[Bellman, R.E.: "Dynamic Programming". Princeton University Press, 1957]

Ejemplo: Cambio en monedas

- La solución óptima para 0.07 euros es 0.05 + 0.02 euros.
- La solución óptima para 0.06 euros es 0.05 + 0.01 euros.
- La solución óptima para 0.13 euros **no** es  $(0.05 + 2) + (0.05 + 0.01)$  euros.

Sin embargo, sí que existe alguna forma de descomponer 0.13 euros de tal forma que las soluciones óptimas a los subproblemas nos den una solución óptima (p.ej. 0.11 y 0.02 euros).

# Definición del problema...

---

Para aplicar programación dinámica:

1. Se comprueba que se cumple el principio de optimalidad de Bellman, para lo que hay que encontrar la “estructura” de la solución.
  
2. Se define recursivamente la solución óptima del problema (en función de los valores de las soluciones para subproblemas de menor tamaño).

# ... y cálculo de la solución óptima

---

3. Se calcula el valor de la solución óptima utilizando un enfoque ascendente:
  - Se determina el conjunto de subproblemas que hay que resolver (el tamaño de la tabla).
  - Se identifican los subproblemas con una solución trivial (casos base).
  - Se van calculando los valores de soluciones más complejas a partir de los valores previamente calculados.
  
4. Se determina la solución óptima a partir de los datos almacenados en la tabla.

# Programación Dinámica

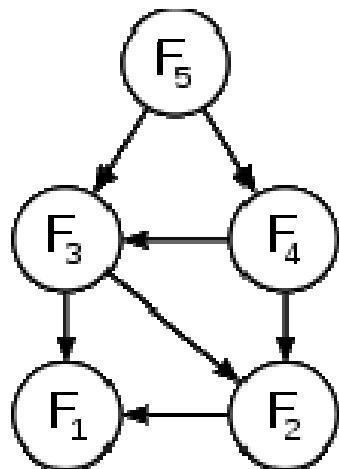
## Ejemplos

### Sucesión de Fibonacci

$$fib(n) = fib(n - 1) + fib(n - 2)$$

- Implementación recursiva:  $O(\varphi^n)$
- Implementación usando programación dinámica:  $\Theta(n)$

```
if (n == 0) return 0;
else if (n == 1) return 1;
else {
    previo = 0; actual = 1;
    for (i=1; i<n; i++) {
        fib = previo + actual;
        previo = actual; actual = fib;
    }
return actual;
}
```



# Programación Dinámica

## Ejemplos

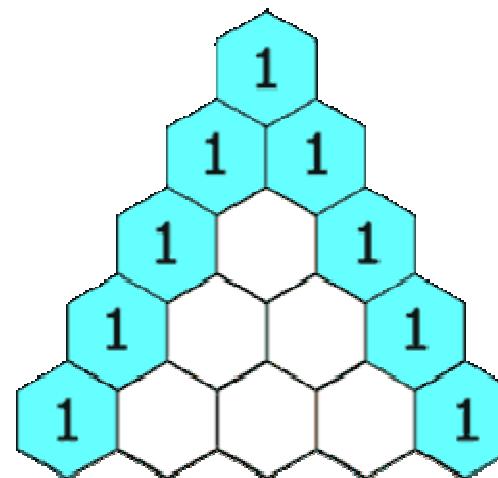
---

### Números combinatorios:

Combinaciones de n sobre p

- Implementación inmediata...  
$$\binom{n}{p} = \frac{n!}{p!(n-p)!}$$
- Implementación usando programación dinámica...  
Triángulo de Pascal

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



# Programación Dinámica

## Ejemplos

### Números combinatorios:

Combinaciones de  $n$  sobre  $p$

( $n$  elementos tomados de  $p$  en  $p$ )

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

	$p = 0$	$p = 1$	$p = 2$	$p = 3$	$p = 4$	$p = 5$	$p = 6$	$p = 7$	$p = 8$	$p = 9$
$n = 0$	1									
$n = 1$		1	1							
$n = 2$			1	2	1					
$n = 3$				1	3	3	1			
$n = 4$					1	4	6	4	1	
$n = 5$						1	5	10	10	5
$n = 6$							1	6	15	20
$n = 7$								1	21	35
$n = 8$									35	21
$n = 9$										1

Orden de eficiencia:  $\Theta(np)$

# Programación Dinámica

## Ejemplos

---

### Números combinatorios:

Combinaciones de n sobre p

```
int combinaciones (int n, int p)
{
    for (i=0; i<=n; i++) {
        for (j=0; j<=min(i,p); j++) {
            if ((j==0) || (j==i))
                c[i][j] = 1
            else
                c[i][j] = c[i-1][j-1]+c[i-1][j];
    }
    return c[n][p];
}
```

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

Orden de eficiencia:  $\Theta(np)$  en tiempo,  $\Theta(np)$  en espacio.

# Programación Dinámica

## Ejemplos

---

### Números combinatorios:

Combinaciones de n sobre p

```
int combinaciones (int n, int p)
{
    b[0] = 1;
    for (i=1; i<=n; i++) {
        b[i] = 1;
        for (j=i-1; j>0; j--) {
            b[j] += b[j - 1];
        }
    }
    return b[p];
}
```

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

Orden de eficiencia:  $\Theta(np)$  en tiempo,  $\Theta(n)$  en espacio.

# Programación Dinámica

## Devolver el cambio...

---

Existen casos para los que no se puede aplicar el algoritmo greedy (por ejemplo, devolver 8 peniques con monedas de 6, 4 y 1 penique).

### Definición recursiva de la solución (usando división entera)

$$cambio(\{m_1, \dots, m_i\}, C) = \min \begin{cases} cambio(\{m_1, \dots, m_{i-1}\}, C) \\ C / m_i + cambio(\{m_1, \dots, m_{i-1}\}, C \% m_i) \end{cases}$$

### Cálculo de la solución con programación dinámica:

- Orden de eficiencia proporcional al tamaño de la tabla,  $O(Cn)$ , donde  $n$  es el número de monedas distintas.

# Programación Dinámica

## Devolver el cambio...

$$cambio(\{m_1, \dots, m_i\}, C) = \min \begin{cases} cambio(\{m_1, \dots, m_{i-1}\}, C) \\ C / m_i + cambio(\{m_1, \dots, m_{i-1}\}, C \% m_i) \end{cases}$$

	0	1	2	3	4	5	6	7	8
{1}	0	1	2	3	4	5	6	7	8
{1,4}	0	1	2	3	1	2	3	4	2
{1,4,6}	0	1	2	3	1	2	1	2	2

# Programación Dinámica

## Multiplicación encadenada de matrices

---

Propiedad asociativa del producto de matrices:

Dadas las matrices  $A_1$  (10x100),  $A_2$  (100x5) y  $A_3$  (5x50),

- $(A_1 A_2) A_3$  implica 7500 multiplicaciones
- $A_1 (A_2 A_3)$  implica 75000 multiplicaciones

Parentizaciones posibles:

$$P(n) = \begin{cases} 1 & \text{si } n=1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{si } n>1 \end{cases}$$
$$\Omega\left(\frac{4^n}{n^2}\right)$$

# Programación Dinámica

## Multiplicación encadenada de matrices

---

Si cada matriz  $A_k$  tiene un tamaño  $p_{k-1}p_k$ ,  
el número de multiplicaciones necesario será:

$$m(1, n) = m(1, k) + m(k + 1, n) + p_0 p_k p_n$$

$$A_1 \times A_2 \times A_3 \times \dots \times A_k$$

x

$$A_{k+1} \times A_{k+2} \times \dots \times A_n$$

De forma general:

$$m(i, j) = m(i, k) + m(k + 1, j) + p_{i-1} p_k p_j$$

# Programación Dinámica

## Multiplicación encadenada de matrices

---

Matriz  $A_k$  de tamaño  $p_{k-1}p_k$

**Definición recursiva de la solución óptima:**

- Si  $i=j$ , entonces

$$m(i, j) = 0$$

- Si  $i \neq j$ , entonces

$$m(i, j) = \min_{i \leq k < j} \{ m(i, k) + m(k + 1, j) + p_{i-1}p_kp_j \}$$

$$A_1 \times A_2 \times A_3 \times \dots \times A_k$$

$\times$

$$A_{k+1} \times A_{k+2} \times \dots \times A_n$$

# Programación Dinámica

## Multiplicación encadenada de matrices

---

### Implementación:

- Tenemos  $n^2$  subproblemas distintos  $m(i,j)$ .
- Para calcular  $m(i,j)$  necesitamos los valores almacenados en la fila  $i$  y en la columna  $j$

$$m(i, j) = \min_k \{m(i, k) + m(k + 1, j) + p_{i-1} p_k p_j\}$$

■ Para calcular cada valor necesitamos  $O(n)$ , por lo que el algoritmo resultante es de orden  $O(n^3)$ .

# Programación Dinámica

## Multiplicación encadenada de matrices

### Implementación (índices de 1 a n):

```
for (i=1; i<=n; i++)
    m[i,i] = 0;

for (s=2; s<=n; s++) {
    for (i=1; i<=n-s+1; i++) {
        j = i+s-1;
        m[i,j] = ∞;
        for (k=i; k<=j-1; k++) {
            q = m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j]
            if (q < m[i,j]) {
                m[i,j] = q;
                s[i,j] = k;
            }
        }
    }
}
```

N	0	1	2	i	j	...	n-1
0	Yellow	Yellow	Yellow				Red
1		Yellow	Yellow	Yellow			
2			Yellow	Yellow	Blue		Red
3				Yellow	Blue	Yellow	Red
4					Blue	Yellow	Yellow
5						Blue	Yellow
6							Yellow
7							

# Programación Dinámica

## Multiplicación encadenada de matrices

---

### Implementación:

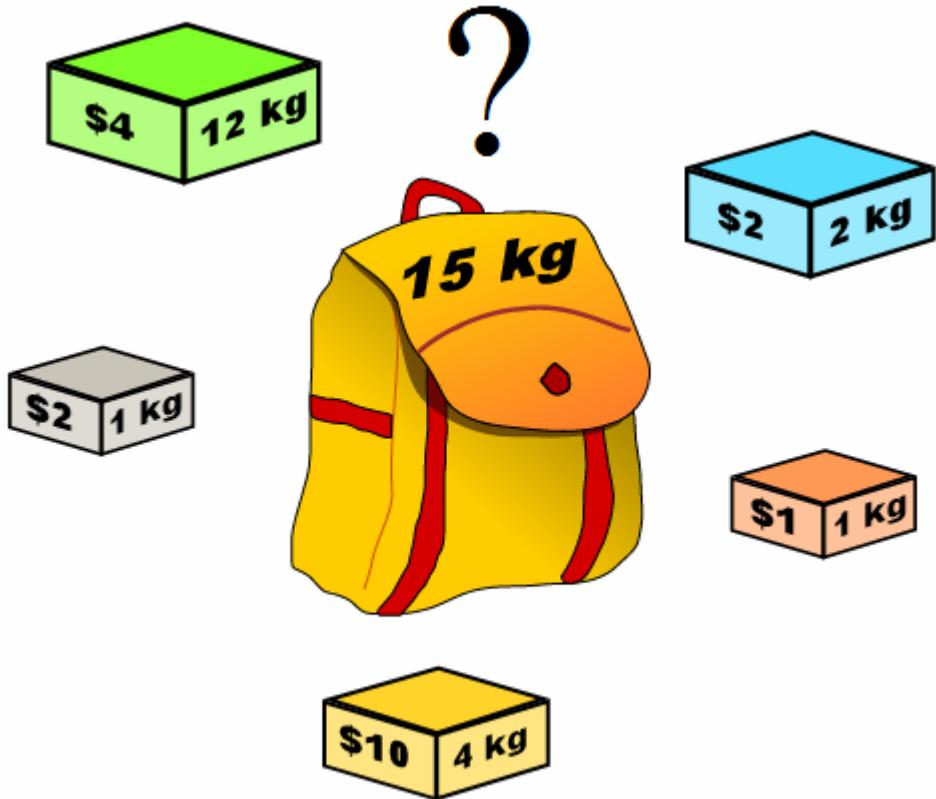
```
// Suponiendo que hemos calculado previamente s[i,j]...

MultiplicaCadenaMatrices (A, i, j)
{
    if (j>i) {
        x = MultiplicaCadenaMatrices (A, i, s[i,j]);
        y = MultiplicaCadenaMatrices (A, s[i,j]+1, j);
        return MultiplicaMatrices(x, y);
    } else {
        return A[i];
    }
}
```

# Programación Dinámica

## El problema de la mochila 0/1

Tenemos un conjunto  $S$  de  $n$  objetos, en el que cada objeto  $i$  tiene un beneficio  $b_i$  y un peso  $w_i$  positivos.



Objetivo: Seleccionar los elementos que nos garantizan un beneficio máximo pero con un peso global menor o igual que  $W$ .

# Programación Dinámica

## El problema de la mochila 0/1

---

Dado el conjunto  $S$  de  $n$  objetos,  
sea  $S_k$  el conjunto de los  $k$  primeros objetos (de 1 a  $k$ ):

Podemos definir  $B(k, w)$  como la ganancia de la mejor solución obtenida a partir de los elementos de  $S_k$  para una mochila de capacidad  $w$ .

Ahora bien, la mejor selección de elementos del conjunto  $S_k$  para una mochila de tamaño  $w$  se puede definir en función de selecciones de elementos de  $S_{k-1}$  para mochilas de menor capacidad...

# Programación Dinámica

## El problema de la mochila 0/1

---

¿Cómo calculamos  $B(k,w)$ ?

- O bien la mejor opción para  $S_k$  coincide con la mejor selección de elementos de  $S_{k-1}$  con peso máximo  $w$  (el beneficio máximo para  $S_k$  coincide con el de  $S_{k-1}$ ),
- o bien es el resultado de añadir el objeto  $k$  a la mejor selección de elementos de  $S_{k-1}$  con peso máximo  $w-w_k$  (el beneficio para  $S_k$  será el beneficio que se obtenía en  $S_{k-1}$  para una mochila de capacidad  $w-w_k$  más el beneficio  $b_k$  asociado al objeto  $k$ ).

# Programación Dinámica

## El problema de la mochila 0/1

---

Definición recursiva de  $B(k,w)$ :

$$B(k, w) = \begin{cases} B(k - 1, w) & \text{si } x_k = 0 \\ B(k - 1, w - w_k) + b_k & \text{si } x_k = 1 \end{cases}$$

Para resolver el problema de la mochila nos quedaremos con el máximo de ambos valores:

$$B(k, w) = \max \{ B(k - 1, w), B(k - 1, w - w_k) + b_k \}$$

# Programación Dinámica

## El problema de la mochila 0/1

---

Cálculo ascendente de  $B(k,w)$   
usando una matriz B de tamaño  $(n+1) \times (W+1)$ :

```
int[][] knapsack (W, w[1..n], b[1..n])
{
    for (p=0; p<=W; p++)
        B[0][p]=0;

    for (k=1; k<=n; k++) {
        for (p=0; p<w[k]; p++)
            B[k][p] = B[k-1][p];
        for (p=w[k]; p<=W; w++)
            B[k][p] = max ( B[k-1][p-w[k]]+b[k] , B[k-1][p] );
    }

    return B;
}
```

# Programación Dinámica

## El problema de la mochila 0/1

---

¿Cómo calculamos la solución óptima a partir de  $B(k,w)$ ?

Calculamos la solución para  $B[k][w]$   
utilizando el siguiente algoritmo:

Si  $B[k][w] == B[k-1][w]$ ,  
entonces el objeto k no se selecciona y se seleccionan los  
objetos correspondientes a la solución óptima para k-1  
objetos y una mochila de capacidad w:  
la solución para  $B[k-1][w]$ .

Si  $B[k][w] != B[k-1][w]$ ,  
se selecciona el objeto k  
y los objetos correspondientes a la solución óptima para k-1  
objetos y una mochila de capacidad  $w-w[k]$ :  
la solución para  $B[k-1][w-w[k]]$ .

# Programación Dinámica

## El problema de la mochila 0/1

---

### Eficiencia del algoritmo

Tiempo de ejecución:  $\Theta(n W)$

- “Pseudopolinómico” (no es polinómico sobre el tamaño de la entrada; esto es, sobre el número de objetos).
- El problema de la mochila es NP.

# Programación Dinámica

## El problema de la mochila 0/1

### Ejemplo

Mochila de tamaño  $W=11$

Número de objetos  $n=5$

Solución óptima  $\{3, 4\}$

Objeto	Valor	Peso
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

	0	1	2	3	4	5	6	7	8	9	10	11
$\phi$	0	0	0	0	0	0	0	0	0	0	0	0
$\{1\}$	0	1	1	1	1	1	1	1	1	1	1	1
$\{1, 2\}$	0	1	6	7	7	7	7	7	7	7	7	7
$\{1, 2, 3\}$	0	1	6	7	7	18	19	24	25	25	25	25
$\{1, 2, 3, 4\}$	0	1	6	7	7	18	22	24	28	29	29	40
$\{1, 2, 3, 4, 5\}$	0	1	6	7	7	18	22	28	29	34	34	40

# Subsecuencia Común de Mayor Longitud (LCS)

---

Aplicacion: comparar dos cadenas de DNA

Ej:  $X = \{A\ B\ C\ B\ D\ A\ B\}$ ,  $Y = \{B\ D\ C\ A\ B\ A\}$

Subsec. Común de Mayor longitud :

$X = A\ \textcolor{red}{B}\ \textcolor{red}{C}\ \textcolor{red}{B}\ D\ \textcolor{red}{A}\ B$

$Y = \textcolor{red}{B}\ D\ \textcolor{red}{C}\ A\ \textcolor{red}{B}\ \textcolor{red}{A}$

Un algoritmo de fuerza bruta compara cualquier subsecuencia de X con los símbolos de Y

# Algoritmo LCS

---

- Si  $|X| = m$ ,  $|Y| = n$ , entonces hay  $2^m$  subsecuencias de  $x$ ; y las debemos comparar con  $Y$  ( $n$  comparaciones)  $\rightarrow O(n 2^m)$
- Sin embargo, LCS tiene *subestructuras optimales*: las soluciones a los subproblemas son parte de la solución final.
- Subproblemas: “Encontrar LCS para pares de prefijos de  $X$  e  $Y$ ”

# Algoritmo LCS

---

- Definimos  $X_i$ ,  $Y_j$  los prefijos de X e Y de longitud  $i$  y  $j$  respectivamente
- Definimos  $c[i,j]$  la longitud de LCS para  $X_i$  e  $Y_j$
- Entonces, LCS de X e Y será  $c[m,n]$

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{si } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{en otro caso} \end{cases}$$

# Definición Recursiva

---

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{si } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{en caso contrario} \end{cases}$$

- *Inicio:*  $i = j = 0$  (subcadena vacía de x e y)

$$( c[0,0] = 0 )$$

- LCS de la cadena vacía y cualquier otra cadena es vacía, por tanto para cada par i,j:

$$c[0, j] = c[i, 0] = 0$$

# Definición Recursiva

---

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{si } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{en caso contrario} \end{cases}$$

- Cuando calculamos  $c[i, j]$ , consideramos dos casos:
- **Primer caso:**  $x[i] = y[j]$ : Se emparejan un símbolo más en X e Y.
- **Segundo caso:**  $x[i] \neq y[j]$ : *Como no se emparejan los símbolos, la longitud no se mejora y es la mejor entre  $LCS(X_i, Y_{j-1})$  y  $LCS(X_{i-1}, Y_j)$*

# Algoritmo LCS

## LCS-Length(X, Y)

- ```
1. m = length(X) // get the # of symbols in X
2. n = length(Y) // get the # of symbols in Y
3. for i = 0 to m    c[i,0] = 0 // special case:  $X_0$  e  $Y_0$ 
4. for j = 1 to n    c[0,j] = 0 // special case:  $X_0$ 
5. for i = 1 to m          // for all  $X_i$ 
6.   for j = 1 to n        // for all  $Y_j$ 
7.     if (  $X_i == Y_j$  )
8.       c[i,j] = c[i-1,j-1] + 1
9.     else c[i,j] = max( c[i-1,j], c[i,j-1] )
10. return c
```

# Ejemplo LCS

---

- $X = ABCB$
- $Y = BDCAB$

$$\text{LCS}(X, Y) = BCB$$

$X = A \ B \ C \ B$

$Y = \quad B \ D \ C \ A \ B$

# Ejemplo LCS (0)

ABCB  
BDCAB

| i | j  | 0  | 1 | 2 | 3 | 4 | 5 |
|---|----|----|---|---|---|---|---|
|   |    | Yj | B | D | C | A | B |
| 0 | Xi |    |   |   |   |   |   |
| 1 | A  |    |   |   |   |   |   |
| 2 | B  |    |   |   |   |   |   |
| 3 | C  |    |   |   |   |   |   |
| 4 | B  |    |   |   |   |   |   |

$$X = ABCB; \ m = |X| = 4$$
$$Y = BDCAB; \ n = |Y| = 5$$

# Ejemplo LCS (1)

ABCB  
BDCAB

| i | j  | 0 | 1 | 2 | 3 | 4 | 5 |
|---|----|---|---|---|---|---|---|
|   | Yj | B | D | C | A | B |   |
| 0 | Xi | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A  | 0 |   |   |   |   |   |
| 2 | B  | 0 |   |   |   |   |   |
| 3 | C  | 0 |   |   |   |   |   |
| 4 | B  | 0 |   |   |   |   |   |

for i = 1 to m  
for j = 1 to n

$c[i,0] = 0$   
 $c[0,j] = 0$

# Ejemplo LCS (2)

ABCB  
BDCAB

| i | j  | 0 | 1 | 2 | 3 | 4 | 5 |
|---|----|---|---|---|---|---|---|
|   | Yj | 0 | B | D | C | A | B |
| 0 | Xi | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A  | 0 | 0 |   |   |   |   |
| 2 | B  | 0 |   |   |   |   |   |
| 3 | C  | 0 |   |   |   |   |   |
| 4 | B  | 0 |   |   |   |   |   |

if (  $X_i == Y_j$  )  
 $c[i,j] = c[i-1,j-1] + 1$   
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$

# Ejemplo LCS (3)

ABCB  
BDCAB

| i | j  | 0 | 1 | 2 | 3 | 4 | 5 |
|---|----|---|---|---|---|---|---|
|   | Yj | B | D | C | A | B |   |
| 0 | Xi | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A  | 0 | 0 | 0 | 0 |   |   |
| 2 | B  | 0 |   |   |   |   |   |
| 3 | C  | 0 |   |   |   |   |   |
| 4 | B  | 0 |   |   |   |   |   |

```
if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

# Ejemplo LCS (4)

ABCB  
BDCAB

| i | j  | 0 | 1 | 2 | 3 | 4 | 5 |
|---|----|---|---|---|---|---|---|
|   | Yj | B | D | C | A | B |   |
| 0 | Xi | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A  | 0 | 0 | 0 | 0 | 1 |   |
| 2 | B  | 0 |   |   |   |   |   |
| 3 | C  | 0 |   |   |   |   |   |
| 4 | B  | 0 |   |   |   |   |   |

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# Ejemplo LCS (5)

ABC  
BDCAB

| i | j  | 0 | 1 | 2 | 3 | 4 | 5 |
|---|----|---|---|---|---|---|---|
|   | Yj | B | D | C | A | B |   |
| 0 | Xi | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A  | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | B  | 0 |   |   |   |   |   |
| 3 | C  | 0 |   |   |   |   |   |
| 4 | B  | 0 |   |   |   |   |   |

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# Ejemplo LCS (6)

ABC  
BDCAB

| i | j  | 0 | 1 | 2 | 3 | 4 | 5 |
|---|----|---|---|---|---|---|---|
|   | Yj | 0 | B | D | C | A | B |
| 0 | Xi | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A  | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | B  | 0 | 1 |   |   |   |   |
| 3 | C  | 0 |   |   |   |   |   |
| 4 | B  | 0 |   |   |   |   |   |

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

A B C B  
B D C A B

# Ejemplo LCS (7)

| i | j  | 0 | 1 | 2 | 3 | 4 | 5 |
|---|----|---|---|---|---|---|---|
|   | Yj | B | D | C | A | B |   |
| 0 | Xi | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A  | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | B  | 0 | 1 | 1 | 1 | 1 |   |
| 3 | C  | 0 |   |   |   |   |   |
| 4 | B  | 0 |   |   |   |   |   |

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# Ejemplo LCS (8)

ABC  
BDCAB

| i | j  | 0 | 1 | 2 | 3 | 4 | 5 |
|---|----|---|---|---|---|---|---|
|   | Yj | B | D | C | A | B |   |
| 0 | Xi | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A  | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | B  | 0 | 1 | 1 | 1 | 1 | 2 |
| 3 | C  | 0 |   |   |   |   |   |
| 4 | B  | 0 |   |   |   |   |   |

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# Ejemplo LCS (10)

ABC  
BDCAB

| i | j  | 0 | 1 | 2 | 3 | 4 | 5 |
|---|----|---|---|---|---|---|---|
|   | Yj |   | B | D | C | A | B |
| 0 | Xi | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A  | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | B  | 0 | 1 | 1 | 1 | 1 | 2 |
| 3 | C  | 0 | 1 | 1 |   |   |   |
| 4 | B  | 0 |   |   |   |   |   |

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# Ejemplo LCS (11)

ABC  
BDCAB

| i | j  | 0 | 1 | 2 | 3 | 4 | 5 |
|---|----|---|---|---|---|---|---|
|   | Yj | B | D | C | A | B |   |
| 0 | Xi | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A  | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | B  | 0 | 1 | 1 | 1 | 1 | 2 |
| 3 | C  | 0 | 1 | 1 | 2 |   |   |
| 4 | B  | 0 |   |   |   |   |   |

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# Ejemplo LCS (12)

ABC  
BDCAB

| i | j  | 0 | 1 | 2 | 3 | 4 | 5 |
|---|----|---|---|---|---|---|---|
|   | Yj | B | D | C | A | B |   |
| 0 | Xi | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A  | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | B  | 0 | 1 | 1 | 1 | 1 | 2 |
| 3 | C  | 0 | 1 | 1 | 2 | 2 | 2 |
| 4 | B  | 0 |   |   |   |   |   |

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# Ejemplo LCS (13)

ABCB  
BDCAB

| i | j  | 0 | 1 | 2 | 3 | 4 | 5 |
|---|----|---|---|---|---|---|---|
|   | Yj | 0 | B | D | C | A | B |
| 0 | Xi | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A  | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | B  | 0 | 1 | 1 | 1 | 1 | 2 |
| 3 | C  | 0 | 1 | 1 | 2 | 2 | 2 |
| 4 | B  | 0 | 1 |   |   |   |   |

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# Ejemplo LCS (14)

ABCB  
BDCAB

| i | j  | 0 | 1 | 2 | 3 | 4 | 5 |
|---|----|---|---|---|---|---|---|
|   | Yj | B | D | C | A | B |   |
| 0 | Xi | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A  | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | B  | 0 | 1 | 1 | 1 | 1 | 2 |
| 3 | C  | 0 | 1 | 1 | 2 | 2 | 2 |
| 4 | B  | 0 | 1 | 1 | 2 | 2 |   |

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# Ejemplo LCS (15)

ABCB  
BDCAB

| i | j  | 0 | 1 | 2 | 3 | 4 | 5 |
|---|----|---|---|---|---|---|---|
|   | Yj | B | D | C | A | B |   |
| 0 | Xi | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A  | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | B  | 0 | 1 | 1 | 1 | 1 | 2 |
| 3 | C  | 0 | 1 | 1 | 2 | 2 | 2 |
| 4 | B  | 0 | 1 | 1 | 2 | 2 | 3 |

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# Cómo encontrar la subsecuencia LCS

---

Cada  $c[i,j]$  depende de  $c[i-1,j]$  y  $c[i,j-1]$

O bien  $c[i-1, j-1]$

Por tanto, a partir del valor  $c[i,j]$  podremos averiguar  
cómo se determinó

|   |   |
|---|---|
| 2 | 2 |
| 2 | 3 |

Por ejemplo  
 $c[i,j] = c[i-1,j-1] + 1 = 2+1=3$

# Cómo encontrar la subsecuencia LCS

---

- Como

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- Empezamos desde  $c[m,n]$  y vamos hacia atrás
- Si  $c[i,j] = c[i-1, j-1] + 1$  y  $x[i]=x[j]$ , guardamos  $x[i]$  (porque  $x[i]$  pertenece a LCS)
- En otro caso seguimos por  $\max(c[i,j-1],c[i-1,j])$
- Si  $i=0$  o  $j=0$  (alcanzamos el principio), devolve-mos los caracteres almacenados en orden inverso.

# Encontramos LCS

| i | j  | 0 | 1     | 2 | 3     | 4 | 5 |
|---|----|---|-------|---|-------|---|---|
|   | Yj | B | D     | C | A     | B |   |
| 0 | Xi | 0 | 0     | 0 | 0     | 0 | 0 |
| 1 | A  | 0 | 0     | 0 | 0     | 1 | 1 |
| 2 | B  | 0 | 1 ← 1 | 1 | 1     | 1 | 2 |
| 3 | C  | 0 | 1     | 1 | 2 ← 2 | 2 | 2 |
| 4 | B  | 0 | 1     | 1 | 2     | 2 | 3 |

# Encontramos LCS

| i | j  | 0 | 1     | 2 | 3     | 4 | 5 |
|---|----|---|-------|---|-------|---|---|
|   | Yj | B | D     | C | A     | B |   |
| 0 | Xi | 0 | 0     | 0 | 0     | 0 | 0 |
| 1 | A  | 0 | 0     | 0 | 0     | 1 | 1 |
| 2 | B  | 0 | 1 ← 1 | 1 | 1     | 1 | 2 |
| 3 | C  | 0 | 1     | 1 | 2 ← 2 | 2 | 2 |
| 4 | B  | 0 | 1     | 1 | 2     | 2 | 3 |

LCS :

B C B

# Programación Dinámica

## Selección de actividades con pesos

---

### Enunciado del problema

Dado un conjunto C de n tareas o actividades, con

$s_i$  = tiempo de comienzo de la actividad i

$f_i$  = tiempo de finalización de la actividad i

$v_i$  = valor (o peso) de la actividad i

encontrar el subconjunto S de actividades compatibles de peso máximo (esto es, un conjunto de actividades que no se solapen en el tiempo y que, además, nos proporcione un valor máximo).

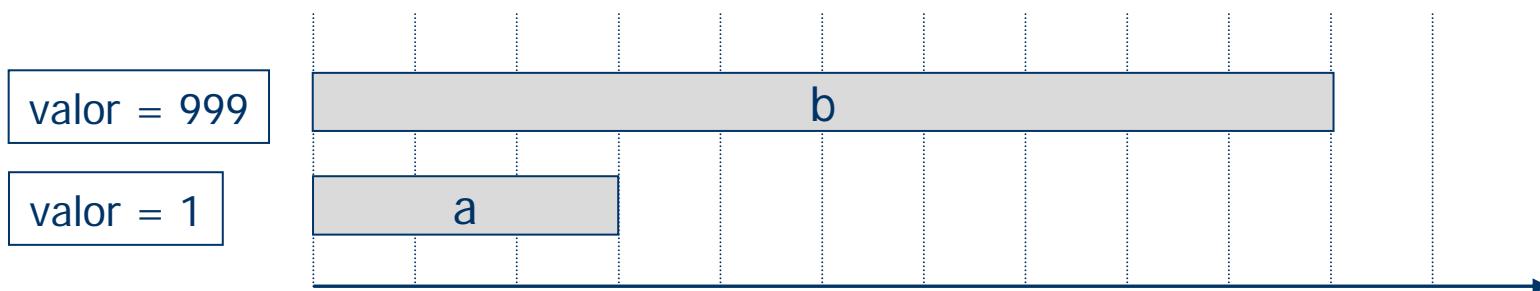
# Programación Dinámica

## Selección de actividades con pesos

### Recordatorio

Existe un algoritmo greedy para este problema cuando todas las actividades tienen el mismo valor (elegir las actividades en orden creciente de hora de finalización).

Sin embargo, el algoritmo greedy no funciona en general:

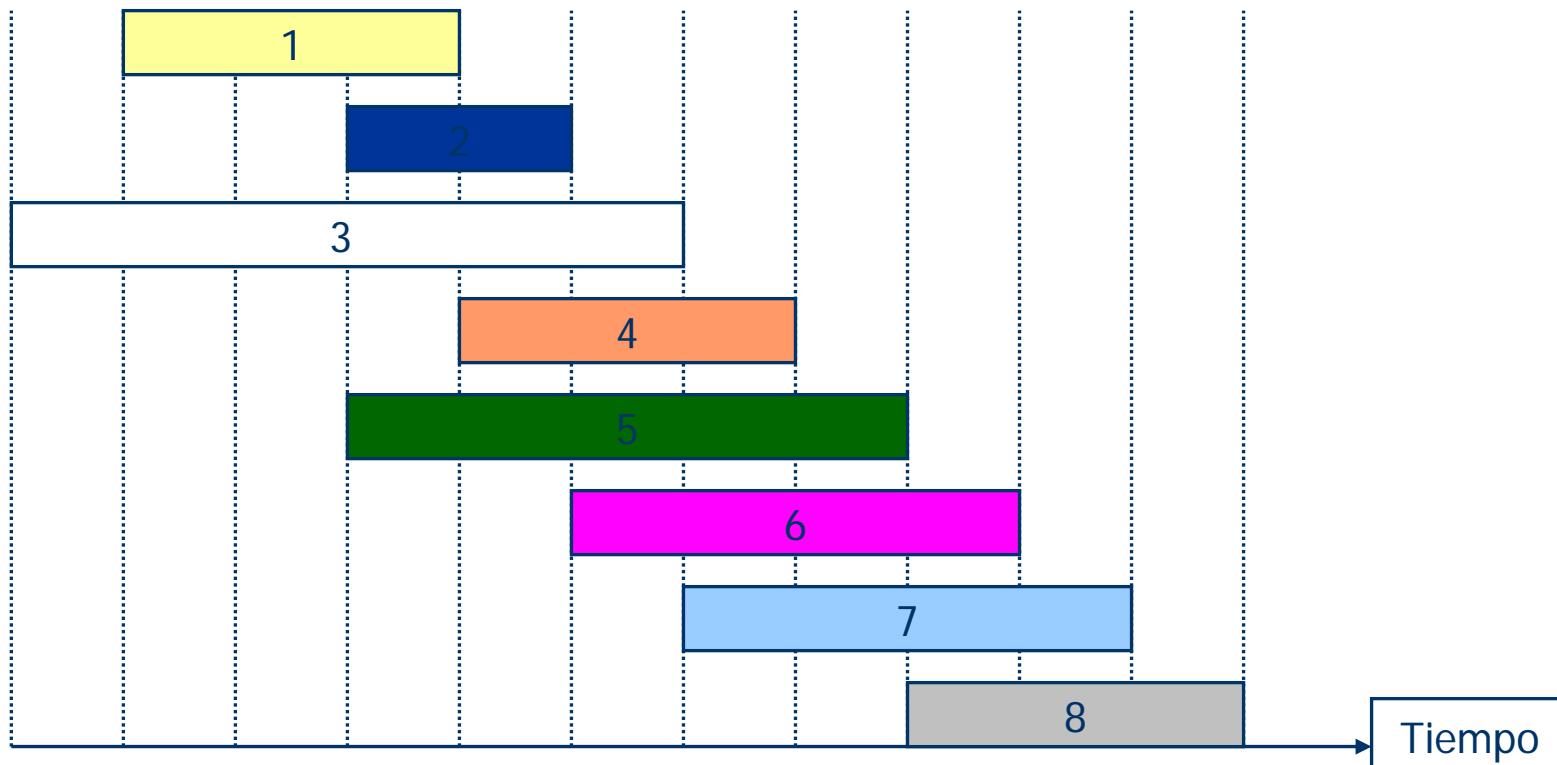


# Programación Dinámica

## Selección de actividades con pesos

### Observación

Si, como en el algoritmo greedy, ordenamos las actividades por su hora de finalización...

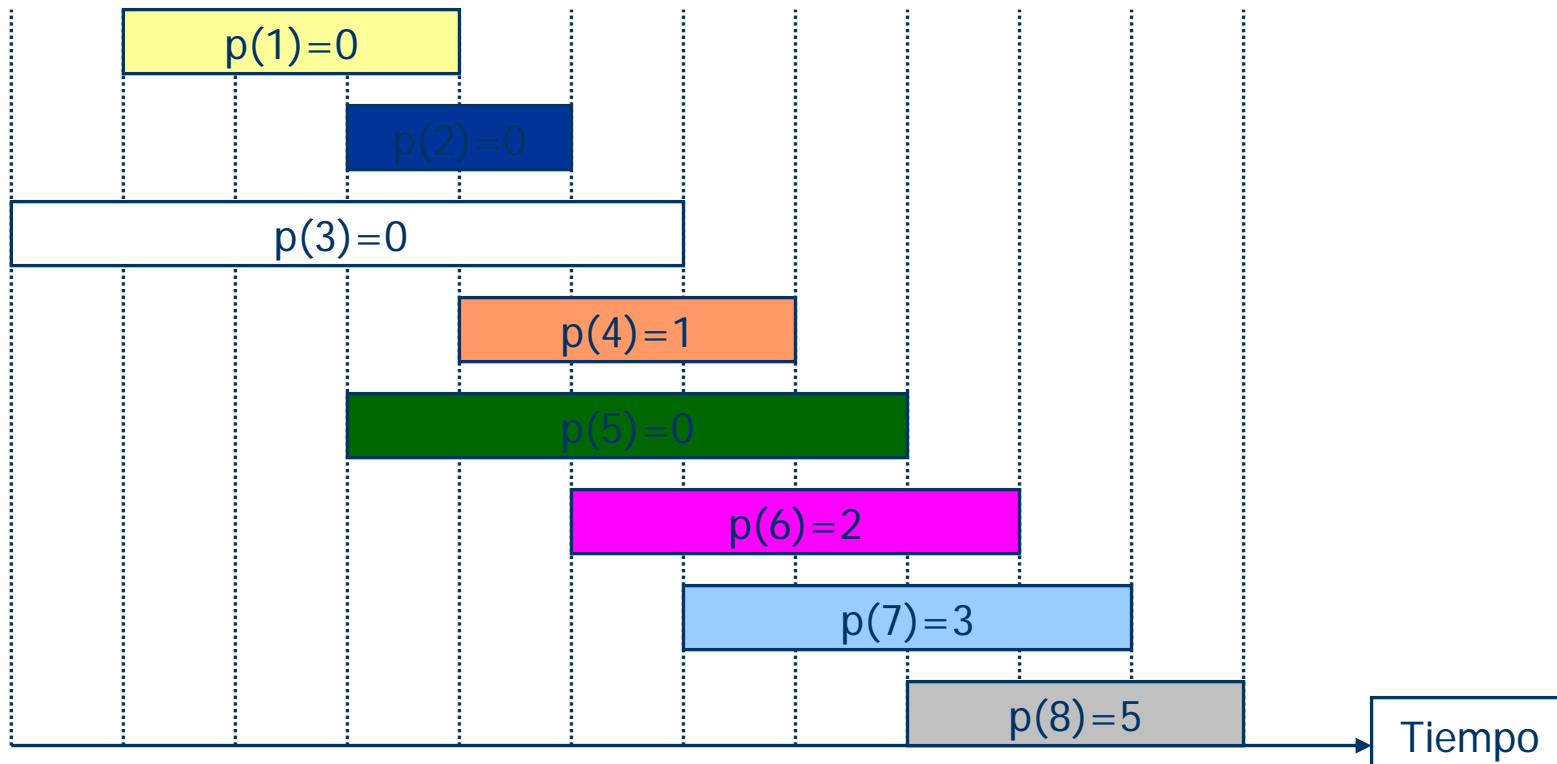


# Programación Dinámica

## Selección de actividades con pesos

### Observación

... podemos definir  $p(j)$  como el mayor índice  $i < j$  tal que la actividad  $i$  es compatible con la actividad  $j$



# Programación Dinámica

## Selección de actividades con pesos

### Definición recursiva de la solución

$$OPT(j) = \begin{cases} 0 & \text{si } j = 0 \\ \max\{v(j) + OPT(p(j)), OPT(j-1)\} & \text{si } j > 0 \end{cases}$$

- Caso 1: Se elige la actividad j.
  - No se pueden escoger actividades incompatibles  $>p(j)$ .
  - La solución incluirá la solución óptima para  $p(j)$ .
- Caso 2: No se elige la actividad j.
  - La solución coincidirá con la solución óptima para las primeras  $(j-1)$  actividades.

# Programación Dinámica

## Selección de actividades con pesos

### Implementación iterativa del algoritmo

```
SelecciónActividadesConPesos (C: actividades): S
{
    Ordenar C según tiempo de finalización;      // O(n log n)
    Calcular p[1]..p[n];                          // O(n log n)

    mejor[0] = 0;                                // O(1)
    for (i=1; i<=n, i++)
        mejor[i] = max ( valor[i]+mejor[p[i]], 
                           mejor[i-1] );

    return Solución(mejor);                      // O(n)
}
```

**Nota:** Faltaría el algoritmo “Solución()” que determinase cuál es la solución óptima para una lista de actividades. Dicho algoritmo chequearía los valores almacenados en “mejor” de manera lineal. Dicho algoritmo queda como ejercicio.

# Programación Dinámica

## Distancia de edición

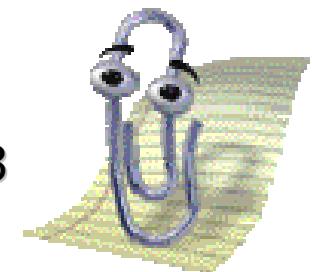
También conocida como distancia Levenshtein, mide la diferencia entre dos cadenas s y t como el número mínimo de operaciones de edición que hay que realizar para convertir una cadena en otra:

$$d(\text{"data mining"}, \text{"data minino"}) = 1$$

$$d(\text{"efecto"}, \text{"defecto"}) = 1$$

$$d(\text{"poda"}, \text{"boda"}) = 1$$

$$d(\text{"night"}, \text{"natch"}) = d(\text{"natch"}, \text{"noche"}) = 3$$



Aplicaciones: Correctores ortográficos, reconocimiento de voz, detección de plagios, análisis de ADN...

Para datos binarios: Distancia de Hamming.

# Programación Dinámica

## Distancia de edición

### Definición recursiva de la solución

$$d(i, j) = \begin{cases} d(i-1, j-1) & \text{si } s[i] = t[j] \\ 1 + \min\{d(i-1, j), d(i, j-1), d(i-1, j-1)\} & \text{si } s[i] \neq t[j] \end{cases}$$

### CASOS

- Mismo carácter:  $d(i-1, j-1)$
- Inserción:  $1 + d(i-1, j)$
- Borrado:  $1 + d(i, j-1)$
- Modificación:  $1 + d(i-1, j-1)$

# Programación Dinámica

## Distancia de edición

```
int LevenshteinDistance (string s[1..m], string t[1..n])
{
    for (i=0; i<=m; i++) d[i,0]=i;
    for (j=0; j<=n; j++) d[0,j]=j;
    for (j=1; j<=n; j++)
        for (i=1; i<=m; i++)
            if (s[i]==t[j])
                d[i,j] = d[i-1, j-1]
            else
                d[i,j] = 1+ min(d[i-1, j],d[i, j-1],d[i-1, j-1]);
    return d[m,n];
}
```

**Nota:** Faltaría el algoritmo que determinase cuál es la solución óptima para dos cadenas dadas. Dicho algoritmo puede encontrarse en los ejercicios de PD resueltos (material adicional).

$$d(i, j) = \begin{cases} d(i-1, j-1) & \text{si } s[i] = t[j] \\ 1 + \min\{d(i-1, j), d(i, j-1), d(i-1, j-1)\} & \text{si } s[i] \neq t[j] \end{cases}$$

# Programación Dinámica

## Caminos mínimos: Algoritmo de Floyd

---

### **Problema:**

Calcular el camino más corto que une cada par de vértices de un grafo, considerando que no hay pesos negativos.

### **Posibles soluciones:**

- Por fuerza bruta (de orden exponencial).
- Aplicar el algoritmo de Dijkstra para cada vértice.
- Algoritmo de Floyd (programación dinámica).

# Programación Dinámica

## Caminos mínimos: Algoritmo de Floyd

---

**Definición recursiva de la solución:**

$D_k(i,j)$ : Camino más corto de  $i$  a  $j$  usando sólo los  $k$  primeros vértices del grafo como puntos intermedios.

Expresión recursiva:

$$D_k(i, j) = \min\{D_{k-1}(i, j), D_{k-1}(i, k) + D_{k-1}(k, j)\}$$

Caso base:

$$D_0(i, j) = c_{ij}$$

# Programación Dinámica

## Caminos mínimos: Algoritmo de Floyd

**Algoritmo de Floyd (1962):  $\Theta(V^3)$**

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        D[i][j] = coste(i,j);

for (k=0; k<n; k++)
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            if (D[i][k] + D[k][j] < D[i][j] )
                D[i][j] = D[i][k] + D[k][j];
```

**Nota:** Faltaría el algoritmo que determinase cuál es la solución óptima para un origen y destino dados. Dicho algoritmo puede encontrarse en los ejercicios de PD resueltos (material adicional).

$$D_k(i, j) = \min\{D_{k-1}(i, j), D_{k-1}(i, k) + D_{k-1}(k, j)\}$$

# Programación Dinámica

## Caminos mínimos: Floyd Vs. Dijkstra

---

Si sólo nos interesan los caminos mínimos desde un vértice concreto del grafo  $G(V,A)$ , podemos utilizar el algoritmo greedy de Dijkstra, de orden  **$O(A \log V)$** , siempre y cuando tengamos **pesos no negativos**.

En caso de necesitar todos los caminos, Floyd es siempre  **$\Theta(V^3)$**  mientras Dijkstra necesita ser ejecutado  $V$  veces, siendo  **$O(VA \log V)$** . Si el grafo es muy denso  $A \approx V^2$ , mientras que si es poco denso  $A \approx V$ . El mejor algoritmo depende por tanto de la densidad del grafo.

# Programación dinámica.

## Conclusiones

- El **razonamiento inductivo** es una herramienta muy potente en resolución de problemas.
- Aplicable no sólo en problemas de optimización.
- ¿Cómo obtener la fórmula? Interpretar el problema como una serie de **toma de decisiones**.
- Descomposición recursiva no necesariamente implica implementación recursiva.
- **Programación dinámica:** almacenar los resultados en una tabla, empezando por los tamaños pequeños y avanzando hacia los más grandes.

# Algorítmica

---

**Tema 1. Planteamiento General**

**Tema 2. La Eficiencia de los Algoritmos**

**Tema 3. Algoritmos “Divide y vencerás”**

**Tema 4. Algoritmos Voraces (“Greedy”)**

**Tema 5. Algoritmos basados en Programación Dinámica**

**Tema 6. Algoritmos para la Exploración de Grafos**

**(“Backtracking”, “Branch and Bound”)**

**Tema 7. Otras metodologías algorítmicas**

# Algorítmica

---

**Tema 1. Planteamiento General**

**Tema 2. La Eficiencia de los Algoritmos**

**Tema 3. Algoritmos “Divide y vencerás”**

**Tema 4. Algoritmos Voraces (“Greedy”)**

**Tema 5. Algoritmos basados en Programación Dinámica**

**Tema 6. Algoritmos para la Exploración de Grafos**

**(“Backtracking”, “Branch and Bound”)**

**Tema 7. Otras metodologías algorítmicas**

# Tema 5: Algoritmos para la Exploración de Grafos (BK y BB)

---

## Bibliografía:

- G. BRASSARD, P. BRATLEY. Fundamentos de Algoritmia. Prentice Hall (1997).  
E. HOROWITZ, S. SAHNI, S. RAJASEKARAN. Computer Algorithms. Computer Science Press (1998).

# Objetivos

---

- Comprender la filosofía de diseño de algoritmos *Backtracking* (“vuelta atrás”) y *Branch and Bound* (“ramifica y poda”)
- Conocer las características de un problema resoluble mediante dichas técnicas
- Resolución de diversos problemas

# Índice

---

- I. LA TÉCNICA BACKTRACKING**
  
- II. SOLUCIONES BACKTRACKING EN DISTINTOS PROBLEMAS**
  
- III. MÉTODOS BRANCH-BOUND**
  
- IV. SOLUCIONES BRANCH-BOUND EN DISTINTOS PROBLEMAS**

# Índice

---

## I. LA TÉCNICA BACKTRACKING

### 1. Introducción: El método General

- Resolución de problemas cuando la solución se puede expresar como una n-tupla
- Ejemplo: El problema de las 8 reinas
- Ejemplo: La suma de subconjuntos

### 2. Espacio de soluciones: Organización del Árbol

- Ejemplo: Espacio solución para el problema de las N-reinas ( $N=4$ )
- Ejemplo: Espacio solución para la suma de subconjuntos
- Terminología utilizada para la organización en árbol
- Ejemplo: *Backtracking* en el problema de las 4 reinas

### 3. Procedimiento *Backtracking*

- Procedimiento Iterativo
- Procedimiento Recursivo

# Método general

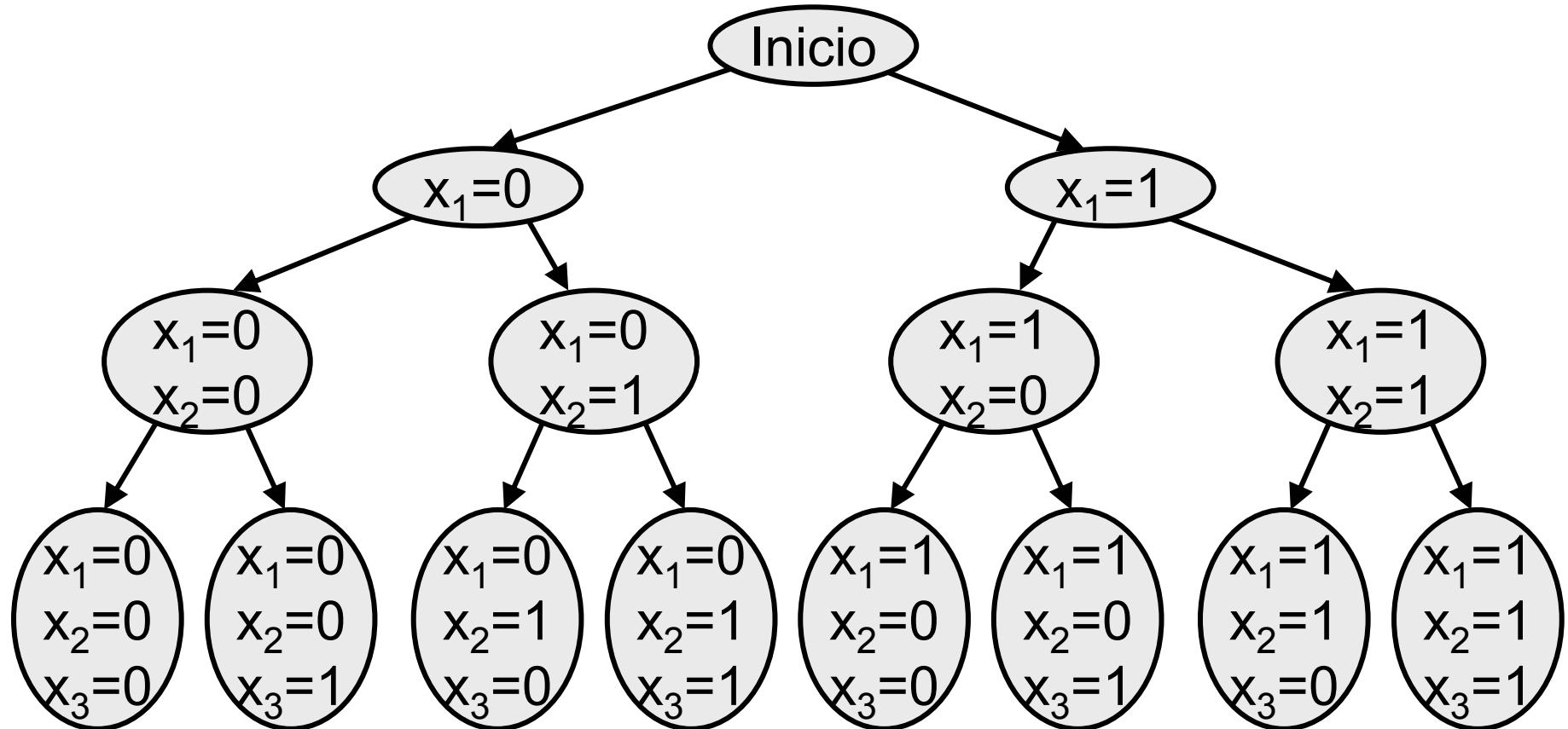
- El **backtracking** (o método de **retroceso** o **vuelta atrás**) es una técnica general de resolución de problemas, aplicable en problemas de **optimización**, **juegos** y otros tipos.
- El **backtracking** realiza una búsqueda exhaustiva y sistemática en el espacio de soluciones. Por ello, suele resultar muy ineficiente.
- Se puede entender como “opuesto” a avance rápido:
  - **Avance rápido:** añadir elementos a la solución y no deshacer ninguna decisión tomada.
  - **Backtracking:** añadir y quitar todos los elementos. Probar todas las combinaciones.

# Método general

- Una **solución** se puede expresar como una tupla:  $(x_1, x_2, \dots, x_n)$ , satisfaciendo unas restricciones y tal vez optimizando cierta función objetivo.
- En cada momento, el algoritmo se encontrará en cierto nivel **k**, con una solución parcial  $(x_1, \dots, x_k)$ .
  - Si se puede añadir un nuevo elemento a la solución  $x_{k+1}$ , se genera y se avanza al nivel **k+1**.
  - Si no, se prueban otros valores para  $x_k$ .
  - Si no existe ningún valor posible por probar, entonces se retrocede al nivel anterior **k-1**.
  - Se sigue hasta que la solución parcial sea una solución completa del problema, o hasta que no queden más posibilidades por probar.

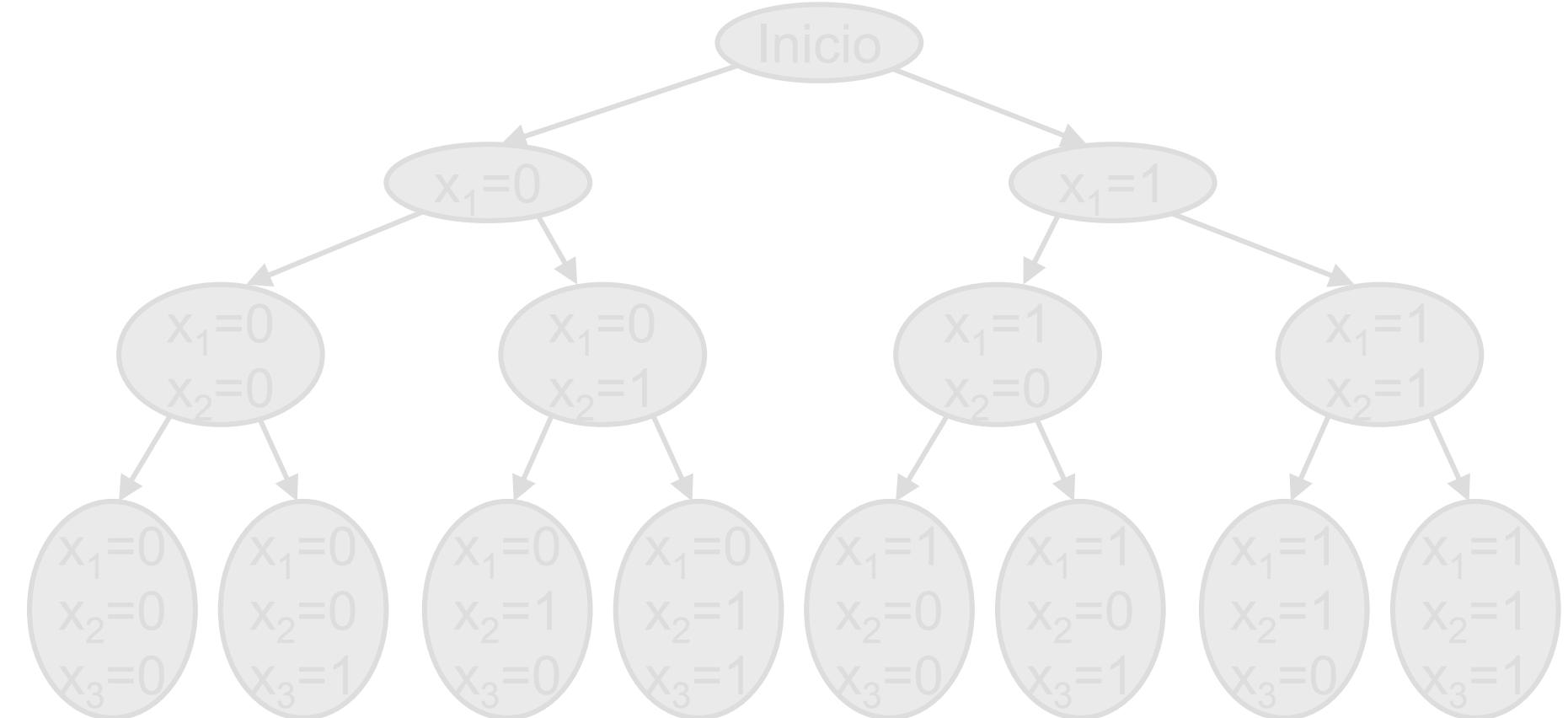
# Método general

- El resultado es equivalente a hacer un **recorrido en profundidad** en el árbol de soluciones.



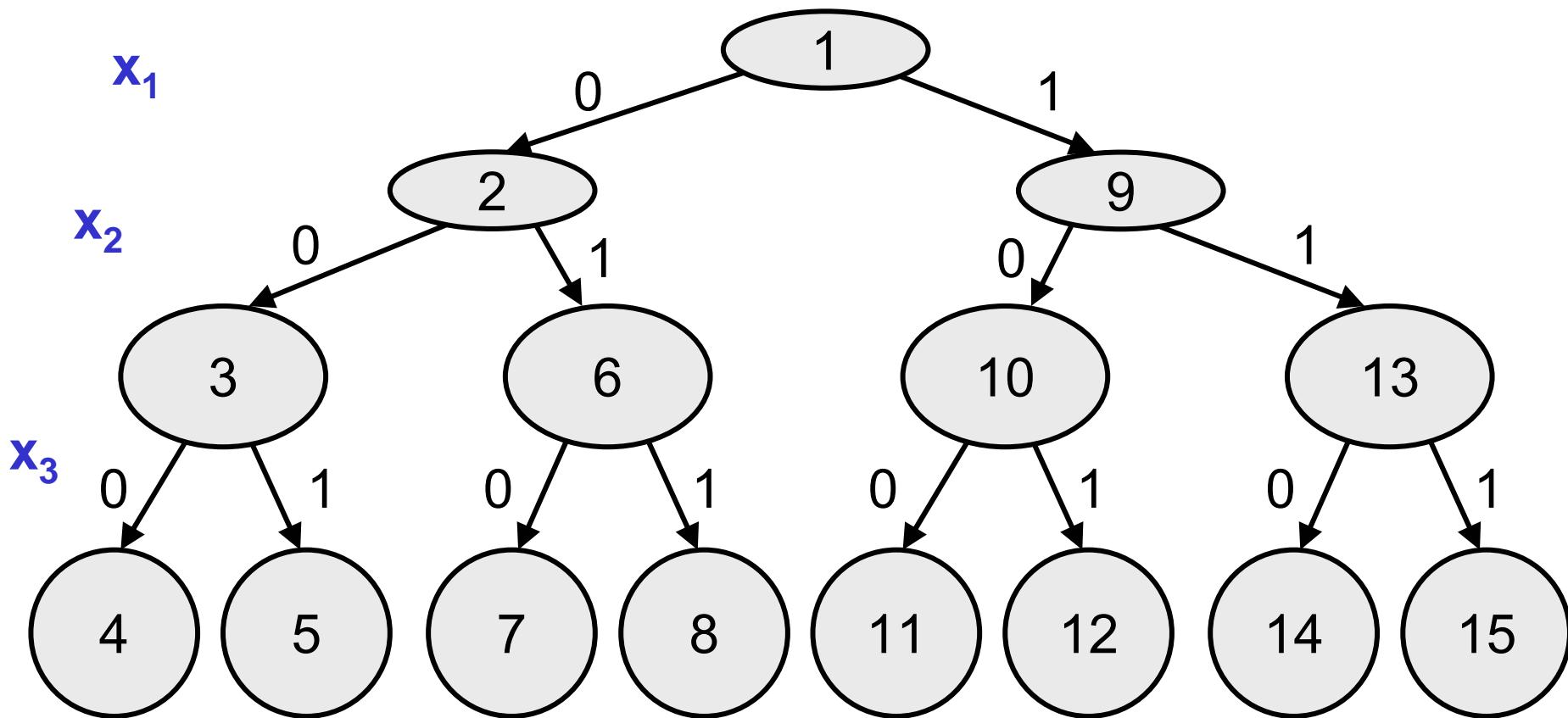
# Método general

- El resultado es equivalente a hacer un **recorrido en profundidad** en el árbol de soluciones.



# Método general

- Representación simplificada del árbol.



# Método general

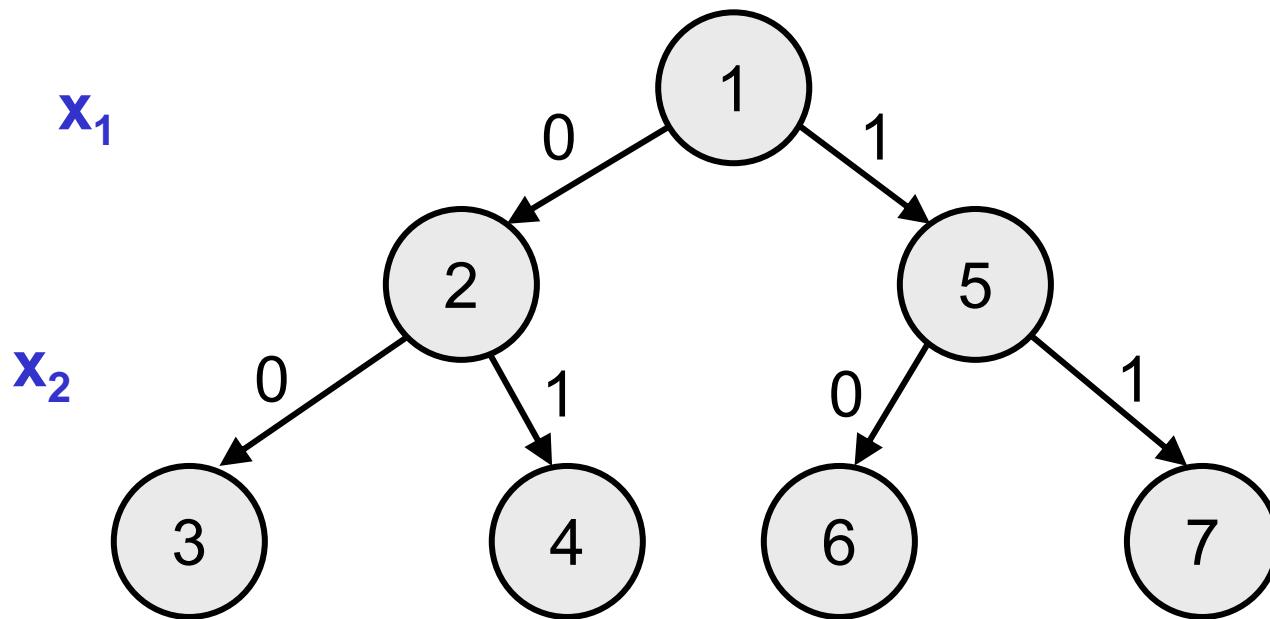
- **Árboles de backtracking:**
  - El árbol es simplemente una forma de representar la ejecución del algoritmo.
  - Es **implícito**, no almacenado (no necesariamente).
  - El recorrido es en **profundidad**, normalmente de izquierda a derecha.
  - La primera decisión para aplicar backtracking: ¿cómo es la forma del árbol?
  - **Preguntas relacionadas:** ¿Qué significa cada valor de la tupla solución ( $x_1, \dots, x_n$ )? ¿Cómo es la representación de la solución al problema?

# Método general

- Tipos comunes de árboles de backtracking:
  - Árboles binarios.
  - Árboles n-arios.
  - Árboles permutacionales.
  - Árboles combinatorios.

# Método general

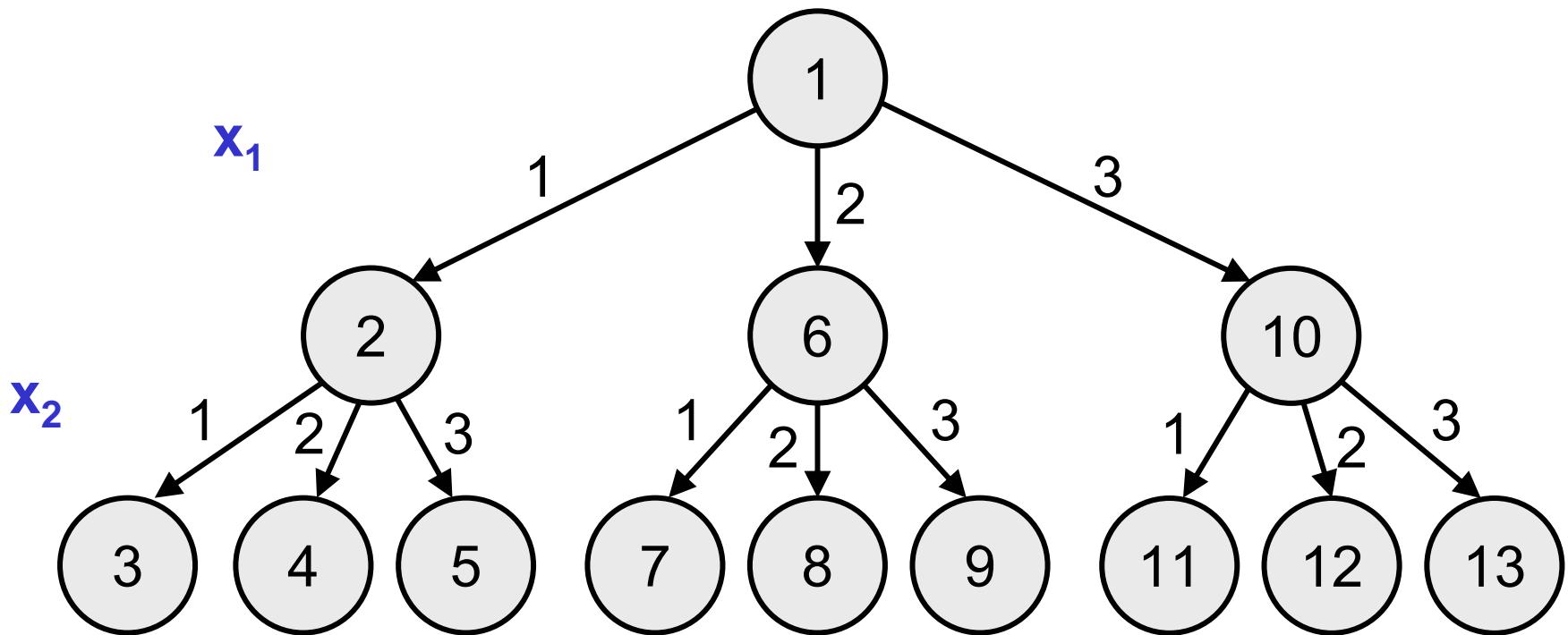
- **Árboles binarios:**  $s = (x_1, x_2, \dots, x_n)$ , con  $x_i \in \{0, 1\}$



- **Tipo de problemas:** elegir ciertos elementos de entre un conjunto, sin importar el orden de los elementos.
  - Problema de la mochila 0/1.
  - Encontrar un subconjunto de  $\{12, 23, 1, 8, 33, 7, 22\}$  que sume exactamente 50.

# Método general

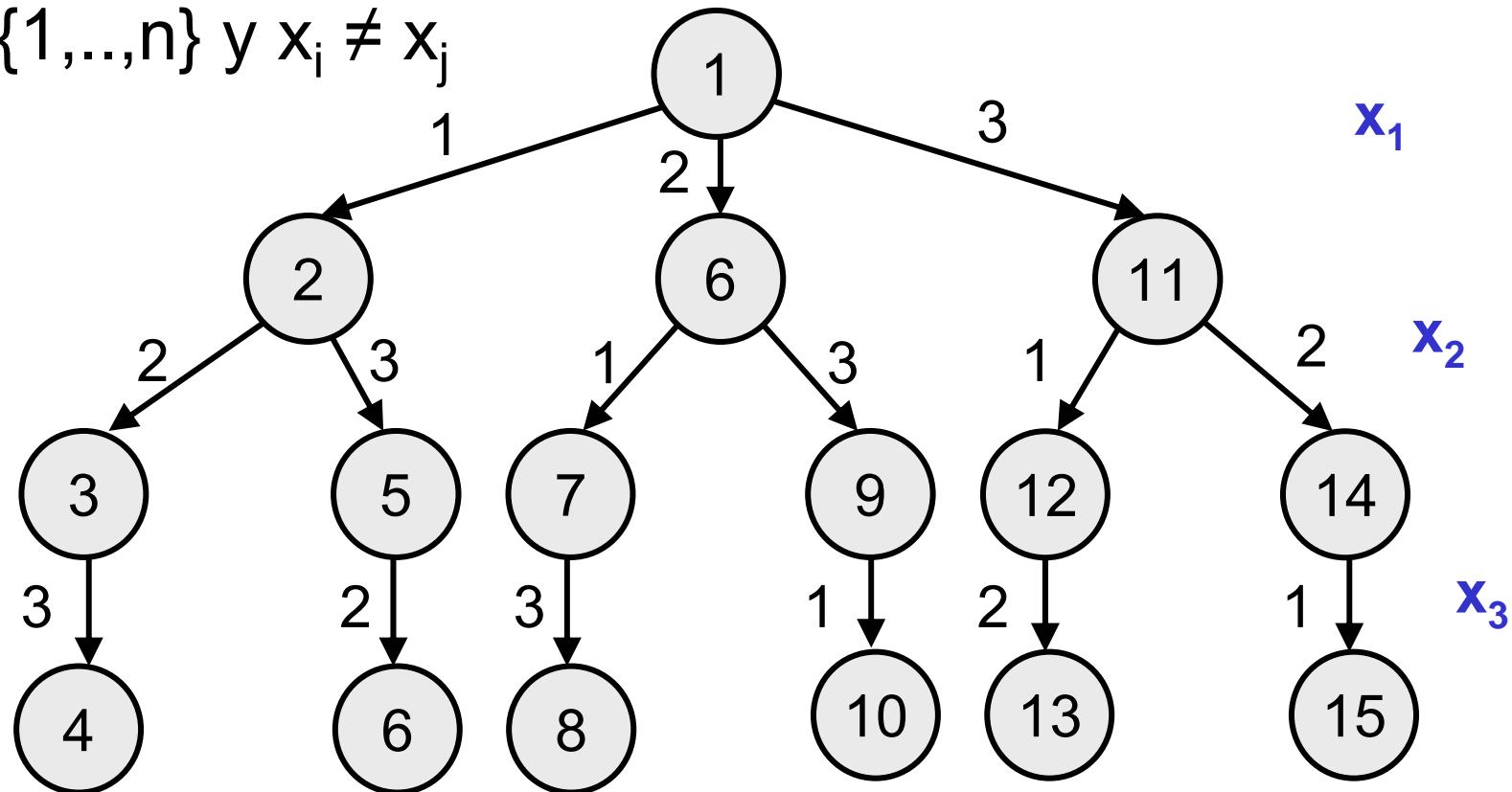
- **Árboles k-arios:**  $s = (x_1, x_2, \dots, x_n)$ , con  $x_i \in \{1, \dots, k\}$



- **Tipo de problemas:** varias opciones para cada  $x_i$ .
  - Problema del cambio de monedas.
  - Problema de las  $n$  reinas.

# Método general

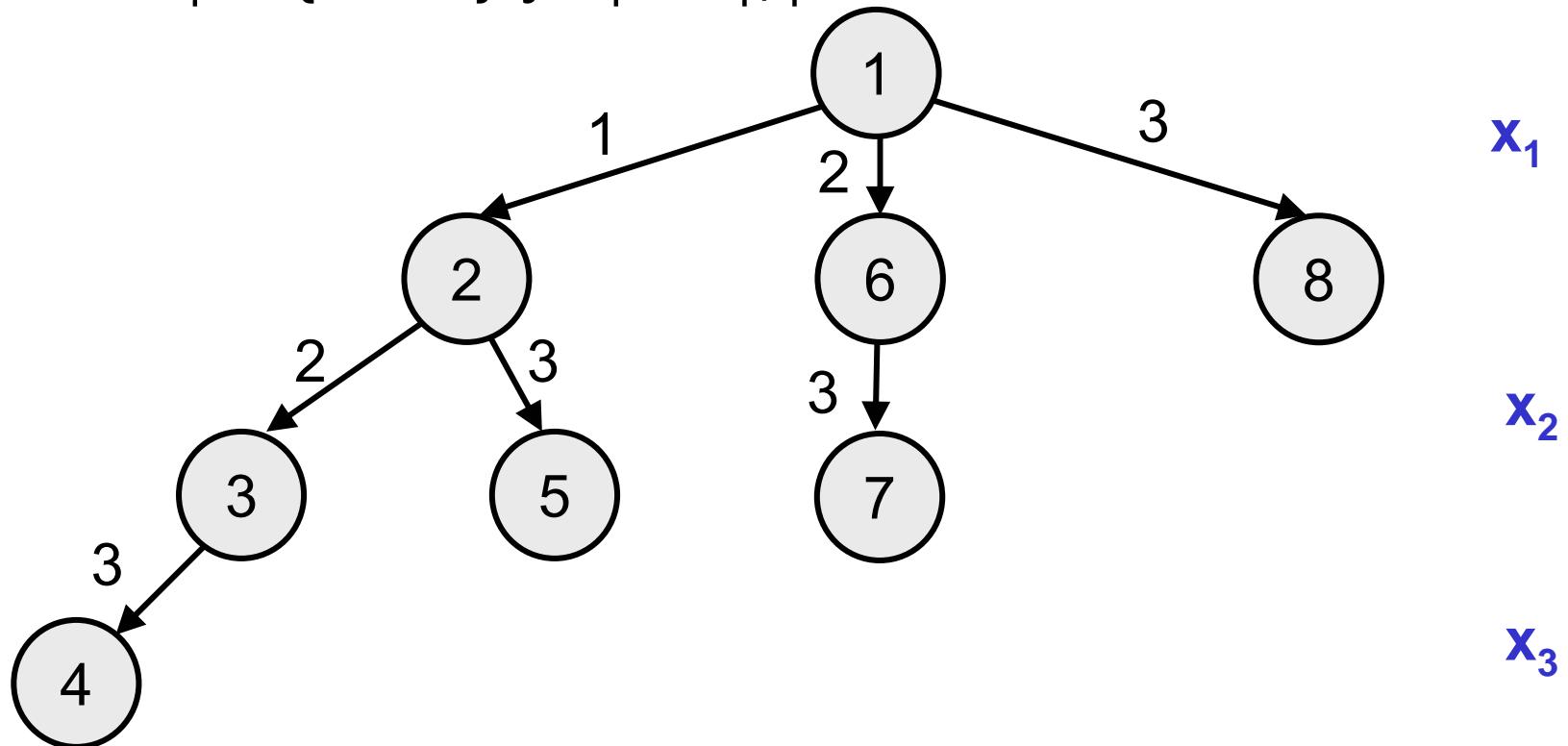
- **Árboles permutacionales:**  $s = (x_1, x_2, \dots, x_n)$ , con  $x_i \in \{1, \dots, n\}$  y  $x_i \neq x_j$



- **Tipo de problemas:** los  $x_i$  no se pueden repetir.
  - Generar todas las permutaciones de  $(1, \dots, n)$ .
  - Asignar  $n$  trabajos a  $n$  personas, asignación uno-a-uno.

# Método general

- **Árboles combinatorios:**  $s = (x_1, x_2, \dots, x_m)$ , con  $m \leq n$ ,  $x_i \in \{1, \dots, n\}$  y  $x_i < x_{i+1}$



- **Tipo de problemas:** los mismos que con árb. binarios.
  - Binario:  $(0, 1, 0, 1, 0, 0, 1) \rightarrow$  Combinatorio:  $(2, 4, 7)$

# Método general

## Cuestiones a resolver antes de programar:

- ¿Qué tipo de árbol es adecuado para el problema?
  - ¿Cómo es la representación de la solución?
  - ¿Cómo es la tupla solución? ¿Qué indica cada  $x_i$  y qué valores puede tomar?
- ¿Cómo generar un recorrido según ese árbol?
  - Generar un nuevo nivel.
  - Generar los hermanos de un nivel.
  - Retroceder en el árbol.
- ¿Qué ramas se pueden descartar por no conducir a soluciones del problema?
  - Poda por restricciones del problema.
  - Poda según el criterio de la función objetivo.

# Método general

- **Esquema general (no recursivo).** Problema de satisfacción de restricciones: buscamos cualquier solución que cumpla cierta propiedad, y se supone que existe alguna.

## Backtracking (var s: TuplaSolución)

nivel:= 1

s:=  $s_{INICIAL}$

fin:= false

repetir

**Generar (nivel, s)**

    si **Solución (nivel, s)** entonces

        fin:= true

    sino si **Criterio (nivel, s)** entonces

        nivel:= nivel + 1

    sino mientras NOT **MasHermanos (nivel, s)** hacer

**Retroceder (nivel, s)**

hasta fin

# Método general

- **Variables:**
  - **s**: Almacena la solución parcial hasta cierto punto.
  - **s<sub>INICIAL</sub>**: Valor de inicialización.
  - **nivel**: Indica el nivel actual en el que se encuentra el algoritmo.
  - **fin**: Valdrá **true** cuando hayamos encontrado alguna solución.
- **Funciones:**
  - **Generar (nivel, s)**: Genera el siguiente hermano, o el primero, para el **nivel** actual.
  - **Solución (nivel, s)**: Comprueba si la tupla (s[1], ..., s[nivel]) es una solución válida para el problema.

# Método general

- **Funciones:**
  - **Criterio (nivel, s):** Comprueba si a partir de ( $s[1], \dots, s[nivel]$ ) se puede alcanzar una solución válida. En otro caso se rechazarán todos los descendientes (**poda**).
  - **MasHermanos (nivel, s):** Devuelve **true** si hay más hermanos del nodo actual que todavía no han sido generados.
  - **Retroceder (nivel, s):** Retrocede un nivel en el árbol de soluciones. Disminuye en 1 el valor de **nivel**, y posiblemente tendrá que actualizar la solución actual, quitando los elementos retrocedidos.
- Además, suele ser común utilizar variables temporales con el valor actual (beneficio, peso, etc.) de la tupla solución.

# Método general

- **Ejemplo de problema:** Encontrar un subconjunto del conjunto  $T = \{t_1, t_2, \dots, t_n\}$  que sume exactamente  $P$ .
- **Variables:**
  - Representación de la solución con un árbol binario.
  - $s$ : array [1..n] de {-1, 0, 1}
    - $s[i] = 0 \rightarrow$  el número i-ésimo no se utiliza
    - $s[i] = 1 \rightarrow$  el número i-ésimo sí se utiliza
    - $s[i] = -1 \rightarrow$  valor de inicialización (número i-ésimo no estudiado)
  - $s_{INICIAL}$ : (-1, -1, ..., -1)
  - **fin**: Valdrá **true** cuando se haya encontrado solución.
  - **tact**: Suma acumulada hasta ahora (inicialmente 0).

# Método general

## Funciones:

- **Generar (nivel, s)**

$s[nivel]:= s[nivel] + 1$

**si**  $s[nivel]==1$  **entonces**  $tact:= tact + t_{nivel}$

- **Solución (nivel, s)**

**devolver** ( $nivel==n$ ) **Y** ( $tact==P$ )

- **Criterio (nivel, s)**

**devolver** ( $nivel<n$ ) **Y** ( $tact\leq P$ )

- **MasHermanos (nivel, s)**

**devolver**  $s[nivel] < 1$

- **Retroceder (nivel, s)**

$tact:= tact - t_{nivel} * s[nivel]$

$s[nivel]:= -1$

$nivel:= nivel - 1$

# Método general

- **Algoritmo:** ¡el mismo que el esquema general!

## **Backtracking (var s: TuplaSolución)**

nivel:= 1

s:= s<sub>INICIAL</sub>

fin:= false

**repetir**

    Generar (nivel, s)

**si** Solución (nivel, s) **entonces**

        fin:= true

**sino si** Criterio (nivel, s) **entonces**

        nivel:= nivel + 1

**sino**

**mientras** NOT MasHermanos (nivel, s) **hacer**

            Retroceder (nivel, s)

**finsi**

**hasta fin**

# Método general

## **Variaciones** del esquema general:

- 1) ¿Y si no es seguro que exista una solución?
- 2) ¿Y si queremos almacenar todas las soluciones (no sólo una)?
- 3) ¿Y si el problema es de optimización (maximizar o minimizar)?

# Método general

- **Caso 1)** Puede que no exista ninguna solución.

## Backtracking (var s: TuplaSolución)

nivel:= 1

s:=  $s_{INICIAL}$

fin:= false

**repetir**

    Generar (nivel, s)

**si** Solución (nivel, s) **entonces**

        fin:= true

**sino si** Criterio (nivel, s) **entonces**

        nivel:= nivel + 1

**sino**

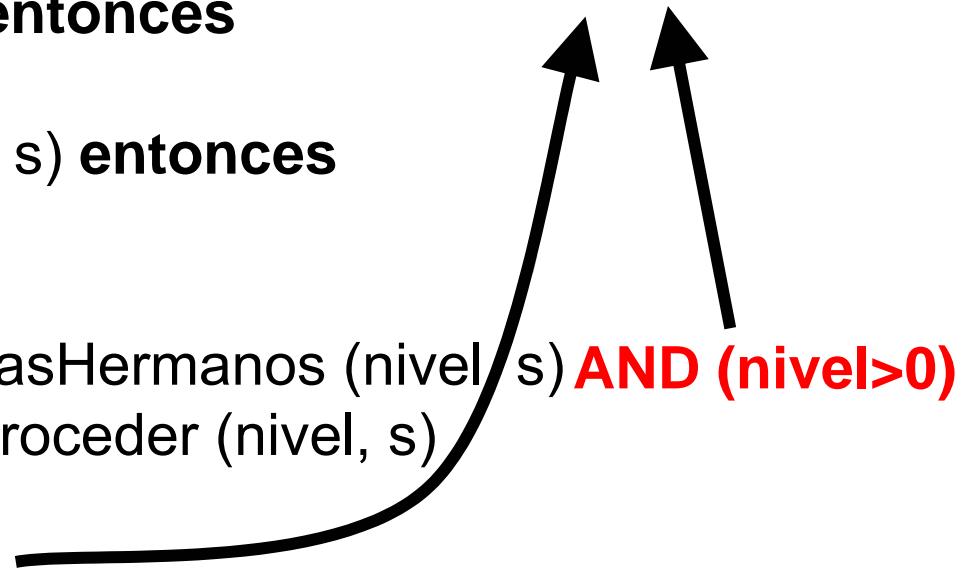
**mientras** NOT MasHermanos (nivel, s) **AND (nivel>0)**

**hacer** Retroceder (nivel, s)

**finsi**

**hasta fin OR (nivel==0)**

Para poder generar  
todo el árbol de  
backtracking



# Método general

- **Caso 2)** Queremos almacenar todas las soluciones.

## Backtracking (var s: TuplaSolución)

nivel:= 1

s:= s<sub>INICIAL</sub>

fin:= false

**repetir**

    Generar (nivel, s)

**si** Solución (nivel, s) **entonces**

**Almacenar (nivel, s)**

**si** Criterio (nivel, s) **entonces**

        nivel:= nivel + 1

**sino**

**mientras** NOT MasHermanos (nivel, s) **AND (nivel>0)**

**hacer** Retroceder (nivel, s)

**finsi**

**hasta** **nivel==0**

- En algunos problemas los nodos intermedios pueden ser soluciones
  - O bien, retroceder después de encontrar una solución



# Método general

- **Caso 3)** Problema de optimización (maximización).

## Backtracking (var s: TuplaSolución)

nivel:= 1

s:=  $s_{INICIAL}$

**voa:=  $-\infty$ ; soa:=  $\emptyset$**

repetir

Generar (nivel, s)

**si Solución (nivel, s) AND Valor(s) > voa entonces**

**voa:= Valor(s); soa:= s**

**si Criterio (nivel, s) entonces**

nivel:= nivel + 1

**sino**

**mientras NOT MasHermanos (nivel, s) AND (nivel>0)**

**hacer Retroceder (nivel, s)**

**finsi**

**hasta nivel==0**

**voa:** valor óptimo actual

**soa:** solución óptima actual

# Método general

- **Ejemplo de problema:** Encontrar un subconjunto del conjunto  $T = \{t_1, t_2, \dots, t_n\}$  que sume exactamente  $P$ , usando el menor número posible de elementos.
- **Funciones:**
  - **Valor(s)**  
**devolver**  $s[1] + s[2] + \dots + s[n]$
  - ¡Todo lo demás no cambia!
- **Otra posibilidad:** incluir una nueva variable:  
**vact: entero.** Número de elementos en la tupla actual.
  - **Inicialización** (añadir):  $vact := 0$
  - **Generar** (añadir):  $vact := vact + s[nivel]$
  - **Retroceder** (añadir):  $vact := vact - s[nivel]$

# *Backtracking: Resumen*

---

- **Si** tenemos que tomar una serie de decisiones entre una gran variedad de opciones donde,
  - *No tenemos suficiente información* como para saber cuál elegir
  - Cada decisión nos lleva a un nuevo conjunto de decisiones
  - Alguna sucesión de decisiones (pero posiblemente más de una) pueden ser solución de nuestro problema
- **Entonces** necesitamos un método de búsqueda de esas sucesiones que conduzca a encontrar una que nos convenga

# *Backtracking: Resumen*

---

- Características del problema:
  - La solución debe poder expresarse como una n-tupla,
$$(x_1, x_2, x_3, \dots, x_n),$$
donde cada  $x_i$  es seleccionado de un conjunto finito  $S_i$
  - El problema se (re)formula como la búsqueda de aquella tupla que maximiza (minimiza) un determinado criterio  $P(x_1, \dots, x_n)$
- *Backtracking* es un método de **búsqueda sistemática** de la solución óptima al problema

# Backtracking Vs Fuerza Bruta

---

## FUERZA BRUTA

- Problema:
  - Generar todas las posibles combinaciones de  $n$  bits
- Aplicaciones:
  - Selección de elementos en un conjunto
    - Selección de actividades
    - Mochila
    - Etc.

|       |
|-------|
| 00000 |
| 00001 |
| 00010 |
| 00011 |
| 00100 |
| 00101 |
| 00110 |
| 00111 |
| ..... |

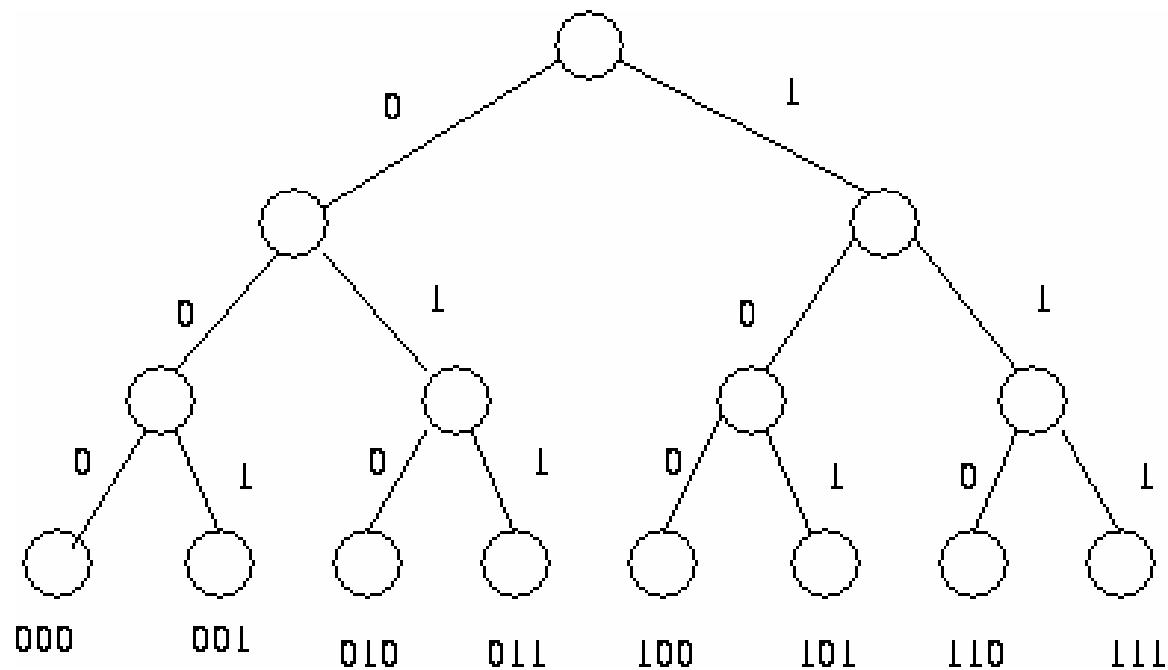
# Backtracking Vs Fuerza Bruta

## FUERZA BRUTA: ANÁLISIS DE EFICIENCIA

- Ecuación de Recurrencia,

$$T(n) = 2 T(n-1) + 1, \text{ es del orden } O(2^n)$$

Arbol de  
Recurrencia



# *Backtracking Vs Fuerza Bruta*

---

## ¿Qué hemos visto?

- Se impone una **estructura de árbol** sobre el conjunto de posibles soluciones (espacio de soluciones)
- La forma en la que se generan las soluciones es equivalente a realizar un **recorrido en pre-orden** del árbol, el espacio de soluciones
- Se procesan las hojas (que se corresponden con soluciones completas)
- Pregunta:
  - ¿Se puede mejorar el proceso? ¿Cuando? ¿Cómo?

*BK y BB !!!!*

# *Backtracking*

---

- ¿Se puede mejorar el proceso?
  - Sí, eliminando la necesidad de alcanzar una hoja para procesar
- ¿Cuándo?
  - Cuando para un nodo interno del árbol podemos asegurar que no alcanzamos una solución (no nos lleva a nodos hoja útiles), entonces **podemos podar la rama**
- ¿Cómo?
  - Realizamos una vuelta atrás (*backtracking*)

VENTAJA: Alcanzamos la misma solución con menos pasos

# *Diferencias con otras Técnicas*

---

- En los algoritmos *greedy* se construye la solución buscada, aprovechando la posibilidad de calcularla a trozos. Pero, con *backtracking* la elección de un sucesor en una etapa no implica su elección definitiva
- El tipo de problemas con el que estamos tratando **no se puede dividir en subproblemas independientes**. No es aplicable divide y vencerás

# Backtracking: Notación

---

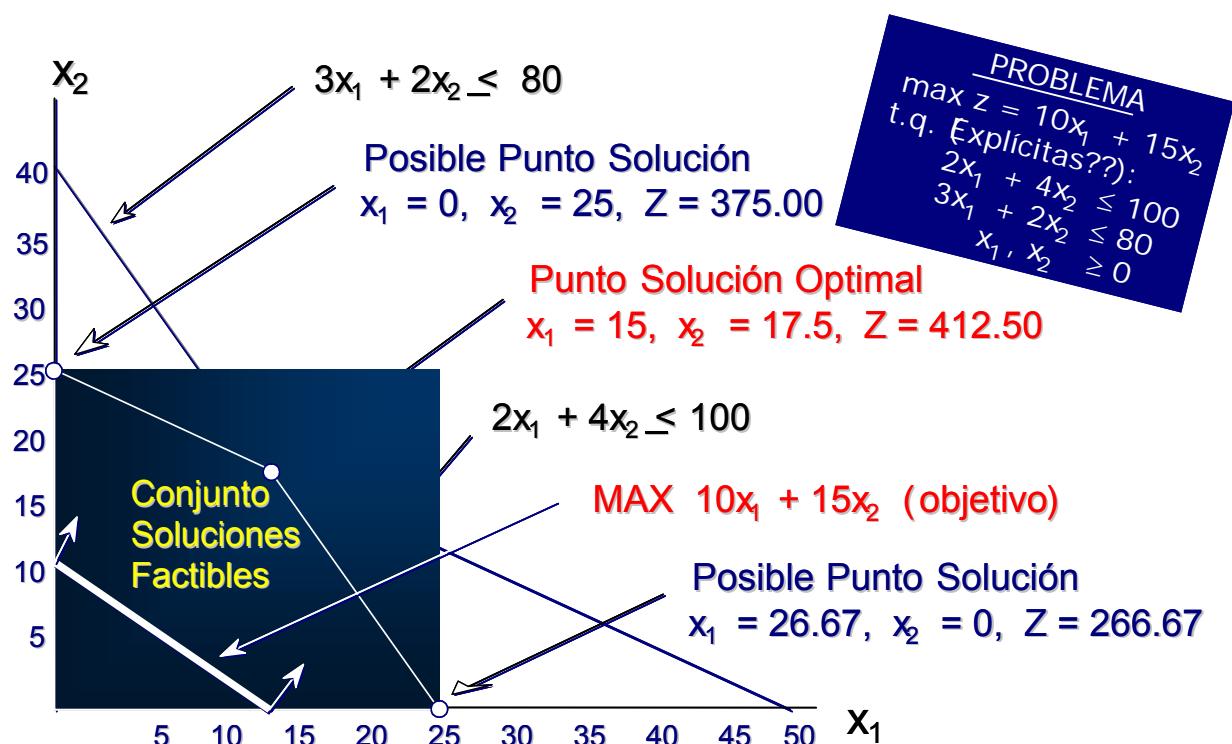
- **Solución Parcial:** Vector solución para el que aún no se han asignado todos sus componentes
- **Función de Poda:** Aquella función que nos permite identificar cuando una solución parcial no conduce a una solución del problema
- **Restricciones Explícitas:** Reglas que restringen el conjunto de valores que puede tomar cada una de las componentes  $x_i$  del vector solución (**determinan el espacio de soluciones**). Ejemplos comunes,

- $x_i \geq 0$                        $\Rightarrow$        $S_i = \{n^{\text{os}} \text{ reales no negativos}\}$
- $x_i = 0,1$                        $\Rightarrow$        $S_i = \{0,1\}$
- $l_i \leq x_i \leq u_i$                $\Rightarrow$        $S_i = \{a: l_i \leq a \leq u_i\}$

# Backtracking: Notación

- **Restricciones Implícitas:** Son aquellas que determinan cuando una solución parcial nos puede llevar a una solución —verifica la función criterio  $P(x_1, \dots, x_n)$ —
  - Describen la forma en que se relacionan las  $x_i$

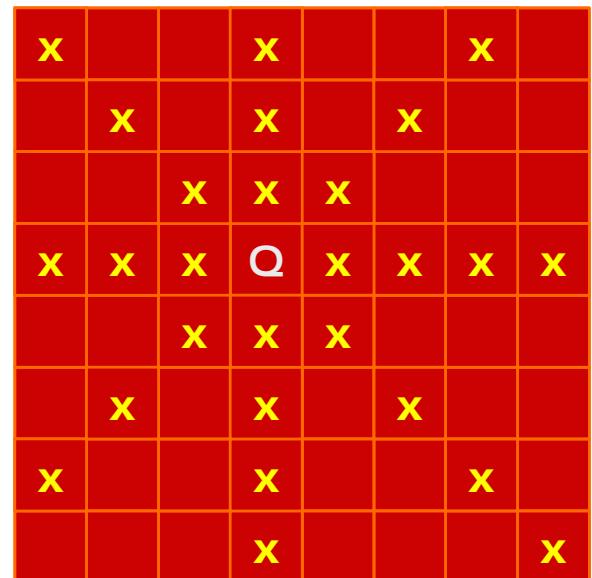
*Ejemplo:  
Interpretación de  
las restricciones*



# El problema de las ocho reinas

---

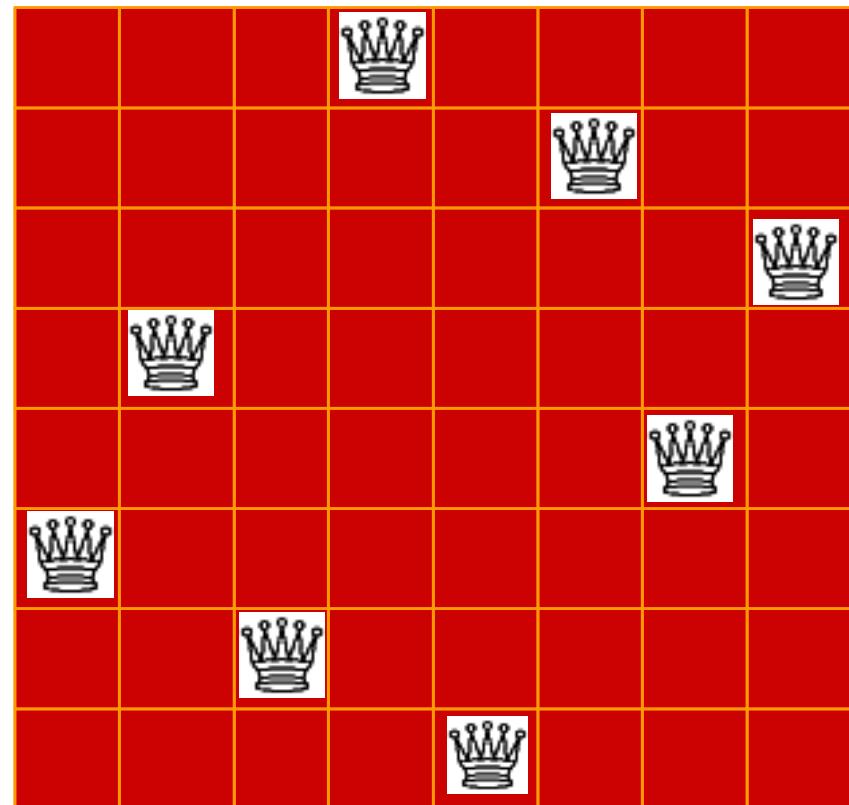
- Un clásico problema combinatorio es el de colocar ocho reinas en una tablero de ajedrez de modo que no haya dos que se ataquen, es decir, que estén en la misma fila, columna o diagonal
- Las filas y columnas se numeran del 1 al 8
- Las reinas se numeran del 1 al 8



Como cada reina debe estar en una fila diferente, sin perdida de generalidad podemos suponer que la reina  $i$  se coloca en la fila  $i$ . Todas las soluciones para este problema, pueden representarse como 8 tuplas  $(x_1, \dots, x_n)$  en las que  $x_i$  es la columna en la que se coloca la reina  $i$ .

# El problema de las ocho reinas

- Las **restricciones explícitas** son  $S_i = \{1,2,3,4,5,6,7,8\}$ ,  $1 \leq i \leq n$
- **Espacio solución** con  $8^8 = 2^{24} = 16M$  tuplas
- **Restricciones implícitas**: ningún par de  $x_i$  puedan ser iguales (todas las reinas deben estar en columnas diferentes). Ningún par de reinas pueden estar en la misma diagonal
- La primera de estas dos restricciones implica que todas las soluciones son **permutaciones** de  $(1,2,3,4,5,6,7,8)$
- Esto lleva a reducir el tamaño del **espacio solución** de  $8^8$  tuplas a  $8! = 40,320$



Una posible **solución del problema** es la ( 4,6,8,2,7,1,3,5 )

# Problema de la suma de subconjuntos

---

- Dados  $n+1$  números positivos:  
 $w_i$ ,  $1 \leq i \leq n$ , y uno mas  $M$ ,
- se trata de encontrar **todos** los subconjuntos de números  $w_i$  cuya suma valga  $M$
- Por ejemplo, si  $n = 4$ ,  $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$  y  $M = 31$ , entonces los subconjuntos buscados son  $(11, 13, 7)$  y  $(24, 7)$

# Problema de la suma de subconjuntos

---

- Para representar la solución podríamos notar el vector solución con los **índices de los correspondientes  $w_i$** ,
- Las dos soluciones se describen por los vectores  $(1,2,4)$  y  $(3,4)$
- Todas **las soluciones son  $k$ -tuplas**  $(x_1, x_2, \dots, x_k)$ ,  $1 \leq k \leq n$ , y soluciones diferentes pueden tener tamaños de tupla diferentes
- **Restricciones explícitas:**  $x_i \in \{j: j \text{ es entero y } 1 \leq j \leq n\}$
- **Restricciones implícitas:** que no haya dos iguales y que la suma de los correspondientes  $w_i$  sea  $M$
- Además, como por ejemplo  $(1,2,4)$  y  $(1,4,2)$  representan el mismo subconjunto, **otra restricción implícita** que hay que imponer es que  $x_i < x_{i+1}$ , para  $1 \leq i < n$

# Problema de la suma de subconjuntos

---

- Puede haber **diferentes formas de formular** un problema de modo que todas las soluciones sean tuplas que satisfacen algunas restricciones
- Otra formulación del problema:
  - Cada subconjunto solución se representa por una  $n$ -tupla  $(x_1, \dots, x_n)$  tal que  $x_i \in \{0, 1\}$ ,  $1 \leq i < n$ , con  $x_i = 0$  si  $w_i$  no se elige y  $x_i = 1$  si  $w_i$  se elige
  - Las soluciones del anterior caso son  $(1, 1, 0, 1)$  y  $(0, 0, 1, 1)$
  - Esta formulación expresa todas las soluciones usando un tamaño de tupla fijo
- Se puede comprobar que para estas dos formulaciones, el espacio solución consiste en ambos casos de  $2^4$  tuplas distintas

# Índice

---

## I. LA TÉCNICA BACKTRACKING

### 1. Introducción: El método General

- Resolución de problemas cuando la solución se puede expresar como una n-tupla
- Ejemplo: El problema de las 8 reinas
- Ejemplo: La suma de subconjuntos

### 2. Espacio de soluciones: Organización del Árbol

- Ejemplo: Espacio solución para el problema de las N-reinas (N=4)
- Ejemplo: Espacio solución para la suma de subconjuntos
- Terminología utilizada para la organización en árbol
- Ejemplo: Backtracking en el problema de las 4 reinas

### 3. Procedimiento *Backtracking*

- Procedimiento Iterativo
- Procedimiento Recursivo

# Espacios de soluciones

---

- Los algoritmos backtracking determinan las soluciones del problema buscando en el **espacio de soluciones** del caso considerado sistemáticamente
- Esta búsqueda se puede representar usando una **organización en árbol** para el espacio solución.
- Para un espacio solución dado, pueden caber **muchas organizaciones** en árbol.
- Los siguientes ejemplos examinan algunas de las formas posibles para estas organizaciones.

# Ejemplo de espacio de soluciones

---

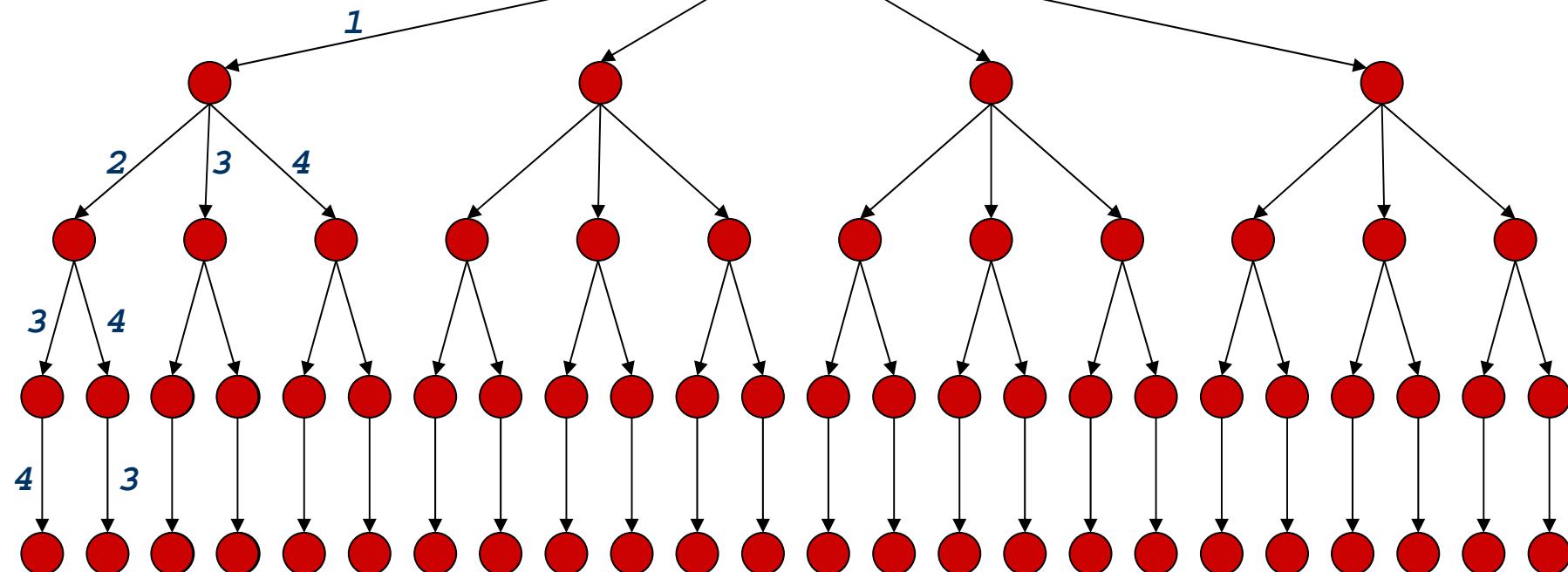
- La generalización del problema de las 8 reinas es el de las  $N$  reinas: Colocar  $N$  reinas en un tablero  $N \times N$  de modo que no haya dos que se ataquen
- Ahora el espacio de soluciones consiste en las  $N!$  **permutaciones** de la  $N$ -tupla  $(1, 2, \dots, N)$
- La generalización nos sirve a efectos didácticos para poder hablar del problema de las 4 reinas
- La siguiente figura muestra una posible organización de las soluciones del problema de las 4 reinas en forma de árbol
- A un árbol como ese se le llama **Árbol de** (búsqueda de soluciones) **Permutación**

# 4-Reinas: Arbol de permutación

Las aristas se etiquetan con los posibles valores de las  $x_i$

Nivel 1

Las aristas desde los nodos del nivel  $i$  al  $i+1$  están etiquetadas con los valores de  $x_i$



El subárbol de la izquierda contiene todas las soluciones con  $x_1=1$  y  $x_2=2$ ,  $x_2=3$ ,  $x_2=4$ , ... El espacio de soluciones está definido por todos los caminos desde el nodo raíz a un nodo hoja. Hay  $4! = 24$  nodos hoja en la figura

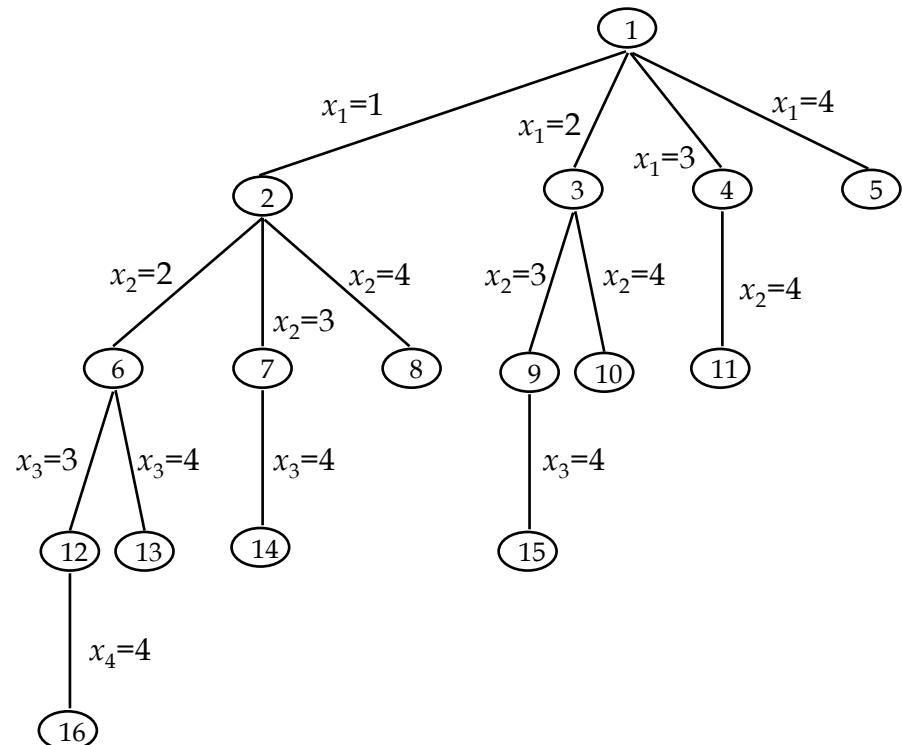
# Ejemplos de Árboles en la Suma de Subconjuntos

---

- Vimos dos posibles formulaciones del espacio solución del problema de la suma de subconjuntos.
  - La primera corresponde a la formulación por el tamaño de la tupla
  - La segunda considera un tamaño de tupla fijo
- Con ambas formulaciones, tanto en este problema como en cualquier otro, el número de soluciones tiene que ser el mismo

# Suma de subconjuntos: árbol 1

- Las aristas se etiquetan de modo que una desde el nivel de nodos  $i$  hasta el  $i+1$  representa un valor para  $x_i$
- Así, el subárbol de la izquierda define todos los subconjuntos conteniendo  $w_1$ , el siguiente todos los que contienen  $w_2$  pero no  $w_1$ , etc.
- En cada nodo, el espacio solución se partitiona en espacios subsolución
- Las posibles soluciones son 16 tuplas, que corresponden al **camino desde la raíz a cada nodo**, (), (1), (12), (123), (1234), (124), (134), (14), (2), (23), etc.



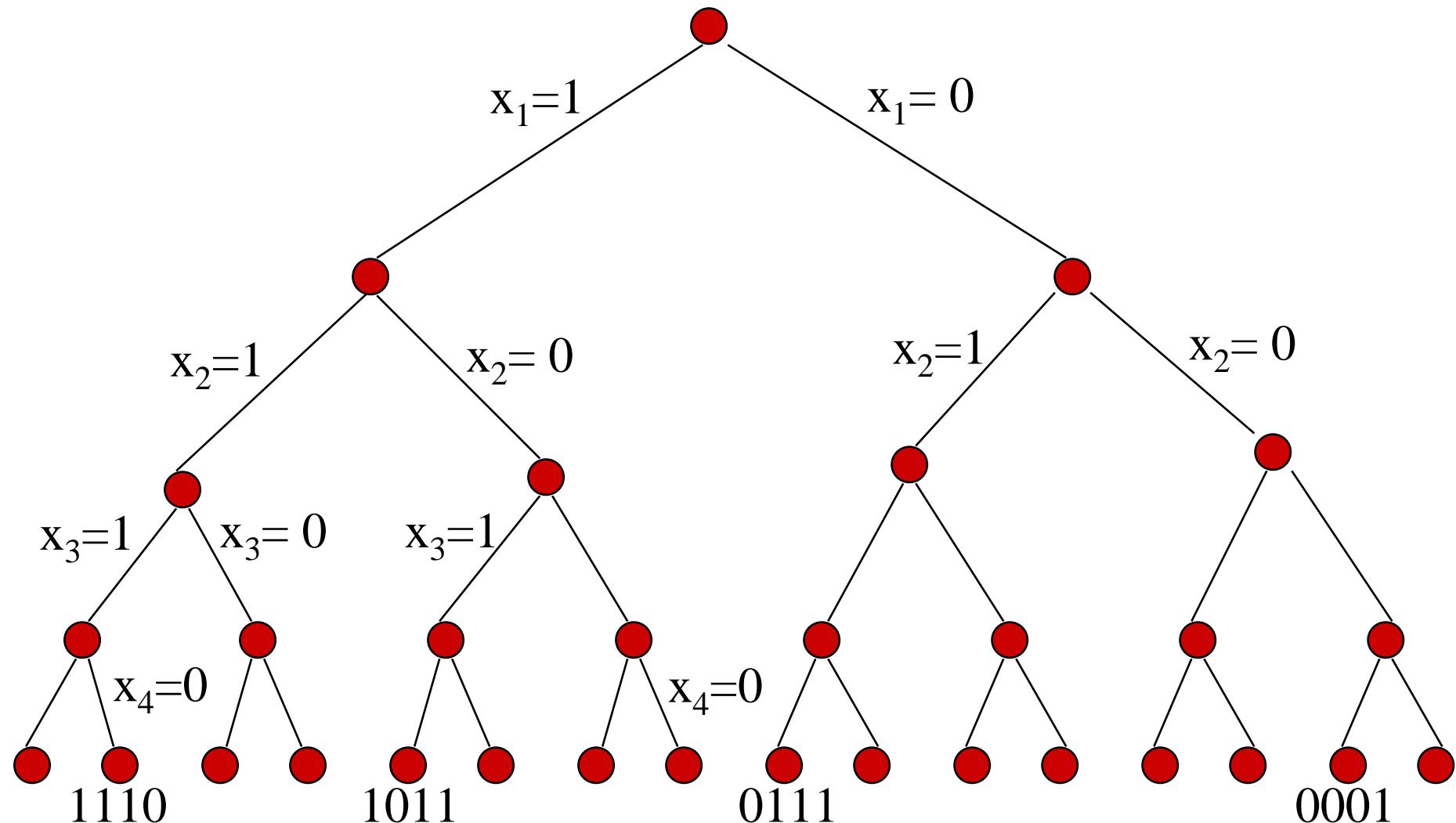
# Suma de subconjuntos: árbol 2

---

- Una arista del nivel  $i$  al  $i+1$  se etiqueta con el valor de  $x_i$  ( $0$  o  $1$ )
- Todos los **caminos desde la raíz a las hojas** definen el espacio solución
- El subárbol de la izquierda define todos los subconjuntos conteniendo  $w_1$ , mientras que el de la derecha define todos los subconjuntos que no contienen  $w_1$ , etc.
- Consideramos el caso de  $n = 4$
- Hay  $2^4$  nodos hoja, que representan **16 posibles tuplas**

# Suma de subconjuntos: árbol 2

---



# Terminología

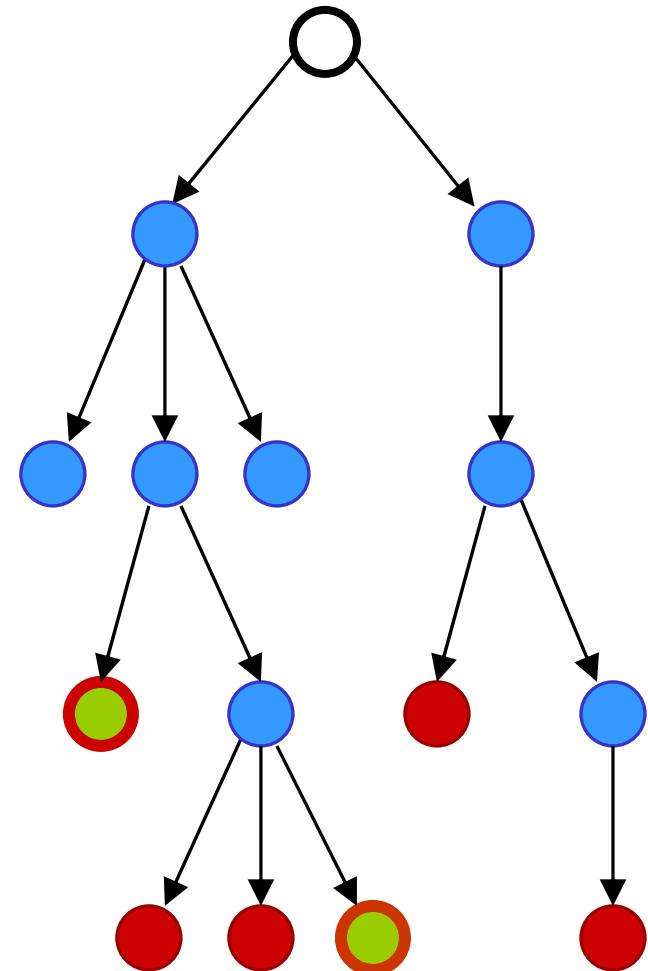
---

- **Estado del problema:** Cada uno de los **nodos del árbol**
- **Estado solución:** Son aquellos **estados (nodos)  $S$**  del problema (árbol) **para los que el camino desde la raíz a  $S$  representa una  $n$ -tupla** en el espacio solución
  - En el primero de los árboles anteriores, todos los nodos son estados solución, mientras que en el segundo sólo los nodos hoja son estados solución
- **Estados respuesta:** Son aquellos estados solución  $S$  que representan una  $n$ -tupla del conjunto de soluciones del problema (aquellas que satisfacen las restricciones implícitas)

# Generación de estados de un problema

---

- Cuando se ha concebido un árbol de estados para algún problema, podemos resolver este problema generando sistemáticamente sus **estados**, determinando cuales de estos son **estados solución**, y finalmente determinando que estados solución son **estados respuesta**



# *Terminología. Generación de Estados*

---

- **Nodo vivo:** Estado del problema que ya ha sido generado, pero del que **aún no se han generado todos sus hijos**
- **Nodo muerto:** Estado del problema que ya ha sido generado, y o bien **se ha podado** o bien **se han generado todos los descendientes**
- **E-nodo (nodo de expansión):** Nodo vivo del que actualmente se están **generando los descendientes**

# Generación de estados de un problema

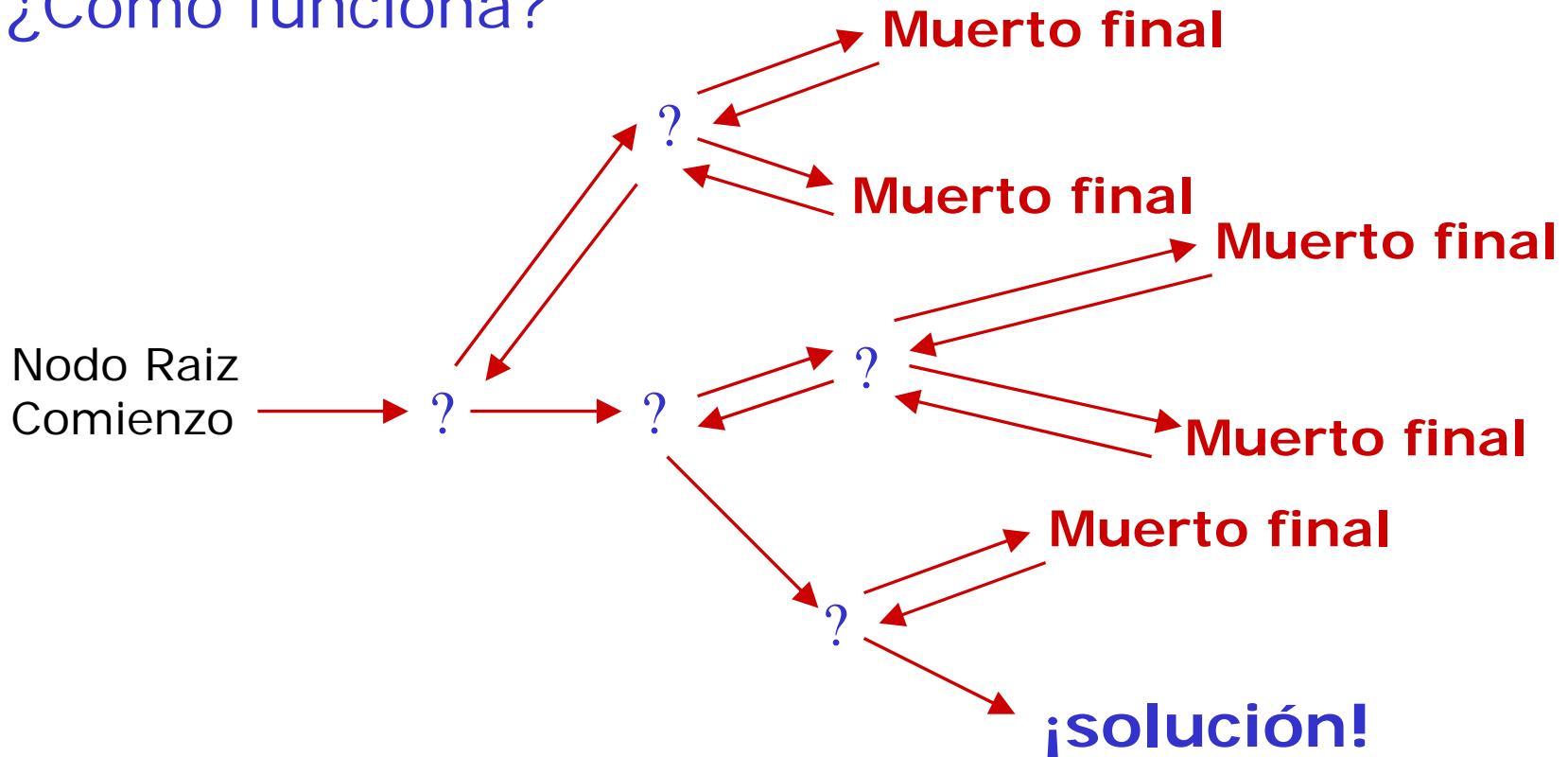
---

- Hay dos formas diferentes de generar los estados del problema
  - Las dos comienzan con el nodo raíz y generan otros nodos
  - En ambos métodos de generar estados del problema tendremos una **lista de nodos vivos**
- En el **primer método**, tan pronto como un nuevo hijo  $C$  del  $E$ -nodo en curso  $R$  ha sido generado, este **hijo se convierte en un nuevo  $E$ -nodo**
  - $R$  se convertirá de nuevo en  $E$ -nodo cuando el subárbol  $C$  haya sido explorado completamente
  - Esto corresponde a una **generación primero en profundidad** de los estados del problema
- Adicionalmente se usan **funciones de acotación** para matar nodos vivos sin tener que generar todos sus nodos hijos

# Generación de estados de un problema

- A esta forma de generación (exploración) se le llama ***Backtracking***

¿Cómo funciona?



# Generación de estados de un problema

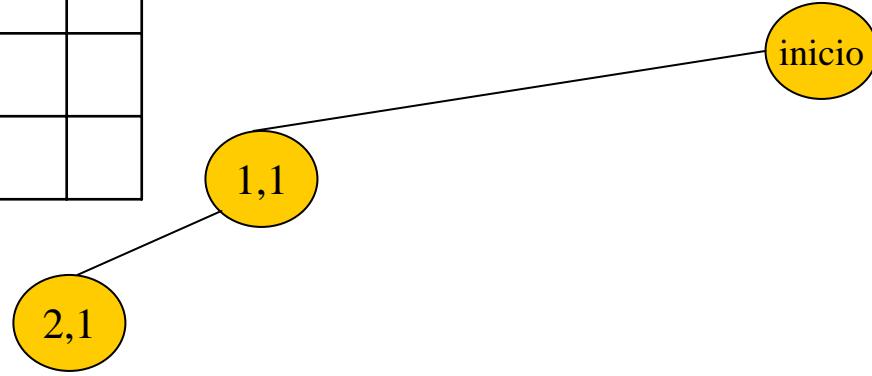
---

- En el **segundo método**, el *E*-nodo permanece como ***E*-nodo hasta que se hace nodo muerto**
- También se usan **funciones de acotación** para detener la exploración en un subárbol
- El método se adapta muy bien a la resolución de problemas de optimización combinatoria (espacio de soluciones discreto): Problema de la Mochila, Soluciones enteras, etc
- En este método, la construcción de las **funciones de acotación es (casi) más importante** que los mecanismos de exploración en si mismos
- A esta segunda forma de generación (exploración) se le llama ***Branch and Bound***

# Ejemplo BK... para $n = 4$

---

| $R$ |  |  |  |  |
|-----|--|--|--|--|
| $x$ |  |  |  |  |
|     |  |  |  |  |
|     |  |  |  |  |
|     |  |  |  |  |



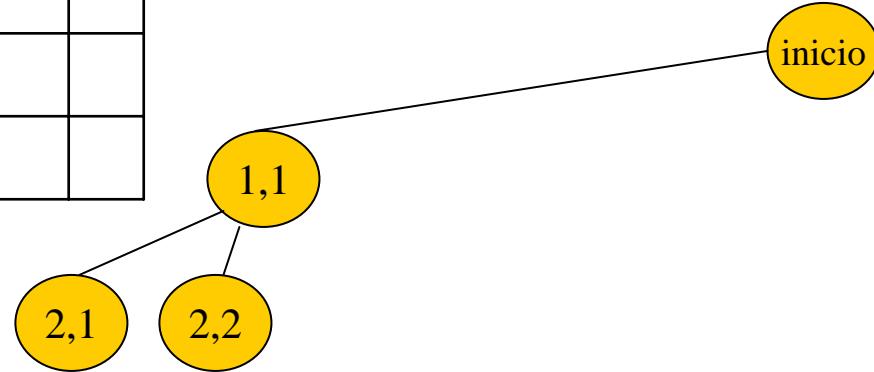
La estrategia ASEGURA  
no ocupar el mismo renglón

NO se cumple el criterio  
(misma columna)

# Ejemplo... para $n = 4$

---

|          |          |  |  |
|----------|----------|--|--|
| <b>R</b> |          |  |  |
| <b>x</b> | <b>x</b> |  |  |
|          |          |  |  |
|          |          |  |  |

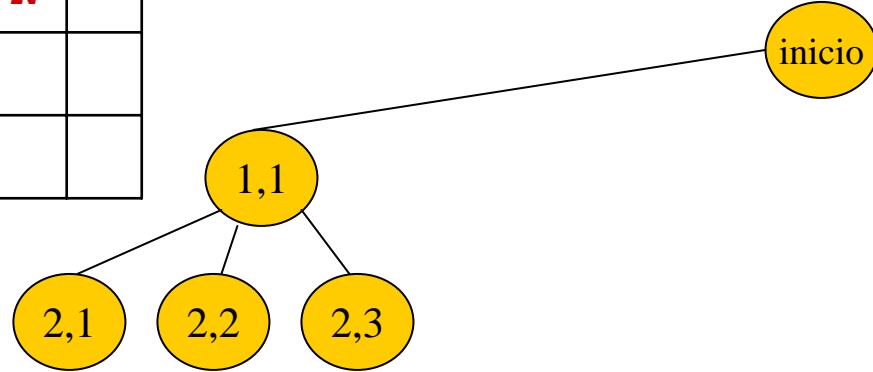


*NO se cumple el criterio  
(misma diagonal)*

# Ejemplo... para $n = 4$

---

|          |          |          |  |
|----------|----------|----------|--|
| <b>R</b> |          |          |  |
| <b>x</b> | <b>x</b> | <b>R</b> |  |
|          |          |          |  |
|          |          |          |  |

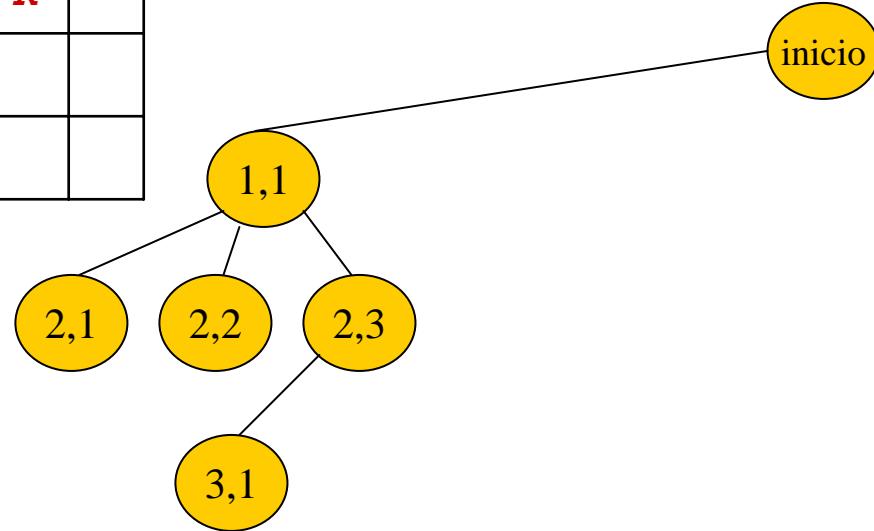


OK... adelante en la  
búsqueda !

# Ejemplo... para $n = 4$

---

|          |          |          |  |
|----------|----------|----------|--|
| <b>R</b> |          |          |  |
| <b>x</b> | <b>x</b> | <b>R</b> |  |
| <b>x</b> |          |          |  |
|          |          |          |  |

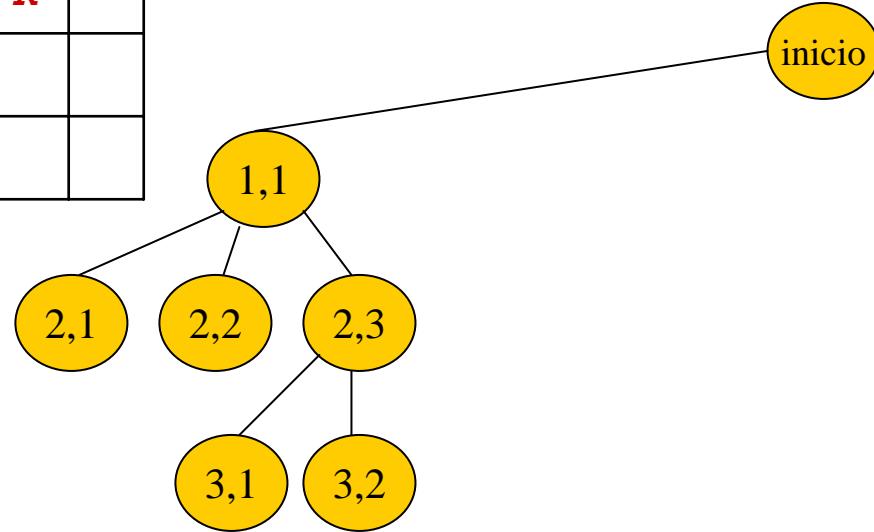


**NO se cumple el criterio  
(misma columna)**

# Ejemplo... para $n = 4$

---

|          |          |          |  |
|----------|----------|----------|--|
| <b>R</b> |          |          |  |
| <b>x</b> | <b>x</b> | <b>R</b> |  |
| <b>x</b> | <b>x</b> |          |  |
|          |          |          |  |

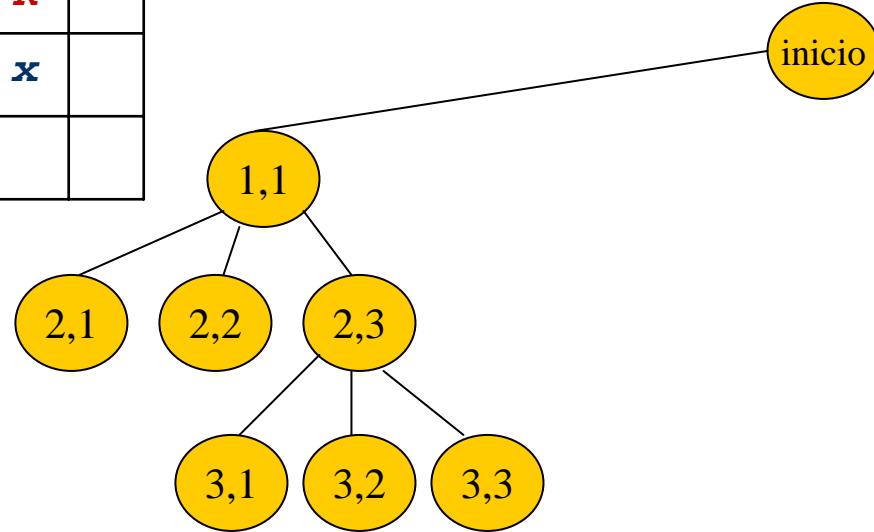


*NO se cumple el criterio  
(misma diagonal que 2,3)*

# Ejemplo... para $n = 4$

---

|          |          |          |  |
|----------|----------|----------|--|
| <b>R</b> |          |          |  |
| <b>x</b> | <b>x</b> | <b>R</b> |  |
| <b>x</b> | <b>x</b> | <b>x</b> |  |
|          |          |          |  |

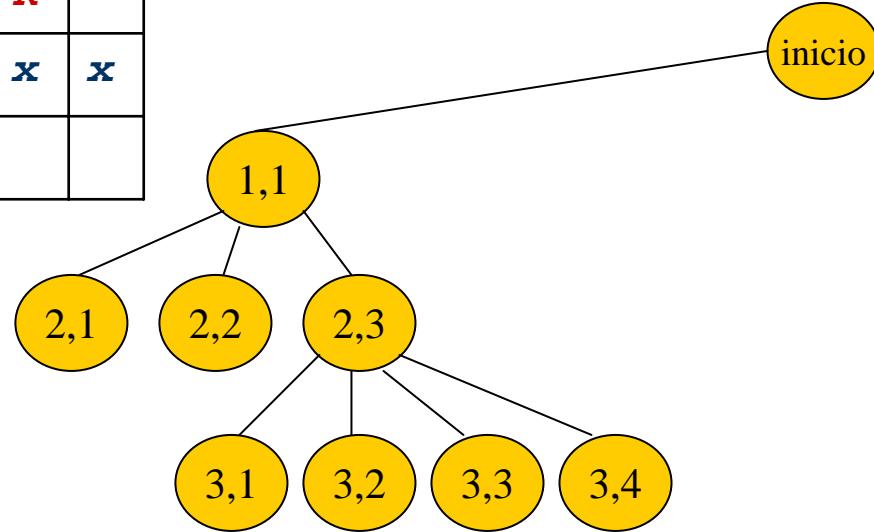


**NO se cumple el criterio  
(misma diagonal que 1,1)**

# Ejemplo... para $n = 4$

---

|          |          |          |          |
|----------|----------|----------|----------|
| <b>R</b> |          |          |          |
| <b>x</b> | <b>x</b> | <b>R</b> |          |
| <b>x</b> | <b>x</b> | <b>x</b> | <b>x</b> |
|          |          |          |          |

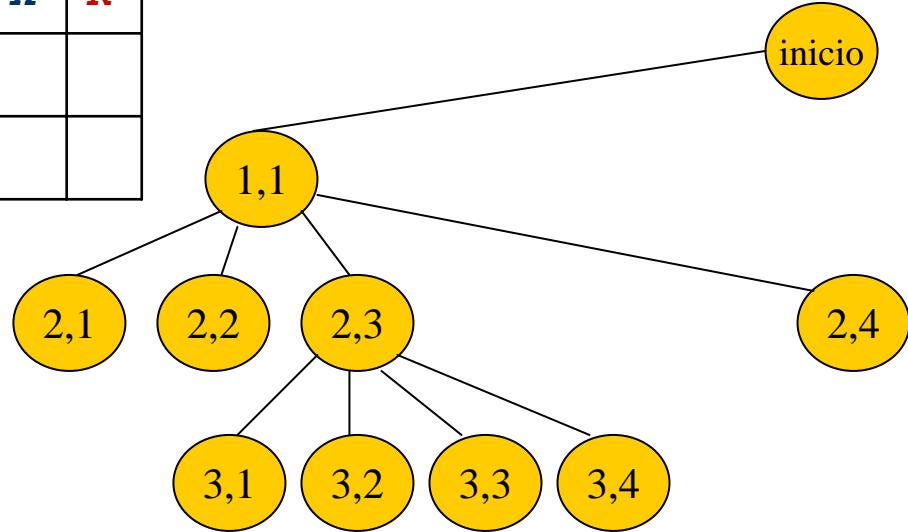


**NO se cumple el criterio  
(misma diagonal que 2,3)**

# Ejemplo... para $n = 4$

---

|          |          |          |          |
|----------|----------|----------|----------|
| <b>R</b> |          |          |          |
| <b>x</b> | <b>x</b> | <b>x</b> | <b>R</b> |
|          |          |          |          |
|          |          |          |          |

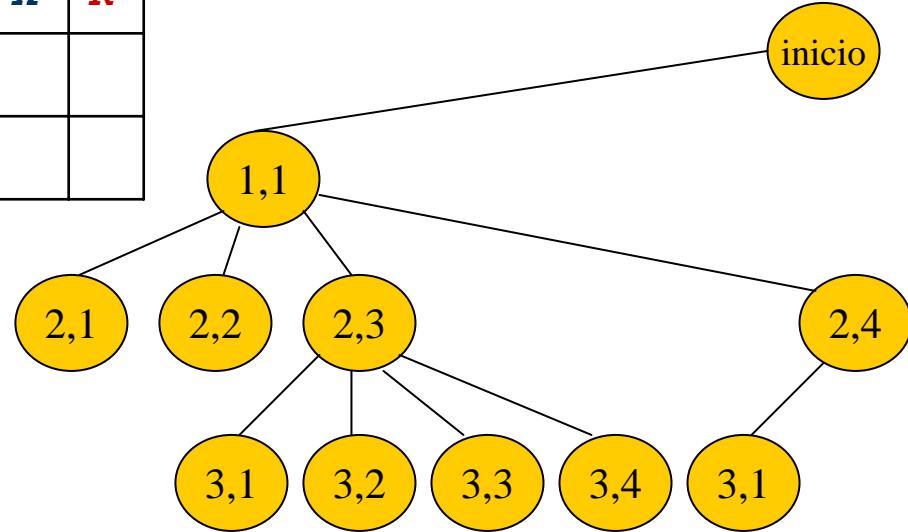


OK... adelante con la  
búsqueda!

# Ejemplo... para $n = 4$

---

|          |          |          |          |
|----------|----------|----------|----------|
| <b>R</b> |          |          |          |
| <b>x</b> | <b>x</b> | <b>x</b> | <b>R</b> |
| <b>x</b> |          |          |          |
|          |          |          |          |

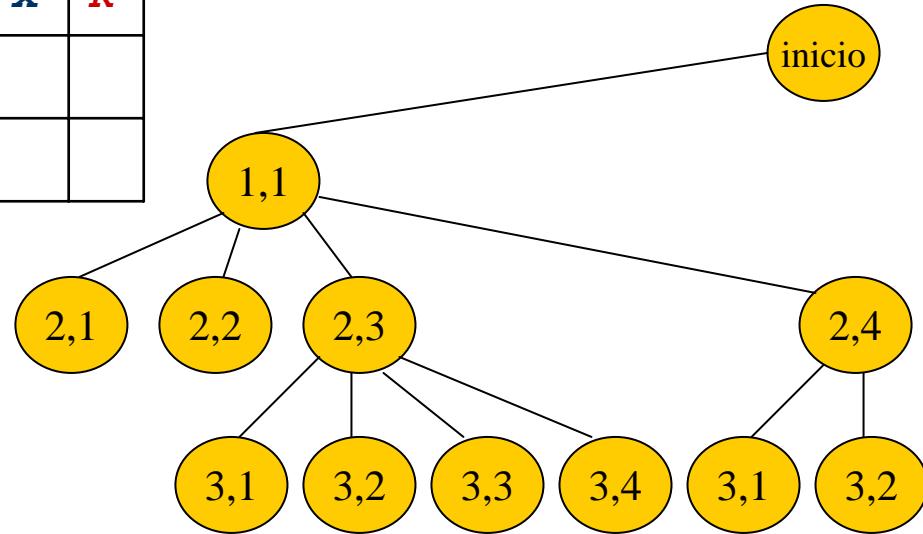


**NO se cumple criterio  
(misma columna)**

# Ejemplo... para $n = 4$

---

|          |          |          |          |
|----------|----------|----------|----------|
| <b>R</b> |          |          |          |
| <b>x</b> | <b>x</b> | <b>x</b> | <b>R</b> |
| <b>x</b> | <b>R</b> |          |          |
|          |          |          |          |

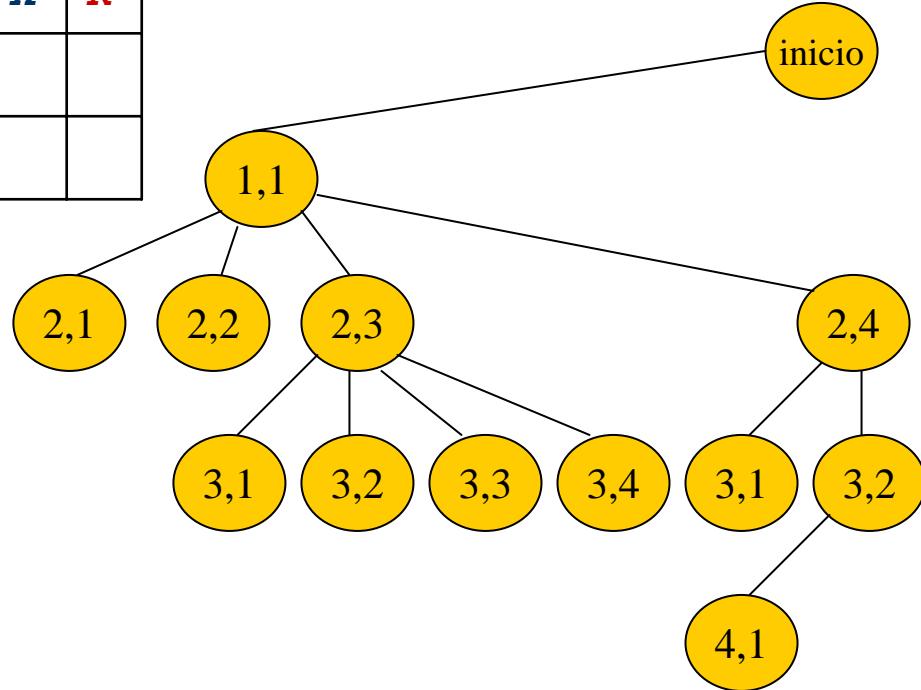


OK... adelante con la  
búsqueda!

# Ejemplo... para $n = 4$

---

|          |          |          |          |
|----------|----------|----------|----------|
| <b>R</b> |          |          |          |
| <b>x</b> | <b>x</b> | <b>x</b> | <b>R</b> |
| <b>x</b> | <b>R</b> |          |          |
| <b>x</b> |          |          |          |

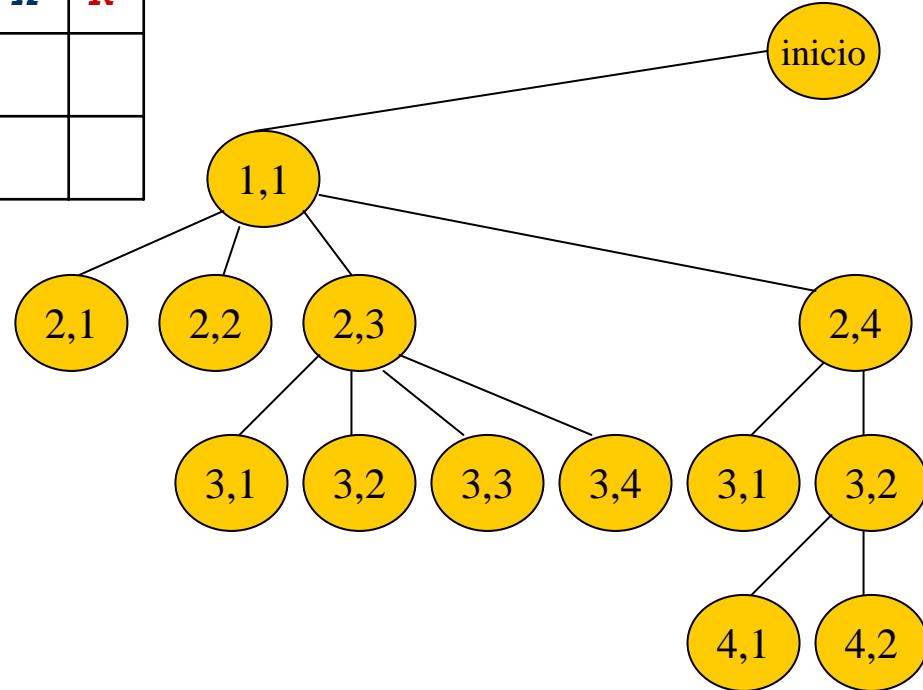


**NO se cumple criterio  
(misma columna)**

# Ejemplo... para $n = 4$

---

|          |          |          |          |
|----------|----------|----------|----------|
| <b>R</b> |          |          |          |
| <b>x</b> | <b>x</b> | <b>x</b> | <b>R</b> |
| <b>x</b> | <b>R</b> |          |          |
| <b>x</b> | <b>x</b> |          |          |

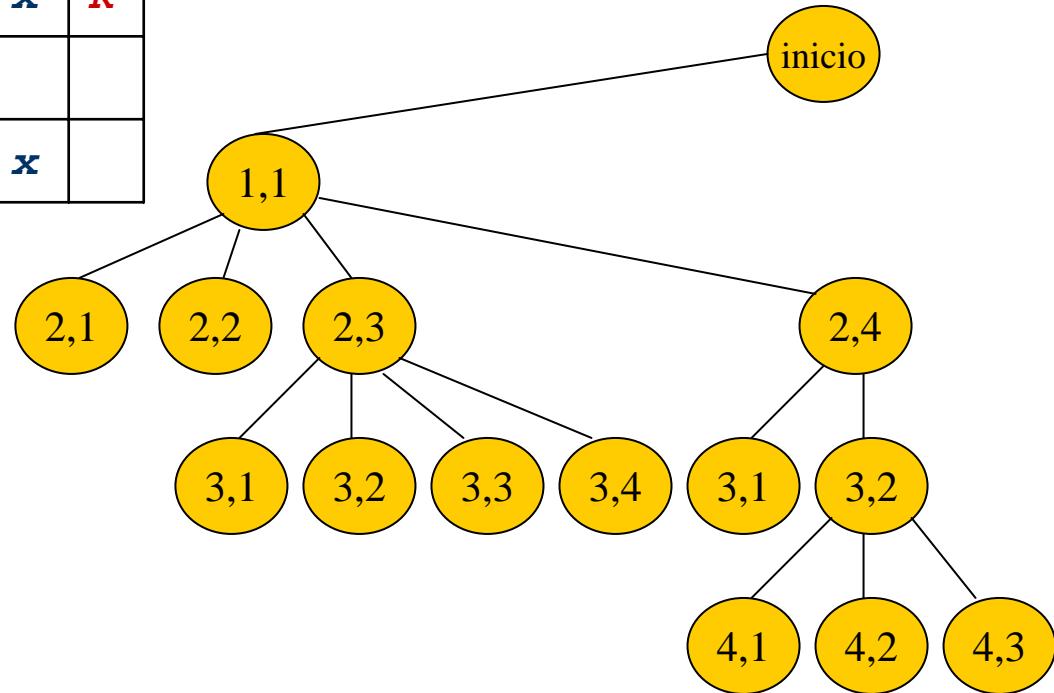


**NO se cumple criterio  
(misma columna)**

# Ejemplo... para $n = 4$

---

|          |          |          |          |
|----------|----------|----------|----------|
| <b>R</b> |          |          |          |
| <b>x</b> | <b>x</b> | <b>x</b> | <b>R</b> |
| <b>x</b> | <b>R</b> |          |          |
| <b>x</b> | <b>x</b> | <b>x</b> |          |

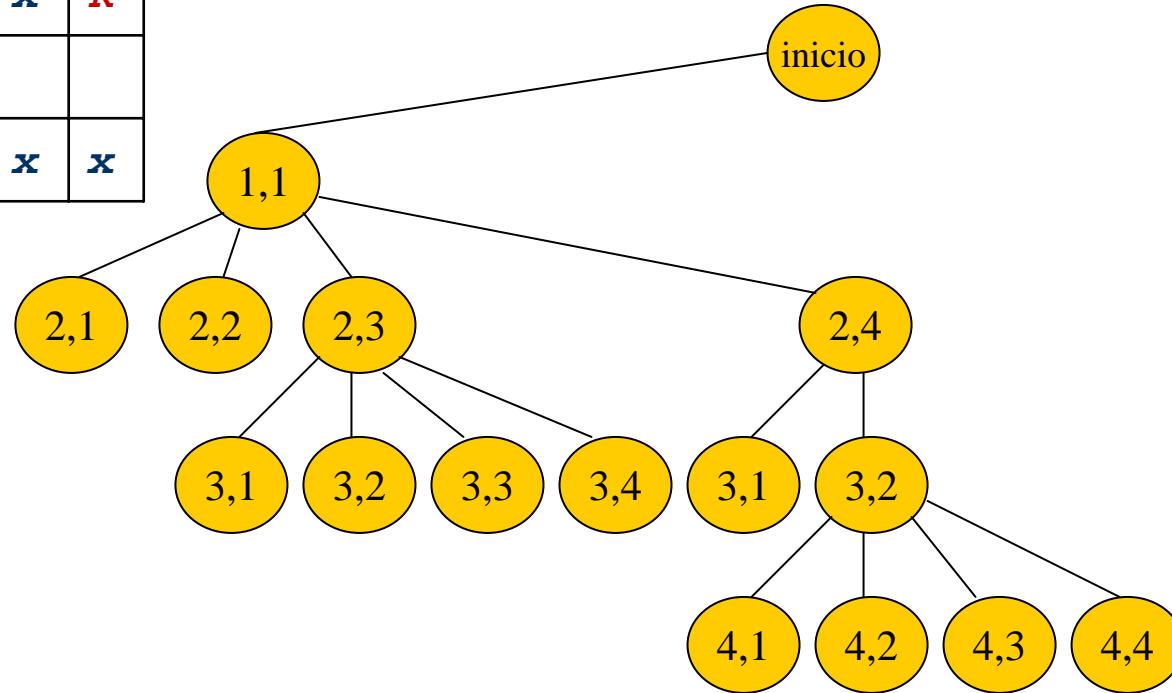


**NO se cumple criterio  
(misma diagonal 3,2)**

# Ejemplo... para $n = 4$

---

|          |          |          |          |
|----------|----------|----------|----------|
| <b>R</b> |          |          |          |
| <b>x</b> | <b>x</b> | <b>x</b> | <b>R</b> |
| <b>x</b> | <b>R</b> |          |          |
| <b>x</b> | <b>x</b> | <b>x</b> | <b>x</b> |

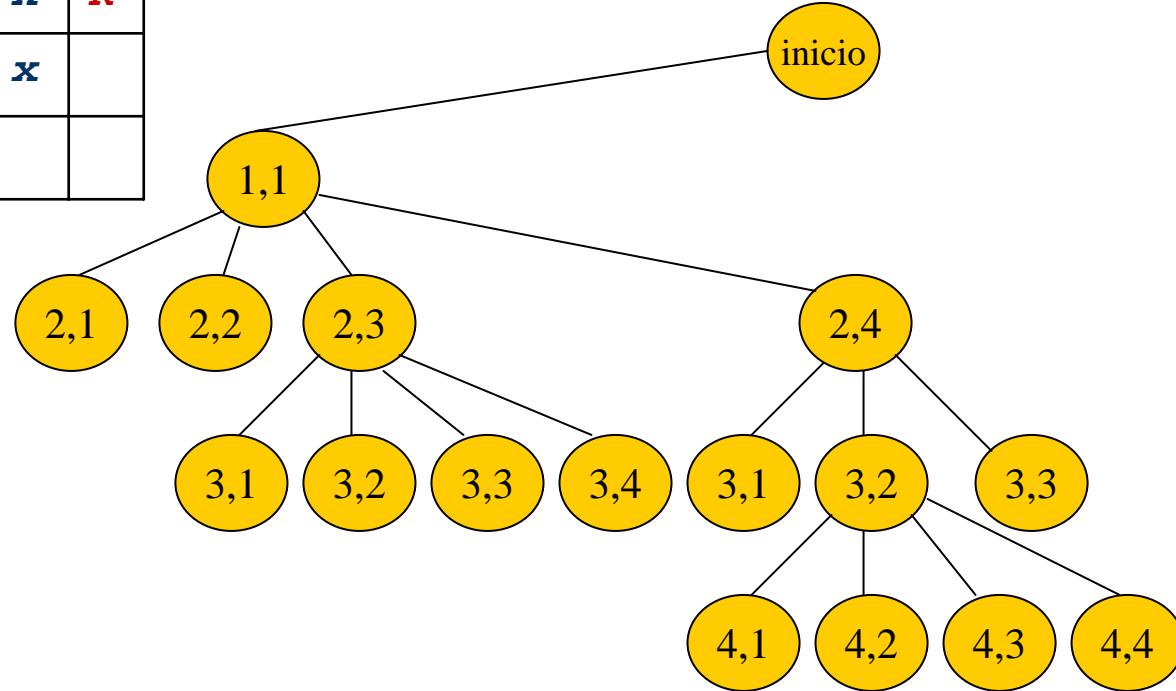


**NO se cumple criterio  
(misma columna)**

# Ejemplo... para $n = 4$

---

|          |          |          |          |
|----------|----------|----------|----------|
| <b>R</b> |          |          |          |
| <b>x</b> | <b>x</b> | <b>x</b> | <b>R</b> |
| <b>x</b> | <b>x</b> | <b>x</b> |          |
|          |          |          |          |

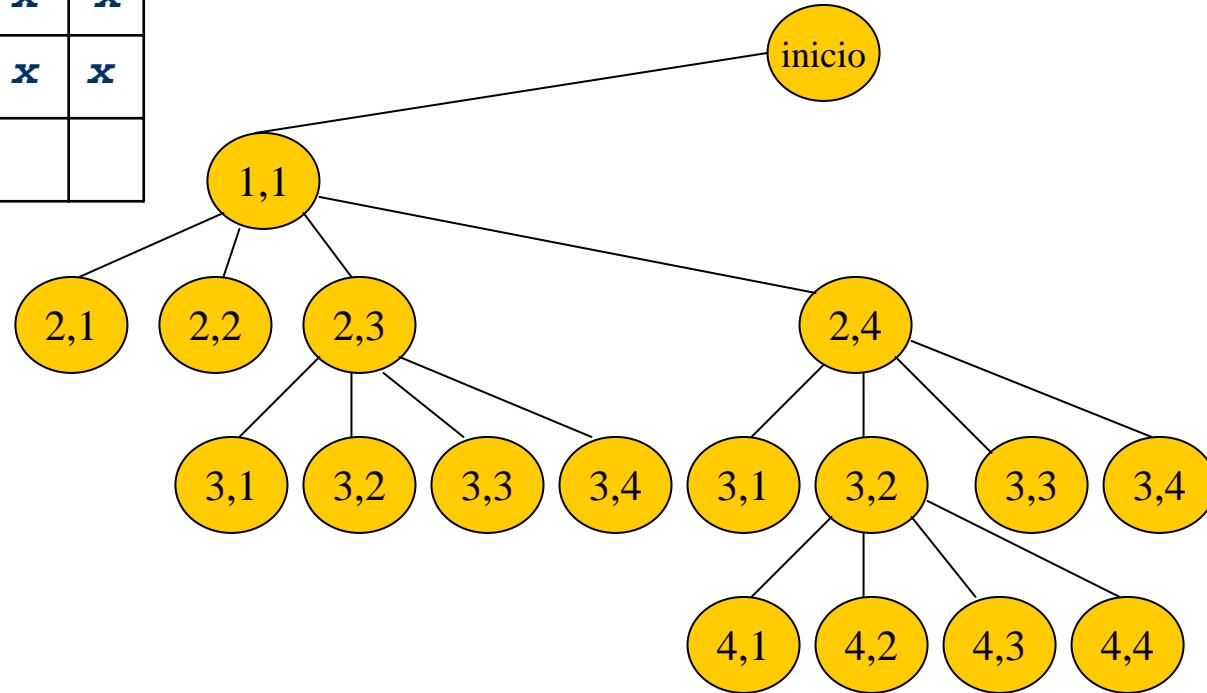


**NO se cumple criterio  
(misma diagonal)**

# Ejemplo... para $n = 4$

---

| <b>R</b> |          |          |          |  |
|----------|----------|----------|----------|--|
| <b>x</b> | <b>x</b> | <b>x</b> | <b>x</b> |  |
| <b>x</b> | <b>x</b> | <b>x</b> | <b>x</b> |  |
|          |          |          |          |  |

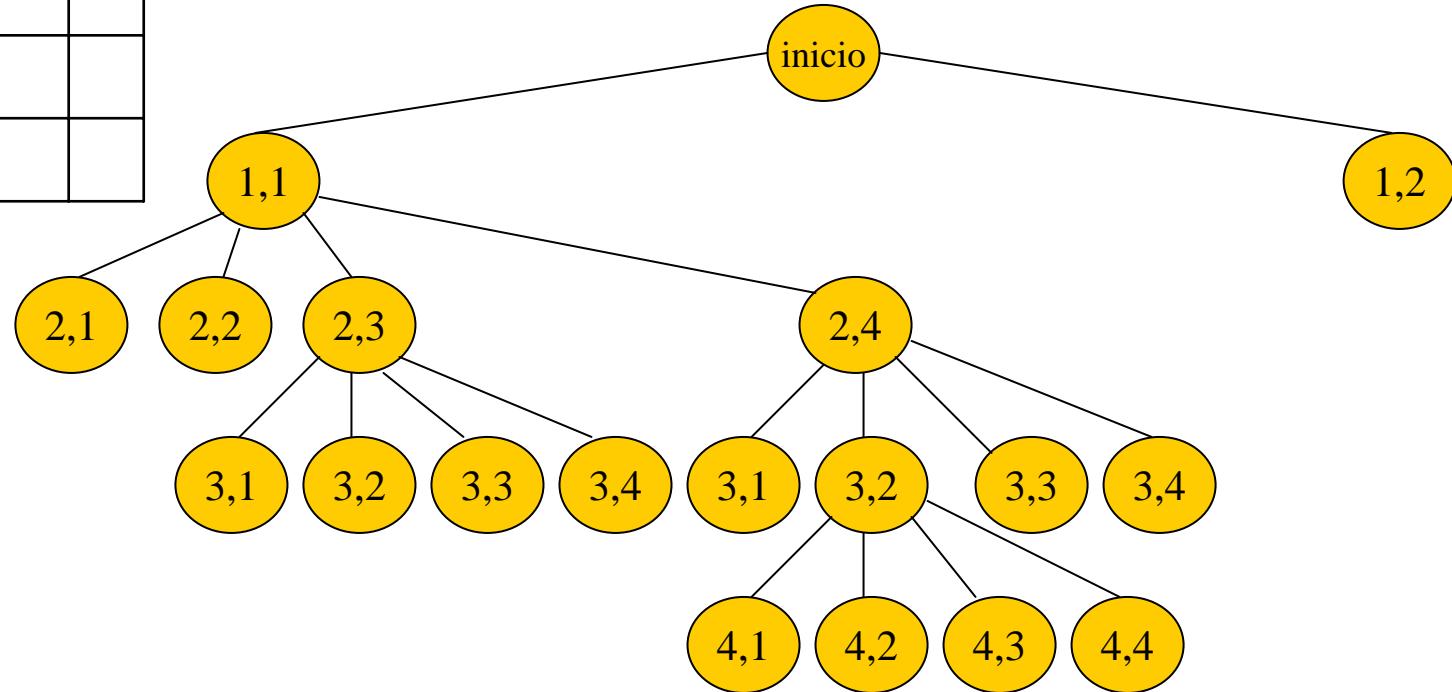


**NO se cumple criterio  
(misma columna)**

# Ejemplo... para $n = 4$

---

| <b>x</b> | <b>R</b> |  |  |
|----------|----------|--|--|
|          |          |  |  |
|          |          |  |  |
|          |          |  |  |

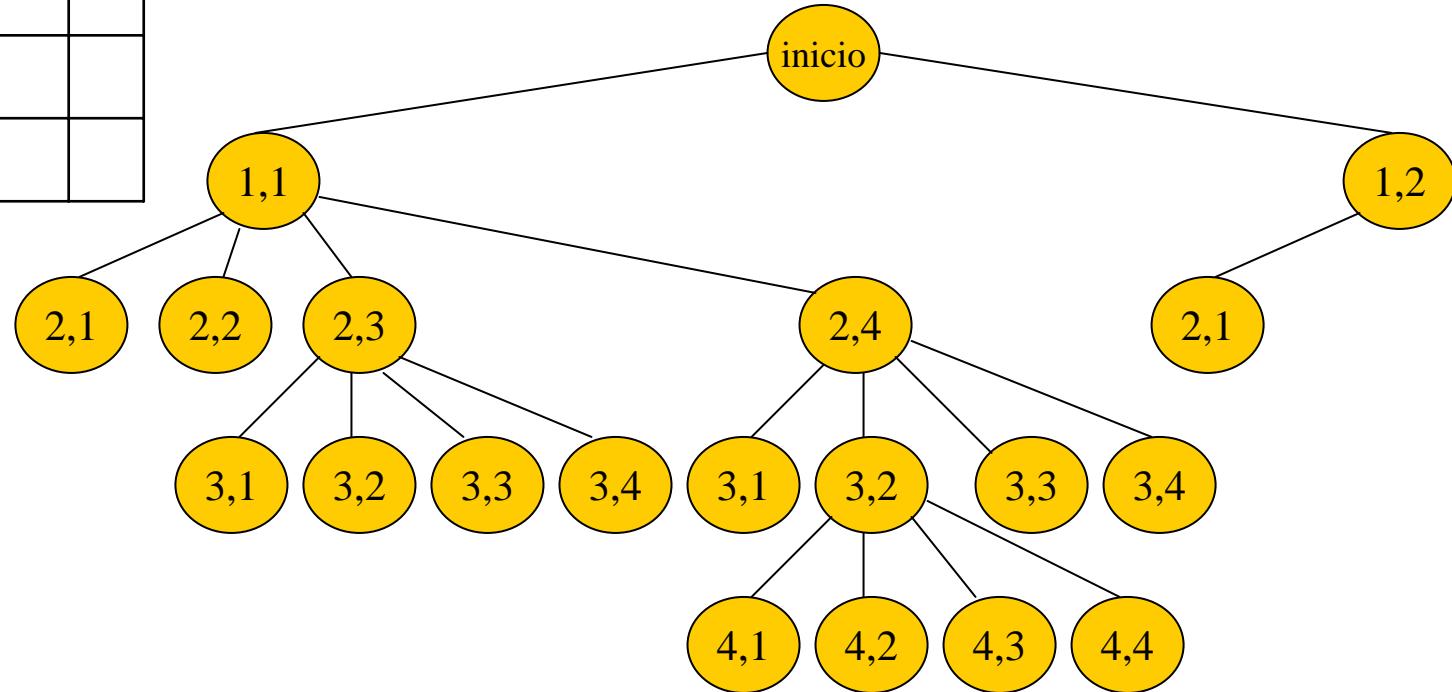


OK... adelante con la  
búsqueda!

# Ejemplo... para $n = 4$

---

| $x$ | $R$ |  |  |
|-----|-----|--|--|
| $x$ |     |  |  |
|     |     |  |  |
|     |     |  |  |

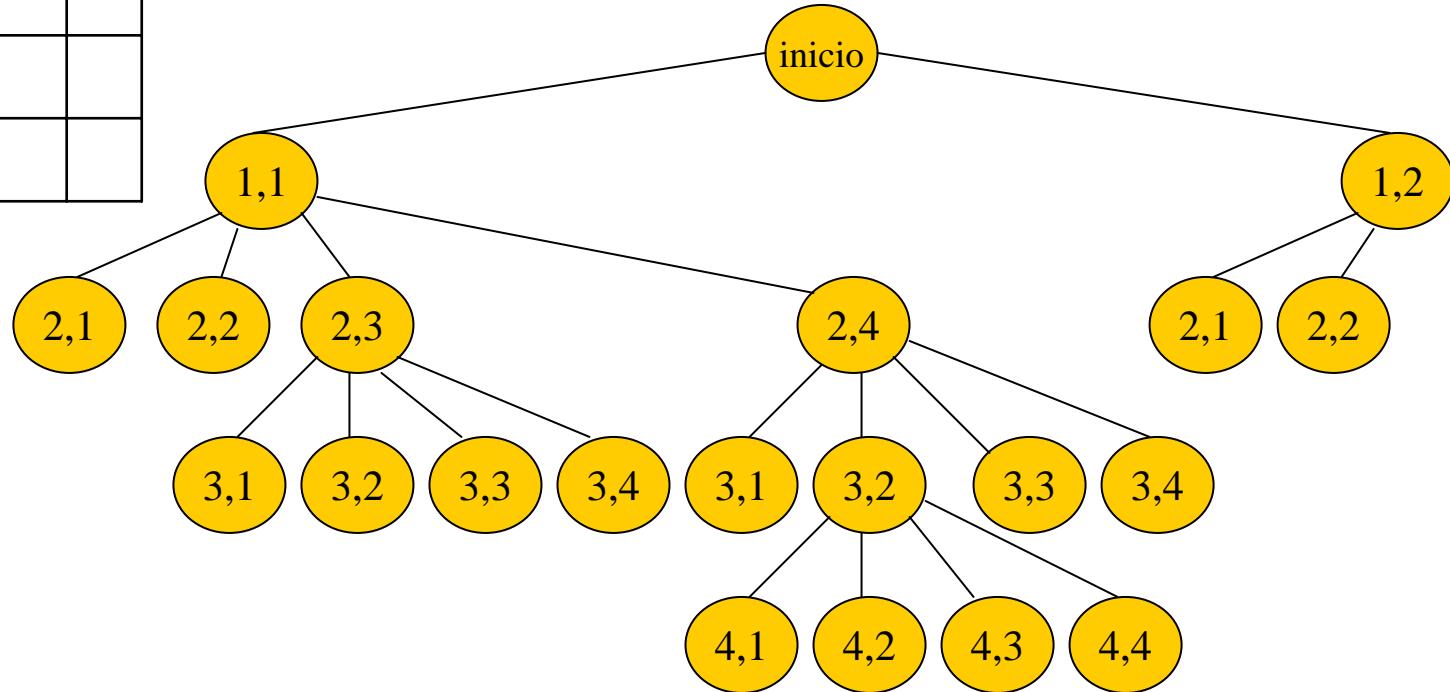


**NO cumple criterio  
(misma diagonal)**

# Ejemplo... para $n = 4$

---

| $x$ | $R$ |  |  |
|-----|-----|--|--|
| $x$ | $x$ |  |  |
|     |     |  |  |
|     |     |  |  |

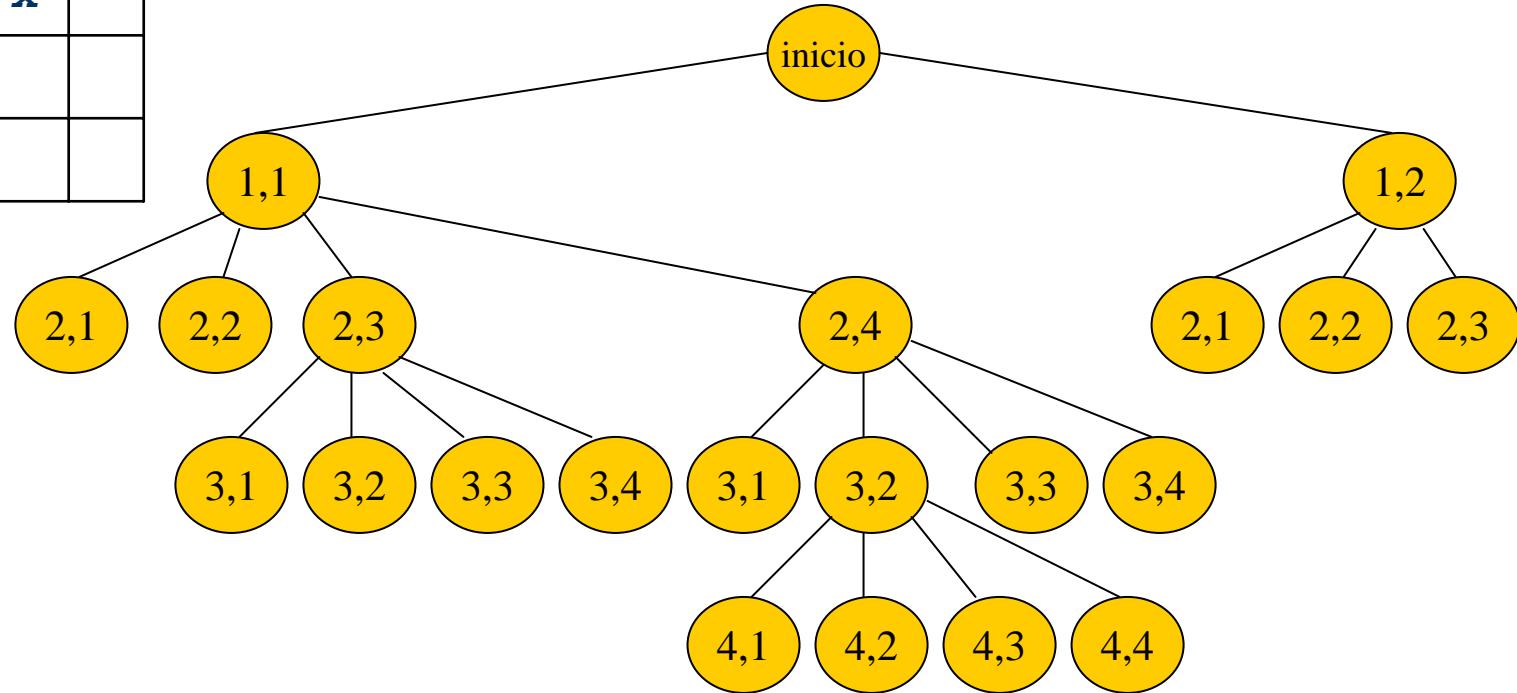


**NO cumple criterio  
(misma columna)**

# Ejemplo... para $n = 4$

---

| $x$ | $R$ |     |  |
|-----|-----|-----|--|
| $x$ | $x$ | $x$ |  |
|     |     |     |  |
|     |     |     |  |

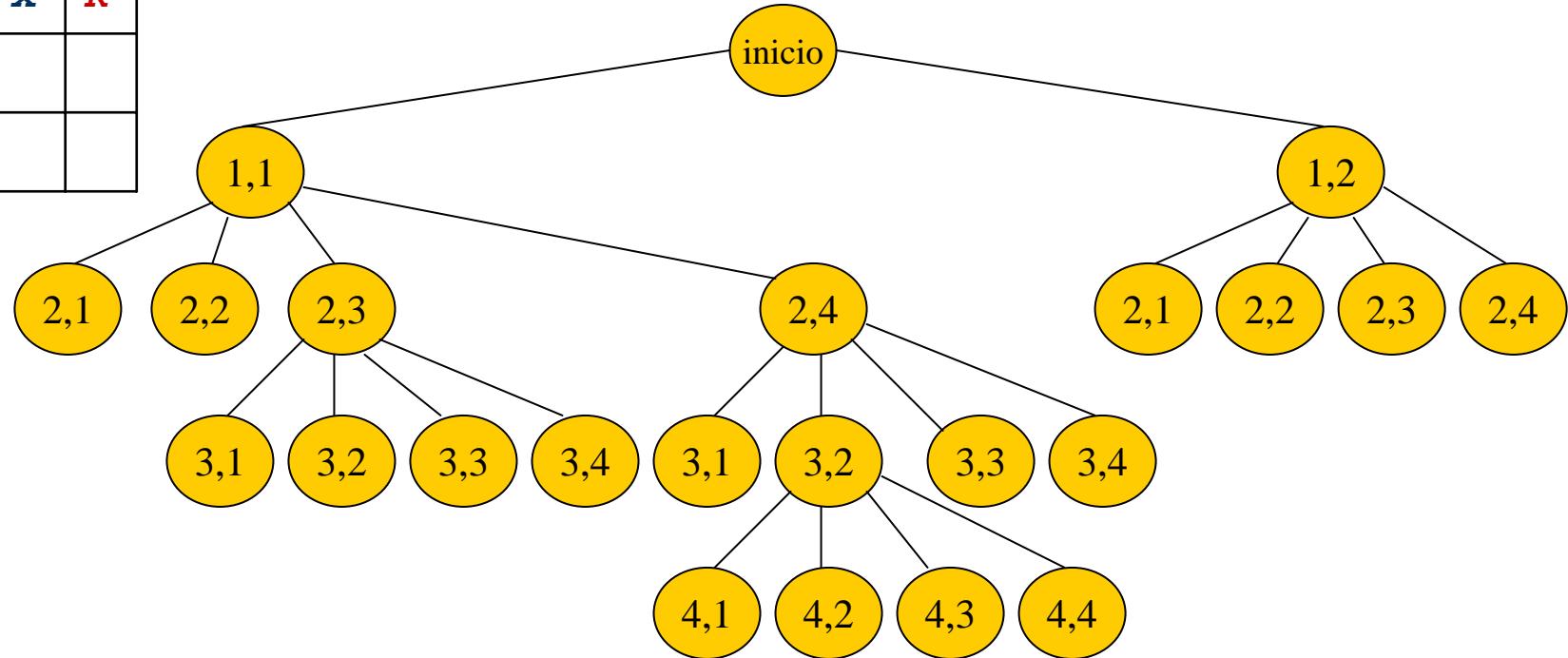


**NO cumple criterio  
(misma diagonal)**

# Ejemplo... para $n = 4$

---

| <b>x</b> | <b>R</b> |          |          |
|----------|----------|----------|----------|
| <b>x</b> | <b>x</b> | <b>x</b> | <b>R</b> |
|          |          |          |          |
|          |          |          |          |

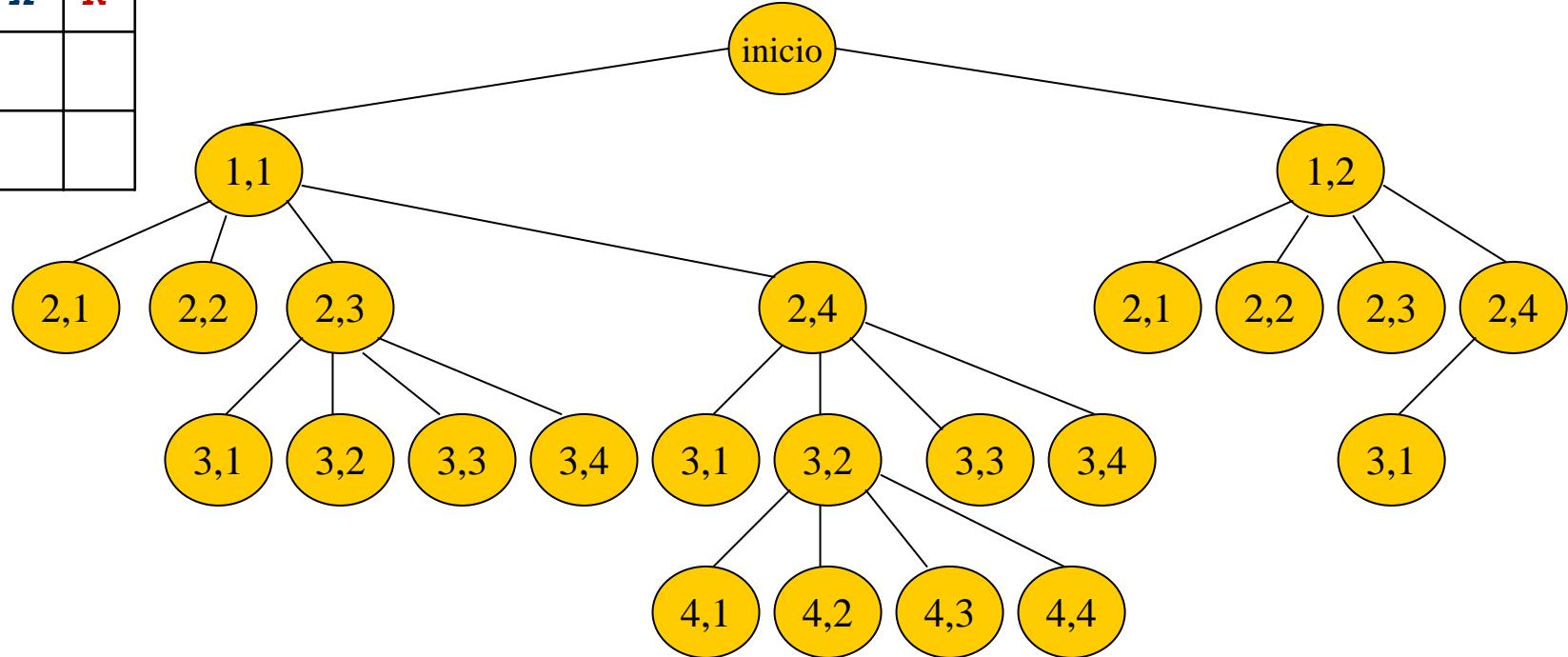


OK... adelante con la  
búsqueda!

# Ejemplo... para $n = 4$

---

|          |          |          |          |
|----------|----------|----------|----------|
| <b>x</b> | <b>R</b> |          |          |
| <b>x</b> | <b>x</b> | <b>x</b> | <b>R</b> |
| <b>R</b> |          |          |          |
|          |          |          |          |

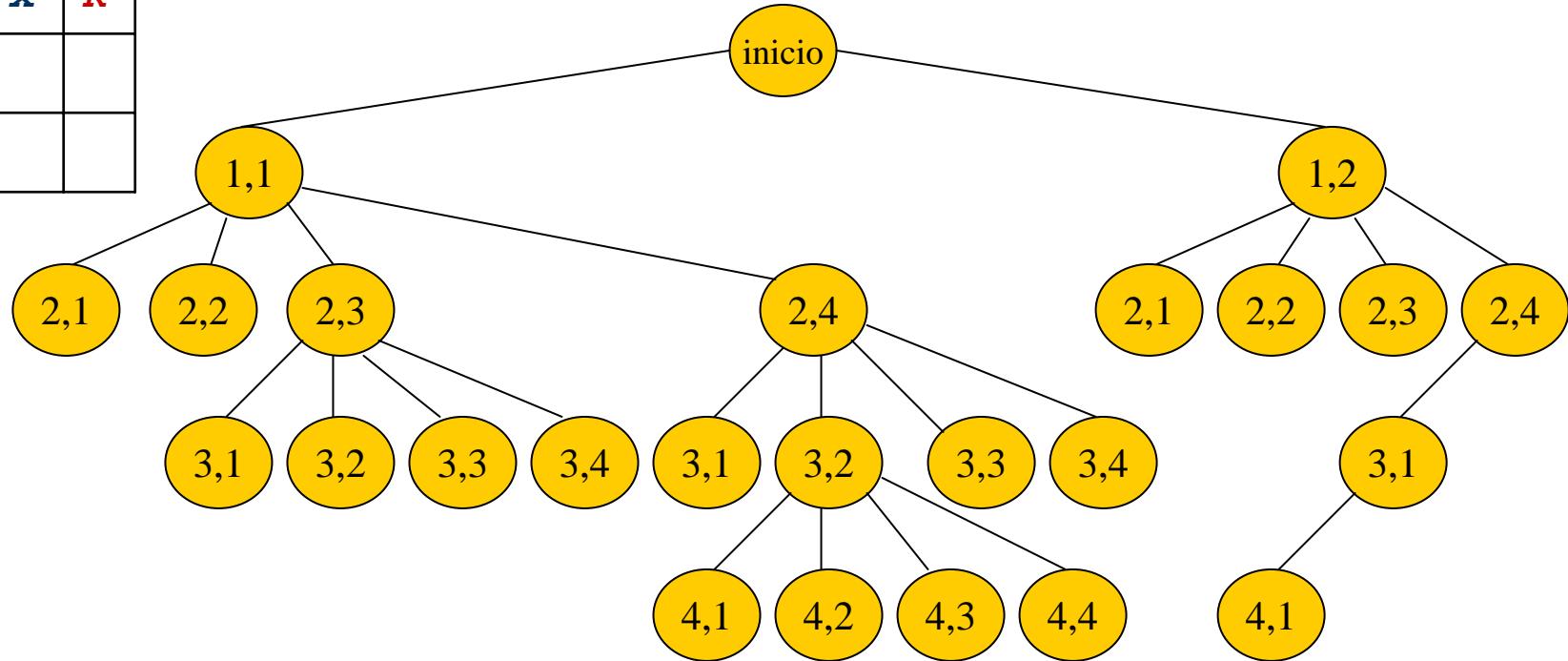


OK... adelante con la  
búsqueda!

# Ejemplo... para $n = 4$

---

|          |          |          |          |
|----------|----------|----------|----------|
| <b>x</b> | <b>R</b> |          |          |
| <b>x</b> | <b>x</b> | <b>x</b> | <b>R</b> |
| <b>R</b> |          |          |          |
| <b>x</b> |          |          |          |

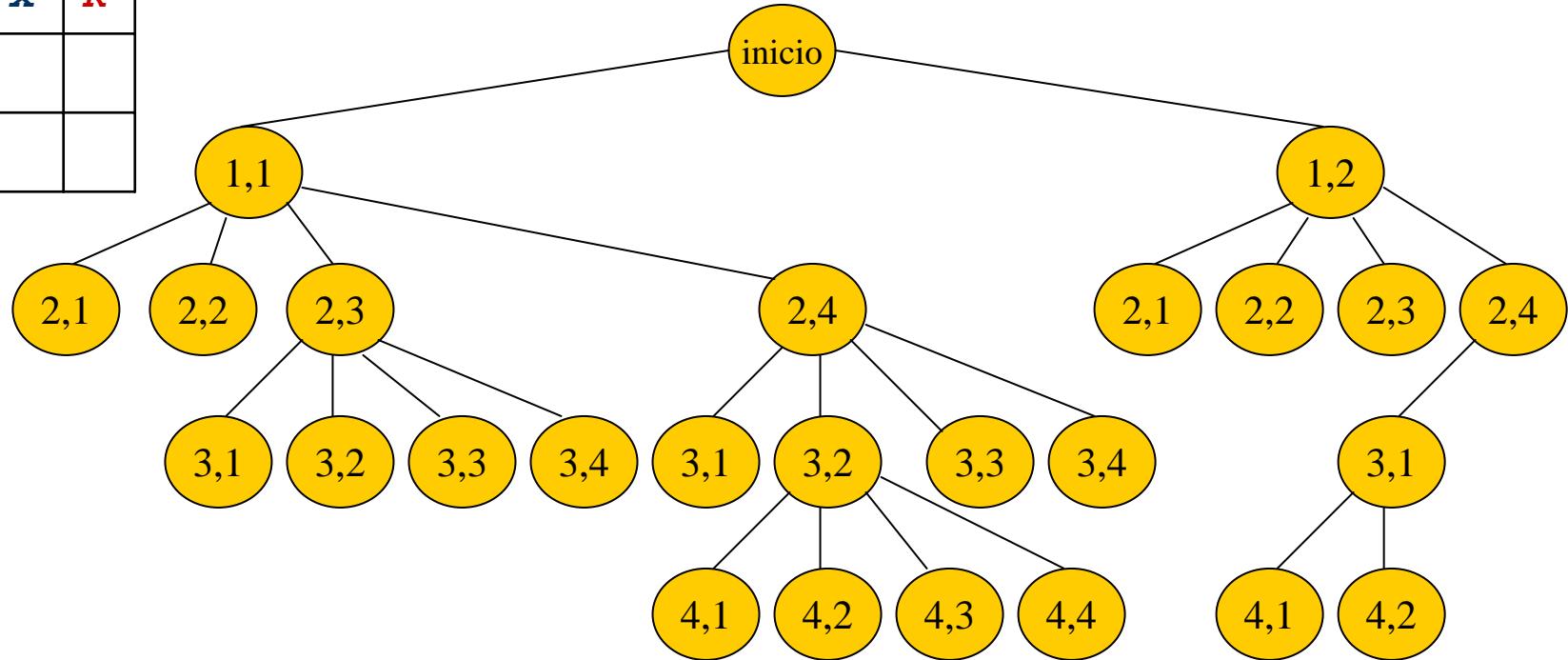


**NO cumple el criterio  
(misma columna)**

# Ejemplo... para $n = 4$

---

|          |          |          |          |
|----------|----------|----------|----------|
| <b>x</b> | <b>R</b> |          |          |
| <b>x</b> | <b>x</b> | <b>x</b> | <b>R</b> |
| <b>R</b> |          |          |          |
| <b>x</b> | <b>x</b> |          |          |

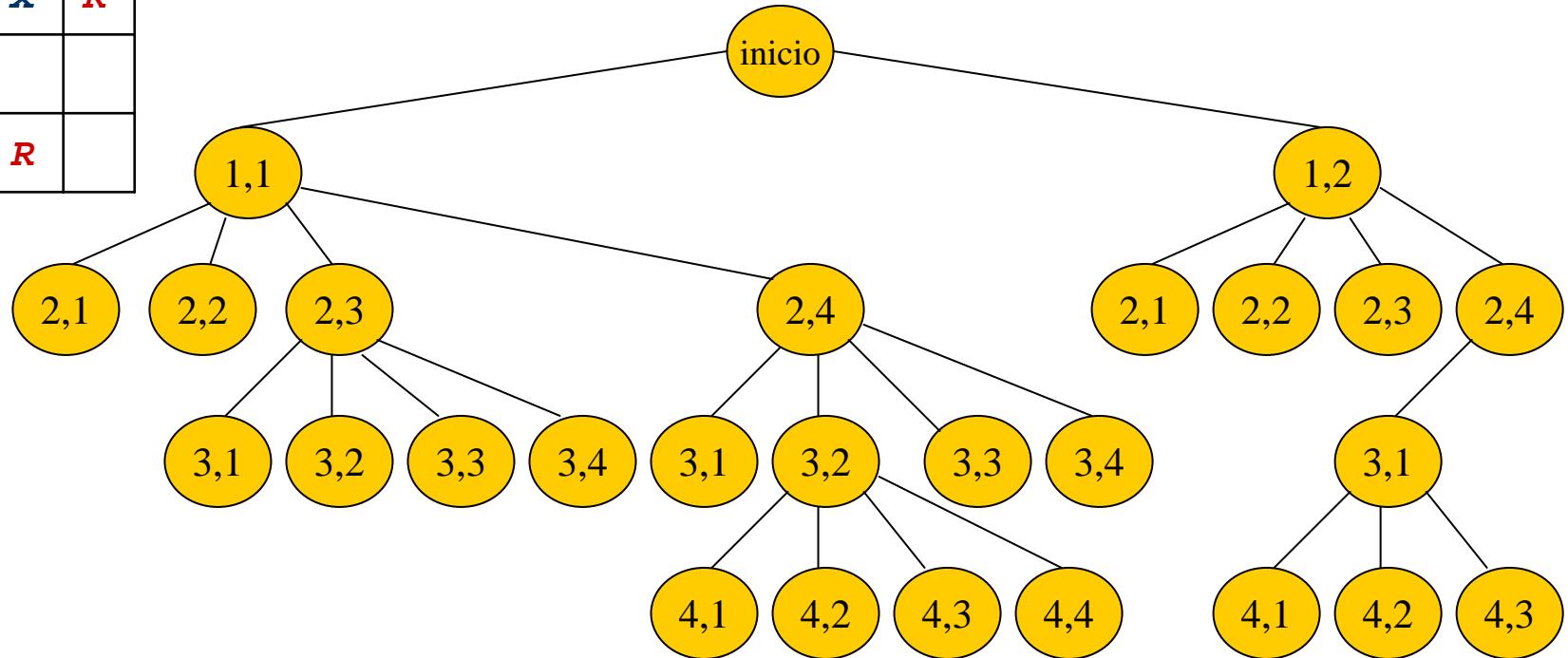


**NO cumple el criterio  
(misma columna)**

# Ejemplo... para $n = 4$

---

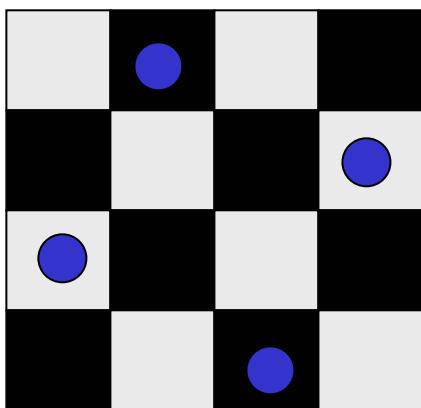
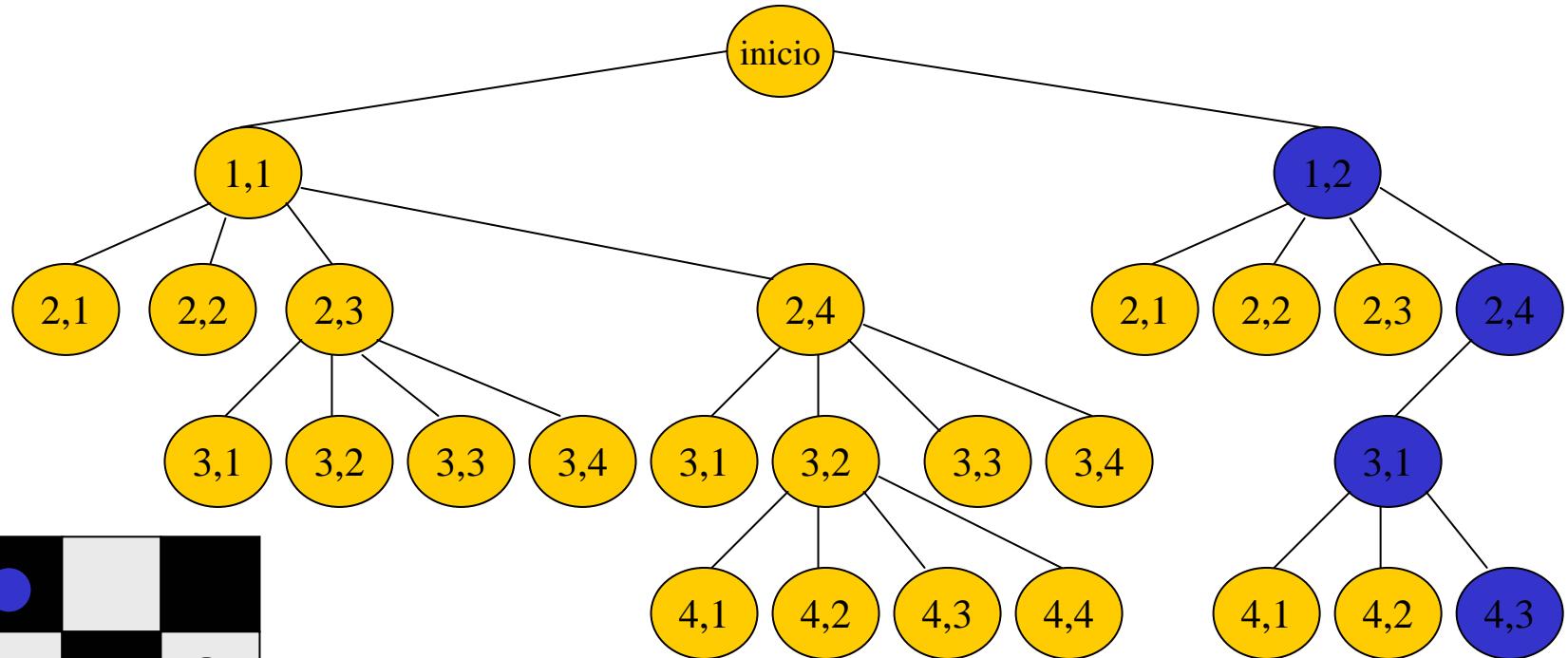
|          |          |          |          |
|----------|----------|----------|----------|
| <b>x</b> | <b>R</b> |          |          |
| <b>x</b> | <b>x</b> | <b>x</b> | <b>R</b> |
| <b>R</b> |          |          |          |
| <b>x</b> | <b>x</b> | <b>R</b> |          |



¡¡OK... se encontró  
solución !!

# Ejemplo... para $n = 4$

---



Se comprobaron 27 nodos... Sin *backtracking* se hubieran comprobado 155 nodos

# Índice

---

## I. LA TÉCNICA BACKTRACKING

### 1. Introducción: El método General

- Resolución de problemas cuando la solución se puede expresar como una n-tupla
- Ejemplo: El problema de las 8 reinas
- Ejemplo: La suma de subconjuntos

### 2. Espacio de soluciones: Organización del Árbol

- Ejemplo: Espacio solución para el problema de las N-reinas ( $N=4$ )
- Ejemplo: Espacio solución para la suma de subconjuntos
- Terminología utilizada para la organización en árbol
- Ejemplo: Backtracking en el problema de las 4 reinas

### 3. Procedimiento Backtracking

- **Procedimiento Iterativo**
- **Procedimiento Recursivo**

# Algoritmo Backtracking

---

- Podemos presentar una **formulación general**, aunque precisa, del proceso de *backtracking*
- Supondremos que hay que encontrar **todos los nodos respuesta**, y no solo uno
- Sea  $(x_1, x_2, \dots, x_i)$  un camino desde la raíz hasta un nodo en el árbol de estados
- Sea  $T(x_1, x_2, \dots, x_i)$  el conjunto de todos los posibles valores  $x_{i+1}$  tales que  $(x_1, x_2, \dots, x_{i+1})$  es también un camino hacia un estado del problema
- Suponemos la **existencia de funciones de acotación**  $B_{i+1}$  (expresadas como predicados) tales que,  $B_{i+1}(x_1, x_2, \dots, x_{i+1})$  es falsa para un camino  $(x_1, x_2, \dots, x_{i+1})$  desde el nodo raíz hasta un estado del problema si el camino no puede extenderse para alcanzar un nodo respuesta
- Así los candidatos para la posición  $i+1$  del vector solución  $X(1..n)$  son aquellos valores que son generados por  $T$  y satisfacen  $B_{i+1}$

# Algoritmo Backtracking

---

- El Procedimiento *Backtrack*, representa el esquema general backtracking haciendo uso de  $T$  y  $B_{i+1}$ .

## Procedimiento **BACKTRACK(n)**

{Todas las soluciones se generan en  $X(1..n)$  y se imprimen tan pronto como se encuentran.  $T(X(1), \dots, X(k-1))$  da todos los posibles valores de  $X(k)$  dado que habíamos escogido  $X(1), \dots, X(k-1)$ . El predicado  $B_k(X(1), \dots, X(k))$  determina los elementos  $X(k)$  que satisfacen las restricciones implícitas}

Begin

$k = 1$

    While  $k > 0$  do

        If queda algún  $X(k)$  no probado tal que

$X(k) \in T(X(1), \dots, X(k-1))$  and  $B_k(X(1), \dots, X(k)) = \text{true}$

            Then if  $(X(1), \dots, X(k))$  es un camino hacia un nodo respuesta

                Then print  $(X(1), \dots, X(k))$

$k = k + 1$

            else  $k = k - 1$

end

# Algoritmo Backtracking recursivo

---

- El siguiente algoritmo, *Rbacktrack*, presenta una formulación recursiva del método, ya que como *backtracking* básicamente es un recorrido en postorden, es natural describirlo así,

## Procedimiento RBACKTRACK( $K$ )

{Se supone que los primeros  $k-1$  valores  $X(1), \dots, X(k-1)$  del vector solución  $X(1..n)$  han sido asignados}

Begin

    for cada  $X(k)$  tal que  $X(k) \in T(X(1), \dots, X(k-1))$  and  
         $B(X(1), \dots, X(k)) = \text{true}$  do  
            If  $(X(1), \dots, X(k))$  es un camino hacia un nodo respuesta  
                Then print  $(X(1), \dots, X(k))$   
                RBACKTRACK( $K+1$ )

End

# Eficiencia de backtracking

---

- La eficiencia de los algoritmos backtracking depende básicamente de **cuatro factores**:
  - el tiempo necesario para generar el siguiente  $X(k)$ ,
  - el número de  $X(k)$  que satisfagan las restricciones explícitas,
  - el tiempo para las funciones de acotación  $B_i$  y
  - el número de  $X(k)$  que satisfagan las  $B_i$  para todo  $i$
- Las funciones de acotación se entienden buenas si reducen considerablemente el número de nodos que generan
- Las buenas funciones de acotación consumen mucho tiempo en evaluaciones, por lo que hay que buscar un equilibrio entre el tiempo global de computación, y la reducción del número de nodos generados.

# Eficiencia de backtracking

---

- De los 4 factores que determinan el tiempo requerido por un algoritmo *backtracking*, **sólo la cuarta**, el número de nodos generados, varía de un caso a otro
- Un algoritmo backtracking en un caso podría generar sólo  $O(n)$  nodos, mientras que en otro (relativamente parecido) **podría generar casi todos los nodos del árbol de espacio de estados**
- Si el número de nodos en el espacio solución es  $2^n$  o  $n!$ , el tiempo del peor caso para el algoritmo *backtracking* sería generalmente  **$O(p(n)2^n)$  u  $O(q(n)n!)$**  respectivamente, con  $p$  y  $q$  polinomios en  $n$
- La importancia del *backtracking* reside en su capacidad para resolver casos con grandes valores de  $n$  en muy poco tiempo
- La dificultad esta en predecir la conducta del algoritmo *backtracking* en el caso que deseemos resolver

# Eficiencia de backtracking

---

- Podemos estimar el número de nodos que se generaran con un algoritmo *backtracking* usando el **método de Monte Carlo**
- Se trata de **generar un camino aleatorio** en el árbol de estados
- Sea  $X$  un nodo en ese camino aleatorio. Supongamos que  $X$  esta en el nivel  $i$  del árbol del espacio de estados
- Las funciones de acotación se usan en el nodo  $X$  para determinar el numero  $m_i$  de sus hijos que no hay que acotar. El siguiente nodo en el camino se obtiene seleccionando aleatoriamente uno de estos  $m_i$  hijos que no se han acotado
- La generación del camino termina en un nodo que sea una hoja o cuyos hijos vayan a acotarse. Usando estos  $m_i$ , **podemos estimar el numero total**,  $m$ , de nodos en el árbol del espacio de estados que no se acotarán

# Índice

---

## **II. SOLUCIONES BACKTRACKING EN DISTINTOS PROBLEMAS**

- 1. El Problema de las 8 Reinas**
- 2. La Suma de Subconjuntos**
- 3. El Problema del Viajante de Comercio**
- 4. El Problema del Coloreo de un Grafo**
- 5. Laberintos y Backtracking**

# Solución para las 8 reinas

---

- Generalizamos el problema para considerar un tablero  $n \times n$  y encontrar todas las formas de colocar  $n$  reinas que no se ataquen
- Podemos tomar  $(x_1, \dots, x_n)$  representando una solución si  $x_i$  es la columna de la  $i$ -ésima fila en la que la reina  $i$  está colocada
- Los  $x_i$ 's serán todos distintos ya que no puede haber dos reinas en la misma columna
- ¿Cómo comprobar que dos reinas no estén en la misma diagonal?

|       |       |       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|
|       |       |       |       | (1,4) |       |       |       |       |
| (2,1) |       |       | (2,3) |       |       |       |       |       |
|       | (3,2) |       |       |       |       |       |       |       |
| (4,1) |       |       | (4,3) |       |       |       |       |       |
|       |       |       |       | (5,4) |       |       |       | (5,8) |
| (6,1) |       |       |       |       | (6,5) |       | (6,7) |       |
|       | (7,2) |       |       |       |       | (7,6) |       |       |
|       |       | (8,3) |       |       | (8,5) |       | (8,7) |       |

# Solución para las 8 reinas

---

- Si las casillas del tablero se numeran como una matriz  $A(1..n,1..n)$ , cada elemento en la misma diagonal que vaya de la parte superior izquierda a la inferior derecha, tiene el mismo valor "*fila-columna*"
- También, cualquier elemento en la misma diagonal que vaya de la parte superior derecha a la inferior izquierda, tiene el mismo valor "*fila+columna*"
- Si dos reinas están colocadas en las posiciones  $(i,j)$  y  $(k,l)$ , estarán en la misma diagonal sólo si,  
$$i - j = k - l \quad \text{ó} \quad i + j = k + l$$
- La primera ecuación implica que  
$$j - l = i - k$$
- La segunda que  
$$j - l = k - i$$
- Así, dos reinas están en la misma diagonal si y solo si  
$$|j-l| = |i-k|$$

# Solución para las 8 reinas

---

- El procedimiento COLOCA( $k$ ) devuelve verdad si la  $k$ -ésima reina puede colocarse en el valor actual de  $X(k)$ . Testea si  $X(k)$  es distinto de todos los valores previos  $X(1), \dots, X(k-1)$ , y si hay alguna otra reina en la misma diagonal. Su tiempo de ejecución de  $O(k-1)$

## Procedimiento COLOCA( $K$ )

{ $X$  es un array cuyos  $k$  primeros valores han sido ya asignados.  $ABS(r)$  da el valor absoluto de  $r$ }

Begin

For  $i:=1$  to  $k-1$  do

    If  $X(i) = X(k)$  or  $ABS(X(i)-X(k)) = ABS(i-k)$

        Then return (false)

    Return (true)

end

# Solución para las 8 reinas

---

## Procedimiento NREINAS(N)

{Usando *backtracking* este procedimiento imprime todos los posibles emplazamientos de n reinas en un tablero  $n \times n$  sin que se ataquen}

Begin

$X(1) := 0, k := 1$

{k es la fila actual}

While  $k > 0$  do

{hacer para todas las filas}

$X(k) := X(k) + 1$

{mover a la siguiente columna}

While  $X(k) \leq n$  and not COLOCA ( $k$ ) do

{puede moverse esta reina?}

$X(k) := X(k) + 1$

If  $X(k) \leq n$

{Se encontró una posición?}

Then if  $k = n$

{Es una solución completa?}

    Then print ( $X$ )

    Else  $k := k + 1; X(k) := 0$

{Ir a la siguiente fila}

{Backtrack}

Else  $k := k - 1$

end

# Solución para la suma de subconjuntos

---

- Tenemos  $n$  números positivos distintos (usualmente llamados pesos) y queremos encontrar todas las combinaciones de estos números que sumen  $M$
- Los anteriores ejemplos 2 y 3 mostraron como podríamos formular este problema usando tamaños de las tuplas fijos o variables
- Consideraremos una solución backtracking usando la estrategia del **tamaño fijo de las tuplas**
- En este caso el elemento  $X(i)$  del vector solución es uno o cero, dependiendo de si el peso  $W(i)$  esta incluido o no.

# Solución para la suma de subconjuntos

---

- Generación de los hijos de cualquier nodo en el árbol:
- Para un nodo en el nivel  $i$ , el hijo de la izquierda corresponde a  $X(i) = 1$ , y el de la derecha a  $X(i) = 0$
- Una posible elección de **funciones de acotación** es  $B_k(X(1), \dots, X(k)) = \text{true}$  si y solo si,  
$$\sum_{1..k} W(i)X(i) + \sum_{k+1..n} W(i) \geq M$$
- Claramente  $X(1), \dots, X(k)$  no pueden conducir a un nodo respuesta si no se verifica esta condición

# Solución para la suma de subconjuntos

---

- Las funciones de acotación pueden fortalecerse si suponemos los  $W(i)$ 's en orden creciente
- En este caso,  $X(1), \dots, X(k)$  no pueden llevar a un nodo respuesta si

$$\sum_{1..k} W(i)X(i) + W(k+1) > M$$

- Por tanto las funciones de acotación que usaremos serán las definidas de la siguiente forma:  $B_k(X(1), \dots, X(k))$  es *true* si y sólo si

$$\sum_{1..k} W(i)X(i) + \sum_{k+1..n} W(i) \geq M$$

y

$$\sum_{1..k} W(i)X(i) + W(k+1) \leq M$$

# Solución para la suma de subconjuntos

---

- Ya que nuestro algoritmo no hará uso de  $B_n$ , no necesitamos preocuparnos por la posible aparición de  $W(n+1)$  en esta función
- Aunque hasta aquí hemos especificado todo lo que es necesario para usar cualquiera de los esquemas Backtracking, resultaría un algoritmo mas simple si diseñamos a la medida del problema que estemos tratando cualquiera de esos esquemas
- Esta simplificación resulta de la comprobación de que si  $X(k) = 1$ , entonces

$$\sum_{1..k} W(i)X(i) + \sum_{k+1..n} W(i) > M$$

# Esquema de algoritmo recursivo

---

## Procedimiento SUMASUB ( $s, k, r$ )

{Los valores de  $X(j)$ ,  $1 \leq j < k$ , ya han sido determinados.  $s = \sum_{1..k-1} W(j)X(j)$  y  $r = \sum_{k..n} W(j)$ . Los  $W(j)$  están en orden creciente. Se supone que  $W(1) \leq M$  y que  $\sum_{1..n} W(i) \geq M\}$

Begin

{Generación del hijo izquierdo. Nótese que  $s + W(k) \leq M$  ya que  $B_{k-1} = true\}$

$X(k) = 1$

{4} If  $s + W(k) = M$

{5} Then For  $i = 1$  to  $k$  print  $X(i)$

Else

{7} If  $s + W(k) + W(k+1) \leq M$

Then SUMASUB ( $s + W(k), k+1, r-W(k)$ )

{Generación del hijo derecho y evaluación de  $B_k$ }

If  $s + r - W(k) \geq M$  and  $s + W(k+1) \leq M$

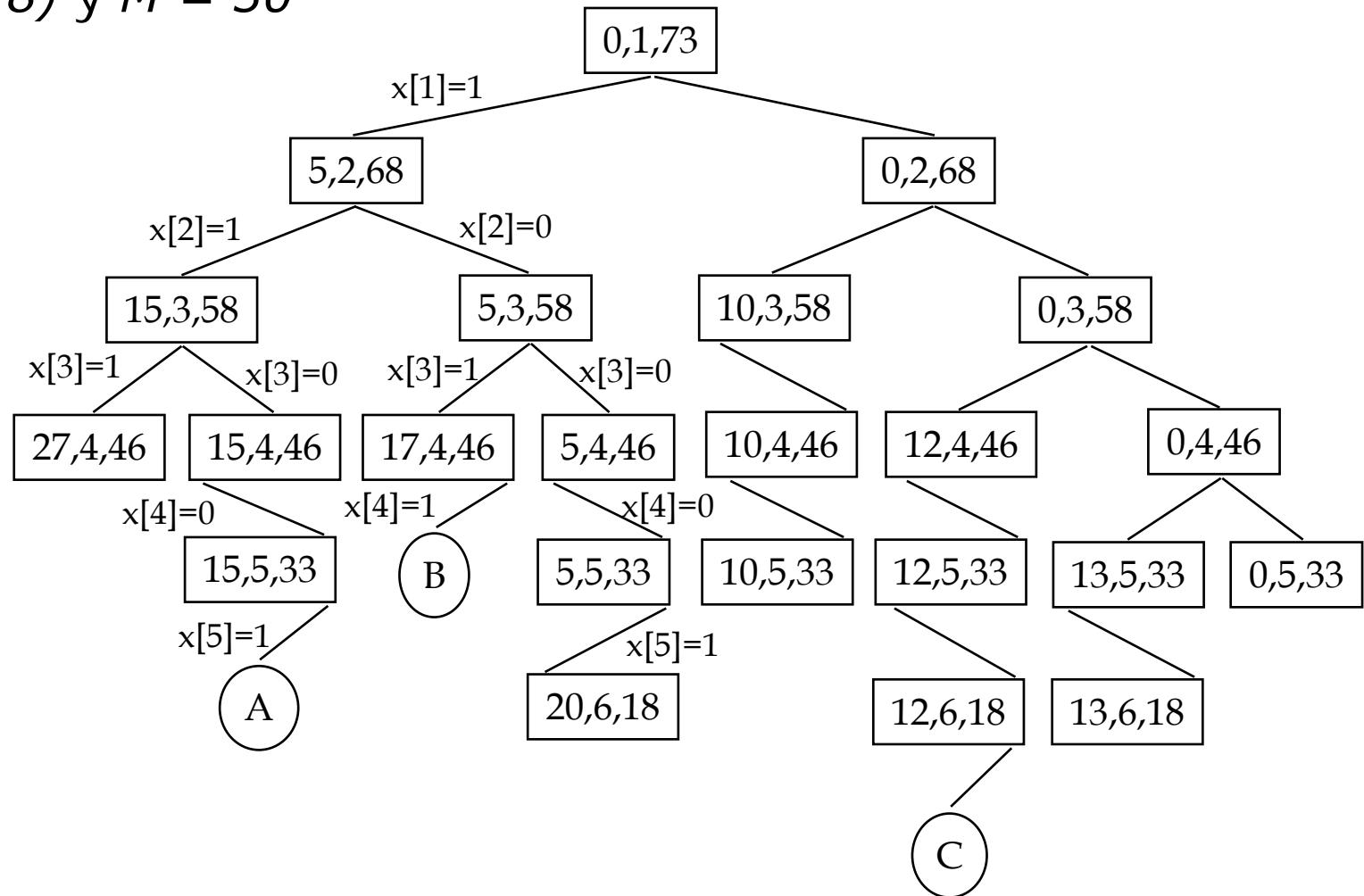
Then  $X(k) = 0$

SUMASUB( $s, k+1, r-w(k)$ )

end

# Ejemplo

Como trabaja SUMASUB para el caso en que:  $W = (5, 10, 12, 13, 15, 18)$  y  $M = 30$



# Problema de asignación de tareas

- Existen  $n$  personas y  $n$  trabajos.
- Cada persona  $i$  puede realizar un trabajo  $j$  con más o menos rendimiento:  $B[i, j]$ .
- **Objetivo:** asignar una tarea a cada trabajador (asignación uno-a-uno), de manera que se maximice la suma de rendimientos.

|          |   | Tareas |   |   |
|----------|---|--------|---|---|
|          |   | 1      | 2 | 3 |
| Personas | B | 4      | 9 | 1 |
|          | 1 | 7      | 2 | 3 |
|          | 3 | 6      | 3 | 5 |

**Ejemplo 1.** ( $P_1, T_1$ ), ( $P_2, T_3$ ), ( $P_3, T_2$ )

$$B_{\text{TOTAL}} = 4 + 3 + 3 = 10$$

**Ejemplo 2.** ( $P_1, T_2$ ), ( $P_2, T_1$ ), ( $P_3, T_3$ )

$$B_{\text{TOTAL}} = 9 + 7 + 5 = 21$$

# Problema de asignación de tareas

- El problema de asignación es un problema **NP-completo** clásico.
- Otras variantes y enunciados:
  - Problema de los **matrimonios estables**.
  - Problemas con **distinto número** de tareas y personas.  
Ejemplo: problema de los árbitros.
  - Problemas de **asignación de recursos**: fuentes de oferta y de demanda. Cada fuente de oferta tiene una capacidad  $O[i]$  y cada fuente de demanda una  $D[j]$ .
  - **Isomorfismo de grafos**: la matriz de pesos varía según la asignación realizada.

# Problema de asignación de tareas

## Enunciado del problema de asignación

- **Datos del problema:**
  - **n**: número de personas y de tareas disponibles.
  - **B**: array [1..n, 1..n] de entero. Rendimiento o beneficio de cada asignación. **B[i, j]** = beneficio de asignar a la persona **i** la tarea **j**.
- **Resultado:**
  - Realizar **n** asignaciones  $\{(p_1, t_1), (p_2, t_2), \dots, (p_n, t_n)\}$ .
- **Formulación matemática:**

Maximizar  $\sum_{i=1..n} B[p_i, t_i]$ , sujeto a la restricción  $p_i \neq p_j, t_i \neq t_j, \forall i \neq j$

[proceso](#)

# Problema de asignación de tareas

## 1) Representación de la solución

- Mediante **pares de asignaciones**:  $s = \{(p_1, t_1), (p_2, t_2), \dots, (p_n, t_n)\}$ , con  $p_i \neq p_j$ ,  $t_i \neq t_j$ ,  $\forall i \neq j$ 
  - La tarea  $t_i$  es asignada a la persona  $p_i$ .
  - Árbol muy ancho. Hay que garantizar muchas restricciones. Representación no muy buena.
- Mediante **matriz de asignaciones**:  $s = ((a_{11}, a_{12}, \dots, a_{1n}), (a_{21}, a_{22}, \dots, a_{2n}), \dots, (a_{n1}, a_{n2}, \dots, a_{nn}))$ , con  $a_{ij} \in \{0, 1\}$ , y con  $\sum_{i=1..n} a_{ij} = 1$ ,  $\sum_{j=1..n} a_{ij} = 1$ .
  - $a_{ij} = 1 \rightarrow$  la tarea  $j$  se asigna a la persona  $i$
  - $a_{ij} = 0 \rightarrow$  la tarea  $j$  no se asigna a la persona  $i$

|          |   | Tareas |   |   |
|----------|---|--------|---|---|
|          |   | 1      | 2 | 3 |
| Personas | a | 1      | 2 | 3 |
|          | 1 | 0      | 1 | 0 |
|          | 2 | 1      | 0 | 0 |
|          | 3 | 0      | 0 | 1 |

# Problema de asignación de tareas

## 1) Representación de la solución

- Mediante **matriz de asignaciones**.
  - Árbol binario, pero muy profundo:  $n^2$  niveles en el árbol.
  - También tiene muchas restricciones.
-  Desde el punto de vista de las **personas**:  $s = (t_1, t_2, \dots, t_n)$ , siendo  $t_i \in \{1, \dots, n\}$ , con  $t_i \neq t_j, \forall i \neq j$ 
  - $t_i \rightarrow$  número de tarea asignada a la persona  $i$ .
  - Da lugar a un árbol permutacional. ¿Cuánto es el número de nodos?
- Desde el punto de vista de las **tareas**:  $s = (p_1, p_2, \dots, p_n)$ , siendo  $p_i \in \{1, \dots, n\}$ , con  $p_i \neq p_j, \forall i \neq j$ 
  - $p_i \rightarrow$  número de persona asignada a la tarea  $i$ .
  - Representación análoga (dual) a la anterior.

# Problema de asignación de tareas

## 2) Elegir el esquema de algoritmo: caso optimización.

Backtracking (var s: array [1..n] de entero)

nivel:= 1; s:= s<sub>INICIAL</sub>

voa:= -∞; soa:= Ø

bact:= 0

repetir

    Generar (nivel, s)

**si** Solución (nivel, s) AND (bact > voa) **entonces**

        voa:= bact; soa:= s

**si** Criterio (nivel, s) **entonces**

        nivel:= nivel + 1

**sino**

**mientras** NOT MasHermanos (nivel, s) AND (nivel>0)

**hacer** Retroceder (nivel, s)

**finsi**

**hasta** nivel == 0

**bact:** Beneficio actual

# Problema de asignación de tareas

## 3) Funciones genéricas del esquema.

- **Variables:**
  - **s: array [1..n] de entero:** cada **s[i]** indica la tarea asignada a la persona **i**. Inicializada a 0.
  - **bact:** beneficio de la solución actual
- **Generar (nivel, s)** → Probar primero 1, luego 2, ..., n
  - s[nivel]:= s[nivel] + 1**
  - si s[nivel]==1 entonces bact:= bact + B[nivel, s[nivel]]**
  - sino bact:= bact + B[nivel, s[nivel]] – B[nivel, s[nivel]-1]**
- **Criterio (nivel, s)**
  - para i:= 1, ..., nivel-1 hacer**
    - si s[nivel] == s[i] entonces devolver false**
    - finpara**
    - devolver true**

# Problema de asignación de tareas

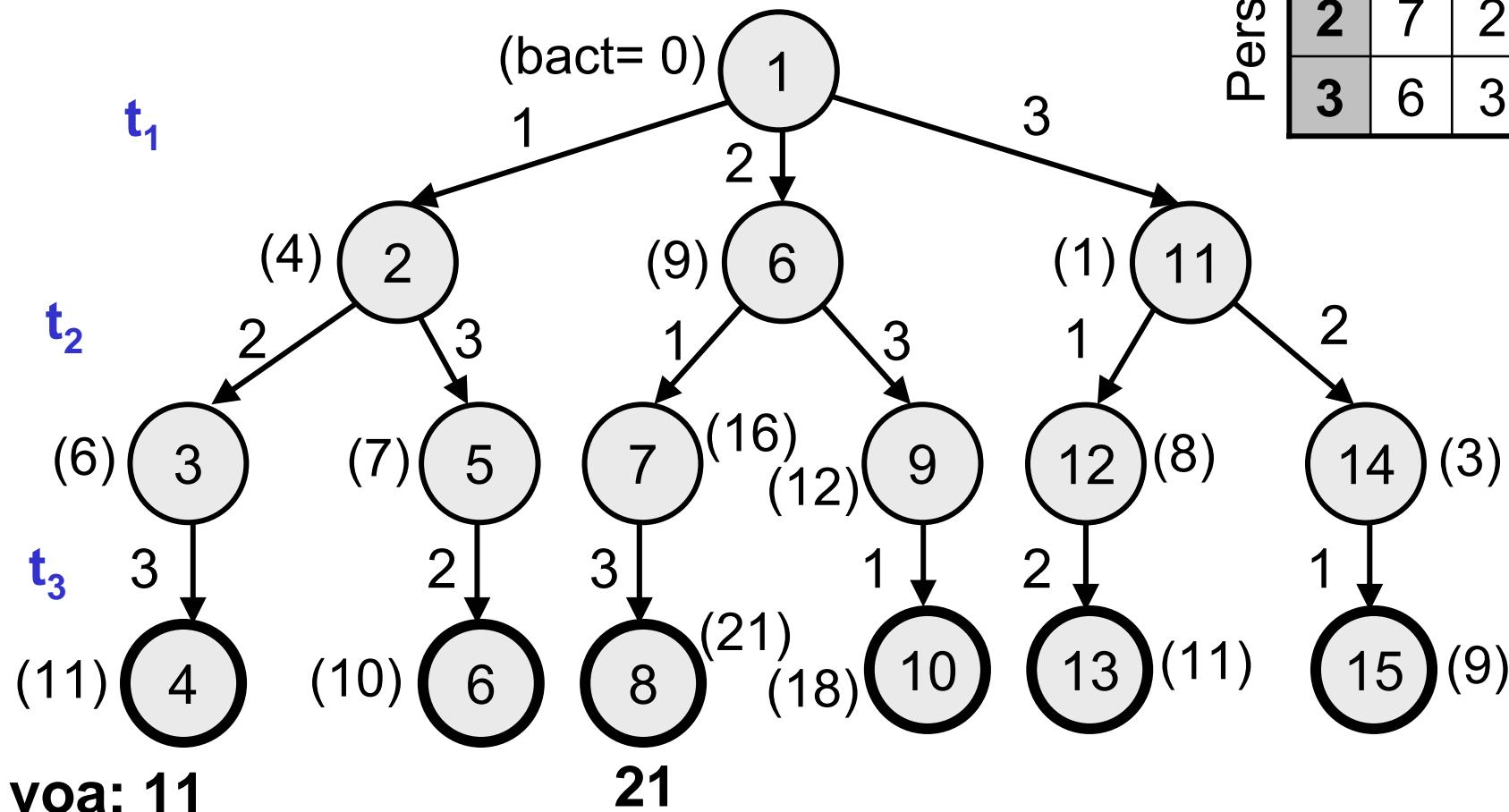
## 3) Funciones genéricas del esquema.

- **Solución (nivel, s)**  
**devolver** (nivel==n) AND Criterio (nivel, s)
- **MasHermanos (nivel, s)**  
**devolver** s[nivel] < n
- **Retroceder (nivel, s)**  
bact:= bact – B[nivel, s[nivel]]  
s[nivel]:= 0  
nivel:= nivel – 1

# Problema de asignación de tareas

Tareas

- Ejemplo de aplicación.  $n = 3$



# Problema de asignación de tareas

- **Problema:** la función **Criterio** es muy lenta, repite muchas comprobaciones.
- **Solución:** usar un array que indique las tareas que están ya usadas en la asignación actual.
  - **usada:** array [0..n] de entero.
  - **usada[i]** indica el número de veces que es usada la tarea *i* en la planificación actual (es decir, en **s**).
  - **Inicialización:** **usada[i] = 0**, para todo *i*.
  - Modificar las funciones del esquema.

# Problema de asignación de tareas

## 3) Funciones genéricas del esquema.

- Criterio (nivel, s)

devolver usada[s[nivel]]==1

- Retroceder (nivel, s)

bact:= bact – B[nivel, s[nivel]]

**usada[s[nivel]]--**

s[nivel]:= 0

nivel:= nivel – 1

- Generar (nivel, s)

**usada[s[nivel]]--**

s[nivel]:= s[nivel] + 1

**usada[s[nivel]]++**

**si** s[nivel]==1 **entonces** bact:= bact + B[nivel, s[nivel]]

**sino** bact:= bact + B[nivel, s[nivel]] – B[nivel, s[nivel]-1]

# Problema de asignación de tareas

## 3) Funciones genéricas del esquema.

- Las funciones **Solución** y **MasHermanos** no se modifican.
- **Conclusiones:**
  - El algoritmo sigue siendo muy ineficiente.
  - Aunque garantiza la solución óptima...

# El viajante de comercio

---

- Sea  $G = (V, E)$  un grafo conexo con  $n$  vértices. Un **ciclo Hamiltoniano** es un camino circular a lo largo de los  $n$  vértices de  $G$  que visita cada vértice de  $G$  una vez y vuelve al vértice de partida, que naturalmente es visitado dos veces
- Estamos interesados en construir un algoritmo backtracking que determine todos los ciclos Hamiltonianos de  $G$ , que puede ser dirigido o no (para quedarnos con el de mínimo costo)
- El vector backtracking solución  $(x_1, \dots, x_n)$  se define de modo que  $x_i$  represente el  $i$ -ésimo vértice visitado en el ciclo propuesto

# El viajante de comercio

---

- Todo lo que se necesita es **determinar** como calcular el conjunto de **posibles vértices para  $x_k$**  si ya hemos elegido  $x_1, \dots, x_{k-1}$
- Si  $k = 1$ , entonces  $X(1)$  puede ser cualquiera de los  $n$  vértices
- Para evitar imprimir el mismo ciclo  $n$  veces, exigimos que  $X(1) = 1$
- Si  $1 < k < n$ , entonces  $X(k)$  puede ser cualquier vértice  $v$  que sea distinto de  $X(1), X(2), \dots, X(k-1)$  que esté conectado por una arista a  $X(k-1)$
- $X(n)$  sólo puede ser el único vértice restante y debe estar conectado a  $X(n-1)$  y a  $X(1)$

# El viajante de comercio

---

- El procedimiento `SiguienteValor(k)` devuelve el siguiente vértice válido para la posición  $k$  o  $0$  si no queda ninguno

## Algoritmo `SiguienteValor (k)`

{ $x$  es un array cuyos  $k-1$  primeros valores han sido ya asignados,  $k$  es la posición del siguiente valor a asignar y  $G$  la matriz de adyacencia que representa el grafo}

```
while (true)
    value = (x[k]++) mod (N+1)
    if (value = 0) then return value
    if (G[x[k-1],value]) {chequeo de distinguibilidad}
        for j = 1 to k-1 if x[j] = value then break
        if (j=k) and (k<N or (k=N and G[x[N],x[1]])))
            then return value
endWhile
```

# El viajante de comercio

---

## Procedimiento Hamiltoniano (k)

{ $x$  es un array cuyos  $k-1$  primeros valores han sido ya asignados,  $k$  es la posición del siguiente valor a colocar (nodo en expansión o e-nodo)}

while

```
     $x[k] = \text{SiguienteValor}(k)$ 
    if ( $x[k] = 0$ ) then return
    if ( $k = N$ ) then print solución
    else Hamiltoniano ( $k+1$ )
```

endWhile

Utilizando este algoritmo podemos particularizar el esquema backtracking recursivo para encontrar todos los ciclos hamiltonianos

# El problema del coloreo de un grafo

---

- Sea  $G$  un grafo y  $m$  un número entero positivo. Queremos saber si los nodos de  $G$  pueden colorearse de tal forma que **no haya dos vértices adyacentes que tengan el mismo color**, y que sólo se usen  $m$  colores para esa tarea
- Este es el **problema de la  $m$ -colorabilidad**
- El problema de optimización de la  $m$ -colorabilidad, pregunta por el menor numero  $m$  con el que el grafo  $G$  puede colorearse. A ese entero se le denomina **Número Cromático del grafo**

# El problema del coloreo de un grafo

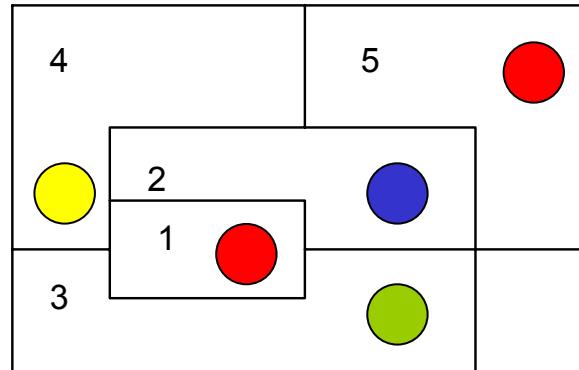
---

- Un grafo se llama plano si y sólo si puede pintarse en un plano de modo que ningún par de aristas se corten entre sí
- Un caso especial famoso del problema de la  $m$ -colorabilidad es **el problema de los cuatro colores** para grafos planos que, dado un mapa cualquiera, consiste en saber si ese mapa podrá pintarse de manera que no haya dos zonas colindantes con el mismo color, y además pueda hacerse ese coloreo sólo con cuatro colores
- Este problema es fácilmente traducible a la nomenclatura de grafos

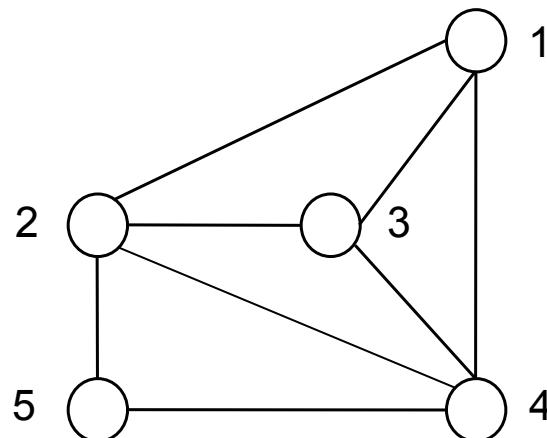
# El problema del coloreo de un grafo

---

- El mapa



puede traducirse en el siguiente grafo



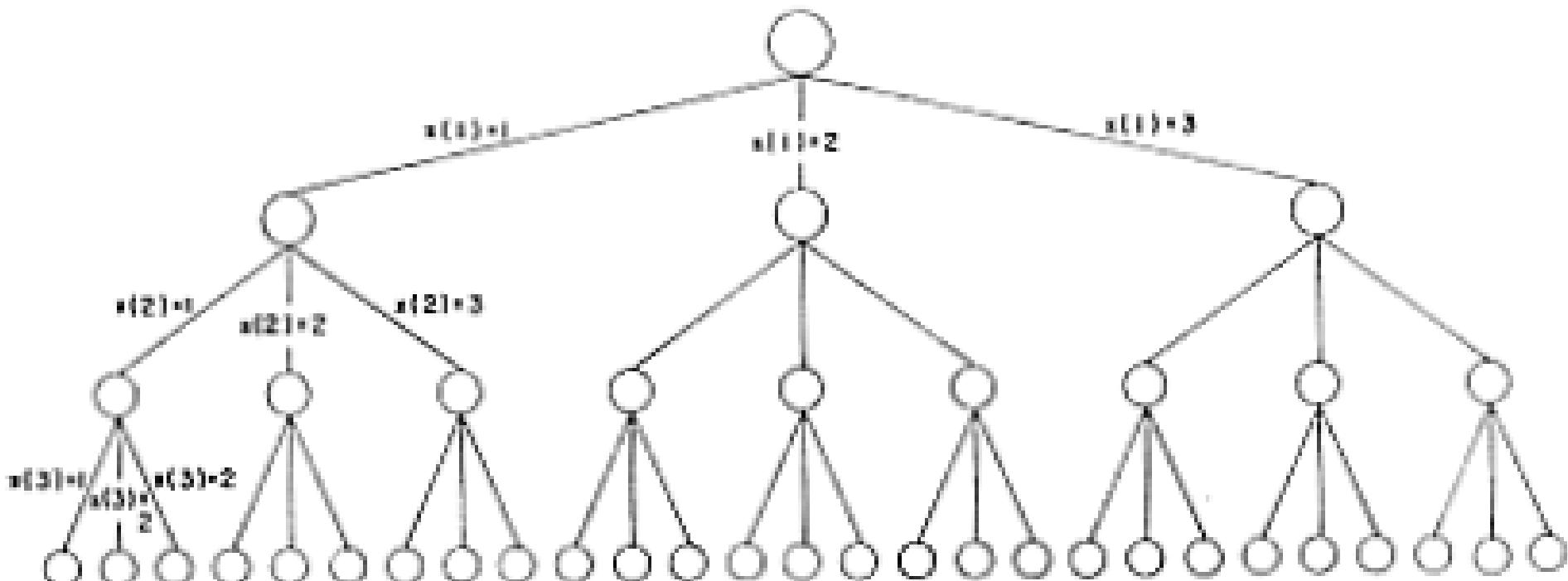
# El problema del coloreo de un grafo

---

- Representamos el grafo por su **matriz de adyacencia**  $\text{GRAFO}(1:n, 1:n)$  siendo  $\text{GRAFO}(i,j) = \text{true}$  si  $(i,j)$  es una arista de  $G$ . En otro caso  $\text{GRAFO}(i,j) = \text{false}$
- Los colores se representan por los enteros  $1, 2, \dots, m$
- Las soluciones vendrán dadas por  $n$ -tuplas  $(X(1), \dots, X(n))$ , donde  $X(i)$  será el color del vértice  $i$
- Usando la formulación recursiva del procedimiento backtracking, puede construirse un algoritmo que trabaja en un tiempo  $O(nm^n)$

# El problema del coloreo de un grafo

- El espacio de estados subyacente es un árbol de **grado  $m$  y altura  $n+1$** , en el que cada nodo en el nivel  $i$  tiene  $m$  hijos correspondientes a las  $m$  posibles asignaciones para  $X(i)$ ,  $1 \leq i \leq n$ , y donde los nodos en el nivel  $n+1$  son nodos hoja



# El problema del coloreo de un grafo

## Algoritmo $M$ -Color ( $k$ )

**while (true)**

# SiguienteValor( $k$ )

if ( $\text{color}[k] = 0$ ) then break

if ( $k = n$ )

then print este coloreo

else  $M$ -Color ( $k + 1$ )

endWhile

- (1) {no hay mas colores para k}
  - (2) {se encontró un coloreo valido para todos los nodos}
  - (3) {intenta colorear el siguiente nodo}

# El problema del coloreo de un grafo

---

## Algoritmo SiguienteValor ( $k$ )

{Devuelve los posibles colores de  $X(k)$  dado que  $X(1)$  hasta  $X(k-1)$  ya han sido coloreados}

```
while (true)
    color[k] = (color[k] + 1) mod (m + 1) {ó n para asegurar}
    if (color[k] = 0) then return (1)
    for i = 1 to k-1
        if (conec[i,k] and color[i] = color[k])
            then break
    endfor
    if (i = k) return (2)
endWhile
```

(1) no hay mas colores para probar

(2) Se ha encontrado un nuevo color (ningún nodo colisiona)

# Eficiencia del algoritmo

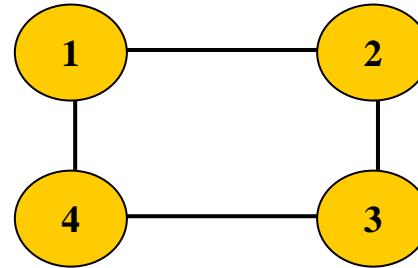
---

- El número de nodos internos en el espacio de estados es  $\sum_{i=0..n-1} m^i$
- En cada nodo interno *SiguienteValor invierte O(nm)* en determinar el hijo correspondiente a un coloreo legal
- El tiempo total esta acotado por  
$$\sum_{i=1..n} m^i n = n(m^{n+1}-1)/(m-1) = O(n m^n)$$

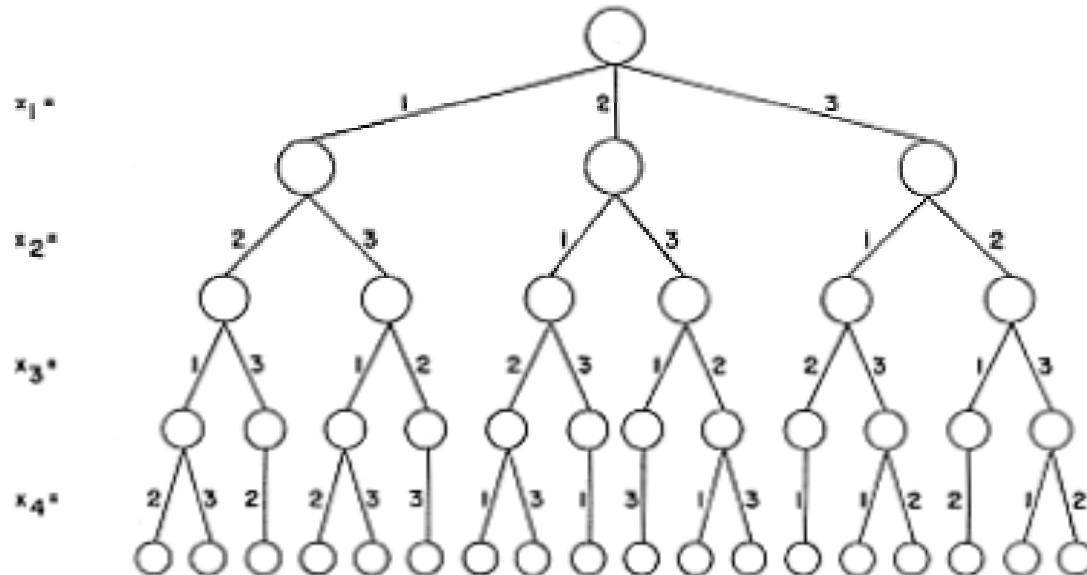
# Ejemplo de coloreo

---

Si consideramos el siguiente grafo

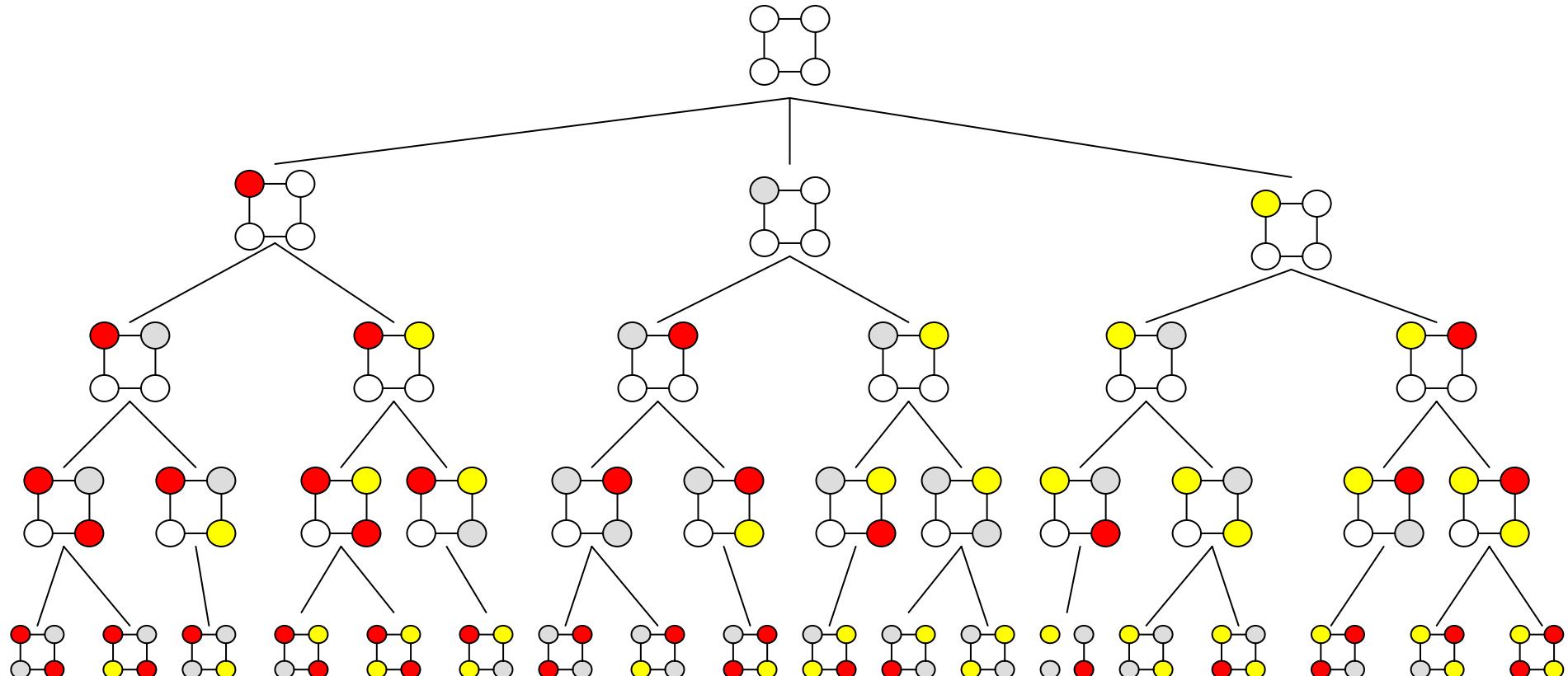


El árbol que genera  $M$ -Color es



# Otra representación del ejemplo

---



# Laberintos y Backtracking

---



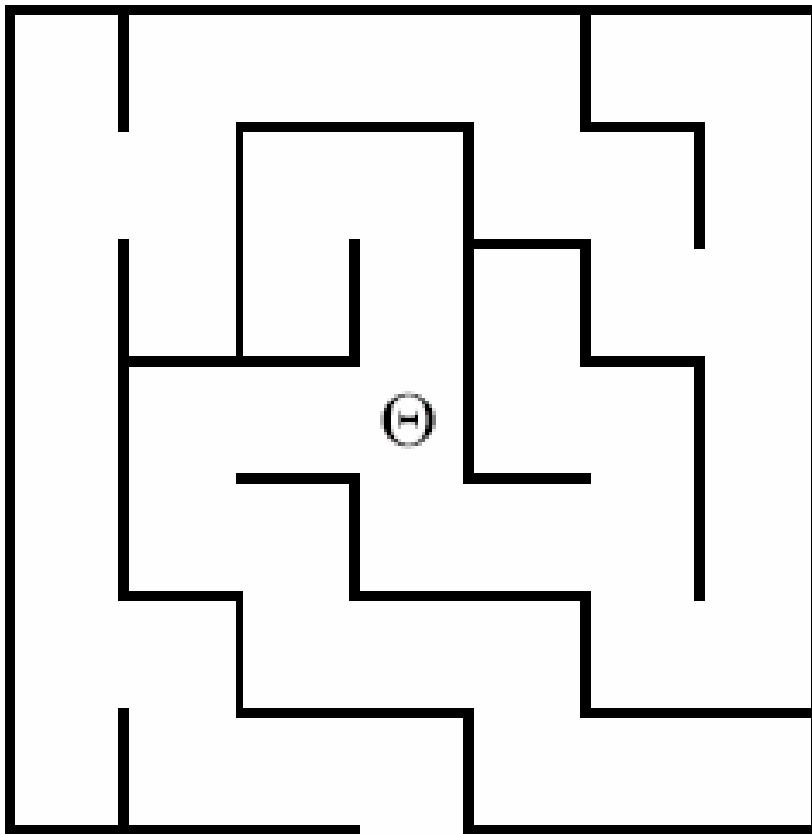
Este mosaico representa un laberinto, y está en la Catedral de Chartres. Antes de estar allí, ya se conocía en Creta mil años antes.

También es conocido en otras culturas.

**Un laberinto puede modelarse como una serie de nodos.** En cada nodo hay que tomar una decisión que nos conduce a otros nodos.

# Un laberinto sencillo

---



Buscar en el laberinto hasta encontrar una salida. Si no se encuentra una salida, informar de ello

# Algoritmo Backtracking Modificado

---

Si la posición actual está fuera, devolver TRUE para indicar que hemos encontrado una solución.

Si la posición actual está marcada, devolver FALSE para indicar que este camino ya ha sido explorado.

Marcar la posición actual.

**For** ( cada una de las 4 direcciones posibles )

{ **Si** ( Esta dirección no está bloqueada por un muro )

{ Moverse un paso en la dirección indicada desde la posición actual.

Intentar resolver el laberinto desde ahí haciendo una llamada recursiva.

Si esta llamada prueba que el laberinto es resoluble, devolver TRUE para indicar este hecho.

}

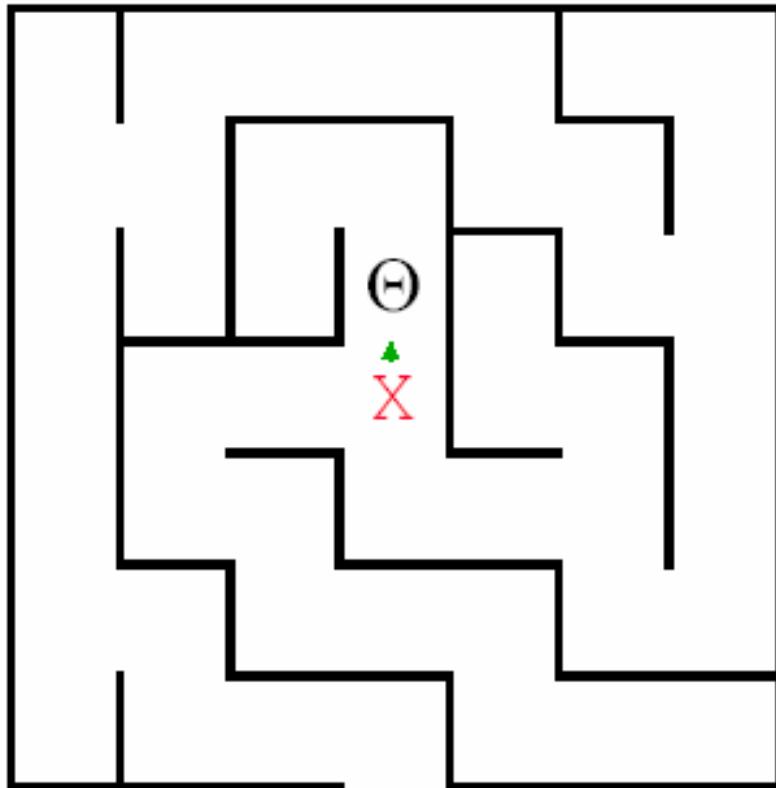
}

Quitar la marca a la posición actual.

Devolver FALSE para indicar que ninguna de las 4 direcciones lleva a una solución

# Backtracking en Acción

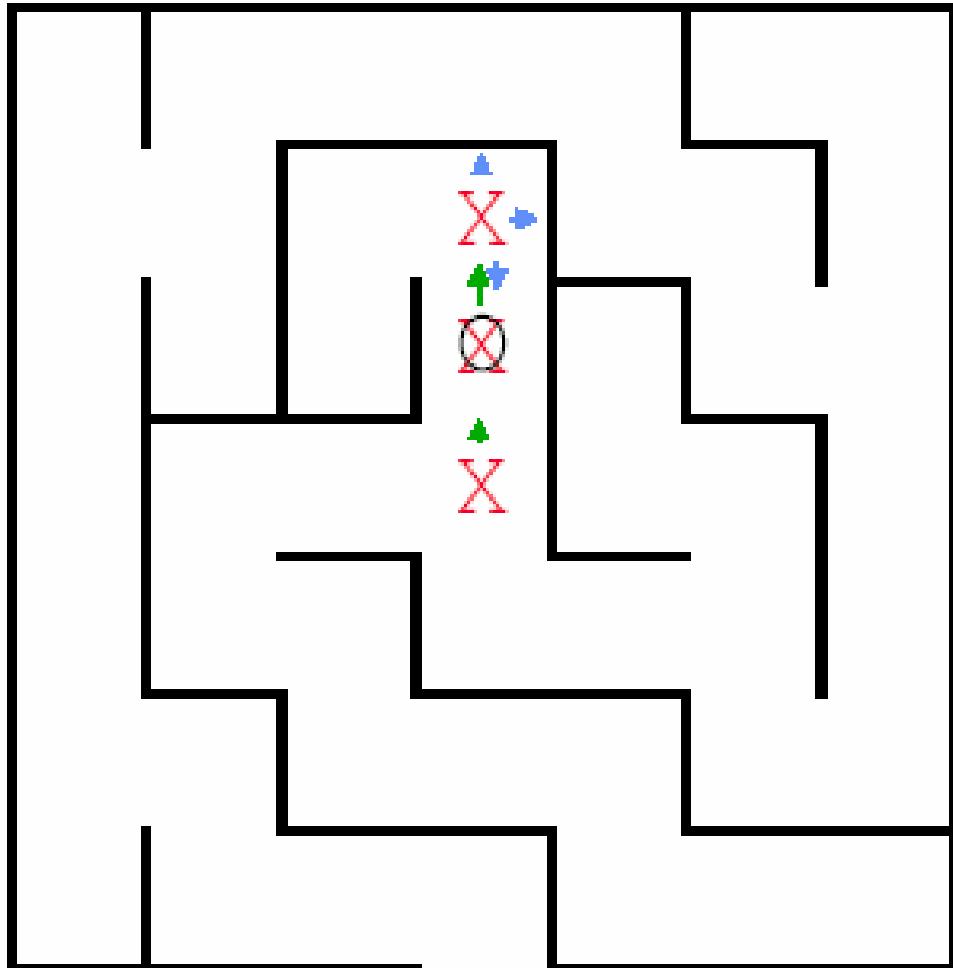
---



La parte crucial del algoritmo es el lazo FOR que nos lleva hacia las posibles alternativas que hay en un punto concreto. Aquí nos movemos hacia el norte.

# Backtracking en Acción

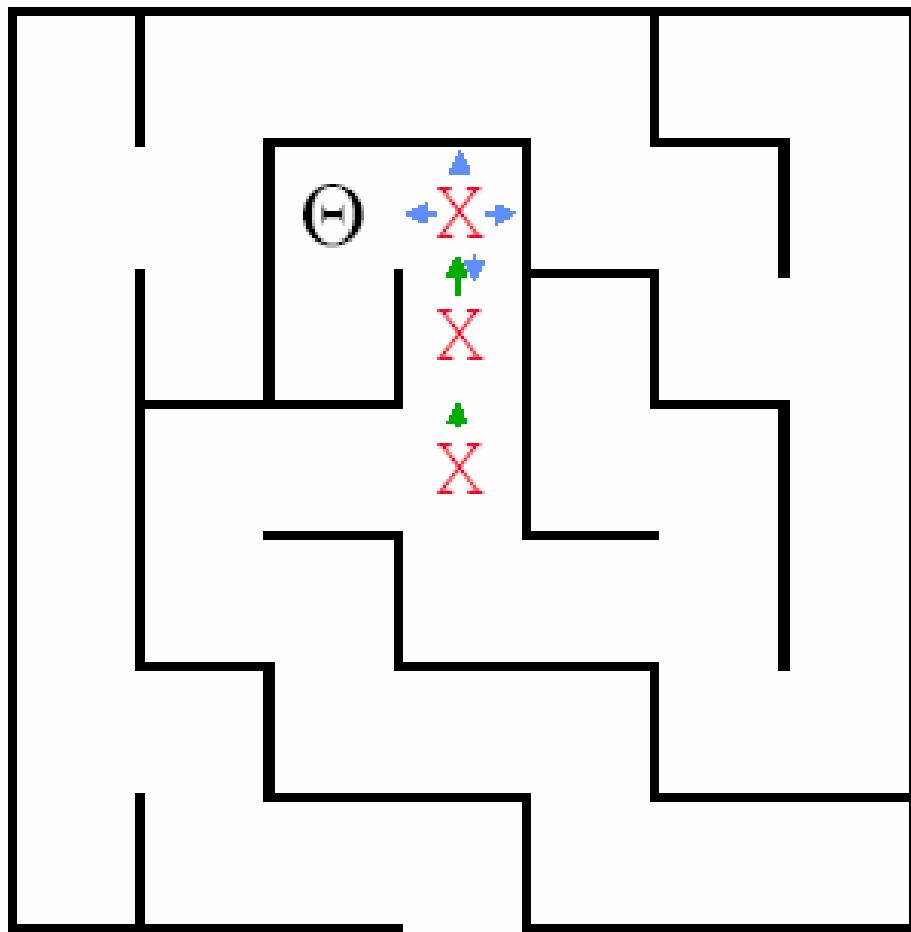
---



Aquí nos movemos hacia el Norte de nuevo, pero ahora la dirección Norte está bloqueada por un muro. El Este tambien está bloqueado, por lo que intentamos el Sur. Esa acción descubre que ese punto está marcado, de modo que volvemos atrás.

# Backtracking en Acción

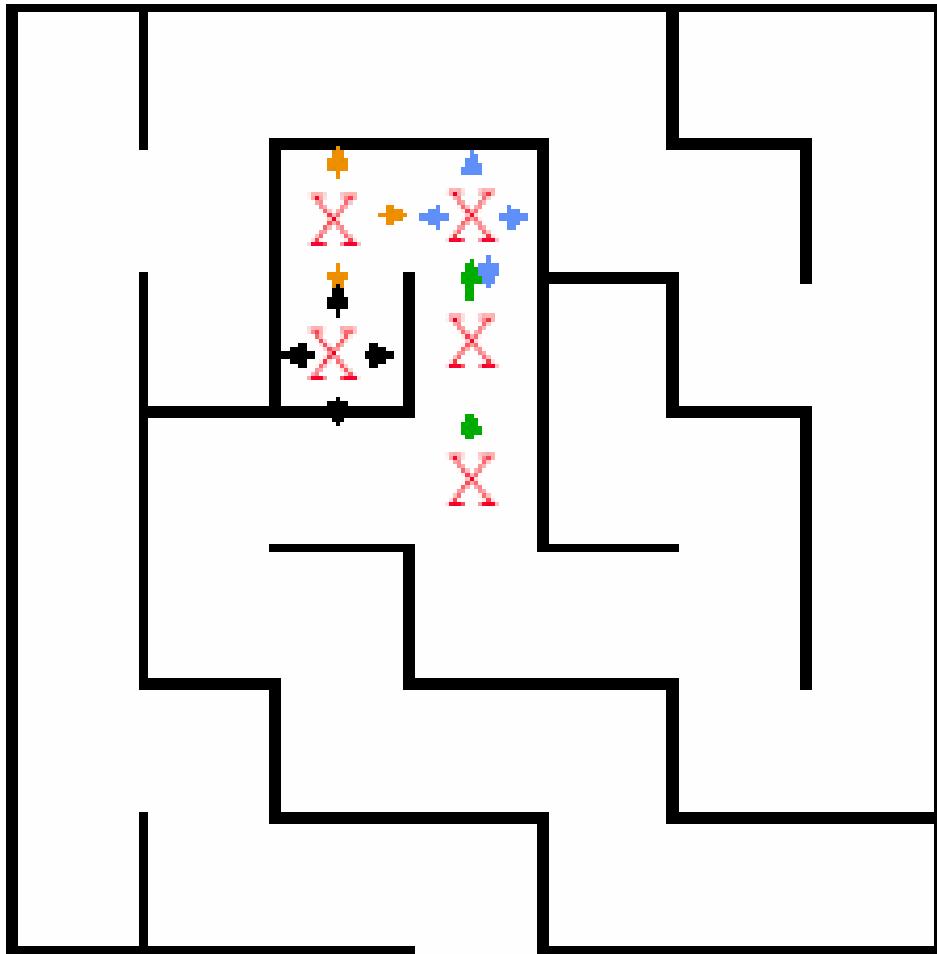
---



Por tanto,  
el siguiente  
movimiento que  
podemos hacer  
es hacia el Oeste

# Backtracking en Acción

---

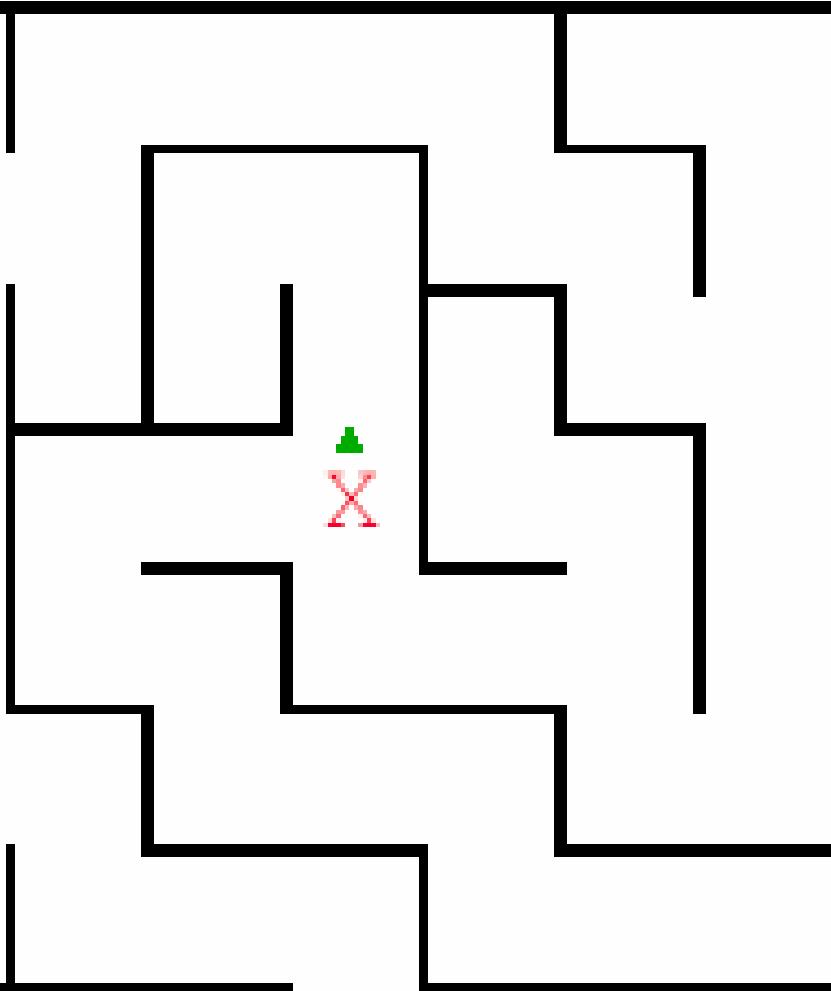


Este camino llega  
a un nodo (final)  
muerto

¡Por tanto, es el  
momento de hacer  
un backtrack!

# Backtracking en Acción

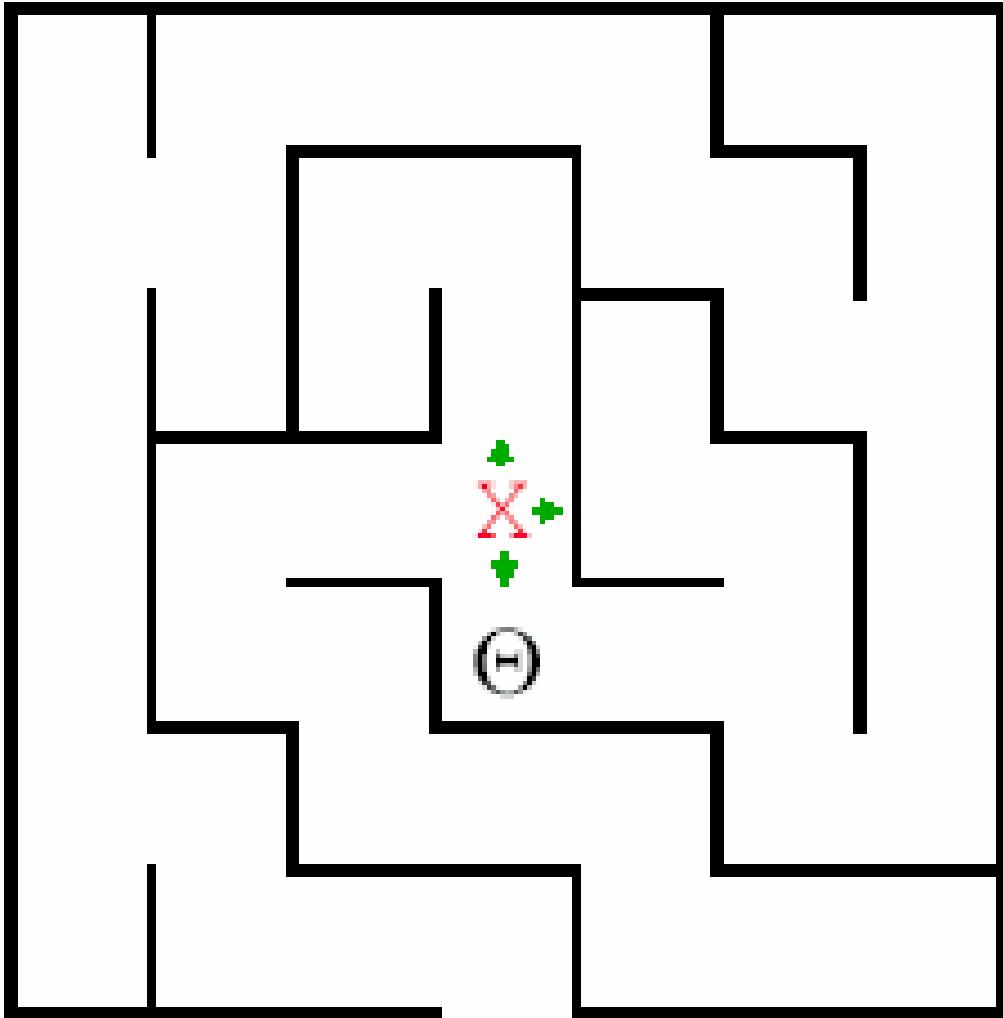
---



Se realizan sucesivas vueltas atrás hasta volvemos a encontrar aquí

# Backtracking en Acción

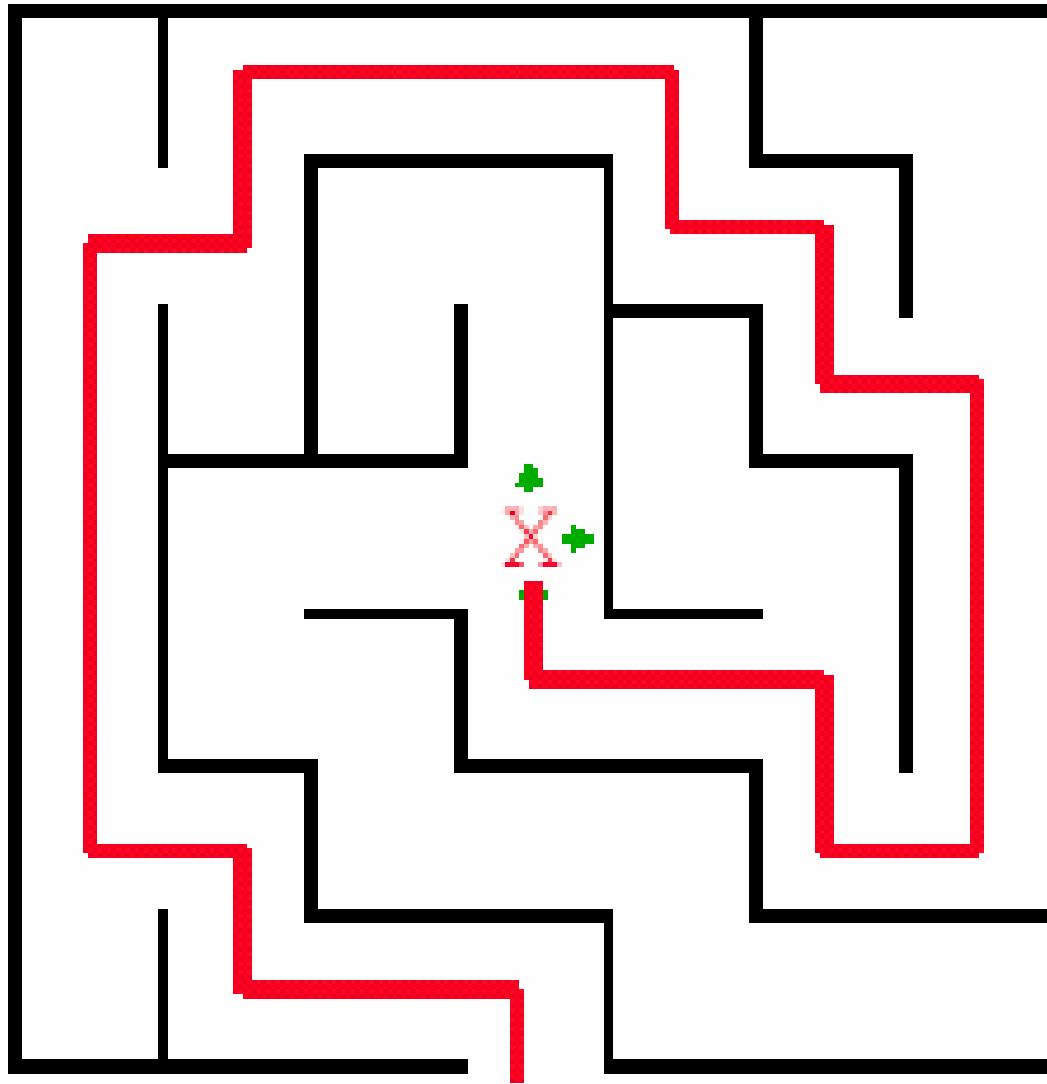
---



Intentamos  
ahora el Sur

# Primer camino que se encuentra

---



# Backtracking: Conclusiones

- **Backtracking:** Recorrido exhaustivo y sistemático en un árbol de soluciones.
- **Pasos para aplicarlo:**
  - Decidir la forma del árbol.
  - Establecer el esquema del algoritmo.
  - Diseñar las funciones genéricas del esquema.
- Relativamente fácil diseñar algoritmos que encuentren soluciones óptimas pero...
- Los algoritmos de backtracking son muy ineficientes.
- **Mejoras:** mejorar los mecanismos de poda, incluir otros tipos de recorridos (no solo en profundidad)  
→ **Técnica de Ramificación y Poda (Branch-Bound)**

# Índice

---

## **III. MÉTODOS BRANCH-BOUND**

- 1. Introducción: Diferencias con Backtracking**
- 2. Descripción general del método**
- 3. Estrategias de ramificación**
- 4. Procedimiento Branch-Bound**

# Branch and bound

---

- **Branch and Bound** es una técnica muy similar a la de Backtracking, y basa su diseño en el análisis del árbol de estados de un problema:
  - Realiza un recorrido sistemático de ese árbol
  - El recorrido no tiene que ser necesariamente en profundidad
- Generalmente se aplica para resolver problemas de Optimización y para jugar juegos

# Branch and bound

---

- Los algoritmos generados por esta técnica son normalmente de orden **exponencial** o **peor en su peor caso**, pero su aplicación en casos muy grandes, ha demostrado ser eficiente (incluso más que backtracking)
- Puede ser vista como una **generalización** (o mejora) de la técnica de **Backtracking**
- Tendremos una **estrategia de ramificación**
- Se tratará como un aspecto importante las **técnicas de poda**, para eliminar nodos que no lleven a soluciones optimas
- La poda se realiza **estimando** en cada nodo **cotas** del beneficio que podemos obtener a partir del mismo

# Branch and bound

---

- Diferencia fundamental con Backtracking:
  - En Backtracking tan pronto como se genera un nuevo hijo del nodo en curso, este hijo pasa a ser el nodo en curso
  - En BB se generan todos los hijos del nodo en curso antes de que cualquier otro nodo vivo pase a ser el nuevo nodo en curso (esta técnica no utiliza la búsqueda en profundidad)
- En consecuencia:
  - En Backtracking los únicos nodos vivos son los que están en el camino de la raíz al nodo en curso
  - En BB pueden haber más nodos vivos. Se almacenan en una estructura de datos auxiliar: **lista de nodos vivos**
- Además
  - En Backtracking el test de comprobación nos decía si era fracaso o no, mientras que en BB la cota nos sirve para podar el árbol y para saber el orden de ramificación, comenzando por las más prometedoras

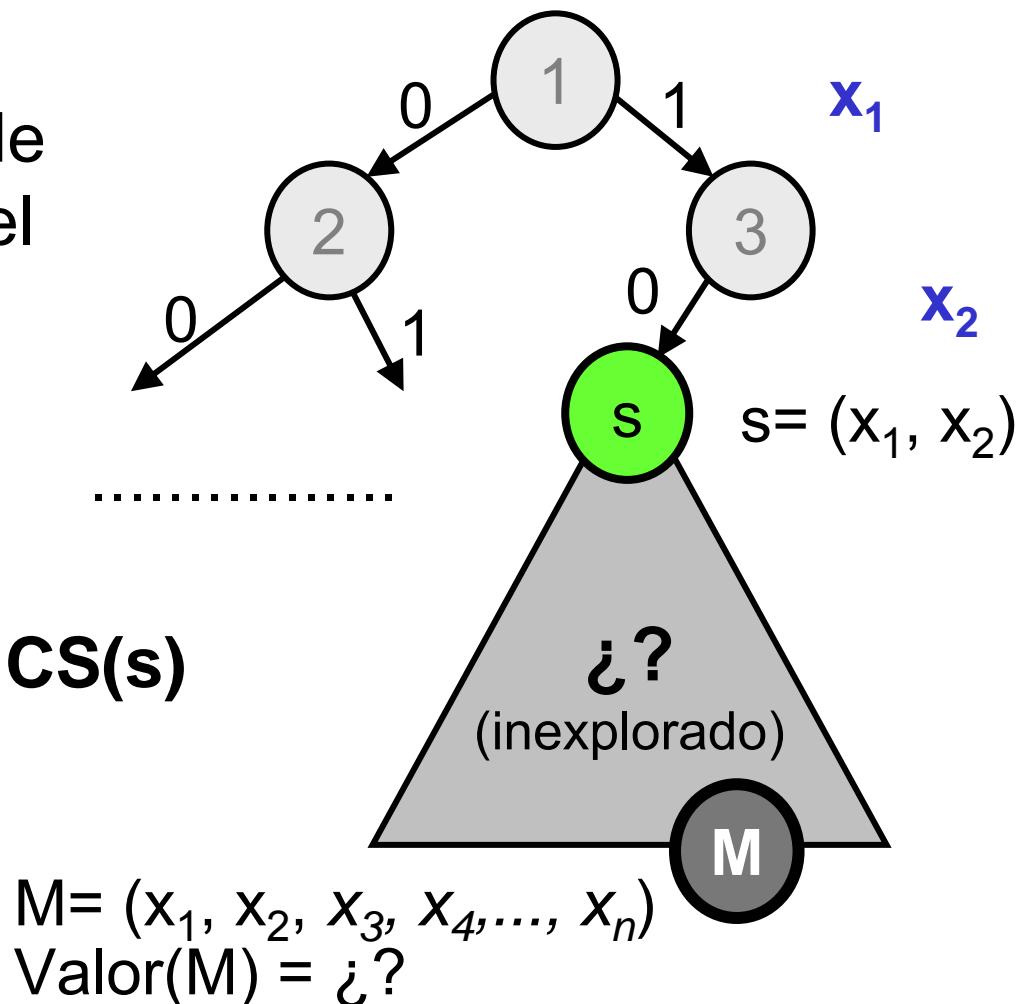
# Idea intuitiva del Branch and Bound

- Como decíamos anteriormente, la **ramificación y poda (branch and bound)** se suele utilizar en problemas de **optimización discreta** y en problemas de **juegos**.
- Puede ser vista como una **generalización** (o mejora) de la técnica de **backtracking**.
- **Similitud:**
  - Igual que backtracking, realiza un **recorrido sistemático** en un árbol de soluciones.
- **Diferencias:**
  - **Estrategia de ramificación:** el recorrido no tiene por qué ser necesariamente en profundidad.
  - **Estrategia de poda:** la poda se realiza **estimando** en cada nodo **cotas** del beneficio óptimo que podemos obtener a partir del mismo.

# Idea intuitiva del Branch and Bound

## Estimación de cotas a partir de una solución parcial

- **Problema:** antes de explorar  $s$ , acotar el beneficio de la mejor solución alcanzable,  $M$ .



## Idea intuitiva del Branch and Bound

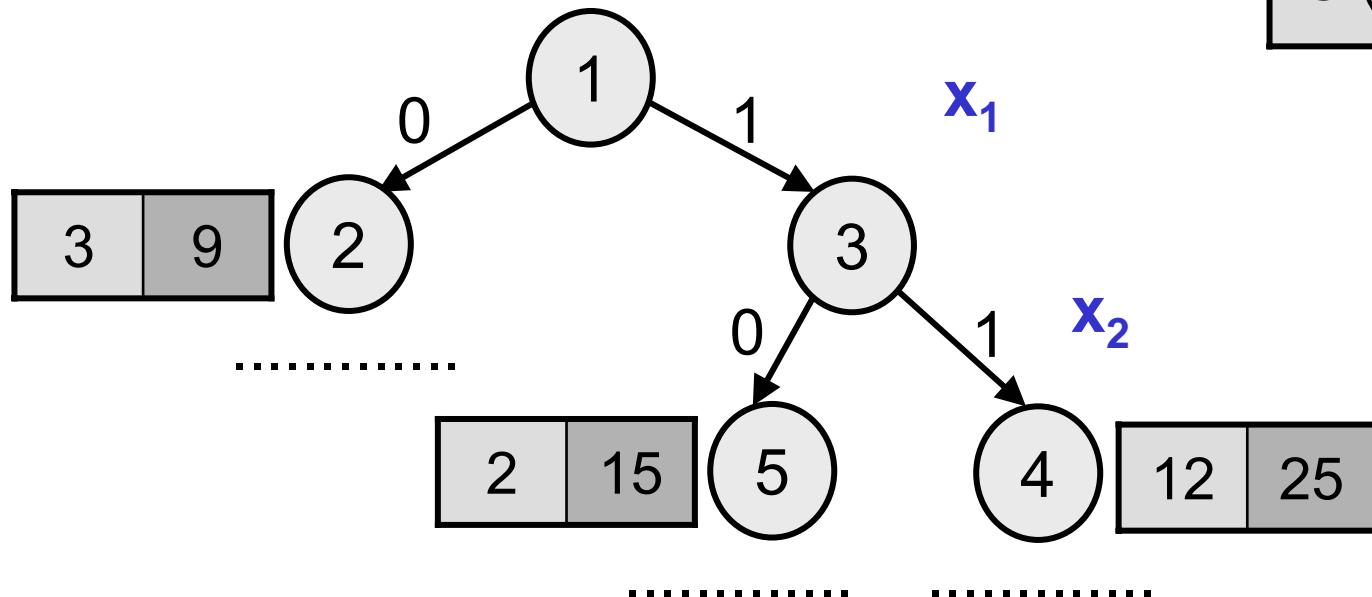
- Para cada nodo  $i$  tendremos:
  - **CS( $i$ )**: **Cota superior** del beneficio (o coste) óptimo que podemos alcanzar a partir del nodo  $i$ .
  - **CI( $i$ )**: **Cota inferior** del beneficio (o coste) óptimo que podemos alcanzar a partir del nodo  $i$ .
  - **BE( $i$ )**: **Beneficio estimado** (o coste) óptimo que se puede encontrar a partir del nodo  $i$ .
- Las cotas deben ser “fiables”: determinan cuándo se puede realizar una poda.
- El beneficio (o coste) estimado ayuda a decidir qué parte del árbol evaluar primero.

# Idea intuitiva del Branch and Bound

## Estrategia de poda

- Supongamos un problema de **maximización**.
- Hemos recorrido varios nodos, estimando para cada uno la cota superior **CS(j)** e inferior **CI(j)**.

|       |       |
|-------|-------|
| CI(j) | CS(j) |
|-------|-------|



- ¿Merece la pena seguir explorando por el nodo 2?
- ¿Y por el 5?

# Idea intuitiva del Branch and Bound

- **Estrategia de poda (maximización).** Podar un nodo  $i$  si se cumple que:
  - $CS(i) \leq CI(j)$ , para algún nodo  $j$  generado  
o bien
  - $CS(i) \leq Valor(s)$ , para algún nodo  $s$  solución final
- **Implementación.** Usar una variable de poda  $C$ :  
 $C = \max(\{CI(j) \mid \forall j \text{ generado}\}, \{Valor(s) \mid \forall s \text{ solución final}\})$ 
  - Podar  $i$  si:  $CS(i) \leq C$
- ¿Cómo sería para el caso de minimización?

# Idea intuitiva del Branch and Bound

## Estrategias de ramificación

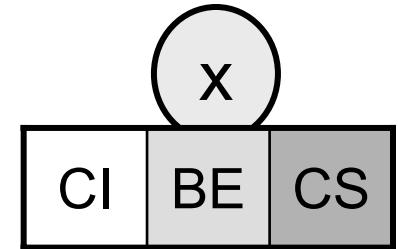
- Igual que en backtracking, hacemos un recorrido en un **árbol** de soluciones (que es **implícito**).
- **Distintos tipos de recorrido:** en profundidad, en anchura, según el beneficio estimado, etc.
- Para hacer los recorridos se utiliza una **lista de nodos vivos**.
- **Lista de nodos vivos (LNV):** contiene todos los nodos que han sido generados pero que no han sido explorados todavía. Son los nodos pendientes de tratar por el algoritmo.

# Idea intuitiva del Branch and Bound

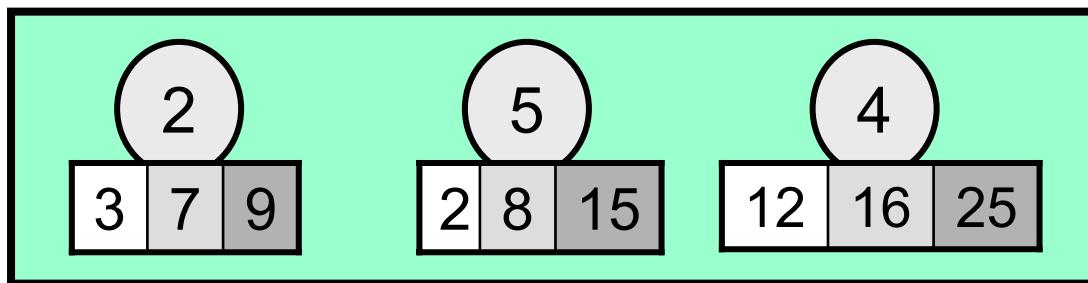
## Estrategias de ramificación

- **Idea básica del algoritmo:**

- Sacar un elemento de la lista LNV.
- Generar sus descendientes.
- Si no se podan, meterlos en la LNV.



LNV



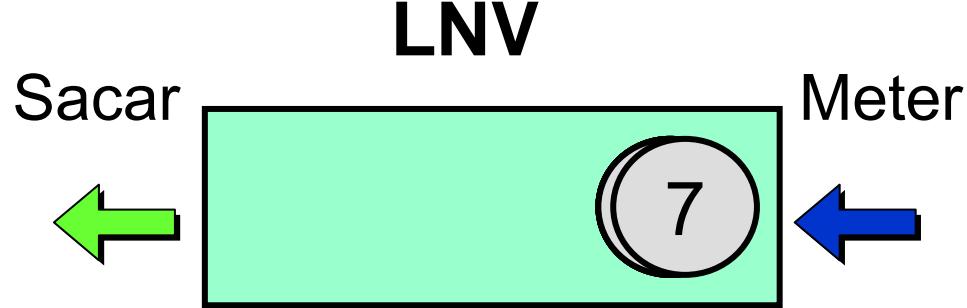
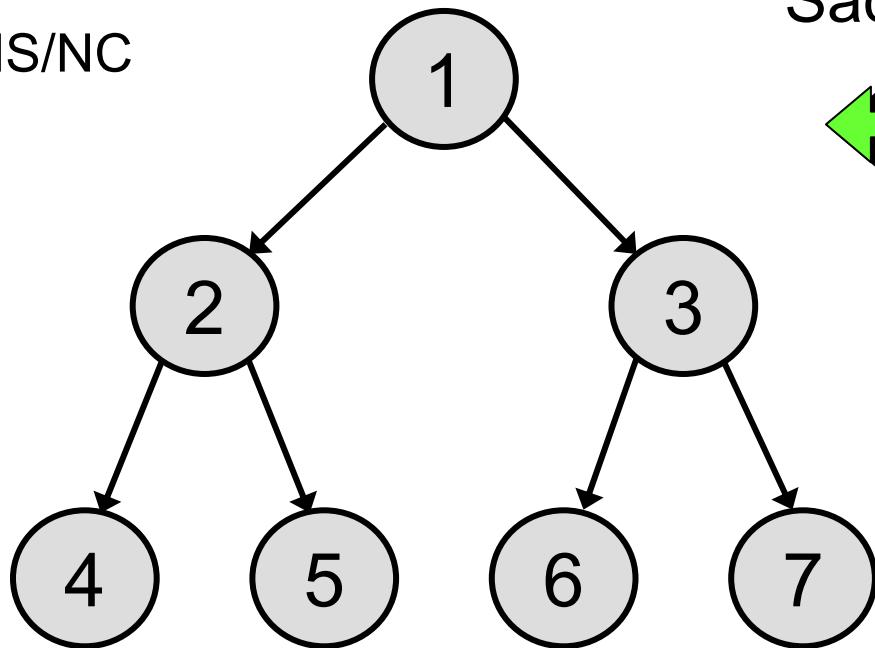
- ¿En qué orden se sacan y se meten?
- Según cómo se maneje esta lista, el recorrido será de uno u otro tipo.

# Idea intuitiva del Branch and Bound

## Estrategia de ramificación FIFO (First In First Out)

- Si se usa la estrategia FIFO, la LNV es una **cola** y el recorrido es:

- a) En profundidad
- b) En anchura
- c) NS/NC

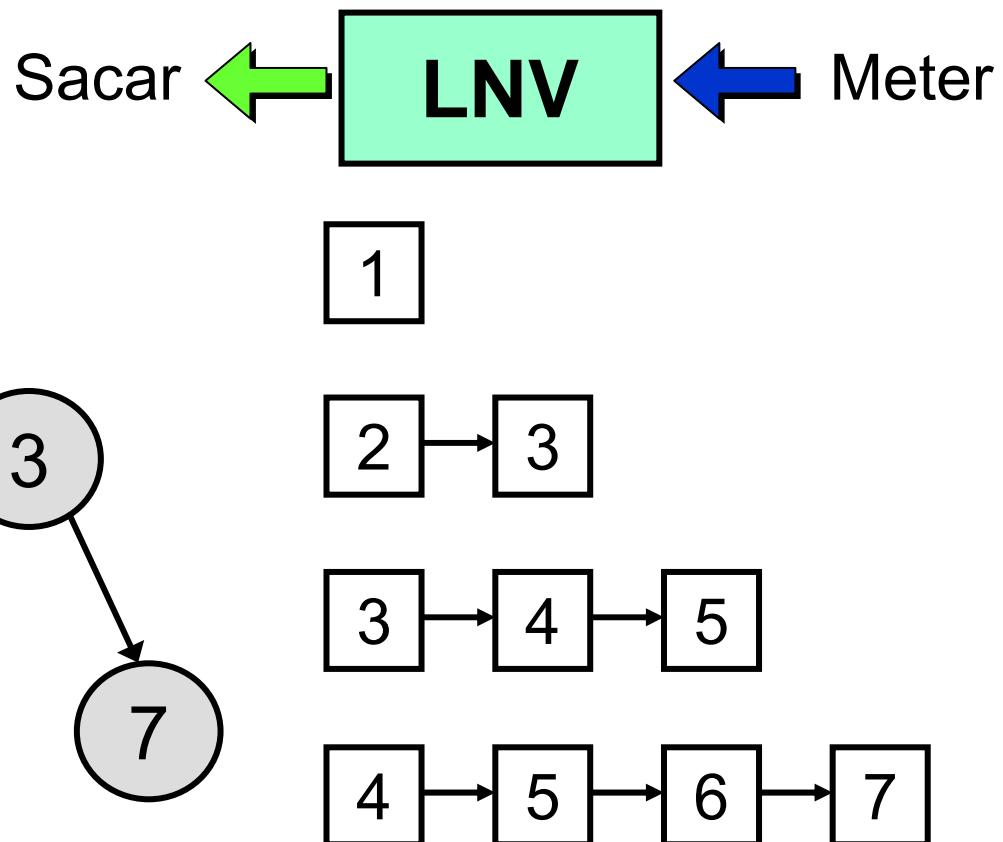
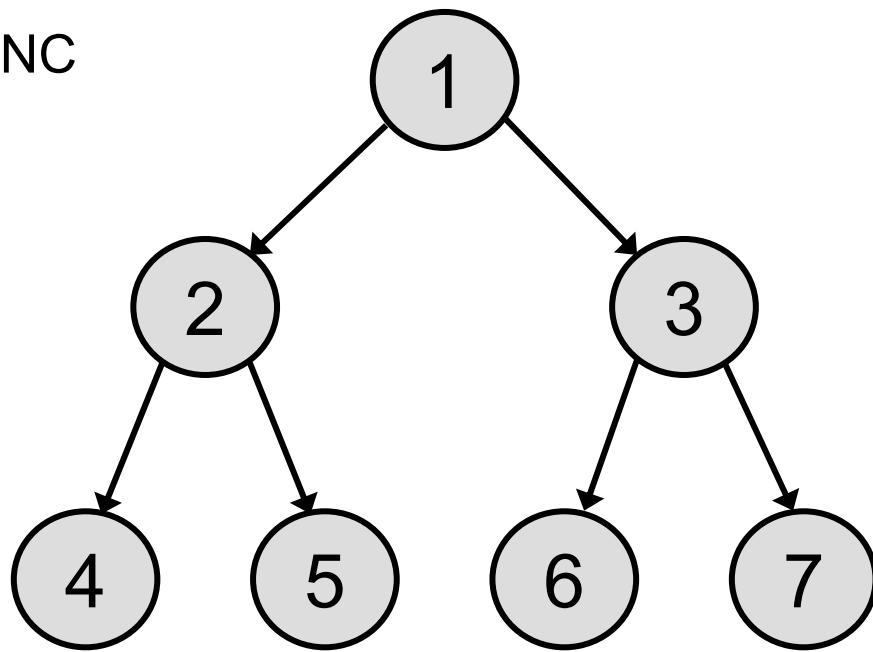


# Idea intuitiva del Branch and Bound

## Estrategia de ramificación FIFO (First In First Out)

- Si se usa la estrategia FIFO, la LNV es una **cola** y el recorrido es:

- a) En profundidad
- b) En anchura
- c) NS/NC

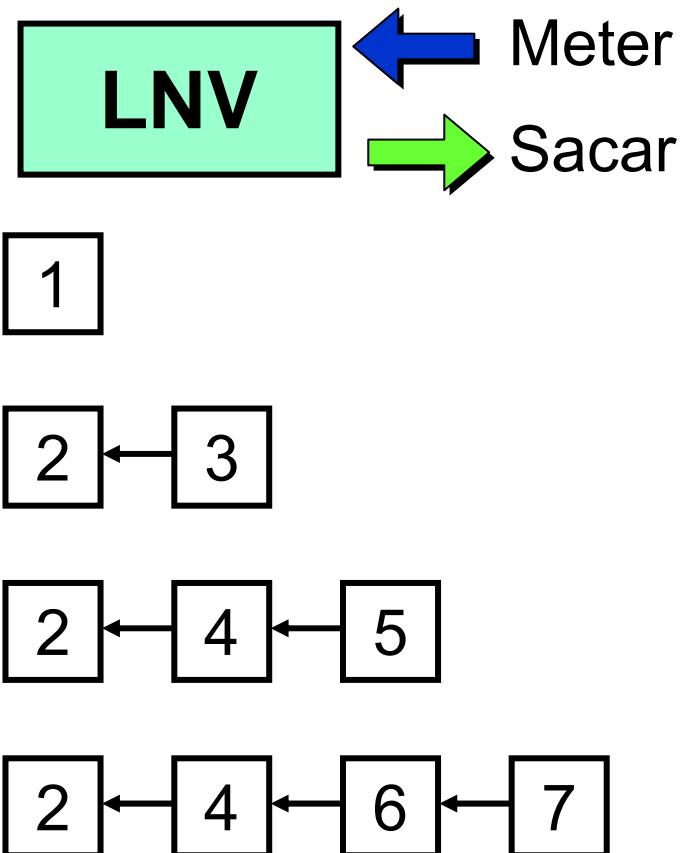
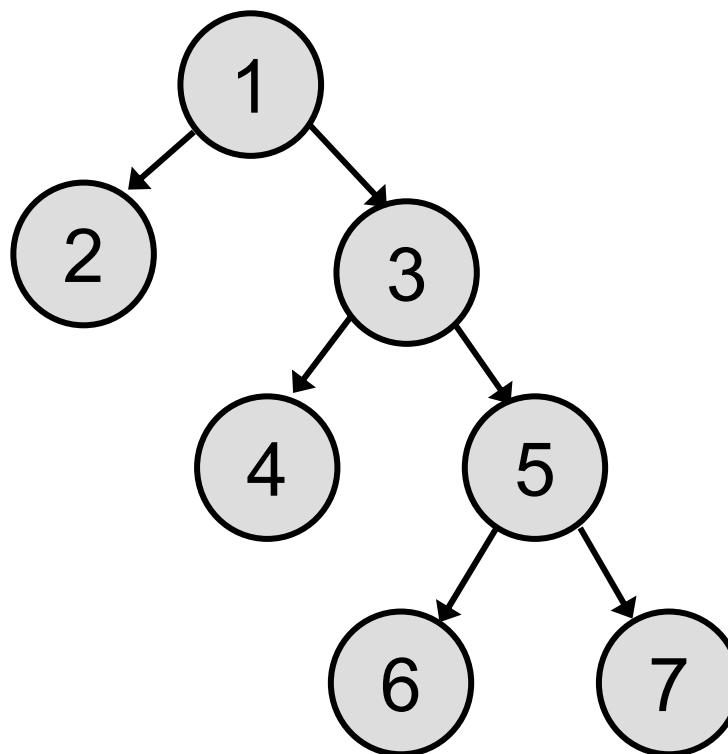


# Idea intuitiva del Branch and Bound

## Estrategia de ramificación LIFO (Last In First Out)

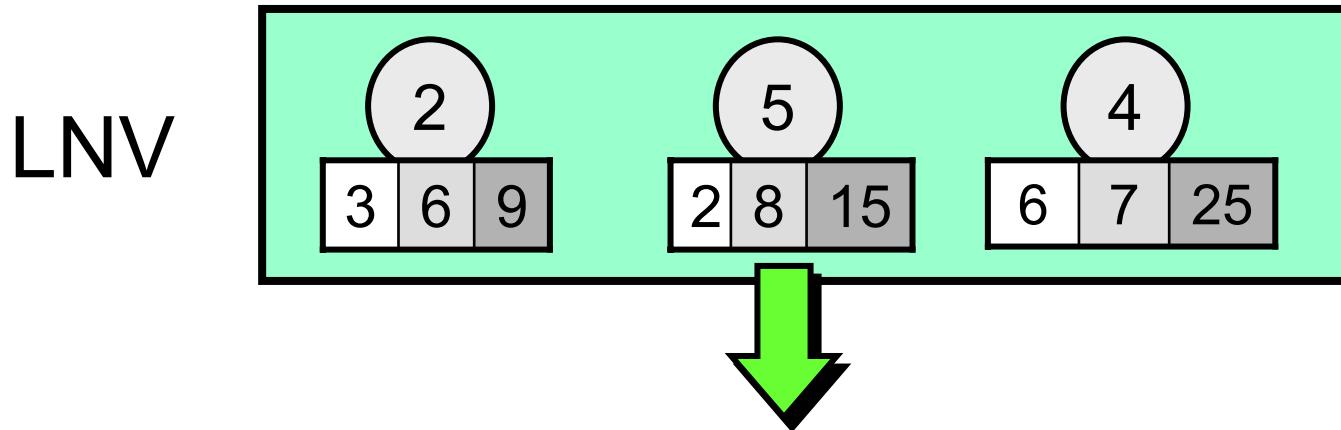
- Si se usa la estrategia LIFO, la LNV es una **pila** y el recorrido es:

- a) En profundidad
- b) En anchura
- c) NS/NC



# Idea intuitiva del Branch and Bound

- Las estrategias FIFO y LIFO realizan una búsqueda “a ciegas”, sin tener en cuenta los beneficios.
- **Usamos la estimación del beneficio:** explorar primero por los nodos con mayor valor estimado.
- **Estrategias LC (Least Cost):** Entre todos los nodos de la lista de nodos vivos, elegir el que tenga mayor beneficio (o menor coste) para explorar a continuación.



# Idea intuitiva del Branch and Bound

## Estrategias de ramificación LC

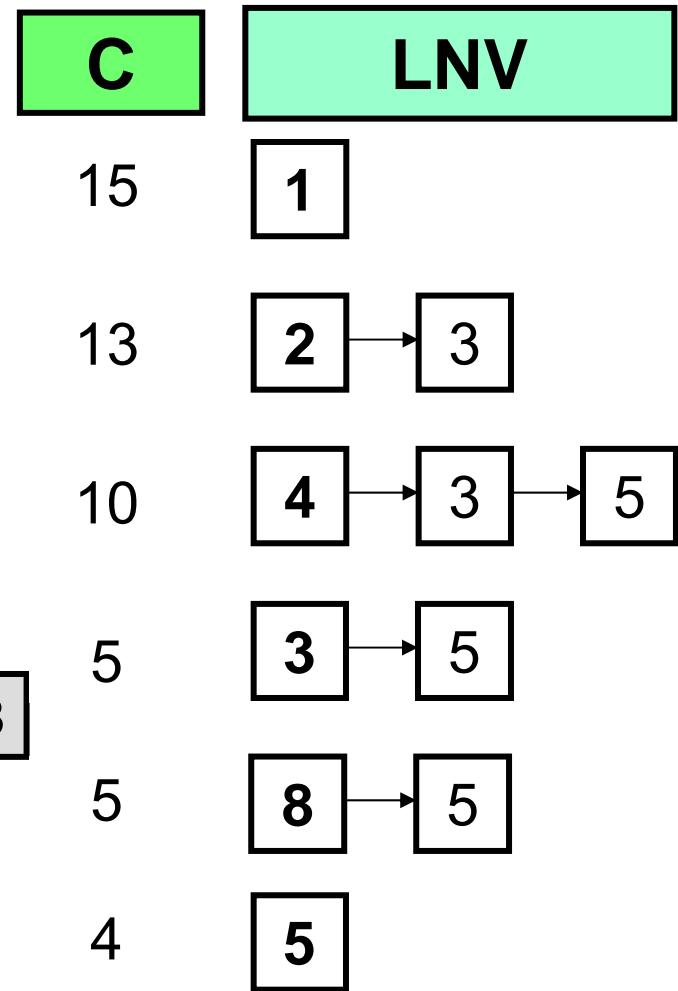
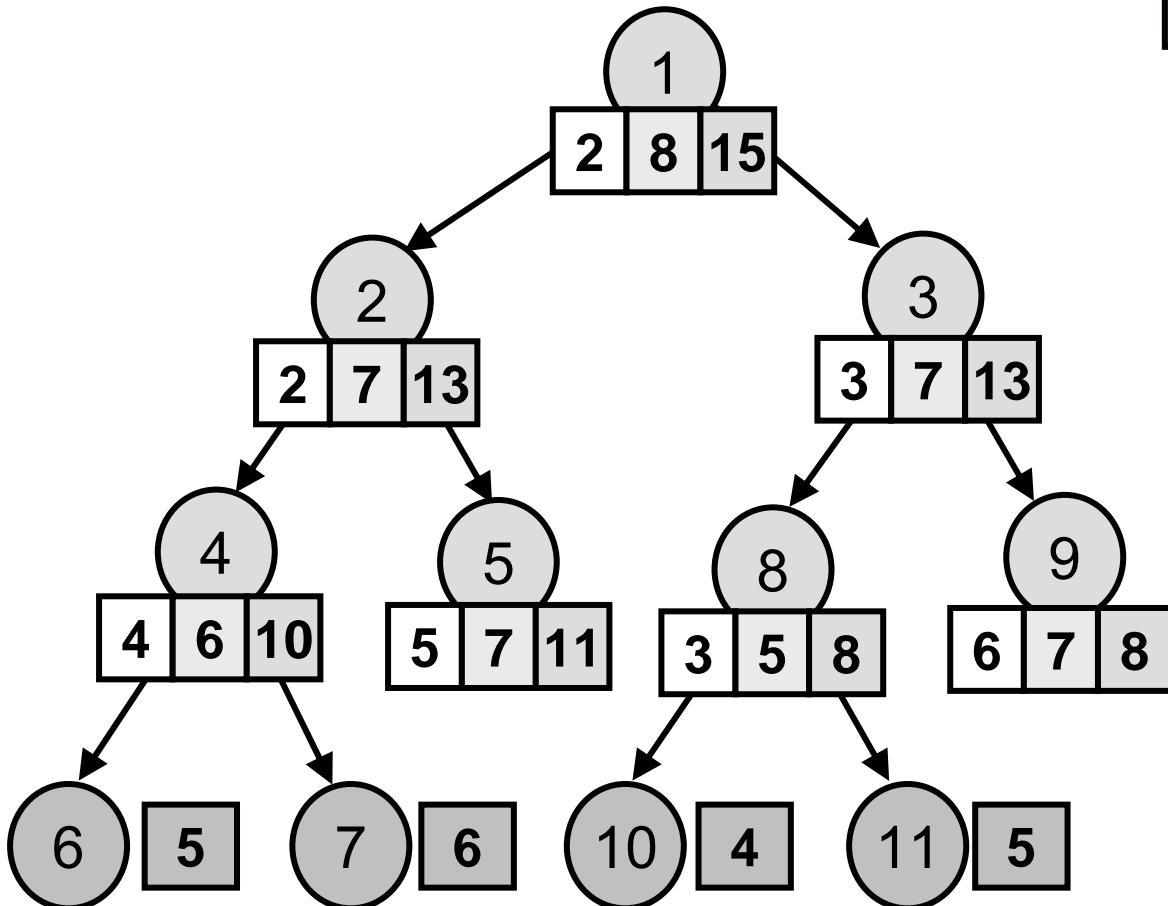
- En caso de empate (de beneficio o coste estimado) deshacerlo usando un criterio **FIFO** ó **LIFO**.
- **Estrategia LC-FIFO:** Seleccionar de LNV el nodo que tenga mayor beneficio y en caso de empate escoger el primero que se introdujo (de los que empatan).
- **Estrategia LC-LIFO:** Seleccionar el nodo que tenga mayor beneficio y en caso de empate escoger el último que se introdujo (de los que empatan).
- ¿Cuál es mejor?
- Se diferencian si hay muchos “empates” a beneficio estimado.

# Idea intuitiva del Branch and Bound

- **Resumen:**
  - En cada nodo  $i$  tenemos:  $CI(i)$ ,  $BE(i)$  y  $CS(i)$ .
  - **Podar** según los valores de  $CI$  y  $CS$ .
  - **Ramificar** según los valores de  $BE$ .
- **Ejemplo. Recorrido con ramificación y poda, usando LC-FIFO.**
  - Suponemos un problema de minimización.
  - Para realizar la poda usamos una variable  $C$  = valor de la menor de las cotas superiores hasta ese momento, o de alguna solución final.
  - Si para algún nodo  $i$ ,  $CI(i) \geq C$ , entonces podar  $i$ .

# Idea intuitiva del Branch and Bound

- Ejemplo. Recorrido con ramificación y poda, usando LC-FIFO.



# Idea intuitiva del Branch and Bound

- **Esquema algorítmico de ramificación y poda.**
  - **Inicialización:** Meter la raíz en la LNV, e inicializar la variable de poda **C** de forma conveniente.
  - **Repetir** mientras no se vacíe la LNV:
    - **Sacar un nodo** de la LNV, según la estrategia de ramificación.
    - Comprobar si debe ser podado, según la **estrategia de poda**.
    - En caso contrario, **generar sus hijos**. Para cada uno:
      - Comprobar si es una **solución final** y tratarla.
      - Comprobar si debe ser **podado**.
      - En caso contrario, **meterlo en la LNV** y **actualizar C** de forma adecuada.

# Idea intuitiva del Branch and Bound

**RamificacionYPoda (raiz: Nodo; var s: Nodo) // Minimización**

LNV:= {raiz}

C:= CS(raiz)

s:=  $\emptyset$

mientras LNV  $\neq \emptyset$  hacer

x:= **Seleccionar(LNV)** // Estrategia de ramificación

LNV:= LNV - {x}

si **CI(x) < C** entonces // Estrategia de poda

para cada y hijo de x hacer

si **Solución(y) AND (Valor(y)<Valor(s))** entonces

s:= y

C:= min (C, **Valor(y)**)

sino si NO **Solución(y) AND (CI(y) < C)** entonces

LNV:= LNV + {y}

C:= min (C, **CS(y)**)

finsi

finpara

finmientras

# Idea intuitiva del Branch and Bound

- **Funciones genéricas:**

- **CI(i), CS(i), CE(i).** Cota inferior, superior y coste estimado, respectivamente.
- **Solución(x).** Determina si  $x$  es una solución final válida.
- **Valor(x).** Valor de una solución final.
- **Seleccionar(LNV): Nodo.** Extrae un nodo de la LNV según la estrategia de ramificación.
- **para cada y hijo de x hacer.** Iterador para generar todos los descendientes de un nodo. Equivalente a las funciones de backtracking.

$y := x$

**mientras** MasHermanos(nivel( $x$ )+1,  $y$ ) **hacer**  
    Generar(nivel( $x$ )+1,  $y$ )  
    **si** Criterio( $y$ ) **entonces** ...

# Idea intuitiva del Branch and Bound

## Algunas cuestiones

- Se comprueba el criterio de poda al meter un nodo y al sacarlo. ¿Por qué esta duplicación?
- ¿Cómo actualizar **C** si el problema es de maximizar? ¿Y cómo es la poda?
- ¿Qué información se almacena en la LNV?

**LNV: Lista[Nodo]**

**tipo**

**Nodo = registro**

tupla: TipoTupla // P.ej. array [1..n] de entero

nivel: entero

CI, CE, CS: real

**finregistro**



Almacenar para no  
recalcular. ¿Todos?

# Idea intuitiva del Branch and Bound

- ¿Qué pasa si para un nodo  $i$  tenemos que  $CI(i)=CS(i)$ ?
- ¿Cómo calcular las cotas?
- ¿Qué pasa con las cotas si a partir de un nodo puede que no exista ninguna solución válida (factible)?
- Estas y otras cuestiones las trataremos de forma sistemática a continuación

# Descripción General del Método

---

BB ES UN MÉTODO DE BÚSQUEDA GENERAL QUE SE APLICA CONFORME A LO SIGUIENTE:

- Explora un árbol **comenzando a partir de un problema raiz** (el problema original con su región factible completa)
- Entonces se aplican procedimientos de **cotas inferiores y superiores** al problema raiz
- Si las cotas cumplen las condiciones que se hayan establecido, habremos encontrado la solución óptima y el procedimiento termina

# Descripción General del Método

---

- Si se encuentra una **solución óptima para un subproblema**, será una **solución factible para el problema completo**, pero no necesariamente el óptimo global
- Cuando en un nodo (subproblema) la **cota local es peor que el mejor valor conocido** en la región, **no puede existir un óptimo global** en el subespacio de la región factible asociada a ese nodo, y por tanto ese nodo puede ser eliminado en posteriores consideraciones
- La búsqueda sigue hasta que se examinan o “podan” todos los nodos, o hasta que se alcanza algún criterio pre-establecido sobre el mejor valor encontrado y las cotas locales de los subproblemas no resueltos

# Estimadores y cotas en Branch and bound

---

- **Cota local:** Se calcula de forma local para cada nodo  $i$ . Siendo  $LOptimo(i)$  el coste/beneficio de la mejor solución que se podría alcanzar al expandir el nodo  $i$ , la cota local es una estimación de dicho valor que debe ser mejor o igual a  $LOptimo(i)$ . Cuanto más cercana sea de  $LOptimo(i)$  mejor será la cota y por lo tanto más se podará, pero debe haber un equilibrio entre eficiencia de cómputo y calidad de la cota.

*(SE PUEDE ASEGURAR QUE NO SE ALCANZARÁ NADA MEJOR AL EXPANDIR  $i$ )*

- **Cota global:** Es el valor de la mejor solución estudiada hasta el momento (o una estimación del óptimo global) y debe ser peor o igual al coste/beneficio de la solución óptima. Inicialmente se le puede asignar el valor dado por un algoritmo Greedy, o en su defecto el peor valor posible. Se actualiza siempre que alcancemos una solución (nodo hoja) con mejor resultado. Cuanto más cercana sea al coste/beneficio del óptimo más se podará, por lo que es importante encontrar cuanto antes soluciones buenas.

*(SE PUEDE ASEGURAR QUE EL ÓPTIMO NUNCA SERÁ PEOR QUE ESTA COTA)*

# Estimadores y cotas en Branch and bound

---

- **Estimador del coste/beneficio local óptimo:** Se calcula para cada nodo  $i$  y sirve para determinar el siguiente nodo a expandir. Es un estimador de  $L_{Optimo}(i)$  como la cota local, pero no tiene por qué ser mejor o igual que  $L_{Optimo}(i)$ . Normalmente se utiliza la cota local como estimador, pero en el caso de que se pueda definir una medida más cercana a  $L_{Optimo}(i)$  sin importar si es mejor o peor que  $L_{Optimo}(i)$  podría interesar utilizar esta medida para decidir el siguiente nodo a expandir.
- **Estrategia de poda:** Además de podar aquellos nodos que no cumplen las restricciones implícitas (soluciones parciales no factibles) se podrán podar aquellos nodos  $i$  cuya cota local sea peor o igual que la cota global. Si sé que lo mejor que se puede alcanzar al expandir  $i$  no puede mejorar a una solución que ya se ha obtenido o se va a obtener, no es necesario expandir dicho nodo. Por la forma en la que están definidas la cota local y global se puede asegurar que no se perderá ninguna solución óptima, ya que si se cumple que,
  - [ $CotaLocal(i)$  mejor o igual que  $L_{Optimo}(i)$ ] y [ $CotaGlobal$  peor o igual que Óptimo],
  - entonces  $L_{Optimo}(i)$  tiene que ser peor que Óptimo cuando [ $CotaLocal(i)$  peor que  $CotaGlobal$ ].

# Estimadores y cotas en Branch and bound

---

Particularizando para problemas de minimización o maximización tenemos:

## ■ Minimización:

- La cota local es una cota inferior  $CI(i)$  del mejor coste que se puede conseguir al expandir el nodo  $i$ , y se debe cumplir:  $CI(i) \leq LOptimo(i)$ .
- La cota global es una cota superior  $CS$  del coste del óptimo global, y se debe cumplir:  $CS \geq \text{Óptimo}$ .
- En este caso se puede podar un nodo  $i$  cuando  $CI(i) > CS$ .

## ■ Maximización:

- La cota local es una cota superior  $CS(i)$  del máximo beneficio que se puede conseguir al expandir el nodo  $i$ , y se debe cumplir:

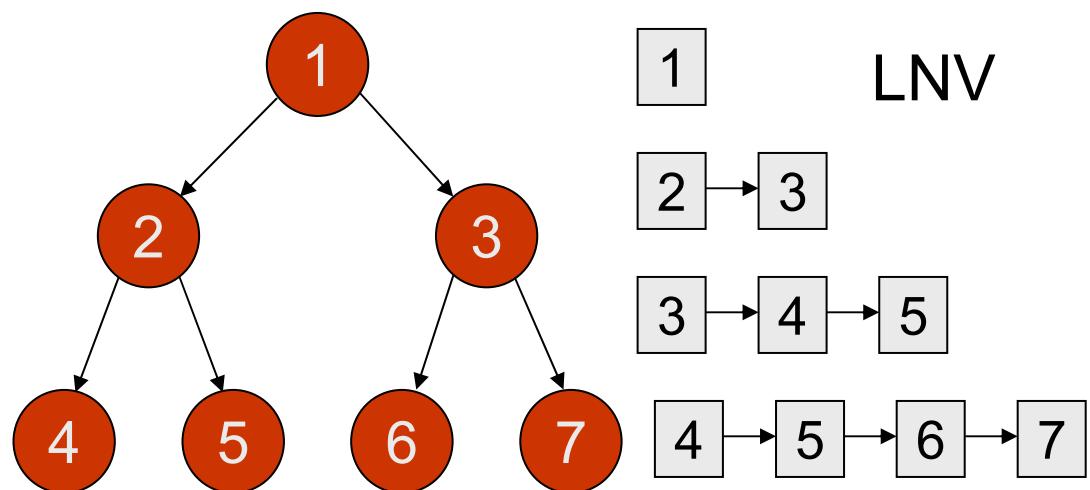
$$CS(i) \geq LOptimo(i).$$

- La cota global es una cota inferior  $CI$  del beneficio del óptimo global, y se debe cumplir:  $CI \leq \text{Óptimo}$ .
- En este caso se puede podar un nodo  $i$  cuando  $CS(i) < CI$ .

# Estrategias de ramificación

- Normalmente el árbol de soluciones es implícito, no se almacena en ningún lugar
- Para hacer el recorrido se utiliza una **lista de nodos vivos**
- **Lista de nodos vivos:** contiene todos los nodos que han sido generados pero que **no han sido explorados todavía**. Son los nodos pendientes de tratar por el algoritmo
- Según cómo sea la lista, el recorrido será de uno u otro tipo.

**ESTRATEGIA FIFO  
(First In First Out)**  
La lista de nodos vivos  
es una cola  
El recorrido es en anchura

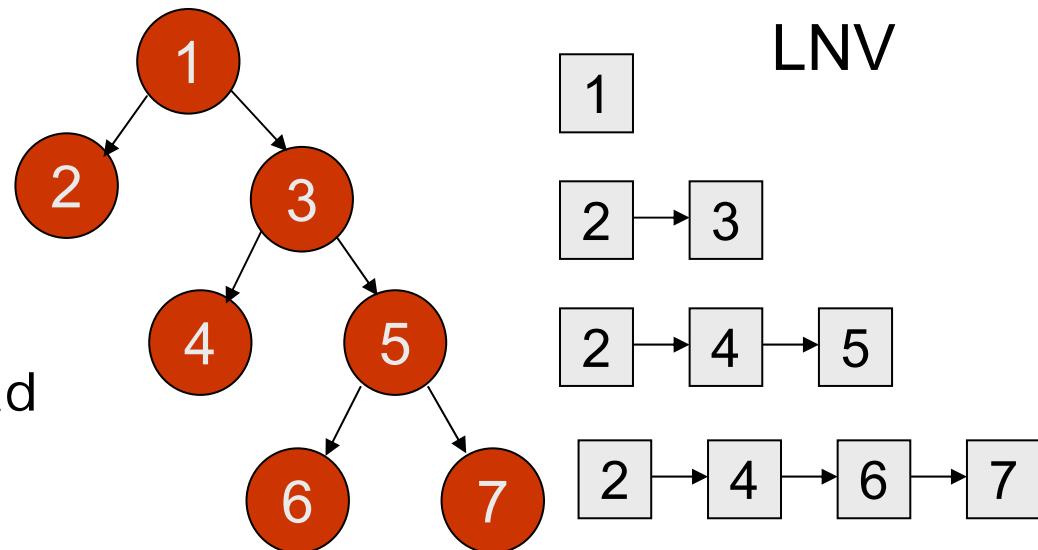


# Estrategias de ramificación

## ESTRATEGIA LIFO (Last In First Out)

La lista de nodos vivos es una pila

El recorrido es en profundidad



- Las estrategias FIFO y LIFO realizan una búsqueda “a ciegas”, sin tener en cuenta los beneficios
- Usando la estimación del beneficio, entonces será mejor buscar primero por los nodos con mayor valor estimado
- **ESTRATEGIAS LC (Least Cost):**
  - Entre todos los nodos de la lista de nodos vivos, elegir el que tenga mayor beneficio (o menor coste) para explorar a continuación

# Estrategias de ramificación LC

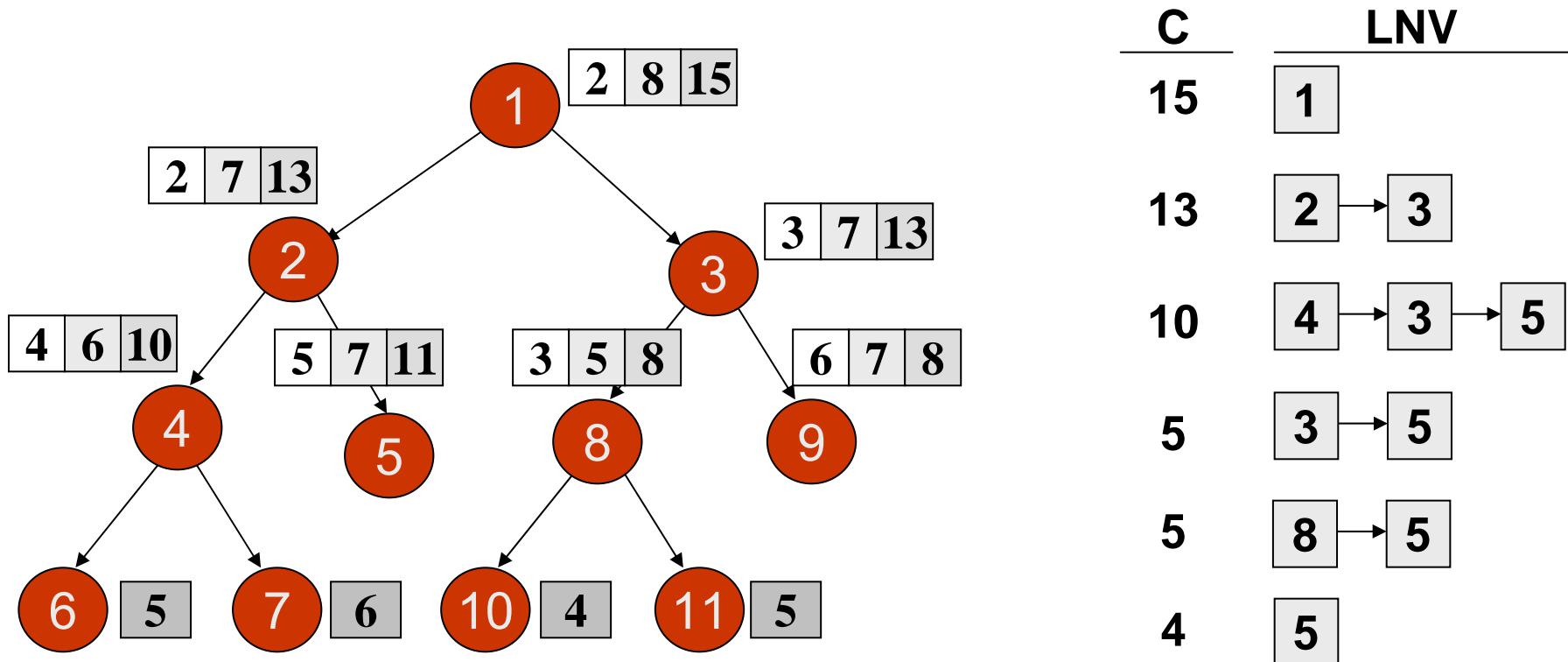
---

- En caso de empate (de beneficio o coste estimado) deshacerlo usando un criterio **FIFO** ó **LIFO**:
  - **Estrategia LC-FIFO:** Seleccionar de la LNV el que tenga mayor beneficio y en caso de empate escoger el primero que se introdujo (de los que empatan)
  - **Estrategia LC-LIFO:** Seleccionar de la LNV el que tenga mayor beneficio y en caso de empate escoger el último que se introdujo (de los que empatan)
- En cada nodo podemos tener: cota inferior de coste, coste estimado y cota superior del coste
- Podar según los valores de las cotas
- Ramificar según los costes estimados

# BB: El Método General

## Ejemplo. Branch and Bound usando LC-FIFO:

- Supongamos un **problema de minimización**, y que tenemos el **caso 1** (a partir de un nodo siempre existe alguna solución)
- Para realizar la poda usaremos una variable  **$C = \text{valor de la menor de las cotas superiores hasta ese momento}$**  (o de alguna solución final)
- Si para algún nodo  $i$ ,  $CI(i) \geq C$ , entonces podar  $i$



# BB: El Método General

---

## Algunas cuestiones

- Sólo se comprueba el criterio de poda cuando se introduce o se saca un nodo de la lista de nodos vivos
- Si un descendiente de un nodo es una solución final entonces no se introduce en la lista de nodos vivos. Se comprueba si esa solución es mejor que la actual, y se actualiza  $C$  y el valor de la mejor solución óptima de forma adecuada
- ¿Qué pasa si a partir de un nodo solución pueden haber otras soluciones?
- ¿Cómo debe ser actualizada la variable  $C$  (variable de poda) si el problema es de maximización?
- ¿Cómo será la poda, para cada uno de los casos anteriores?
- ¿Cuándo acaba el algoritmo?
- ¿Cómo calcular las cotas?

# BB: El Método General

- **Esquema general.** Problema de minimización, suponiendo el caso en que existe solución a partir de cualquier nodo

```

RamificacionYPoda (NodoRaiz: tipo_nodo; var s: tipo_solucion);
    LNV:= {NodoRaiz};
    C, s := CS (NodoRaiz); {Primera solución y cota asociada}
    Mientras LNV ≠ ∅ hacer
        x:= Seleccionar (LNV); {Según un criterio FIFO, LIFO, LC-FIFO ó LC-LIFO}
        LNV:= LNV - {x};
        Si CI (x) <= C entonces { Si no se cumple se poda x }
            Para cada y hijo de x hacer
                Si y es una solución final mejor que s entonces
                    S:= y;
                    C:= Coste (y);
                    Sino si y no es solución final y (CI (y) <= C) entonces
                        LNV:= LNV + {y};
                        Cttmp, s_tmp := CS (y);
                        si (Cttmp < C)
                            C:= Cttmp;
                            s := s_tmp;
            FinPara;
        FinMientras;
    
```

# BB: Análisis de tiempos de ejecución

---

- El tiempo de ejecución depende de:
  - **Número de nodos recorridos**: depende de la efectividad de la poda
  - **Tiempo gastado en cada nodo**: tiempo de hacer las estimaciones de coste y tiempo de manejo de la lista de nodos vivos
- En el peor caso, el tiempo es igual que el de un algoritmo con backtracking (ó peor si tenemos en cuenta el tiempo que requiere la LNV)
- En el caso promedio se suelen obtener mejoras respecto a backtracking
- ¿Cómo hacer que un algoritmo BB sea más eficiente?
  - **Hacer estimaciones de costo muy precisas**: Se realiza una poda exhaustiva del árbol. Se recorren menos nodos pero se gasta mucho tiempo en realizar las estimaciones
  - **Hacer estimaciones de costo poco precisas**: Se gasta poco tiempo en cada nodo, pero el número de nodos puede ser muy elevado. No se hace mucha poda
- **Se debe buscar un equilibrio** entre la exactitud de las cotas y el tiempo en calcularlas

# Índice

---

## **IV. SOLUCIONES BRANCH-BOUND EN DISTINTOS PROBLEMAS**

- 1. El Problema de la Mochila 0/1**
- 2. El Problema del Viajante de Comercio**

# El Problema de la Mochila 0/1

---

## ■ Diseño del algoritmo BB:

- Definir una representación de la solución. A partir de un nodo, cómo se obtienen sus descendientes
- Dar una manera de calcular el valor de las cotas y la estimación del beneficio
- Definir la estrategia de ramificación y de poda

## ■ Representación de la solución:

- **Mediante un árbol binario:**  $(s_1, s_2, \dots, s_n)$ , con  $s_i = (0, 1)$   
Hijos de un nodo  $(s_1, s_2, \dots, s_k)$ :  
 $(s_1, \dots, s_k, 0)$  y  $(s_1, \dots, s_k, 1)$
- **Mediante un árbol combinatorio:**  $(s_1, s_2, \dots, s_m)$  donde  $m \leq n$  y  $s_i \in \{1, 2, \dots, n\}$   
Hijos de un nodo  $(s_1, \dots, s_k)$ :  
 $(s_1, \dots, s_k, s_k+1), (s_1, \dots, s_k, s_k+2), \dots, (s_1, \dots, s_k, n)$

# El Problema de la Mochila 0/1

---

## ■ Cálculo de cotas:

- **Cota inferior**: Beneficio que se obtendría sólo con los objetos incluidos hasta ese nodo
- **Estimación del beneficio**: A la solución actual, sumar el beneficio de incluir los objetos enteros que quepan, utilizando avance rápido. Suponemos que los objetos están ordenados por orden decreciente de  $v_i/w_i$ ,
- **Cota superior**: Resolver el problema de la mochila continuo a partir de ese nodo (con un algoritmo ~~greedy~~), y quedarse con la parte entera.

## ■ Ejemplo. $n = 4, M = 7, v = (2, 3, 4, 5), w = (1, 2, 3, 4)$

Nodo actual:  $(1, 1)$   $(1, 2)$

Hijos:  $(1, 1, 0), (1, 1, 1)$   $(1, 2, 3), (1, 2, 4)$

Cota inferior:  $CI = v_1 + v_2 = 2 + 3 = 5$

Estimación del beneficio:  $EB = CI + v_3 = 5 + 4 = 9$

Cota superior:  $CS = CI + \lfloor MochilaGreedy(3, 4) \rfloor = 5 + \lfloor 4 + 5/4 \rfloor = 10$

# El Problema de la Mochila 0/1

---

## ■ Forma de realizar la poda:

- En una variable  $C$  guardar el valor de la mayor cota inferior hasta ese momento dado
- Si para un nodo, su cota superior es menor o igual que  $C$  entonces se puede podar ese nodo

## ■ Estrategia de ramificación:

- Puesto que tenemos una estimación del coste, usar una estrategia  $LC$ : explorar primero las ramas con mayor valor esperado ( $MB$ )
- ¿ $LC$ -FIFO ó  $LC$ -LIFO? Usaremos la  $LC$ -LIFO: en caso de empate seguir por la rama más profunda ( $MB$ -LIFO)

# El Problema de la Mochila 0/1

---

```
Mochila01BB (v, w: array [1..n] of integer; M: integer; var s: nodo);
inic:= NodoInicial (v, w, M);
C:= inic.CI;
LNV:= {inic};
s.v_act:= -∞;
Mientras LNV ≠ ∅ hacer
    x:= Seleccionar (LNV); {Según el criterio MB-LIFO}
    LNV:= LNV - {x};
    Si x.CS > C Entonces {Si no se cumple se poda x}
        Para i:= 0, 1 Hacer
            y:= Generar (x, i, v, w, M);
            Si (y.nivel = n) Y (y.v_act > s.v_act) Entonces
                s:= y;
                C:= max (C, s.v_act );
            Sino Si (y.nivel < n) Y (y.CS > C) Entonces
                LNV:= LNV + {y};
                C:= max (C, y.CI );
        FinPara;
    FinSi;
FinMientras;
```

# El Problema de la Mochila 0/1

---

**NodoInicial (v, w: array [1..n] of integer; M: integer) : nodo;**

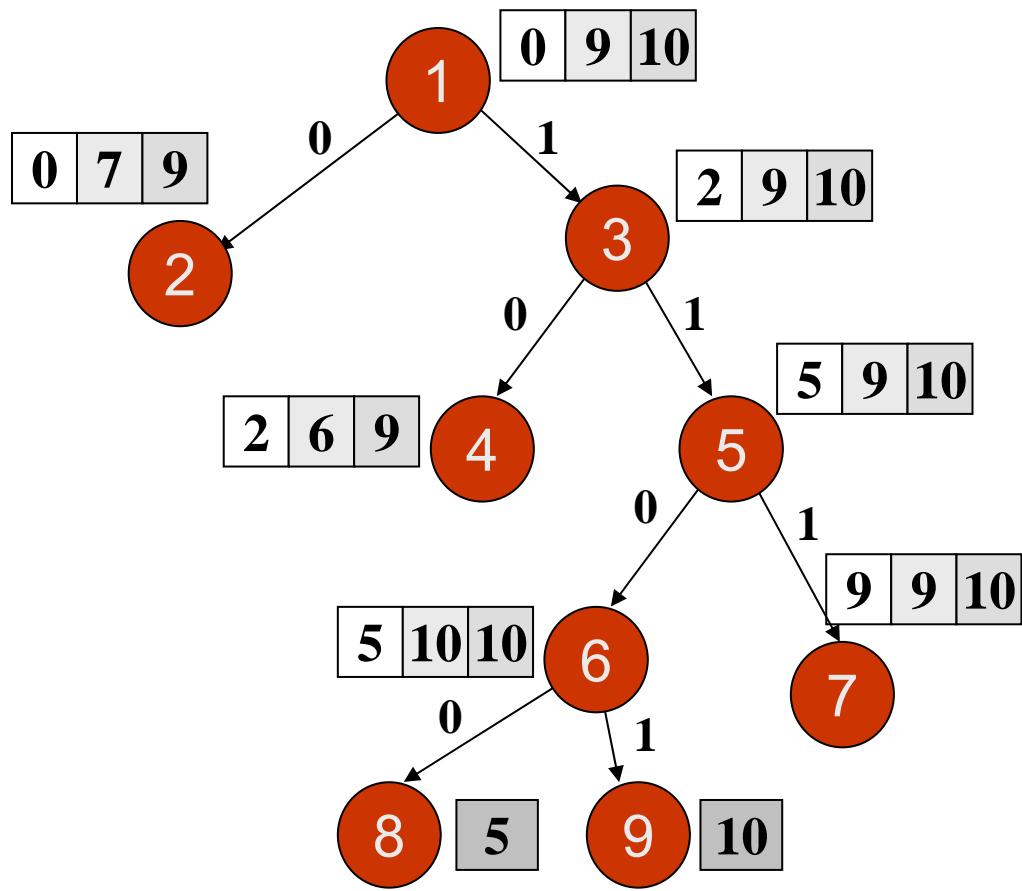
```
res.CI := 0;  
res.CS := [MochilaVoraz (1, M, v, w)];  
res.BE := Mochila01Voraz (1, M, v, w);  
res.nivel := 0;  
res.v_act := 0;      res.w_act := 0;  
Devolver res;
```

**Generar (x: nodo; i: (0, 1); v,w: array [1..n] of int; M: int): nodo;**

```
res.tupla := x.tupla;  
res.nivel := x.nivel + 1;  
res.tupla[res.nivel] := i;  
Si i = 0 Entonces res.v_act := x.v_act; res.w_act := x.w_act;  
Sino res.v_act := x.v_act + v[res.nivel]; res.w_act := x.w_act +  
w[res.nivel];  
res.CI := res.v_act;  
res.BE := res.CI + Mochila01Voraz (res.nivel+1, M - res.w_act, v, w);  
res.CS := res.CI + [MochilaVoraz (res.nivel+1, M - res.w_act, v, w)];  
Si res.w_act > M Entonces {Sobrepasa el peso M: descartar el nodo}  
    res.CI := res.CS := res.BE := -∞;  
Devolver res;
```

# El Problema de la Mochila 0/1

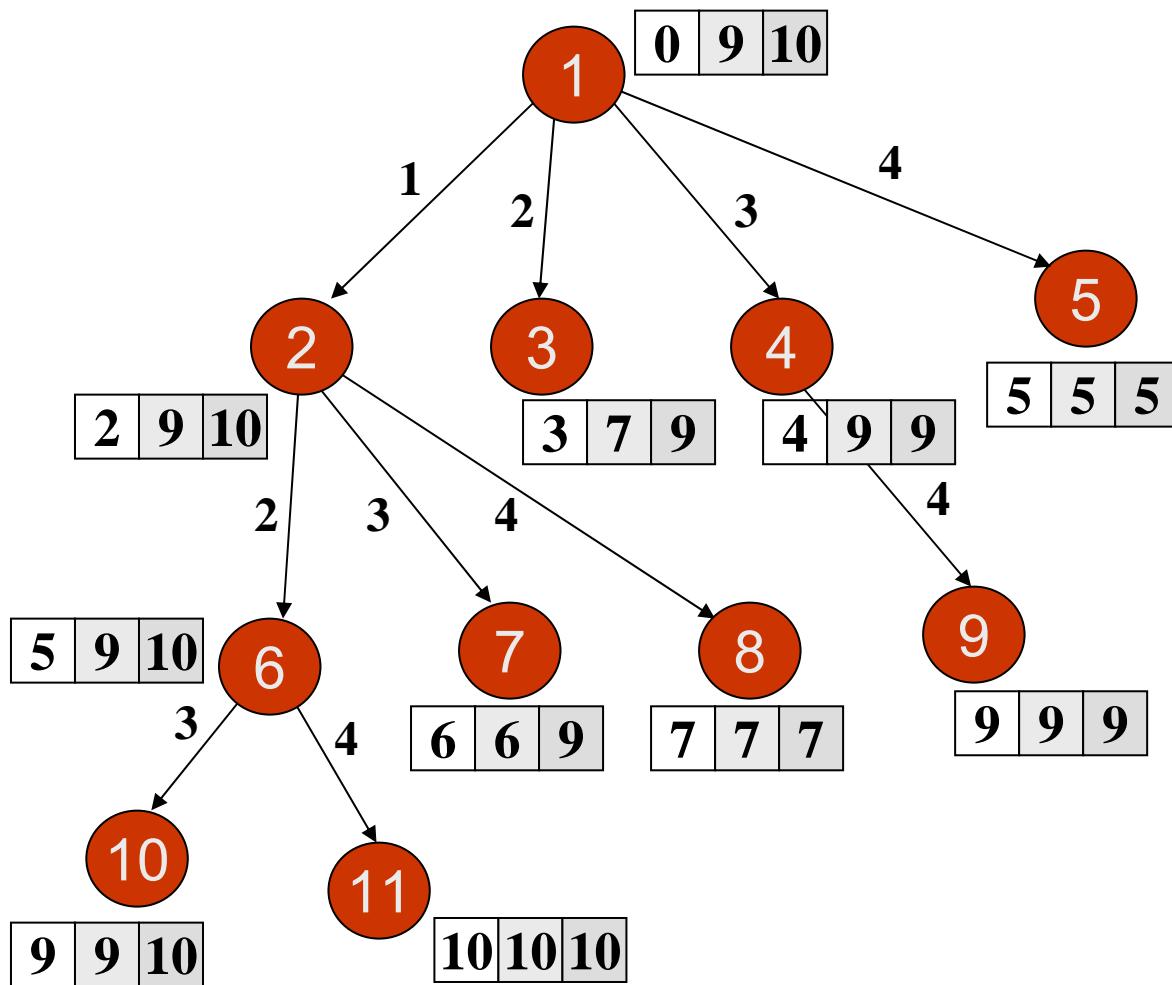
- Ejemplo.  $n = 4, M = 7, v = (2, 3, 4, 5), w = (1, 2, 3, 4)$



| C  | LNV                                           |
|----|-----------------------------------------------|
| 0  | $1$                                           |
| 2  | $3 \rightarrow 2$                             |
| 5  | $5 \rightarrow 2 \rightarrow 4$               |
| 9  | $6 \rightarrow 7 \rightarrow 2 \rightarrow 4$ |
| 10 | $7 \rightarrow 2 \rightarrow 4$               |
| 10 | $2 \rightarrow 4$                             |
| 10 | $4$                                           |

# El Problema de la Mochila 0/1

- Ejemplo. Utilizando un árbol combinatorio y *LC-FIFO*,  $n = 4$ ,  $M = 7$ ,  $v = (2, 3, 4, 5)$ ,  $w = (1, 2, 3, 4)$



| C  | LNV           |
|----|---------------|
| 0  | 1             |
| 5  | 2 → 4 → 3     |
| 7  | 4 → 6 → 3 → 7 |
| 9  | 6 → 3 → 7     |
| 10 | 3 → 7         |
| 10 | 7             |

# Recordemos...

## **Aplicación de ramificación y poda (proceso metódico):**

- 1) Definir la representación de la solución. A partir de un nodo, cómo se obtienen sus descendientes.
- 2) Dar una manera de calcular el valor de las cotas y la estimación del beneficio.
- 3) Definir la estrategia de ramificación y de poda.
- 4) Diseñar el esquema del algoritmo.

# El problema de asignación de tareas

## Enunciado del problema de asignación

- **Datos del problema:**
  - **n**: número de personas y de tareas disponibles.
  - **B**: array [1..n, 1..n] de entero. Rendimiento o beneficio de cada asignación. **B[i, j]** = beneficio de asignar a la persona **i** la tarea **j**.
- **Resultado:**
  - Realizar **n** asignaciones  $\{(p_1, t_1), (p_2, t_2), \dots, (p_n, t_n)\}$ .
- **Formulación matemática:**

Maximizar  $\sum_{i=1..n} B[p_i, t_i]$ , sujeto a  
la restricción  $p_i \neq p_j, t_i \neq t_j, \forall i \neq j$

|          |   | Tareas |   |   |
|----------|---|--------|---|---|
|          |   | 1      | 2 | 3 |
| Personas | B | 5      | 6 | 4 |
|          | 1 | 3      | 8 | 2 |
|          | 2 | 6      | 5 | 1 |

# El problema de asignación de tareas

## 1) Representación de la solución

- Desde el punto de vista de las **personas**:

$s = (t_1, t_2, \dots, t_n)$ , siendo  $t_i \in \{1, \dots, n\}$ , con  $t_i \neq t_j, \forall i \neq j$

–  $t_i \rightarrow$  número de tarea asignada a la persona  $i$ .

**tipo**

**Nodo = registro**

    tupla: **array [1..n] de entero**

    nivel: entero

    bact: entero

    CI, BE, CS: entero

**finregistro**

**1.a) ¿Cómo es el nodo raíz?**

**1.b) ¿Cómo generar los hijos de un nodo?**

**1.c) ¿Cómo es la función Solución(x: Nodo): booleano?**

# El problema de asignación de tareas

**1.a) Nodo raíz**

raiz.nivel:= 0

raiz.bact:= 0

**1.b) Para cada y hijo de un nodo x**

**para i:= 1, ..., n hacer**

y.nivel:= x.nivel+1

y.tupla:= x.tupla

**si Usada(x, i) entonces break**

y.tupla[y.nivel]:= i

y.bact:= x.bact + B[y.nivel, i]

.....

**operación Usada(m: Nodo; t: entero): booleano**

**para i:= 1,..., m.nivel hacer**

**si m.tupla[i]==t entonces devolver TRUE**

**devolver FALSE**

# El problema de asignación de tareas

- **Otra posibilidad:** almacenar las tareas usadas en el nodo.  
**tipo**  
**Nodo = registro**  
    tupla: **array [1..n] de entero**  
    nivel: entero  
    bact: entero  
    usadas: **array [1..n] de booleano**  
    CI, BE, CS: entero  
**finregistro**
- **Resultado:** se tarda menos tiempo pero se usa más memoria.

**1.c) Función Solución(x: Nodo): booleano**

**devolver x.nivel==n**

# El problema de asignación de tareas

## 2) Cálculo de las funciones $CI(x)$ , $CS(x)$ , $BE(x)$

|          |   | Tareas |   |   |
|----------|---|--------|---|---|
|          |   | 1      | 2 | 3 |
| Personas | B | 5      | 6 | 4 |
|          | 1 | 3      | 8 | 2 |
|          | 2 | 6      | 5 | 1 |
|          | 3 |        |   |   |

### 2) Posibilidad 1. Estimaciones triviales:

- **CI.** Beneficio acumulado hasta ese momento:  $x.CI := x.bact$
- **CS.** CI más suponer las restantes asignaciones con el máximo global:  $x.CS := x.bact + (n-x.nivel)*\max(B[\cdot,\cdot])$
- **BE.** La media de las cotas:  $x.BE := (x.CI+x.CS)/2$

# El problema de asignación de tareas

## 2) Posibilidad 2. Estimaciones precisas:

- **Cl.** Resolver el problema usando un algoritmo voraz.  
 $x.Cl := x.bact + \text{AsignaciónVoraz}(x)$
- **AsignaciónVoraz(x):** Asignar a cada persona la tarea libre con más beneficio.

**operación AsignaciónVoraz(m: Nodo): entero**

bacum:= 0

**para** i:= m.nivel+1, ..., n **hacer**

k:=  $\operatorname{argmax}_{\forall j \in \{1..n\} \text{ } m.usadas[j] == \text{FALSE}} B[i, j]$

m.usadas[k]:= TRUE

bacum:= bacum + B[i, k]

**finpara**

**devolver** bacum

# El problema de asignación de tareas

## 2) Posibilidad 2. Estimaciones precisas:

- **CS.** Asignar las tareas con mayor beneficio (aunque se repitan).  
 $x.CS := x.bact + \text{MáximosTareas}(x)$

**operación MáximosTareas(m: Nodo): entero**

bacum:= 0

**para** i:= m.nivel+1, ..., n **hacer**

k:=  $\operatorname{argmax}_{\forall j \in \{1..n\} \text{ and } m.usadas[j] == \text{FALSE}} B[i, j]$

bacum:= bacum + B[i, k]

**finpara**

**devolver** bacum

- **BE.** Tomar la media:  $x.BE := (x.CI + x.CS)/2$

# El problema de asignación de tareas

## 2) Cálculo de las funciones $CI(x)$ , $CS(x)$ , $BE(x)$

- **Cuestión clave:** ¿podemos garantizar que la solución óptima a partir de  $x$  estará entre  $CI(x)$  y  $CS(x)$ ?
- **Ejemplo.**  $n= 3$ . ¿Cuánto serían  $CI(\text{raíz})$ ,  $CS(\text{raíz})$  y  $BE(\text{raíz})$ ? ¿Cuál es la solución óptima del problema?

|          |   | Tareas |   |   |
|----------|---|--------|---|---|
|          |   | 1      | 2 | 3 |
| Personas | B | 5      | 6 | 4 |
|          | 1 | 3      | 8 | 2 |
|          | 2 | 6      | 5 | 1 |
|          | 3 |        |   |   |

# El problema de asignación de tareas

## 3) Estrategia de ramificación y de poda

### 3.a) Estrategia de poda

- **Variable de poda C:** valor de la mayor cota inferior o solución final del problema.
- **Condición de poda:** podar  $i$  si:  $i.CS \leq C$

### 3.b) Estrategia de ramificación

- **Usar una estrategia MB-LIFO:** explorar primero los nodos con mayor BE y en caso de empate seguir por la rama más profunda.

# El problema de asignación de tareas

4) Esquema del algoritmo. (Exactamente el mismo que antes)

**AsignaciónRyP (n: ent; B: array[1..n,1..n] de ent; var s: Nodo)**

  LNV:= {raiz}

  C:= raiz.Cl

  S:=  $\emptyset$

**mientras** LNV  $\neq \emptyset$  **hacer**

    x:= Seleccionar(LNV)       *// Estrategia MB-LIFO*

    LNV:= LNV - {x}

**si** x.CS > C **entonces**       *// Estrategia de poda*

**para cada** y **hijo de** x **hacer**

**si** Solución(y) AND (y.bact > s.bact) **entonces**

          s:= y

          C:= max (C, y.bact)

**sino si** NO Solución(y) AND (y.CS > C) **entonces**

          LNV:= LNV + {y}

          C:= max (C, y.Cl)

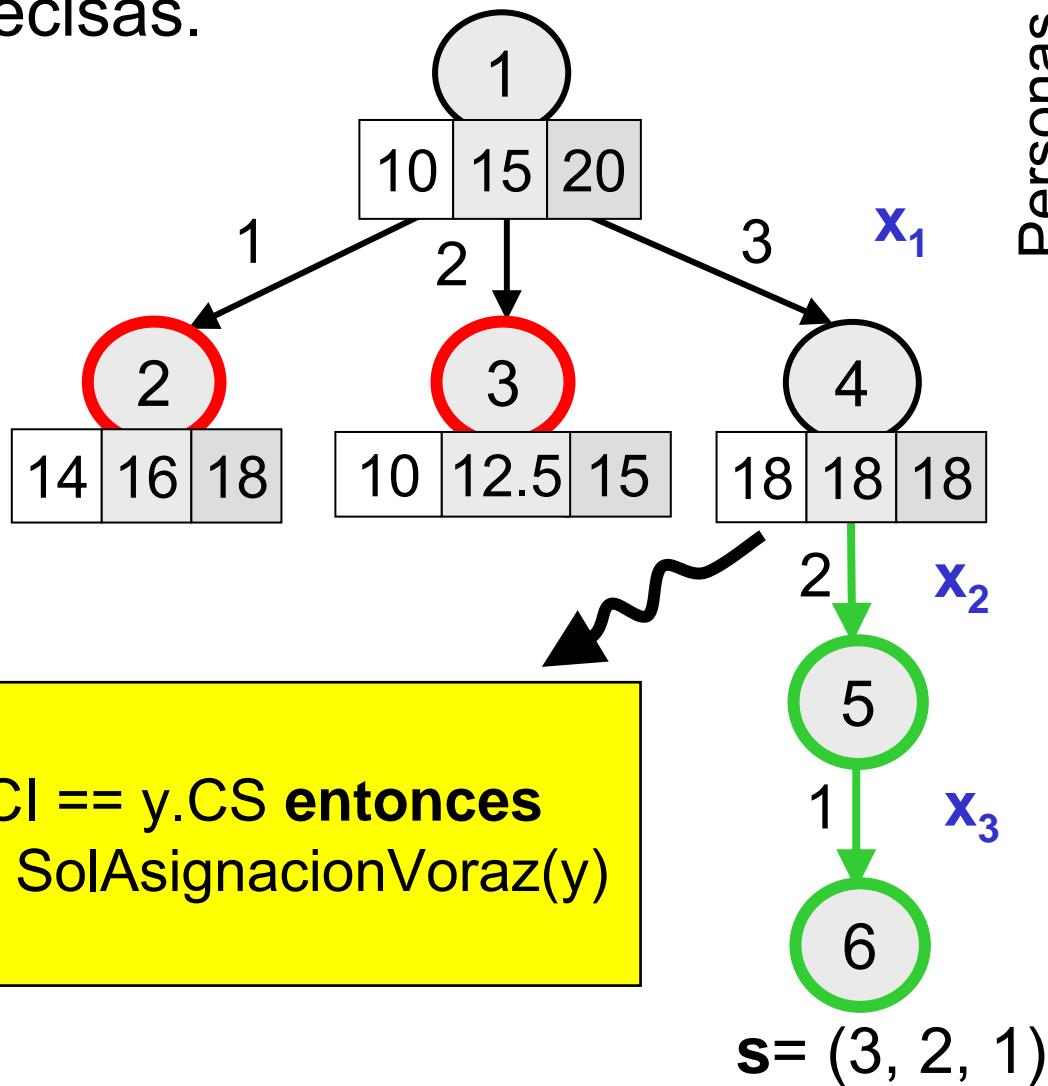
**finsi**

**finpara**

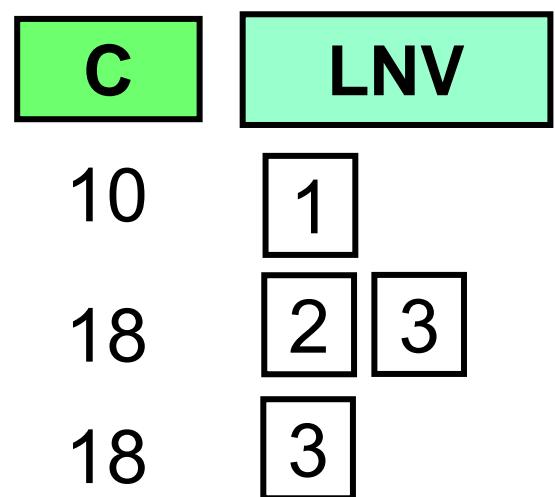
**finmientras**

# El problema de asignación de tareas

- Ejemplo.  $n= 3$ . Estimaciones precisas.

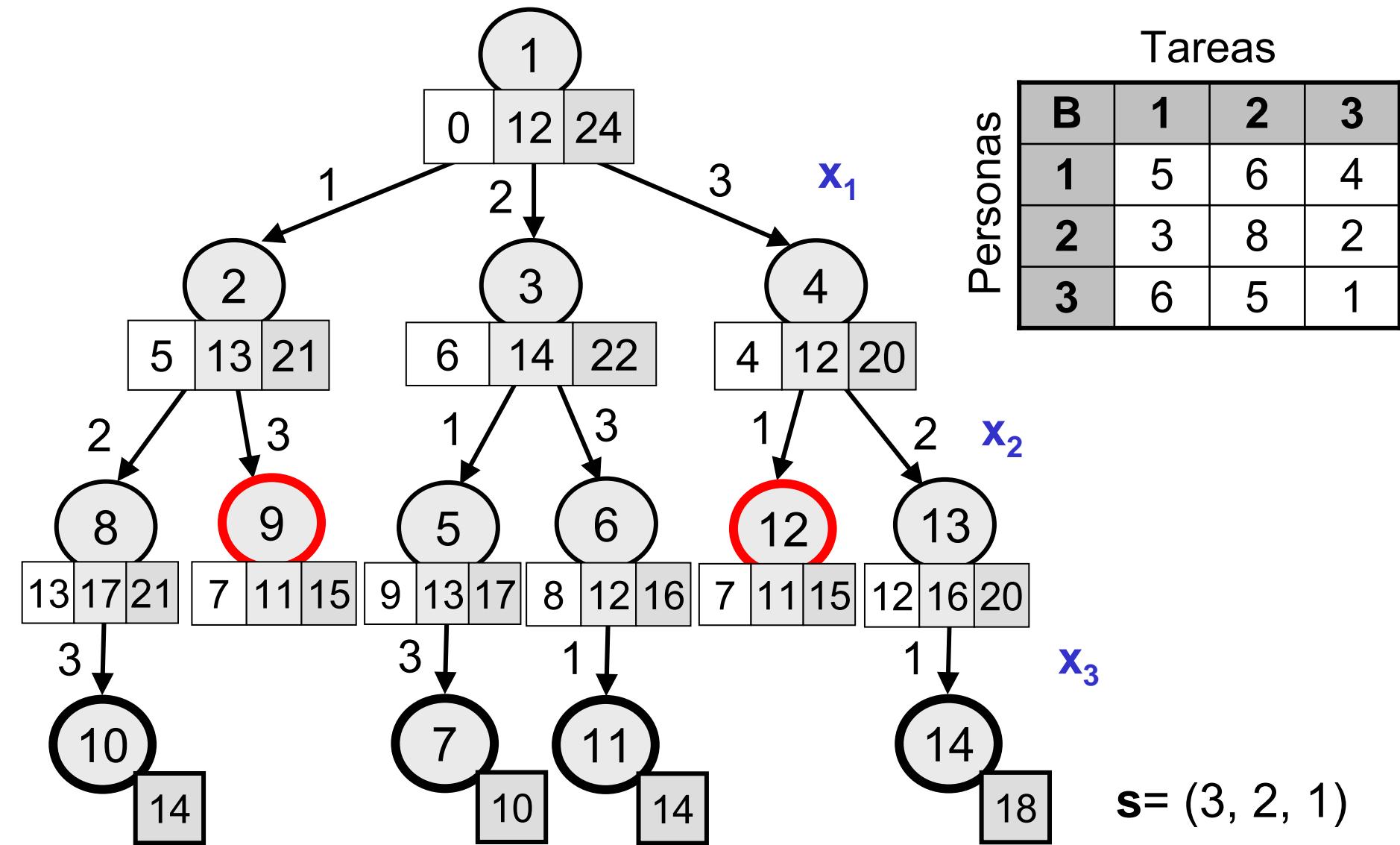


|          |   | Tareas |    |    |
|----------|---|--------|----|----|
|          |   | 1      | 2  | 3  |
| Personas | B | 10     | 15 | 20 |
|          | 1 | 5      | 6  | 4  |
| 2        | 3 | 8      | 2  |    |
| 3        | 6 | 5      | 1  |    |



# El problema de asignación de tareas

- Ejemplo.  $n= 3$ . Usando las estimaciones triviales.



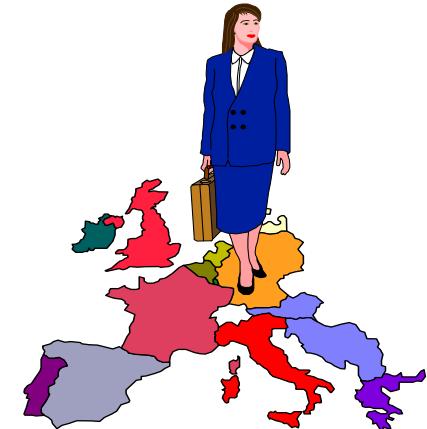
# El problema de asignación de tareas

- Con estimaciones precisas: 4 nodos generados.
- Con estimaciones triviales: 14 nodos generados.
- ¿Conviene gastar más tiempo en hacer estimaciones más precisas?
- ¿Cuánto es el tiempo de ejecución en el peor caso?
- Estimaciones triviales: **O(1)**
- Estimaciones precisas: **O(n(n-nivel))**

# El Problema del Viajante de Comercio

---

- Este problema fue resuelto con **Programación Dinámica**, obteniendo un algoritmo de orden  $O(n^2 2^n)...$
- Para un ' $n$ ' grande, el algoritmo es ineficiente...
- Branch and bound se adapta para solucionarlo



## Recordatorio:

- Encontrar un recorrido de longitud mínima para una persona que tiene que visitar varias ciudades y volver al punto de partida, conocida la distancia existente entre cada dos ciudades.
- Es decir, dado un grafo dirigido con arcos de longitud no negativa, se trata de encontrar un circuito de longitud mínima que comience y termine en el mismo vértice y pase exactamente una vez por cada uno de los vértices restantes

# El Problema del Viajante de Comercio

---

- Formalización:
  - Sean  $G = (V, A)$  un grafo dirigido,  $V = \{1, 2, \dots, n\}$ ,
  - $D[i, j]$  la longitud de  $(i, j) \in A$ ,  $D[i, j] = \infty$  si no existe el arco  $(i, j)$
  - El circuito buscado empieza en el vértice 1
- **Candidatos:**  
 $E = \{ 1, X, 1 \mid X \text{ es una permutación de } (2, 3, \dots, n) \}$   
 $|E| = (n-1)!$
- **Soluciones factibles:**  
 $E = \{ 1, X, 1 \mid X = x_1, x_2, \dots, x_{n-1}, \text{ es una permutación de } (2, 3, \dots, n) \text{ tal que } (i_j, i_{j+1}) \in A, 0 < j < n, (1, x_1) \in A, (x_{n-1}, 1) \in A \}$
- **Función objetivo:**  
 $F(X) = D[1, x_1] + D[x_1, x_2] + D[x_2, x_3] + \dots + D[x_{n-2}, x_{n-1}] + D[x_n, 1]$

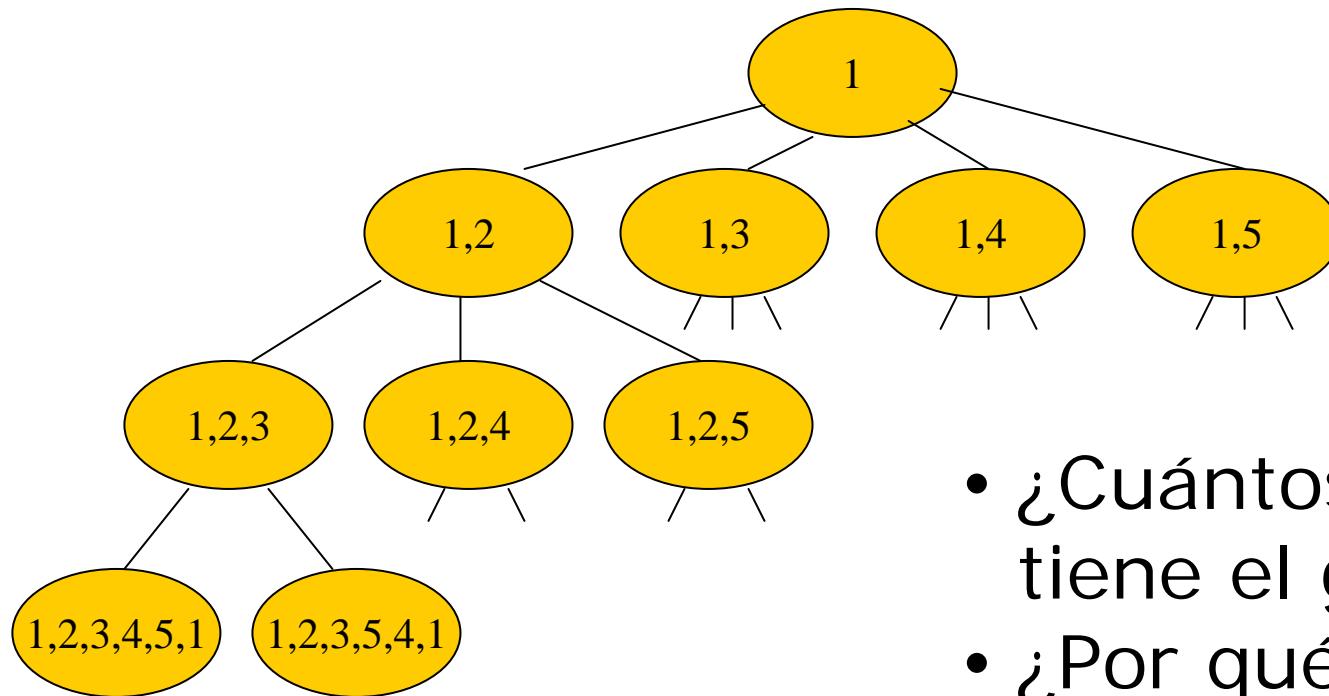
# El Problema del Viajante de Comercio

---

- **Recordatorio:** Ciclo en el grafo en el que TODOS los vértices del grafo se visitan sólo una vez al menor costo
- **Arbol de búsqueda de soluciones:**
  - La raíz del árbol (nivel 0) es el vértice de inicio del ciclo
  - En el nivel 1 se consideran TODOS los vértices menos el inicial
  - En el nivel 2 se consideran TODOS los vértices menos los 2 que ya fueron visitados
  - Y así sucesivamente hasta el nivel ' $n-1$ ' que incluirá al vértice que no ha sido visitado

# Ejemplo

---



- ¿Cuántos vértices tiene el grafo?
- ¿Por qué 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 se deja de explorar ahí...

# 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 los ceros)
- La sumatoria de los mejores arcos de cada vértice faltante, más **el costo del camino ya acumulado**, es una estimación válida para tomar decisiones respecto a las podas en el árbol

# Ejemplo

---

- Dada la siguiente matriz de adyacencias, ¿cuál es el costo mínimo posible de visitar todos los nodos una sola vez?

|    |    |    |    |    |
|----|----|----|----|----|
| 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  |

$$\begin{array}{l} \longrightarrow \text{Mínimo} = 4 \\ \longrightarrow \text{Mínimo} = 7 \\ \longrightarrow \text{Mínimo} = 4 \\ \longrightarrow \text{Mínimo} = 2 \\ \longrightarrow \text{Mínimo} = 4 \\ \hline \text{TOTAL} = 21 \end{array}$$

# Ejemplo

---

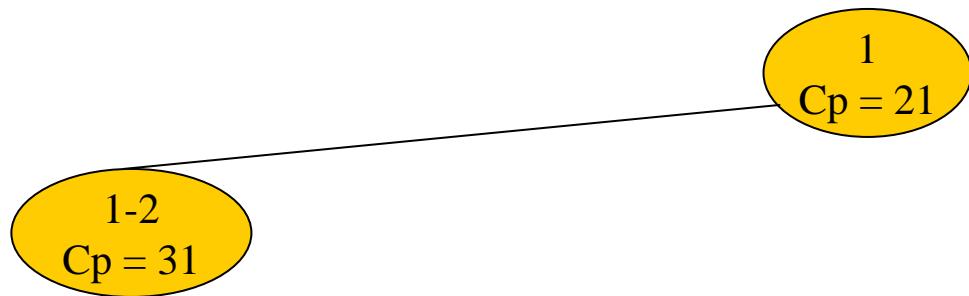
|    |    |    |    |    |
|----|----|----|----|----|
| 0  | 14 | 4  | 10 | 20 |
| 14 | 0  | 7  | 8  | 7  |
| 4  | 5  | 0  | 7  | 16 |
| 11 | 7  | 9  | 0  | 2  |
| 18 | 7  | 17 | 4  | 0  |

$$\begin{matrix} 1 \\ \text{Cp} = 21 \end{matrix}$$

***Costo máximo = ∞***

# 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 máximo = ∞**

## 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

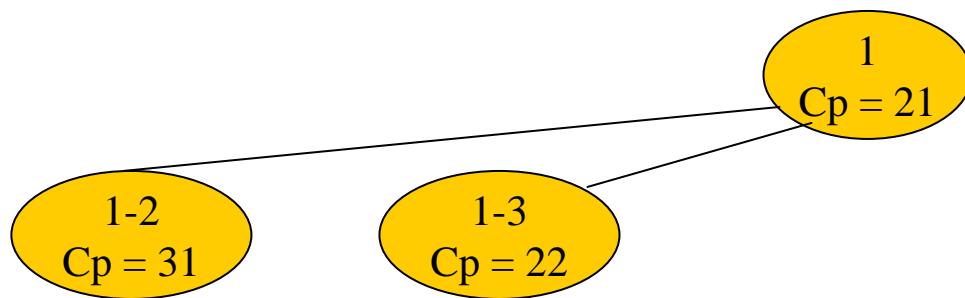
Más mínimo de 4-1, 4-3 y 4-5: 2

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 máximo =  $\infty$*

## 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**

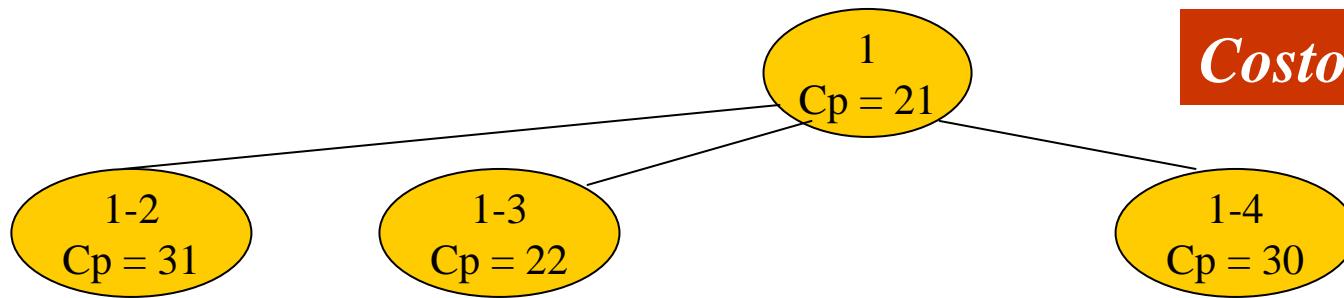
Más mínimo de 4-1, 4-3 y 4-5: **2**

Más mínimo de 5-1, 5-3 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 máximo =  $\infty$*

## 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**

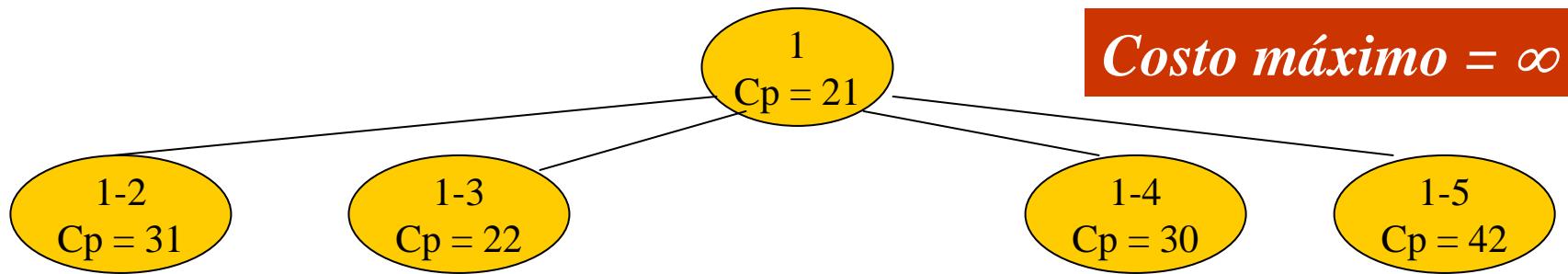
Más mínimo de 2-1, 2-3 y 2-5: **7**

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  |



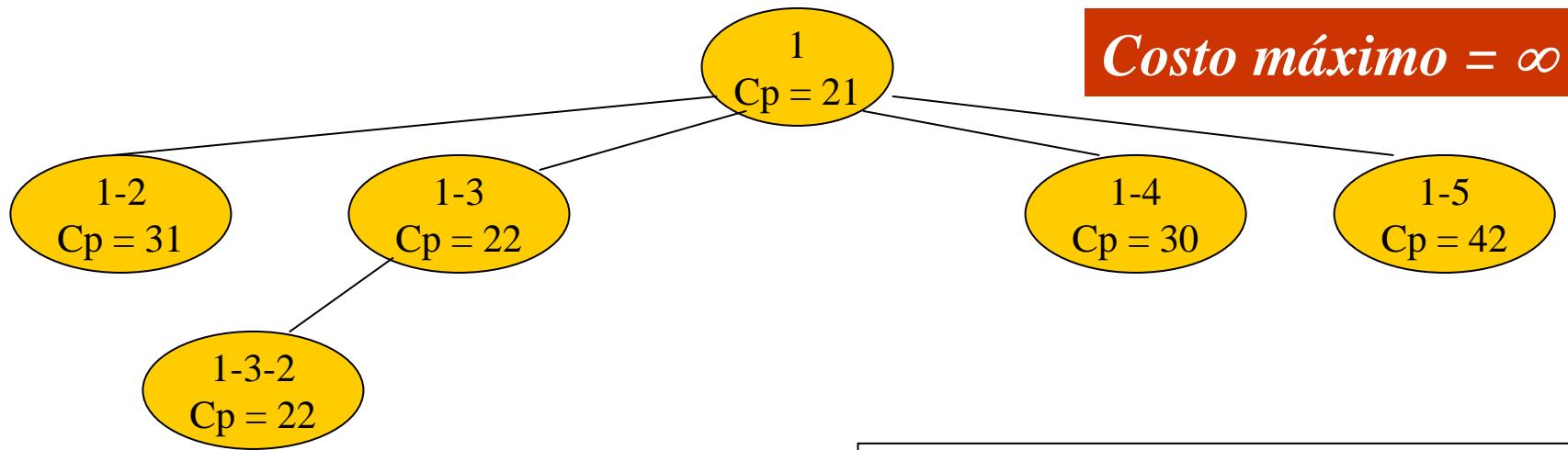
¿Cuál es el mejor nodo para expandir?

### 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

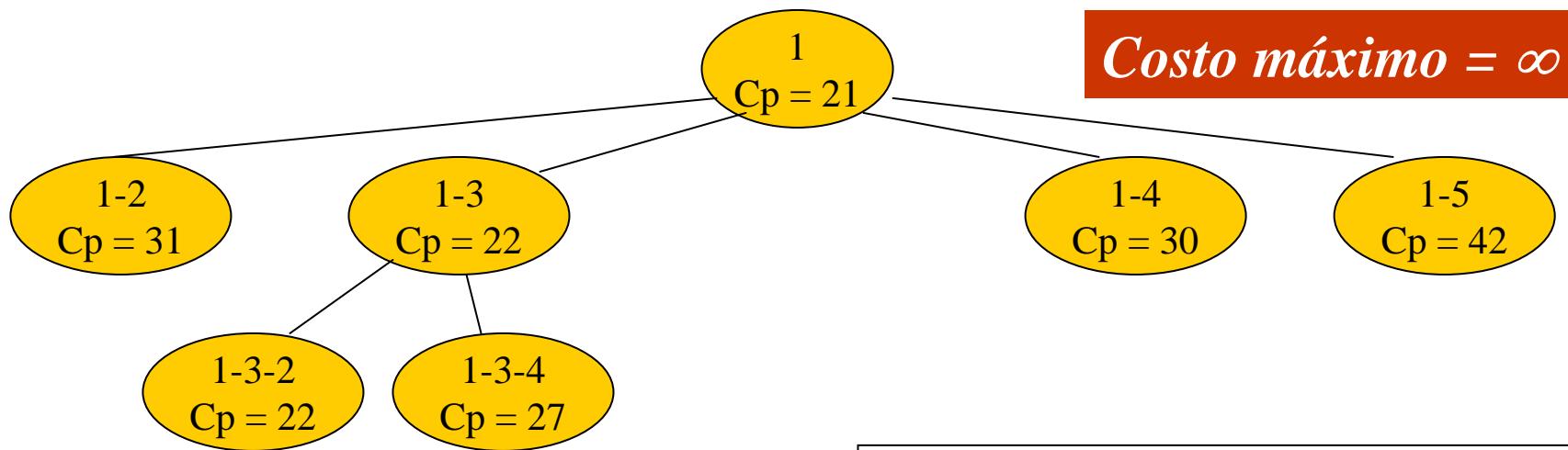
|    |    |    |    |    |
|----|----|----|----|----|
| 0  | 14 | 4  | 10 | 20 |
| 14 | 0  | 7  | 8  | 7  |
| 4  | 5  | 0  | 7  | 16 |
| 11 | 7  | 9  | 0  | 2  |
| 18 | 7  | 17 | 4  | 0  |



**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  |

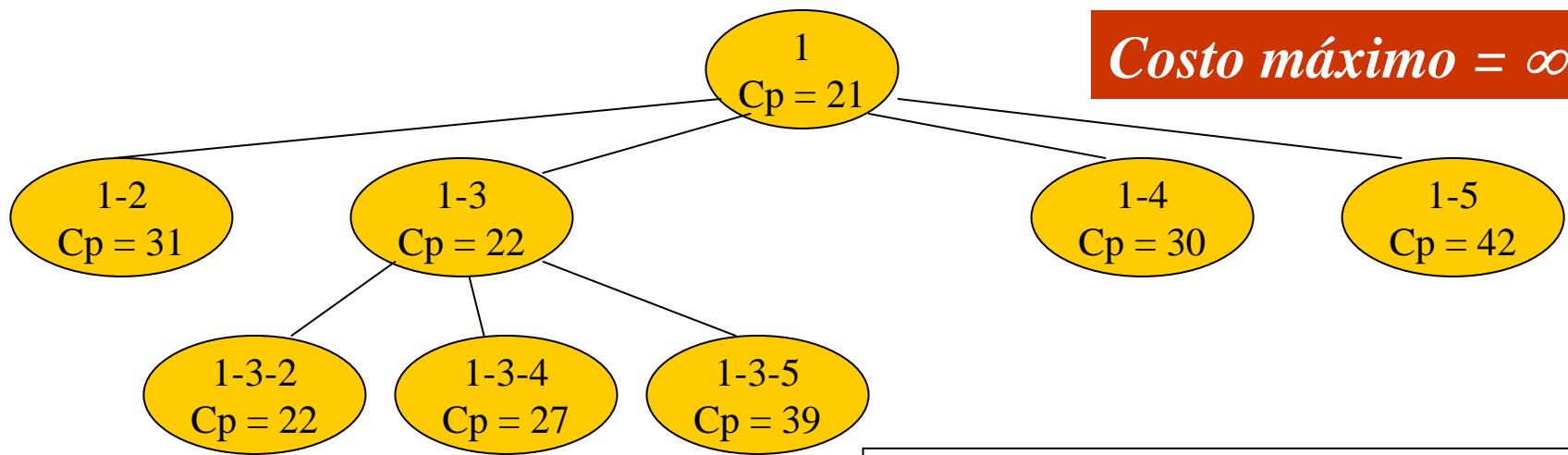


**Cálculo del Costo posible:**

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  |

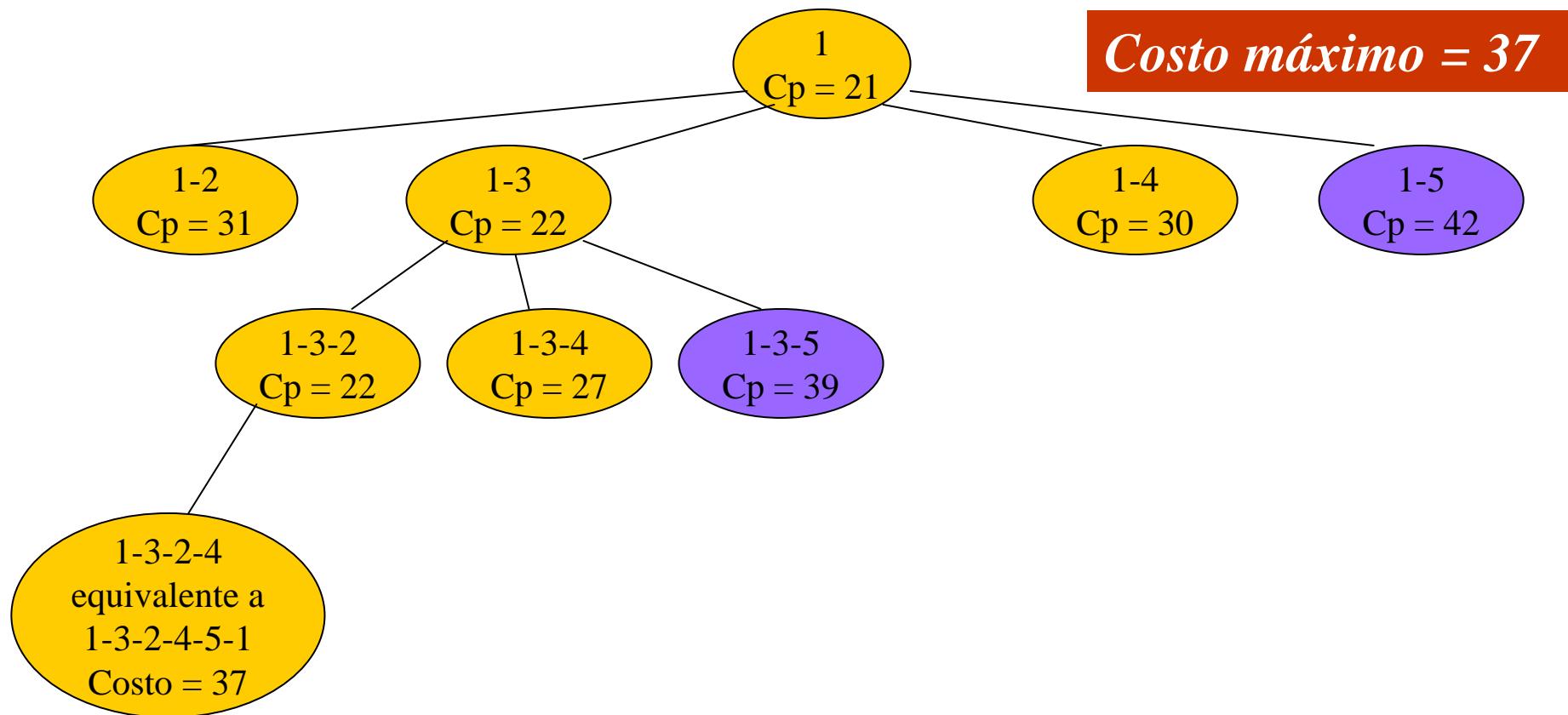


¿Cuál es el mejor nodo para expandir?

**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**

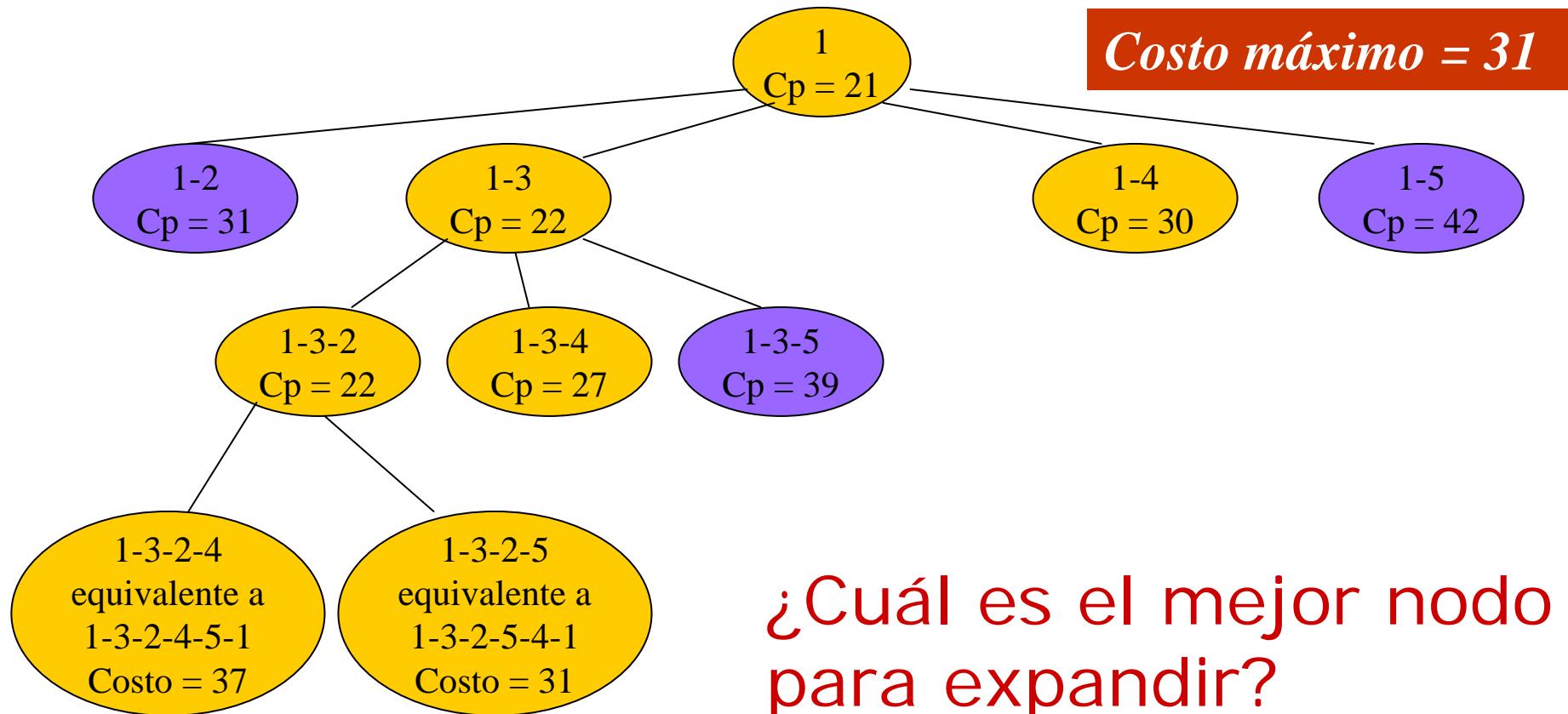
# Ejemplo

|    |    |    |    |    |
|----|----|----|----|----|
| 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  |



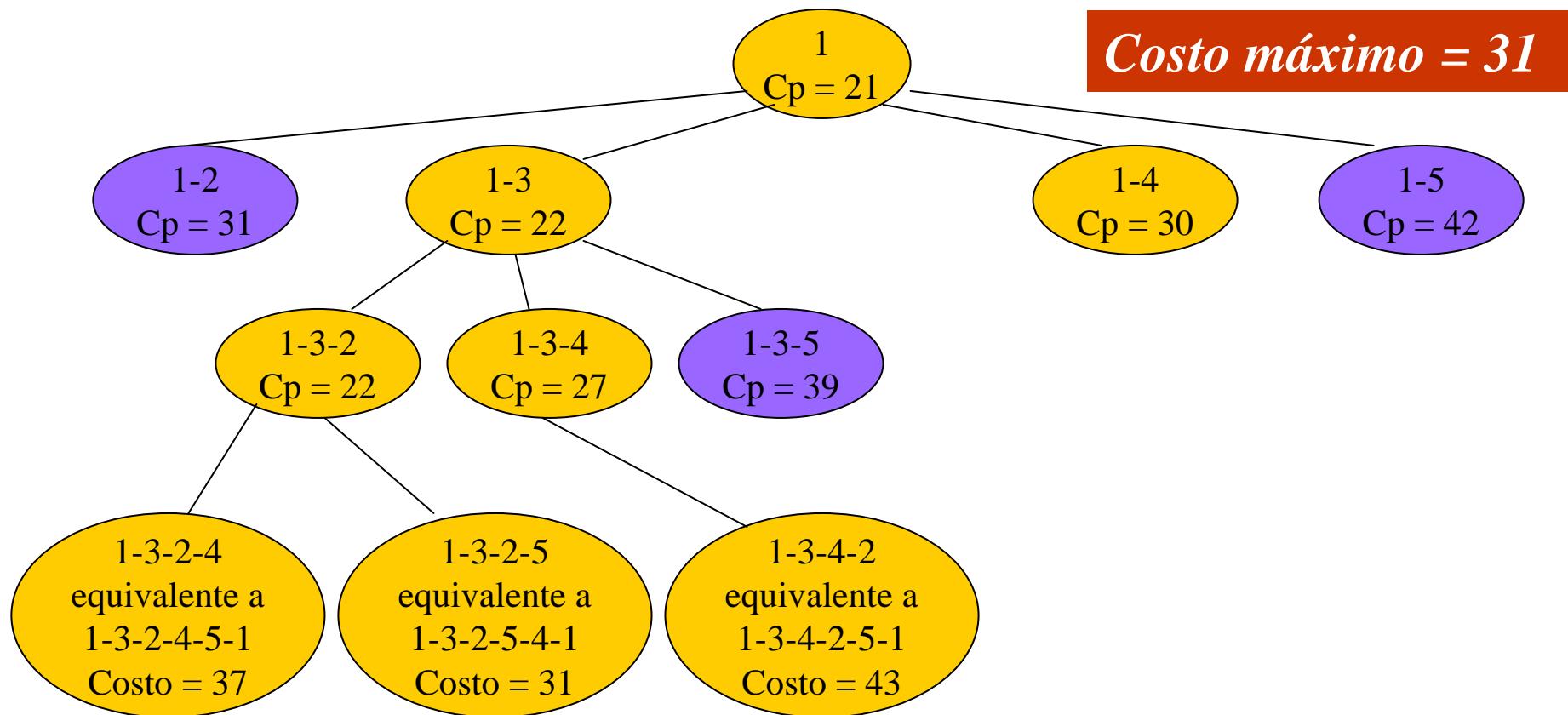
# Ejemplo

|    |    |    |    |    |
|----|----|----|----|----|
| 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  |



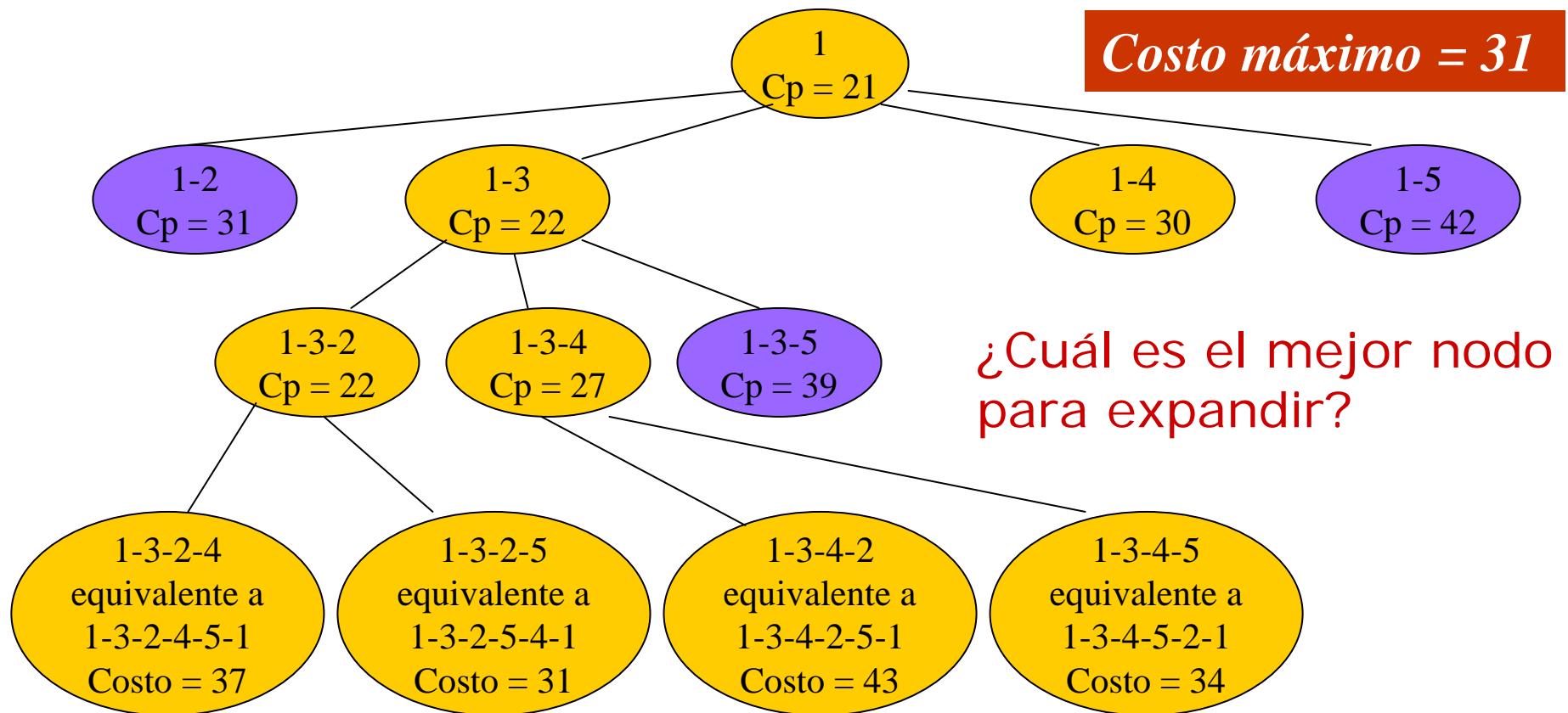
# Ejemplo

|    |    |    |    |    |
|----|----|----|----|----|
| 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  |



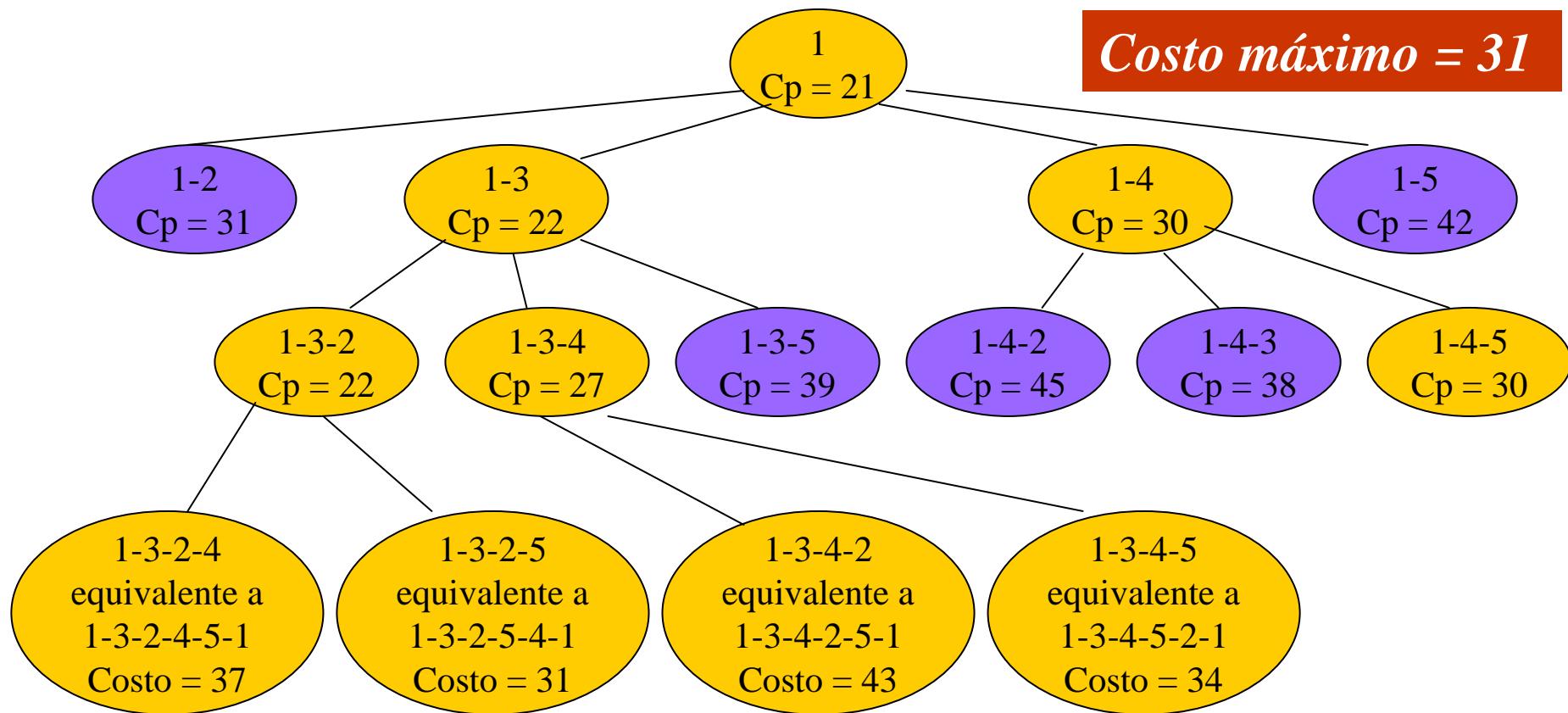
# Ejemplo

|    |    |    |    |    |
|----|----|----|----|----|
| 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

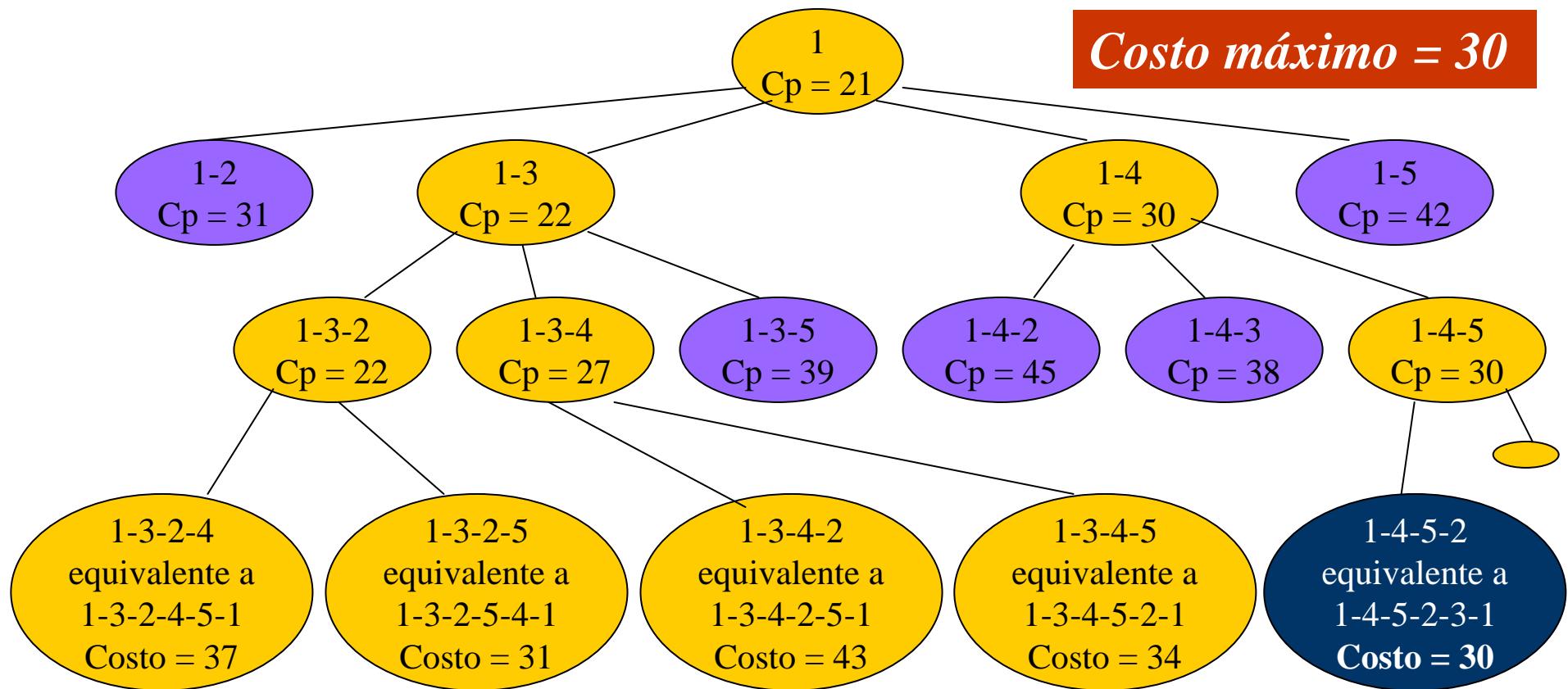
|    |    |    |    |    |
|----|----|----|----|----|
| 0  | 14 | 4  | 10 | 20 |
| 14 | 0  | 7  | 8  | 7  |
| 4  | 5  | 0  | 7  | 16 |
| 11 | 7  | 9  | 0  | 2  |
| 18 | 7  | 17 | 4  | 0  |



¿Cuál es el mejor nodo para expandir?

# 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  |



# El Problema del Viajante de Comercio (Conclusión final)

---

- Branch and bound ofrece una opción más de solución del problema del Viajante de Comercio
- Sin embargo, NO asegura tener un buen comportamiento en cuanto a eficiencia, ya que en el peor caso tiene un tiempo exponencial...
- El problema puede ser resuelto con algoritmos heurísticos: SA, AG, TS, ...

# Ramificación y poda: Conclusiones

**Ramificación y poda:** mejora y generalización de la técnica de backtracking.

- **Idea básica.** Recorrido implícito en árbol de soluciones:
  - Distintas estrategias de ramificación.
  - Estrategias LC: explorar primero las ramas más prometedoras.
  - Poda basada en acotar el beneficio a partir de un nodo: CI, CS.
- **Estimación de cotas:** aspecto clave en RyP. Utilizar algoritmos de avance rápido.
- **Compromiso tiempo-exactitud.** Más tiempo → mejores cotas. Menos tiempo → menos poda.

# Algorítmica

---

**Tema 1. Planteamiento General**

**Tema 2. La Eficiencia de los Algoritmos**

**Tema 3. Algoritmos “Divide y vencerás”**

**Tema 4. Algoritmos Voraces (“Greedy”)**

**Tema 5. Algoritmos basados en Programación Dinámica**

**Tema 6. Algoritmos para la Exploración de Grafos  
 (“Backtracking”, “Branch and Bound”)**

**Tema 7. Otras metodologías algorítmicas**