

ALGORÍTMICA

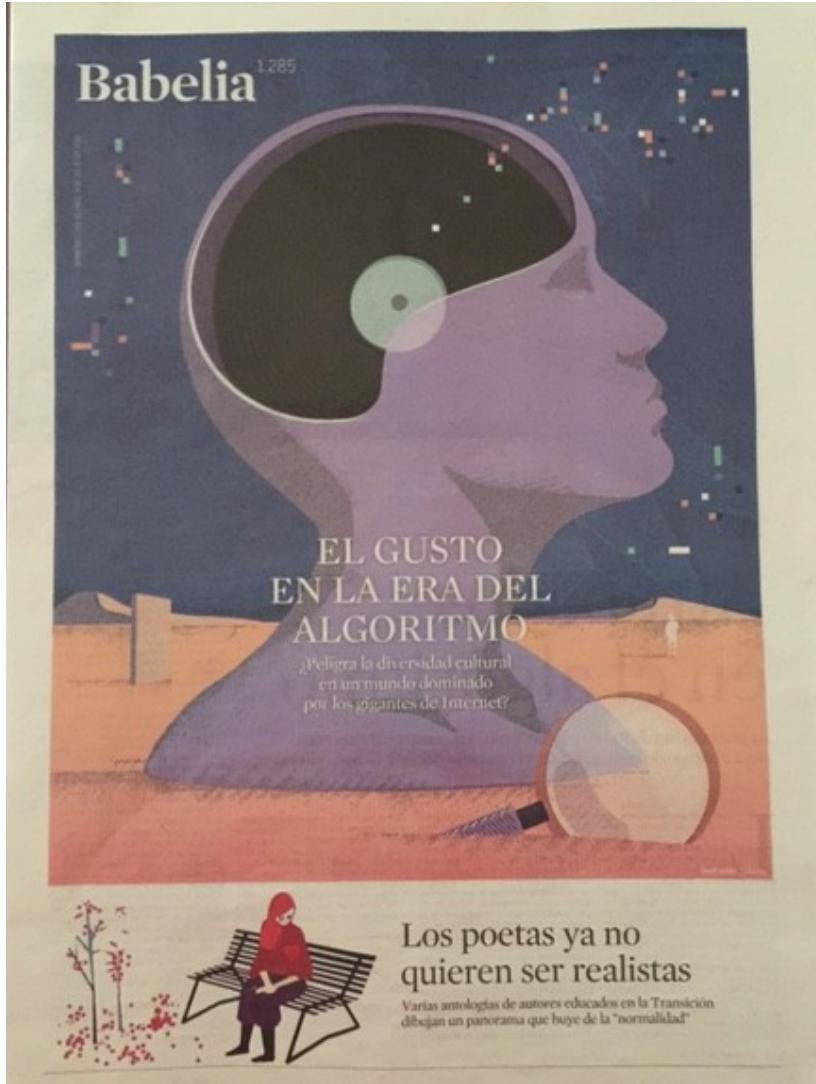
Capítulo 1: La Eficiencia de los Algoritmos

Tema 1: Planteamiento general

- Ciencia de la Computación
- El concepto de algoritmo
- Elección de un algoritmo
- Problemas y casos
- Distintos tipos de casos



¿Es actual este tema?



- **El gusto en la era del algoritmo**
- La prescripción artificial en plataformas digitales como Amazon, Netflix, Google o Facebook eleva el riesgo de homogeneizar la identidad y los hábitos de consumo cultural
- El algoritmo, sostienen sus críticos, nos hace aburridos, previsibles, y empobrece nuestra curiosidad cultural
- De algún modo, Internet y las plataformas de streaming cultural han alumbrado un universo parecido al que describía Borges en La biblioteca de Babel
- Daniel Verdú, 8 de julio de 2016

¿Que es la Ciencia de la Computación?

- **La Ciencia de la Computación** es el estudio de los computadores.
- **La Ciencia de la Computación** es el estudio de cómo escribir programas.
- **La Ciencia de la Computación** es el estudio de las aplicaciones de computación.

El estudio de cómo usar un computador/software es una parte de la Ciencia de la Computación igual que el manejo de un móvil es una parte de la Ingeniería de Telecomunicaciones.

La Ciencia de la Computación es a los computadores lo mismo que la Astronomía es a los telescopios, la Biología a los microscopios o la Química a las pipetas.

La programación es una parte muy importante de la Ciencia de la Computación. Pero es una herramienta para implementar ideas y soluciones. Un programa es un medio para lograr un objetivo, pero no un objetivo.

¿Entonces que es la Ciencia de la Computación?

- Las definiciones anteriores:

Aunque no necesariamente (completamente) equivocadas, son incompletas, y por tanto incorrectas

- Una definición mas precisa:

La Ciencia de la Computación es el estudio de los Algoritmos, incluyendo sus propiedades, su hardware, sus aspectos lingüísticos (sintácticos y semánticos) y sus aplicaciones.

¿Qué es un algoritmo?

- **Definición formal:**

Una secuencia finita y ordenada de pasos, exentos de ambigüedad, tal que al llevarse a cabo con fidelidad, dará como resultado que se realice la tarea para la que se ha diseñado (se obtenga la solución del problema planteado) con recursos limitados y en tiempo finito.

- O, más informalmente:

- Un método por etapas para realizar alguna tarea.
- No hay una única definición de algoritmo (hay un “algoritmo para calcular el número π , pero ...)
- Un programa es una serie de instrucciones ordenadas, codificadas en lenguaje de programación que expresa un algoritmo y que puede ser ejecutado en un computador.

**Algoritmo y programa son conceptos diferentes.
Nuestro interés se centra en los algoritmos.**

¿Que es un algoritmo?

- Esta definición es un tanto imprecisa
 - ¿Cuando un algoritmo estará completamente exento de ambigüedad?
 - ¿Como se define fidelidad?
 - Un gran número de años es un tiempo finito?
- Se puede formalizar mucho mas:
- **Definición de Bazaraa, Sherly y Shetty (2006) :**

Un método de solución (un algoritmo) para resolver un problema es un proceso iterativo que genera una sucesión de puntos, conforme a un conjunto dado de instrucciones, y un criterio de parada

¿Qué es un algoritmo?

- **Definición de Knuth (1997)**
- Un método computacional es una cuaterna (Q, I, Ω, f) en la que Q es un conjunto que contiene a I y a Ω como subconjuntos y f es una función de Q en Q tal que
$$f(q) = q, \quad \forall q \in \Omega$$
- Y donde Q es el conjunto de los estados del cálculo, I el input, Ω el output y f la regla de cálculo que se esté aplicando. Cada input $x \in I$ define una sucesión computacional, x_0, x_1, x_2, \dots , como sigue:
$$x_0 = x \text{ y } x_{k+1} = f(x_k) \text{ cuando } k \geq 0$$
- Se dice que la sucesión computacional termina en k etapas si k es el menor entero para el que x_k está en Ω , y entonces en ese caso se dice que a partir de x se obtiene como output x_k .
- Algunas sucesiones computacionales pueden no terminar nunca.
- **Un algoritmo es un método computacional que termina en un número finito de etapas $\forall x \in I$.**

Ejemplo de la definición de Knuth

- El Algoritmo de Euclides (“Elementos de Euclides”, libro 7, proposiciones 1 y 2) calcula el máximo común divisor (mcd) de dos números.

Algoritmo E (Algoritmo de Euclides). Dados dos números enteros positivos m y n , encontrar su mcd, es decir, el mayor número entero positivo que divide exactamente tanto a m como a n .

E1. [Cálculo del resto] Dividir m por n . Sea r el resto ($0 \leq r < n$)

E2. [¿Es cero?] Si $r = 0$ el algoritmo termina. La respuesta es n .

E3. [Descenso] Tomar $m = n$, $n = r$ y volver a la etapa E1.

- Sea Q el conjunto de todos los singletones (n), todos los pares ordenados (m,n) y todas las cuaternas ordenadas $(m,n,r,1)$, $(m,n,r,2)$ y $(m,n,p,3)$, donde m , n y p son números enteros positivos y r es un entero no negativo. Sea I el subconjunto de todos los pares (m,n) y sea Ω el subconjunto de todos los singletones (n).
- ¿Cómo definir la función f ?

Ejemplo de la definición de Knuth

- Si definimos f del siguiente modo,

$$f((m,n)) = (m,n,0,1); f((n)) = (n)$$

$$f((m,n,r,1)) = (m,n, \text{resto de dividir } m \text{ por } n, 2)$$

$$\begin{aligned} f((m,n,r,2)) &= (n) \text{ si } r = 0 \\ &= (m,n,r,3) \text{ en otro caso} \end{aligned}$$

$$f((m,n,p,3)) = (n,p,p,1)$$

- Entonces la correspondencia entre esta notación y el Algoritmo de Euclides es evidente.



Happening Now | August 01, 2016

Print

Donald E. Knuth Awarded SIAM's Highest Honor, Delivers the John von Neumann Lecture

Philadelphia, PA- The Society for Industrial and Applied Mathematics (SIAM) awards the 2016 [John von Neumann Lecture](#) prize to Donald E. Knuth of Stanford University for his transformative contributions to mathematics and computer science. Knuth delivered the associated prize lecture, "Satisfiability and Combinatorics," at the SIAM Annual Meeting in Boston, Massachusetts, on July 12. The highest honor awarded by SIAM, the flagship lecture recognizes outstanding and distinguished contributions to the field of applied mathematical sciences and the effective communication of these ideas to the community.



Pam Cook, University of Delaware; Donald Knuth, Stanford CSD; Jim Crowley, SIAM Executive Director

Knuth founded the field of analysis of algorithms and gave it a rigorous mathematical footing. His ongoing *The Art of Computer Programming* book series represents the definitive reference on algorithms and their analysis. His TeX and Metafont typesetting software has changed the face of mathematical publishing and benefited every mathematician. Knuth is recognized as a brilliant communicator at all levels, from his research monographs to his more popular books such as *Surreal Numbers* and his textbook *Concrete Mathematics*.

Knuth is Professor Emeritus of The Art of Computer Programming at Stanford University, where he supervised the PhD dissertations of approximately 28 students since becoming a professor in 1968. He is the author of

numerous books, including four volumes (so far) of *The Art of Computer Programming*, five volumes of *Computers and Typesetting*, nine volumes of collected papers, and a non-technical book entitled *Bible Texts Illuminated*.

Knuth received his B.S. and M.S. in Mathematics at Case Institute of Technology (now Case Western Reserve University) in 1960 and received his PhD in Mathematics at California Institute of Technology in 1963.

About SIAM News Blogs

The *SIAM News* Blog brings together updates on cutting edge research, events and happenings, as well as insights on broader issues of interest to the applied math and computational science community. Learn more or submit an article or idea.

[LEARN MORE →](#)

Most Recent



Research Nuggets
[Bayesian Model Improves Use of X-ray Radiography in National Security](#)



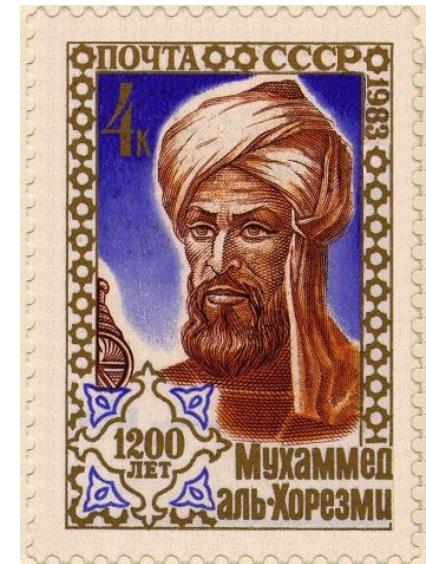
Current Issue
[Car Parts, Neutrons, and Bridges](#)



Announcements
[AN16 Thank You & SIAG Update](#)

¿De donde provienen?

- Etimología.
 - "algos" = pena, sufrimiento en griego.
 - "algor" = frio en latín.
 - "arithmos " = número en griego
 - degeneración de "logaritmo"
- Muhammad ibn Musa (Al-Khuwarizmi) fue un matemático persa (siglo 9 adc)



**Su libro "Al-Jabr wa-al-Muqabilah"
("Algoritmi de Numero Indorum") es la
clave**

Algo de historia

- 300 A. C.
 - Algoritmo de Euclides (mcd)
- 780-850 D.C.
 - Mohammed Ibn Musa al-Khwarizmi.
- 1424 D.C.
 - $\pi = 3.1415926535897932\dots$
- 1845.
 - Lamé: El algoritmo de Euclides hace a lo sumo $1 + \log_{\phi}(n \sqrt{5})$ etapas.
- 1900.
 - Décimo problema de Hilbert (Concepto de algoritmo)
- 1910
 - Pocklington: complejidad en bits
- 1920-1936
 - Post, Goëdel, Church, Turing, Von Neuman ...
- 1965
 - Edmonds: Algoritmos polinomiales vs. Algoritmos exponenciales
- 1971
 - Teorema de Cook (SAT es NP-completo), Reducciones de Karp
- 1997
 - Deep Blue vence a Kasparov
- 20xx
 - $P \neq NP$
 - Lee Sedol pierde frente a AlphaGo.

“Algoritmos” en la vida diaria

- Usamos los “algoritmos” todos los días:
 - Cada vez que realizamos una tarea rutinaria (instrucciones para montar aparatos).
 - Comprando por internet
 - Indicaciones para llegar a sitios y recorrer lugares (PRoA)
 - Al conectarnos con el teléfono (contrataciones, averías, ...)
 - Buscando y comparando información (donación de órganos, hostelería y turismo, ...)
 - Controlando la gestión de los aeropuertos y del tráfico aéreo
 - En el área económica (banca, telebanca, pago ubicuo, ...)
 - Diseño y construcción de proyectos de Ingeniería Civil
 - Ámbito hospitalario (datos, diagnósticos, cirugía de precisión, reconocimiento de imágenes y patrones ...)
 - Casi cualquier tarea que imaginemos tiene detrás un algoritmo

Un algoritmo no es una receta

- Un algoritmo tiene cinco características primordiales
 - a) **Finitud:** ha de terminar después de un tiempo acotado superiormente
 - b) **Especificidad:** cada etapa debe estar precisamente definida; las acciones que hay que llevar a cabo deben estar rigurosamente especificadas para cada caso.
 - c) **Input:** Un algoritmo tiene cero o mas inputs.
 - d) **Output:** uno o mas outputs.
 - 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 (usando solo lápiz y papel, ¿tecnología?)

Tecnología

- Aquí admitiremos como Agente Tecnológico cualquier ente capaz de realizar las etapas que describe el algoritmo, es decir, capaz de ejecutar el algoritmo.
- Por tanto, puede ser una persona, una secuencia de ADN o, claro está, un computador
- En nuestro caso, generalmente, será un computador.
- La necesidad de que una implementación de un algoritmo acabe en un tiempo finito depende de la tecnología



La solución interminable

Tres matemáticos esclarecen un problema que parecía inextricable, aunque se necesitarían 10.000 millones de años para leer la respuesta

ANDRÉS JIMÉNEZ

A veces encontrar la solución es un problema. Que se lo digan si no a los tres matemáticos que han esclarecido lo que era un enigma inextricable. Tres sabios han resuelto el misterio de la bicoloración de las ternas pitagóricas. Hasta aquí todo bien. Lo malo es que leer la solución al problema lleva

aíslar a invertir 10.000 millones de años. Algunos pensarán que mejor permanecer en la bendita ignorancia.

La cuestión de la bicoloración de las ternas pitagóricas tiene tela. Tan tanta que el problema lleva ocupando a la comunidad de matemáticos desde hace 35 años.

Ayer, en la conferencia científica internacional 'SAT 2016', que se celebraba en Burdeos, por fin tres informáticos británico-estadounidenses lograron dilucidar el problema, gracias a un algoritmo de concepción francesa y a la potencia de una supercalculadora. El problema viene ahora. Si se pusiera por escrito la solución, la enumeración equi-

valdría a todos los textos digitalizados de la biblioteca del Congreso de EE UU, es decir, 200 terabytes. A la calculadora Stampede de la Universidad de Texas sólo le hicieron falta dos días para dar con el quid de la cuestión.

Para los aficionados a los pasatiempos sesudos, ahí va el problema, cuyo enunciado es sencillo, a decir de los matemáticos. «Es posible colorear cada entero positivo (1, 2, 3, 4, 5...) de azul o rojo de forma que en ninguna terna (grupo de tres elementos) de enteros a, b y c que responde a la famosa ecuación de Pitágoras $a^2+b^2=c^2$ sean todos del mismo color?». Dicho de otra manera, si en la serie 3, 4 y 5, el 3 y

La Ciencia de la Computación aborda

1) Propiedades formales y Matemáticas

- Como diseñar algoritmos para resolver una gran variedad de problemas.
- Como determinar si los problemas son (eficientemente) computables, es decir, ¡si se pueden especificar por un algoritmo!
- Estudiar la conducta de los algoritmos para decidir si trabajan correctamente y con cuanta eficiencia.

2) Hardware

- El diseño y construcción de equipos capaces de ejecutar algoritmos.
- Los incesantes avances tecnológicos:
 - Computadores cada vez mas rápidos, redes, ...
 - Computación paralela
 - Computación cuántica, molecular...

La Ciencia de la Computación aborda

3) Aspectos lingüísticos

- El diseño de lenguajes de programación y de la traducción a estos lenguajes de los algoritmos para que el hardware disponible pueda ejecutarlos.
- Programación funcional, Programación orientada a objetos, Programación visual, ...

4) Aplicaciones cada vez mas novedosas

- Identificar nuevos problemas importantes para los computadores y del diseño del software que los resuelva.
- Los primeros computadores se usaron sobre todo para cálculos numéricos y tratamiento masivo de datos.
- Ahora, ... se usan en negocios, grafismo, multimedia, domótica, Internet, WWW, ... ¿qué será los siguiente?

¿Como y donde se estudia todo esto?

Algorítmica

- El estudio de los algoritmos incluye el de diferentes e importantes áreas de investigación y docencia, y se suele llamar **Algorítmica (también Teoría de Algoritmos)**
- La Algorítmica considera:
 1. La construcción
 2. La expresión
 3. La validación
 4. El análisis, y
 5. El test de los programas

1. La construcción de algoritmos

- El acto de crear un algoritmo es un arte que nunca debiera automatizarse.
- Pero, indudablemente, 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 mas útiles.

2. La expresión de algoritmos

- Los algoritmos han de tener una expresión lo mas clara y concisa posible.
- Como este un tema clave del estudio de los algoritmos, requerirá que se le dedique tanto esfuerzo como sea necesario, a fin de conseguir lo que se suele denominar un buen estilo.

3. Validación de algoritmos

- Cuando se ha construido el algoritmo, se hace necesario demostrar que calcula correctamente sobre inputs legales.
- Este proceso se conoce con el nombre de validación del algoritmo.
- La validación persigue asegurar que el algoritmo trabajará correctamente independientemente del lenguaje empleado, o la tecnología usada.
- Cuando se ha demostrado la validez del método, es cuando puede escribirse el programa, comenzando una segunda fase del trabajo (la verificación del programa).

4. Análisis de algoritmos

- El análisis de algoritmos se refiere al proceso de determinar cuanto tiempo de cálculo y cuanto almacenamiento requerirá un algoritmo.
- Nos permite hacer juicios cuantitativos sobre el valor de un algoritmo sobre otros.
- A menudo tendremos varios algoritmos para un mismo problema. Habrá que decidir cual es el mejor o cual es el que tenemos que escoger, según algún criterio pre-fijado, para resolverlo
- Puede permitirnos predecir si nuestro software necesitará algún requisito especial.

... y, 5. Test de los programas

- El test de un programa es la última fase que se lleva a cabo y no es propiamente una tarea algorítmica.
- Básicamente supone
 - la corrección de los errores que se detectan, y
 - la comprobación del tiempo y espacio que son necesarios para su ejecución.

Elección de un algoritmo

- A veces no tenemos donde elegir

CAPÍTULO XXIX Radicación

393

II. RAÍZ CUADRADA DE POLINOMIOS

RAÍZ CUADRADA DE POLINOMIOS ENTEROS

Para extraer la raíz cuadrada de un polinomio se aplica la siguiente regla práctica:

- 1) Se ordena el polinomio dado.
- 2) Se halla la raíz cuadrada de su primer término, que será el primer término de la raíz cuadrada del polinomio; se eleva al cuadrado esta raíz y se resta del polinomio dado.
- 3) Se bajan los dos términos siguientes del polinomio dado y se divide el primero de éstos por el doble del primer término de la raíz. El cociente es el segundo término de la raíz. Este 2º término de la raíz con su propio signo se escribe al lado del doble del primer término de la raíz y se forma un binomio; este binomio se multiplica por dicho 2º término y el producto se resta de los dos términos que habíamos bajado.
- 4) Se bajan los términos necesarios para tener tres términos. Se duplica la parte de raíz ya hallada y se divide el primer término del residuo entre el primero de este doble. El cociente es el 3º término de la raíz.

Este 3º término, con su propio signo, se escribe al lado del doble de la parte de raíz hallada y se forma un trinomio; este trinomio se multiplica por dicho 3º término de la raíz y el producto se resta del residuo.

- 5) Se continúa el procedimiento anterior, dividiendo siempre el primer término del residuo entre el primer término del doble de la parte de raíz hallada, hasta obtener residuo cero.

- 1) Hallar la raíz cuadrada de $a^4 + 29a^2 - 10a^3 - 20a + 4$. Ordenando el polinomio se obtiene:

$$\begin{array}{r} a^4 - 10a^3 + 29a^2 - 20a + 4 \\ \underline{- a^4} \quad \quad \quad \quad \quad \quad a^2 - 5a + 2 \\ \quad \quad - 10a^3 + 29a^2 \\ \quad \quad \underline{10a^3 - 25a^2} \\ \quad \quad \quad \quad 4a^2 - 20a + 4 \\ \quad \quad \quad \underline{- 4a^2 + 20a - 4} \\ \quad \quad \quad \quad \quad \quad \quad 0 \end{array}$$

Ejemplos

EXPLICACIÓN

Hallamos la raíz cuadrada de a^4 que es a^2 , este es el primer término de la raíz del polinomio. a^2 se eleva al cuadrado y da a^4 , este cuadrado se resta del primer término del polinomio y bajamos los dos términos siguientes $-10a^3 + 29a^2$; Hallamos el doble de a^2 que es $2a^2$.

Dividimos $-10a^3 + 2a^2 = -5a$, este es el segundo término de la raíz. Escribimos $-5a$ al lado de $2a^2$ y formamos el binomio $2a^2 - 5a$; este binomio lo multiplicamos por

394

BALDOR ÁLGEBRA

$-5a$ y nos da $-10a^3 + 25a^2$. Este producto lo restamos (cambiándole los signos) de $-10a^3 + 29a^2$; la diferencia es $4a^2$. Bajamos los dos términos siguientes y tenemos $4a^2 - 20a + 4$. Se duplica la parte de raíz hallada $2(a^2 - 5a) = 2a^2 - 10a$. Dividimos $4a^2 + 2a^2 = 2$, este es el tercer término de la raíz.

Este 2 se escribe al lado de $2a^2 - 10a$ y formamos el trinomio $2a^2 - 10a + 2$, que se multiplica por 2 y nos da $4a^2 - 20a + 4$. Este producto se resta (cambiando los signos) del residuo $4a^2 - 20a + 4$ y nos da 0.

PRUEBA

Se eleva al cuadrado la raíz cuadrada $a^2 - 5a + 2$ y si la operación está correcta debe dar la cantidad subradical.

- 2) Hallar la raíz cuadrada de

$$9x^6 + 25x^4 + 4 - 6x^5 - 20x^3 + 20x^2 - 16x$$

Ordenando el polinomio y aplicando la regla dada, se tiene:

$$\begin{array}{r} \sqrt{9x^6 - 6x^5 + 25x^4 - 20x^3 + 20x^2 - 16x + 4} & 3x^3 - x^2 + 4x - 2 \\ - 9x^6 & (6x^3 - x^2)(-x^2) = -6x^5 + x^4 \\ \hline - 6x^5 + 25x^4 & (6x^3 - 2x^2 + 4x)4x \\ 6x^3 - x^4 & = 24x^4 - 8x^3 + 16x^2 \\ \hline 24x^4 + 20x^3 + 20x^2 & (6x^3 - 2x^2 + 8x - 2)(-2) \\ - 24x^4 + 8x^3 - 16x^2 & = -12x^3 + 4x^2 - 16x + 4 \\ \hline - 12x^3 + 4x^2 - 16x + 4 & 0 \\ 12x^3 - 4x^2 + 16x - 4 & \end{array}$$

214

Hallar la raíz cuadrada de:

1. $16x^2 - 24xy^2 + 9y^4$
2. $25a^4 - 70a^3x + 49a^2x^2$
3. $x^4 + 6x^2 - 4x^3 - 4x + 1$
4. $4a^3 + 5a^2 + 4a^4 + 1 + 2a$
5. $29n^2 - 20n + 4 - 10n^3 + n^4$
6. $x^8 - 10x^5 + 25x^4 + 12x^3 - 60x^2 + 36$
7. $16a^6 + 49a^4 - 30a^2 - 24a^6 + 25$
8. $x^2 + 4y^2 + z^2 + 4xy - 2xz - 4yz$
9. $9 - 6x^3 + 2x^9 - 5x^5 + x^{12}$
10. $25x^8 - 70x^6 + 49x^4 + 30x^5 + 9x^2 - 42x^3$
11. $4a^4 + 8a^3b - 8a^2b^2 - 12ab^3 + 9b^4$
12. $x^6 - 2x^5 + 3x^4 + 1 + 2x - x^2$
13. $5x^4 - 6x^3 + x^2 + 16x^3 - 8x^2 - 8x + 4$
14. $x^8 + 6x^6 - 8x^5 + 19x^4 - 24x^3 + 46x^2 - 40x + 25$
15. $16x^8 - 8x^7 + x^8 - 22x^4 + 4x^5 + 24x^3 + 4x^2 - 12x + 9$
16. $9 - 36a + 42a^2 + 13a^4 - 2a^5 - 18a^3 + a^6$
17. $9x^4 - 24x^3 + 28x^4 - 22x^2 + 12x^2 - 4x + 1$
18. $16x^6 - 40x^5 + 73x^4 - 84x^3 + 66x^2 - 36x + 9$
19. $m^6 - 4m^5n + 4m^4n^2 + 4m^3n^4 - 8m^2n^5 + 4n^6$
20. $9x^6 - 6x^5y + 13x^4y^2 - 16x^3y^3 + 8x^2y^4 - 8xy^5 + 4y^6$

Elección de un algoritmo

- Pero esa no es la situación general
- Supongamos que ante un cierto problema tenemos varios algoritmos para emplear.
- ¿Que algoritmo elegir?.
- Queremos buenos algoritmos en algún sentido propio de cada usuario.
- El problema central que se quiere resolver es el siguiente:
- **Dado un algoritmo, determinar ciertas características que sirven para evaluar su rendimiento.**

Elección de un algoritmo

- Multiplicación de enteros
 - Algoritmo clásico
 - Algoritmo de multiplicación a la rusa (del campesino ruso)
 1. Escribir multiplicador y multiplicando en dos columnas
 2. Hasta que el número bajo el multiplicador sea 1
REPETIR:
 - a) Dividir el número bajo el multiplicador por 2, ignorando los decimales
 - b) Doblar el número bajo el multiplicando sumándolo a si mismo
 - c) Rayar cada fila en la que el numero bajo el multiplicador sea par, o añadir los números que queden en la columna bajo el multiplicando.

Elección de un algoritmo

- 1) Escribir multiplicador y multiplicando en dos columnas
- 2) Repetir las siguientes operaciones hasta que el número bajo el multiplicador sea 1:
 - a) Dividir el número bajo el multiplicador por 2, ignorando los decimales
 - b) Doblar el número bajo el multiplicando sumándolo a si mismo
 - c) Rayar cada fila en la que el número bajo el multiplicador sea par, o añadir los números que queden en la columna bajo el multiplicando.

Multiplicador	Multiplicando	Resultado	Resultado acumulado
45		19	19
19			
22	38	--	19
11	76	76	95
5	152	152	247
2	304	---	247
1	608	608	855

Elección de un algoritmo

- Posibles **criterios** para la elección son:
- La adaptabilidad del algoritmo a los computadores
- Su simplicidad y elegancia
- El costo económico que su confección y puesta a punto puede acarrear.
- La duración del tiempo consumido para llevar a cabo el algoritmo (esto puede expresarse en términos del número de veces que se ejecuta cada etapa).

Problemas y casos

- (19,45) es un **Caso del Problema** de multiplicar dos enteros positivos
- Los problemas mas interesantes incluyen una colección infinita de casos (¿ajedrez?).
- Un algoritmo debe trabajar correctamente en cualquier caso del problema en cuestión.
- Para demostrar que un algoritmo es incorrecto, solo necesitamos encontrar un caso del problema que no produzca la respuesta correcta.
- La diferencia que existe entre **algoritmo** y **programa** es que cualquier sistema de computo real tiene un límite sobre el tamaño de los casos que puede manejar. Sin embargo, este límite no puede atribuirse al algoritmo que vayamos a usar.

Problemas y casos

- Formalmente:
 - El tamaño de un caso x es el número de bits necesarios para representar el caso en un computador usando un código precisamente definido y razonablemente compacto.
- Informalmente:
 - El tamaño de un caso es cualquier entero que, de algún modo, mida el número de componentes del caso.
- Ejemplos:
 - Ordenación: longitud del array, Matrices: Número de filas y columnas, Grafos: Número de vértices y arcos

Diferentes tipos de casos

- El tiempo consumido por un algoritmo puede variar mucho entre dos casos diferentes del mismo tamaño.
- Consideremos dos algoritmos de ordenación elementales: **Insercion** y **Seleccion**.

Procedimiento Insercion (T[1..n])

```
for i := 2 to n do
    x:= T[i]; j := i-1
    while j > 0 and x < T[j] do
        T[j+1] := T[j]
        j := j-1
    T[j+1] := x
```

Procedimiento Seleccion

```
(T[1..n])
for i:= 1 to n-1 do
    minj := i; minx := T[i]
    for j := i+1 to n do
        if T[j] < minx then minj := j
        minx := T[j]
    T[minj] := T[i];
    T[i] := minx
```

Diferentes tipos de casos

- U y V dos arrays de n elementos: U ordenado en orden ascendente y V en orden descendente.
- **Comportamiento de Seleccion:**
 - Indiferente de U o V, el tiempo requerido para ordenar con Seleccion no varia en mas de un 15%.
- **Comportamiento de Insercion:**
 - Insercion(U) consume menos de 1/5 de segundo si U es un array de 5.000 elementos
 - Insercion(V) consume tres minutos y medio si V es un array de 5.000 elementos.

Si pueden darse esas diferencias, ¿Podremos hablar del tiempo consumido por un algoritmo solo en función del tamaño del caso?

Diferentes tipos de casos

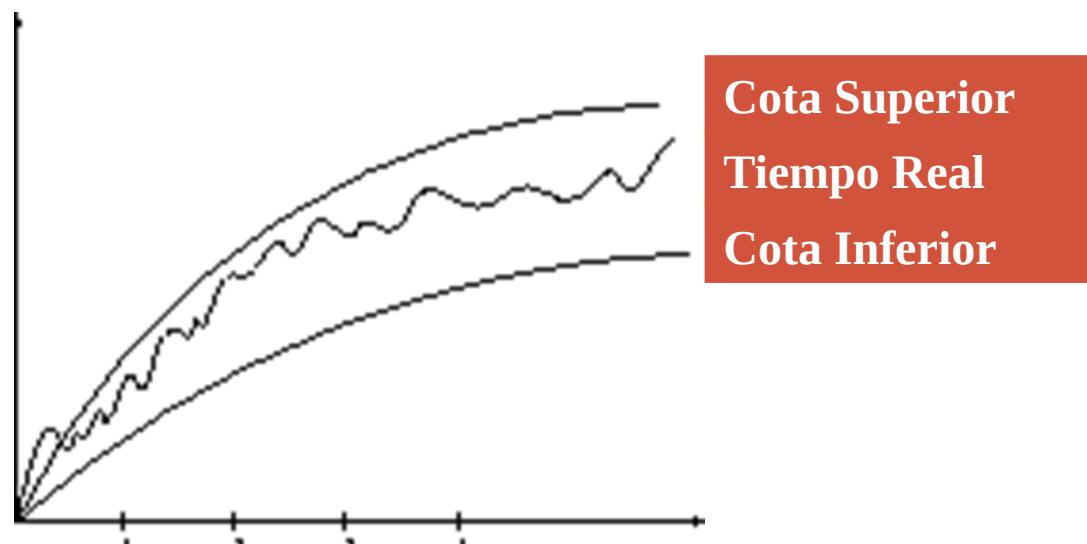
- El tiempo que se obtiene para V da el máximo tiempo que se puede emplear para resolver el problema: V describe el **peor caso**.
- El peor caso es útil cuando necesitamos una garantía total acerca de lo que durará la ejecución de un programa. El tiempo del peor caso, para un n dado, se calcula a partir del caso de tamaño n que tarda más.
- Si el problema se ha de resolver en muchos casos distintos, es más informativo el tiempo del **caso promedio**: la media de los tiempos de todos los posibles casos del mismo tamaño (**¡hay que conocer la distribución de probabilidad asociada a los casos!**)
- U, sin embargo, describe el **mejor caso** de este problema, es decir, el caso sobre el que se tarda menos tiempo

Diferentes tipos de casos

- Consideramos el **Tiempo de Ejecución**,
- **Tiempo del Peor Caso:**
 - La función definida por el *máximo* número de etapas que se realizan en cualquier caso de tamaño n
- **Tiempo del Mejor Caso:**
 - La función definida por el *mínimo* número de etapas que se realizan en cualquier caso de tamaño n
- **Tiempo del Caso promedio:**
 - La función definida por el número *medio* de etapas que se realizan en cualquier caso de tamaño n
 - El caso promedio no existe como tal

Diferentes tipos de casos

- Es difícil estimar exactamente el tiempo de ejecución
 - El mejor caso depende del input
 - El caso promedio es difícil de calcular
 - Por tanto generalmente nos referiremos al Peor Caso
 - Es fácil de calcular
 - Suele aproximarse al tiempo de ejecución real



ALGORÍTMICA

Capítulo 1: La Eficiencia de los Algoritmos

Tema 2: Tiempo de ejecución. Notaciones para la eficiencia de los algoritmos

- La eficiencia de los algoritmos. Métodos para evaluar la eficiencia. Notaciones O y Ω
- La notacion asintotica de Brassard y Bratley
- Análisis teórico del tiempo de ejecución de un algoritmo
- Análisis práctico del tiempo de ejecución de un algoritmo
- Análisis de programas con llamadas a procedimientos
- Análisis de procedimientos recursivos
- Algunos ejemplos prácticos



La eficiencia de los algoritmos

- ¿En que unidad habrá que expresar la eficiencia de un algoritmo?.
- Independientemente de cual sea la medida que nos la evalúe, hay tres métodos de calcularla:
- a) El enfoque empírico (o a posteriori), es dependiente del agente tecnológico usado.
- b) El enfoque teórico (o a priori), no depende del agente tecnológico empleado, sino en cálculos matemáticos.
- c) El enfoque híbrido, la forma de la función que describe la eficiencia del algoritmo se determina teóricamente, y entonces cualquier parámetro numérico que se necesite se determina empíricamente sobre un programa y una máquina particulares.

La eficiencia de los algoritmos

- la selección de la unidad para medir la eficiencia de los algoritmos la vamos a encontrar a partir del denominado **Principio de Invariancia:**
- Dos implementaciones diferentes de un mismo algoritmo no difieren en eficiencia mas que, a lo sumo, en una constante multiplicativa.
- Si 2 implementaciones consumen $t_1(n)$ y $t_2(n)$ unidades de tiempo, respectivamente, en resolver un caso de tamaño n , entonces siempre existe una constante positiva c tal que $t_1(n) \leq ct_2(n)$, siempre que n sea suficientemente grande.
- Este Principio es independiente del agente tecnológico usado:
 - Un cambio de máquina puede permitirnos resolver un problema 10 o 100 veces mas rápidamente, pero solo un cambio de algoritmo nos dará una mejora de cara al aumento del tamaño de los casos.

La eficiencia de los algoritmos

- Parece por tanto oportuno referirnos a la eficiencia teórica de un algoritmo en términos de tiempo.
- Algo que conocemos de antemano es el denominado **Tiempo de Ejecución** de un programa, que depende de,
 - a) **El input del programa**
 - b) La calidad del código que genera el compilador que se use para la creación del programa,
 - c) La naturaleza y velocidad de las instrucciones en la máquina que se esté empleando para ejecutar el programa,
 - d) La **complejidad en tiempo** del algoritmo que subyace en el programa.
- El tiempo de ejecución no depende directamente del input, sino del tamaño de este
- $T(n)$ notará el tiempo de ejecución de un programa para un input de tamaño n , y también el del algoritmo en el que se basa.

La eficiencia de los algoritmos

- No habrá unidad para expresar el tiempo de ejecución de un algoritmo. Usaremos una constante para acumular en ella todos los factores relativos a los aspectos tecnológicos.
- Diremos que un algoritmo consume un tiempo de orden $t(n)$, si existe una constante positiva c y una implementación del algoritmo capaz de resolver cualquier caso del problema en un tiempo acotado superiormente por $ct(n)$ segundos, donde n es el tamaño del caso considerado.
- El uso de segundos es mas que arbitrario, ya que solo necesitamos cambiar la constante (**oculta**) para expresar el tiempo en días o años.

La eficiencia de los algoritmos

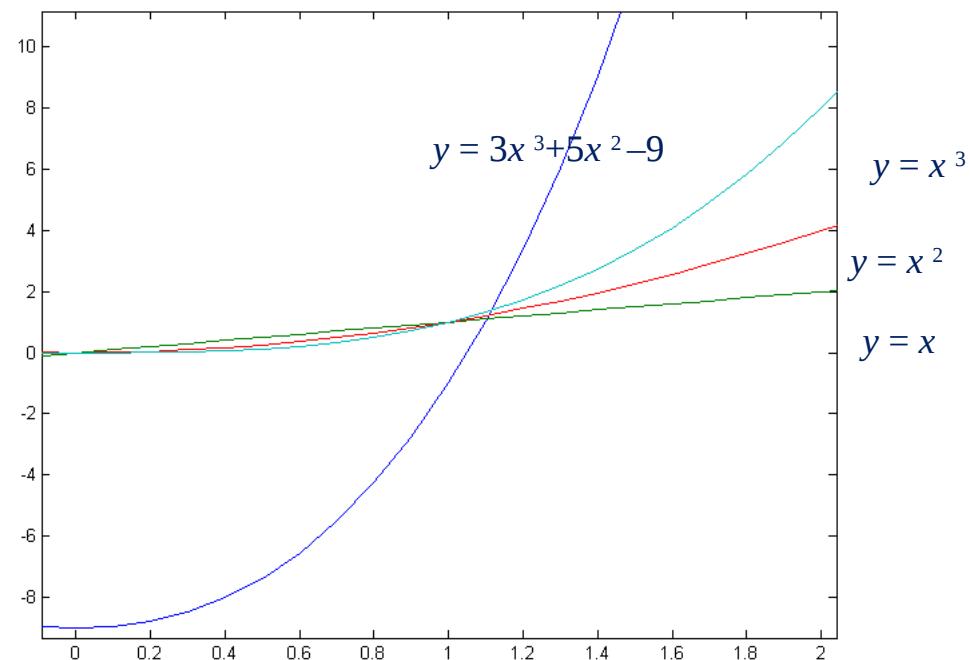
- Sean dos algoritmos cuyas implementaciones, consumen n^2 días y n^3 segundos para resolver un caso de tamaño n .
- Solo en casos que requieran mas de 20 millones de años para resolverlos, es donde el algoritmo cuadrático puede ser mas rápido que el algoritmo cúbico.
- El primero es asintóticamente mejor que el segundo: su eficiencia teórica es mejor en todos los casos grandes
- Desde un punto de vista práctico el alto valor que tiene la constante **oculta** recomienda el empleo del cúbico.

Notación Asintótica O , Ω y Θ

- Para poder comparar los algoritmos empleando los tiempos de ejecución, y las constantes ocultas, se emplea la denominada **notación asintótica**
- La notación asintótica sirve para comparar funciones.
- Es útil para el cálculo de la eficiencia teórica de los algoritmos, es decir para calcular el tiempo que consume una implementación de un algoritmo.

Notación Asintótica O , Ω y Θ

- La notación asintótica captura la conducta de las funciones para valores grandes de x .
- P. ej., el término dominante de $3x^3+5x^2-9$ es x^3 .
- Para x pequeños no está claro por que x^3 domina mas que x^2 o incluso que x ; pero conforme aumenta x , los otros términos se hacen insignificantes y solo x^3 es relevante



Definición Formal para O

- Intuitivamente una función $f(n)$ está asintóticamente dominada por $g(n)$ si cuando multiplicamos $g(n)$ por alguna constante lo que se obtiene es realmente mayor que $f(n)$ para los valores grandes de n .
- Formalmente:
- Sean f y g funciones definidas de \mathbf{N} en $\mathbf{R}_{\geq 0}$. Se dice que f es de orden g , que se nota $O(g(n))$, si existen dos constantes positivas C y k tales que

$$\forall n \geq k, f(n) \leq C \cdot g(n)$$

es decir, pasado k , f es menor o igual que un múltiplo de g .

Confusiones usuales

- Es verdad que $3x^3 + 5x^2 - 9 = O(x^3)$ como demostraremos, pero también es verdad que:
 - $3x^3 + 5x^2 - 9 = O(x^4)$
 - $x^3 = O(3x^3 + 5x^2 - 9)$
 - $\sin(x) = O(x^4)$
- El uso de la notación O en Algorítmica supone mencionar solo el término mas dominante.

“El tiempo de ejecucion es $O(x^{2.5})$ ”
- Matemáticamente la notación O tiene mas aplicaciones (comparación de funciones)

Ejemplo de notación O

- Probar que $3n^3 + 5n^2 - 9 = O(n^3)$.
- A partir de la experiencia de la gráfica que vimos, basta que tomemos $C = 5$.
- Veamos para qué valor de k se verifica

$$3n^3 + 5n^2 - 9 \leq 5n^3 \text{ para } n > k:$$

- Ha de verificarse: $5n^2 \leq 2n^3 + 9$
- ¿A partir de qué k se verifica $5n^2 \leq n^3$?
- ¡ $k = 5$!
- Así para $n > 5$, $5n^2 \leq n^3 \leq 2n^3 + 9$
- Solución: $C = 5$, $k = 5$ (no única!)

Un ejemplo negativo de O

- $x^4 \neq O(3x^3 + 5x^2 - 9)$:
- Probar que no pueden existir constantes C, k tales que pasado k , siempre se verifique que $C(3x^3 + 5x^2 - 9) \geq x^4$.
- Esto es fácil de ver con límites:

$$\begin{aligned}\lim_{x \rightarrow \infty} \frac{x^4}{C(3x^3 + 5x^2 - 9)} &= \lim_{x \rightarrow \infty} \frac{x}{C(3 + 5/x - 9/x^3)} \\ &= \lim_{x \rightarrow \infty} \frac{x}{C(3 + 0 - 0)} = \frac{1}{3C} \cdot \lim_{x \rightarrow \infty} x = \infty\end{aligned}$$

- Así que no hay problema con C porque x^4 siempre es mayor que $C(3x^3 + 5x^2 - 9)$

La notación O y los límites

- Los límites puede ayudar a demostrar relaciones en notación O :
- **LEMA:** Si existe el límite cuando $n \rightarrow \infty$ de $|f(n)/g(n)|$ (no es infinito) entonces $f(n) = O(g(n))$.
- **Ejemplo:** $3n^3 + 5n^2 - 9 = O(n^3)$.

$$\lim_{x \rightarrow \infty} \frac{x^3}{3x^3 + 5x^2 - 9} = \lim_{x \rightarrow \infty} \frac{1}{3 + 5/x - 9/x^3} = \frac{1}{3}$$

Notaciones Ω y Θ

- Ω es exactamente lo contrario de O :

$$f(n) = \Omega(g(n)) \leftrightarrow g(n) = O(f(n))$$

- **DEF:** Sean f y g funciones definidas de \mathbf{N} en $\mathbf{R}_{\geq 0}$. Se dice que f es $\Omega(g(n))$ si existen dos constantes positivas C y k tales que

$$\forall n \geq k, f(n) \geq C \cdot g(n)$$

Así Ω dice que asintóticamente $f(n)$ domina a $g(n)$

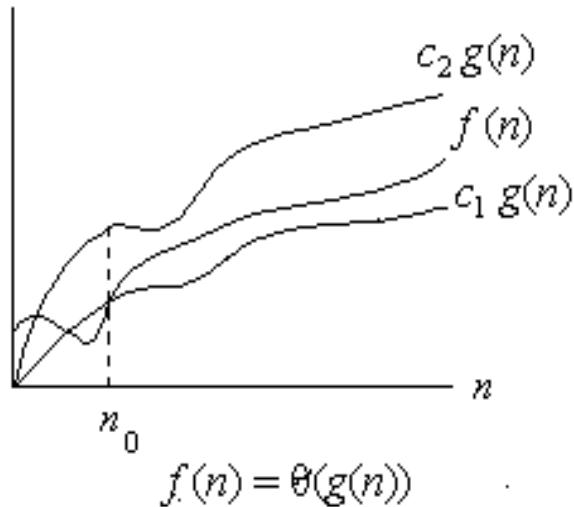
Notaciones Ω y Θ

- Θ , que se conoce como el “orden exacto”, establece que cada función domina a la otra, de modo que son asintóticamente equivalentes, es decir

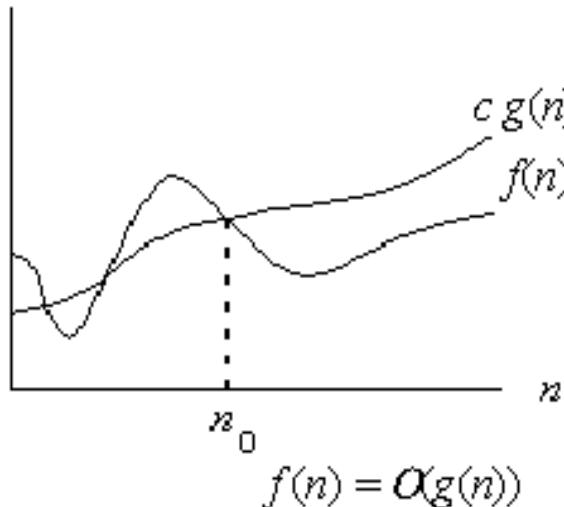
$$\begin{aligned} f(n) &= \Theta(g(n)) \\ &\leftrightarrow \\ f(n) &= O(g(n)) \quad \wedge \quad f(n) = \Omega(g(n)) \end{aligned}$$

- Sinónimo de $f = \Theta(g)$, es “ f **es de orden exacto** g ”

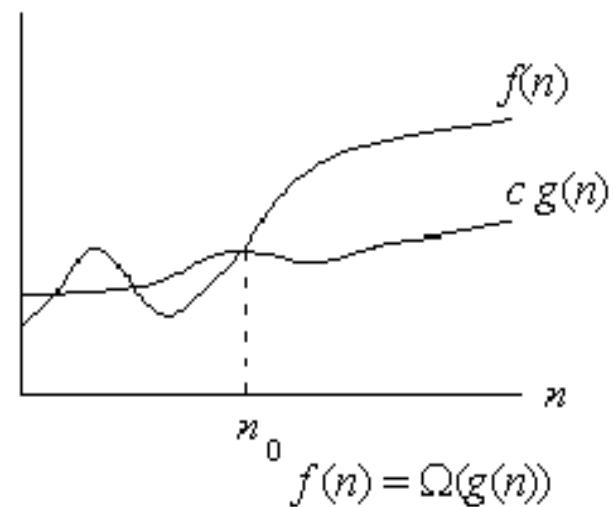
Ejemplos gráficos



(a)



(b)



(c)

La dictadura de la Tasa de Crecimiento

- Si un algoritmo tiene un tiempo de ejecución $O(f(n))$, a $f(n)$ se le llama **Tasa de Crecimiento**.
- Suponemos que los algoritmos podemos evaluarlos comparando sus tiempos de ejecución, despreciando sus constantes de proporcionalidad.
- Así, un algoritmo con tiempo de ejecución $O(n^2)$ es mejor que uno con tiempo de ejecución $O(n^3)$.
- Es posible que a la hora de las implementaciones, con una combinación especial compilador-máquina, el primer algoritmo consuma $100n^2$ milisg., y el segundo $5n^3$ milisg, entonces ¿no podría ser mejor el algoritmo cúbico que el cuadrático?.

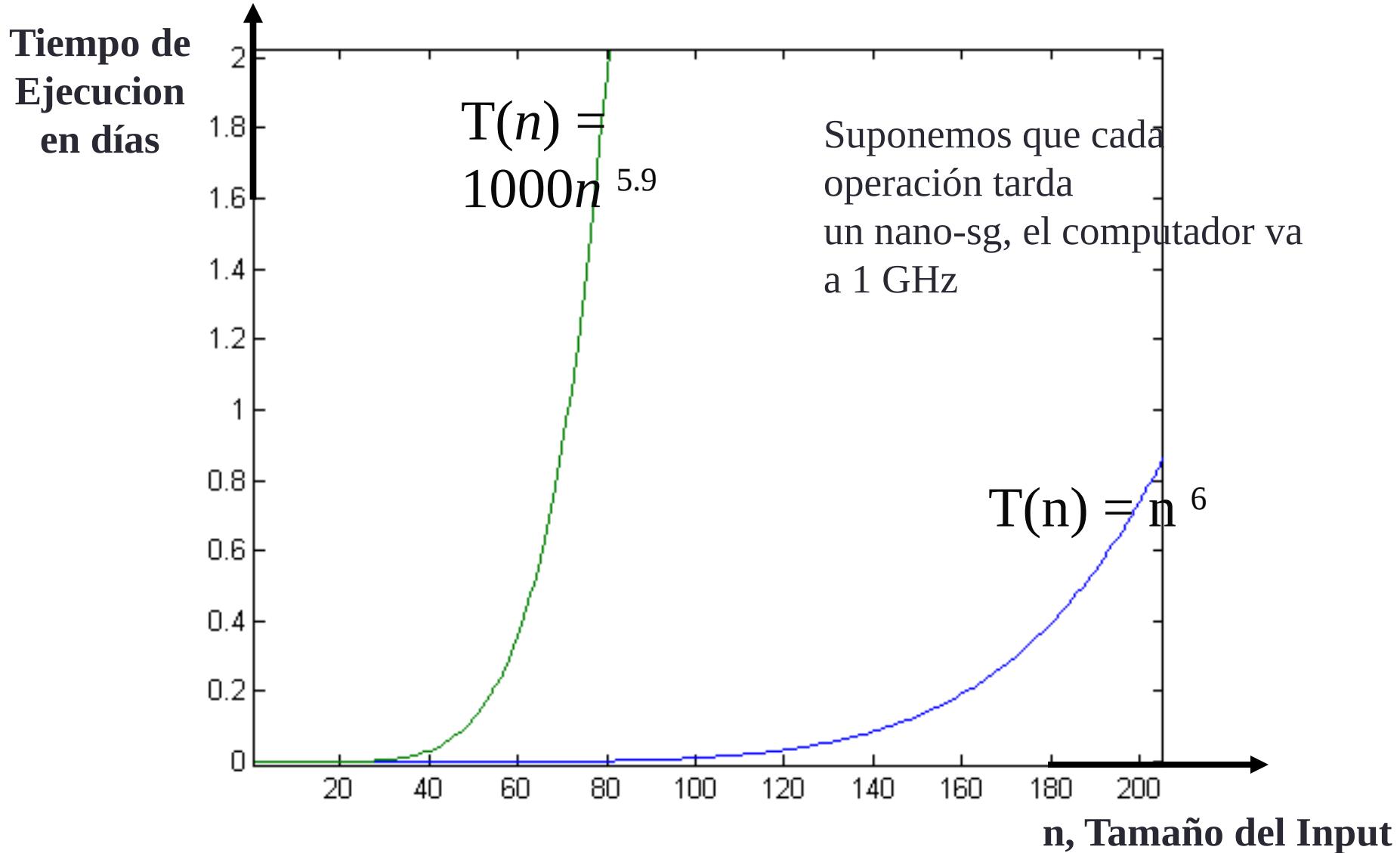
La dictadura de la Tasa de Crecimiento

- La respuesta está en función del tamaño de los inputs que se esperan procesar.
- Para inputs de tamaños $n < 20$, el algoritmo cúbico será mas rápido que el cuadrático.
- Si el algoritmo se va a usar con inputs de gran tamaño, realmente podríamos preferir el programa cúbico. Pero cuando n se hace grande, la razón de los tiempos de ejecución, $5n^3 / 100n^2 = n/20$, se hace arbitrariamente grande.
- Así cuando el tamaño del input aumenta, el algoritmo cúbico tardará mas que el cuadrático.
- **¡Ojo!** que puede haber funciones incomparables

Una reflexión

- La notación O funciona bien en general, pero en la práctica no siempre actúa correctamente.
- Consideremos las tasas n^6 vs. $1000n^{5.9}$.
Asintóticamente, la segunda es mejor
- A esto se le suele dar mucho crédito en las revistas científicas.
- Ahora bien...

Una reflexión



Una reflexión

$1000n^{5.9}$ solo iguala a n^6 cuando

$$1000n^{5.9} = n^6$$

$$1000 = n^{0.1}$$

$$n = 1000^{10} = 10^{30} \text{ operaciones}$$

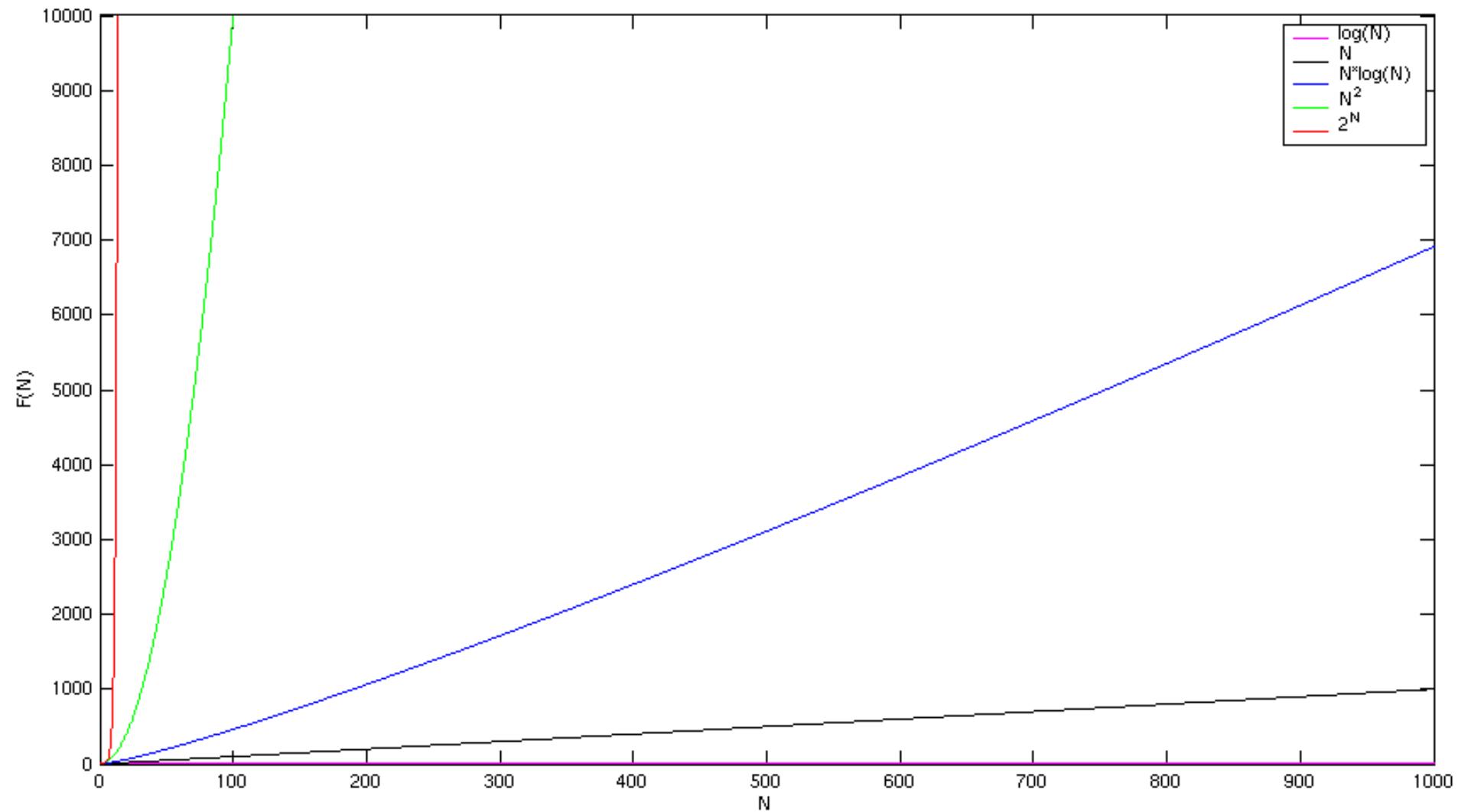
$$= 10^{30}/10^9 = 10^{21} \text{ segundos} \qquad \approx$$

$$10^{21}/(3 \times 10^7) \approx 3 \times 10^{13} \text{ años}$$

$$\approx 3 \times 10^{13}/(2 \times 10^{10})$$

≈ 1500 veces el tiempo de vida estimado del universo!

Diferencias entre tasas



Ejemplos

Ordenar las siguientes tasas de crecimiento de menor a mayor, y agrupar todas las funciones que son respectivamente Θ unas de otras:

$$x + \sin x, \ln x, x + \sqrt{x}, \frac{1}{x}, 13 + \frac{1}{x}, 13 + x, e^x, x^e, x^x$$

$$(x + \sin x)(x^{20} - 102), x \ln x, x(\ln x)^2, \lg_2 x$$

Ejemplos

$$1. \frac{1}{x}$$

$$2. 13 + \frac{1}{x}$$

$$3. \ln x \lg_2 x \text{ (cambiando de base)}$$

$$4. x + \sin x, x + \sqrt{x}, 13 + x$$

5.

$$6. x \ln x$$

$$7. x(\ln x)^2$$

$$8. x^e$$

$$9. (x + \sin x)(x^{20} - 102)$$

$$10. e^x$$

$$x^x$$

Ejemplos

Demostrar que $f(n) = \frac{1}{2}n^2 - 3n = \Theta(n^2)$

Debemos encontrar c_1 y c_2 tales que

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

Dividiendo ambos miembros por n^2 tenemos

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

Para

$$n_0 \geq 7, \quad \frac{1}{2}n^2 - 3n = \Theta(n^2)$$

Ejemplos

$$f(n) = 3n^2 - 2n + 5 = \Theta(n^2)$$

Ya que :

$$3n^2 - 2n + 5 = \Omega(n^2)$$

$$3n^2 - 2n + 5 = O(n^2)$$

Otras notaciones estándar

- Logaritmos:

$$\lg n = \log_2 n$$

$$\ln n = \log_e n$$

$$\log^k n = (\log n)^k$$

$$\lg \lg n = \lg(\lg n)$$

Otras notaciones estándar

- Logaritmos:

Para cualesquiera números reales $a > 0$, $b > 0$, $c > 0$ y n

$$a = b^{\log_b a}$$

$$\log_c(ab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b(1/a) = -\log_b a$$

$$a^{\log_b c} = c^{\log_b a}$$

$$\log_b a = \frac{1}{\log_a b}$$

Otras notaciones estándar

- Factoriales

Para $n \geq 0$ la Aproximación de Stirling:

$$n! = \sqrt{2\pi n} \left| \frac{n}{e} \right|^n \left| 1 + \Theta \left(\frac{1}{n} \right) \right|$$

$$n! = o(n^n)$$

$$n! = \omega(2^n)$$

$$\lg(n!) = \Theta(n \lg n)$$

Notación asintótica minúscula

- Una función $f(n)$ es $o(g(n))$ si existen constantes positivas c y n_0 tales que

$$f(n) < c g(n) \quad \forall n \geq n_0$$

- Una función $f(n)$ es $\omega(g(n))$ si existen constantes positivas c y n_0 tales que

$$c g(n) < f(n) \quad \forall n \geq n_0$$

- Intuitivamente,

- $o()$ es como <
- $O()$ es como \leq
- $\omega()$ es como >
- $\Omega()$ es como \geq
- $\Theta()$ es como =

Notacion asintotica de Brassard y Bratley

- Sea $f:N \rightarrow R^*$ una función arbitraria. Definimos,

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

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

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

- La condición **$\exists n_0 \in N: \forall n \geq n_0$** puede evitarse (?)

- Probar para funciones arbitrarias f y $g: N \rightarrow R^*$ que,

a) $O(f(n)) = O(g(n))$ ssi $f(n) \in O(g(n))$ y $g(n) \in O(f(n))$

b) $O(f(n)) \subset O(g(n))$ ssi $f(n) \in O(g(n))$ y $g(n) \notin O(f(n))$

c) $f(n) \in O(g(n))$ si y solo si $g(n) \in \Omega(f(n))$

Notacion asintotica de Brassard y Bratley

- **Caso de diversos parámetros**

Sea $f:N \rightarrow R^*$ una función arbitraria. Definimos,

$$O(f(m,n)) = \{t: N \times N \rightarrow R^* / \exists c \in R^+, \exists m_0, n_0 \in N:$$

$$\forall m \geq m_0 \ \forall n \geq n_0 \Rightarrow t(m,n) \leq cf(m,n)\}$$

¿Puede eliminarse ahora que

$$\exists m_0, n_0 \in N: m \geq m_0 \ \forall n \geq n_0 ?$$

- **Notación asintótica condicional**

$$O(f(n)/P(n)) = \{t: N \rightarrow R^* / \exists c \in R^+, \exists n_0 \in N: \forall n \geq n_0 P \Rightarrow t(n) \leq cf(n)\}$$

donde P es un predicado booleano

Excepciones (1)

- Si un algoritmo se va a usar solo unas pocas veces, el costo de escribir el programa y corregirlo domina todos los demás, por lo que su tiempo de ejecución raramente afecta al costo total. En tal caso lo mejor es escoger aquel algoritmo que se mas fácil de implementar.
- Si un programa va a funcionar solo con inputs pequeños, la tasa de crecimiento del tiempo de ejecución puede que sea menos importante que la constante oculta.

Excepciones (2)

- Un algoritmo complicado, pero eficiente, puede no ser deseable debido a que una persona distinta de quien lo escribió, podría tener que mantenerlo mas adelante.
- En el caso de algoritmos numéricos, la exactitud y la estabilidad son tan importantes, o mas, que la eficiencia.
- Si solo hay un algoritmo para resolver un problema, la cuestión de la eficiencia no tiene demasiado interés.

Cálculo de la Eficiencia: Reglas teóricas

- Supongamos, en primer lugar, que $T^1(n)$ y $T^2(n)$ son los tiempos de ejecución de dos segmentos de programa, P^1 y P^2 , que $T^1(n)$ es $O(f(n))$ y $T^2(n)$ es $O(g(n))$. Entonces el tiempo de ejecución de P^1 seguido de P^2 , es decir $T^1(n) + T^2(n)$, es $O(\max(f(n), g(n)))$.
- Por la propia definición se tiene

$$\exists c_1, c_2 \in \mathbb{R}, \exists n_1, n_2 \in \mathbb{N}: \forall n \geq n_1 \Rightarrow T^1(n) \leq c_1 f(n),$$
$$\forall n \geq n_2 \Rightarrow T^2(n) \leq c_2 g(n)$$

- Sea $n_0 = \max(n_1, n_2)$. Si $n \geq n_0$, entonces

$$T^1(n) + T^2(n) \leq c_1 f(n) + c_2 g(n)$$

luego,

$$\forall n \geq n_0 \Rightarrow T^1(n) + T^2(n) \leq (c_1 + c_2) \text{Max}(f(n), g(n))$$

Cálculo de la Eficiencia: Reglas teóricas

- Si $T^1(n)$ y $T^2(n)$ son los tiempos de ejecución de dos segmentos de programa, P^1 y P^2 , $T^1(n)$ es $O(f(n))$ y $T^2(n)$ es $O(g(n))$, entonces $T^1(n) \cdot T^2(n)$ es
$$O(f(n) \cdot g(n))$$
- La demostración es trivial sin mas que considerar el producto de las constantes.
- De esta regla se deduce que $O(cf(n))$ es lo mismo que $O(f(n))$ si c es una constante positiva, así que por ejemplo $O(n^2/2)$ es lo mismo que $O(n^2)$.

Cálculo de la Eficiencia: Reglas teóricas

- Cualquier polinomio es Θ de su mayor término
 - EG: $x^4/100000 + 3x^3 + 5x^2 - 9 = \Theta(x^4)$
- La suma de dos funciones es *O de la mayor*
 - EG: $x^4 \ln(x) + x^5 = O(x^5)$
- Las constantes no nulas son irrelevantes:
 - EG: $17x^4 \ln(x) = O(x^4 \ln(x))$
- El producto de dos funciones es *O del producto*
 - EG: $x^4 \ln(x) \cdot x^5 = O(x^9 \cdot \ln(x))$

Cálculo de la Eficiencia: Reglas prácticas

- **Caso de procedimientos no recursivos,**
 - analizamos aquellos procedimientos que no llaman a ningún otro procedimiento,
 - entonces evaluamos los tiempos de ejecución de los procedimientos que llaman a otros procedimientos cuyos tiempos de ejecución ya han sido determinados.
 - procedemos de esta forma hasta que hayamos evaluado los tiempos de ejecución de todos los procedimientos.

Cálculo de la Eficiencia: Reglas prácticas

- **Caso de funciones**
- las llamadas a funciones suelen aparecer en asignaciones o en condiciones, y además puede haber varias en una sentencia de asignación o en una condición.
- Para una sentencia de asignación o de escritura que contenga una o mas llamadas a funciones, tomaremos como cota superior del tiempo de ejecución la suma de las cotas de los tiempos de ejecución de cada llamada a funciones.

Cálculo de la Eficiencia: Reglas prácticas

- **Análisis de procedimientos recursivos**
- Requiere que asociemos con cada procedimiento P en el programa, un tiempo de ejecución desconocido $T^P(n)$ que define el tiempo de ejecución de P en función de n, tamaño del argumento de P.
- Entonces establecemos una definición inductiva, llamada **una relación de recurrencia**, para $T^P(n)$, que relaciona $T^P(n)$ con una función de la forma $T^Q(k)$ de los otros procedimientos Q en el programa y los tamaños de sus argumentos k.
- Si P es directamente recursivo, entonces la mayoría de los Q serán el mismo P.

Cálculo de la Eficiencia: Reglas prácticas

- **Análisis de procedimientos recursivos**
- Cuando se sabe como se lleva a cabo la recursión en función del tamaño de los casos que se van resolviendo, podemos considerar dos casos:
 - El tamaño del argumento es lo suficientemente pequeño como para que P no haga llamadas recursivas. Este caso corresponde a la base de una definición inductiva sobre $T^P(n)$.
 - El tamaño del argumento es lo suficientemente grande como para que las llamadas recursivas puedan hacerse (con argumentos menores). Este caso se corresponde a la etapa inductiva de la definición de $T^P(n)$.

Cálculo de la Eficiencia: Reglas prácticas

- Análisis de procedimientos recursivos

Ejemplo:

Funcion Factorial (n: integer)

Begin

If n < = 1 Then

 Fact := 1

Else

 Fact := n x Fact (n-1)

End

Base: $T(1) = O(1)$

Induccion: $T(n) = O(1) + T(n-1)$, $n > 1$

Cálculo de la Eficiencia: Reglas prácticas

$$T(1) = O(1)$$

$$T(n) = O(1) + T(n-1), \quad n > 1$$

$$\Leftrightarrow$$

$$T(n) = d, \quad n \leq 1$$

$$T(n) = c + T(n-1), \quad n > 1$$

Para $n > 2$, como $T(n-1) = c + T(n-2)$, podemos expandir $T(n)$ para obtener,

$$T(n) = 2c + T(n-2), \quad \text{si } n > 2$$

Volviendo a expandir $T(n-2)$, $T(n) = 3c + T(n-3)$, si $n > 3$
y así sucesivamente. En general

$$T(n) = ic + T(n-i), \quad \text{si } n > i$$

y finalmente cuando $i = n-1$, $T(n) = c(n-1) + T(1) = c(n-1) + d$

De donde concluimos que **$T(n)$ es $O(n)$.**

Cálculo de la Eficiencia: Reglas prácticas

Análisis de procedimientos recursivos Ejemplo

```
Funcion Ejemplo (L: lista; n: integer): Lista
L1,L2 : Lista
Begin
    If n = 1
        Then Return (L)
    Else begin
        Partir L en dos mitades L1,L2 de longitudes n/2
        Return (Ejem(Ejemplo(L1,n/2), Ejemplo(L2,n/2)))
    end
End
```

$$\begin{aligned} T(n) &= c_1 && \text{si } n = 1 \\ T(n) &= 2T(n/2) + c_2 n && \text{si } n > 1 \end{aligned}$$

Cálculo de la Eficiencia: Reglas prácticas

Análisis de procedimientos recursivos

Ejemplo

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

- La expansión de la ecuación no es posible
- Solo puede aplicarse cuando n es par (Brassard-Bratley)
- Siempre podemos suponer que $T(n)$ esta entre $T(2^i)$ y $T(2^{i+1})$ si n se encuentra entre 2^i y 2^{i+1} .
- Podríamos sustituir el termino $2T(n/2)$ por
 - $T((n+1)/2) + T((n-1)/2)$ para n > 1 impares.
- Solo si necesitamos conocer **la solución exacta**

Ejemplos prácticos: ordenación

- Algoritmos elementales: Inserción, Selección, ... $O(n^2)$
- Otros (Quicksort, heapsort, ...) $O(n \log n)$
- N pequeño: diferencia inapreciable.
- Quicksort es ya casi el doble de rápido que el de inserción para $n = 50$ y el triple de rápido para $n = 100$
- Para $n = 1000$, inserción consume mas de tres segundos, y quicksort menos de un quinto de segundo.
- Para $n = 5000$, inserción necesita minuto y medio en promedio, y quicksort poco mas de un segundo.
- En 30 segundos, quicksort puede manejar 100.000 elementos; se estima que el de inserción podría consumir nueve horas y media para finalizar la misma tarea.

Ejemplos: Enteros grandes

- Algoritmo clásico: $O(mn)$
- Algoritmo de multiplicación a la rusa: $O(mn)$
- Otros son $O(nm^{\log(3/2)})$, o aproximadamente $O(nm^{0.59})$, donde n es el tamaño del mayor operando y m es el tamaño del menor.
- Si ambos operandos son de tamaño n, el algoritmo consume un tiempo en el orden de $n^{1.59}$, que es preferible al tiempo cuadrático consumido por el algoritmo clásico.
- La diferencia es menos espectacular que antes

Ejemplos: Determinantes

$$M = (a_{ij}), i = 1, \dots, n; j = 1, \dots, n$$

- El determinante de M , $\det(M)$, se define recursivamente: Si $M[i,j]$ nota la submatriz $(n-1) \times (n-1)$ obtenida de la M eliminando la i -esima fila y la j -esima columna, entonces

$$\det(M) = \sum_{i=1..n} (-1)^{j+1} a_{ij} \det(M[1,j])$$

si $n = 1$, el determinante se define por $\det(M) = a_{11}$.

- Algoritmo recursivo $O(n!)$, Algoritmo de Gauss-Jordan $O(n^3)$
- El algoritmo de Gauss-Jordan encuentra el determinante de una matriz 10×10 en $1/100$ segundos; alrededor de 5.5 segundos con una matriz 100×100
- El algoritmo recursivo consume un tiempo proporcional a mas de 20 seg. con una matriz 5×5 y a 10 minutos con una 10×10 . Se estima que consumiría mas de 10 millones de años para una matriz 20×20

El algoritmo de Gauss-Jordan tardaría $1/20$ de segundo

Ejemplos: Calculo del m.c.d.

funcion mcd(m,n),

i = min(m,n) +1

repetir i = i - 1 hasta que i divide tanto a m como a n exactamente

devolver i

funcion Euclides (m,n)

mientras m > 0 hacer

t = m

m = n mod m

n := t

devolver n

El tiempo consumido por este algoritmo es proporcional a la diferencia entre el menor de los dos argumentos y su máximo común divisor. Cuando m y n son de tamaño similar y primos entre si, toma por tanto un tiempo lineal (n).

Como las operaciones aritméticas son de costo unitario, este algoritmo consume un tiempo en el orden del logaritmo de sus argumentos, aun en el peor de los casos, por lo que es mucho mas rápido que el precedente

Ejemplos: Sucesión de Fibonacci

$$f_0 = 0; f_1 = 1, f_n = f_{n-1} + f_{n-2}, n \geq 2$$

los primeros 10 términos son 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.



Leonardo Pisano es más conocido por su apodo Fibonacci. Nació en Pisa, pero fue educado en África del Norte donde su padre ocupaba un puesto diplomático.

La sucesión de Fibonacci posee una gran variedad de propiedades y curiosidades.

El cálculo del término n -ésimo de esa sucesión siempre ha sido un test para los algoritmos.

Ejemplos: Sucesión de Fibonacci

- De Moivre probó la siguiente formula,

$$f_n = (1/5)^{1/2} [\phi^n - (-\phi)^{-n}]$$

donde $\phi = (1 + \sqrt{5})/2$ es la razón áurea.

- Como $\phi^{-1} < 1$, el término $(-\phi)^{-n}$ puede despreciarse cuando n es grande, lo que significa que el valor de f_n es $O(\phi^n)$
- Pero esta formula es de poca ayuda para el cálculo exacto de f_n
- ¿Por qué?



Ejemplos: Sucesión de Fibonacci

```
funcion fib1 (n)
if n < 2 then return n
else return fib1(n-1) + fib1(n-2)
```

$$t_1(n) = \phi^{n-20} \text{ segundos}$$

```
function fib2(n)
i := 1; j := 0
for k := 1 to n do j := i + j; i := j - i
return j
```

$$t_2(n) = 15n \text{ microsgs}$$

```
function fib3(n)
i := 1; j := 0; k := 0; h := 1
while n > 0 do
    if n es impar then t := jh; j := ih +jk + t; i := ik + t
                           t := h ; h := 2kh + t; k := k + t; n := n div 2
    return j
```

$$t_3(n) = (1/4)\log n \text{ miliseg.}$$

Ejemplos: Algoritmos de Fuerza Bruta

- La Fuerza Bruta es una enfoque directo para resolver un problema, que se basa exclusivamente en el planteamiento del problema y en las definiciones y conceptos que intervienen en el mismo.
- No recurre a algoritmos, métodos, procedimientos o técnicas que no vayan incorporadas en el problema en sí mismo.
- **Ejemplos**
 - ✖ Cálculo de a^n ($a > 0$, n entero no negativo)
 - ✖ Cálculo de $n!$
 - ✖ Multiplicación (estándar) de matrices
 - ✖ Búsqueda de valores claves en una lista

Fuerza Bruta y ordenación

- Algunos algoritmos simples de ordenación donde se aplica:
 - Búsqueda Secuencial $O(n)$
 - Ordenación por Burbuja $O(n^2)$
 - Ordenación por Selección $O(n^2)$
 - Recorrer el array para encontrar el menor elemento y entonces intercambiarlo con el primero.
Esto se repite a partir del segundo elemento hasta conseguir tener ordenado el array
$$A[0] \leq \dots \leq A[i-1] | A[i], \dots, A[min], \dots, A[n-1]$$



Procedimiento Selección ($T[1..n]$)

```
for i := 1 to n-1 do
    minj := i; minx := T[i]
    for j := i+1 to n do
        if T[j] < minx then minj := j
                    minx := T[j]
    T[minj] := T[i];
    T[i] := minx
```

Fuerza Bruta y evaluación de polinomios

Se trata de encontrar el valor del polinomio

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x^1 + a_0$$

en el valor $x = x_0$

Algoritmo de Fuerza Bruta

```
p ← 0.0
for i ← n downto 0 do
    power ← 1
    for j ← 1 to i do      //calcular  $x^i$ 
        power ← power * x
    p ← p + a[i] * power
return p
```

La eficiencia es claramente de $O(n^2)$

Mejora de la evaluación de polinomios

Podemos hacerlo mejor evaluando de derecha a izquierda

Algoritmo de Fuerza Bruta Mejor

$p \leftarrow a[0]$

$power \leftarrow 1$

for $i \leftarrow 1$ to n do

$power \leftarrow power * x$

$p \leftarrow p + a[i] * power$

return p

Ahora la eficiencia es de $O(n)$

Aún hay mas algoritmos (como el de Horner)

Problema del par más cercano

- Encontrar los dos puntos más cercanos en un conjunto de n puntos (en el plano Cartesiano bi-dimensional)

Algoritmo de Fuerza Bruta

Calcular la distancia que existe entre cualquier pareja de puntos distintos
Devolver los indices de los puntos que tengan la distancia más corta

Algoritmo ParDePuntosmasCercanos (P)

```
dmin ← ∞  
for i ← 1 to n-1 do  
    for j ← i + 1 to n do  
        d ← sqrt ((xi - xj) 2 + (yi - yj)2)  
        if d < dmin  
            dmin ← d
```

Eficiencia: O(n²)

Fortalezas y debilidades

- Fortalezas,
 - Gran aplicabilidad, simplicidad
 - Proporcionan algoritmos razonables para algunos problemas (búsqueda, strings o multiplicación de matrices)
 - Son algoritmos estándar para realizar tareas simples de cálculo (sumas y productos de n números o búsquedas de máximos y mínimos en listas).
- Debilidades
 - Los algoritmos de Fuerza Bruta raramente proporcionan algoritmos eficientes
 - Algunos algoritmos de Fuerza Bruta son inaceptablemente lentos
 - El enfoque de la Fuerza Bruta ni es constructivo, ni creativo como otras técnicas de diseño

Algoritmos de Búsqueda Exhaustiva

- Son algoritmos del tipo Fuerza Bruta para problemas que suponen la búsqueda de algún elemento con una propiedad especial, generalmente entre conjuntos generados por permutaciones, combinaciones o subconjuntos de un conjunto.
- Suelen llamarse también Algoritmos Combinatorios
- Método:
 - Se genera una lista de todas las potenciales soluciones del problema de una manera sistemática
 - Se evalúan una por una las potenciales soluciones, despreciando las infactibles y, si el problema fuera de optimización, guardando la mejor encontrada hasta el momento
 - Cuando termina la búsqueda se presenta la solución o soluciones encontradas
- Muy frecuentes en recorridos sobre grafos, problemas de optimización, ...

Algoritmos de Búsqueda Exhaustiva

- Los algoritmos de búsqueda exhaustiva se ejecutan en tiempos razonables solo en un número muy pequeño de problemas y casos
- En casi todos los casos en los que se ve claramente que se pueden aplicar, hay alternativas que son mejores:
 - Circuitos Eulerianos
 - Caminos mínimos
 - Árboles generadores minimales
 - Problemas de asignación
- En muchos casos sin embargo, la búsqueda exhaustiva o sus variantes son los únicos procedimientos disponibles para encontrar una solución exacta

¿Diseño de Algoritmos?

- El problema de la asignación consiste en asignar n personas a n tareas de manera que, si en cada tarea cada persona recibe un sueldo, el total que haya que pagar por la realización de los n trabajos por las n personas, sea mínimo
- Hay $n!$ posibilidades que analizar
- Formulación matemática

$$\text{Min}_{i, j} \llcorner \llcorner c_{ij} x_{ij}$$

$$\text{s.a: } \sum_j x_{ij} = 1 \quad \text{para cada persona } i$$

$$\sum_i x_{ij} = 1 \quad \text{para cada tarea } j$$

$$x_{ij} = 0 \text{ o } 1 \quad \text{para todo } i \text{ y } j.$$

¿Diseño de Algoritmos?

- Consideremos el caso en que $n = 70$. Hay $70!$ posibilidades
- Si tuviéramos un computador que examinara un billón de asignaciones/sg, evaluando esas $70!$ posibilidades, desde el instante del Big Bang hasta hoy, la respuesta sería no.
- Si tuviéramos toda la tierra recubierta de máquinas de ese tipo, todas en paralelo, la respuesta seguiría siendo no
- Solo si dispusiéramos de 10^{50} Tierras, todas recubiertas de computadores de velocidad del nanosegundo, programados en paralelo, y todos trabajando desde el instante del Big Bang, hasta el día en que se termine de enfriar el sol, quizás entonces la respuesta fuera si.

**El Algoritmo Húngaro lo resuelve
en algo menos de 9 minutos**

Problemas P y NP

- ¿Por qué estudiaremos los problemas que aparecen en el programa de la asignatura?
 - ¿Que es la clase P?
 - ¿Que es la clase NP?
 - ¿Que hace que algo sea NP?
- Los algoritmos elementales que hemos visto hasta ahora se consideran algoritmos con tiempo polinomial porque sus tiempos de ejecución son funciones polinomiales
 - Se dice que todos esos algoritmos están en la **Clase P**
- ¿Qué es la Clase NP?
 - Hay algoritmos para los que la única forma de que tengan un 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

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 los problemas de la clase P, pero no los tenemos para los de la clase NP.
- El gran desafío es demostrar que

Clase NP \subseteq Clase P

¿Es $P = NP$?

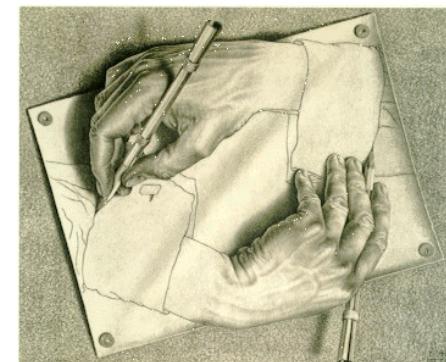
- Ya probamos que la clase P es un subconjunto de la clase NP.
- Para que ambas clases sean iguales, todos los problemas en la clase NP deberían tener algoritmos en tiempo polinomial
- De nuestros anteriores comentarios parece ridículo incluso sugerir esta posibilidad.
- Pero hasta este momento, nadie ha sido capaz de encontrar un algoritmo en tiempo polinomial ni de demostrar lo contrario.
- Por tanto, la cuestión de si “ $P = NP$ ” hoy por hoy es un problema abierto.

ALGORÍTMICA

Capítulo 1: La Eficiencia de los Algoritmos

Tema 3: Resolución de Recurrencias Asintóticas

- Metodologías de demostración. Inducción
- Resolución de recurrencias con función característica
- Ecuaciones homogéneas
- Ecuaciones no homogéneas
- Cambio de variable
- Transformaciones del rango



M.C. Escher,
Drawings Hands,
1948

Metodologías de demostración

- Demostración por Contradicción
 - Suponemos que un resultado (teorema) es falso. Demostramos que esta hipótesis implica que una propiedad que sabemos que es cierta, es falsa. Se concluye entonces que la hipótesis original es verdadera
- Demostración mediante un Contraejemplo
 - Se trata de usar un ejemplo concreto para demostrar que una propiedad no se verifica siempre
- Inducción Matemática
 - Se demuestra un caso base , se supone que la hipótesis que queremos demostrar es cierta hasta k, y entonces se demuestra que es cierta para $k+1$

Motivación

- Las ecuaciones recurrentes son frecuentes en Teoría de Algoritmos
- La inducción se usa en las demostraciones matemáticas asociadas a algoritmos recursivos
 - e.g. quicksort, búsqueda binaria
- También se usa para obtener estimaciones de los tiempos de ejecución
 - A partir de los tamaños de los datos input data
 - e.g. El tiempo crece **linealmente** con el número de datos que se procesan
 - e.g. Tiempos basados en las veces que se ejecuta un lazo

El método inductivo

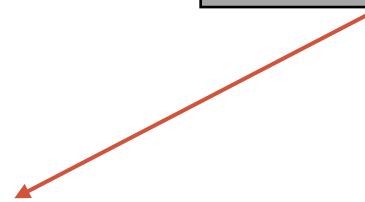
- La inducción se usa para resolver problemas como:
 - ¿Es cierto $S(n)$ para todos los valores de n ?
 - generalmente para todo $n \geq 0$ o todo $n \geq 1$
- Ejemplo:
 - Sea $S(n)$ " $n^2 + 1 > 0$ "
 - Es cierto $S(n)$ para todo $n \geq 1$?

$S(n)$ puede ser mucho mas complicado, por ejemplo un programa que se tenga que ejecutar para un valor n

El método inductivo

- ¿Cómo demostramos la veracidad o falsedad de $S(n)$?
- Una forma sería hacer la demostración para cada valor de n :
 - ¿Es $S(1)$ cierto?
 - ¿Es $S(2)$ cierto?
 - ...
 - ¿Es $S(10,000)$ cierto?
 - ... ¡¡¡El método de la fuerza bruta!!!

No es muy práctico



El método inductivo

- La inducción es una técnica para demostrar rápidamente la veracidad o falsedad de $S(n)$ para todo n
 - Sólo hay que hacer dos cosas
- Primero demostrar que **$S(1)$ es cierto**
- Segundo, suponer que $S(n)$ es cierto, y usarlo para probar que $S(n+1)$ es cierto
- Entonces $S(n)$ es cierto para todo $n \geq 1$.

Ejemplo

- Demostrar que $S(n)$: " $n^2 + 1 > 0$ " $\forall n \geq 1$
- Primero probamos que $S(1)$ es verdadero
- $S(1) = 1^2 + 1 = 2$, que es > 0
 - Así $S(1)$ es verdadero
- Ahora probamos que $S(n+1)$ es cierto, dado que $S(n)$ lo es
- Si $S(n)$ es cierto, entonces $n^2 + 1 > 0$, luego
 - $S(n+1) = (n+1)^2 + 1 = n^2 + 2n + 1 + 1 = (n^2 + 1) + 2n + 1$
 - Cómo $n^2 + 1 > 0$, entonces $(n^2 + 1) + 2n + 1 > 0$
 - así $S(n+1)$ es cierto, dado que $S(n)$ lo era
 - Por tanto $S(n) \rightarrow S(n+1)$

La Inducción mas formalmente

- Tres hechos:
 - 1. Hay que demostrar una **propiedad** $S(n)$
La propiedad debe estar planteada sobre un valor entero n
 - 2. Una **base** para la demostración.
Esta es la propiedad $S(b)$ para algún entero. A menudo $b = 0, 1$.
 - 3. Una **etapa inductiva** para la demostración.
Probamos la sentencia:
$$“S(n) \rightarrow S(n+1)” \forall n .$$
- A $S(n)$ se le suele llamar **Hipótesis de Inducción**
 - Se concluye que $S(n)$ es cierto para todo $n \geq b$
 $S(n)$ podría no ser cierto para algún $< b$

Ejemplo 1

- Demostrar $S(n)$: $\sum_{i=1}^n i = \frac{n(n+1)}{2} \quad \forall n \geq 1$

- e.g. $1+2+3+4 = (4*5)/2 = 10$

- **Base.** $S(1)$, $n = 1 \quad \sum_{i=1}^1 i = 1_1 = (1*2)/2$

- **Inducción.** Suponemos que $S(n)$ es cierto.
Demostramos $S(n+1)$, es decir:

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

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

Ejemplo 2

- Probar $S(n): \sum_{i=0}^n 2^i = 2^{n+1} - 1 \quad \forall n \geq 0$

- e.g. $1+2+4+8 = 16-1$

- **Base.** $S(0), n = 0 \quad \sum_{i=0}^0 2^i = 2^0 \cdot 2^0 = 2^1 - 1$

- **Inducción.** Suponemos $S(n)$ cierta, y probamos $S(n+1)$, que es:

$$\sum_{i=0}^{n+1} 2^i = 2^{n+2} - 1$$

$$\sum_{i=0}^{n+1} 2^i = \sum_{i=0}^n 2^i + 2^{n+1}$$

Ejemplo 3

- Demostrar $S(n)$: $n! \geq 2^{n-1} \forall n \geq 1$
 - e.g. $5! \geq 2^4$, o lo que es lo mismo $120 \geq 16$
- **Base.** $S(1)$, $n = 1$: $1! \geq 2^0$
así tenemos $1 \geq 1$
- **Inducción.** Suponemos $S(n)$ cierta y probamos $S(n+1)$, es decir: $(n+1)! \geq 2^{(n+1)-1} \geq 2^n$

$$(n+1)! = n! * (n+1)$$

Inducción parcial constructiva

- Supongamos la hipótesis (**parcialmente especificada**) de que cualquier entero ≥ 24 puede escribirse como $5a+7b$ para enteros no negativos a y b.
- La idea es aplicar el método inductivo y a lo largo de las demostraciones que hay que realizar, reunir información sobre a y b como para que se verifique la hipótesis inicial

Inducción parcial constructiva: Ejemplo

- Sea la función $f: \mathbb{N} \rightarrow \mathbb{N}$ definida por la siguiente recurrencia,

$$\begin{aligned} f(n) &= 0 && \text{si } n = 0 \\ &= n + f(n-1) && \text{en otro caso} \end{aligned}$$

está claro que $f(n) = \sum_{i=0..n} i$.

- Supongamos que no sabemos que $f(n)$ vale $n(n+1)/2$, pero que estamos buscando una formula como esa. Obviamente,

$$f(n) = \sum_{i=0..n} i \leq \sum_{i=0..n} n = n^2,$$

y por tanto $f(n)$ es $O(n^2)$.

- Esto sugiere que formulemos la hipótesis de que $f(n)$ podría ser un polinomio cuadrático.
- Ensayaremos la hipótesis de inducción parcialmente especificada $H_I(n)$ de acuerdo con la cual $f(n) = an^2 + bn + c$.
- Esta hipótesis es parcial en el sentido de que a , b , y c no son aún conocidos.

Inducción parcial constructiva: Ejemplo

- Supongamos que $HI(n-1)$ es cierta para algun $n > 1$, sabemos que,

$$\begin{aligned}f(n) &= n + f(n-1) = n + a(n-1)^2 + b(n-1) + c = \\&= an^2 + (1+b-2a)n + (a-b+c)\end{aligned}$$

- Para concluir $HI(n)$, tiene que darse que

$$f(n) = an^2 + bn + c$$

- Igualando los coeficientes de cada potencia de n ,

$$1 + b - 2a = b$$

$$a - b + c = c.$$

- Así $a = b = 1/2$, estando el valor de c aún no restringido.
- Por tanto, $HI(n)$: $f(n) = n^2/2 + n/2 + c$.

Inducción parcial constructiva: Ejemplo

- Acabamos de demostrar que si $HI(n-1)$ es cierta para algún $n > 1$, entonces también lo es $HI(n)$.
- Queda por establecer la veracidad de $HI(0)$ para concluir que $HI(n)$ es cierta para cualquier entero n .
- Pero $HI(0)$ dice que

$$f(0) = a_0 + b_0 + c = c.$$

- Sabiendo que $f(0) = 0$, concluimos que

$$c = 0$$

y que

$$f(n) = n^2/2 + n/2$$

es cierto para cualquier entero n .

Resolucion de recurrencias

- **Método de la función característica**
 - Recurrencias homogéneas

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

- Los t_i son los valores que buscamos. La recurrencia es lineal porque no contiene términos de la forma $t_i t_{i+1}$, t_i^2 ,
- Los coeficientes a_i son constantes, y
- La recurrencia es homogénea porque la combinación lineal de los t_i es igual a cero.
- La intuición nos sugiere intentar una solución de la forma

$$t_n = x^n$$

donde x^n es una constante aún desconocida

Recurrencias homogéneas

- Si ensayamos esa solución, obtenemos,

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

- Esta ecuación se satisface si $x = 0$, o en caso contrario si

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

- Esta ecuación de grado k en x es la que se llama la **ecuación característica de la recurrencia**.
- Si las k raíces de esta ecuación, r_1, \dots, r_k , son todas distintas (¡podrían ser números complejos!), entonces

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

es una solución de la recurrencia, donde las k constantes c_i se determinan mediante condiciones iniciales. (Necesitamos exactamente k condiciones iniciales para determinar los valores de esas k constantes).

Ejemplo

$$t_n - 3t_{n-1} - 4t_{n-2} = 0, t_0 = 0, t_1 = 1.$$

- Ecuación característica,

$$x^2 - 3x - 4 = 0$$

- Raíces: -1 y 4. Por tanto, la solución general tiene la forma,

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

- El uso de las condiciones iniciales produce,

$$c_1 + c_2 = 0, \quad n = 0$$

$$-c_1 + 4c_2 = 1, \quad n = 1$$

- $c_1 = -1/5$ y $c_2 = 1/5$, obteniendo finalmente,

$$t_n = (1/5)[4^n - (-1)^n]$$

Ejemplo

$$t_n = t_{n-1} + t_{n-2}, \quad n \geq 2, \quad t_0 = 0 \text{ y } t_1 = 1.$$

- Esta recurrencia se corresponde con el algoritmo para calcular el término general de la sucesión de Fibonacci
- Puede escribirse como

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

- Ecuación característica: $x^2 - x - 1 = 0$

- Obtenemos,

$$t_n = (1/\sqrt{5})(r_1^n - r_2^n)$$

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

- Es fácil demostrar que este es el mismo resultado que el obtenido por De Moivre, con su fórmula para calcular números de la sucesión de Fibonacci

Recurrencias homogéneas

- Las raíces de la ecuación característica no son distintas.
- Sea $p(x) = a_0x^k + a_1x^{k-1} + \dots + a_k$ y r una raíz múltiple. Para cualquier valor $n \geq k$, consideramos el polinomio,

$$h(x) = x[x^{n-k}p(x)]' = a_0nx^n + a_1(n-1)x^{n-1} + \dots + a_k(n-k)x^{n-k}$$

- Sea $q(x)$ el polinomio tal que $p(x) = (x-r)^2q(x)$.
- Entonces,

$$h(x) = x[(x-r)^2x^{n-k}q(x)]' = x[2(x-r)x^{n-k}q(x) + (x-r)^2[x^{n-k}q(x)]']$$

- Como $h(r) = 0$, se demuestra que,

$$a_0nr^n + a_1(n-1)r^{n-1} + \dots + a_k(n-k)r^{n-k}$$

es decir, $t = nr^n$ es también una solución

Recurrencias homogéneas

- Mas generalmente, si m es la multiplicidad de la raíz r, entonces

$$t_1 = r, t_2 = nr^n, t_3 = n^2r^n, \dots, t_m = n^{m-1}r^n$$

son todas las posibles soluciones de la ecuación.

- La solución general es una combinación lineal de estos términos y de los términos contribuidos por otras raíces de la ecuación característica.
- Así, de nuevo hay k constantes a determinar por las condiciones iniciales

Ejemplo

$$t_n = 5t_{n-1} - 8t_{n-2} + 4t_{n-3}, \quad n \geq 3, \quad t_0=0, \quad t_1=1 \quad \text{y} \quad t_2=2.$$

- Esta recurrencia puede escribirse,

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

- Característica: $x^3 - 5x^2 + 8x - 4 = 0 \Leftrightarrow (x-1)(x-2)^2 = 0$

- Raíces: 1 (simple) y 2 (doble).

- Por tanto, la solución general es $t_n = c_1 1^n + c_2 2^n + c_3 n 2^n$.

- Las condiciones iniciales dan,

$$c_1 + c_2 = 0, \quad n = 0$$

$$c_1 + 2c_2 + 2c_3 = 1, \quad n = 1$$

$$c_1 + 4c_2 + 8c_3 = 2, \quad n = 2$$

- Así, $c_1 = -2$, $c_2 = 2$, $c_3 = -(1/2)$ y $t_n = 2^{n+1} - n2^{n-1} - 2$.

Recurrencias no homogéneas

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

- El primer miembro es lo mismo que el de las homogéneas, pero en el segundo tenemos $b^n p(n)$, donde
 - b es una constante, y
 - $p(n)$ es un polinomio en n de grado d .
- Por ejemplo, la recurrencia podría ser,

$$t_n - 2t_{n-1} = 3^n$$

en cuyo caso $b = 3$ y $p(n) = 1$ es un polinomio de grado cero

Recurrencias no homogéneas

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

- La ecuación característica que le corresponde se organiza como
(Ecuación característica de la homogénea) $(x-b)^{d+1} = 0$
 $(a_0 x^k + a_1 x^{k-1} + \dots + a_k)(x-b)^{d+1} = 0$
y se resuelve como en el caso de las homogéneas
- Si $t_n - 2t_{n-1} = 3^n$ entonces $(x-2)(x-3) = 0$
- Se aplican las mismas normas para raíces simples o múltiples que en el caso anterior

Ejemplo

$$t_n - 2t_{n-1} = (n+5) 3^n$$

- Ecuación característica

$$(x-2)(x-3)^2 = 0$$

- Y por tanto la solución es

$$t_n = c_1 2^n + c_2 3^n + c_3 n 3^n$$

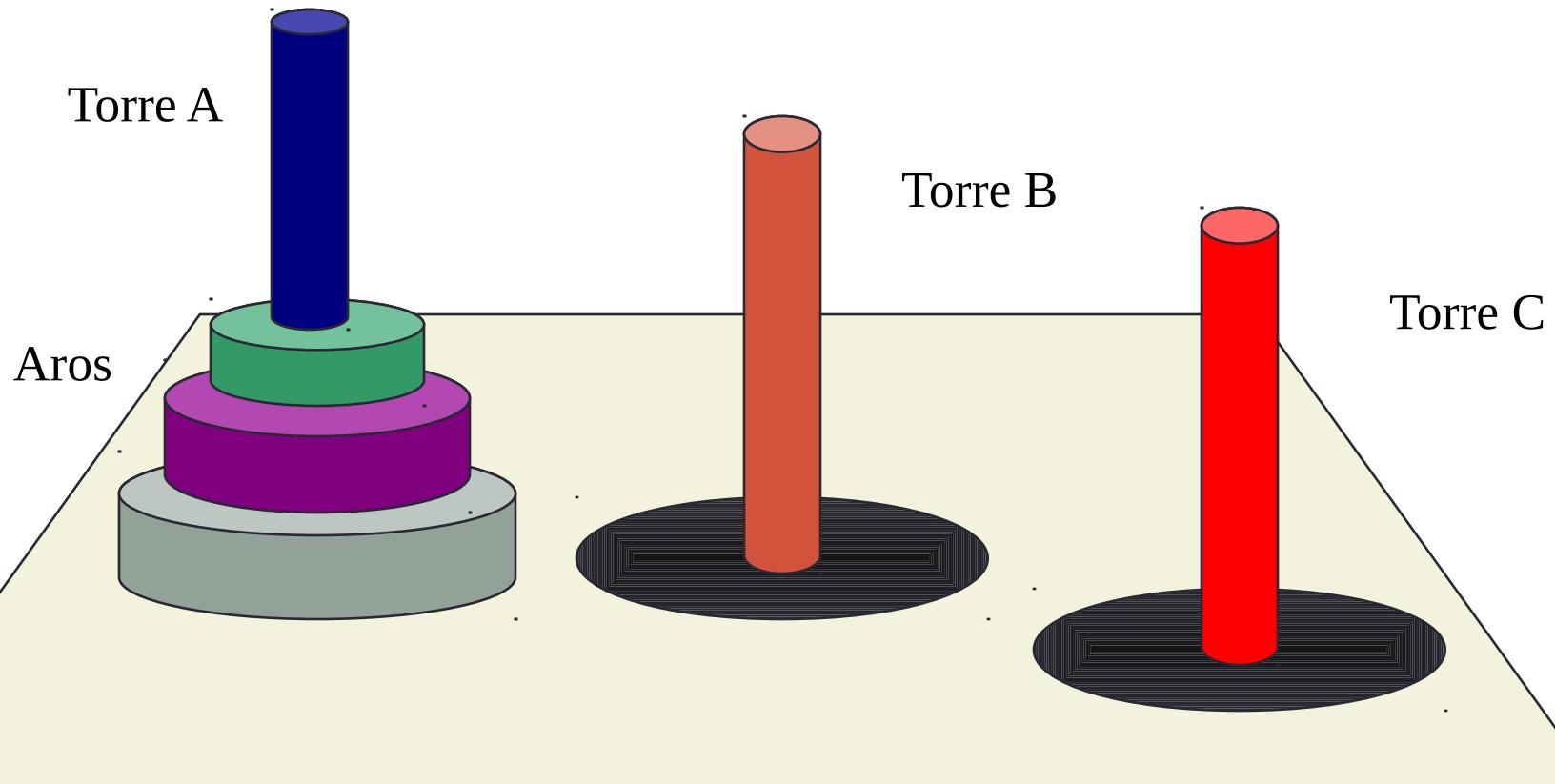
- Las constantes solo son utiles para conocer la solución con exactitud, pero no de cara a saber el orden del algoritmo del que proviene la recurrencia que estamos resolviendo
- Lo normal es conocer las condiciones iniciales a partir de datos experimentales

Ejemplo: Las Torres de Hanoi

- Según Edouard Lucas (1842-1891), en el Gran Templo de Benarés, bajo el domo que marca el centro del mundo, descansa un plato de bronce en el que están fijas tres agujas de diamante.
- En una de estas agujas un dios colocó 64 discos de oro, el disco más largo descansa sobre el plato y sobre éste está uno más pequeño y así sucesivamente hasta la punta.
- Día y noche los monjes pasan los discos de una aguja a otra, de acuerdo a fijas e inmutables leyes que requieren
 - que el sacerdote de turno no mueva mas que un disco a la vez, y
 - que el disco se coloque en un aguja de tal forma que no haya un disco más pequeño bajo éste.
- Cuando los 64 discos hayan sido transferidos de la aguja donde el dios los colocó a una de las otras, los brahmanes se convertirán en polvo, y con un trueno el mundo conocido desaparecerá.

Ejemplo: Las Torres de Hanoi

Problema: trasladar todos los aros de la barra A a la B.



$$t_n = 2t_{n-1} + 1, n \geq 1, \text{ con } t_0 = 0 \Rightarrow t_n = c_1 1^n + c_2 2^n$$

Ejemplo

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

- Puede reescribirse como $t_n - 2t_{n-1} = n$
- $b = 1$ y $p(n) = n$ un polinomio de grado 1.
- Ecuación característica: $(x-2)(x-1)^2 = 0$
- Raíces 2 (con multiplicidad 1) y 1 (con multiplicidad 2).
- Solución general: $t_n = c_1 2^n + c_2 1^n + c_3 n 1^n$
- En el problema buscamos una solución para la que $t_n \geq 0$ para cualquier n . Si esto es así podemos concluir inmediatamente que t_n debe ser $O(2^n)$.

Generalización

- Sea

$$a_0 t_n + a_1 t_{n-1} + \dots + a_k t_{n-k} = b_1^n p_1(n) + b_2^n p_2(n) + \dots$$

donde las b_i son constantes distintas y los $p_i(n)$ son polinomios en n de grado d_i respectivamente.

- Basta escribir la ecuación característica,

$$(a_0 x^k + a_1 x^{k-1} + \dots + a_k)(x - b_1)^{d_1+1} (x - b_2)^{d_2+1} = 0$$

- Por ejemplo $t_n = 2t_{n-1} + n + 2^n$, $n \geq 1$, con $t_0 = 0$.
- $b_1 = 1$, $p_1(n) = n$, $b_2 = 2$, $p_2(n) = 1$.
- Ecuación característica: $(x-2)(x-1)^2 (x-2) = 0$,
- Solución general: $t_n = c_1 1^n + c_2 n 1^n + c_3 2^n + c_4 n 2^n$

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 $T(n)$ está por tanto es $O(n^2)$ (n es una potencia de 2)

Cambio de variable

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

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

- 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

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

- y (n) es $O(n^2 \log n)$ / 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 $O(n \log^2 n)$ (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,

$$T(2^k) = 3T(2^{k-1}) + c2^k$$

$$t_k = 3t_{k-1} + c2^k$$

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

$$t_k = c_1 3^k + c_2 2^k$$

$$T(n) = c_1 3^{\lg n} + c_2 n$$

- y como $a^{\lg b} = b^{\lg a}$, $T(n) = c_1 n^{\lg 3} + c_2 n$

y finalmente $T(n)$ es $O(n^{\lg 3})$ (n es 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 así

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

- 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 así,

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

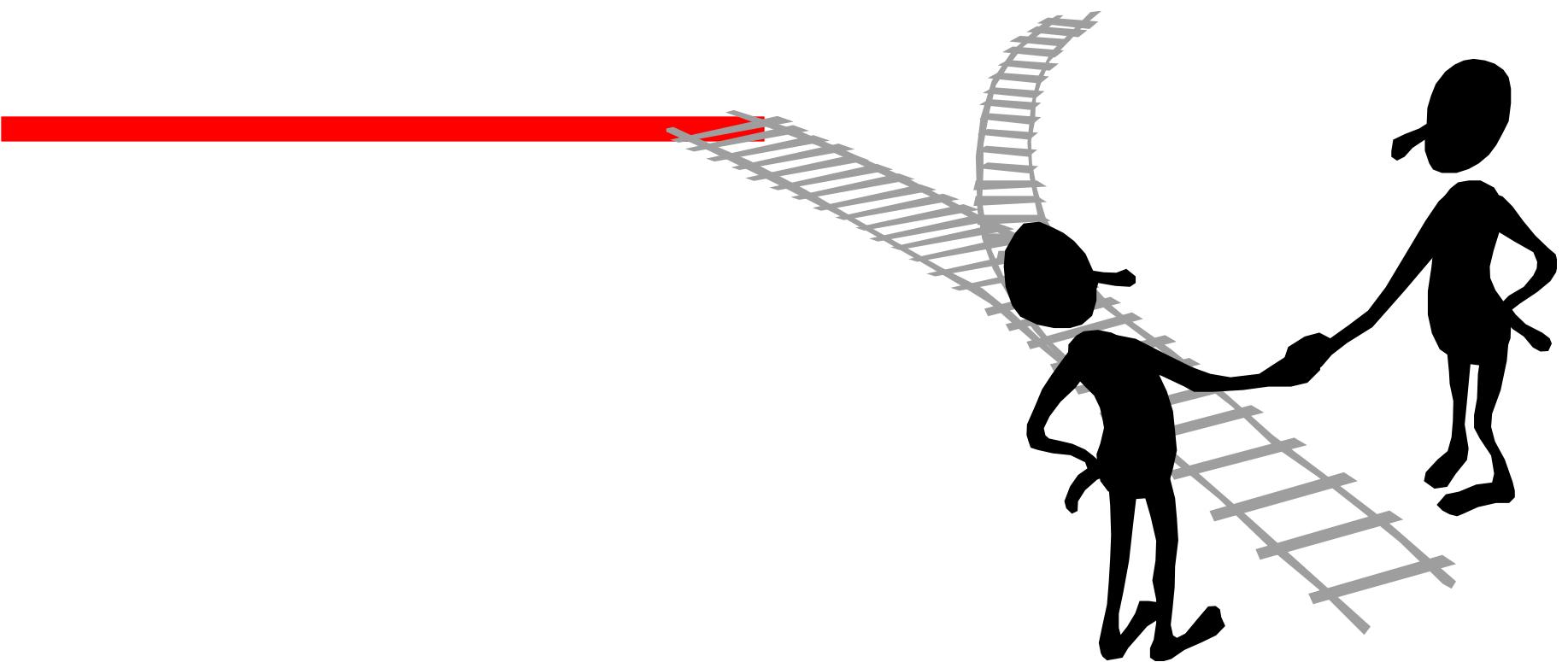
Algorítmica

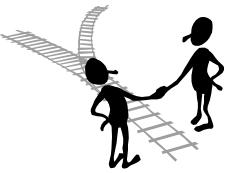
Capítulo 2: Algoritmos Divide y Vencerás

Tema 4: Método General DV

- El método general
- Determinación del umbral

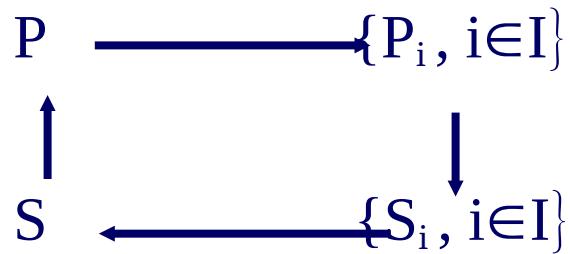
Divide y Vencerás



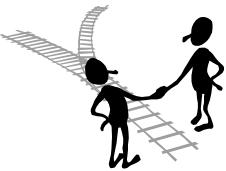


Divide y Vencerás

- **Dividir** el problema (P) en varios subproblemas (P_i)
- **Vencer** los subproblemas (resolverlos)
- **Combinar** las soluciones (S_i) de los subproblemas para obtener la solución (S) del problema inicial



- Este enfoque, sobre todo cuando se utiliza recursivamente, a menudo proporciona soluciones eficientes para problemas en los que los subproblemas son **versiones reducidas** y **resolvibles** del problema original.
- Las ecuaciones recurrentes serán naturales en este método



Recordatorio de recurrencias

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

$$T(n) = 3T(n/2) + dn \quad (d \text{ es constante}, n \geq 1).$$

- Obtenemos sucesivamente,

$$T(2^k) = 3T(2^{k-1}) + d2^k$$

$$t_k = 3t_{k-1} + d2^k$$

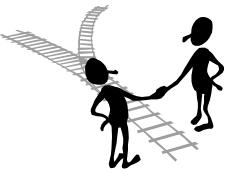
- Ecuación característica: $(x-3)(x-2) = 0$, y así,

$$t_k = c_1 3^k + c_2 2^k$$

$$T(n) = c_1 3^{\lg n} + c_2 n$$

- y como $a^{\lg b} = b^{\lg a}$,

$$\mathbf{T(n) = c_1 n^{\lg 3} + c_2 n}$$



Divide y Vencerás

- La eficacia de la técnica DV dependerá de varios factores
- Se trata de un proceso complejo que requiere muchas comprobaciones:
 - No sabemos si el problema P podrá descomponerse.
 - Los P_i tendrán la misma naturaleza entre ellos y con P ; han de ser razonablemente pequeños en número; no pueden ser muy numerosos; cada P_i debe ser más sencillo de resolver que P .
 - Hay que esperar que tenga sentido integrar las S_i para obtener la solución final S .
 - Debemos esperar que S se corresponda con la solución del problema P .
- Aplicando este procedimiento debemos obtener un algoritmo que sea mejor que otro algoritmo que resuelva P sin esta técnica en peor tiempo.

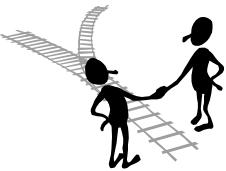


Divide y Vencerás, justificación

- Supongamos un problema P, de tamaño n, que sabemos puede resolverse con un algoritmo (básico) A
$$t_A(n) \leq cn^2.$$
- Dividimos P en 3 subproblemas de tamaños $n/2$, siendo cada uno de ellos del mismo tipo que A, y consumiendo un tiempo lineal la combinación de sus soluciones: $t(n) \leq dn$
- Tenemos un nuevo algoritmo B, Divide y Vencerás, que consumirá un tiempo

$$\begin{aligned} t_B(n) &= 3 t_A(n/2) + t(n) \leq 3 t_A(n/2) + dn \leq \\ &\leq (3c/4) n^2 + dn. \end{aligned}$$

- B tiene un tiempo de ejecución mejor que el algoritmo A, ya que disminuye la constante oculta



Divide y Vencerás, justificación

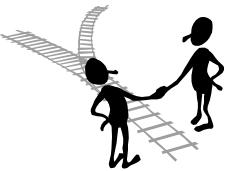
- Pero si cada subproblema se resuelve de nuevo con Divide y Vencerás, podemos hacer un tercer algoritmo C recursivo que tendría un tiempo:

$$t_C(n) = 3 t_C(n/2) + t(n)$$

- de forma que:

$$\begin{aligned} t_C(n) &= t_A(n) && \text{si } n \leq n_0 \\ &= 3 t_C(n/2) + t(n) && \text{si } n \geq n_0 \end{aligned}$$

$$t_C = c_1 n^{\lg 3} + c_2 n$$



Divide y Vencerás, justificación

- Pero si cada subproblema se resuelve de nuevo con Divide y Vencerás, podemos hacer un tercer algoritmo C recursivo que tendría un tiempo:

$$t_C(n) = 3 t_C(n/2) + t(n)$$

- de forma que:

$$t_C(n) = t_A(n) \quad \text{si } n \leq n_0$$

$$= 3 t_C(n/2) + t(n) \quad \text{si } n \geq n_0.$$

- $t_C(n)$ es mejor en eficiencia que los algoritmos A y B.
- El dividir el problema P en una serie de subproblemas, no significa que dichos subproblemas sean disjuntos. La división de P es exhaustiva pero no excluyente.
- Al valor n_0 se le denomina umbral y es fundamental para que funcione bien la técnica



Algoritmo Divide y Vencerás

Funcion DV(P)

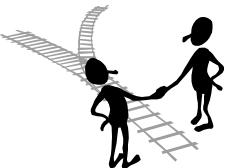
Si P es suficientemente pequeño o simple entonces devolver **basico** (P).

Descomponer P en subcasos P(1), P(2), ..., P(k) mas pequeños
para i = 1 hasta k hacer $S(i) = DV(P(i))$

recombinar las S(i) en S (solucion de P)

Devolver (S)

- Cuando $k = 1$ DV se llama simplificación.
- No hay ninguna regla para hallar k, sólo basándose en la experiencia sabemos que los DV con pocos subproblemas de tamaños parecidos entre ellos funcionan mejor.
- En función del valor del umbral n_0 , obtendremos mejores o peores resultados. Su determinación es clave.



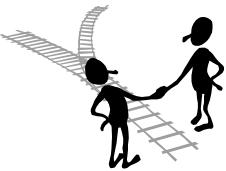
Análisis de algoritmos DV

- Cuando un algoritmo contiene una llamada recursiva a si mismo, generalmente su tiempo de ejecución puede describirse por una recurrencia que da el tiempo de ejecución para un caso de tamaño n en función de inputs de menor tamaño.

- En el caso de DV nos encontraremos recurrencias como:

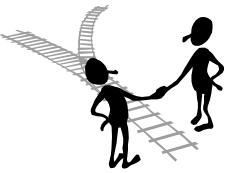
$$T(n) = \begin{cases} t(n) & \text{si } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{en otro caso} \end{cases}$$

- donde a es el numero de subproblemas, n/b el tamaño de estos, D(n) el tiempo de dividir el problema en los sub-problemas y C(n) el tiempo de combinacion de las soluciones de los subproblemas



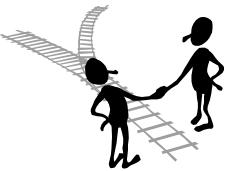
Ejemplo: Las Torres de Hanoi

- Problema: Mover los n discos del tubo A al B usando el tubo C como tubo intermedio temporal
- Enfoque Divide y Vencerás:
 - Dos subproblemas de tamaño $n-1$:
 - (1) Mover los $n-1$ discos mas pequeños de A a C
 - (*) Mover el n° disco mas pequeño de A a B es fácil
 - (2) Mover los $n-1$ discos mas pequeños de C a B
 - El movimiento de los $n-1$ discos mas pequeños se hace con la aplicación recursiva del método



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 estaríamos resolviendo con el algoritmo básico siempre.
 - 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

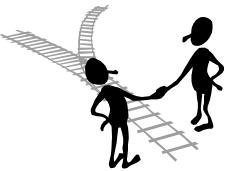


Determinación del umbral

- Supongamos el algoritmo anterior, en el que

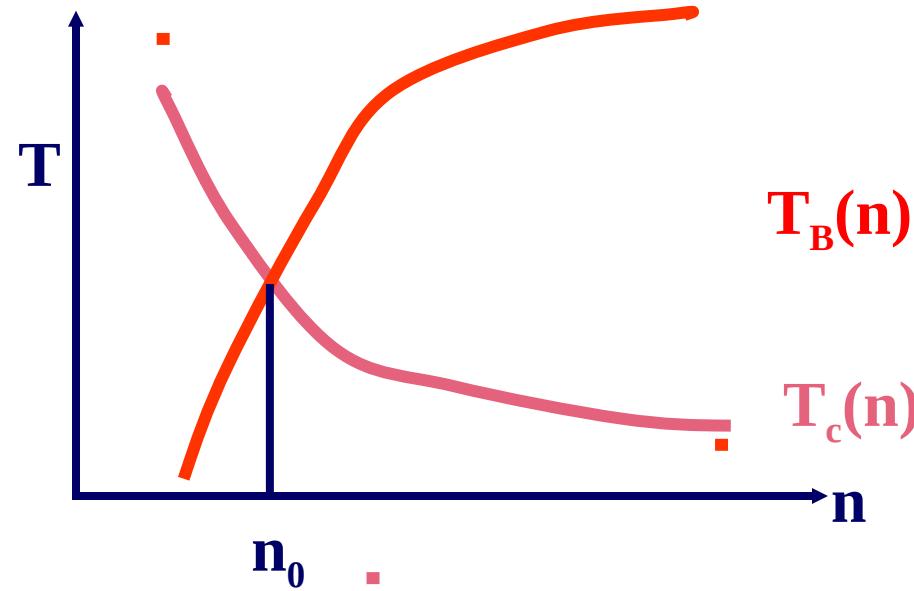
$$\begin{aligned} t_C(n) &= t_A(n) && \text{si } n \leq n_0 \\ &= 3t_C(n/2) + t(n) && \text{si } n \geq n_0 \end{aligned}$$

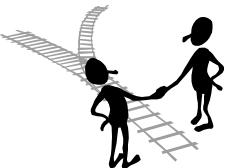
- Una implementación concreta $t_A(n) = n^2$ y $t(n) = 16n$ (ms), y un caso de tamaño $n = 1024$.
- Las dos posibilidades extremas nos llevan a
 - Si $n_0 = 1$, $t_C(n) = 32$ m y 34 sg
 - Si $n_0 = \infty$, $t_C(n) = 17$ m y 40 sg
- Si puede haber tan grandes diferencias, ¿cómo podremos determinar el valor óptimo del umbral?
- Dos métodos: Experimental y Teórico



Determinación del umbral

- **Método experimental**
- Implementamos el algoritmo básico 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.





Determinación del umbral

- **Método teórico**

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

$$t_C(n) = t_A(n) \quad \text{si } n \leq n_0 \\ = 3 t_C(n/2) + t(n) \quad \text{si } n \geq n_0$$

- Cuando coinciden los tiempos de los dos algoritmos

$$t_A(n) = 3 t_A(n/2) + t(n); t_C(n) = t_A(n) \text{ y } n = n_0$$

- Para una implementación concreta (por ejemplo, la anterior, $t_A(n) = n^2$ y $t(n) = 16n$ (ms) y $n = 1024$)

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

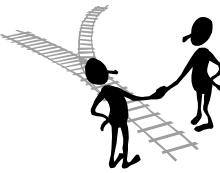
$$\mathbf{n_0 = 64}$$

Algorítmica

Capítulo 2: Algoritmos Divide y Vencerás

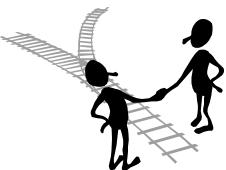
Tema 5: Búsqueda y ordenación

-
- **Algoritmos de búsqueda**
 - **Algoritmos de ordenación**
 - **Ordenación por mezcla**
 - **Quicksort**



Métodos de Búsqueda y Ordenación

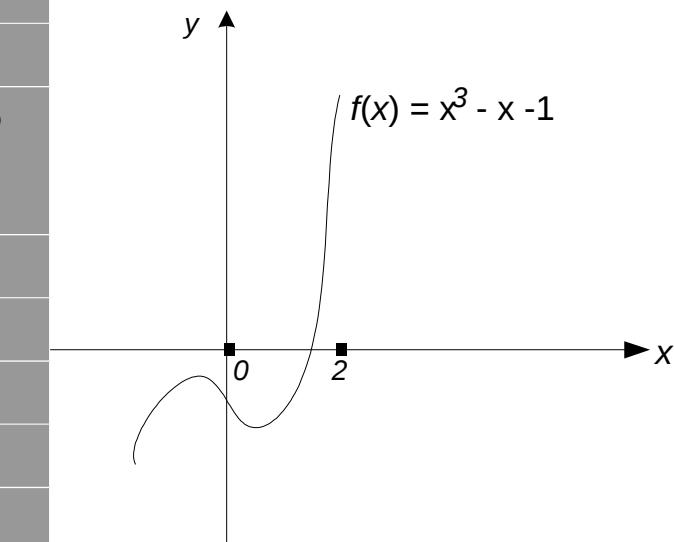
- Los problemas más comunes en la informática son la búsqueda y la ordenación.
 - Número de preguntas diarias en Google: 3 billones de búsquedas por día, i.e. 90 billones al mes (2015)
- Por lo tanto, la eficiencia de la búsqueda es importante
- La ordenación consiste en ordenar los elementos de un conjunto con el fin de acelerar la búsqueda.
- Los algoritmos Divide y Vencerás son idóneos para resolver buena parte de los problemas de ordenación.
- Pero, los métodos de búsqueda anteceden al uso de los computadores.



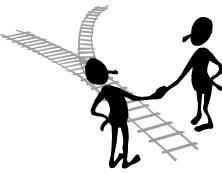
Algoritmos de búsqueda: Bisección

- Encontrar la raíz de $x^3 - x - 1 = 0$, con un error ≤ 0.01

n	a_n	b_n	x_n	$f(x_n)$
1	0.0-	2.0+	1.0	-1.0
2	1.0-	2.0+	1.5	0.875
3	1.0-	1.5+	1.25	-0.296875
4				
5				
6				
7				
8	1.3125-	1.328125+	1.3203125	-0.018711



$$x \approx 1.3203125$$



Algoritmos de búsqueda

- Búsqueda lineal (secuencial)

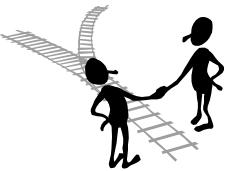
- Granada
- Palencia
- Castellón
- Cáceres
- Pamplona
- Murcia
- Huesca
- Santiago
- Valladolid
- Soria
- Gerona
- Guadalajara
- Logroño
- Bilbao

- 14 items
- Número minimo de comparaciones = 1
- Número maximo de comparaciones = 13
- En promedio: $13/2 = 6$ o 7 comparaciones

No es un método muy eficiente

Por tanto no se usará ¡nunca!

- Búsqueda lineal sobre una lista ordenada
- Búsqueda binaria



Búsqueda lineal en una lista ordenada

Bilbao
Cáceres
Castellón
Gerona
Granada
Guadalajara
Huesca
Logroño
Murcia
Palencia
Pamplona
Santiago
Soria
Valladolid

- 14 items
- Número minimo de comparaciones = 1
- Número maximo de comparaciones = 13
- Número promedio: $13/2 = 6$ o 7

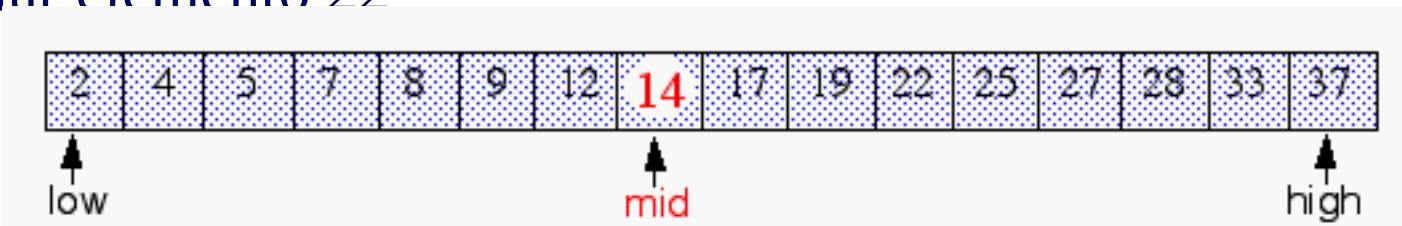
¿Por que entonces es mas eficiente buscar sobre una lista ordenada que en una no ordenada?

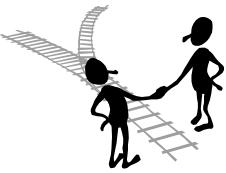
¿Deberiamos ordenar siempre la lista antes de buscar?



Algoritmo Búsqueda Binaria

Hallar elemento 22





Algoritmo Búsqueda Binaria

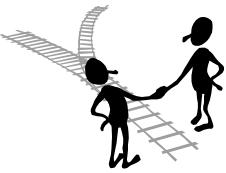
Hallar elemento 22

2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----



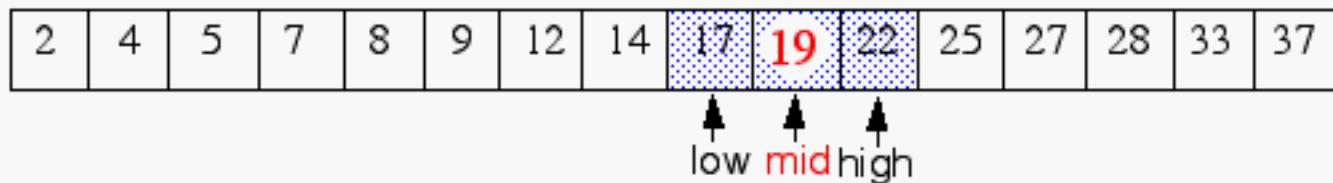
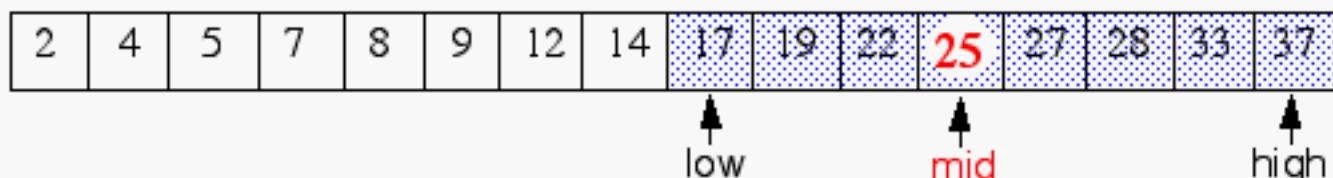
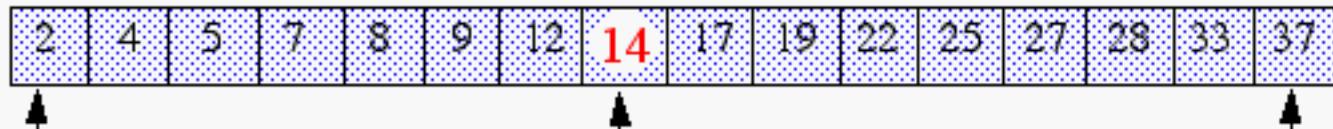
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

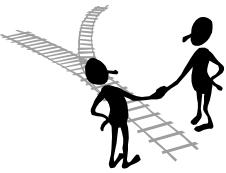




Algoritmo Búsqueda Binaria

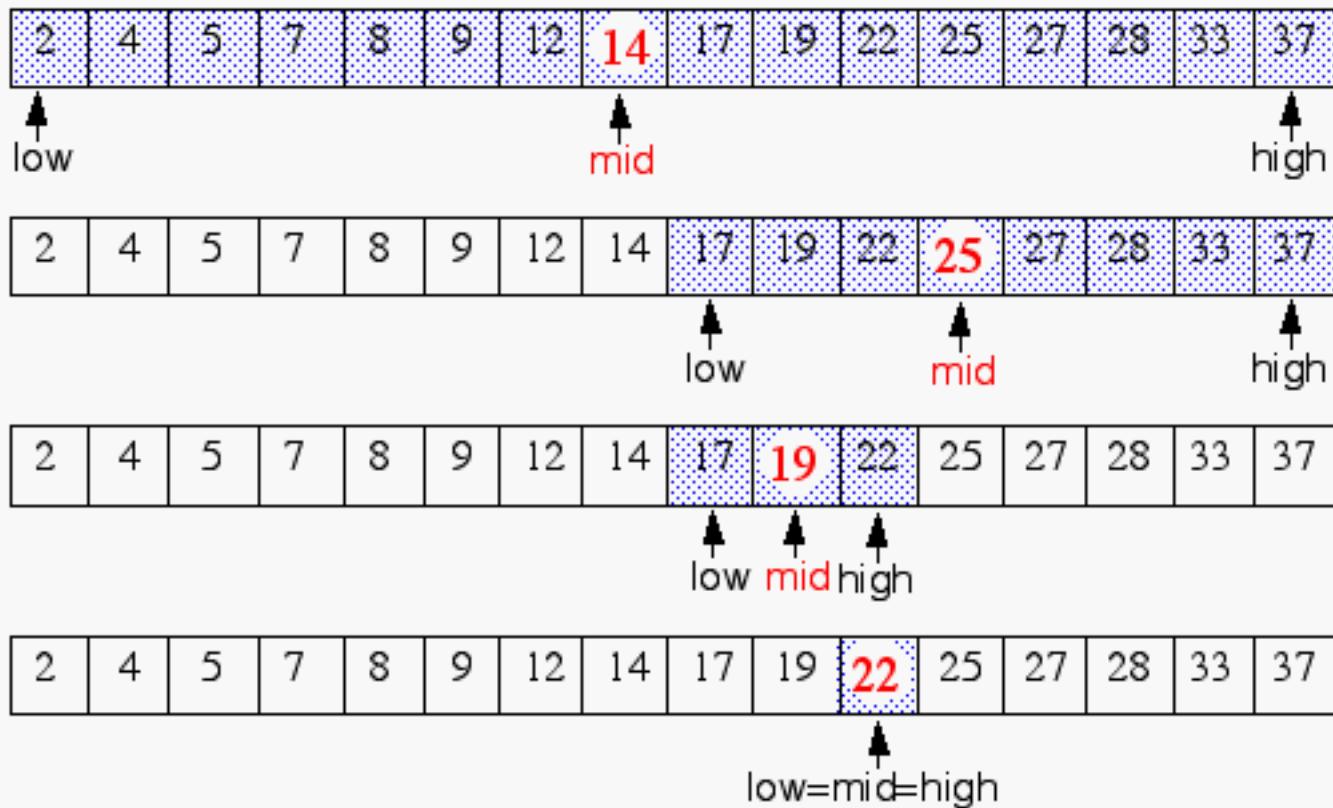
Hallar elemento 22

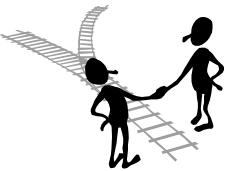




Algoritmo Búsqueda Binaria

Hallar elemento 22





Búsqueda lineal en una lista ordenada

```

Funcion BuscaBin (T[1, 2, ..., n])
    If n = 0 or x < T[1]
        then return 0
    Return Binrec (T, x)

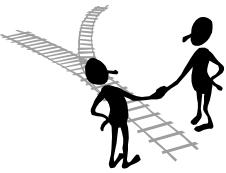
```

```

Fucion Binrec (T[i, ...,j], x)
  if i = j then return i
  k = i + j + 1/ div 2
  if x < t[k]
    then return binrec (t[i, ..., k - 1], x)
  else return binrec (t[k, ..., j])

```

- $T(n) = O(1)$ si $n \leq 0$
= $T(n/2) + a$ si $n \geq 0$
 - $T(n) = c_1 \log n$, entonces $T(n)$ es $O(\log n)$.
 - La búsqueda binaria mas que ser una técnica DV pura, es un caso de simplificación.



Algoritmos de Ordenación

- El esquema general de ordenación Divide y Vencerás es el siguiente

Algoritmo General 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

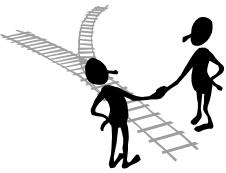
Iniciar Ordenar(izquierda)

Iniciar Ordenar(derecha)

Combinar izquierda y derecha

End

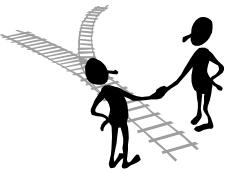
End Algoritmo



Ordenacion por mezcla

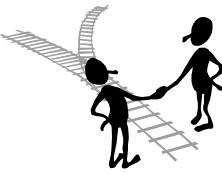
- Aplicamos el método DV a la resolución del siguiente problema de ordenación
- **Problema:** Dados n elementos de la misma naturaleza, ordenarlos en orden no decreciente
- Divide y Vencerás:
 - Si $n=1$ terminar (toda lista de 1 elemento está ordenada)
 - Si $n>1$, partir la lista de elementos en dos o mas subcolecciones; ordenar cada una de ellas; combinar en una sola lista.

Pero, ¿**Como hacer la partición?**



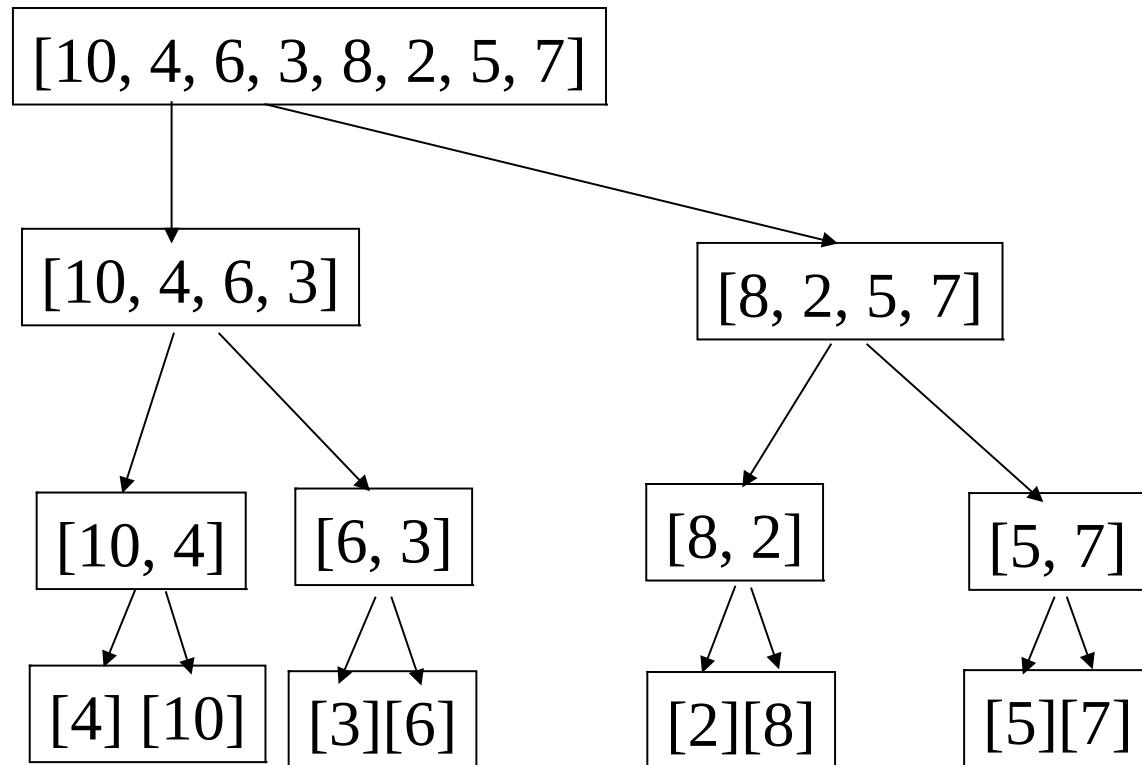
Ordenación por mezcla

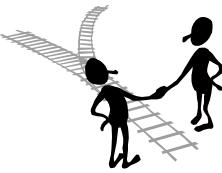
- Buscamos hacer una partición equilibrada de la lista en dos partes A y B
- En A habrá n/k elementos, y en B el resto
- Ordenamos entonces A y B recursivamente
- Combinamos las listas ordenadas A y B usando un procedimiento llamado **mezcla**, que combina las dos listas en una sola
- El problema queda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



Ejemplo

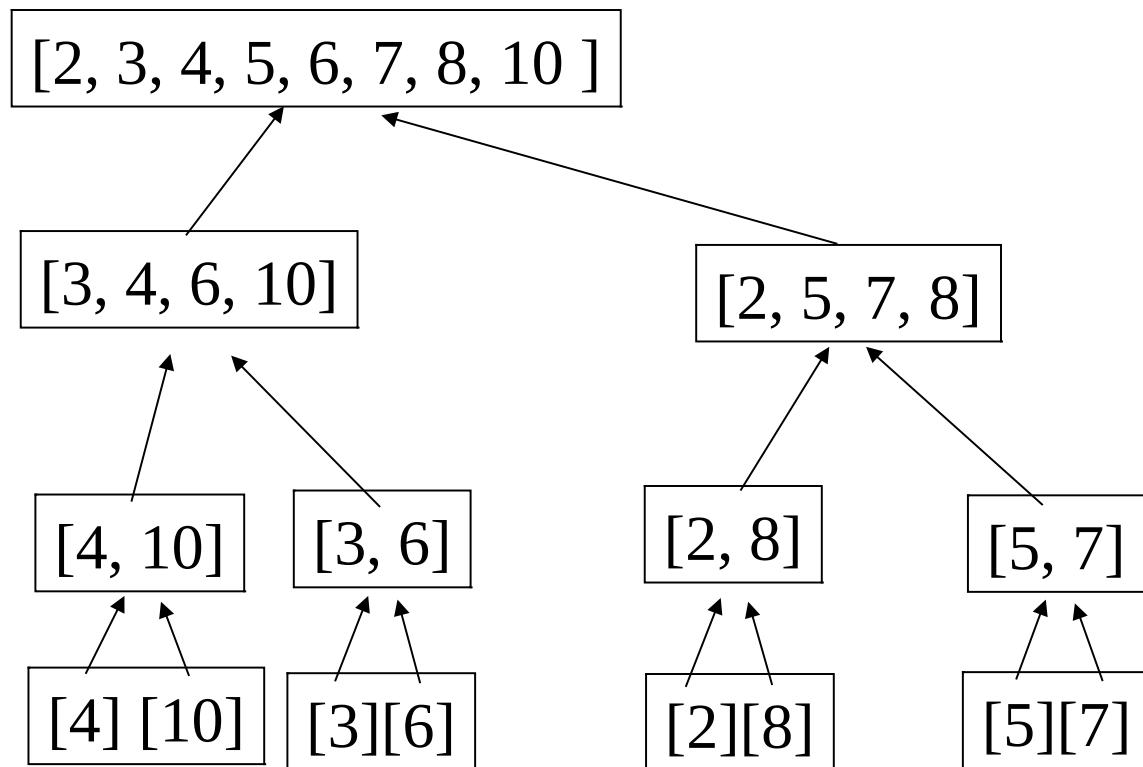
- Sea $k=2$
- Partimos la lista en otras dos de tamaños $n/2$

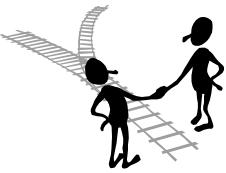




Ejemplo

- La operación de mezcla para k=2 produciría





Un código de ordenación por mezcla

```
void mergeSort(Comparable [a, int left, int right)
{
    // sort a[left:right]
    if (left < right)
        { // at least two elements
            int mid = (left+right)/2; //midpoint
            mergeSort(a, left, mid);
            mergeSort(a, mid + 1, right);
            merge(a, b, left, mid, right); // merge from a to b
            copy(b, a, left, right); //copy result back to a
        }
}
```



Cálculo de la eficiencia

- **Ecuación recurrente:**
- Suponemos que n es potencia de 2

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

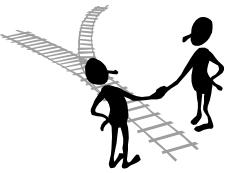
- Podemos intentar la solución por expansión

$$T(n) = 2T(n/2) + c_2n; \quad T(n/2) = 2T(n/4) + c_2n/2$$

$$T(n) = 4T(n/4) + 2c_2n; \quad T(n) = 8T(n/8) + 3c_2n$$

- En general,

$$T(n) = 2^i T(n/2^i) + i c_2 n$$



Solucion

- Tomando $n = 2^k$, la expansión termina cuando llegamos a $T(1)$ en el lado de la derecha, lo que ocurre cuando $i=k$

$$T(n) = 2^k T(1) + k c_2 n$$

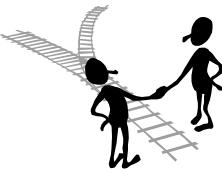
- Como $2^k = n$, entonces $k = \log n$;
- Como además

$$T(1) = c_1$$

- Tenemos

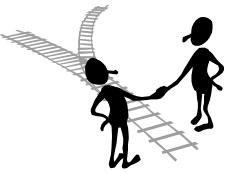
$$T(n) = c_1 n + c_2 n \log n$$

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



Quick Sort

- También se le conoce con el nombre de Algoritmo de Hoare
- Es el algoritmo (general) de ordenación mas eficiente
- En síntesis
 - Ordena el array A eligiendo un valor clave v entre sus elementos, que actúa como pivote
 - Organiza tres secciones: izquierda, pivote y 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 seria que el pivote se colocara en la mediana para que la parte izquierda y la derecha tuvieran el mismo tamaño



Pseudo Código para quicksort

Algoritmo QUICKSORT(S,T)

// Precondición: Existe el conjunto S y es finito.

// Postcondición: los elementos de S son de la misma naturaleza, están dispuestos en una estructura lineal y son ordenables en una estructura T.

Begin Algoritmo

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 $x < p < z$ and $y = p$

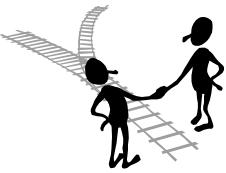
2. $TAMAÑO(S1) < TAMAÑO(S)$ y $TAMAÑO(S3) < TAMAÑO(S)$

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

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

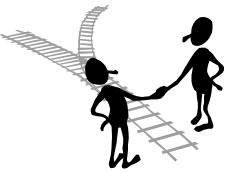
Combinación: $T = T1 \parallel S2 \parallel T3$ // S2 es el elemento intermedio entre cada mitad ordenada

End Algoritmo



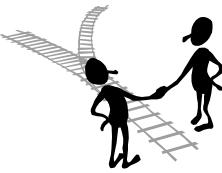
La elección del pivote

- Cada uno podemos diseñar hoy mismo nuestro propio algoritmo Quicksort (otra cosa es que funcione mejor que los que ya hay...):
La elección condiciona el tiempo de ejecución
- El pivote puede ser cualquier elemento en el dominio, pero no necesariamente tiene que estar en S
 - Podría ser la media de los elementos seleccionados en S
 - Podría elegirse aleatoriamente, pero la función RAND() consume tiempo, que habría que añadírselo al tiempo total del algoritmo
- Pivotes posibles también son la mediana de un mínimo de tres elementos, o el elemento medio de S.



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%
- Escoger como pivote el elemento en la posición central del array, dividiendo este en dos mitades



Ejemplo

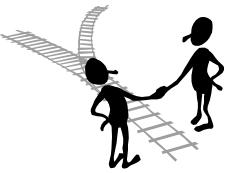
array:

5	89	35	10	24	15	37	13	20	17	70
---	----	----	----	----	----	----	----	----	----	----

tamaño: 11

Con este ejemplo vamos a ver como funcionaría un algoritmo de este tipo “por dentro”, es decir, como se construyen los sub-problemas

El objetivo no es conocer el algoritmo en si, de cara a posibles implementaciones, sino ilustrar el funcionamiento



Ejemplo

array:

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

*“elemento
pivot”*



Ejemplo

array:

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

Particion esperada al final:



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



Ejemplo

array:

0	5	89	35	14	24	15	37	13	20	7	70
---	---	----	----	----	----	----	----	----	----	---	----

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



Ejemplo

array:

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

\uparrow
 $k=1$



Ejemplo

índice:0



array:

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



k:1



Ejemplo

índice:0



array:

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



k:2



Ejemplo

índice:0



array:

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



k:3



Ejemplo

índice: 1



array:

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



k: 3



Ejemplo

índice: 1



array:

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



k:4



Ejemplo

índice: 1



array:

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



k: 5



Ejemplo

índice:2



array:

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



k:5



Ejemplo

índice:2



array:

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



k:6



Ejemplo

índice:2



array:

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



k:7 *etc...*



Ejemplo

índice:4



array:

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

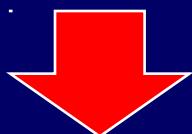


k:11



Ejemplo

Último elemento introducido



array:

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



Ejemplo

índice:4



array:

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

$x < 15$

$15 \leq x$



Ejemplo

El pivote ahora está en la posición correcta

array:

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

$x < 15$

$15 \leq x$



Ejemplo

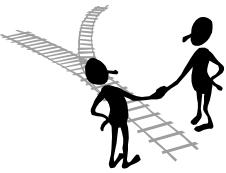
7	14	5	13	15	35	37	89	20	24	70
---	----	---	----	----	----	----	----	----	----	----

Ordena

7	14	5	13
---	----	---	----

Ordena

35	37	89	20	24	70
----	----	----	----	----	----



Algoritmo Quicksort (Hoare)

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

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

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

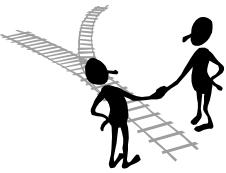
Caso contrario pivot ($T[i..j]$, l)

{tras el pivoteo, $i \leq k < l \Rightarrow T[k] \leq T[l]$ y, $l < k \leq j \Rightarrow T[k] > T[l]$ }

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

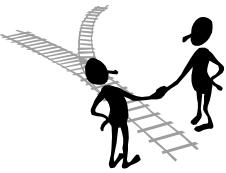
quicksort ($T[l+1..j]$)

- No es difícil diseñar un algoritmo en tiempo lineal para el pivoteo. Sin embargo, es crucial en la práctica que la constante oculta sea pequeña.
- La mejor forma de pivotear es escoger como pivote, entre los dos primeros elementos del array, el mayor de ellos



Pivoteo lineal

- 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 $T[l]$ están intercambiados. Este proceso continua mientras que $k < l$.
- Finalmente, $T[i]$ y $T[l]$ se intercambian para poner el pivote en su posición correcta.



Algoritmo de pivoteo

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

{permute 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 $T[i]$ }

$p = T[i]$

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

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

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

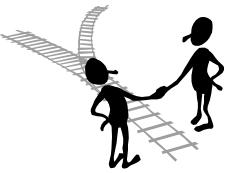
Mientras $k < l$ hacer

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 $T[l]$



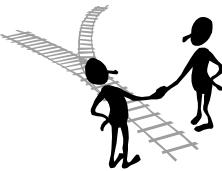
Ejemplo

Como hemos dicho el pivote que se usa mas frecuentemente es el mayor de los dos primeros elementos del vector

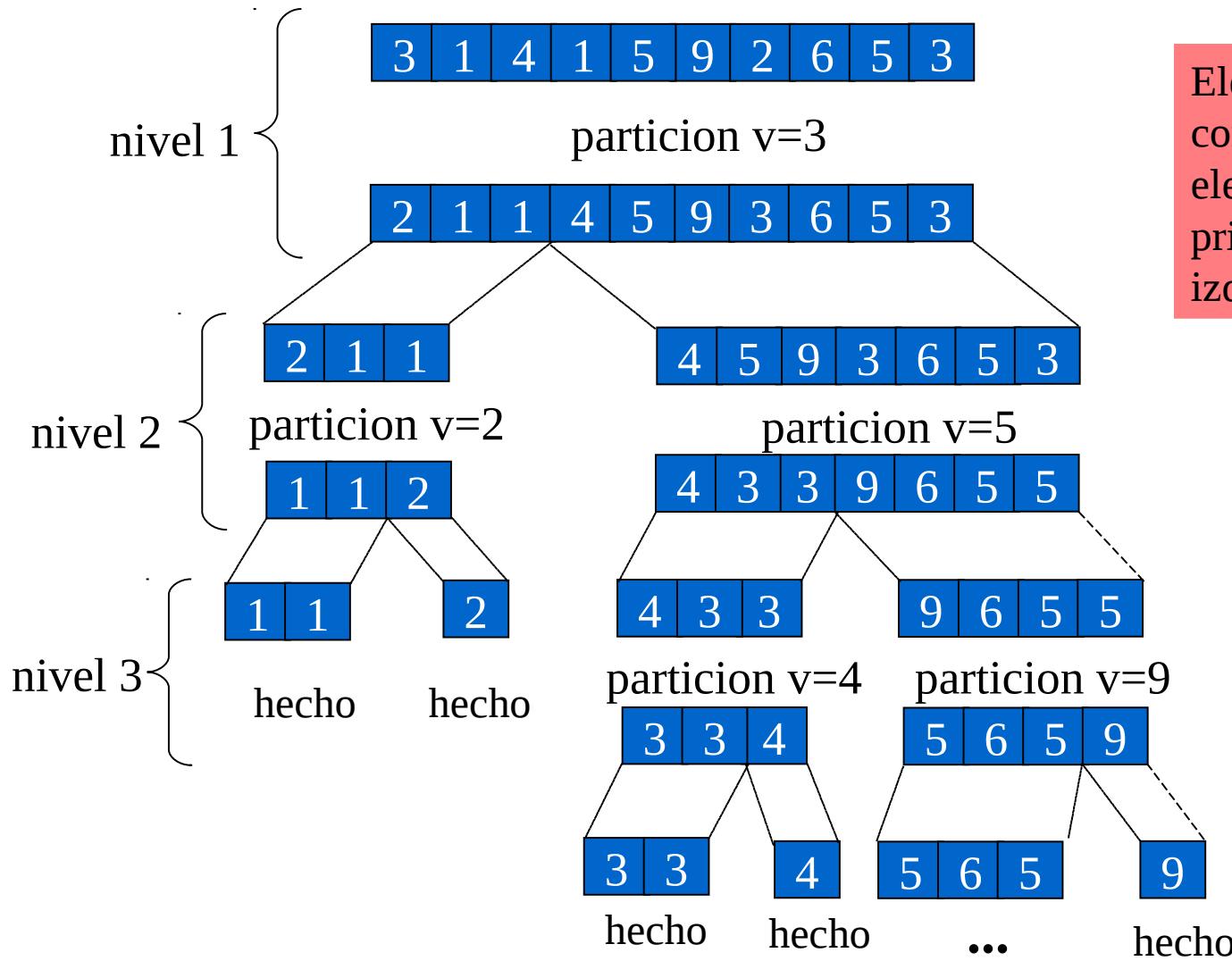
Si quisieramos ordenar

$$(3, 1, 4, 1, 5, 9, 2, 6, 5, 3)$$

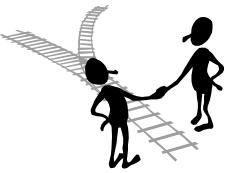
El pivote sería 3, y el algoritmo se desarrollaría de la siguiente manera



Ejemplo

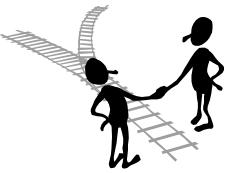


Elegimos el pivote como el mayor elemento de los dos primeros por la izquierda



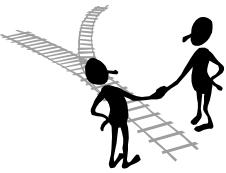
Eficiencia de quicksort

- 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



Análisis de Quicksort

- Recordemos que el algoritmo de ordenación por Inserción hacía aproximadamente $(1/2)n^2 - 1/n$ 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 también de selección).
- Es que el número de intercambios que hace quicksort es unas 3 veces el número de intercambios que hace el de inserción.
- Sin embargo, en la práctica quicksort es el mejor algoritmo de ordenación que se conoce...
- ¿Qué pasará con el tiempo del caso promedio?



Análisis del caso promedio

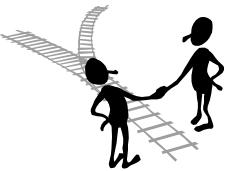
- Suponemos que la lista está dada en orden aleatorio
- Suponemos que todos los posibles ordenes del array son igualmente probables
- El pivote puede ser cualquier elemento
- Puede demostrarse que en el caso promedio quicksort tiene un tiempo $T(n) = 2n \ln n + O(n)$, que se debe al número de comparaciones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

Algorítmica

Capítulo 2: Algoritmos Divide y Vencerás

Tema 6: Aplicaciones

-
- **Otras aplicaciones**
 - Multiplicación de enteros
 - Multiplicación de matrices.
 - Método de Strassen.
 - Multiplicación de polinomios
 - El problema del “skyline”

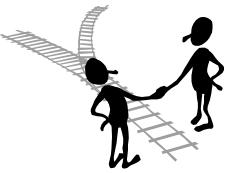


Multiplicación de enteros

- El problema que consideramos es el de la multiplicación de enteros muy grandes.
- Sean x e y dos números de n bits.
- Los métodos tradicionales de multiplicación requieren $O(n^2)$ operaciones sobre los bits:

$$\begin{array}{r} x = 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \\ y = 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \\ \hline 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \\ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \\ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \\ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \\ \hline 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \end{array}$$

$$\begin{array}{r} x = 1 \ 0 \ 1 \ 1 \\ y = 0 \ 1 \ 1 \ 0 \\ \hline 1 \ 1 \ 0 \ 1 \ 1 \\ 1 \ 0 \ 1 \ 1 \\ \hline 1 \ 0 \ 0 \ 0 \ 1 \ 0 \end{array}$$

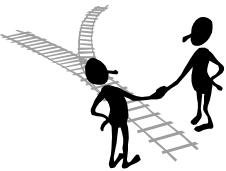


Multiplicación de enteros

- Supongamos que $n = 2^m$,
- Tratamos a x e y como dos strings de n bits y los partimos en dos mitades como $x = a * b$, $y = c * d$, de manera que las concatenaciones de a, b y de c, d reproducen x e y .
- Como son números binarios, podemos expresarlos como
$$xy = (a \cdot 2^{n/2} + b)(c \cdot 2^{n/2} + d) \\ = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd \dots\dots\dots (*)$$

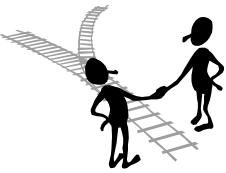
Ejemplo: Si $x = 13 \rightarrow x = 1101$ ($n = 4$) $\rightarrow A = 3$ y $B = 1 \rightarrow 13 = A \cdot 2^{n/2} + B = 3 \cdot 2^{n/2} + 1 = (11)_{(2)} \cdot 2^{n/2} + (01)_{(2)}$.

- De esta manera el cálculo de xy se simplifica a la realización de cuatro multiplicaciones de números de $(n/2)$ -bits y algunas adiciones y desplazamientos de 0 y 1 (las multiplicaciones son por potencias de 2).



Multiplicación de enteros

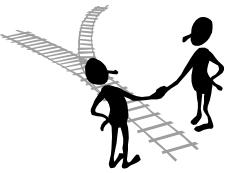
- Sea $T(n)$ el número de operaciones requeridas para multiplicar los dos números de n bits.



Multiplicación de enteros

- Sea $T(n)$ el número de operaciones requeridas para multiplicar los dos números de n bits.
- Está claro que,

$$\begin{aligned} T(n) &= 4T(n/2) + cn \\ &= \\ &= \\ &= \\ &= \\ &= \\ &= \\ &= \end{aligned}$$



Multiplicación de enteros

- Sea $T(n)$ el número de operaciones requeridas para multiplicar los dos números de n bits.
- Esta claro que,

$$\begin{aligned} T(n) &= 4T(n/2) + cn \\ &= 4(4T(n/2^2) + cn/2) + cn \\ &= 4^2T(n/2^2) + 2cn + cn \\ &= 4^3T(n/2^3) + 2^2cn + 2cn + cn \\ &\quad \vdots \\ &= 4^mT(1) + (2^{m-1} + \dots + 1) cn \\ &= 2^m2^m + (2^m - 1) cn \\ &\equiv O(n^2) \end{aligned}$$

- Con lo que se demuestra que no hemos adelantado nada



Multiplicación de enteros

Pero, consideremos la siguiente recurrencia

$$T(n) = \begin{cases} & \text{si } n = 1 \\ & \lceil(n/c) + bn \quad \text{si } n > 1 \end{cases}$$

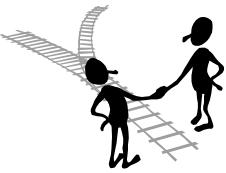
tomando $n = c^m$ podemos resolverla y obtener ($r = a/c$)

$$a < c ; r < 1, r^m \rightarrow 0, T(n) \rightarrow \frac{bc}{c-a} n = O(n)$$

$$a = c ; r = 1, T(n) = bn(m+1) = O(n\log n)$$

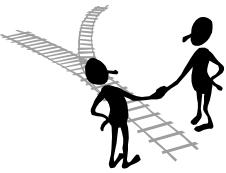
$$\text{when } a > c ; r > 1, T(n) \rightarrow bn r^m = bn \left(\frac{a}{c}\right)^{\log_c n} = ba^{\log_c n} = O(n^{\log_c a})$$

- $T(n)$ depende del numero de subproblemas y de su tamaño.
- Si el numero de subproblemas fuera 1, 3, 4, 8, entonces los algoritmos serian de ordenes n , $n^{\log 3}$, n^2 , n^3 respectivamente.



Multiplicación de enteros

- Si observamos la situación, en nuestro caso, la multiplicación de enteros va a producir algoritmos $O(n^2)$ si el número de subproblem as de tamaño mitad es cuatro, ya que el 4 que acompaña al $t(n/2)$ es el que nos hace el orden cuadrático,
- Por tanto intentamos disminuir el número de subproblemas DV (concretamente con tres productos en vez de 4).
- Para reducir la complejidad en tiempo intentaremos reducir el número de subproblemas (y por tanto de multiplicaciones) a costa de hacer mas adiciones y multiplicaciones por potencias de dos, y haremos así una multiplicación menos
- El truco es especialmente significativo conforme mas grande es n , es decir, asintóticamente



Multiplicación de enteros

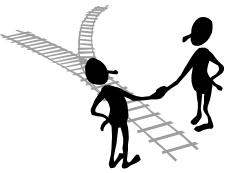
- Teníamos

$$xy = (a2^{n/2} + b)(c2^{n/2} + d)$$

- Ahora

$$x \cdot y = ac \cdot 2^n + [(a-b)(d-c) + ac+bd] \cdot 2^{n/2} + bd$$

- Es evidente que con esta forma de multiplicar x e y solo tenemos que hacer
 - tres multiplicaciones de dos números de $(n/2)$ -bits
 - seis adiciones y
 - multiplicaciones por potencias de 2
- Por tanto el correspondiente algoritmo DV debe ser mas eficiente



Multiplicación de enteros

- Podemos usar la rutina de multiplicación recursivamente para calcular los tres productos. Las adiciones y desplazamientos requieren un tiempo $O(n)$. Así, la complejidad en tiempo de la multiplicación de dos enteros de n bits está acotada superiormente por

$$T(n) = \begin{cases} pn^2 & \text{si } n \leq n_0 \\ T(n/2) + kn & \text{si } n \geq n_0 \end{cases}$$

- Donde k es una constante que incorpora las adiciones y las transferencias de bits.
- Es evidente que el tiempo de este algoritmo es $O(n^{\log 3}) = O(n^{1.59})$.



Ejemplo

$$\begin{array}{r} x = 1 \ 0 \ 1 \ 1 \\ y = 0 \ 1 \ 1 \ 0 \end{array}$$

$$\begin{array}{r} a = 1 \ 0 \\ c = 0 \ 1 \end{array}$$

$$\begin{array}{r} b = 1 \ 1 \\ d = 1 \ 0 \end{array}$$

$$u = (a+b)(c+d) = (1\ 0\ 1)(1\ 1) = 1\ 1\ 1\ 1$$

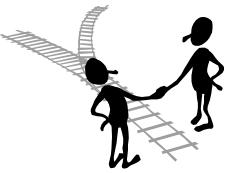
$$v = ac = (1\ 0)(0\ 1) = 1\ 0$$

$$w = bd = (1\ 1)(1\ 0) = 1\ 1\ 0$$

$$z = u - v - w = 1\ 1\ 1$$

$$xy = v2^4 + z2^2 + w = 1\ 0\ 0\ 0\ 0\ 0 + 1\ 1\ 1\ 0\ 0 + 1\ 1\ 0 = 1\ 0\ 0\ 0\ 0\ 1\ 0$$

Este mismo enfoque puede emplearse para diseñar un algoritmo que multiplique dos polinomios de grado n

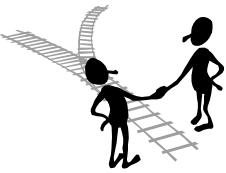


Multiplicación de matrices

- 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 mas (?) eficiente para multiplicar matrices.



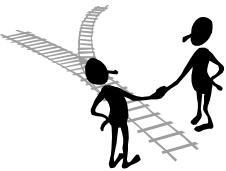
Multiplicación de matrices

La multiplicación puede hacerse como sigue:

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

\uparrow \uparrow \uparrow
 C A B

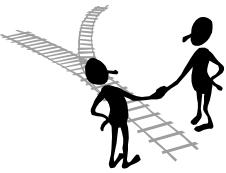
- Esta formulación divide una matriz $n \times n$ en matrices de tamaños $n/2 \times n/2$, con lo que divide el problema en 8 subproblemas de tamaños $n/2$.
- Notese que n se usa como tamaño del caso aunque la dimensión de la matriz es n^2
- Este enfoque da la siguiente recurrencia,



Multiplicación de matrices

$$T(n) = \begin{cases} & \text{si } n = 1 \\ & \cup \Gamma(n/2) + bn^2 \text{ si } n > 1 \end{cases}$$

- A partir de la cual, es evidente que $T(n)$ sigue siendo $O(n^3)$.
- Pero, basándonos en el enfoque DV que empleamos para multiplicar enteros, la multiplicación de las matrices también puede calcularse como sigue.



El método de Strassen

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

\uparrow \uparrow \uparrow
 C A B

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

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

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

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

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

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

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

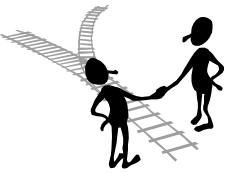
$$r = P+S-T+V$$

$$s = R+T$$

$$t = Q+S$$

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

- Es evidente que solo se necesitan 7 multiplicaciones y 18 adiciones/sustracciones, en lugar de las anteriores 8.



El método de Strassen

$$A \times B = C$$

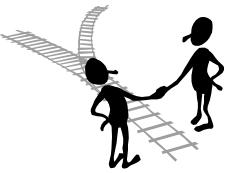
$$\begin{array}{|c|c|} \hline A_0 & A_1 \\ \hline A_2 & A_3 \\ \hline \end{array} \times \begin{array}{|c|c|} \hline B_0 & B_1 \\ \hline B_2 & B_3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline A_0 \times B_0 + A_1 \times B_2 & A_0 \times B_1 + A_1 \times B_3 \\ \hline A_2 \times B_0 + A_3 \times B_2 & A_2 \times B_1 + A_3 \times B_3 \\ \hline \end{array}$$

Divide las matrices en submatrices A_0, A_1, A_2 etc.

Recursivamente sigue dividiendo las submatrices en otras submatrices

Usa ecuaciones de multiplicación de matrices en bloques

Multiplica recursivamente las submatrices

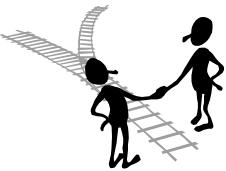


El método de Strassen

$$\begin{aligned} T(n) &= 7T(n/2) + bn^2 \\ &= \tilde{O}(7^m) = O(n^{\log 7}) = O(n^{2.81}) \end{aligned}$$

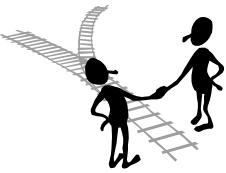
Se conocen mejoras del algoritmo, pero en la práctica las rebajas que consiguen son a costa de grandes aumentos en los valores de las correspondientes constantes ocultas

¿Y si las matrices no fueran cuadradas ?



El método de Strassen: Comentarios

- Algoritmo de Strassen: $O(n^{2.80736})$
- Algoritmo de Coppersmith–Winograd (1990): $O(n^{2.376})$
 - Frecuentemente usado como subprocedimiento de otros algoritmos para calcular mejoras teóricas en las cotas de eficiencia.
 - No se usa en la práctica ya que solo proporciona mejoras efectivas cuando se trata de multiplicar matrices extremadamente grandes
- Mejor cota alcanzada hasta la fecha: $O(n^{2.3727})$ en 2011
- ¿Cuál será la eficiencia del mejor caso?
 - $\Omega(n^2)$
 - Al menos tendrémos que llenar la matriz de la respuesta



Multiplicación de enteros

- Al método de multiplicación de enteros se le conoce con el nombre de **Algoritmo de Karatsuba**
- Sean a, b dos números con dos dígitos cada uno. Los partimos en dos

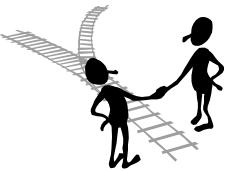
$$a = p \times 10 + q \text{ y } b = r \times 10 + s.$$

- Si $a = 78, b = 21$, entonces

$$p = 7, q = 8, \text{ and } r = 2, s = 1.$$

$$\begin{aligned}a \times b &= (p \times 10 + q) \times (r \times 10 + s) \\&= (p \times r) \times 100 + (p \times s + q \times r) \times 10 + q \times s.\end{aligned}$$

- $78 \cdot 21 = (7 \cdot 2) \cdot 100 + (7 \cdot 1 + 8 \cdot 2) \cdot 10 + 8 \cdot 1 = 1638.$
- 4 multiplicaciones de números de un digito + sumas



Algoritmo de Karatsuba (n =2)

- **Karatsuba:** multiplicar dos números de 2 dígitos utilizando tres multiplicaciones de números de 1 dígito

- $u = p \times r, v = (q - p) \times (s - r), w = q \times s.$

- $a \times b = u \times 10^2 + (u + w - v) \times 10 + w.$

- $u + w - v = p \times r + q \times s - (q - p) \times (s - r)$

- $= p \times s + q \times r.$

- En nuestro ejemplo

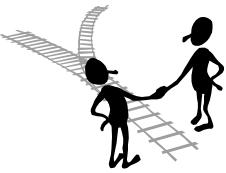
- $u = 7 \times 2 = 14, v = (8 - 7) \times (1 - 2) = -1, w = 8 \times 1 = 8.$

- $78 \times 21 = 14 \times 100 + (14 + 8 - (-1)) \times 10 + 8$

$$= 1400 + 230 + 8$$

$$= 1638.$$

$$\begin{aligned} a &= p \times 10 + q \\ b &= r \times 10 + s \end{aligned}$$



Algoritmo de Karatsuba (n =4)

- Cada mitad es un número de dos dígitos
- $a = p \times 10^2 + q$ y $b = r \times 10^2 + s.$
- $u = p \times r, v = (q - p) \times (s - r), w = q \times s.$
- Ahora $a \times b = u \times 10^4 + (u + w - v) \times 10^2 + w.$
- Ejemplo $a = 5678$ and $b = 4321.$

$$a \rightarrow p = 56, q = 78; b \rightarrow r = 43 y s = 21.$$

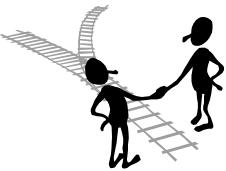
$$u = 56 \times 43 = 2408,$$

$$v = (78 - 56) \times (21 - 43) = -484,$$

$$w = 78 \times 21 = 1638.$$

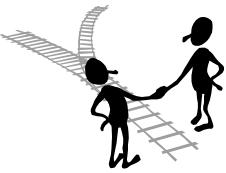
$$\begin{aligned} 5678 \times 4321 &= 2408 \times 10000 + (2408 + 1638 - (-484)) \times 100 + 1638 \\ &= 24080000 + 453000 + 1638 \\ &= 24534638. \end{aligned}$$

(producto de dos
números de 2 dígitos)



Algoritmo de Karatsuba

digits	Karatsuba	long multiplication
$1 = 2^0$	1	1
$2 = 2^1$	3	4
$4 = 2^2$	9	16
$8 = 2^3$	27	64
$16 = 2^4$	81	256
$32 = 2^5$	243	1024
$64 = 2^6$	729	4096
$128 = 2^7$	2187	16 384
$256 = 2^8$	6561	65 536
$512 = 2^9$	19 638	262 144
$1024 = 2^{10}$	59 049	1 048 576
$1 048 576 = 2^{20}$	3 486 784 401	1 099 511 627 776
...
$n = 2^k$	3^k	4^k



Multiplicación de polinomios

- Supongamos dos polinomios:

$$P(x) = p_0 + p_1 x + p_2 x^2 + \dots + p_{n-1} x^{n-1}$$

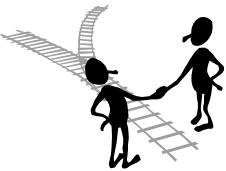
$$Q(x) = q_0 + q_1 x + q_2 x^2 + \dots + q_{n-1} x^{n-1}$$

- Con el exponente n par: hay n coeficientes y el grado de cada polinomio es n – 1.
- DV sugiere dividir los polinomios por la mitad, quedando de la siguiente manera:
- P(x):

$$P_I(x) = p_0 + p_1 x + p_2 x^2 + \dots + p_{n/2-1} x^{n/2-1}$$

$$P_D(x) = p_{n/2} + p_{n/2+1} x + p_{n/2+2} x^2 + \dots + p_{n-1} x^{n/2-1}$$

- Q(x) (análogo a P(x)): Q_I(x) y Q_D(x)



Multiplicacion de polinomios

- Entonces:

$$P(x) = P_I(x) + P_D(x)x^{n/2}$$

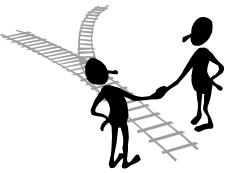
$$Q(x) = Q_I(x) + Q_D(x)x^{n/2}$$

- El algoritmo DV nos llevaría a:

$$P(x) \cdot Q(x) =$$

$$P_I(x)Q_I(x) + (P_I(x)Q_D(x) + P_D(x)Q_I(x))x^{n/2} + P_D(x)Q_D(x)x^n$$

- El algoritmo subyacente es de orden cuadrado, como el convencional que se desprende de la multiplicación de polinomios



Multiplicación de polinomios

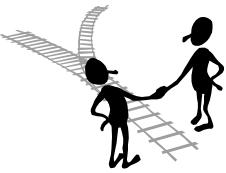
- Con el mismo método que en los problemas anteriores, la multiplicación de los polinomios puede hacerse teniendo en cuenta que

$$R_I(x) = P_I(x) Q_I(x)$$

$$R_D(x) = P_D(x) Q_D(x)$$

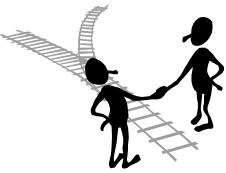
$$R_H(x) = (P_I(x) + P_D(x)) (Q_I(x) + Q_D(x))$$

- Entonces la multiplicación de P por Q queda:
- $P(x) \cdot Q(x) = R_I(x) + (R_H(x) - R_I(x) - R_D(x)) x^{n/2} + R_D(x) x^n$
- Que conlleva “solo” tres multiplicaciones de polinomios de grado mitad que los originales



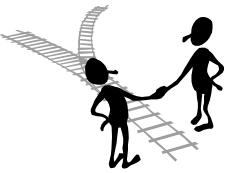
Ejemplo

- $P(x) = 1 + x + 3x^2 - 4x^3$
- $Q(x) = 1 + 2x - 5x^2 - 3x^3$
- Entonces,
$$R_I(x) = (1 + x)(1 + 2x) = 1 + 3x + 2x^2$$
$$R_D(x) = (3 - 4x)(-5 - 3x) = -15 + 11x + 12x^2$$
$$R_H(x) = (4 - 3x)(-4 - x) = -16 + 8x + 3x^2$$
- y solo queda hacer las multiplicaciones



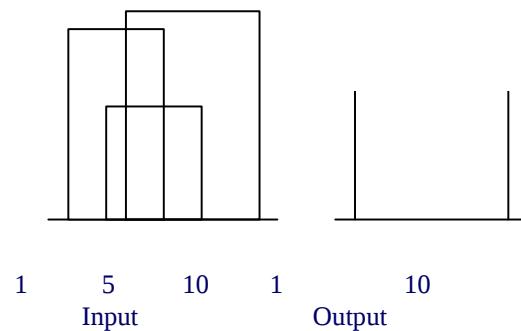
El problema de la linea del horizonte

- Con la comercialización y popularización de las estaciones de trabajo gráficas de alta velocidad, el CAD (“computer-aided design”) y otras areas (CAM, diseño VLSI) hacen un uso masivo y efectivo de los computadores.
- Un importante problema a la hora de dibujar imágenes con computadores es la eliminación de líneas ocultas, es decir, la eliminación de líneas que quedan ocultadas por otras partes del dibujo.
- Este interesante problema recibe el nombre de **Problema de la Linea del Horizonte ("Skyline Problem")**.

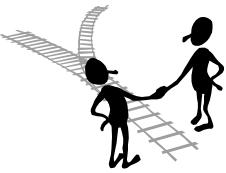


El problema de la linea del horizonte

- El problema se establece en los siguientes sencillos términos
- Dadas las situaciones exactas (coordenadas)
 - Por ejemplo (1,11,5), (2.5,6,7), (2.8,13,10), donde cada valor es la coordenada izquierda, la altura y la coordenada derecha de un edificio
- y las formas de n edificios rectangulares en una ciudad bi-dimensional,

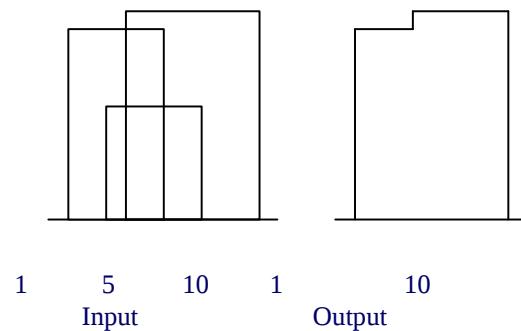


- construir un algoritmo Divide y Vencerás que calcule eficientemente el "skyline " (en dos dimensiones) de esos edificios, eliminando las líneas ocultas

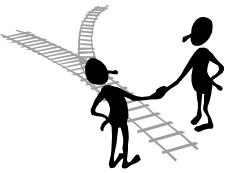


El problema de la linea del horizonte

- El problema se establece en los siguientes sencillos términos
- Dadas las situaciones exactas (coordenadas)
 - Por ejemplo (1,11,5), (2.5,6,7), (2.8,13,10), donde cada valor es la coordenada izquierda, la altura y la coordenada derecha de un edificio
- y las formas de n edificios rectangulares en una ciudad bi-dimensional,

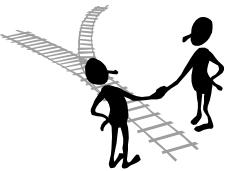


- construir un algoritmo Divide y Vencerás que calcule eficientemente el "skyline" (en dos dimensiones) de esos edificios, eliminando las líneas ocultas



Particularización del método DV

- **1. Tamaño:** el problema es una colección de edificios, por tanto el número de edificios en una colección es una elección natural para la forma de medir el tamaño del problema, así
 - $\text{Tamaño(Edificios)} = \text{número de edificios en el input.}$
- **2. Caso base del problema:** Como el input está restringido a la colección de edificios, entonces el caso base del problema es una colección constituida por un solo edificio, puesto que el "skyline" de un único edificio es el mismo edificio. Por tanto
 - $\text{TamañoCasoBase} = 1$

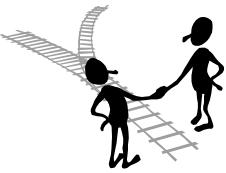


Particularización del método DV

- **3. Solución del Caso Base:** Como la solución para un único edificio es el edificio en si mismo, el cuerpo del procedimiento EncuentraSolucionCasoBase(Edificios; Skyline)

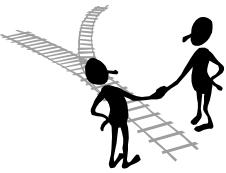
puede definirse como sigue:

```
Procedimiento EncuentraSolucionCasoBase(Edificios; Skyline()
Begin
  Skyline = Edificios[1[
end;
```



Particularización del método DV

- **4. División del problema:** El único requerimiento es que el tamaño de los subproblemas obtenidos, de la colección de edificios, debe ser estrictamente menor que el tamaño del problema original.
- Teniendo en cuenta la complejidad en tiempo del algoritmo, funcionará mejor si los tamaños de los subproblemas son parecidos.
- Diseñaremos un procedimiento de división de manera que los dos subproblemas que obtendremos, EdificiosIzquierda y EdificiosDerecha, tengan tamaños aproximadamente iguales
- **5. Combinación:** Hay que combinar dos "skylines" subsoluciones en un único "skyline" :
 - En esencia la parte de combinación del algoritmo lo que hace es tomar dos "skylines ", los analiza punto a punto desde la izquierda a la derecha, y en cada punto toma el mayor valor de los dos "skylines ".
 - Ese punto es el que toma como valor del "skyline" combinado



Algoritmo DV del Skyline

Procedimiento Skyline(Edificios,Skyline)

Begin

If CasoBase(Edificios)

Then

EncuentraSolucionCasoBase(Edificios,Skyline)

Else

Begin

Dividir(Edificios, EdificiosIzquierda, EdificiosDerecha)

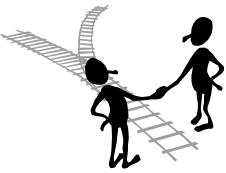
EncuentraSkyline(EdificiosIzquierda, SkylineIzquierda)

EncuentraSkyline(EdificiosDerecha, SkylineDerecha)

Combina(SkylineIzquierda, SkylineDerecha, Skyline)

End

End;

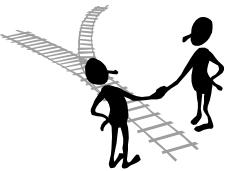


Algoritmo DV del "Skyline"

- Por tanto siguiendo la estrategia del DV, en ese algoritmo primero comprobamos si el array de edificios que nos dan contiene un solo edificio.
- Si es así, la solución es ese mismo edificio
- En caso contrario, dividimos el conjunto de edificios, resolvemos recursivamente cada mitad, y finalmente combinamos los dos "skylines" obtenidos
- El tiempo del algoritmo se encuentra a partir de la recurrencia

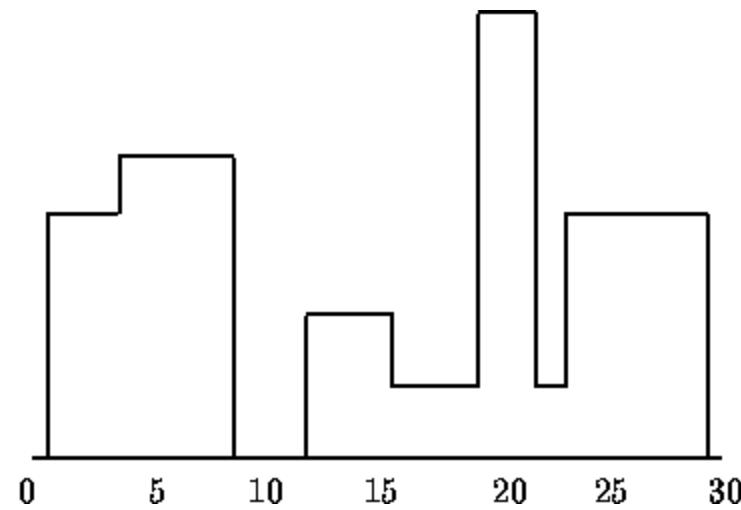
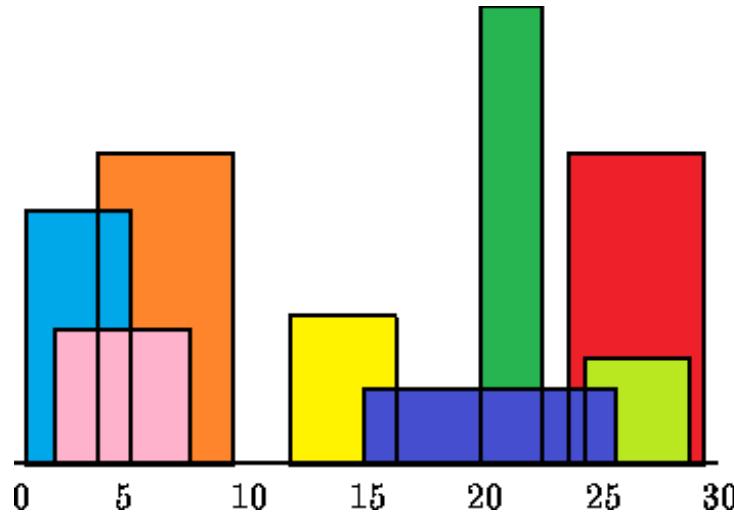
$$T(n) = 2T(n/2)+O(n)$$

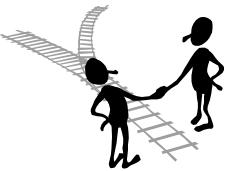
- Por tanto el algoritmo tiene orden $O(n\log n)$



Ejemplo del Algoritmo DV del Skyline

- Considerar el diagrama de la izquierda, en el que :
 - $(1,11,5), (2,6,7), (3,13,9), (12,7,16), (14,3,25), (19,18,22), (23,13,29), (24,4,28)$
- Obtener entonces el correspondiente Skyline
 - $(1, 11, 3, 13, 9, 0, 12, 7, 16, 3, 19, 18, 22, 3, 23, 13, 29, 0)$





Otras aplicaciones de los Algoritmos DV

- Compresión de datos
 - Códigos de Huffman
- Informática gráfica
 - Multiplicación de matrices
- Biología computacional
- Gestión de cadenas de aprovisionamiento
- Interpolación
- Organización de la carga de trabajo en supercomputadores
- Teoría de la Señal
 - Transformada Rápida de Fourier
 - Algoritmo de Horner para evaluar repetidamente polinomios

Algorítmica

Capítulo 3. Algoritmos Greedy*

Tema 7. Algoritmos Greedy

- El enfoque greedy
- Repaso de grafo

* Voraz en español

La filosofía greedy



¡Cómete siempre todo
lo que tengas a mano!

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

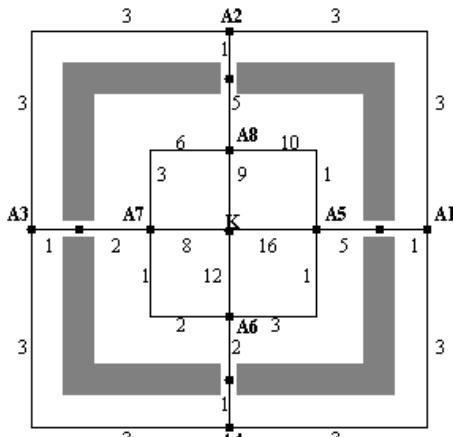
La filosofía greedy

Como funciona la técnica Greedy.

Suponemos las murallas de una ciudad, diferentes puntos dentro y fuera de ella y el tiempo necesario para movernos entre cada par de puntos, que suponemos depende de diversas razones (seguridad, distancia, ...).

Queremos encontrar el camino mas corto para llegar desde el punto A(1) hasta K, centro de la ciudad.

El enfoque greedy intentará ir al punto mas cercano al A(1) dentro de la ciudad, que es el A(5), empleando 6 unidades de tiempo, pero despues de eso, ya no hay forma de encontrar el camino de menor costo.



Camino Greedy

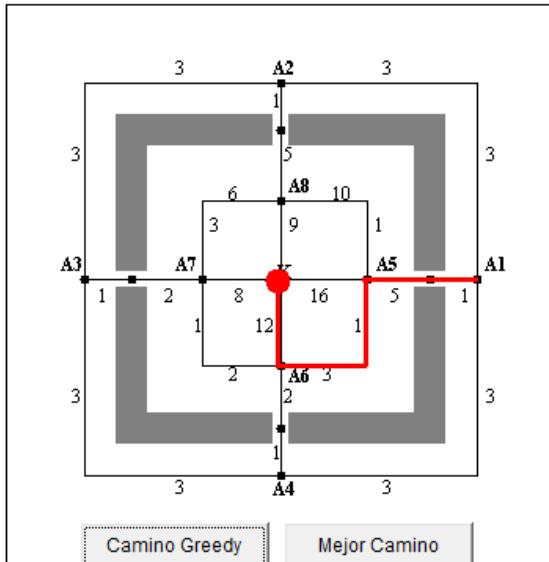
Mejor Camino

Como funciona la técnica Greedy.

Suponemos las murallas de una ciudad, diferentes puntos dentro y fuera de ella y el tiempo necesario para movernos entre cada par de puntos, que suponemos depende de diversas razones (seguridad, distancia, ...).

Queremos encontrar el camino mas corto para llegar desde el punto A(1) hasta K, centro de la ciudad.

El enfoque greedy intentará ir al punto mas cercano al A(1) dentro de la ciudad, que es el A(5), empleando 6 unidades de tiempo, pero despues de eso, ya no hay forma de encontrar el camino de menor costo.

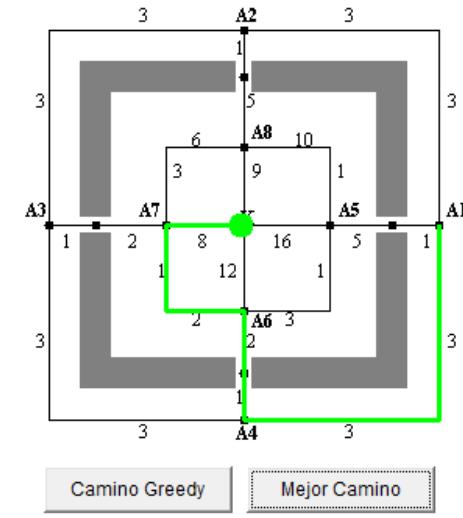


Como funciona la técnica Greedy.

Suponemos las murallas de una ciudad, diferentes puntos dentro y fuera de ella y el tiempo necesario para movernos entre cada par de puntos, que suponemos depende de diversas razones (seguridad, distancia, ...).

Queremos encontrar el camino mas corto para llegar desde el punto A(1) hasta K, centro de la ciudad.

El enfoque greedy intentará ir al punto mas cercano al A(1) dentro de la ciudad, que es el A(5), empleando 6 unidades de tiempo, pero despues de eso, ya no hay forma de encontrar el camino de menor costo.



El enfoque greedy

- La idea básica consiste en seleccionar en cada momento lo mejor de entre un conjunto de candidatos, sin tener en cuenta lo ya hecho, hasta obtener una solución para el problema.
- Cuando para resolver un problema se le aplica el **enfoque greedy**, el algoritmo resultante se denomina **Algoritmo Greedy**
- El enfoque greedy solo se puede aplicar a problemas que reúnan un conjunto de características: **Problemas Greedy**.
- Suelen ser problemas de optimización

Características de un problema greedy

- Para poder resolver un problema con un enfoque greedy, ha de reunir las 6 siguientes características, que no son necesarias, pero si suficientes:
 - 1 Un **conjunto de candidatos**: Las tareas a ejecutar, los nodos de un grafo, etc.
 - 2 Una lista de **candidatos ya usados**.
 - 3 Un criterio (**función**) **solución** que dice cuándo un conjunto de candidatos forma una solución (no necesariamente óptima).

Características de un problema greedy

- 4 Un criterio que dice cuándo un conjunto de candidatos (sin ser necesariamente una solución) es factible, es decir, podrá llegar a ser una solución (no necesariamente optima).
- 5 Una función de selección que indica en cualquier instante cuál es el candidato más prometedor de los no usados todavía.
- 6 Una función objetivo que a cada solución le asocia un valor, y que es la función que intentamos optimizar (a veces coincide con la de selección)

Enfoque greedy

- Un algoritmo Greedy procede siempre de la siguiente manera:
 - Se parte de un conjunto de candidatos a solución vacío: $S = \emptyset$
 - De la lista de candidatos 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
 - Si el candidato anterior no es válido, lo borramos de la lista de candidatos posibles, y nunca mas es considerado
 - Evaluamos la función objetivo. Si no estamos en el optimo
 - Seleccionamos con la función de selección otro candidato y repetimos el proceso anterior hasta alcanzar la solución.
- La primera solución que se consigue suele ser la Solución Óptima del problema

El enfoque greedy

- FUNCION GREEDY

$$S = \emptyset$$

Mientras S no sea una solución y $C \neq \emptyset$ Hacer:

X = elemento de C que maximiza $SELEC(X)$

$$C = C - \{X\}$$

Si $(S \cup \{X\})$ es factible Entonces $S = S \cup \{X\}$

Si S es una solución entonces devolver S

caso contrario “NO HAY SOLUCION”

- C es la lista de candidatos.
- 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.

Ejemplo

- Se desea dar cambio usando el menor número posible de monedas de 1, 5, 10 y 25
- ¿Es greedy el problema?:
 - Candidatos: 1, 5, 10, 25 (con una moneda de cada tipo por lo menos).
 - Usados: Podremos definirlo.
 - Solución: Lista de candidatos tal que la suma de los mismos coincida exactamente con el cambio pedido.
 - Criterio de factibilidad: Que no se supere el cambio.
 - Criterio de selección: Se escoge la moneda de mayor valor entre las disponibles.
 - Objetivo: El número de monedas ha de ser mínimo.

Ejemplo

- Si tenemos, por ejemplo, una moneda de 100 que queremos cambiar y la máquina dispone de 3 monedas de 25, 1 de 10, 2 de 5 y 25 de 1, entonces la primera solución que alcanza el algoritmo greedy directo es justamente la Solución Optima: 3 de 25, 1 de 10, 2 de 5 y 5 de 1.
- Pero si tenemos 10 monedas de 1, 5 de 5, 3 de 10, 3 de 12 y 2 de 25, la solución del algoritmo para una moneda de 100 sería 2 de 25, 3 de 12, 1 de 10 y 4 de 1, en total diez monedas.
- La solución no es óptima, ya que existe otra mejor que utiliza nueve monedas en vez de diez, que es 2 de 25, 3 de 10 y 4 de 5.

Greedy y Optimalidad

- Los algoritmos greedy no alcanzan soluciones optimales siempre
- Esta desventaja es una ventaja en problemas en los que es imposible (o muy difícil) alcanzar el óptimo: Heurísticas Greedy:
 - Problema del Coloreo de un Grafo,
 - Problema del Viajante de Comercio,
- Pueden alcanzar óptimos locales, pero no los óptimos globales de los problemas.
- Por eso en cada caso habrá que demostrar la corrección del algoritmo

Almacenamiento óptimo 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),

$$(1/n) \sum_{1 \leq j \leq n} t_j$$

- Nos piden encontrar una permutación de los n programas tal que cuando estén 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

- El problema reúne todos los ingredientes greedy
- Sea $n = 3$ y $(l_1, l_2, l_3) = (5, 10, 3)$

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	$3 + 3 + 10 + 3 + 10 + 5 = 34$

Solución Greedy

- Partiendo de la cinta vacía

Para i := 1 to n do

 grabar el siguiente programa mas corto
 ponerlo a continuación en la cinta

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

Almacenamiento óptimal en cintas

- Teorema
- Si $l_1 \leq l_2 \leq \dots \leq l_n$ entonces el orden de colocación $i_j = j$, $1 \leq j \leq n$ minimiza

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

- Para todas las posibles permutaciones de i_j
- Pero ¿siempre podremos considerar un óptimo local como uno global?

GREEDY Y OPTIMALIDAD



Greedy y Optimalidad

- Hay casos en los que un óptimo local podrá considerarse como global
- Típico problema de optimización, $f : R^n \rightarrow R$,

$$\text{Min } \{f(x) : x \in S\}$$

- A $\{x \in R^n : x \in S\}$ se le llama solución factible del problema.
- Si $x^* \in S : f(x^*) \leq f(x), \forall x \in S$, entonces x^* es un óptimo global.
- Si $x^* \in S$ y existe un entorno $V(x^*)$ tal que

$$f(x^*) \leq f(x), \forall x \in S \cap V(x^*)$$

entonces x^* es un óptimo local.

Greedy y Optimalidad

- Supongamos que S (el conjunto factible) es un conjunto contenido en R^n ($S \subseteq R^n$)
- S no vacío y convexo;
- Sea $f: S \rightarrow R$ y el problema

$$\text{Min: } f(x), x \in S$$

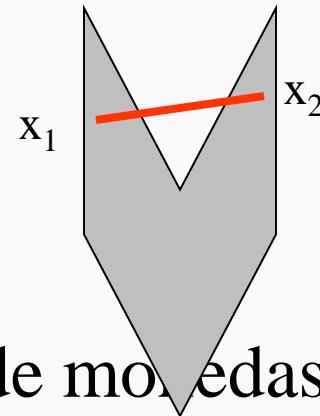
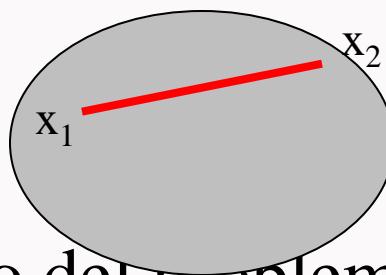
- Si x^* es un óptimo local, entonces si f es una **función convexa**, el óptimo local es un óptimo global.

Your company

¿Se verificaba esto en el anterior problema del cambio de monedas?

Convexidad

- S es un conjunto convexo cuando $\forall x_1, x_2 \in S$ y $\forall \lambda \in [0,1] \rightarrow \lambda x_1 + (1 - \lambda) x_2 \in S$, lo que se traduce como: “Dados dos puntos del conjunto S , todo el segmento lineal que los une está en el conjunto”.



- En el caso del problema del cambio de monedas, el conjunto factible no era convexo

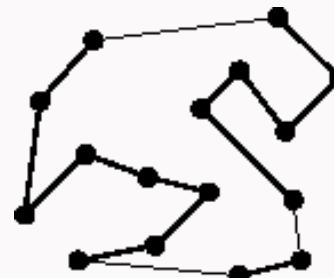
Algoritmos Greedy para Grafos

- ¿Por que hay que estudiar grafos?
- Podemos abstraer grafos de distintas situaciones físicas del mundo real para:
 - Resolver problemas de recorridos dando servicios eficientes a nuestros clientes o a los usuarios de los sistemas:
 - Por ejemplo el Problema del Viajante de Comercio
 - Diseñar Redes poco costosas de computadores de telefonía, etc.
- Son básicos en Inteligencia Artificial, pero también en Arquitectura y en otras muchas Ingenierías
- Desde luego en Robótica

Algoritmos Greedy para Grafos

- Supongamos que tenemos un robot que maneja un soldador.
- Para que el robot haga las soldaduras que queremos, debemos darle el orden en que debe visitar los puntos de soldadura, de modo que visite (y suelde) el primer punto, luego el segundo, etc., hasta que concluya su tarea
- Como los robots son caros, necesitamos encontrar el orden que minimiza el tiempo (es decir, la distancia recorrida) que tarda en realizar todas las soldaduras del panel

Your company name

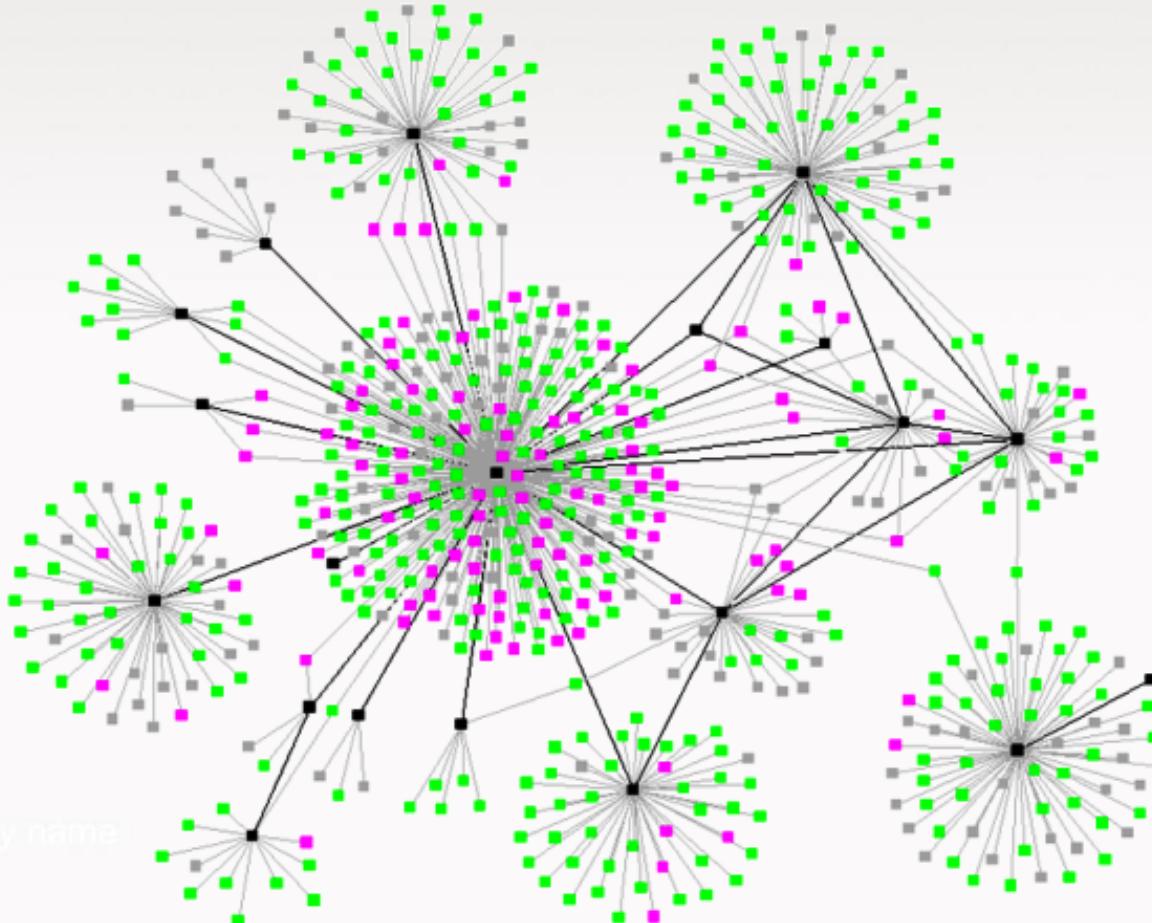


- Pero hay muchos mas...

Ejemplos de grafos



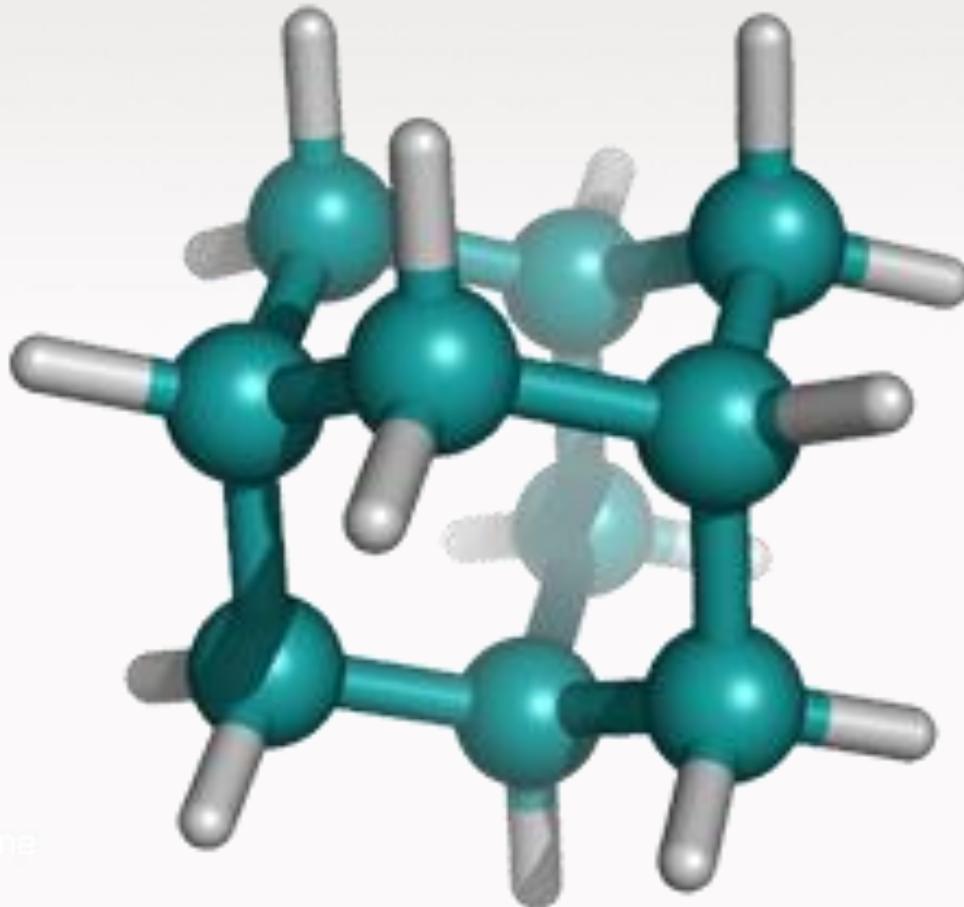
Ejemplos de grafos



**Redes
Sociales**

Your company name

Ejemplos de grafos



Your company name

Modelos químicos

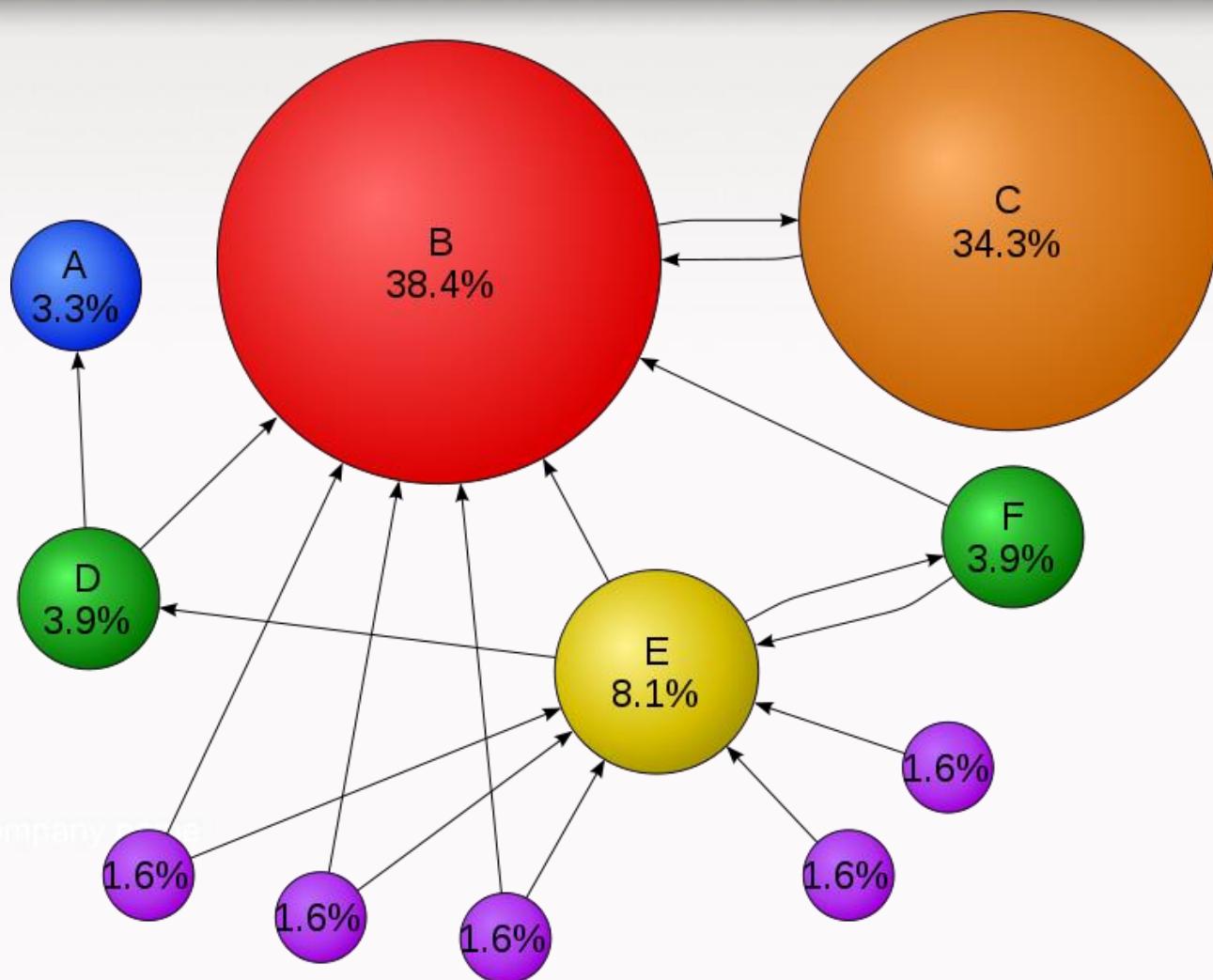
Grafos, internet y Google

- Google ve internet como un grafo gigante.
- Cada página web es un nodo, y las páginas están conectadas por aristas si existen links entre ellas.
- Notese que en internet las aristas SI tienen dirección (estrictamente habría que hablar de “arcos”).
- El algoritmo que usa Google para ordenar sus búsquedas se llama **PageRank**.

¿Como trabaja PageRank?

- **Idea:** cuantos mas links apúntan a una página, mas importante se supone que es esa página.
- **Segunda idea:** Si una página importante apúnta a tu página, eso es mucho mas importante que si lo hace una página sin importancia
- Por ejemplo, que nos refencie Wikipedia es mucho mas importante que el que lo haga la web de Panadería Pepi.

Ejemplo



¡PageRank se usa para ganar dinero!

- Las personas que saben como se usa PageRank pueden lucrarse a través del uso de ese algoritmo.
- Por ejemplo, negocios o personas individuales con altos valores de page rank pueden vender links a aquellos otros que desean elevar su page rank.
- También se usa algoritmos similares para ordenar las universidades en el mercado de trabajo (¡no todos somos iguales!)

Complemento: Social networking

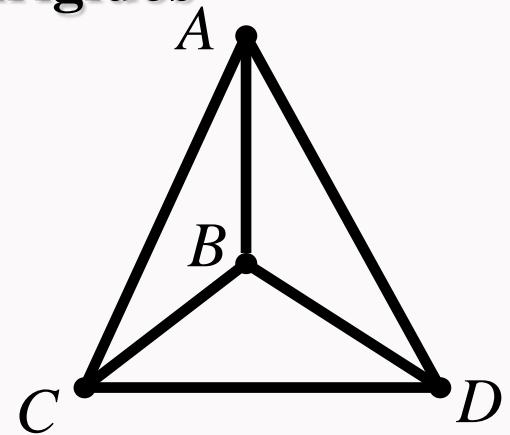
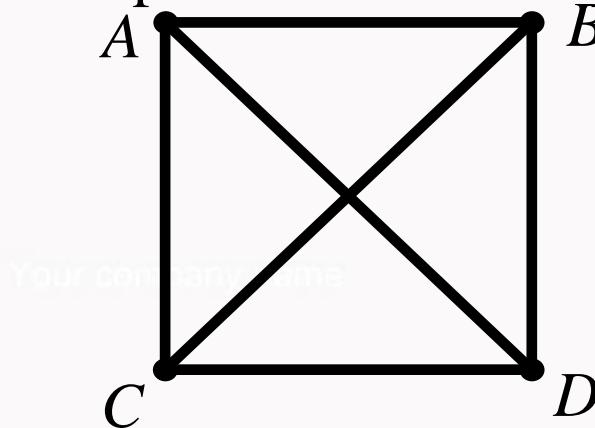
- Los grafos también son importantes para los sitios de “social networking” como Facebook.



- Analizando las preferencias de nuestros amigos y las páginas que visitamos (“like”), Facebook puede dirigir de manera muy acertada su publicidad.

Nociones básicas de grafos

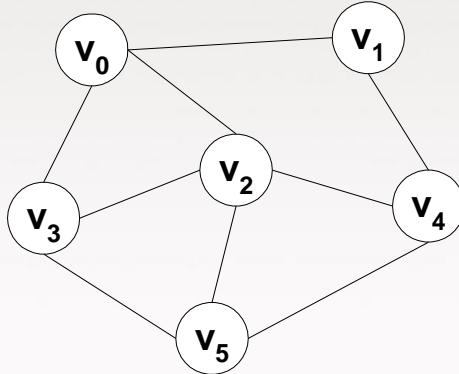
- Un grafo se define con dos conjuntos:
 - Un conjunto de **vértices** (nodos), y
 - Un conjunto de **aristas**
- Cuando las aristas tienen origen y final (dirección), se habla de **Grafos Dirigidos**, y en lugar de aristas tendremos arcos. Tambien hay **Grafos Ponderados**
- Supondremos en lo que sigue **Grafos no Dirigidos**



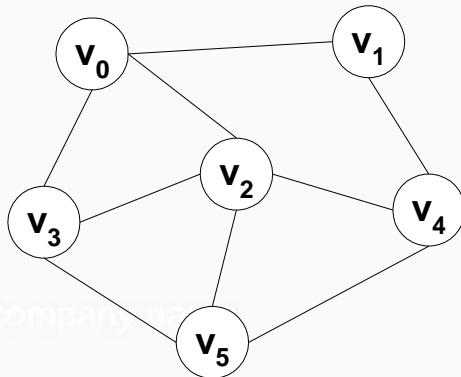
Definición formal

- Sea $X = \{x_1, x_2, \dots, x_n\}$ un conjunto finito, no vacío. Sea $A \subseteq X \times X$ una relación. Al par $G = (X, A)$ se le llama **grafo dirigido**.
- Sea $I_X = \{(x_i, x_i), i = 1, 2, \dots, n\}$ y $X_-^2 = (X \times X) - I_X$ (para eliminar lazos). Definimos en X_-^2 una relación R tal que:
$$(x_i, x_j)R(x_h, x_k) \Leftrightarrow (x_i, x_j) = (x_h, x_k) \text{ ó } (x_i, x_j) = (x_k, x_h) \text{ (para quitar la dirección de los puntos)}$$
- R es una relación de equivalencia. Definimos el conjunto cociente (X_-^2 / R) para esa relación. Al par $G = (X, B)$, con $B \subseteq X_-^2 / R$ se le llama **grafo no dirigido**

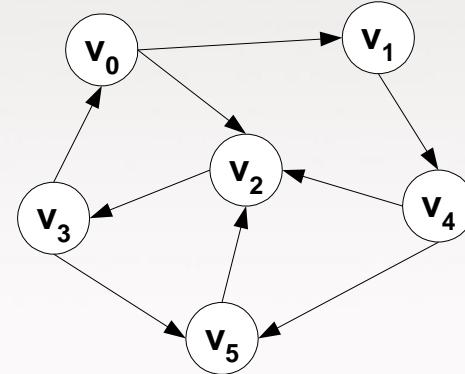
Ejemplos



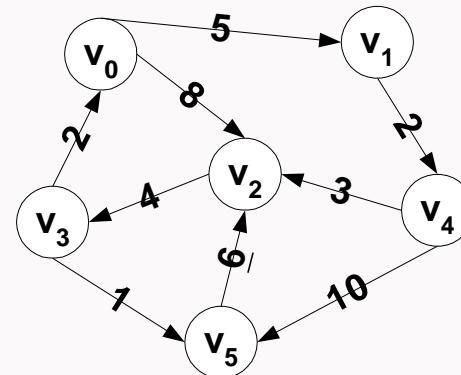
No Dirigido



No Ponderado



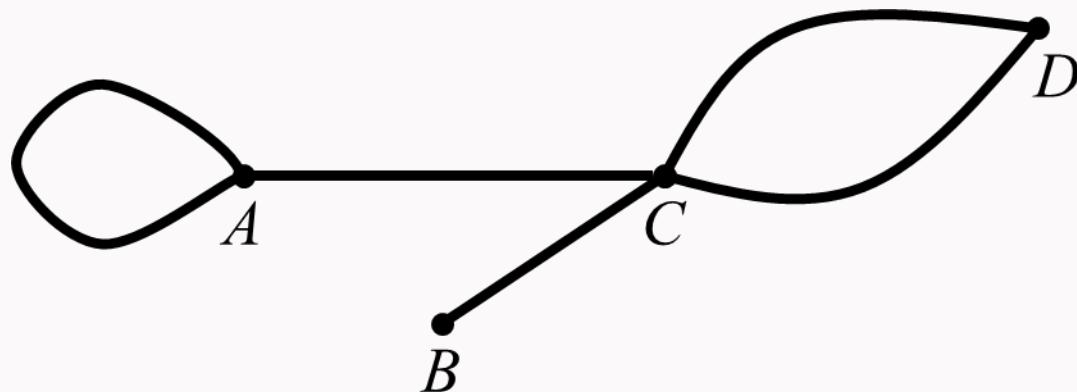
Dirigido



Ponderado

Nociones básicas de grafos

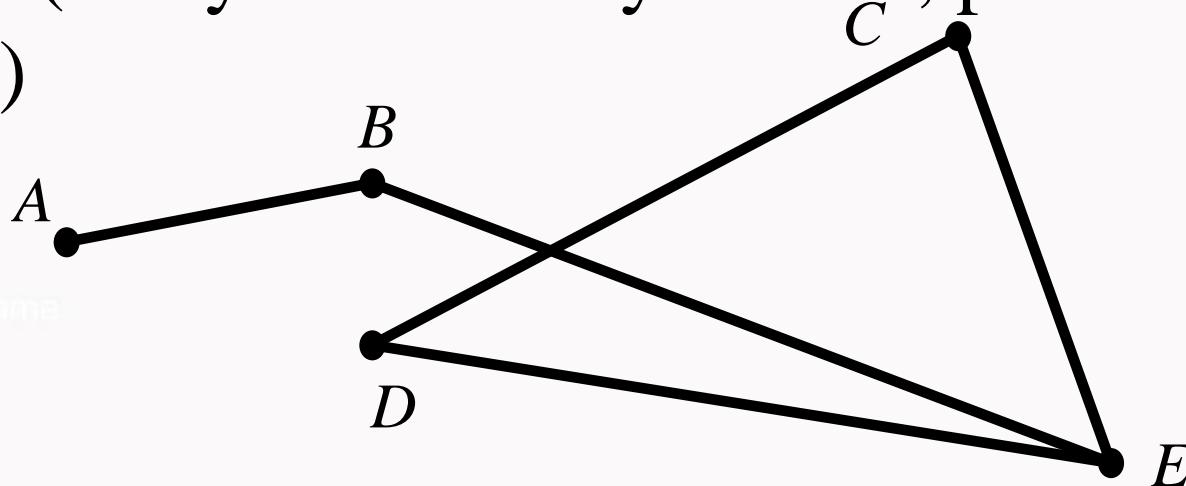
- Podemos unir dos vértices varias veces, obteniendo múltiples aristas
- Podemos unir un vértice a si mismo, para formar un lazo



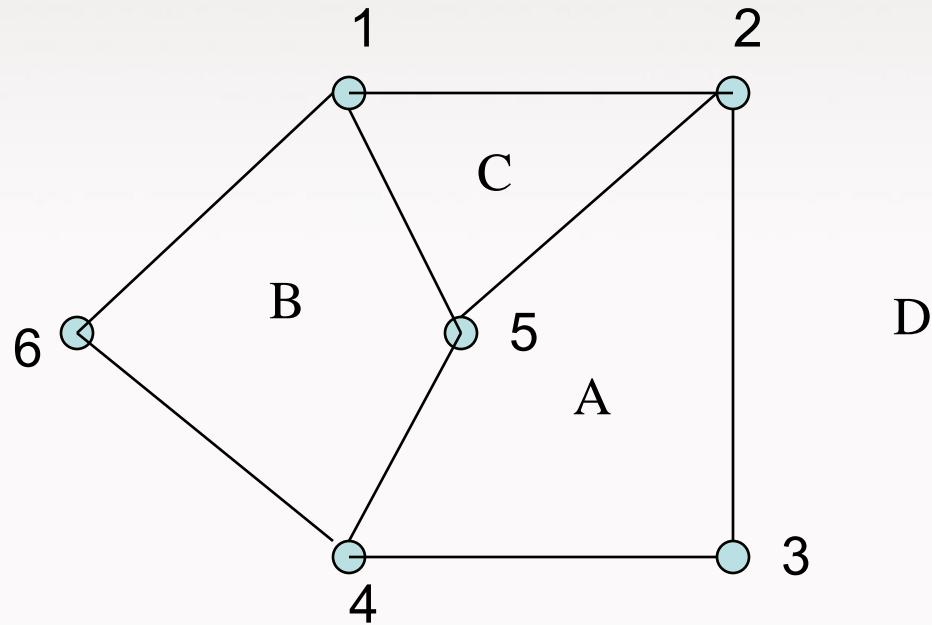
- Un grafo es completo si cualquier par de vértices distintos está unido por una arista

Nociones básicas de grafos

- Dos vértices son adyacentes si existe una arista que los une (B y E son adyacentes, pero el B y el D no)
- Dos aristas son adyacentes si comparten un vertice (AB y BE son adyacentes, pero las AB y CE no)



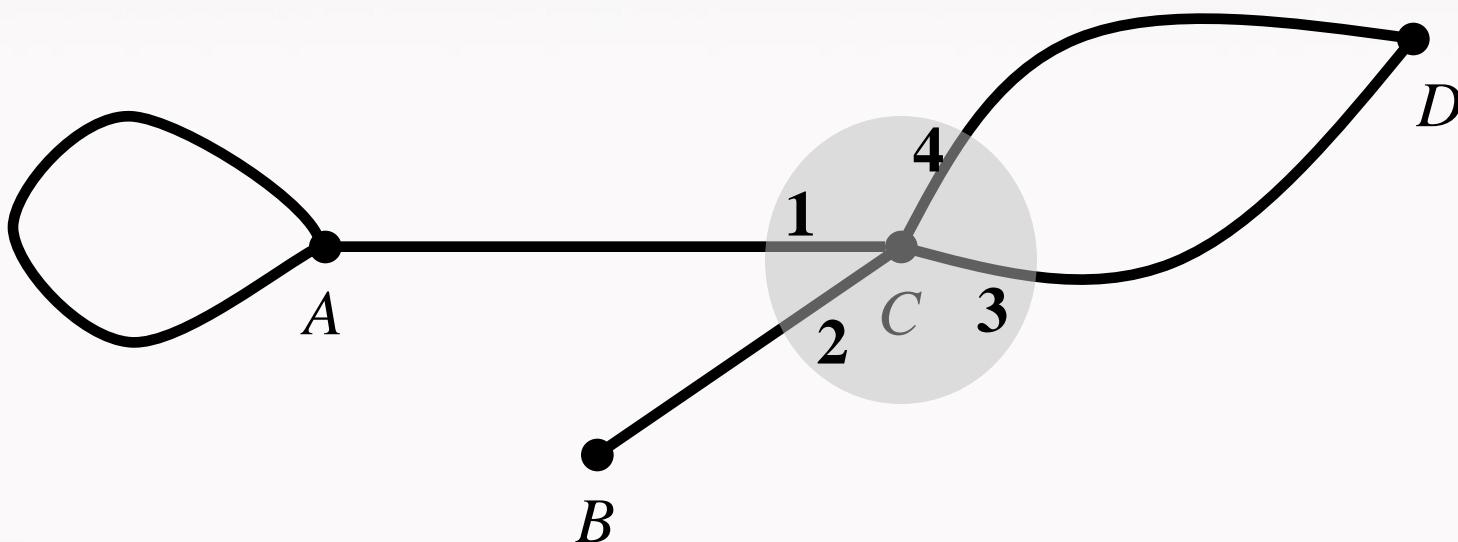
Nociones básicas de grafos



Un grafo se llama plano si no puede pintarse en el plano (o en una esfera) sin que se crucen sus aristas.

Nociones básicas de grafos

- El grado de un vértice es el número de aristas que pasan por ese vértice.

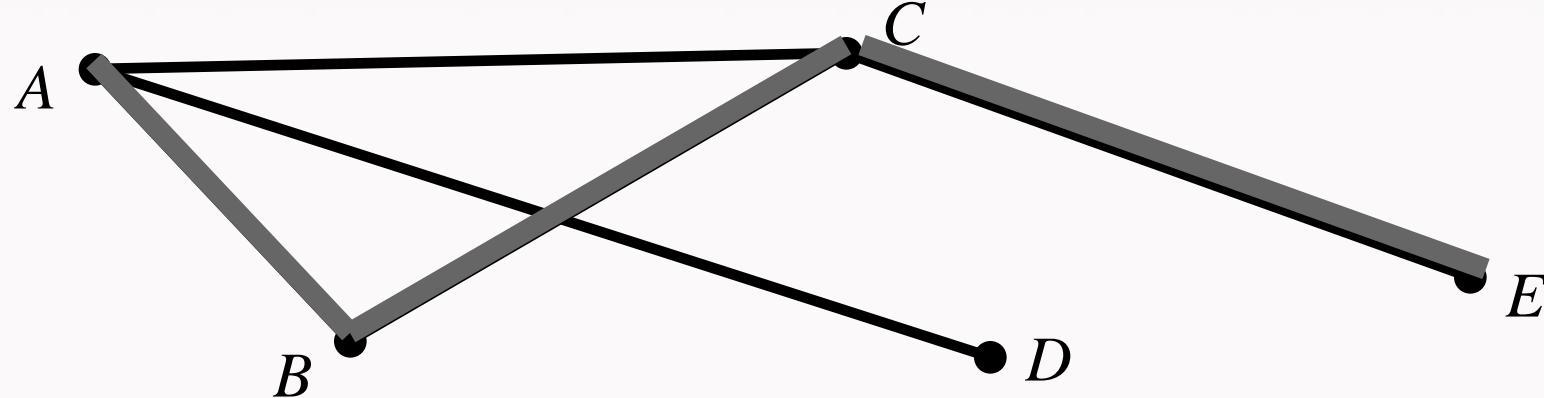


YOUR company name

$$\text{Gra}(C) = 4, \text{Gra}(A) = 3, \text{Gra}(B) = 1, \text{Gra}(D) = 2$$

Nociones básicas de grafos

- Un **camino** es una sucesión de aristas distintas adyacentes.
 - ¡No se permite repetir aristas en un camino!

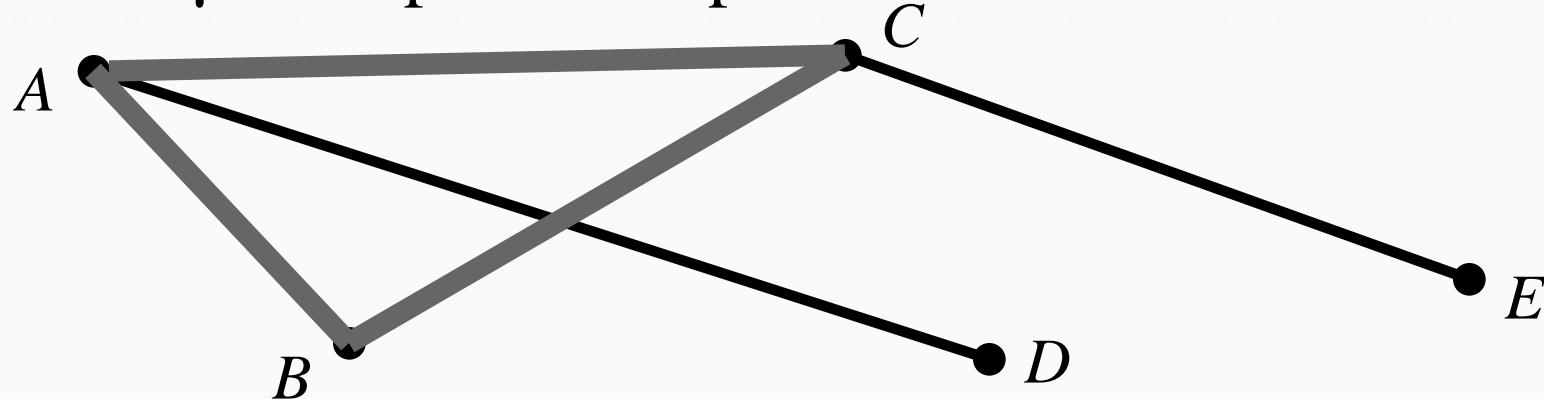


Las aristas AB, BC, y CE forman el camino A, B, C, E

Las aristas AD, DA y AC no forman un camino (repetición)

Nociones básicas de grafos

- Un **circuito** es un camino que comienza y termina en el mismo vértice.
 - ¡No se permite repetir aristas!

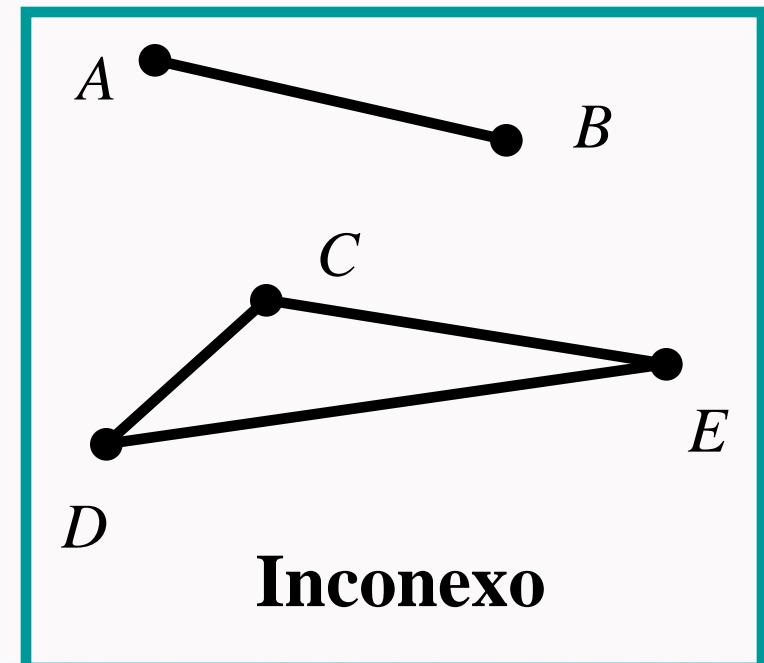
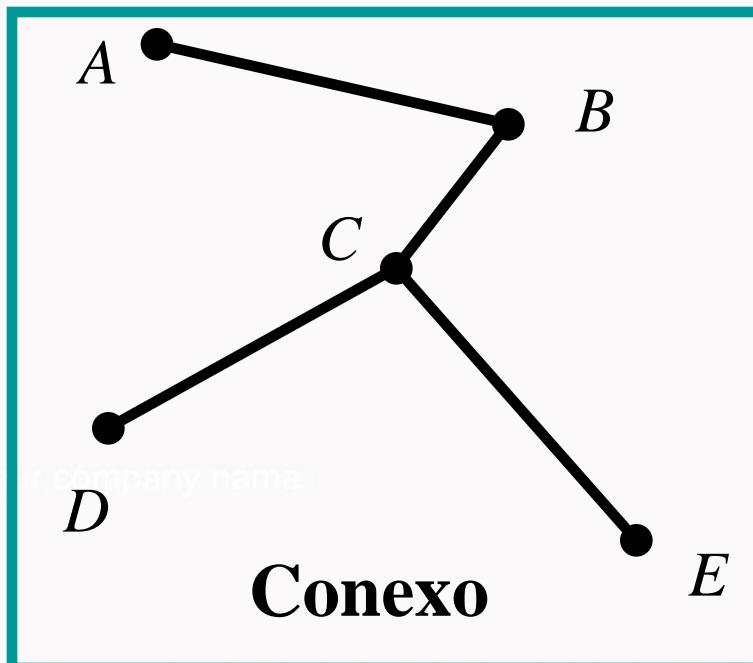


AB, BC, and CA form the circuit A, B, C, A

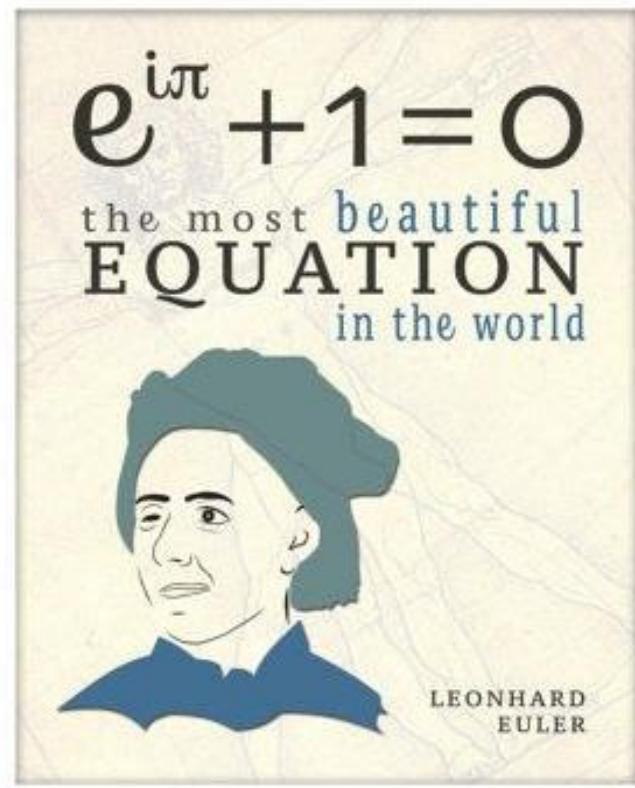
Las aristas BA and AD do not form a circuit

Nociones básicas de grafos

- Un grafo se dice **conexo** si cualesquiera dos vértices pueden unirse por un camino. Si un grafo no es conexo, se llama **inconexo**



Nociones básicas de grafos

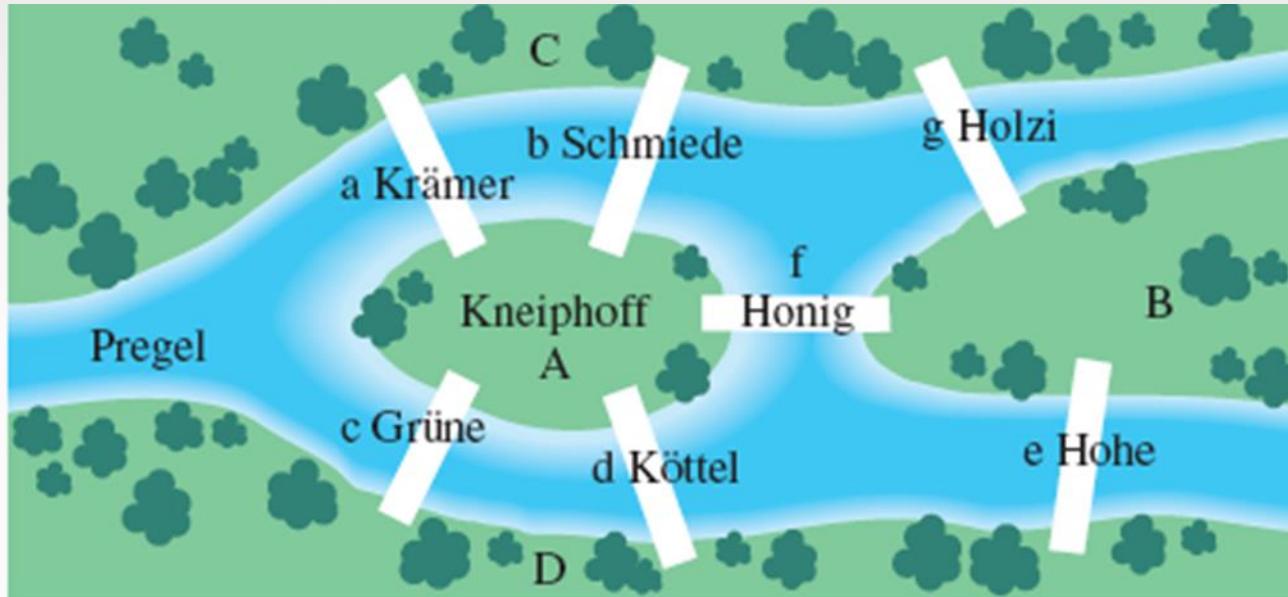


Leonhard Euler (1707-1783) ha sido el matemático mas prolífico de todos los tiempos, con contribuciones importantes en Geometría, Cálculo, Física, ... y Grafos.

Aproximadamente la mitad de sus publicaciones las escribió después de quedar ciego. Cuando perdió la vista comentó: “Así ahora me distraere menos.”

Tuvo por lo menos 13 hijos

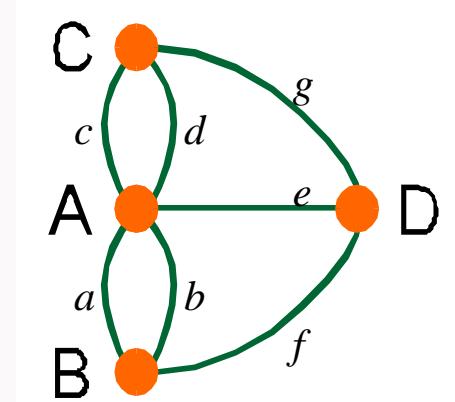
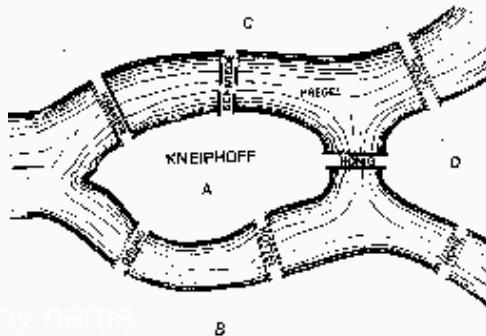
Nociones básicas de grafos



En la ciudad de **Königsberg** en Austria, hay una isla llamada "**Kneiphoff**" bordeada por el río **Pregel**. Hay **siete puentes** conectando las orillas. El problema es saber si una persona puede recorrer todos estos puentes, pasando por todos y cada uno de ellos solamente una vez, y volviendo a su punto de partida.

Nociones básicas de grafos

- Cuando Euler llegó a Königsberg, había consenso en la imposibilidad de hacer aquel recorrido, pero nadie lo aseguraba con certeza
- Euler planteó el problema como uno de grafos:
 - Cada parte de tierra supondría un vértice, y
 - Cada puente representaría una arista



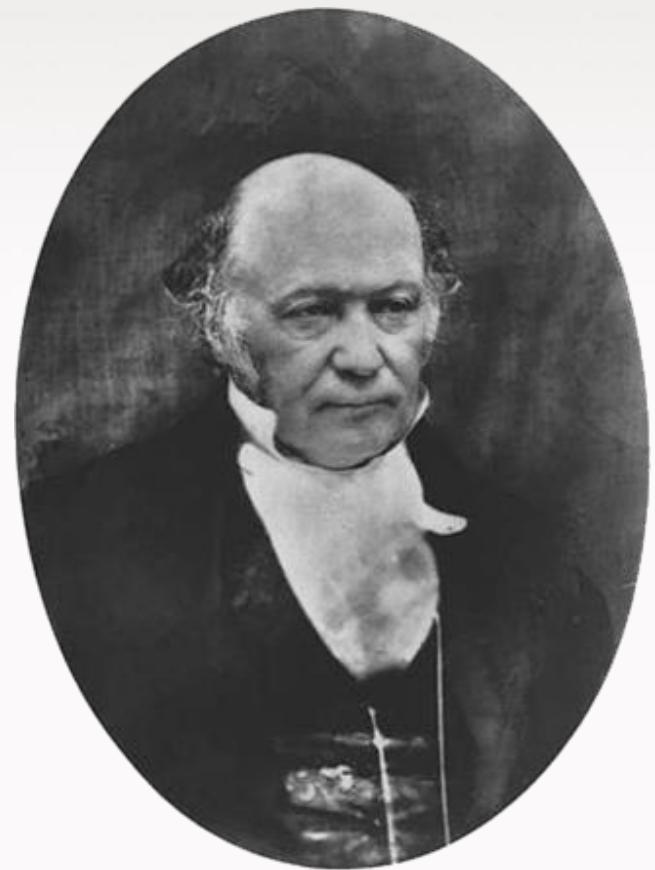
Y en 1736 demostró la imposibilidad de dar un paseo como el que se quería dar.

Nociones básicas de grafos

- Un **Camino Euleriano** es un camino que pasa a través de cada arista del grafo una y solo una vez
- Un **Círculo Euleriano** es un circuito que pasa por cada arista del grafo una y solo una vez
- **Teorema de Euler**
 - Si todos los vértices de un grafo son de grado impar, entonces no existen circuitos eulerianos.
 - Si un grafo es conexo y todos sus vértices son de grado par, existe al menos un circuito euleriano.

Nociones básicas de grafos

- Un **Círculo Hamiltoniano** es un circuito que pasa a través de cada vértice una y solo una vez, y termina en el mismo vértice en el que comenzó.
- Los **Circuitos Hamiltonianos y los Circuitos Eulerianos** son conceptos distintos y separados: En un grafo podemos tener de unos, y no de otros.
- A diferencia de los Circuitos Eulerianos, no tenemos un resultado simple que nos diga si un grafo tiene o no Circuitos Hamiltonianos.



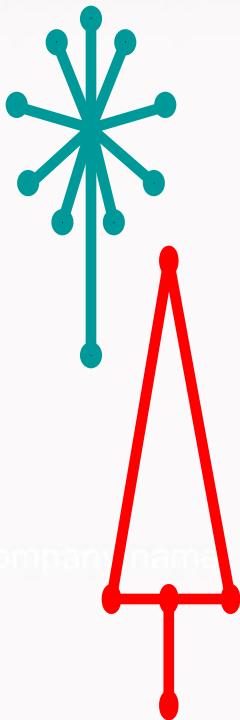
William R.
Hamilton

Ejemplos

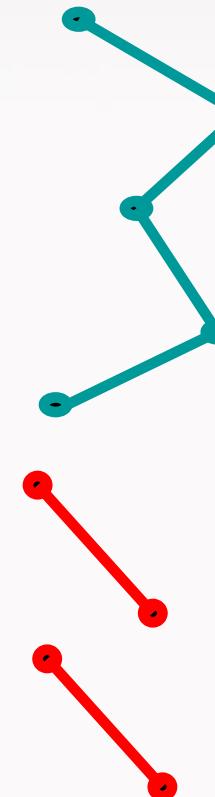
- La determinación de un circuito euleriano minimal es lo que se conoce con el nombre del Problema del Cartero Chino:
 - Encontrar el circuito de longitud minimal que recorre cada arista de un grafo al menos una vez.
- A la búsqueda de un circuito hamiltoniano minimal se la conoce con el nombre de Problema del Viajante de Comercio:
 - Hallar el circuito de longitud mínima que recorre todos los nodos de un grafo una y solo una vez, comenzando y terminando por el mismo vértice

Nociones básicas de grafos

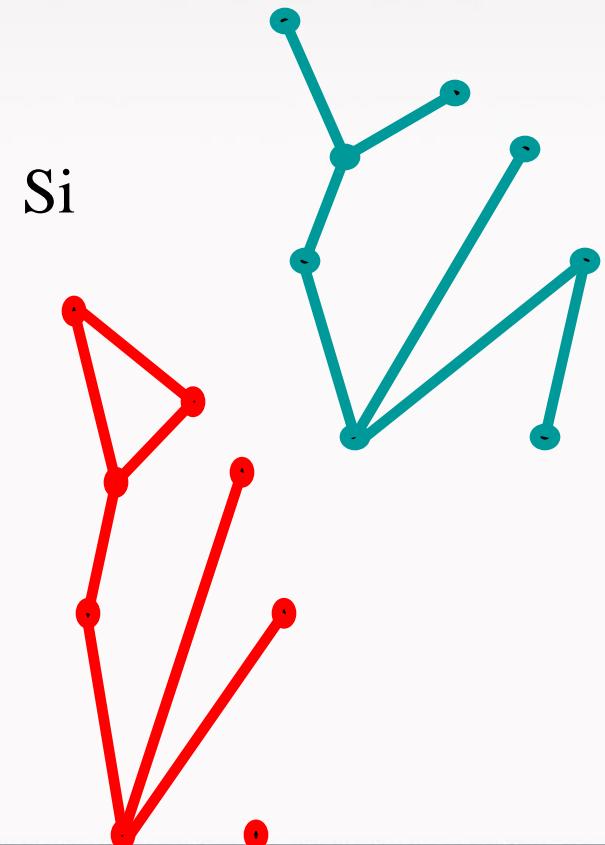
- Un **árbol** es un grafo que no tiene ciclos.
- Un grafo conexo y sin circuitos, se llama un **árbol**



Si



No

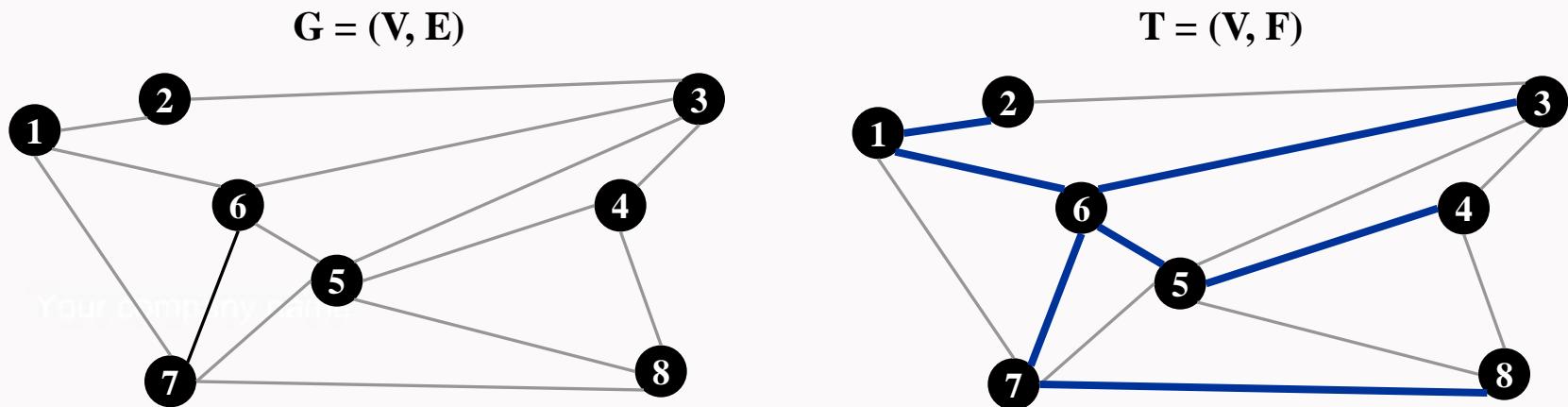


Si

No

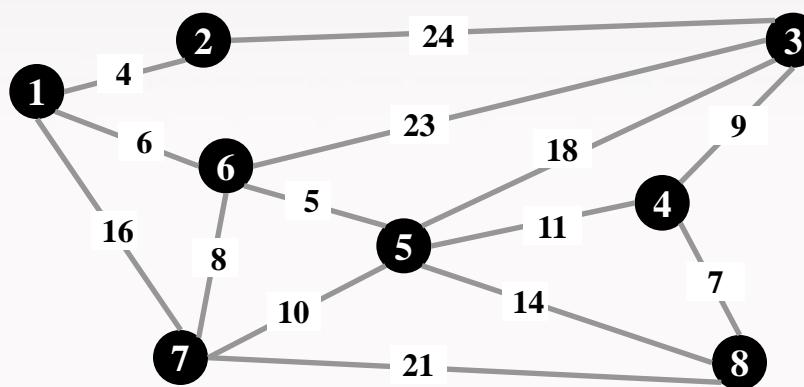
Árbol Generador de un Grafo

- Sea $T = (V, F)$ un subgrafo de $G = (V, E)$.
 - T es un arbol generador de G :
 - T es acíclico y conexo.
 - T es conexo y tiene $|V| - 1$ arcos.
 - T es acíclico y tiene $|V| - 1$ arcos.

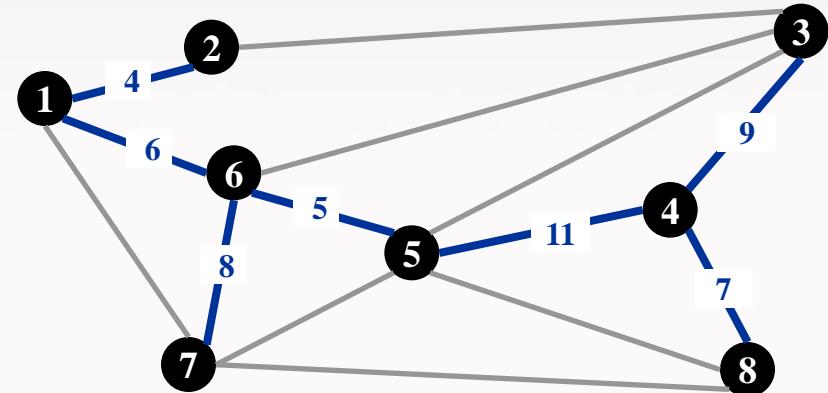


Árbol Generador Minimal

- Dado un grafo conexo G con pesos en sus arcos c_e , un **Árbol Generador Minimal** es un árbol generador de G en el que la suma de los pesos de sus arcos es mínima.



$$G = (V, E)$$



$$T = (V, F)$$

$$l(T) = 50$$

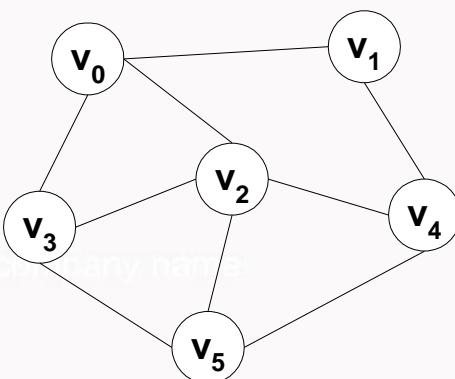
- Teorema de Cayley (1889).** Hay n^{n-2} arboles generadores de K_n (el grafo completo de n vértices)
 - Por tanto el empleo de la fuerza bruta para encontrar el AGM de un grafo no es un método recomendable

La matriz de adyacencia

- Si suponemos un grafo $G = (X, E)$ con n vértices, entonces su matriz de adyacencia es:

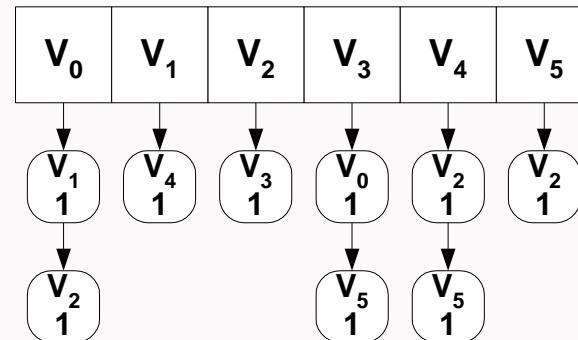
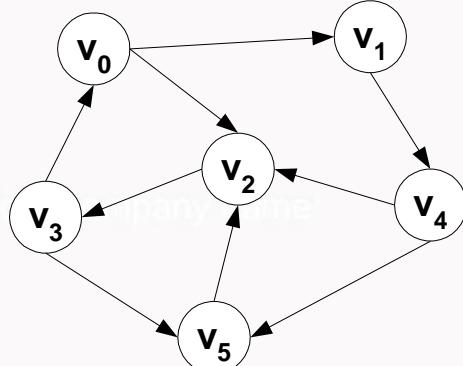
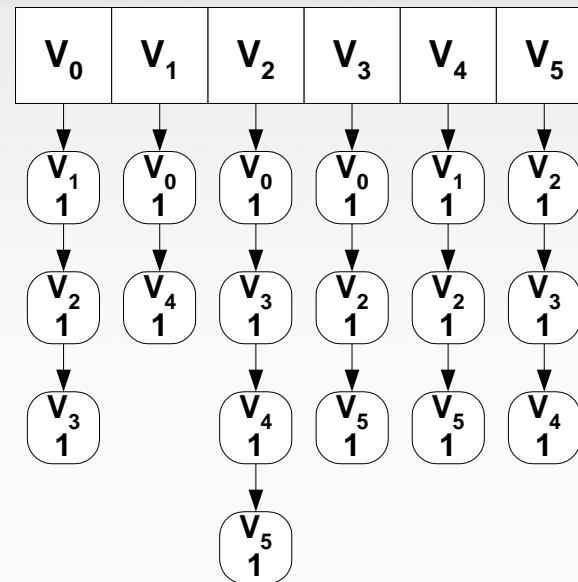
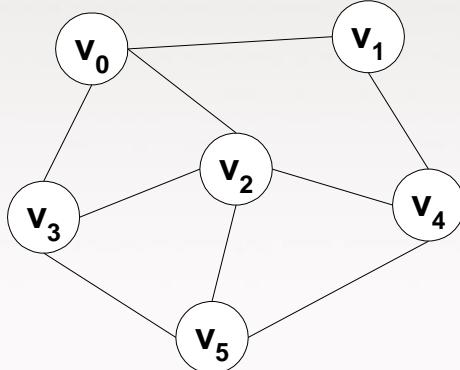
$$A_G(i, j) = \begin{cases} 1 & \dots \text{si } (x_i, x_j) \in E \\ 0 & \dots \text{si } (x_i, x_j) \notin E \end{cases}$$

- Cuando el grafo es ponderado, el valor que aparece en cada casilla es el peso de la arista correspondiente



left → right	0	1	2	3	4	5
0	-	1	1	1	-	-
1	1	-	-	-	1	-
2	1	-	-	1	1	1
3	1	-	1	-	-	1
4	-	1	1	-	-	1
5	-	-	1	1	1	-

Representación por listas de adyacencia



Matriz de Incidencia

- Si se trata de un grafo no dirigido, entonces la matriz es:

$$B_G(i, j) = \begin{cases} 1 & \dots si \dots x_i \dots esta \dots en \dots a_j \\ 0 & \dots en \dots otro \dots caso \end{cases}$$

donde a_j es la arista que conecta x_i con x_j .

- Si se trata de un grafo dirigido, entonces la matriz es:

$$B(i, j) = \begin{cases} +1 & \dots si \dots x_i \dots es \dots inicial \dots en \dots a_j \\ 0 & \dots en \dots otro \dots caso \dots (bucle) \\ -1 & \dots si \dots x_i \dots es \dots final \dots en \dots a_j \end{cases}$$

donde a_j es el arco que conecta x_i con x_j .

Algorítmica

Capítulo 3. Algoritmos Greedy

Tema 8. Greedy sobre grafos

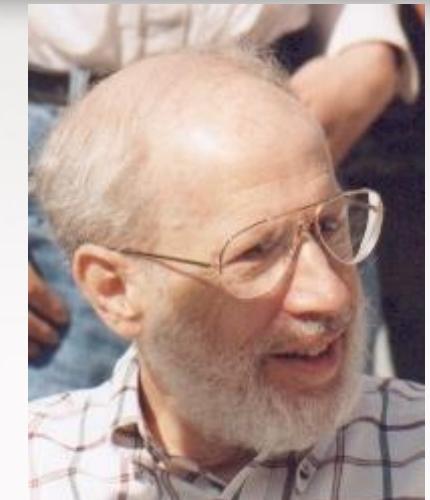
- **Algoritmos para el árbol generador minimal: Algoritmos de Kruskal y Prim**
- **Algoritmos para caminos mínimos: Algoritmo de Dijkstra**

El Problema del Árbol Generador Minimal

- Se trata de encontrar el árbol generador minimal de un grafo, es decir, el árbol generador de mínima longitud.
- Suponemos un grafo conexo $G = (V, A)$, sobre el que hay definida una matriz de pesos $L(i,j) \geq 0$.
- Queremos encontrar un árbol $T \subseteq A$ tal que todos los nodos permanezcan conectados cuando solo se usen aristas de T , siendo la suma de los pesos de sus aristas mínima.
- Al grafo (V, T) se le llama Árbol Generador Minimal del grafo G .

El Problema del Árbol Generador Minimal

Joseph B. Kruskal investigador del Math Center de los Laboratorios Bell, en 1956 descubrió su algoritmo para la resolución del problema del Árbol Generador Minimal. Este problema es un problema típico de optimización combinatoria, que fue considerado originalmente por Otakar Boruvka (1926) mientras estudiaba la necesidad de electrificación rural en el sur de Moravia en Checoslovaquia.



- Las aplicaciones de este problema lo hacen muy importante.
 - Diseño de redes físicas.
 - teléfonos, eléctricas, hidráulicas, TV por cable, computadores, carreteras, ...
 - Análisis de Clusters.
 - Eliminación de aristas largas entre vértices irrelevantes
 - Búsqueda de cúmulos de quasares y estrellas
 - Solución aproximada de problemas NP.
 - PVC, árboles de Steiner, ...
 - Distribución de mensajes entre agentes
 - Aplicaciones indirectas.
 - Plegamiento de proteínas, Reconocimiento de células cancerosas, ...

El Problema del Arbol Generador Minimal

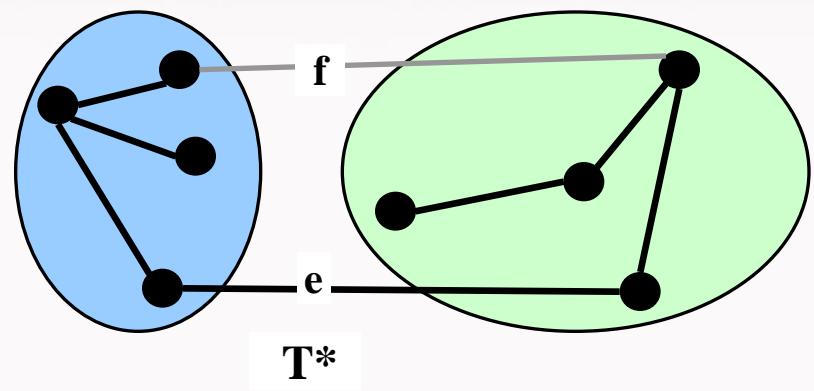
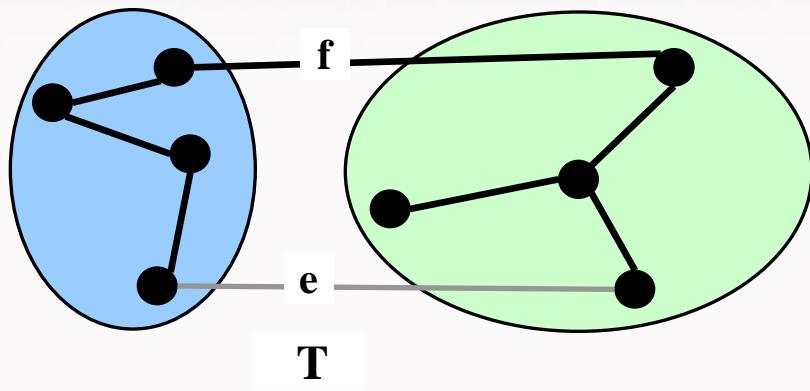
- El problema es resoluble con el enfoque greedy:
- Tenemos una lista de aristas: A partir de ella pueden darse las listas de candidatos o no a solución.
- Una solución será un conjunto de aristas que forme un AGM.
- La condición de factibilidad es que la arista que se vaya a incluir no forme un ciclo con las ya incluidas.
- El criterio de selección será escoger en cada momento la arista de mínimo peso.
- El objetivo es que la suma de los pesos de las aristas en el árbol sea mínima.

El Problema del Arbol Generador Minimal

- Se verifica la **Propiedad del AGM**:
- Sea $G = (V, A)$ un grafo no dirigido y conexo donde cada arista tiene una longitud conocida. Sea $U \subseteq V$ un subconjunto propio (lo que significa que U no puede coincidir con V) de los nodos de G . Si (u, v) es una arista tal que $u \in U$ y $v \in V - U$ y, además, es la arista del grafo que verifica esa condición con el menor peso, entonces existe un AGM T que incluye a (u, v) .
- Esta propiedad garantizará que el algoritmo greedy que diseñemos proporcione la solución óptima del problema, y por tanto que funcione correctamente.

Demostración de la propiedad del AGM

- **Demostración por contradicción:** Supongamos que T es un AGM que no contiene a $e = (u, v)$

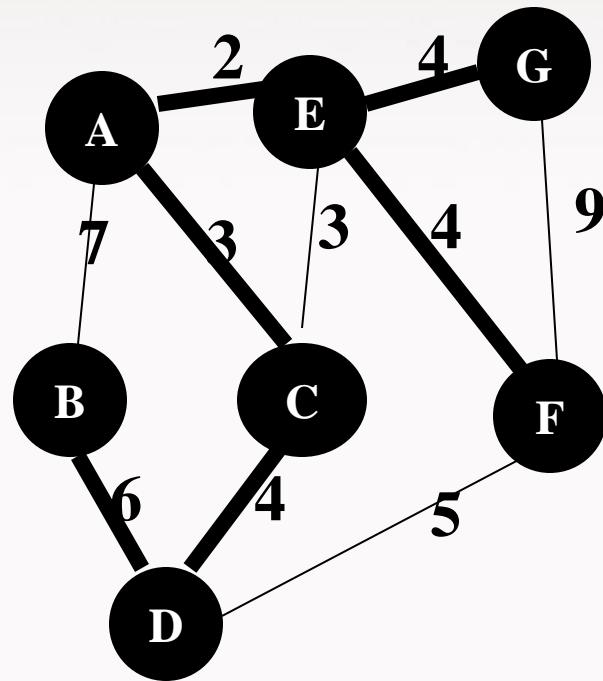


- Si añadimos e a T se crea un ciclo C . Si rompemos ese ciclo (eliminando una arista f conectando U y V), obtenemos un nuevo AGT* que (por incluir a e) tendrá menor longitud que T . Eso es una contradicción

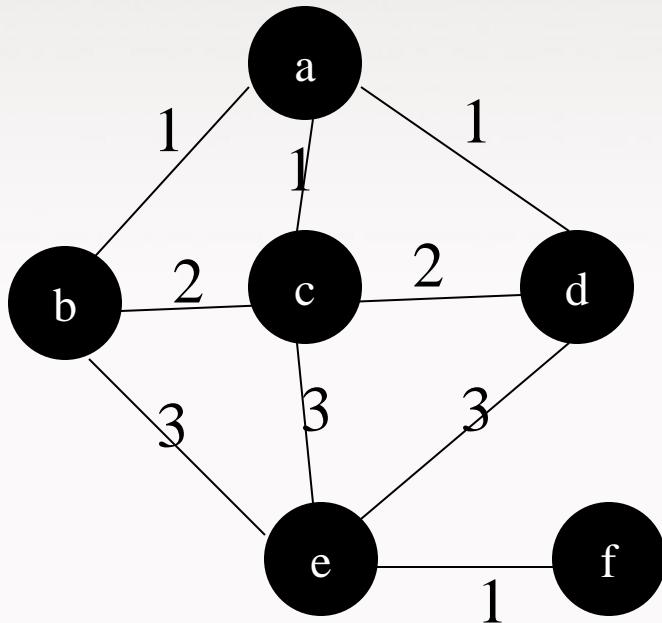
El Problema del Arbol Generador Minimal

Algoritmo de Kruskal

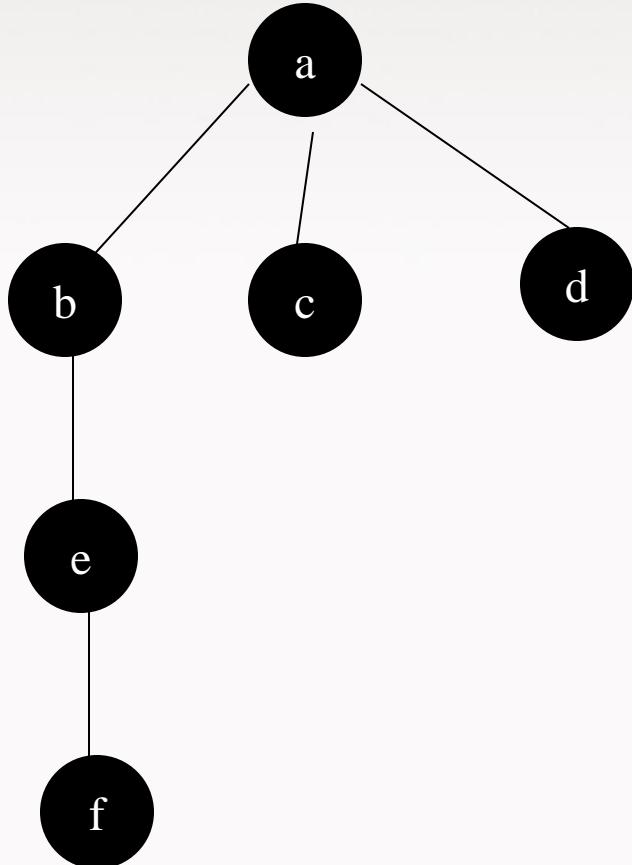
- Ordenar las aristas de forma creciente de costo
- Repetir
 - Coger la arista mas corta.
 - Borrar la arista de E
 - Aceptar la arista si no forma un ciclo en el arbol parcial,
Rechazarla en caso contrario,
- Hasta que tengamos $|V| - 1$ aristas correctas.
- Si nuestro grafo tiene n vértices y a aristas, el tiempo de este algoritmo es $O(a \log a)$



Ejemplo



(a,b), (a,c), (a,d), (e,f), (b,c), (c,d), (b,e), (c,e), (d,e)



Implementación del algoritmo

FUNCION KRUSKAL (G: GRAFO): Conjunto de aristas.

Ordenar las aristas por longitudes crecientes

$$N = |V|$$

$$T = \emptyset$$

Iniciar N conjuntos (1 elemento cada)

Repetir

$\{U,V\}$ = arista mas corta aun no considerada

 COMP U = BUSCA (U)

 COMP V = BUSCA (V)

 SI COMPU \neq COMPV ENTONCES

 UNIR (COMPU, COMPV)

$$T = T \cup (\{U,V\})$$

Hasta que $|T| = N - 1$

DEVOLVER (T)

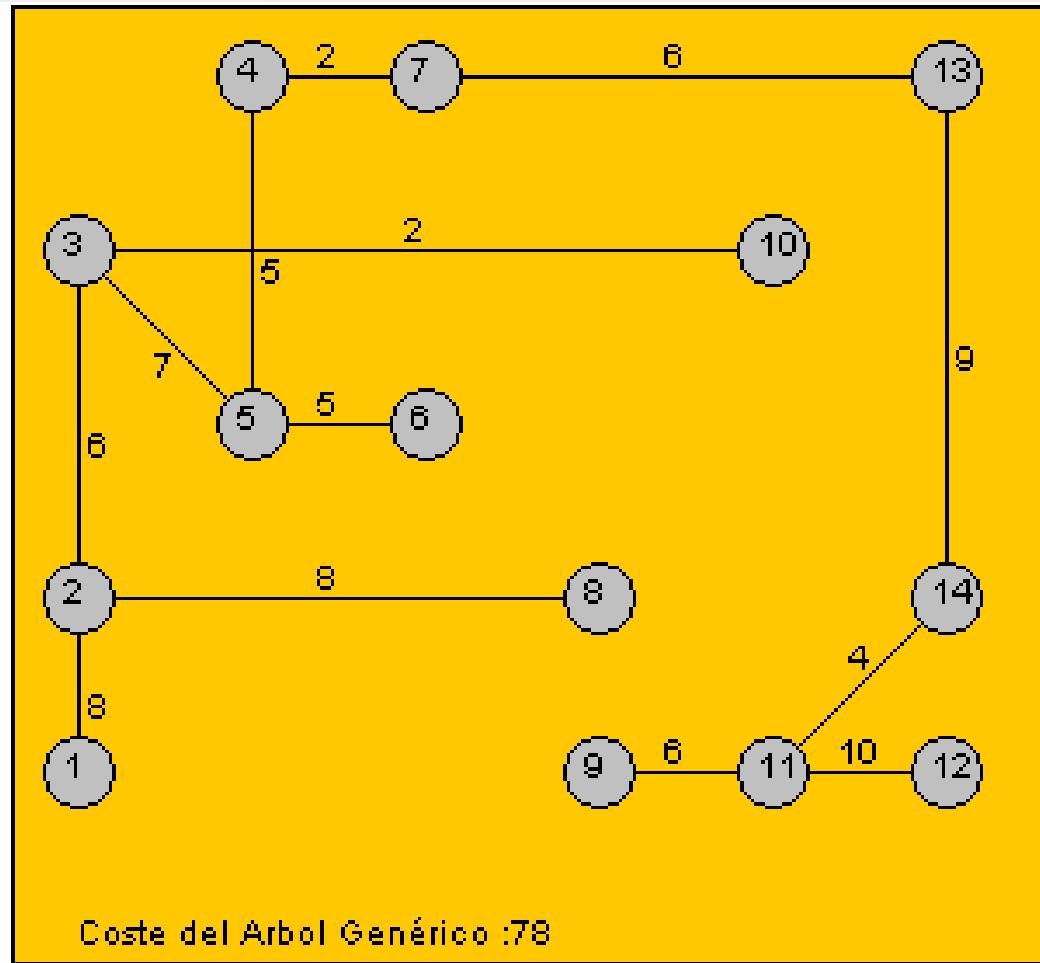
Análisis del algoritmo

- Donde:
 - Ordenar las aristas es $O(a \log a)$ para “a” aristas.
 - N es el número de vértices en el grafo.
 - T es el conjunto donde construiremos AGM.
 - BUSCA () lleva a cabo la búsqueda de la arista que se considere y es (si se hace con búsqueda binaria) $O(\log n)$.
 - UNION es $O(\log n)$ y lleva a cabo la unión de dos conjuntos.
- El orden del algoritmo de Kruskal es $O(a \log a)$, pero como tenemos n vértices, y en un grafo conexo siempre se verifica:

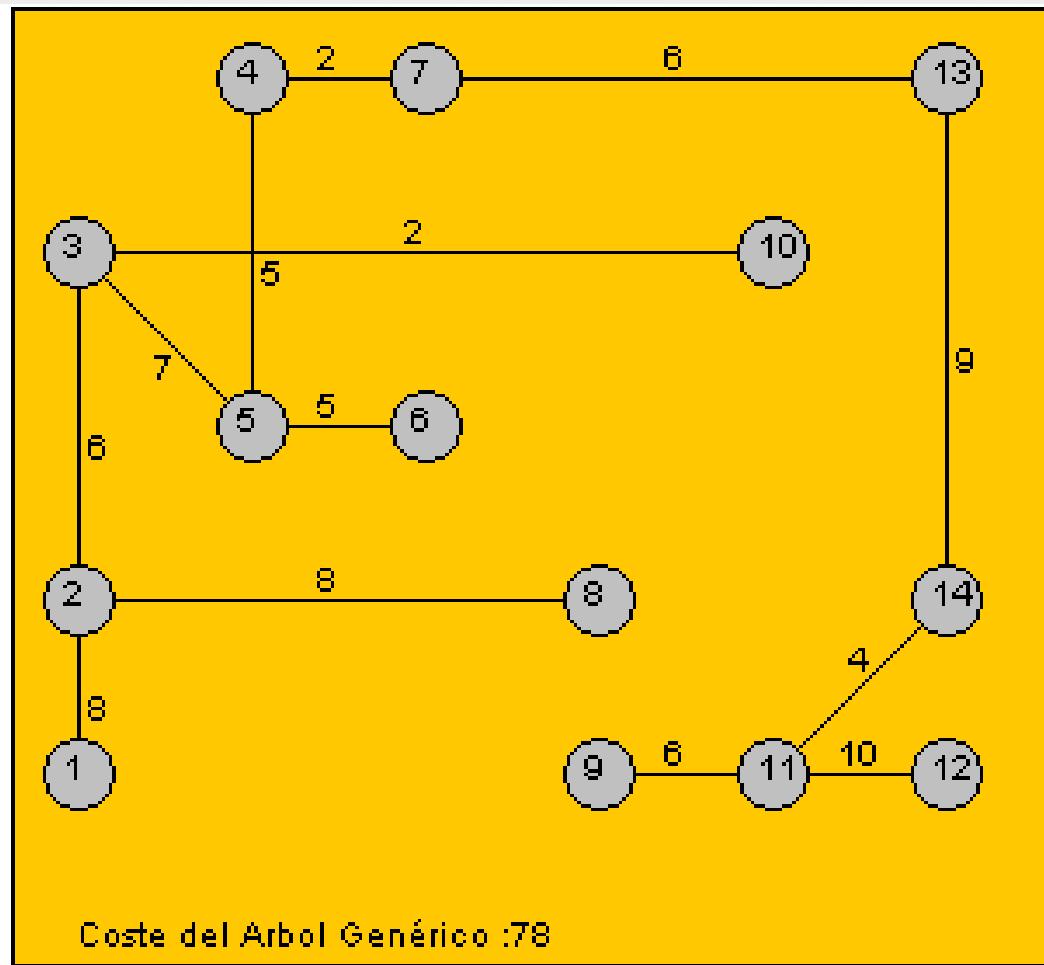
$$n-1 \leq a < \frac{n(n-1)}{2}$$

se tiene que el algoritmo es $O(a \log n)$

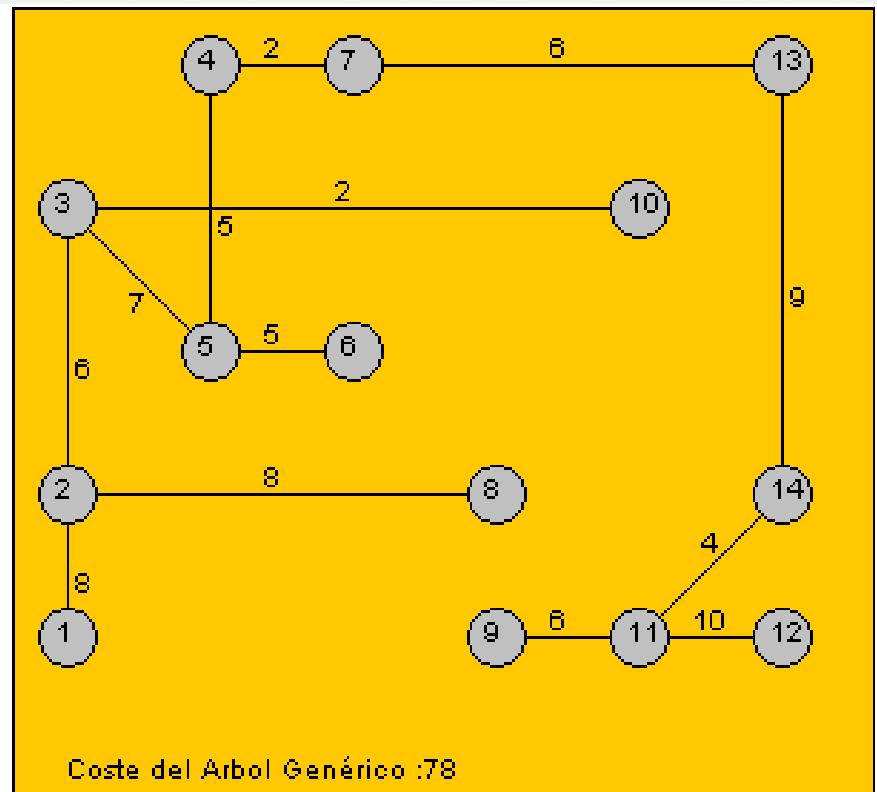
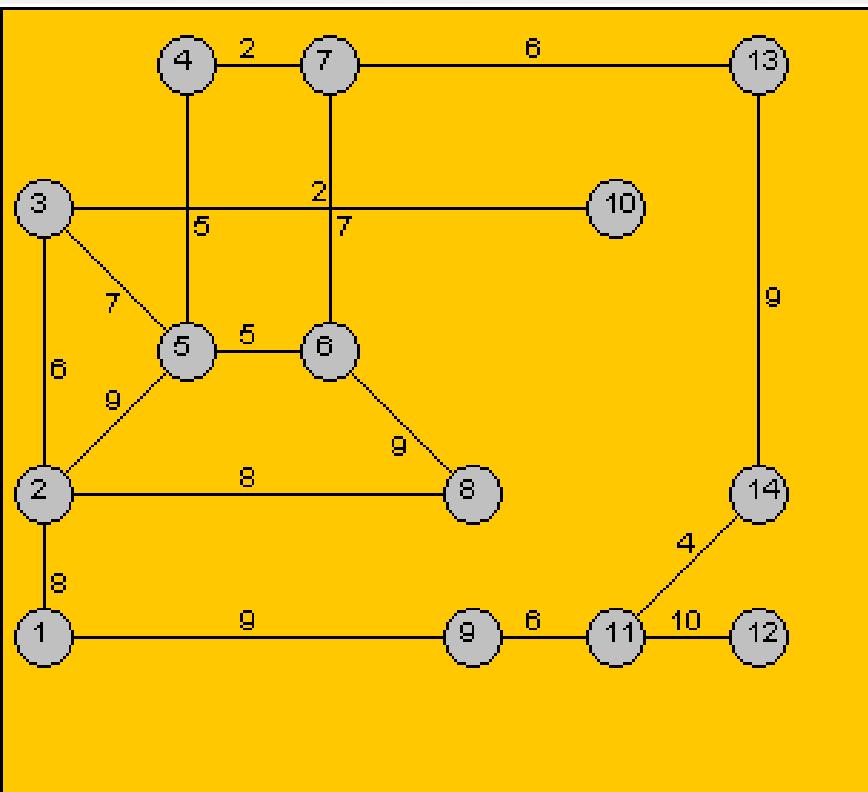
Ejemplo-1



Ejemplo-1

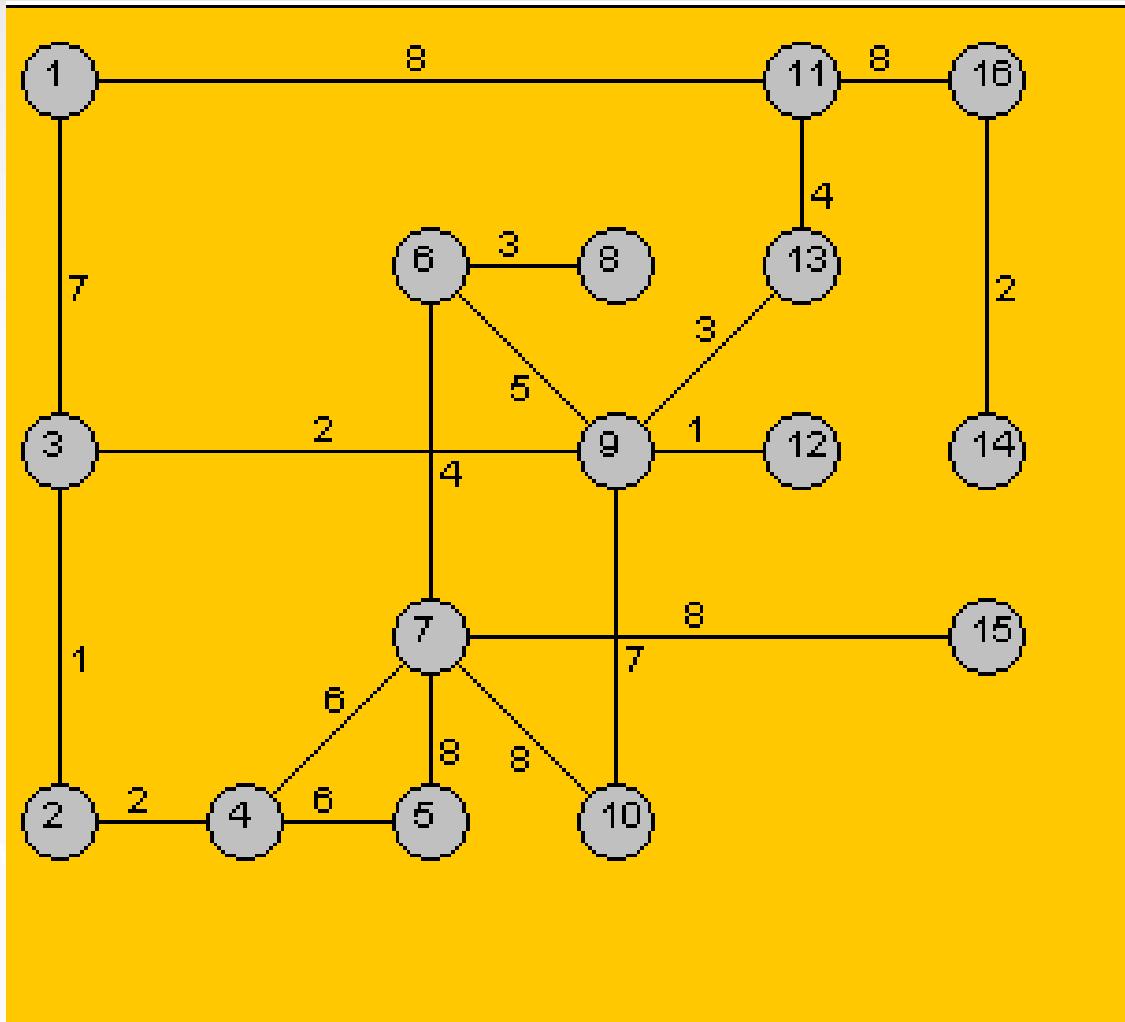


Ejemplo-1

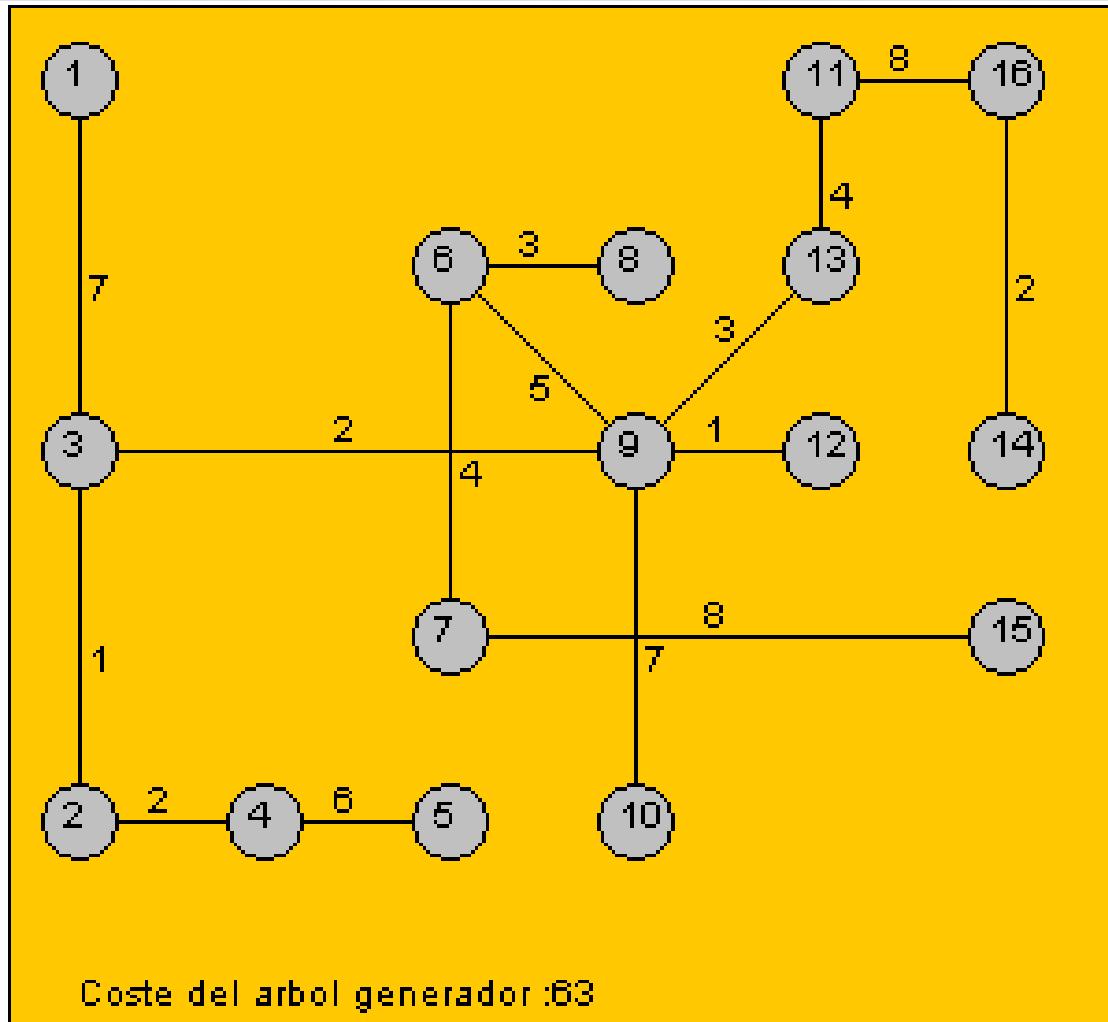


Coste del Arbol Genérico :78

Ejemplo-2

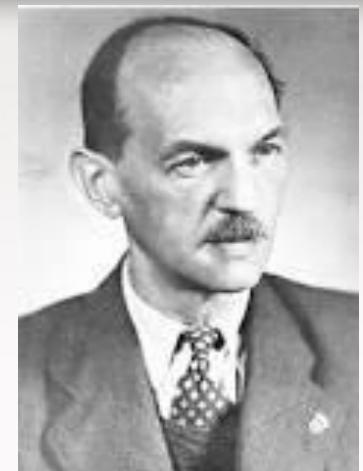


Ejemplo-2



Algoritmo de Prim

- Es otro método de resolver el problema del AGM pero también se basa en la construcción del algoritmo en función de la propiedad del AGM.
- En el algoritmo de Kruskal partíamos de la selección de la arista más corta que hubiera en la lista de aristas, lo que implica un crecimiento desordenado del AGM.
- Para evitarlo, el algoritmo de Prim propone que el crecimiento del AGM sea ordenado (a partir de una raíz).
- El algoritmo fue diseñado en 1930 por el matemático checo Vojtech Jarnik y luego de forma independiente por Robert C. Prim en 1957



Vojtech Jarnik



Robert C. Prim

Algoritmo de Prim

- La idea del algoritmo de Prim es la siguiente:
 - Se toma un conjunto U de nodos, que inicialmente contiene al nodo raíz.
 - Formamos el conjunto T de soluciones (aristas).
 - En cada etapa el algoritmo busca la arista más corta que conecta U con $V - U$, siendo V el conjunto de candidatos.
 - Añade el vértice obtenido al conjunto U y la arista obtenida a T .
 - En cada instante, las aristas que están en T constituyen un AGM para los nodos que están en U .
 - Esto lo hacemos hasta que $U = n$.

Implementación del Algoritmo de Prim

FUNCION PRIM ($G = (V, A)$) conjunto de aristas.

(Inicialización)

$T = \emptyset$ (Contendrá las aristas del AGM que buscamos).

$U =$ un miembro arbitrario de V

MIENTRAS $|U| = N$ HACER

BUSCAR $e = (u, v)$ de longitud mínima tal que

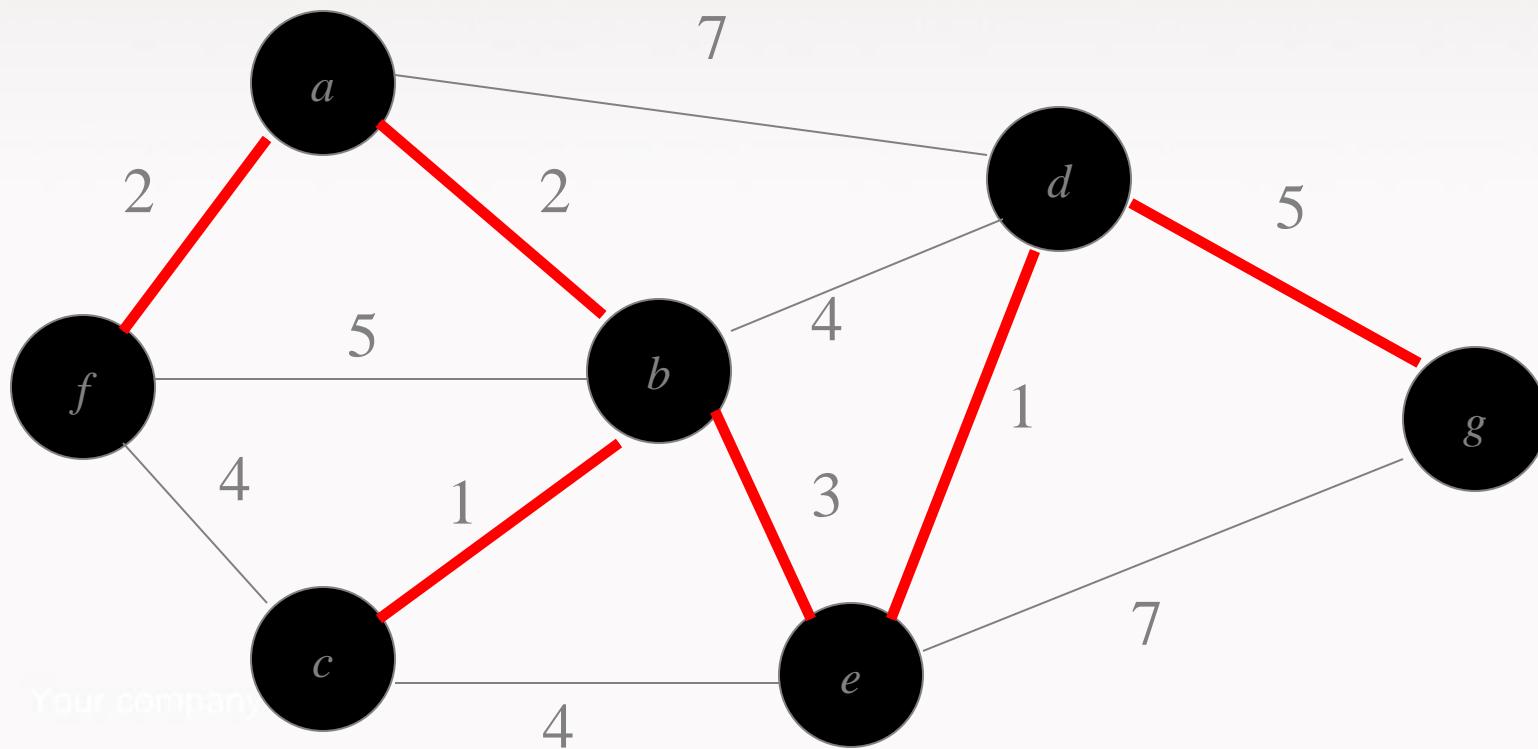
$u \in U$ y $v \in V - U$

$T = T + e$

$U = U + v$

DEVOLVER (T)

Ejemplo del Algoritmo de Prim



Implementación del Algoritmo de Prim

- Para estudiar la eficiencia de Prim, es necesario elaborar un poco más su implementación, por lo que suponemos:
- $L [I , J] \geq 0$ una matriz de distancias.
- $MasProximo [x]$ es un vector que nos da el nodo U que está más cercano al vértice x .
- $DistMin [x]$ es un vector que nos da la distancia entre x y $MasProximo [x]$.

Implementación del Algoritmo de Prim

Funcion Prim ($L[1...n, 1...n]$): conjunto de aristas
{al comienzo solo el nodo 1 se encuentra en U }

$T = \emptyset$ (contendrá las aristas del AGM)

Para $i = 2$ hasta n hacer

 MasProximo [i] = 1; DistMin [i] = $L[i, 1]$

Repetir $n - 1$ veces

$\min = \infty$

 Para $j = 2$ hasta n hacer

 Si $0 \leq \text{DistMin}[j] < \min$ entonces $\min = \text{DistMin}[j]$

$T = T + (\text{MasProximo}[k], k)$

$\text{DistMin}[k] = -1$ (estamos añadiendo k a U)

 para $j = 2$ hasta n hacer

 si $L[j, k] < \text{DistMin}[j]$ entonces

$\text{DistMin}[k] = L[j, k];$

$\text{MasProximo}[j] = k$

Devolver T .

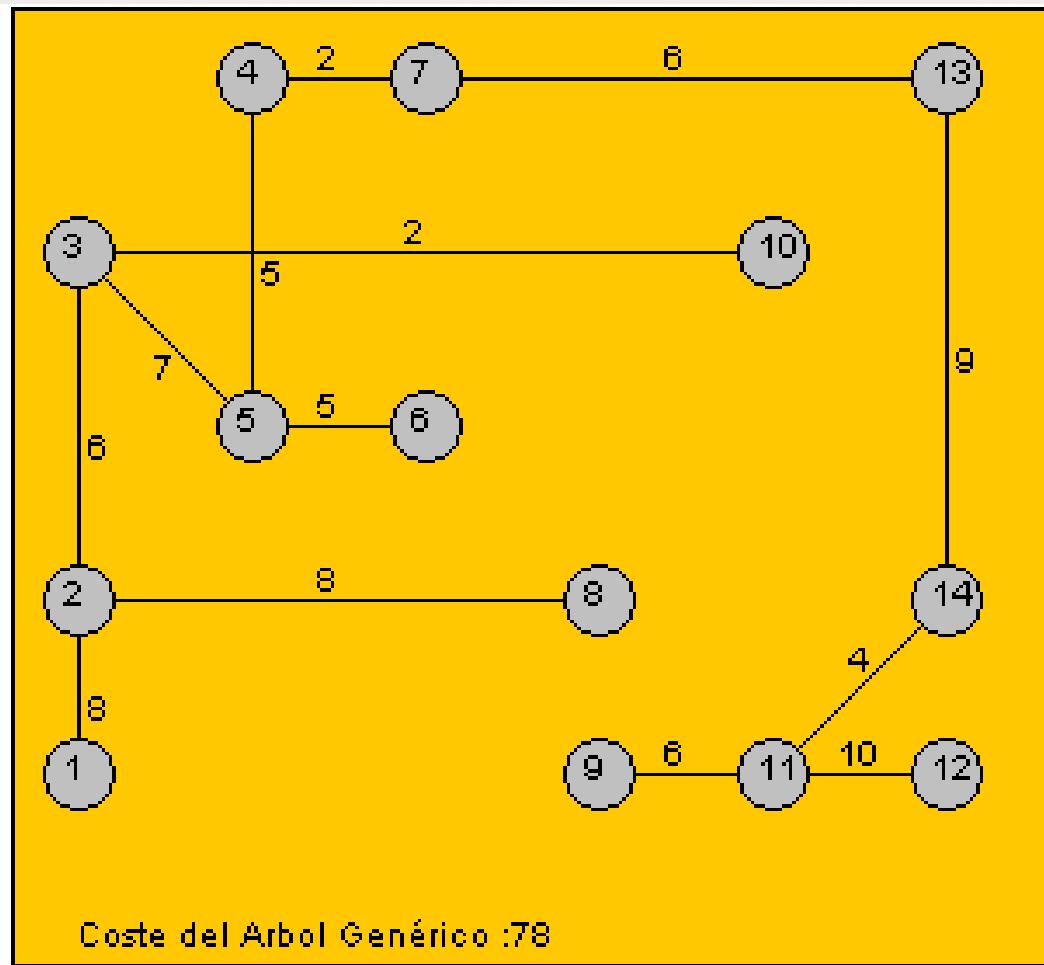
Análisis del algoritmo de Prim

- El bucle principal del algoritmo se ejecuta $n - 1$ veces.
- En cada iteración, el bucle “para” anidado requiere un tiempo $O(n)$. Por tanto, el algoritmo de Prim requiere un tiempo $O(n^2)$.
- Como el Algoritmo de Kruskal era $O(a \log n)$, siendo a el numero de aristas del grafo,
- ¿Que algoritmo usar Prim o Kruskal?
- Sabemos que

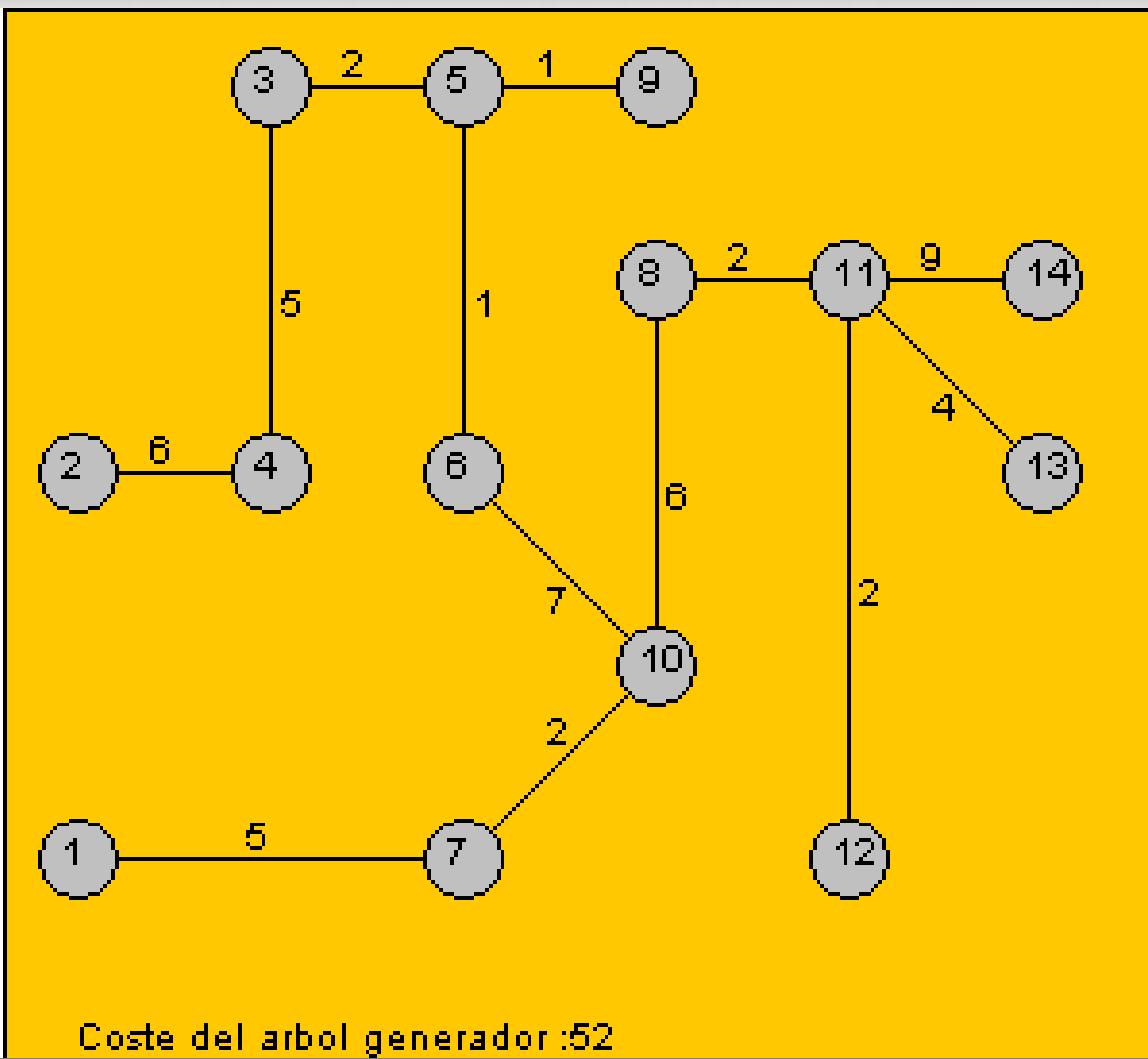
$$n - 1 \leq a < \frac{n(n - 1)}{2}$$

- Luego en grafos poco densos, lo mejor seria emplear Kruskal, y si el grafo es muy denso, entonces el de Prim

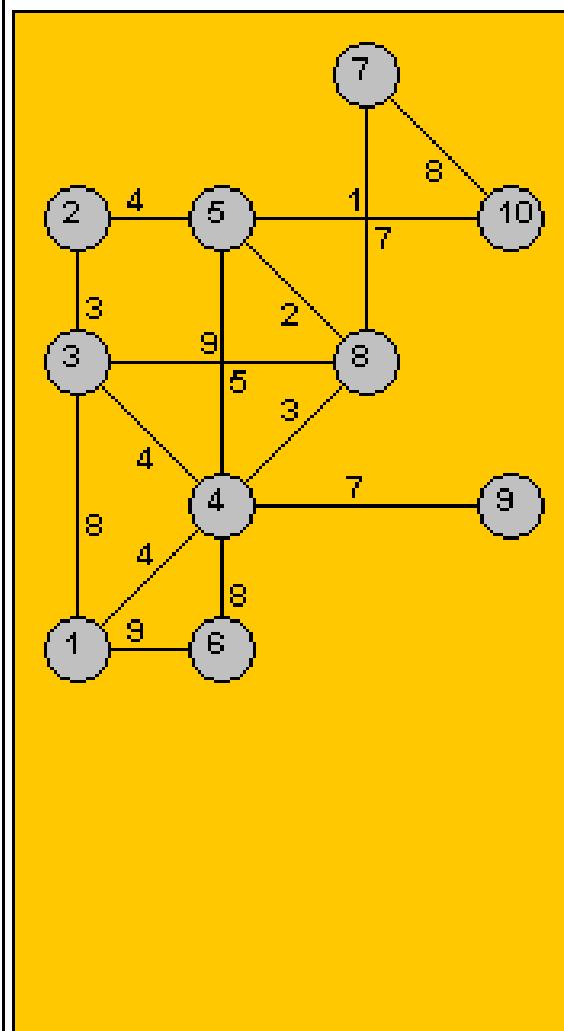
Ejemplo-1



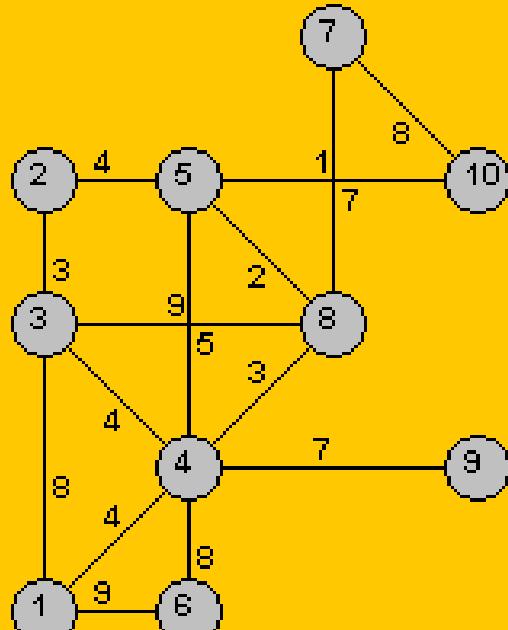
Ejemplo-1



Ejemplo-2: Prim usando matriz de adyacencia



EJEMPLO-2: PRIM USANDO MATRIZ DE ADYACENCIA

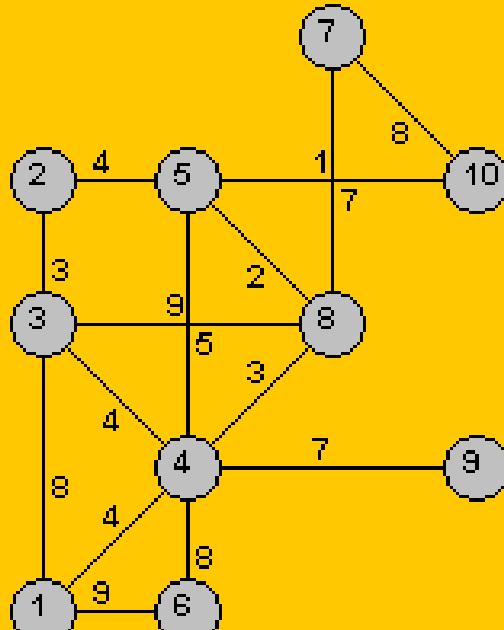


Matriz de adyacencia

	1	2	3	4	5	6	7	8	9	10
1		8	4		9					
2			3		4					
3	8	3		4				9		
4	4		4		5	8		3	7	
5		4		5				2	1	
6	9		8							
7							7		8	
8		9	3	2		7				
9			7							
10					1	8				

Blue checkmarks are placed along the diagonal of the matrix, indicating the selected edges in the Minimum Spanning Tree.

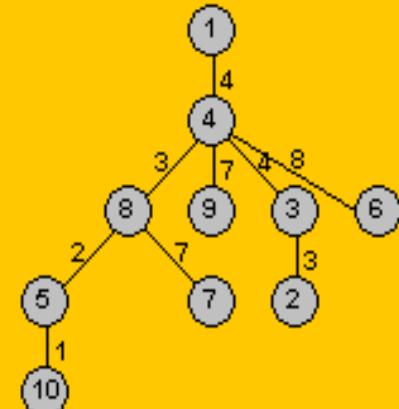
EJEMPLO-2: PRIM USANDO MATRIZ DE ADYACENCIA



Matriz de adyacencia

	1	2	3	4	5	6	7	8	9	10
1		8	4	9						
2			3	4						
3	8	3		4			9			
4	4	4		5	8		3	7		
5		4	5				2		1	
6	9		8							
7					7		8			
8		9	3	2		7				
9			7							
10				1	8					

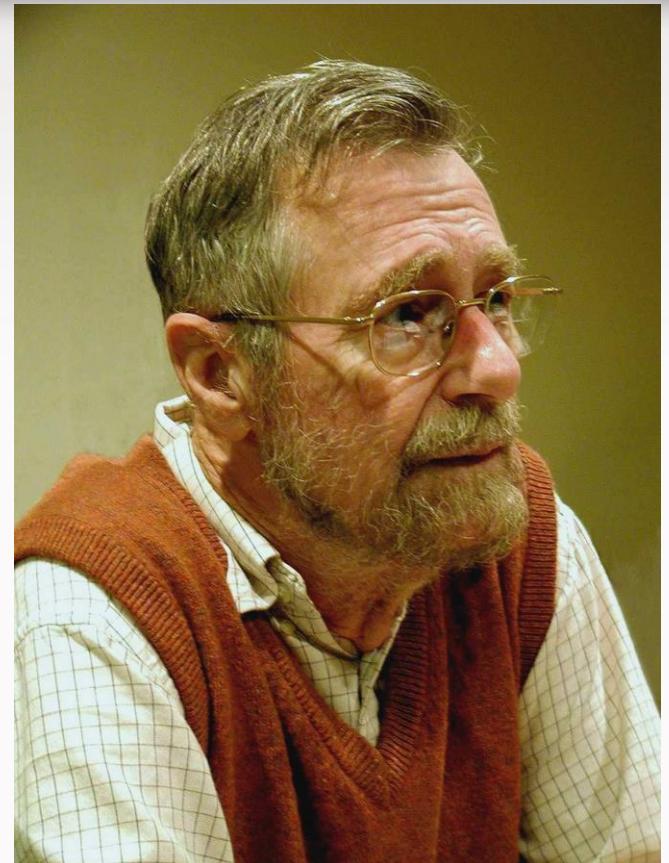
Árbol Generador



Coste del árbol genérico : 39

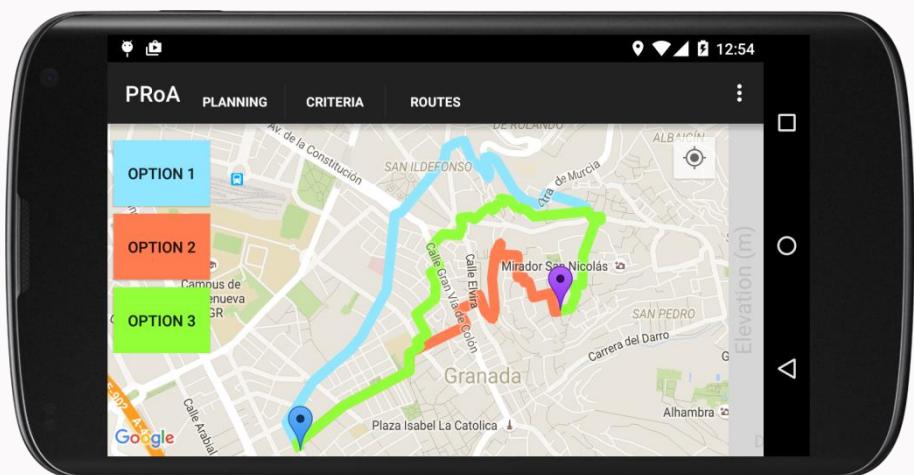
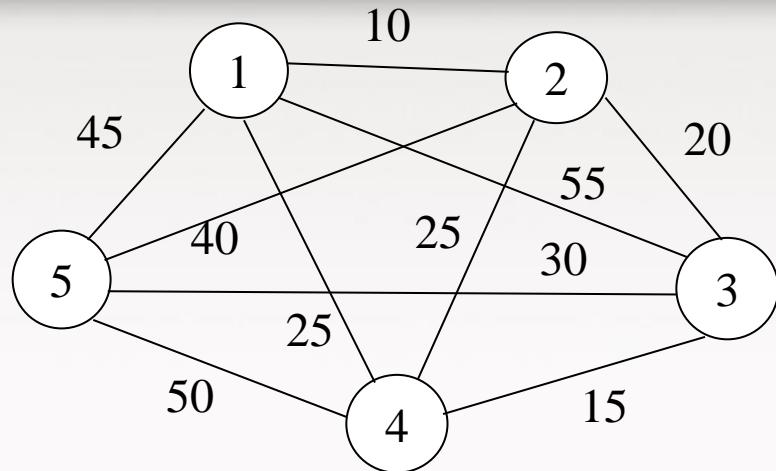
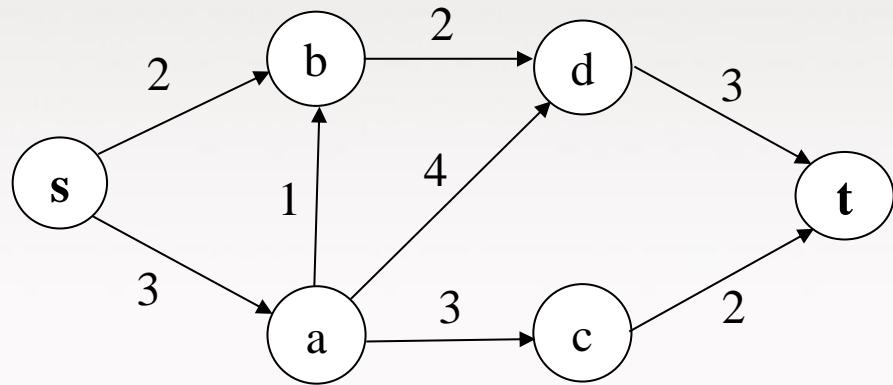
Algoritmos de Camino Mínimo

- El problema de determinar el camino mínimo (de longitud mínima) entre dos vértices de un grafo es de una importancia extraordinaria en todas las ramas de las Ingenierías, y en particular en Inteligencia Artificial
- El algoritmo que resuelve este problema de manera mas eficiente es el conocido **Algoritmo de Dijkstra**, que diseño el mismo Edsger W. Dijkstra en 1959, cuando tenía 29 años
- Murió el 6 de agosto de 2002.



“El esfuerzo de utilizar las máquinas para emular el pensamiento humano siempre me ha parecido bastante estúpido. Preferiría usarlas para emular algo mejor.” (Edsger W. Dijkstra)

Algoritmos de Camino Mínimo



El problema del Camino Mínimo

- Suponemos:
- Un grafo dirigido $G = (V,E)$ con E un conjunto de arcos y V un conjunto de vértices.
 - Una distancia definida entre los nodos que viene dada por una matriz $L[1\dots n, 1\dots n] \geq 0$.
- Se trata de hallar la distancia de los caminos mínimos desde un nodo raíz (el nodo 1) a todos los demás.
- Es un problema típicamente Greedy, en el que se identifican fácilmente las seis características.
- La aplicación del enfoque greedy conduce al **Algoritmo de Dijkstra**, del que las principales características son las siguientes

Algoritmo de Dijkstra

- Llamamos S al conjunto de los nodos elegidos.
 S contendrá los nodos cuya distancia desde el origen es mínima
- Inicialmente, S solo contiene el origen y cuando finalice el algoritmo, S contendrá todos los nodos del grafo.
- En cada etapa, elegiremos aquel nodo cuya distancia al origen sea menor, para ponerlo en S .

Algoritmo de Dijkstra

- Diremos que un camino del origen a otro nodo es **especial** si todos los nodos intermedios están en S .
- En cada etapa del algoritmo se emplea un vector D que contiene la longitud del camino especial más corto a cada nodo del grafo, de manera que en cada etapa, como añadimos un vértice nuevo a S , todos los valores de $D[V]$ se actualizan.
- Para la actualización vemos si con el nuevo vértice introducido en S podemos llegar al vértice v por un camino de longitud más corta que el que había. Si es posible, se actualiza $D[V]$.

Algoritmo de Dijkstra

FUNCION DIJKSTRA

$$C = \{2, 3, \dots, N\}$$

PARA $I = 2$ HASTA N HACER $D[I] = L[1, I]$

$$I = 2, \dots, N$$

REPETIR $N - 2$ VECES

$V =$ algún elemento de C que minimice $D[V]$

$$C = C - (V)$$

PARA CADA $w \in C$ HACER

SI $D[w] > D[V] + L[v, w]$ ENTONCES

$$D[w] = D[V] + L[v, w]$$

YOUR COMPANY NAME

DEVOLVER D

Algoritmo de Dijkstra

FUNCION DIJKSTRA

$C = \{2, 3, \dots, N\}$

PARA $I = 2$ HASTA N HACER $D[I] = L[1, I]$

$I = 2, \dots, N$

$P[I] = 1$

REPETIR $N - 2$ VECES

$w = \text{algún elemento de } C \text{ que minimice } D[V]$

$C = C - (w)$

PARA CADA $v \in C$ HACER

SI $D[v] > D[w] + L[w,v]$ ENTONCES

$D[v] = D[w] + L[w,v]$

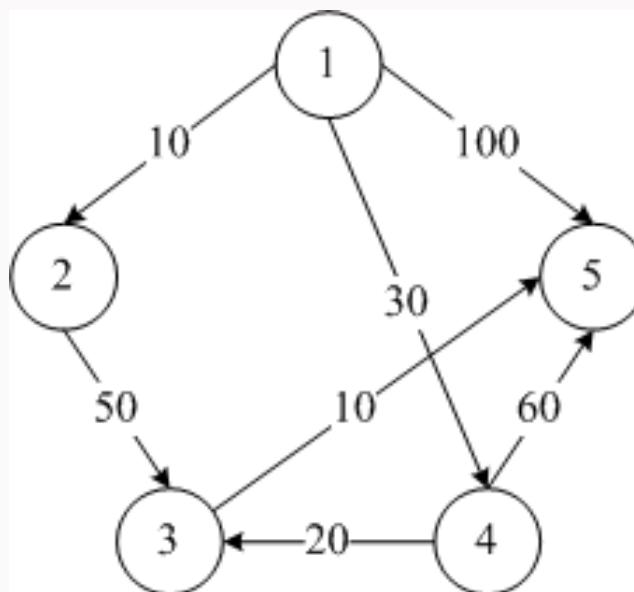
$P[v] = w$

DEVOLVER D

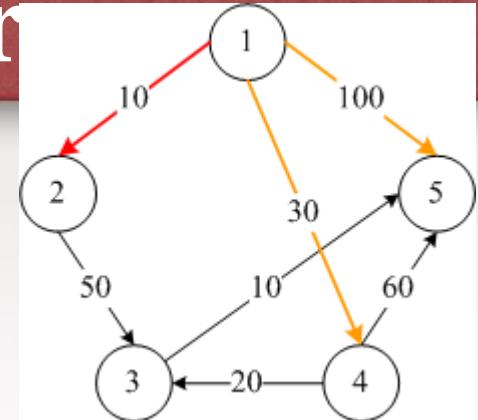
Donde P es un vector que nos permite conocer por donde pasa cada camino de longitud minima desde el origen

Algoritmo de Dijkstra

- Encontrar los caminos más cortos entre el vértice 1 y todos los demás del siguiente grafo dirigido.

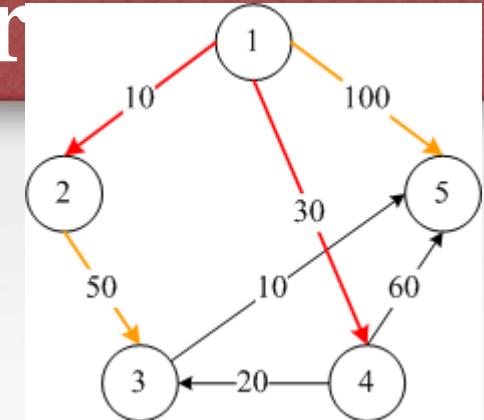


Algoritmo de Dijkstra



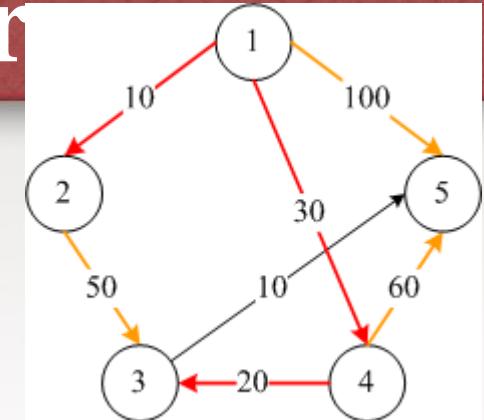
ITERACIÓN	S	w	$D[2]$	$D[3]$	$D[4]$	$D[5]$
INICIAL	{1}	—	10	∞	30	100

Algoritmo de Dijkstra



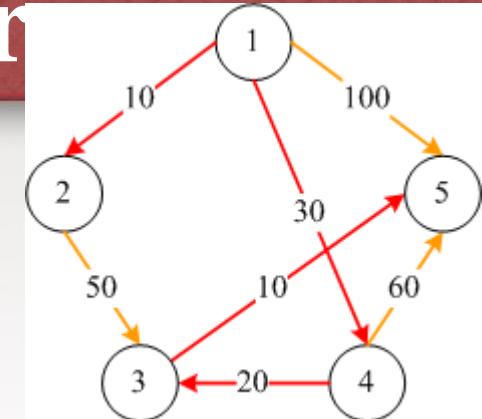
ITERACIÓN	S	w	$D[2]$	$D[3]$	$D[4]$	$D[5]$
INICIAL	{1}	—	10	∞	30	100
1	{1,2}	2	10	<u>60</u>	30	100

Algoritmo de Dijkstra



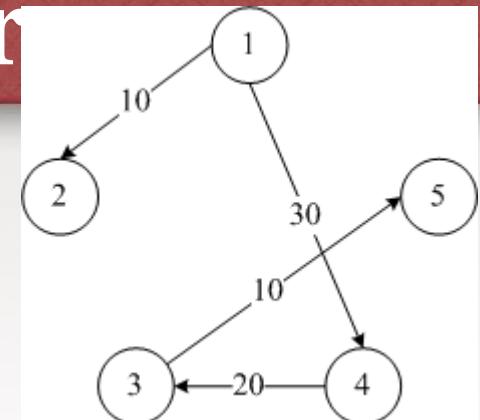
ITERACIÓN	S	w	$D[2]$	$D[3]$	$D[4]$	$D[5]$
INICIAL	{1}	—	10	∞	30	100
1	{1,2}	2	10	<u>60</u>	30	100
2	{1,2,4}	4	10	<u>50</u>	30	<u>90</u>

Algoritmo de Dijkstra



ITERACIÓN	S	w	$D[2]$	$D[3]$	$D[4]$	$D[5]$
INICIAL	{1}	—	10	∞	30	100
1	{1,2}	2	10	<u>60</u>	30	100
2	{1,2,4}	4	10	<u>50</u>	30	<u>90</u>
3	{1,2,4,3}	3	10	50	30	<u>60</u>

Algoritmo de Dijkstra



ITERACIÓN	S	w	$D[2]$	$D[3]$	$D[4]$	$D[5]$
INICIAL	{1}	--	10	∞	30	100
1	{1,2}	2	10	<u>60</u>	30	100
2	{1,2,4}	4	10	<u>50</u>	30	<u>90</u>
3	{1,2,4,3}	3	10	50	30	<u>60</u>
4	{1,2,4,3,5}	5	10	50	30	60

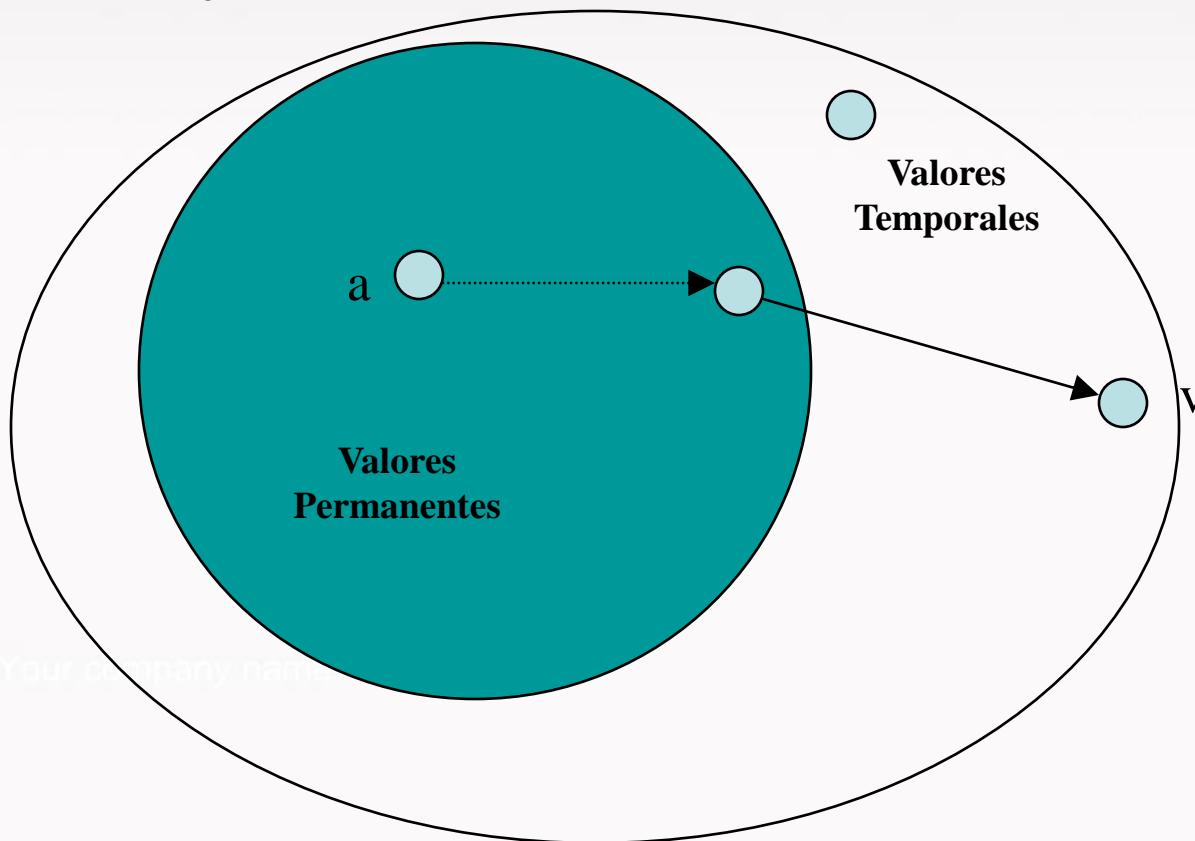
Demostración de la corrección

- **Hipotesis de Inducción $T(i)$:**
 - en la i -ésima iteracion del lazo del algoritmo, el valor $D[V]$ del vértice v se hace permanente
 - $D[v]$ siempre es el valor mínimo entre los valores temporales

El valor $D[v]$ da el camino mínimo desde el origen hasta el vértice v
- **Base ($i = 1$).**
 - Inicialmente $D[a] = 0$, y todos los demás D 's son mayores que cero, por tanto se elige el origen a
 - $D[a]$ es el camino mas corto desde a a a
 - $T(1)$ es cierta.

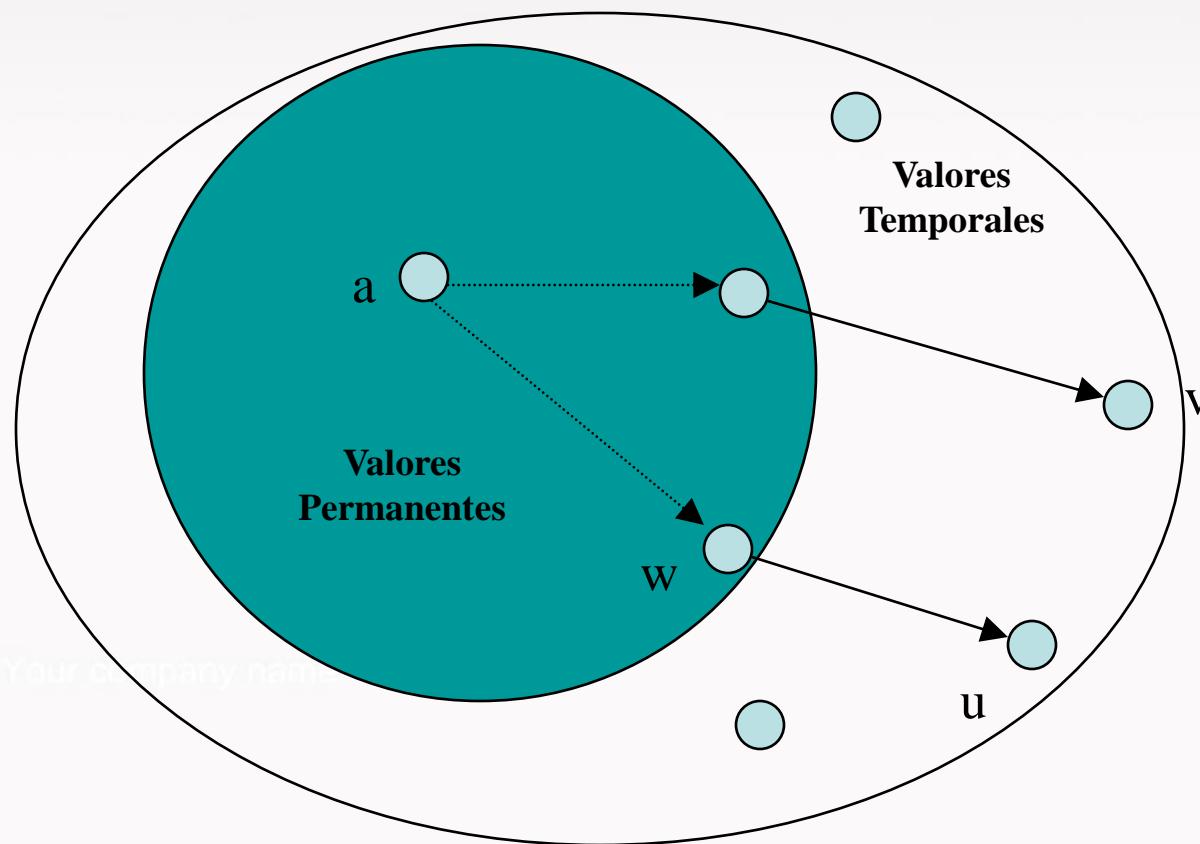
Demostración de la corrección

- Supongamos que $T(k)$ es cierto para todo $k < i$.
 - En la iteración $k+1$, seleccionamos el nodo v .
¿Es $D(v)$ el camino más corto desde a a v ?



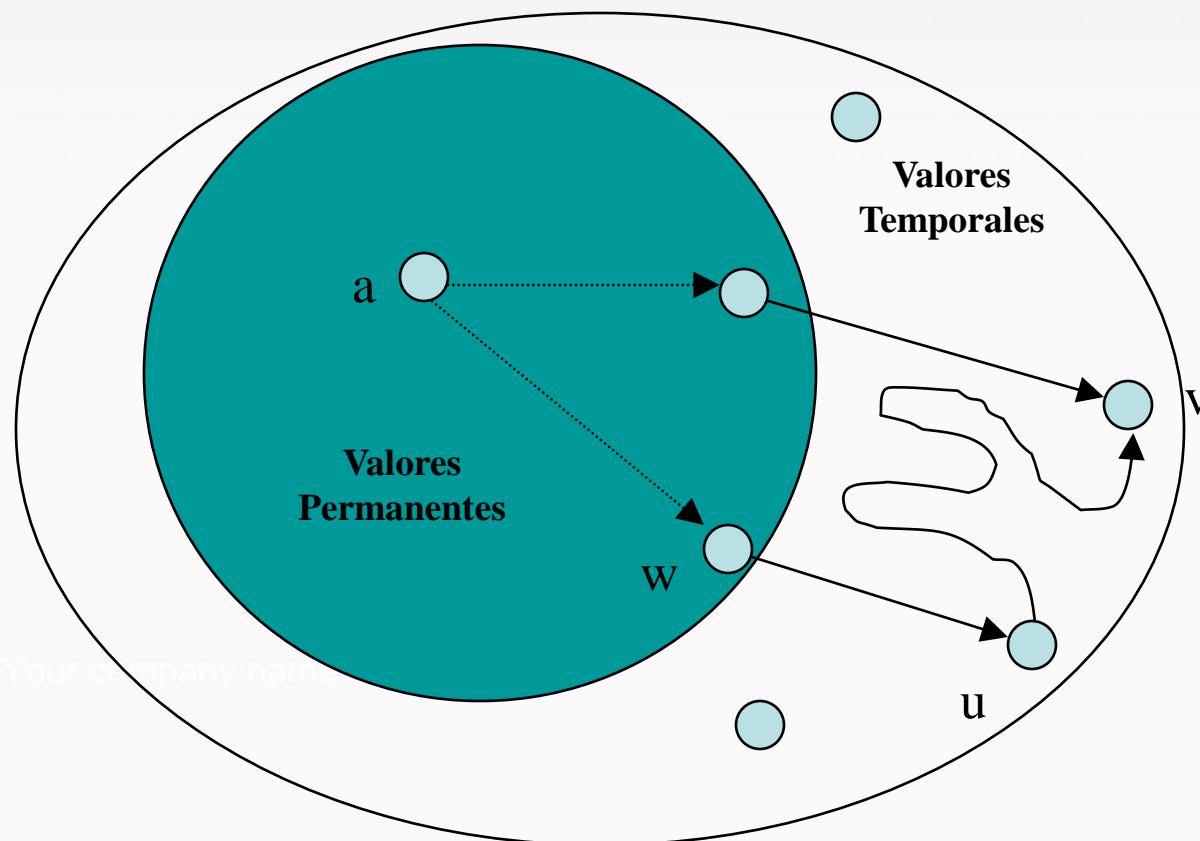
Demostración de la corrección

Supongamos que hay **otro vértice u** tal que el camino hasta v que pasa por u es mas corto que el anterior (el que nos daba $D(v)$).



Demostración de la corrección

Obligatoriamente el camino que pasa a través de **u** tiene que ser como el de la figura



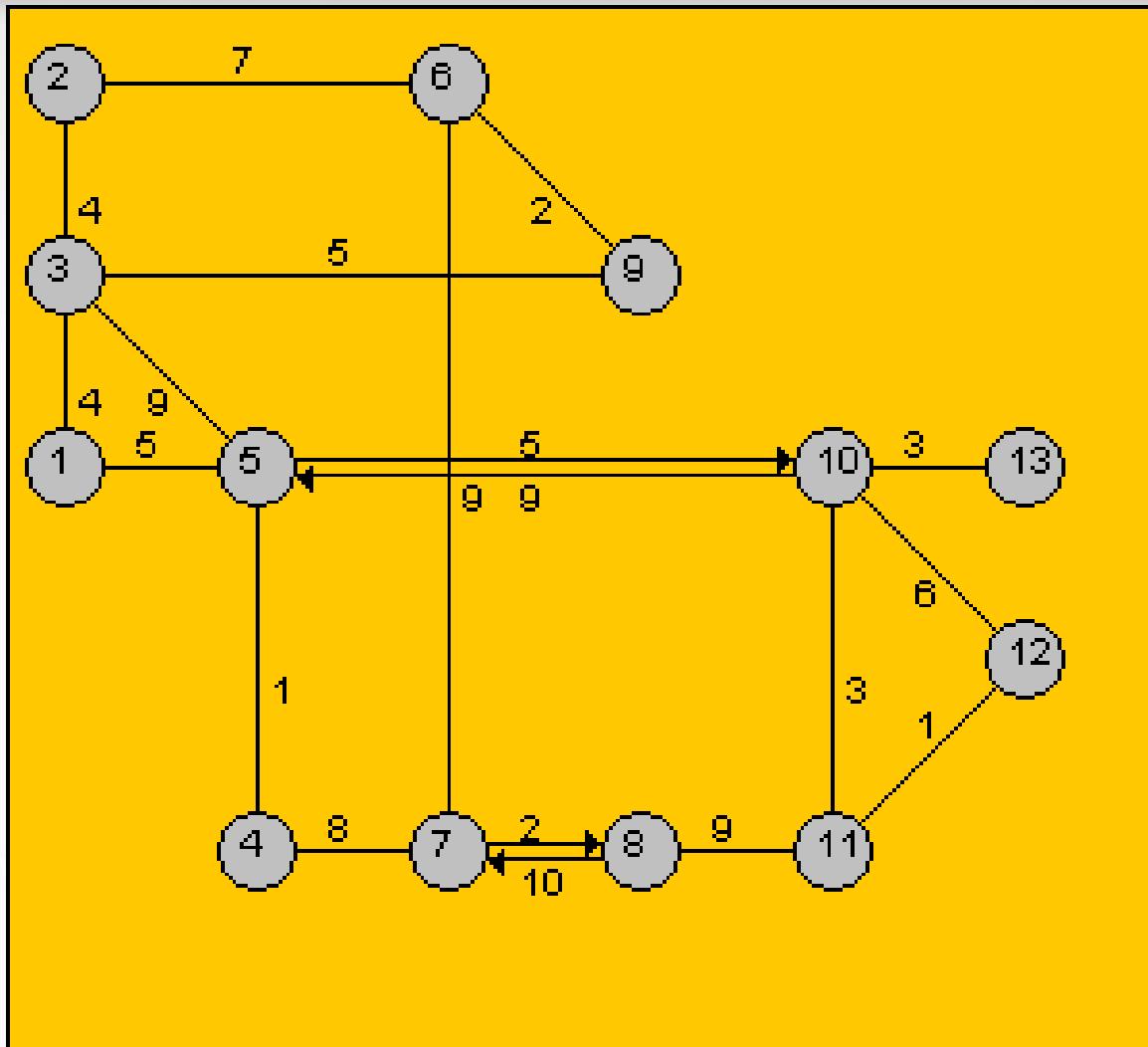
Demostración de la corrección

- La hipótesis de inducción era:
 - En la iteracion $k+1$ seleccionar el nodo v .
¿Es $D(v)$ el camino mas corto de a a v ?
 - La respuesta es SI.
 - Por tanto, $T(k+1)$ es cierto.
- Como la base y la hipótesis de inducción son ciertas, $T(i)$ es cierta para todo i .

Análisis de la eficiencia

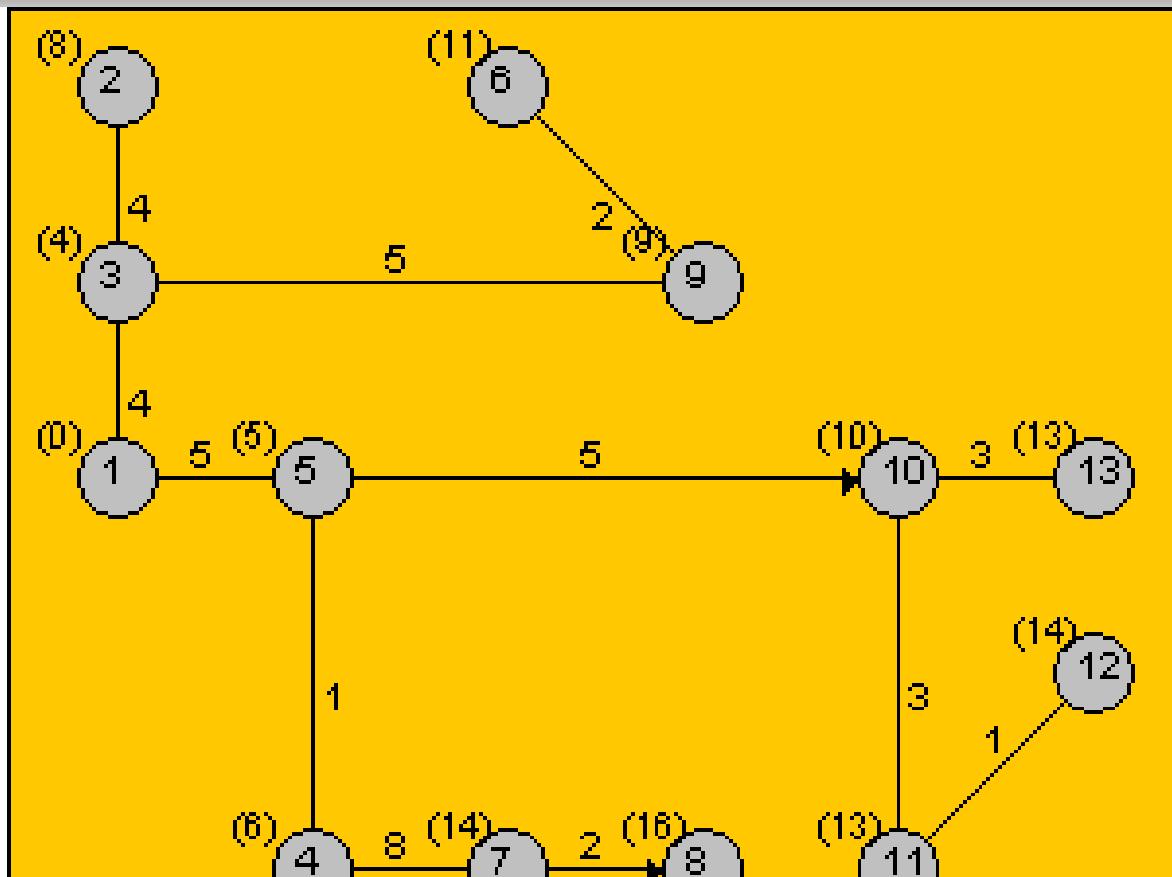
- El algoritmo es obvio que consume un tiempo en $O(n^2)$ en el peor caso
- La hipótesis de que las distancias sean no negativas es esencial
 - Si hay alguna distancia negativa el algoritmo no funciona correctamente: La hipótesis de inducción no puede demostrarse porque no se verifica la propiedad triangular.
- El algoritmo puede extenderse fácilmente para que de la distancia entre todos los pares de vértices: $O(n^3)$.

Ejemplo-1



Your company

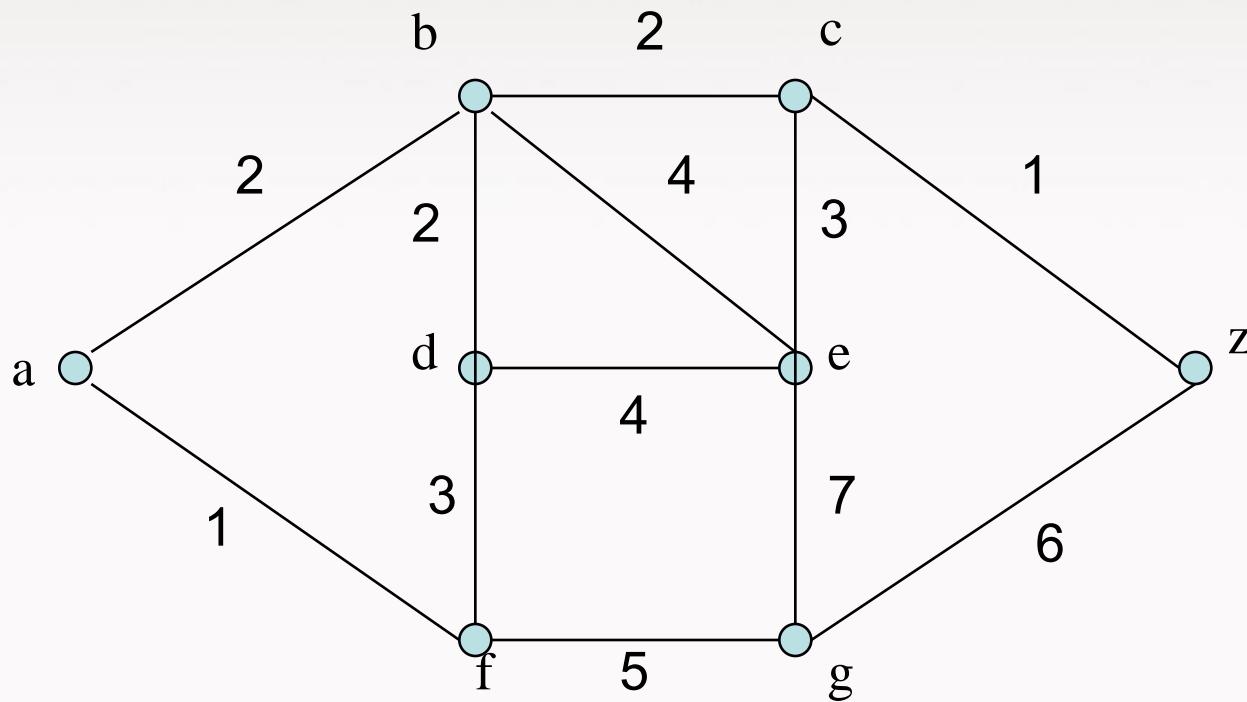
Ejemplo-1



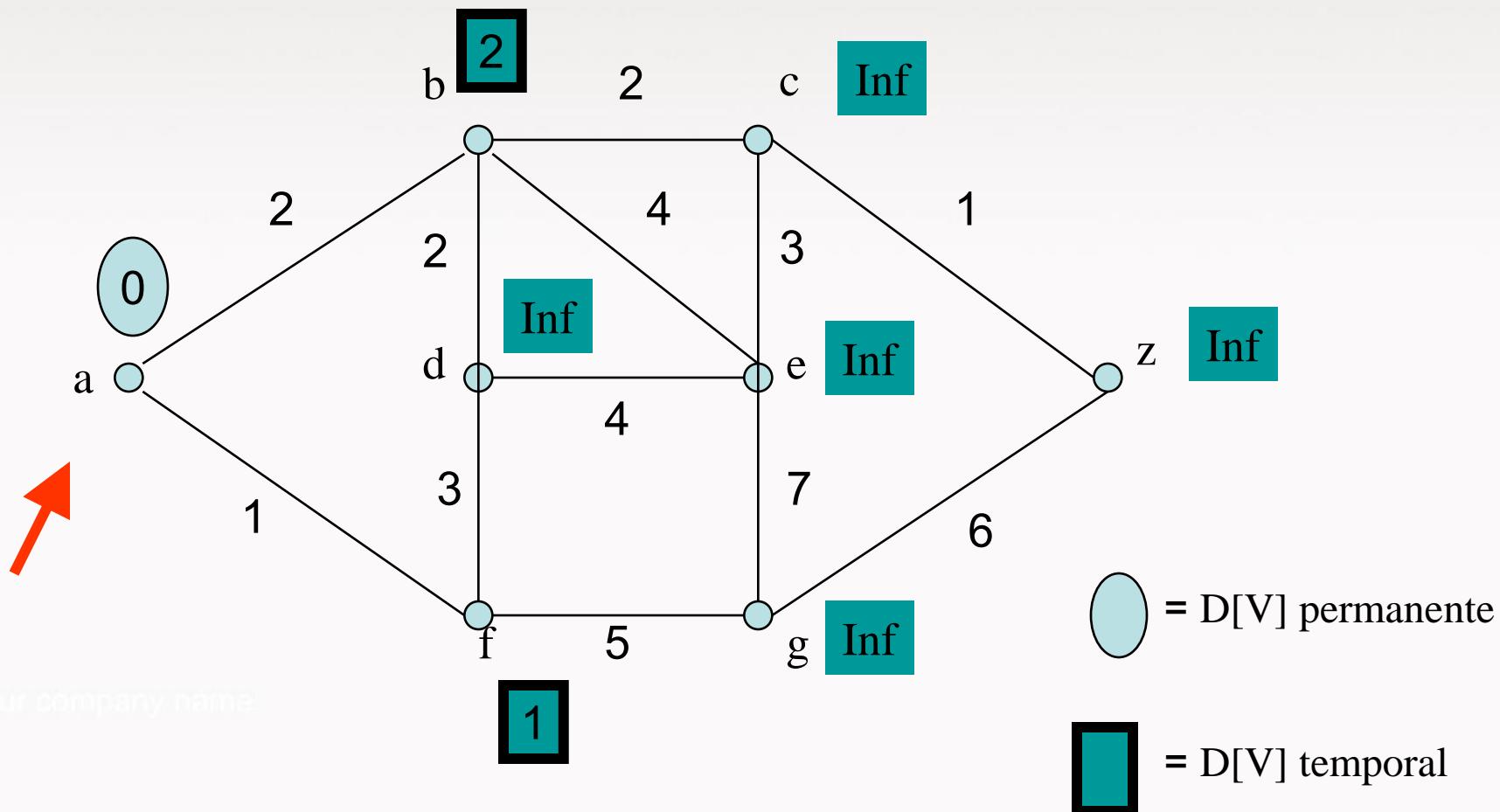
Your company

Los paréntesis contienen el mínimo coste para alcanzar el nodo

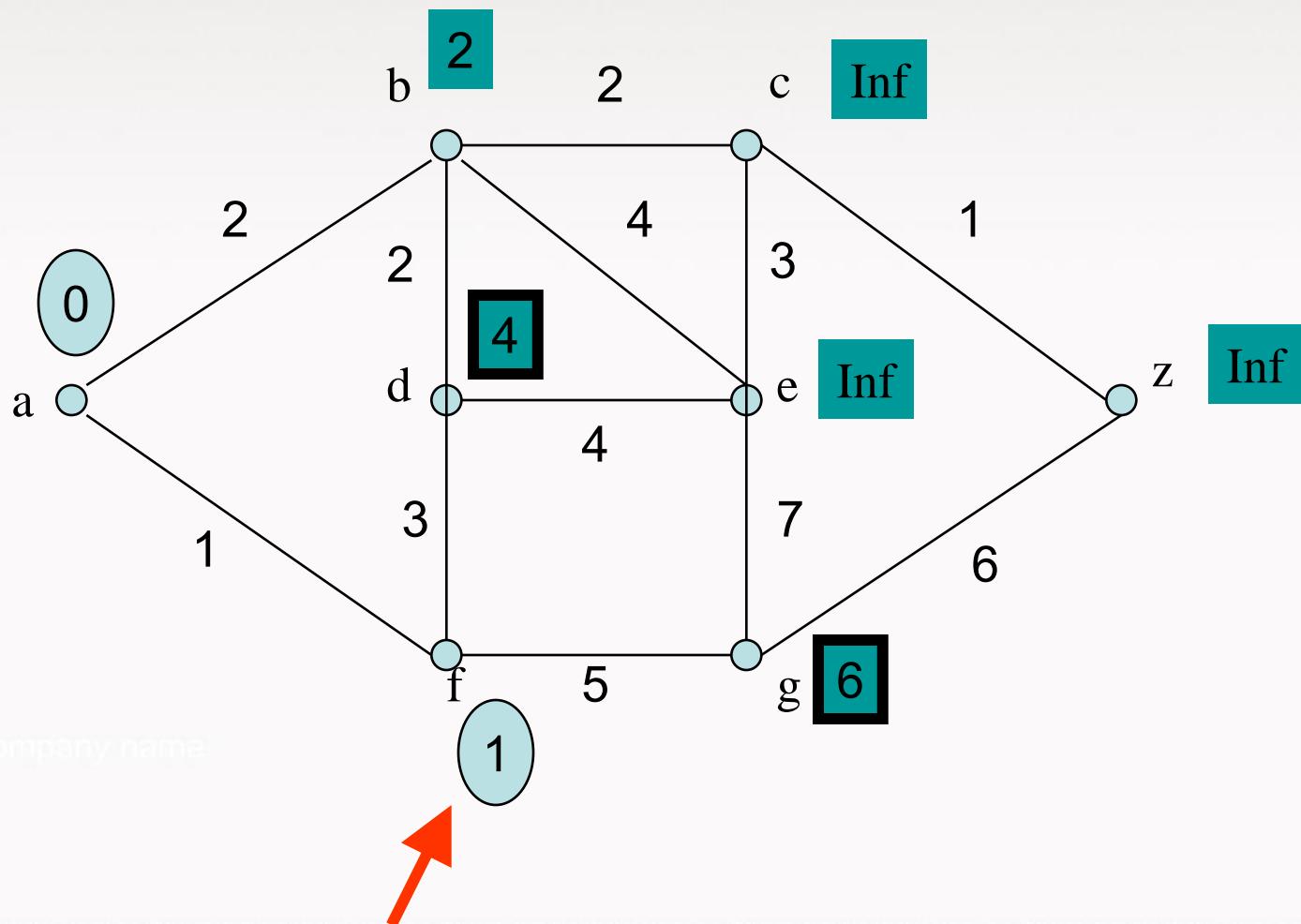
Ejemplo (Solo entre a y z)



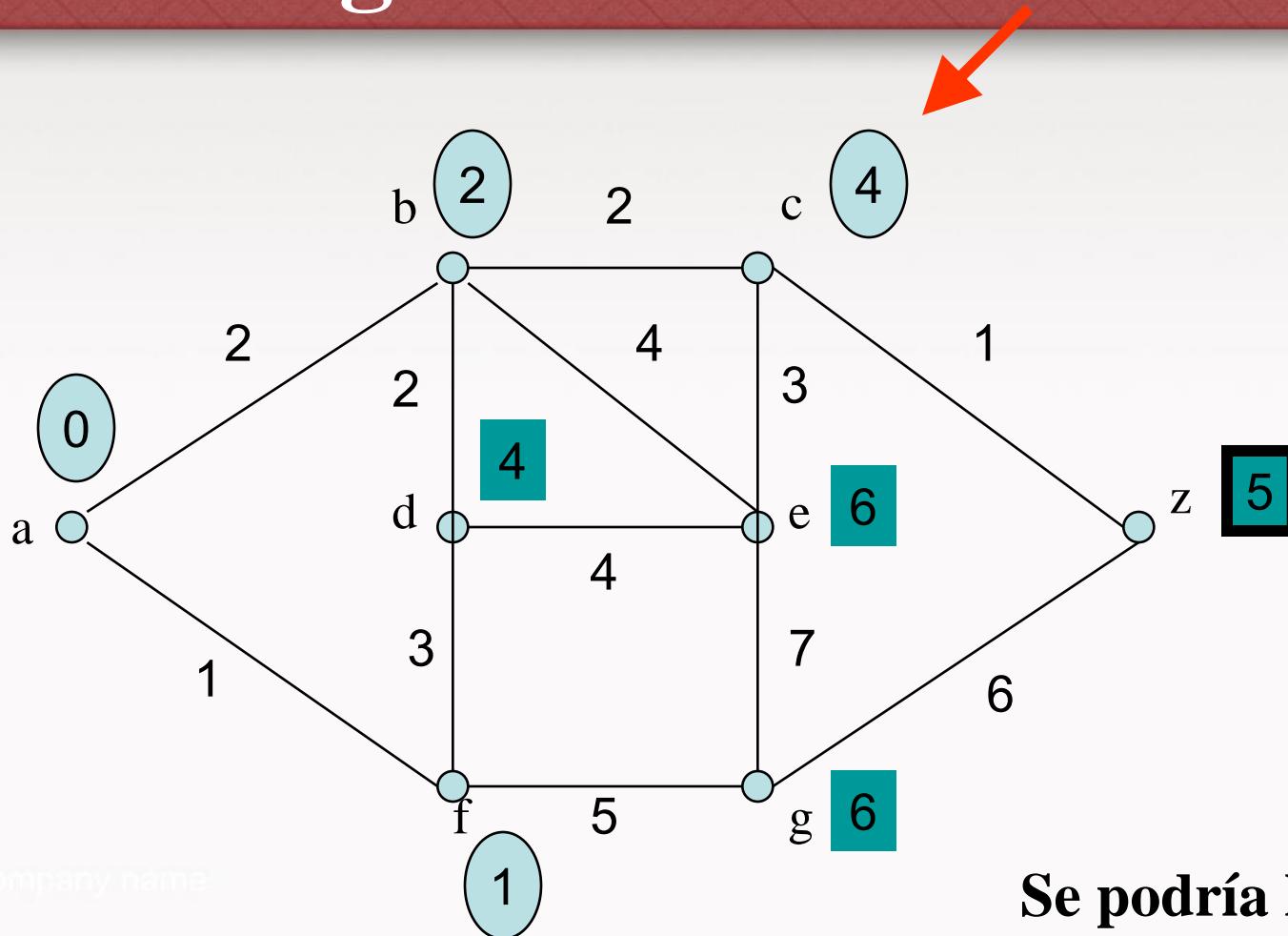
Inicialización



Primera Iteración

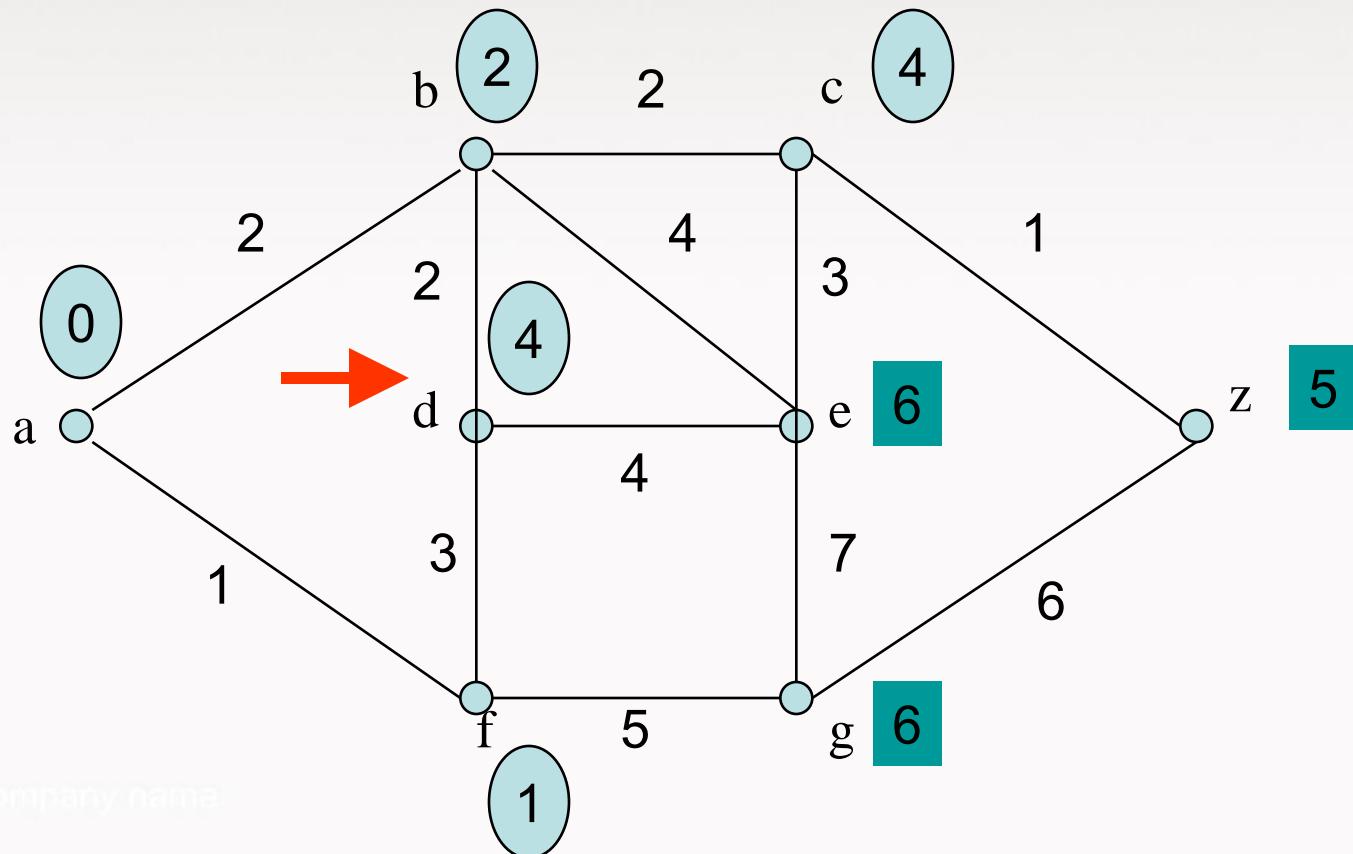


Segunda Iteración

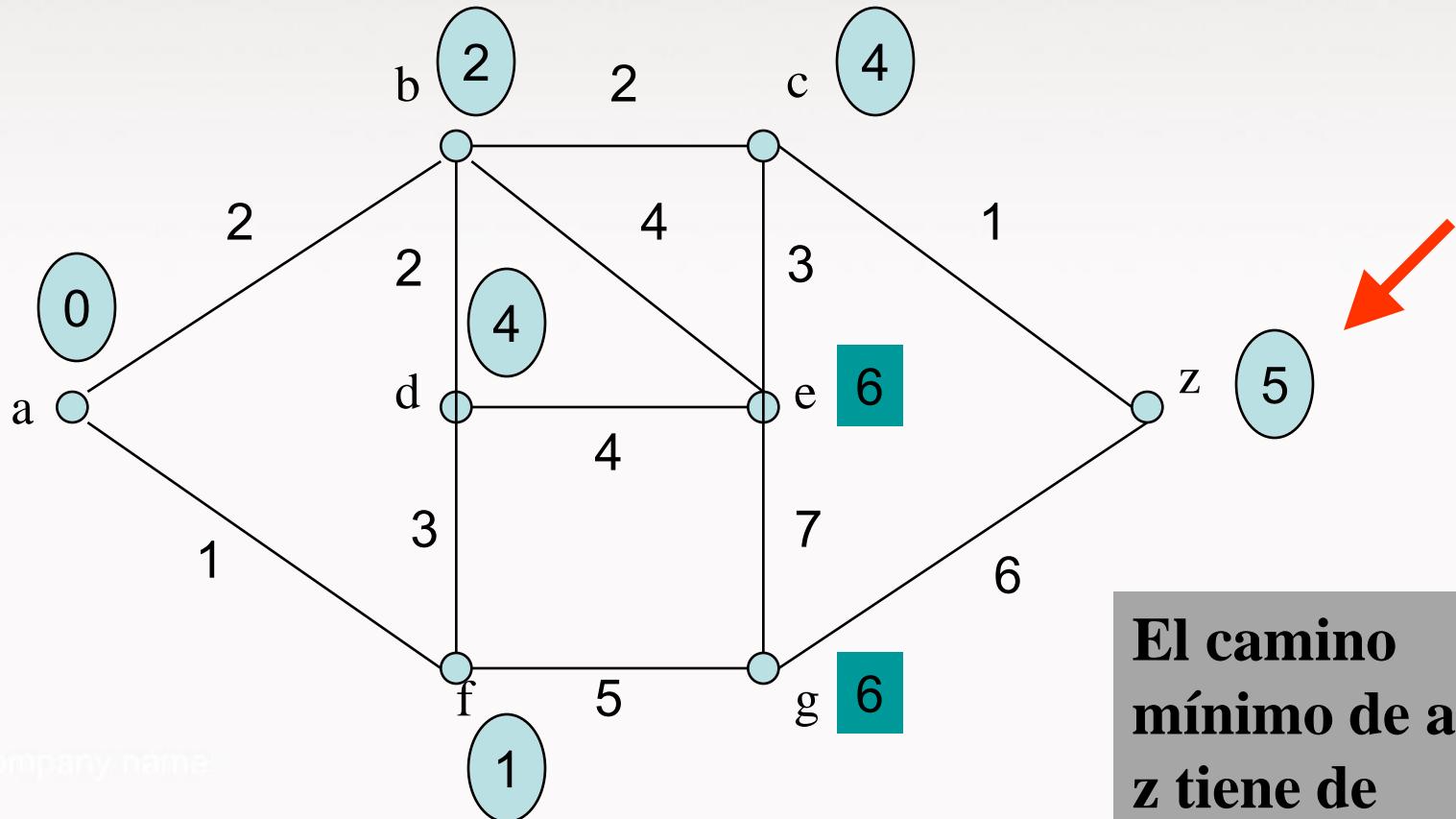


Se podría haber elegido tambien 'd'

Tercera Iteración



Cuarta (y última) Iteración



El camino mínimo de a a z tiene de longitud 5

Algorítmica

Capítulo 3. Algoritmos Greedy

Tema 9. Heurísticas Greedy

- Algoritmos greedy como heurísticas:

El Problema del Coloreo de un Grafo. El problema del Viajante de Comercio. El Problema de la Mochila. El Problema de la Asignación de Tareas

Heurísticas

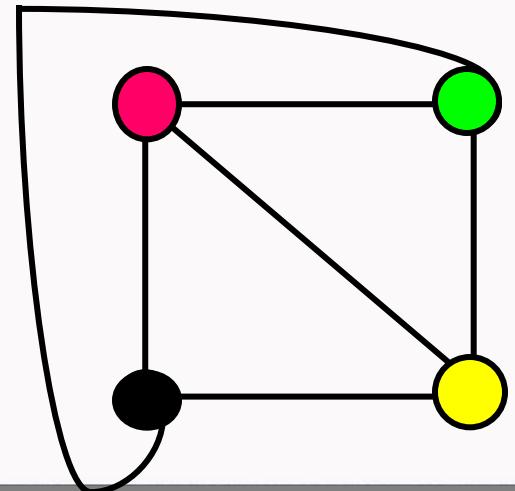
- Son procedimientos que, basados en la experiencia, proporcionan buenas soluciones a problemas concretos
 - Algoritmos Genéticos, Enfriamiento (Recocido) Simulado, Búsqueda Tabú,
 - Computación Evolutiva, GRASP (Greedy Randomized Adaptive Search Procedures), Búsqueda Dispersa, Colonias de Hormigas, Búsqueda por Entornos Variables, Búsqueda Local Guiada, Búsqueda Local Iterativa
 - Métodos Ruidosos, Aceptación de Umbrales, Algoritmos Meméticos, Redes de Neuronas, ...

Heurísticas Greedy

- Es mejor satisfacer que optimizar
- El tiempo efectivo que se tarda en resolver un problema es un factor clave
- Los algoritmos greedy son muy buenos como heurísticas
 - El Problema del Coloreo de un grafo
 - El Problema del Viajante de Comercio
 - El Problema de la Mochila
 - Problemas de Recubrimiento, de Rutas, ...
- Suelen usarse también para encontrar una primera solución (óptimo local)

El Problema del Coloreo de un Grafo

- Planteamiento
 - Dado un grafo plano $G = (V, E)$, determinar el mínimo número de colores que se necesitan para colorear todos sus vértices, y que no haya dos de ellos adyacentes pintados con el mismo color
- Si el grafo no es plano puede requerir tantos colores como vértices haya
- Las aplicaciones son muchas
 - Representación de mapas
 - Diseño de páginas webs
 - Diseño de carreteras



El Problema del Coloreo de un Grafo

- El problema es NP y por ello se necesitan heurísticas para resolverlo
- El problema reúne 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

Implementación del algoritmo

Funcion COLOREO

{COLOREO pone en NuevoColor los vertices 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

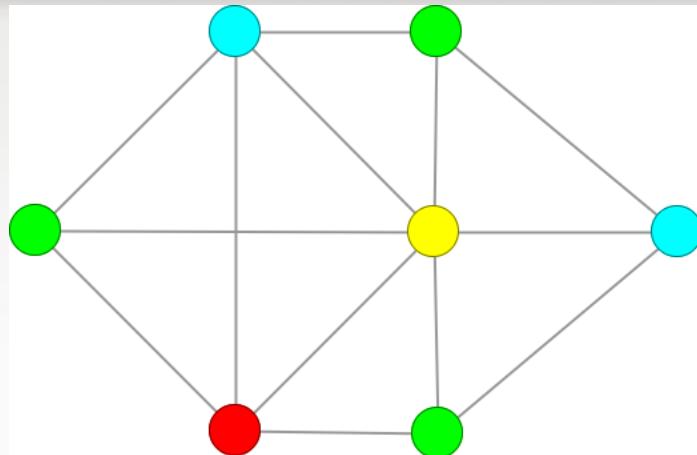
 Marcar v como coloreado

 Añadir v a NuevoColor

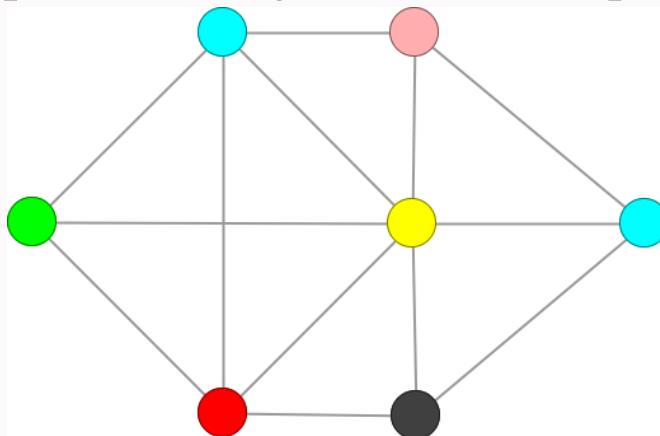
End

Se trata de un algoritmo que funciona en **O(n)**, pero que no siempre da la solución óptima

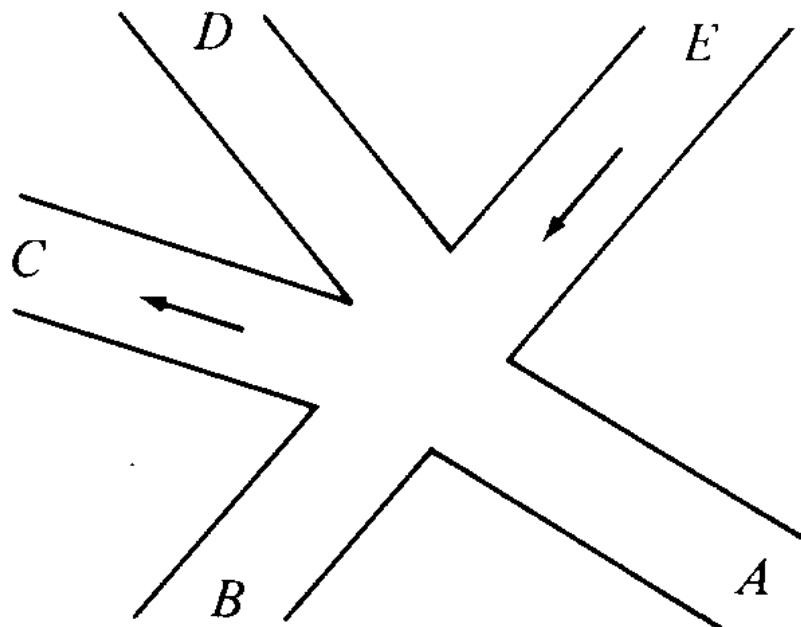
Ejemplo



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 siempre es así



Ejemplo: Diseño de cruces de semáforos



- A la izquierda tenemos un cruce de calles que señalan los sentidos de circulación.
- La falta de flechas, significa que podemos ir en las dos direcciones.
- Queremos diseñar un patrón de semáforos con el mínimo número de semáforos
- Suponemos un grafo cuyos vértices representan turnos, y cuyas aristas unen esos turnos que no pueden realizarse simultáneamente sin que haya colisiones
- El problema se convierte en uno de Coloreo de los Vértices de un Grafo

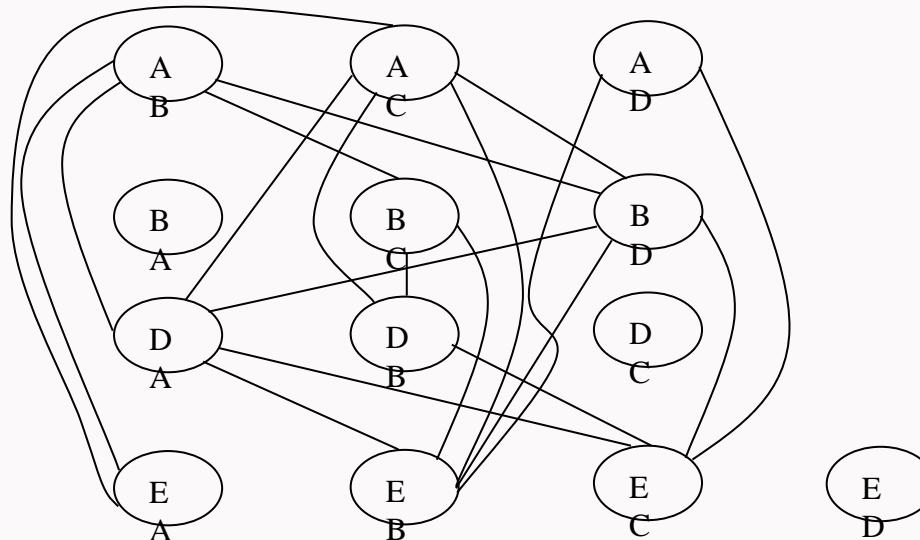
Y OUT COMING UP NEXT

Las calles sin dirección, son en doble sentido. Por lo tanto tendríamos:

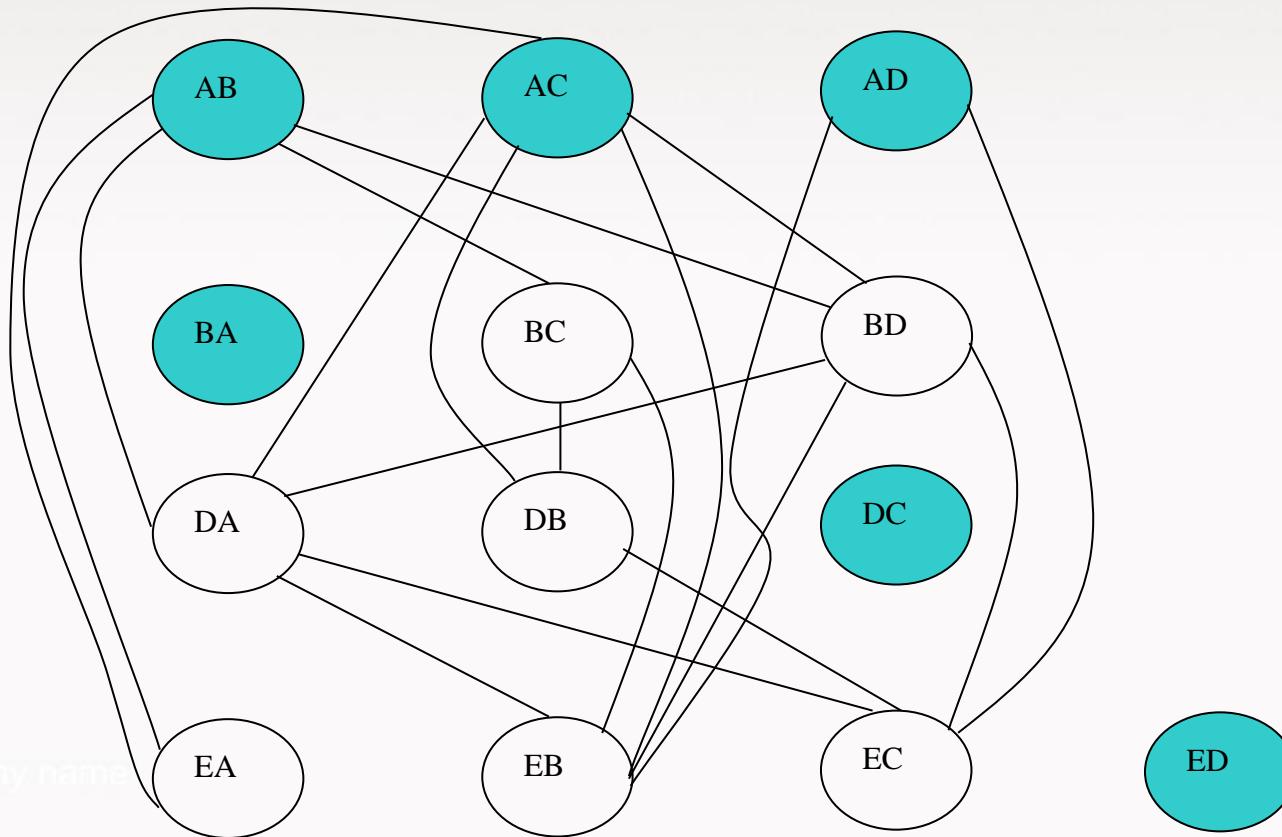
{AB, AC, AD, BA, BC, BD, DC, DA, DB, ED, EC, EA, EB}

Ejemplo: Diseño de cruces de semáforos

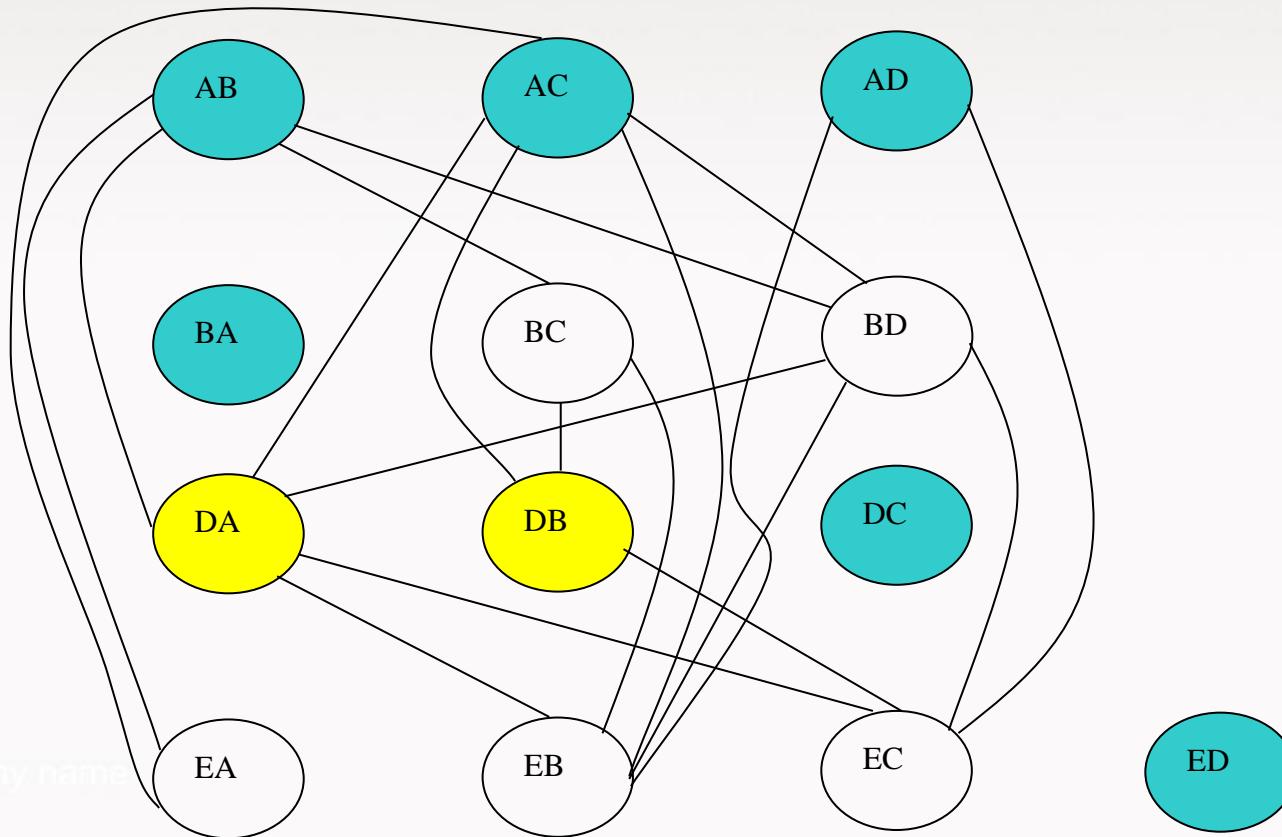
- Los nodos representan giros, y dos nodos son adyacentes si sólo si éstos son incompatibles.
- Una heurística razonable para este problema es la siguiente:
 - Comenzar coloreando todos los nodos que se pueda con un color, sin causar conflictos entre nodos adyacentes.
 - Si quedan nodos sin colorear, tomar un nuevo color y colorear tantos nodos no coloreados como se pueda, nuevamente sin causar conflictos.
 - Continuar el proceso con nuevos colores hasta que ya no queden nodos sin colorear.



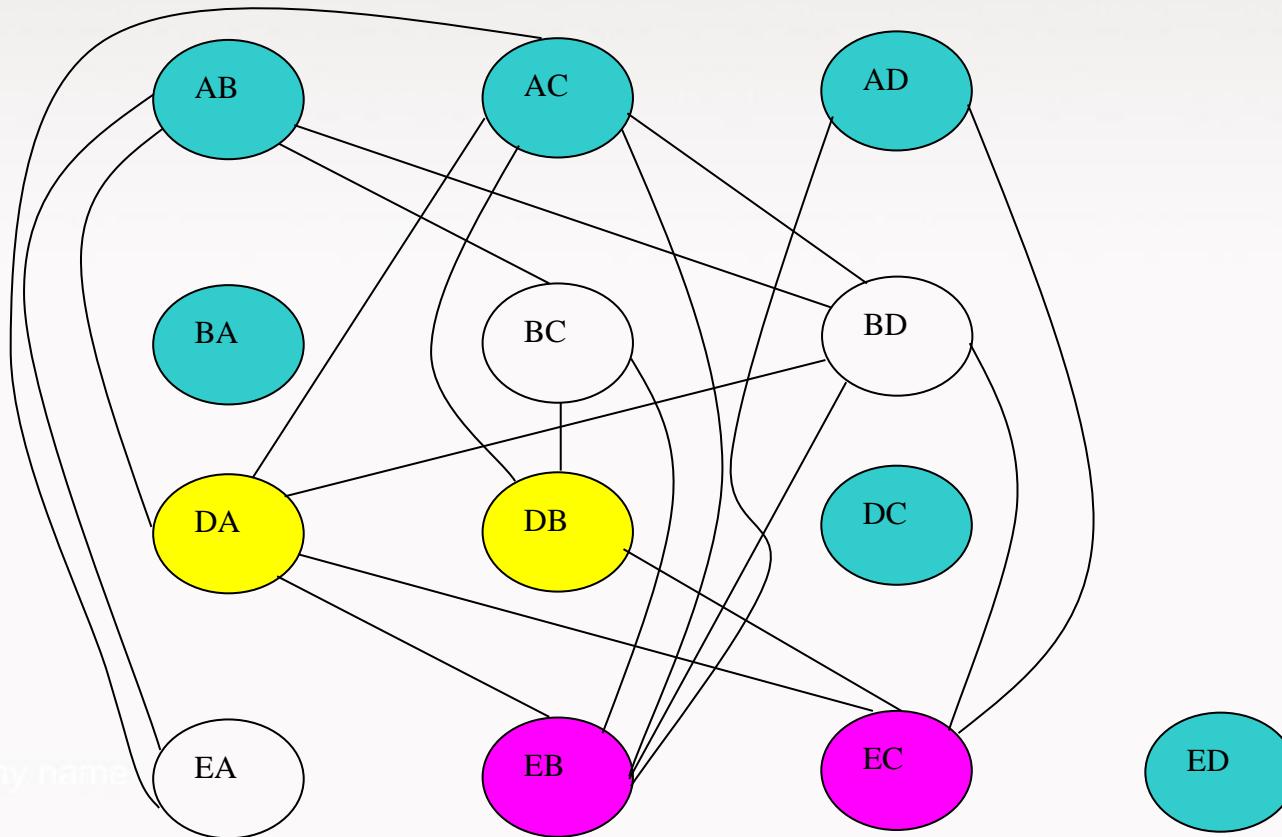
Ejemplo: Diseño de cruces de semáforos



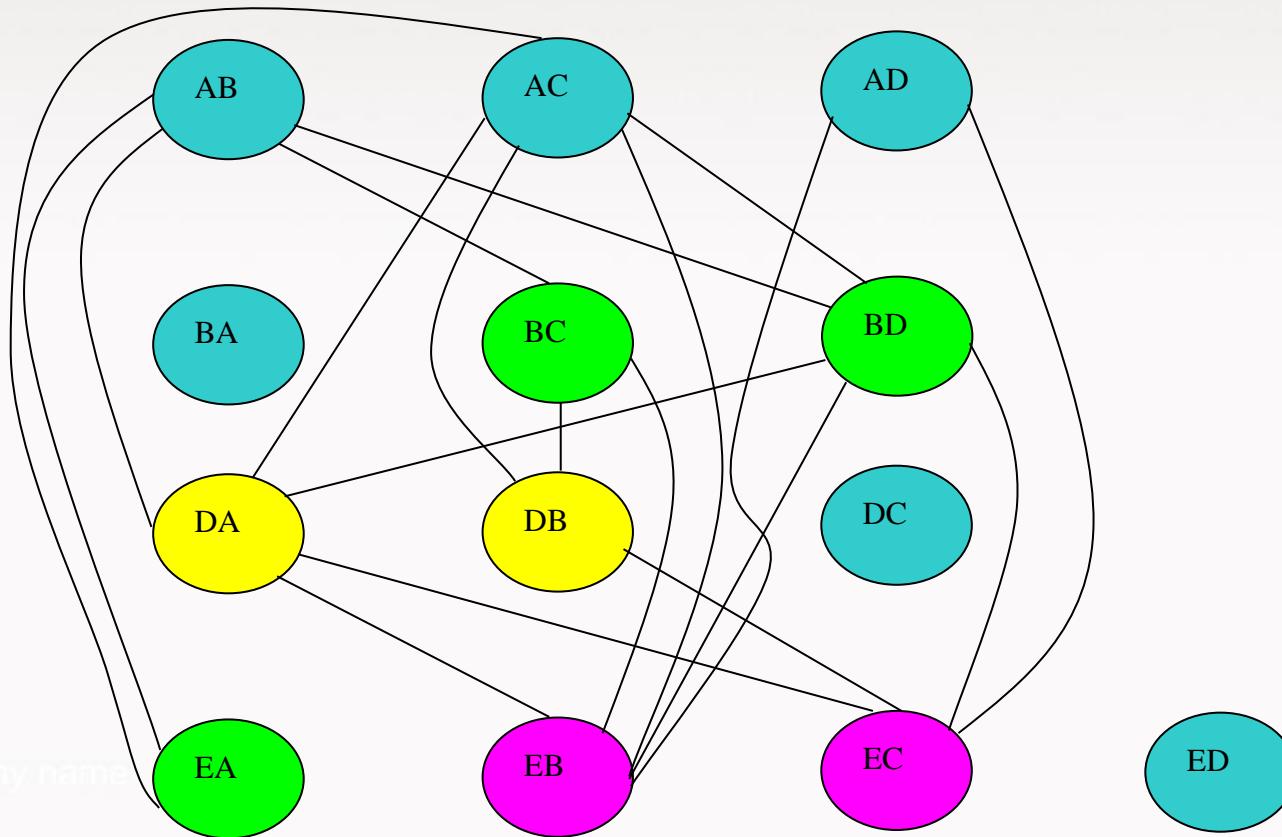
Ejemplo: Diseño de cruces de semáforos



Ejemplo: Diseño de cruces de semáforos



Ejemplo: Diseño de cruces de semáforos



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 de tamaños “pequeños”.
- Para grafos grandes, es necesario utilizar heurísticas, ya que el problema se hace intratable en el tiempo.
- El PVC es uno de los mas importantes en Algorítmica

El Problema del Viajante de Comercio

- Matemáticamente puede formularse como un problema de Programación Lineal con Números Enteros:

$$\begin{aligned} & \min \sum_{i=0}^n \sum_{j \neq i, j=0}^n c_{ij} x_{ij} \\ & 0 \leq x_{ij} \leq 1 \quad i, j = 0, \dots, n \\ & x_{ij} \text{ integer} \quad i, j = 0, \dots, n \\ & \sum_{i=0, i \neq j}^n x_{ij} = 1 \quad j = 0, \dots, n \\ & \sum_{j=0, j \neq i}^n x_{ij} = 1 \quad i = 1, \dots, n \\ & u_i - u_j + nx_{ij} \leq n - 1 \quad 1 \leq i \neq j \leq n. \end{aligned}$$

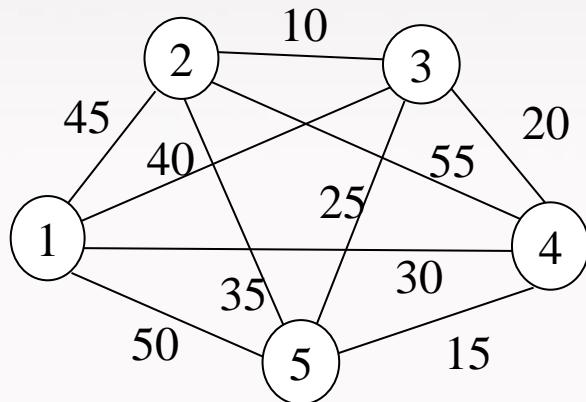
- La primera restricción asegura que desde cada origen $0, \dots, n$ se llegue a un destino, y la segunda asegura que desde cada ciudad $1, \dots, n$ se salga exactamente hacia una ciudad (ambas restricciones también implican que exista exactamente una salida desde la ciudad 0.)
- La última restricción obliga a que un solo camino cubra todas las ciudades y no dos o más caminos disjuntos cubran conjuntamente todas las ciudades.

El Problema del Viajante de Comercio

- En el contexto de la Algorítmica, suponemos un grafo no dirigido y completo $G = (N, A)$ y L una matriz de distancias no negativas referida a G .
- Se quiere encontrar un **Circuito Hamiltoniano Minimal**.
- Este es un problema Greedy típico, que presenta las 6 condiciones para poder ser enfocado con un algoritmo greedy
- Destaca de esas 6 características **la condición de factibilidad**:
 - que al seleccionar una arista no se formen ciclos,
 - que las aristas que se escojan cumplan la condición de no ser incidentes en tercera posición al nodo escogido

El Problema del Viajante de Comercio

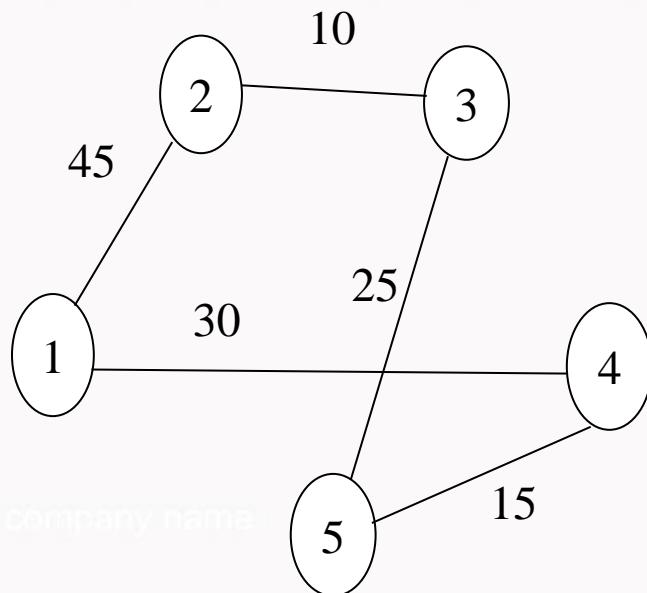
- Consideremos el siguiente grafo



- Posibilidades:
 - Los nodos son los candidatos. Empezar en un nodo cualquiera y en cada paso moverse al nodo no visitado más próximo al último nodo seleccionado.
 - Las aristas son los candidatos. Hacer igual que en el Algoritmo de Kruskal, pero garantizando que se forme un ciclo.

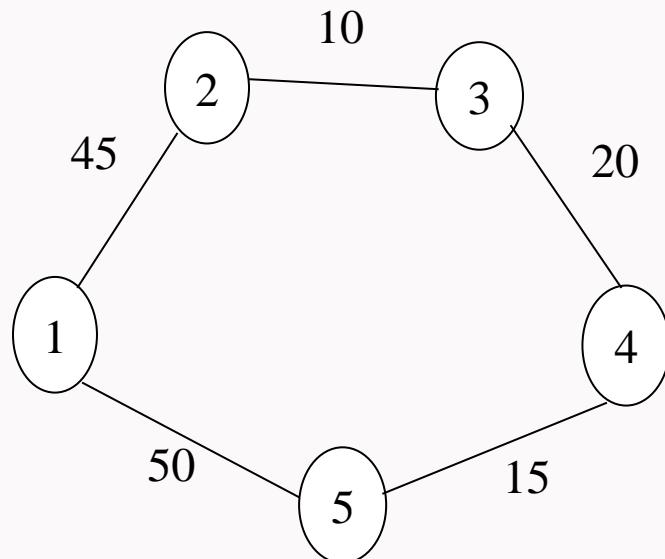
El Problema del Viajante de Comercio

- Solución con la primera heurística
- Solución empezando en el nodo 1



Solución: (1, 4, 5 ,3, 2), **125**

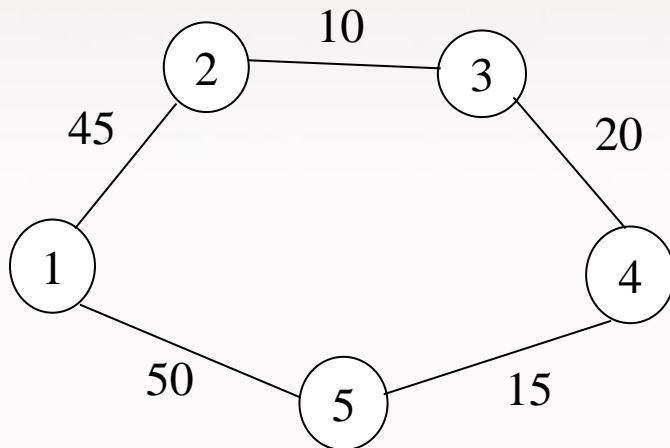
- Solución empezando en el nodo 5



Solución: (5,4,3, 2,1), **140**

El Problema del Viajante de Comercio

- Solución con la segunda heurística



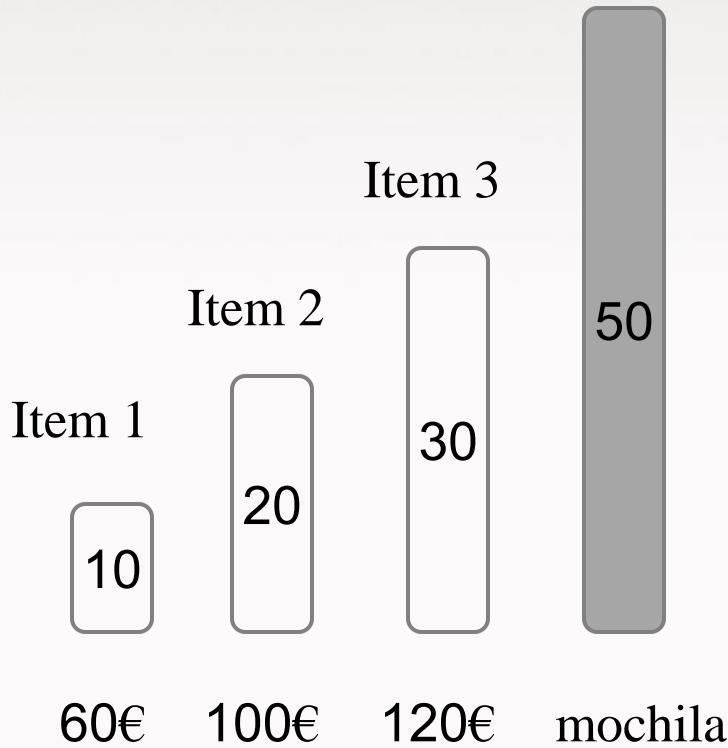
- Solución: ((2, 3), (4, 5), (3, 4), (1, 2), (1, 5))
Coste = $10+15+20+45+50 = 140$
- En todos los casos la eficiencia es la del algoritmo de ordenación que se use

El problema de la Mochila Fraccional

- 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

$$\begin{aligned} & \text{maximizar} && \sum_{1 \leq i \leq n} p_i x_i \\ & \text{sujeto a} && \sum_{1 \leq i \leq n} w_i x_i \leq M \\ & && \text{con } 0 \leq x_i \leq 1, 1 \leq i \leq n \end{aligned}$$

Ejemplo



Es un claro problema de tipo greedy

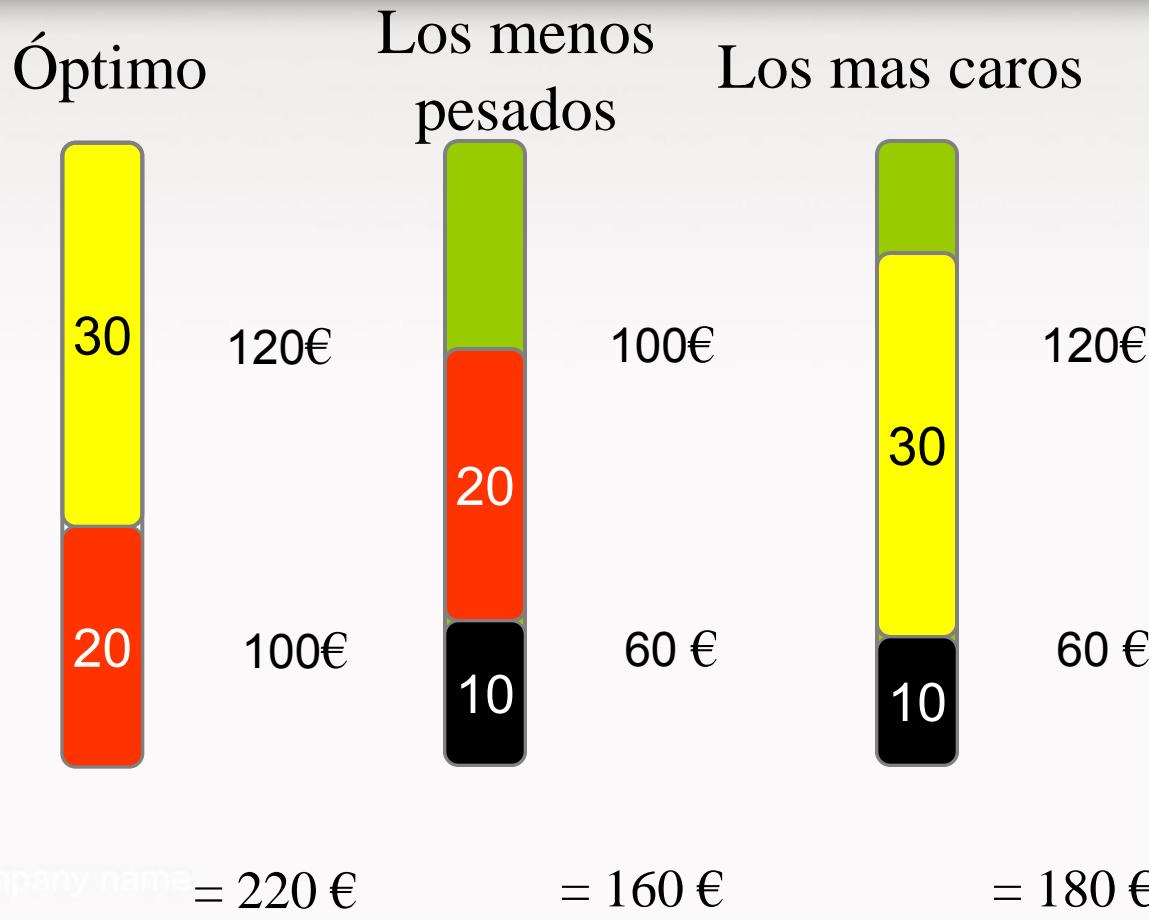
Sus aplicaciones son innumerables

Es un banco de pruebas algorítmico

La técnica greedy produce soluciones optimales para este tipo de problemas

También hay problemas de mochila booleanos (0 ó 1).

Mochila 0/1



¿Como seleccionamos los items?

Mochila fraccional

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

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

- Método 1 elegir primero el menos pesado**
 - $\text{Peso total} = 10 + 20 + 30 + 40 = 100$
 - $\text{Costo total} = 20 + 30 + 65 + 40 = 155$
- Método 2 elegir primero el mas caro**
 - $\text{Peso Total} = 30 + 50 + 20 = 100$
 - $\text{Costo Total} = 65 + 60 + 20 = 145$

Mochila fraccional

- Método 3 elegir primero el que tenga mayor valor por unidad de peso (razón 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

$$\begin{aligned} - \text{Peso Total} &= 30 + 10 + 20 + 40 = 100 \\ - \text{Costo Total} &= 65 + 20 + 30 + 48 = 163 \end{aligned}$$

Otro ejemplo de Mochila Fraccional

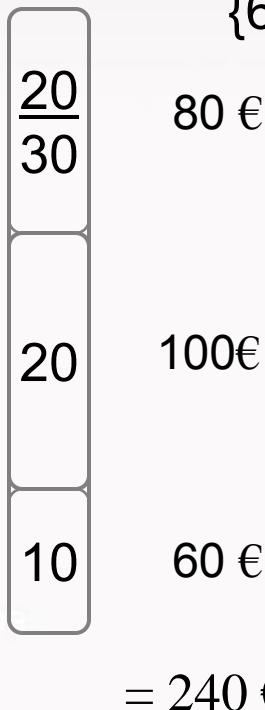
- Supongamos el siguiente caso de problema de la mochila: $n = 3$, $M = 20$, $(p_1, p_2, p_3) = (25, 24, 15)$ y $(w_1, w_2, w_3) = (18, 15, 10)$

(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1) $(1/2, 1/3, 1/4)$	16.5	24.25
2) $(1, 2/15, 0)$	20	28.2
3) $(0, 2/3, 1)$	20	31
4) $(0, 1, 1/2)$	20	31.5

Mochila fraccional

- Tomando los items en orden de mayor valor por unidad de peso, se obtiene una solucion optimal

$$\{60/10, 100/20, 120/30\}$$



Silvano Martello



Paolo Toth

Solución 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.
- 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.
- Primero, se ordenan los objetos por densidad no creciente, es decir,

$$p_i/w_i \geq p_{i+1}/w_{i+1} \text{ para } 1 \leq i < n.$$

- Entonces se actúa de la siguiente manera

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 $P(I)/W(I) \geq P(I+1)/W(I+1)$. M es la capacidad de la mochila y X(1:n) es el vector solution//

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

 Si $I \leq n$ Entonces $X(I) = cr/W(I)$ endif

End MOCHILA_GREEDY

Demostración de la corrección

- Vamos a demostrar que el algoritmo siempre encuentra la solución óptima del problema
- Sea $p_1/w_1 \geq p_2/w_2 \geq \dots \geq p_n/w_n$
- Sea $X = (x_1, x_2, \dots, x_n)$ la solución generada por MOCHILA_GREEDY, para una capacidad M
- Sea $Y = (y_1, y_2, \dots, y_n)$ una solución factible cualquiera
- Queremos demostrar que

Your company name

$$\sum_{i=1}^n (x_i - y_i) p_i \geq 0$$

Demostración de la corrección

1 2

n

1	1	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---

- Si todos los x_i son 1, la solución es claramente óptima (es la única solución). En otro caso, sea k el menor número tal que $x_k < 1$.

1 2

k

n

1	1	1	x_k	0	0	0	0
---	---	----	----	---	-------	---	---	----	----	---	---

Demostración de la corrección

1 2 k n

1	1	1	x_k	0	0	0	0
---	---	----	----	---	-------	---	---	----	----	---	---

1 2

$$\sum_{i=1}^n (x_i - y_i)p_i = \sum_{i=1}^{k-1} (x_i - y_i)w_i \frac{p_i}{w_i} + (x_k - y_k)w_k \frac{p_k}{w_k} + \sum_{i=k+1}^n (x_i - y_i)w_i \frac{p_i}{w_i}$$

Demostración de la corrección

- Consideremos cada uno de esos bloques

$$\sum_{i=1}^{k-1} (x_i - y_i) w_i \frac{p_i}{w_i} \geq \sum_{i=1}^{k-1} (x_i - y_i) w_i \frac{p_k}{w_k}$$

$$(x_k - y_k) w_k \frac{p_k}{w_k} = (x_k - y_k) w_k \frac{p_k}{w_k}$$

$$\sum_{i=k+1}^n (x_i - y_i) w_i \frac{p_i}{w_i} \geq \sum_{i=k+1}^n (x_i - y_i) w_i \frac{p_k}{w_k}$$

Demostración de la corrección

$$\sum_{i=1}^n (x_i - y_i)p_i \geq \sum_{i=1}^n (x_i - y_i)w_i \frac{p_k}{w_k}$$

$$\frac{p_k}{w_k} \sum_{i=1}^n (x_i - y_i)w_i$$

$\sum_i x_i w_i = M$ por hipótesis, pero $\sum_i y_i w_i \leq M$

= W

Por tanto, como siempre $W > 0$, entonces

$$\sum_{i=1}^n (x_i - y_i)p_i \geq 0$$

La asignación de tareas

- Supongamos que disponemos de n trabajadores y n tareas. Sea $b_{ij} > 0$ el coste de asignarle el trabajo j al trabajador i . Una asignación de tareas puede ser expresada como una asignación de los valores 0 ó 1 a las variables x_{ij} , donde
 - $x_{ij} = 0$ significa que al trabajador i no le han asignado la tarea j , y
 - $x_{ij} = 1$ indica que sí.
- Una asignación válida es aquella en la que a cada trabajador sólo le corresponde una tarea y cada tarea está asignada a un trabajador.

La asignación de tareas

- Dada una asignación válida, definimos el coste de dicha asignación como:

$$\sum_{i=1}^n \sum_{j=1}^n x_{ij} b_{ij}$$

- Diremos que una asignación es óptima si es de mínimo coste.
- Cara a diseñar un algoritmo greedy para resolver este problema podemos pensar en dos estrategias distintas: asignar cada trabajador la mejor tarea posible, o bien asignar cada tarea al mejor trabajador disponible.
- Sin embargo, ninguna de las dos estrategias tiene por qué encontrar siempre soluciones óptimas. ¿Es alguna mejor que la otra?

La asignación de tareas

- Lamentablemente, el algoritmo greedy no funciona para todos los casos como pone de manifiesto la siguiente matriz:

		1	2	3
		16	20	18
Trabajador	1	16	20	18
	2	11	15	17
	3	17	1	20

- Para ella, el algoritmo produce una matriz de asignaciones en donde los “unos” están en las posiciones (1,1), (2,2) y (3,3), esto es, asigna la tarea i al trabajador i ($i = 1, 2, 3$), con un valor de la asignación de 51 (= 16 + 15 + 20).
- Sin embargo la asignación óptima se consigue con los “unos” en posiciones (1,3), (2,1) y (3,2), esto es, asigna la tarea 3 al trabajador 1, la 1 al trabajador 2 y la tarea 2 al trabajador 3, con un valor de la asignación de 30 (= 18 + 11 + 1).

Algoritmo Húngaro

- Para resolver este problema hay un algoritmo exacto:
- Algoritmo Húngaro
 - Nos referimos a él en el segundo capítulo cuando hablamos de la necesidad de diseñar nuevos algoritmos
 - Haremos un ejercicio para ver como funciona, y que se conozca su existencia.

Ejemplo

Cada uno de cuatro laboratorios, A,B,C y D tienen que ser equipados con uno de cuatro equipos informáticos. El costo de instalación de cada equipo en cada laboratorio lo da la tabla. Queremos encontrar la asignación menos costosa.

	1	2	3	4
A	48	48	50	44
B	56	60	60	68
C	96	94	90	85
D	42	44	54	46

Algoritmo Húngaro:

- Etapa 1:
 - Encontrar el elemento de menor valor en cada fila de la matriz $n \times n$ de costos.
 - Construir una nueva matriz restando a cada costo el menor costo de su fila.
 - En esta nueva matriz, encontrar el menor costo de cada columna.
 - Construir una nueva matriz (llamada de **costos reducidos**) restando a cada costo el menor costo de su columna.

Algoritmo Húngaro:

- Etapa 2:
 - Rayar el minimo numero de lineas (horizontal y/o vertical) que se necesiten para tachar todos los ceros de la matriz de costos reducidos.
 - Si se necesitan n lineas para tachar todos los ceros, hemos encontrado una solución óptima en esos ceros tachados.
 - Si tenemos menos de n lineas para tachar todos los ceros, ir a la etapa 3.

Algoritmo Húngaro

- Etapa 3:
 - Encontrar el menor elemento no cero (llamar a su valor k) en la matriz de costos reducidos que no este tachado por alguna linea de las pintadas en la etapa 2.
 - Restar k a cada elemento no tachado en la matriz de costos reducidos y sumar k a cada elemento de la matriz de costos reducidos que este tachado por dos lineas.
 - Volver a la Etapa 2.

Ejemplo

Cada uno de cuatro laboratorios, A,B,C y D tienen que ser equipados con uno de cuatro equipos informáticos. El costo de instalación de cada equipo en cada laboratorio lo da la tabla. Queremos encontrar la asignación menos costosa.

	1	2	3	4
A	48	48	50	44
B	56	60	60	68
C	96	94	90	85
D	42	44	54	46

Aplicación del método húngaro

Restamos 44 en la fila 1

	1	2	3	4
A	4	4	6	0
B	56	60	60	68
C	96	94	90	85
D	42	44	54	46

Aplicación del método húngaro

Restamos 56 en la fila 2, 85 en la fila 3 y 42 en la fila 4

	1	2	3	4
A	4	4	6	0
B	0	4	4	12
C	11	9	5	0
D	0	2	12	4

Aplicación del método húngaro

Restamos 2 en la columna
2 y 4 de la columna 3

Hay una solución factible
En las celdas con ceros?

	1	2	3	4
A	4	2	2	0
B	0	2	0	12
C	11	7	1	0
D	0	0	8	4

$$3 \neq 4$$

Aplicación del método húngaro

Minimo elemento no tachado

	1	2	3	4
A	4	2	2	0
B	0	2	0	12
C	11	7	1	0
D	0	0	8	4

The diagram illustrates the application of the Hungarian method to a 4x5 cost matrix. The matrix rows are labeled A, B, C, D and columns 1, 2, 3, 4. Horizontal lines are drawn under rows B, C, and D, and vertical lines are drawn through columns 2, 3, and 4. The cell at (B, 2) contains the value 2, which is circled in red. The cell at (C, 3) contains the value 1, also circled in red. A red arrow points from the text "Minimo elemento no tachado" to the cell (B, 2). Another red arrow points from the same text to the cell (C, 3).

Cost Matrix:

	1	2	3	4
A	4	2	2	0
B	0	2	0	12
C	11	7	1	0
D	0	0	8	4

APLICACIÓN DEL MÉTODO HÚNGARO

	1	2	3	4
A	$4-1=3$	$2-1=1$	$2-1=1$	0
B	0	2	0	$12+1=13$
C	$11-1=10$	$7-1=6$	$1-1=0$	0
D	0	0	8	$4+1=5$

Aplicación del método húngaro

	1	2	3	4	
A	3	1	1	0	
B	0	2	0	13	
C	10	6	0	0	
D	0	0	8	5	

Your company name

Ahora es posible la asignación óptima
usando las celdas con ceros

APLICACIÓN DEL MÉTODO HÚNGARO

	1	2	3	4
A	3	1	1	X 0
B	X 0	2	0	13
C	10	6	0 X	0
D	0	X 0	8	5



Algorítmica

Capítulo 5: Algoritmos para la Exploración de Grafos.

Tema 14: La técnica “Backtracking”

- Método general “backtracking”.
- Árboles. Espacio de estados.
- Algoritmo general Backtracking.
- Eficiencia del método
- El problema de las 8 reinas
- El problema de la suma de subconjuntos
- Circuitos hamiltonianos

El método General

- El backtracking es una de las técnicas mas generales para la exploración de Grafos.
- Muchos problemas que buscan en un conjunto de soluciones o que buscan una solución óptima que satisfaga algunas restricciones, pueden resolverse con backtracking.
- El nombre backtrack lo acuñó D.H. Lehmer en los 50.
- R.J. Walker, dio una versión algorítmica (1960)
- Golomb y Baumert presentaron una descripción general del backtracking asociada a una gran variedad de aplicaciones.

Backtrack Programming

SOLOMON W. GOLOMB* AND LEONARD D. BAUMERT

Jet Propulsion Laboratory, Pasadena, Calif.

Abstract. A widely used method of efficient search is examined in detail. This examination provides the opportunity to formulate its scope and methods in their full generality. In addition to a general exposition of the basic process, some important refinements are indicated. Examples are given which illustrate the salient features of this searching process.

1. Introduction

Virtually all problems of search, optimization, simultaneous or sequential decision, etc. can be fitted into the following formulation framework.

From the direct product space $X_1 \times X_2 \times \cdots \times X_n$ of the n "selection spaces" X_1, X_2, \dots, X_n , which may or may not be copies of one another, find the "sample vector" (x_1, x_2, \dots, x_n) with x_i an element of X_i , which is extremal with respect to (e.g., which maximizes) the criterion function $\phi(x_1, x_2, \dots, x_n)$. The value of the criterion function for the extremal case is also to be determined.

If the extremal vector is not unique, it may suffice to find one such vector or it may be necessary to find them all, depending on the nature of the problem. Frequently the criterion function is two valued, corresponding to acceptable and unacceptable (1 and 0), and it may not be known in advance whether acceptable solutions exist. The formulation covers this case as well as the multivalued or continuous-valued criterion function, since one or all sample vectors which maximize the criterion function are still sought.

Techniques of differential calculus and of the calculus of variations are appropriate for dealing with certain kinds of differentiable criterion functions on certain kinds of continuous product spaces. The methods of linear programming apply to linear criterion functions on convex polyhedra. For higher degree polynomial criterion functions there are more elaborate techniques (e.g., quadratic programming). For multistage decision processes which satisfy Bellman's "principle of optimality," the viewpoint of dynamic programming (which is indeed more of a viewpoint than a precise method) is applicable. Undoubtedly there are other special techniques for other specialized situations. However the purpose of this article is to describe a completely general approach to all such problems, whether or not more specialized techniques are applicable. This rather universal method was named *backtrack* by Professor D. H. Lehmer of the University of California at Berkeley. Backtrack has been independently "discovered" and applied by many people. A fairly general exposition of backtrack was given by R. J. Walker [11]. Even so, it is believed that this is the first attempt to formulate the scope and methods of backtrack programming in its full generality. Naturally when more specialized techniques are applicable they are frequently more efficient than backtrack, although this is not always the case.

2. Basic Formulation of Backtrack

We wish to determine the vector (x_1, x_2, \dots, x_n) from the Cartesian product space $X_1 \times X_2 \times \cdots \times X_n$ which maximizes the criterion function

* Present address: University of Southern California.

Backtracking

- Supongamos que tenemos que tomar una serie de decisiones entre una gran variedad de opciones donde,
 - No tenemos suficiente informacion como para saber que elegir
 - Cada decision nos lleva a un nuevo conjunto de decisiones
 - Alguna sucesion de decisiones (pero posiblemente mas de una) pueden ser solucion de nuestro problema
- El Backtracking es un metodo de busqueda de esas sucesiones de decisiones que conduce a encontrar una que nos convenga

El método General

- Para aplicar el método backtracking, la solución deseada debe ser expresable como una n-tupla (x_1, \dots, x_n) en la que x_i se elige de algún conjunto finito S_i .
- A menudo el problema a resolver trata de encontrar un vector que maximiza (o minimiza) una función criterio $P(x_1, \dots, x_n)$.
- A veces, también se trata de encontrar todos los vectores que satisfagan P .

¿No se parece esto a las técnicas Greedy?

Un ejemplo

- Ordenar los enteros en $A(1..n)$ es un problema cuya solución es expresable mediante una n-tupla en la que x_i es el índice en A del i-esimo menor elemento.
- La función de criterio P es la desigualdad
$$A(x_i) \leq A(x_{i+1}), 1 \leq i \leq n.$$
- El conjunto S_i es finito e incluye a todos los enteros entre 1 y n .
- La ordenación no es uno de los problemas que habitualmente se resuelven con backtracking

El método general

- Sea m_i el tamaño del conjunto S_i .
- Hay $m = m_1 \cdot m_2 \cdot \dots \cdot m_n$ n-tuplas posibles candidatos a satisfacer la función P .
- El enfoque de la fuerza bruta propondría formar todas esas tuplas y evaluar cada una de ellas con P , escogiendo la que diera un mejor valor.
- Backtracking proporciona la misma solución pero en mucho menos de m intentos.

El método general

- La idea básica es construir el mismo vector escogiendo una componente cada vez, y usando funciones de criterio modificadas $P_i(x_1, \dots, x_n)$, que a veces se llaman funciones de acotación, para testear si el vector que se está formando tiene posibilidad de éxito.
- La principal ventaja de este método es que si a partir del vector parcial (x_1, x_2, \dots, x_i) se deduce que no se podrá construir una solución, entonces pueden ignorarse por completo $m_{i+1} \cdot \dots \cdot m_n$ posibles test de vectores.

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
- En Programación Dinámica, la solución se construye por etapas a partir del principio de optimalidad, y los resultados se almacenan para no tener que recalcular, lo que no es posible en Backtracking

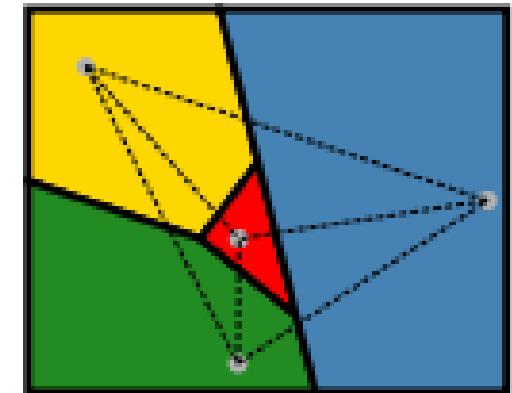
Ejemplo: Salir de un laberinto

- En un laberinto, encontrar camino desde la entrada
- En cada intersección hay que entre varias alternativas:
 - Seguir recto
 - Girar a la izquierda
 - Girar a la derecha
- No tenemos suficiente información para decidir bien
- Cada elección nos lleva a otro conjunto de elecciones
- Una sucesión de elecciones pueden (o no) llevarnos a una solución
- Muchos recorridos de laberintos pueden resolverse con backtracking



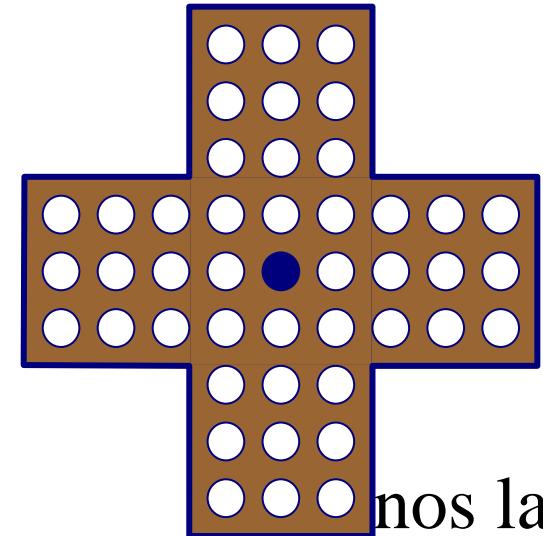
El coloreo de un mapa

- Queremos colorear un mapa con 4 colores a lo sumo
 - rojo, amarillo, azul, verde
- Los países adyacentes deben tener colores diferentes
- No tenemos suficiente información para elegir los colores
- Cada elección nos conduce a otro conjunto de elecciones
- Una sucesión de elecciones puede (o no) llevarnos a una solución
- Muchos problemas de coloreo pueden resolverse con backtracking



Resolver un puzzle

- En este puzzle, todos las piedras menos una son blancas
- Podemos saltar de una piedra a otra
- Las piedras saltadas se eliminan
- Queremos eliminar todas las piedras
- No tenemos suficiente información para saltar correctamente
- Cada elección lleva a otro conjunto de elecciones
- Una sucesión de elecciones puede (o no) llevarnos a una solución
- Muchos problemas de puzzles pueden resolverse con backtracking



nos la

Tipos de restricciones en backtracking

- Muchos de los problemas que resolveremos usando backtracking requieren que todas las soluciones satisfagan un complejo conjunto de restricciones.
- En cualquier problema que consideremos, estas restricciones podrán dividirse en dos categorías:
 - explícitas e implícitas.

Tipos de restricciones en backtracking

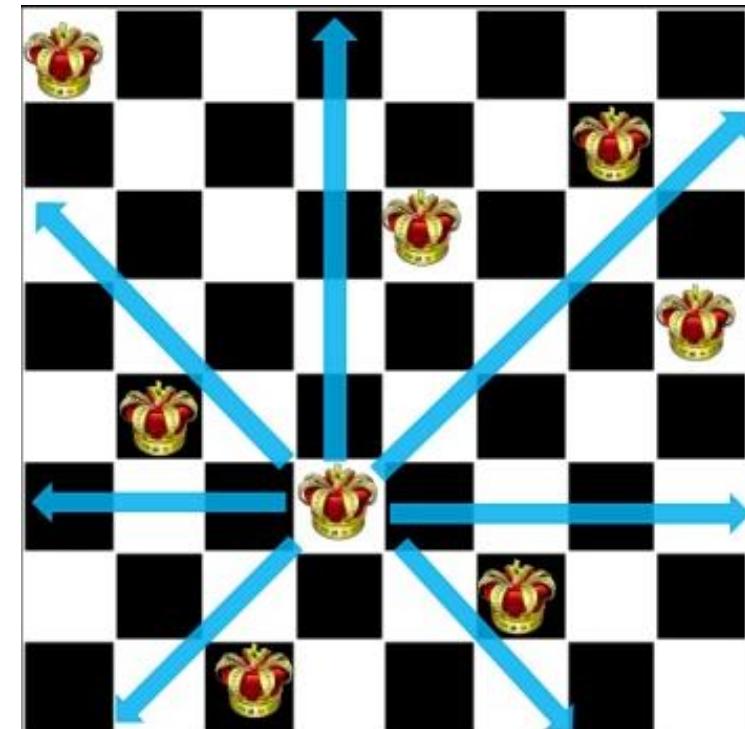
- Las **restricciones explícitas** son reglas que restringen cada x_i a tomar valores solo en un conjunto dado. Ejemplos comunes de restricciones explícitas son,
 - $x_i \geq 0 \quad S_i = \{n^{\text{os}} \text{ reales no negativos}\}$
 - $x_i = 0,1 \quad S_i = \{0,1\}$
 - $l_i \leq x_i \leq u_i \quad S_i = \{a: l_i \leq a \leq u_i\}$
- Las restricciones explícitas pueden depender o no del caso particular del problema a resolver.
- Las restricciones explícitas son condiciones frontera, es decir, vienen dadas por el propio planteamiento del problema.

Tipos de restricciones en backtracking

- Todas las tuplas que satisfacen las restricciones explícitas definen un espacio de soluciones del caso que estamos resolviendo.
- Las **restricciones implícitas** determinan cual de las tuplas en ese espacio solución satisface la función de criterio, es decir, describen la forma en la que las x_i deben relacionarse entre si.
- Por tanto:
 - Las restricciones implícitas son las restricciones que impone el camino que define la solución.
- En definitiva, dan las tuplas del espacio solución que satisfacen la función de criterio.

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 .

Tipos de restricciones: ejemplo

- Para este problema,

$$\max z = 10x_1 + 15x_2$$

s.a:

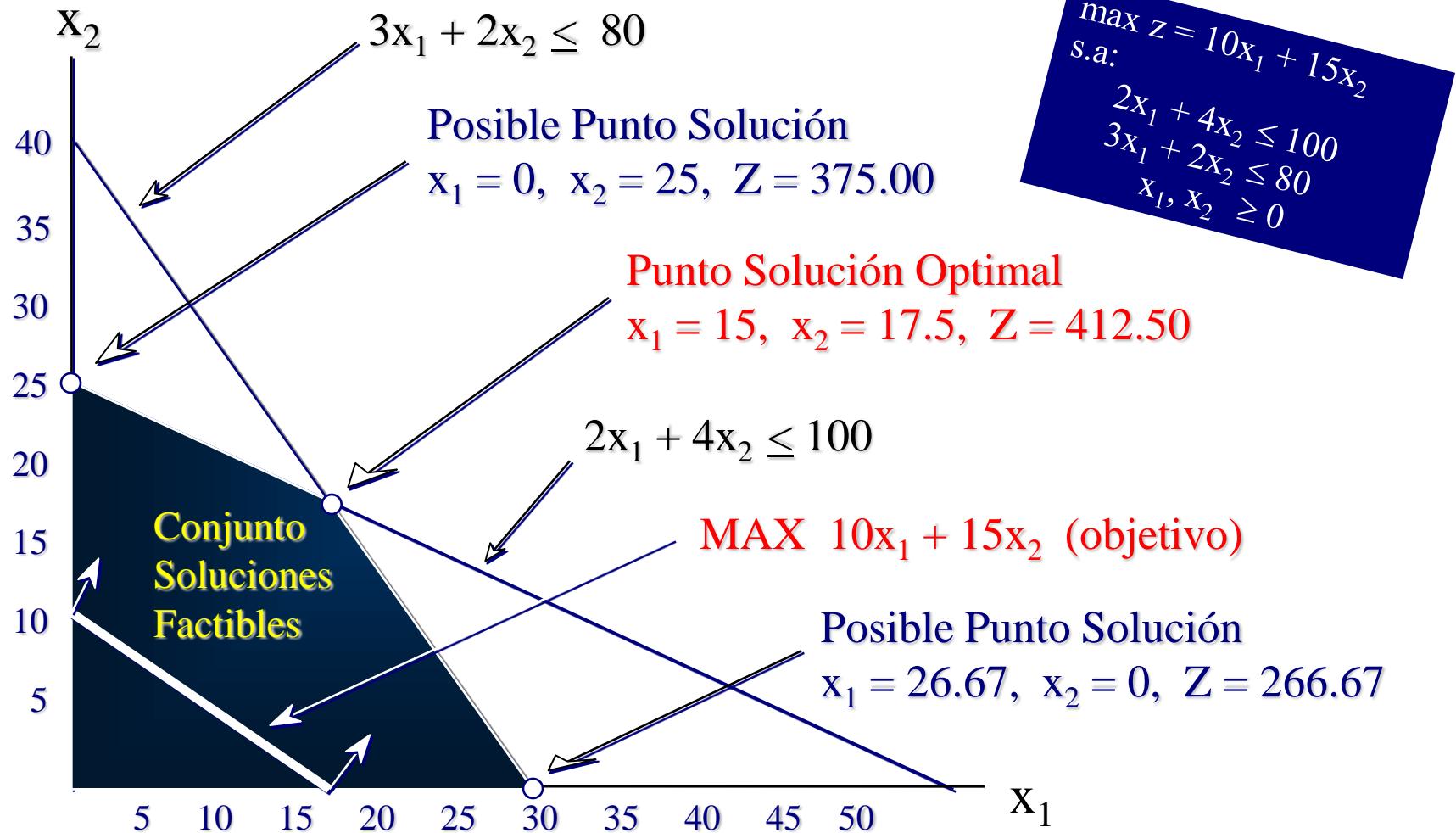
$$2x_1 + 4x_2 \leq 100$$

$$3x_1 + 2x_2 \leq 80$$

$$x_1, x_2 \geq 0$$

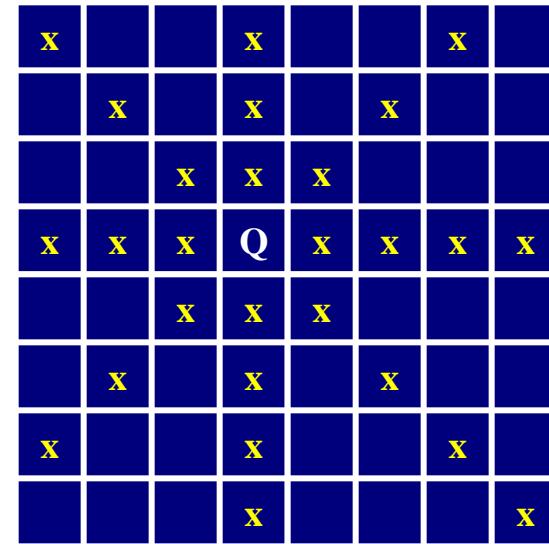
- Las restricciones explicitas las describe el conjunto de soluciones factibles (el conjunto donde toman valores las variables)
- Las restricciones implicitas describen los puntos que pueden ser solucion del problema (los puntos extremos del poliedro)

Tipos de restricciones: Interpretación



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.

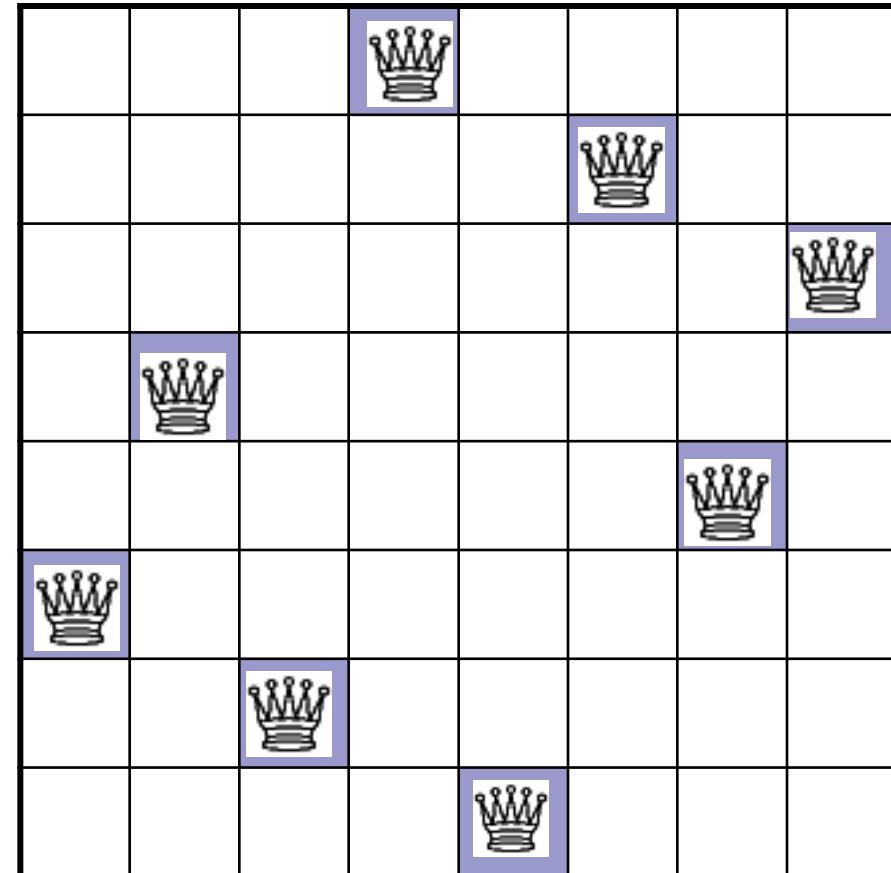


Como cada reina debe estar en una fila diferente, sin perdida de generalidad podemos suponer que la reina i se coloca en la fila i .

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.
- 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 satisfaciendo algunas restricciones.
- Otra formulación del problema:
 - Cada subconjunto solución se representa por una n-tupla (x_1, \dots, x_n) tal que $x_i \in \{0,1\}$, $1 \leq i < n$, y $x_i = 0$ si w_i no se elige y $x_i = 1$ si se elige w_i .
 - 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.

Espacios de soluciones

- Los algoritmos backtracking determinan las soluciones del problema buscando en el espacio de soluciones del caso considerado sistemáticamente.
- Esta búsqueda se facilita 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.

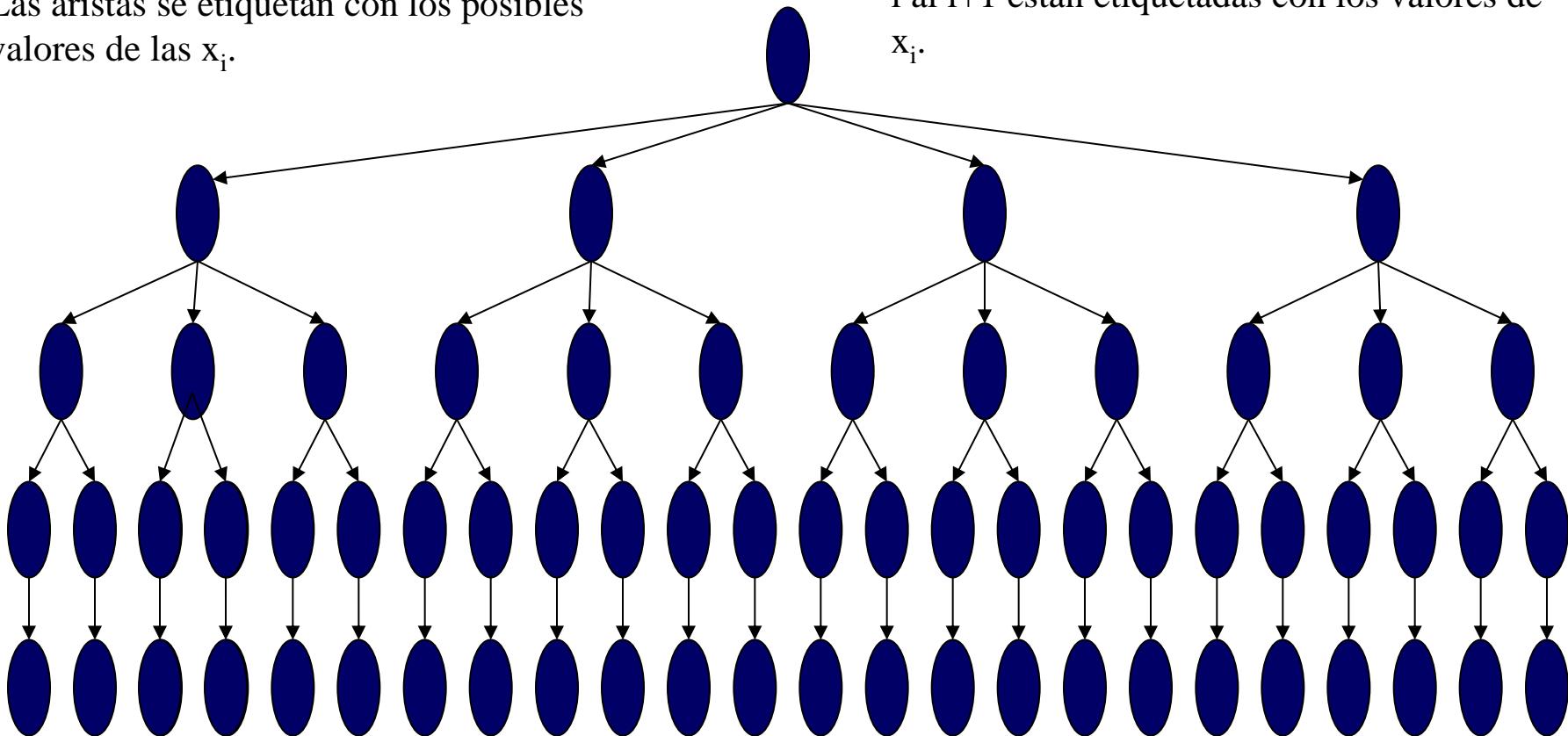
Ejemplo de espacio de soluciones

- La generalización del problema de las 8 reinas es el de las N reinas: Colocar N reinas en un tablero NxN 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,...,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: Árbol de permutación

Las aristas se etiquetan con los posibles valores de las x_i .

Las aristas desde los nodos del nivel i al $i+1$ están etiquetadas con los valores de x_i .



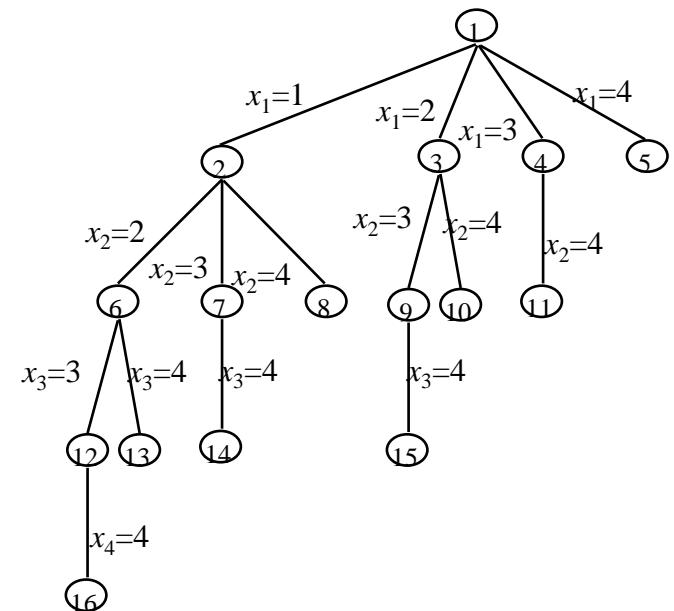
El subárbol de la izquierda contiene todas las soluciones con $x_1 = 1$ y $x_2 = 2$, etc. El espacio de soluciones esta definido por todos los caminos desde el nodo raíz a un nodo hoja. Hay $4! = 24$ nodos hoja en la figura.

Suma de subconjuntos: árbol

- 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 numero de soluciones tiene que ser el mismo

Suma de subconjuntos: árbol 1

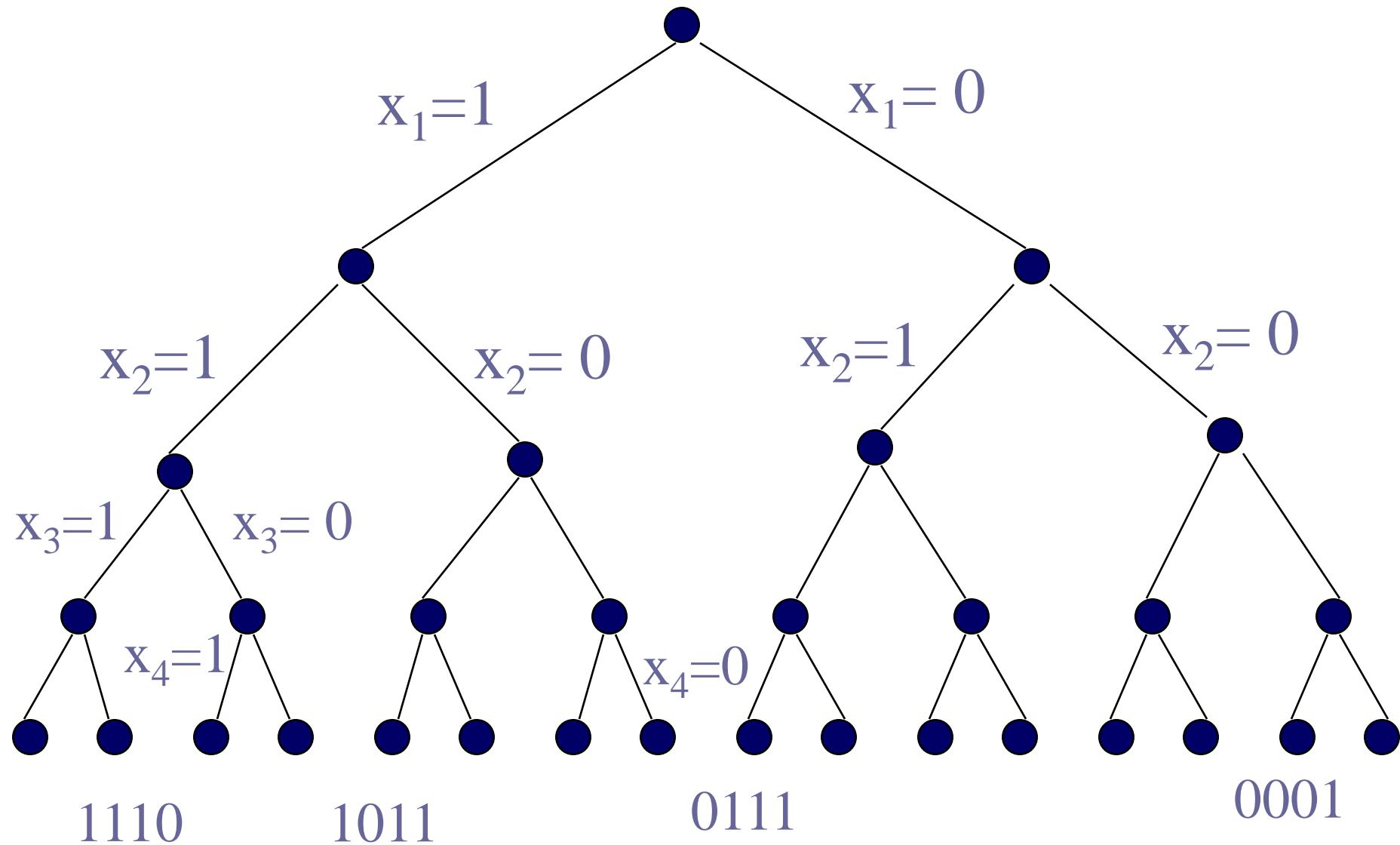
- Las aristas se etiquetan de modo que una desde el nivel de nodos i hasta el $i+1$ representa un valor para x_i .
- En cada nodo, el espacio solución se partitiona en espacios subsolución.
- Los posibles caminos son \emptyset , que corresponde al camino vacío desde la raíz a si misma, (1) , (12) , (123) , (1234) , (124) , (134) , (2) , (23) , etc.
- 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.



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

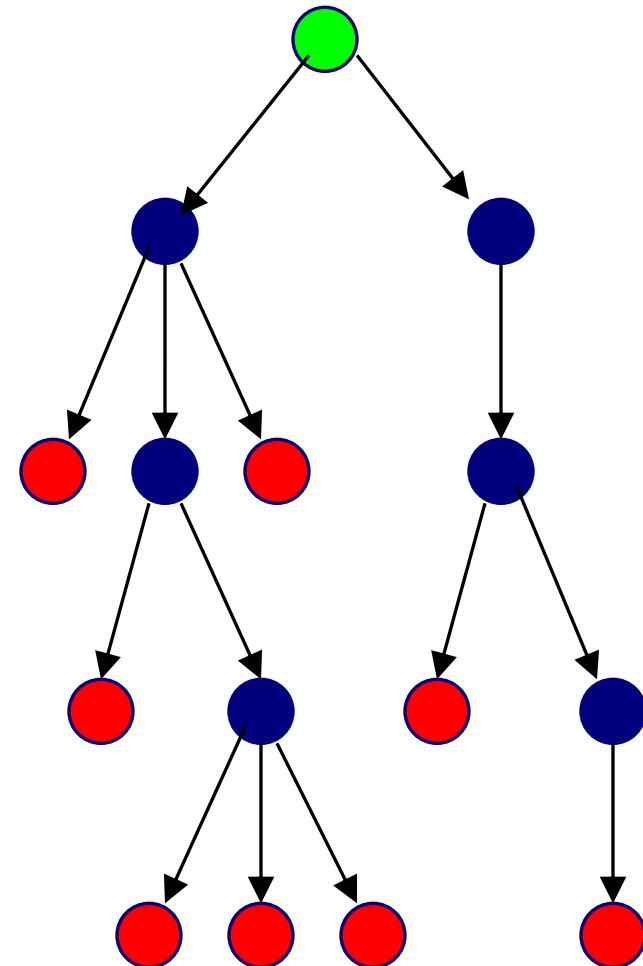
- La organización en árbol del espacio solución se llama **árbol de estados** y en él cada nodo define un **estado** del problema.
- Todos los caminos desde la raíz a otros nodos definen el **espacio de estados** del problema.
- **Estados solución** son aquellos estados del problema S para los que el camino desde la raíz a S define 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 solo los nodos hoja son estados solución.
- **Estados respuesta** son aquellos estados solución S para los que el camino desde la raíz hasta S define una tupla que es miembro del conjunto de soluciones (es decir satisfacen las restricciones implícitas) del problema.

Terminología

- Las organizaciones en árbol del problema de las 8 reinas se llaman **árboles estáticos**, porque son independientes del caso del problema que se vaya a resolver.
- En algunos problemas es ventajoso usar diferentes organizaciones de árbol para diferentes casos del problema.
- Entonces la organización en árbol se determina dinámicamente como el espacio solución que esta siendo buscado.
- Las organizaciones en árbol que son dependientes del caso concreto del problema a resolver se llaman **árboles dinámicos**.

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.



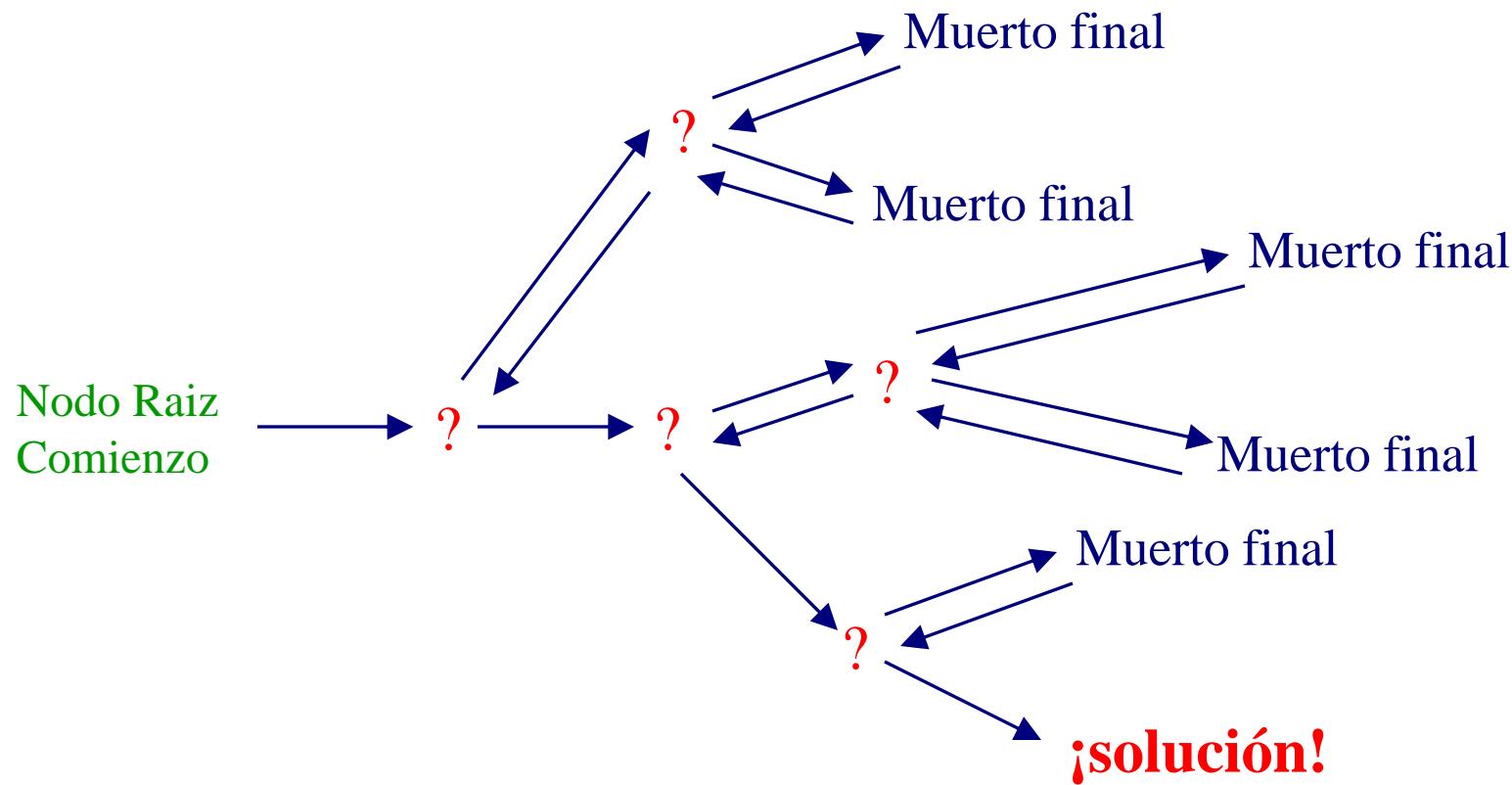
Generación de estados de un problema

- Hay dos formas diferentes de generar los estados del problema.
- Las dos comienzan con el nodo raíz y generan otros nodos.
- Un nodo que ha sido generado, pero para el que no se han generado aun todos sus nodos hijos, se llama un **nodo vivo**.
- El nodo vivo cuyos hijos están siendo generados en ese momento se llama un **E-nodo** (nodo en expansión).
- Un **nodo muerto** es un nodo generado que o no se va a expandir o que todos sus hijos ya han sido generados.
- En ambos métodos de generar estados del problema tendremos una lista de nodos vivos.

Generación de estados de un problema

- 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.
- Esto habrá de hacerse cuidadosamente para que a la conclusión del proceso al menos se haya generado un nodo respuesta, o se hayan generado todos los nodos respuesta si el problema requiriera encontrar todas las soluciones.
- A esta forma de generación (exploración) se le llama **Backtracking**

Como procede el Backtracking



Generación de estados de un problema

- En el segundo método, el E-nodo permanece como E-nodo hasta que se hace nodo muerto
- Como en el otro método, también se usan funciones de acotación para detener la exploración en un sub-árbol.
- 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) mas importante que los mecanismos de exploración en si mismos.
- A esta segunda forma de generación (exploración) se le llama **Branch and Bound**

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 de x_{i+1} tales $(x_1, x_2, \dots, x_{i+1})$ 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

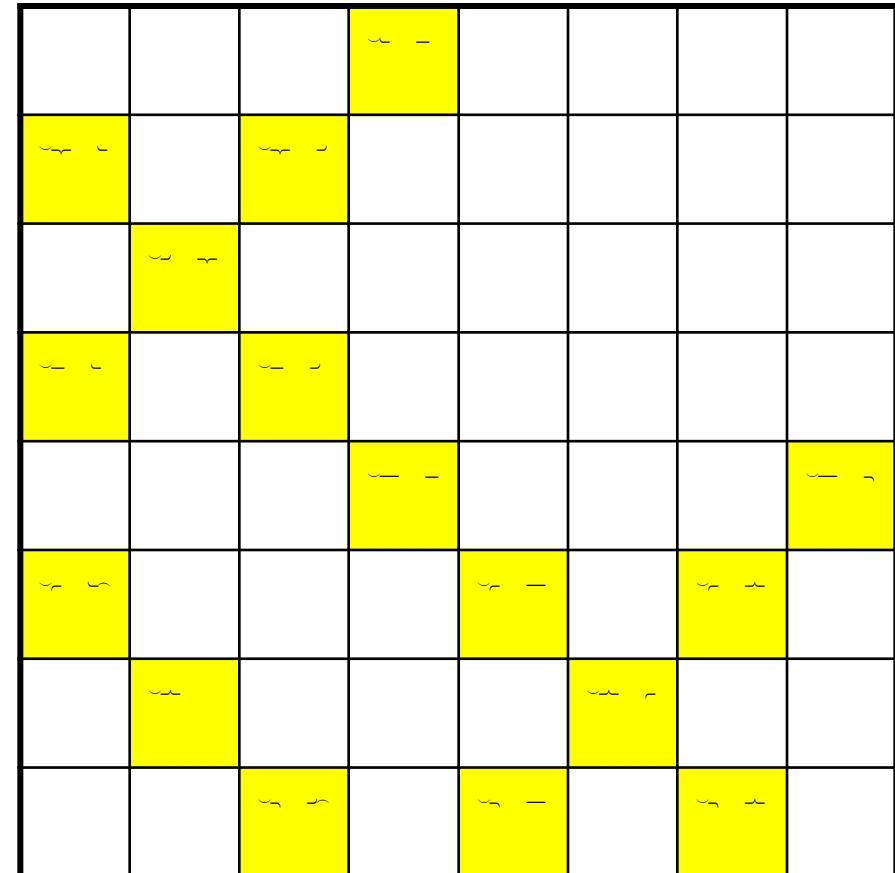
- De los 4 factores que determinan el tiempo requerido por un algoritmo backtracking, solo la cuarta, el número de nodos generados, varía de un caso a otro.
- Un algoritmo backtracking en un caso podría generar solo $O(n)$ nodos, mientras que en otro (relativamente parecido) podría generar casi todos los nodos del árbol de espacio de estados.
- Si el numero de nodos en el espacio solución es 2^n o $n!$, el tiempo del peor caso para el algoritmo backtracking será 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 backtrack en el caso que deseemos resolver.

Eficiencia de backtracking

- Podemos estimar el número de nodos que se generarán 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 está en el nivel i del árbol del espacio de estados.
- Las funciones de acotación se usan en el nodo X para determinar el número 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 's podemos estimar el número total, m , de nodos en el árbol del espacio de estados que no se acotarán.

Solución para las 8 reinas

- Generalizamos el problema para considerar un tablero nxn 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-esima fila en la que la reina i esta colocada.
- Los x_i 's serán todos distintos ya que no puede haber dos reinas en la misma columna.
- ¿Como comprobar que dos reinas no estén en la misma diagonal?



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 solo si,

$$i - j = k - l \text{ ó } i + j = k + l$$

- 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 -esima 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 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 nxn 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) ≤ n and not COLOCA (k) do

{puede moverse esta reina?}

X(k) := X(k) + 1

If X(k) ≤ n

{Se encontró una posición}

Then if k = n

{Es una solución completa?}

Then print (X)

Else k := k + 1; X(k) := 0

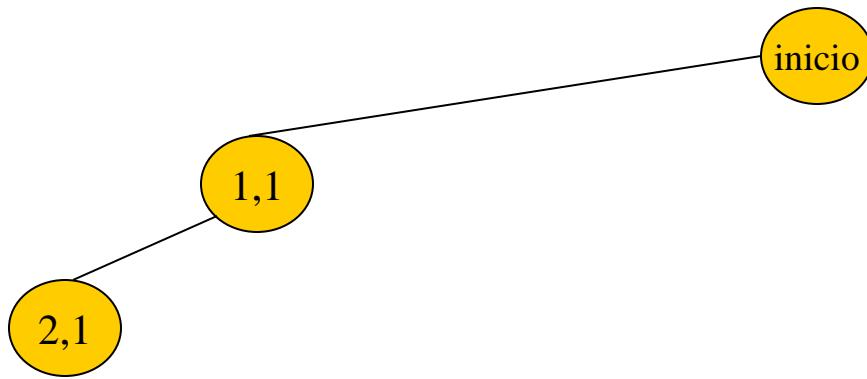
{Ir a la siguiente fila}

Else k := k - 1

{Backtrack}

end

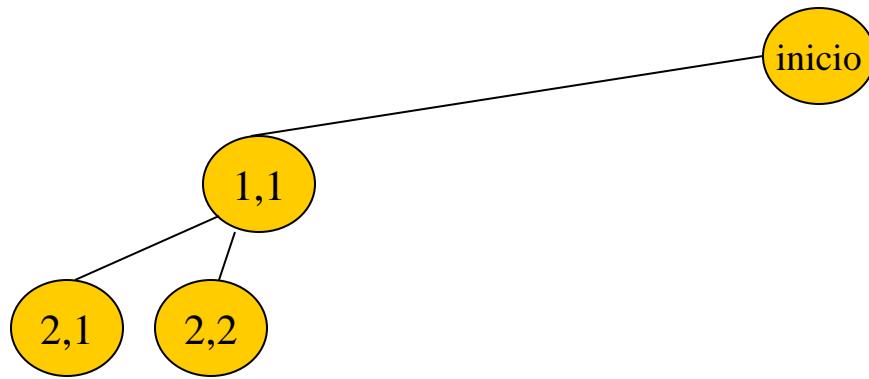
Ejemplo... para $n = 4$



La estrategia ASEGURA
no ocupar el mismo renglón

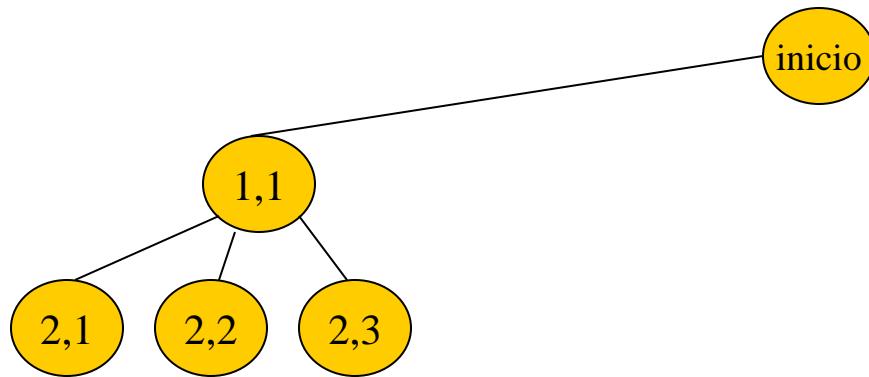
NO se cumple el criterio
(misma columna)

Ejemplo... para $n = 4$



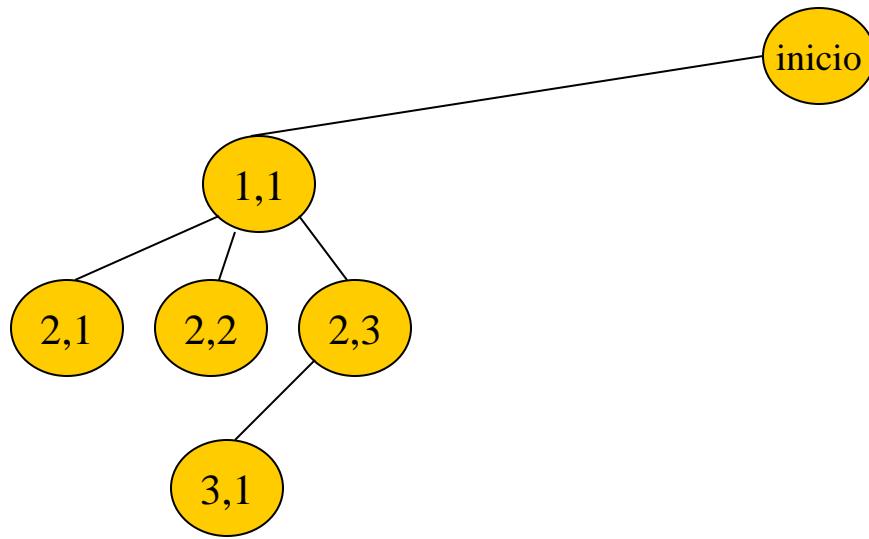
*NO se cumple el criterio
(misma diagonal)*

Ejemplo... para $n = 4$



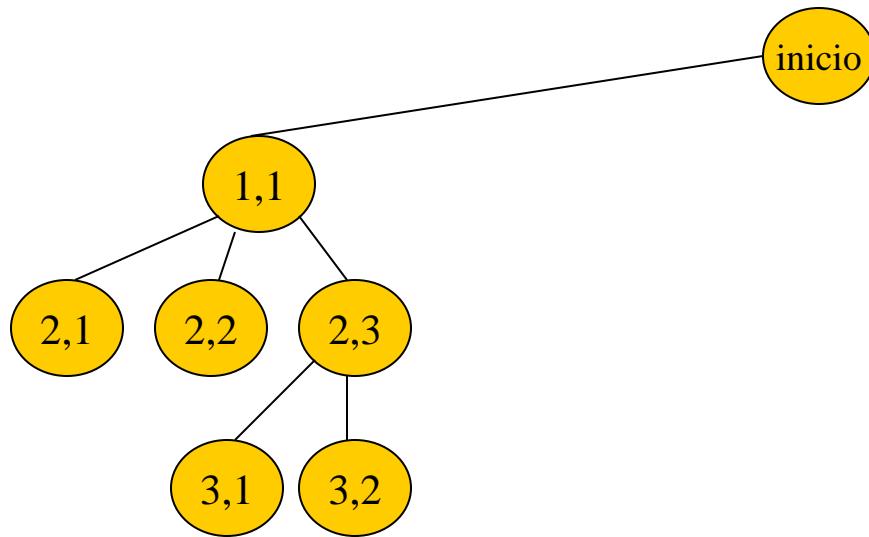
OK... adelante en la
búsqueda !

Ejemplo... para $n = 4$



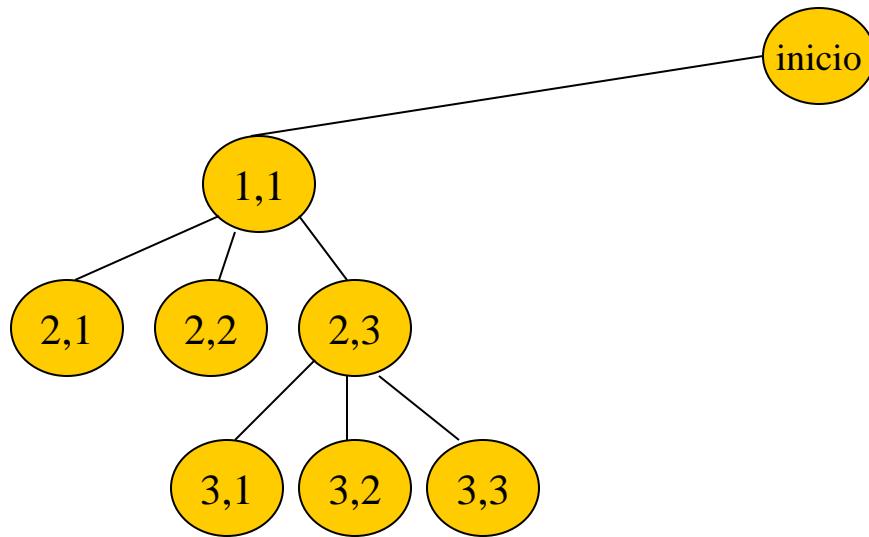
**NO se cumple el criterio
(misma columna)**

Ejemplo... para $n = 4$



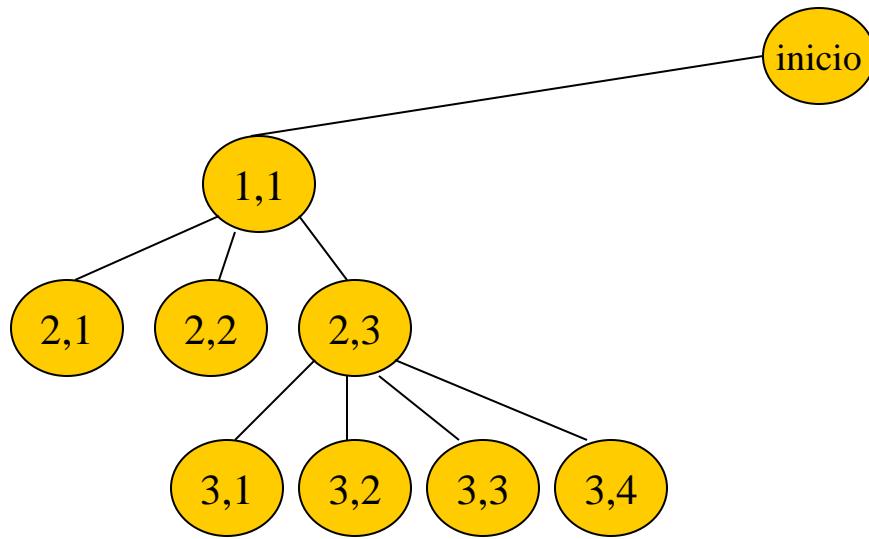
*NO se cumple el criterio
(misma diagonal que 2,3)*

Ejemplo... para $n = 4$



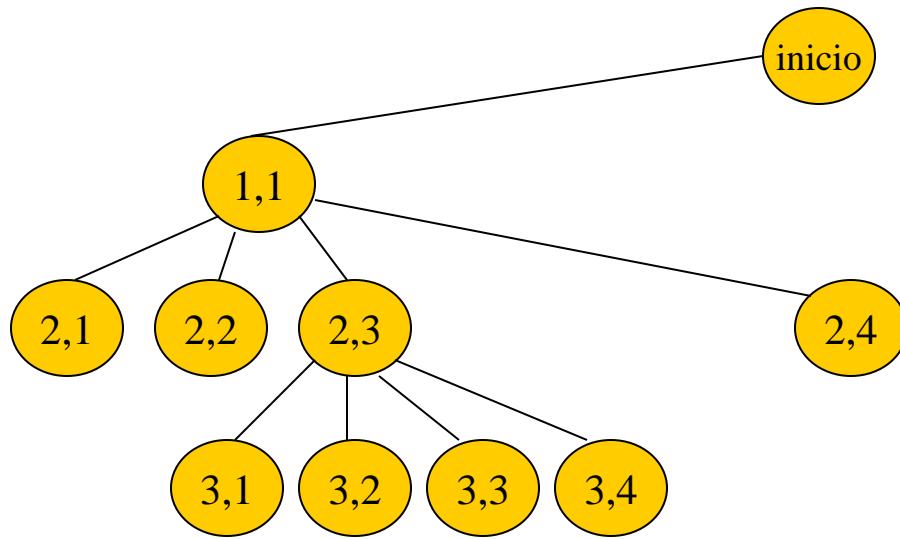
**NO se cumple el criterio
(misma diagonal que 1,1)**

Ejemplo... para $n = 4$



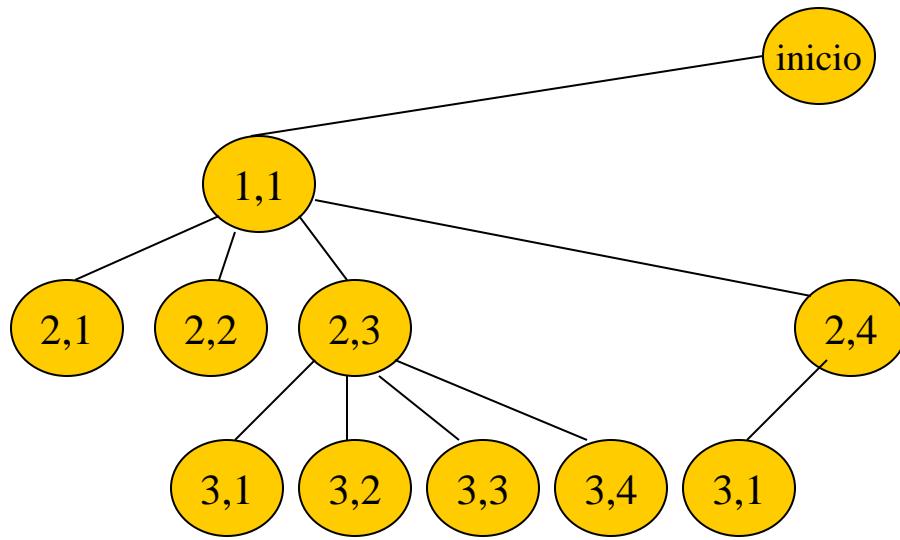
**NO se cumple el criterio
(misma diagonal que 2,3)**

Ejemplo... para $n = 4$



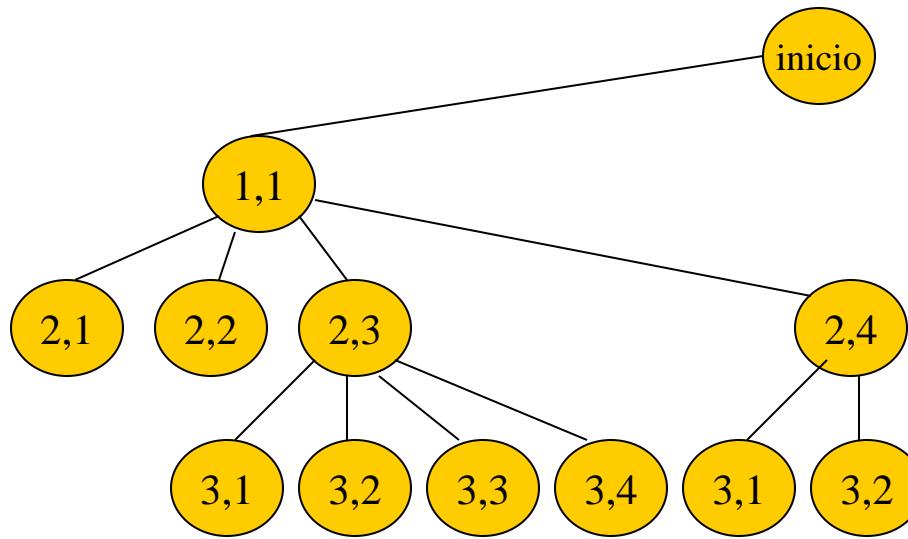
OK... adelante con la
búsqueda!

Ejemplo... para $n = 4$



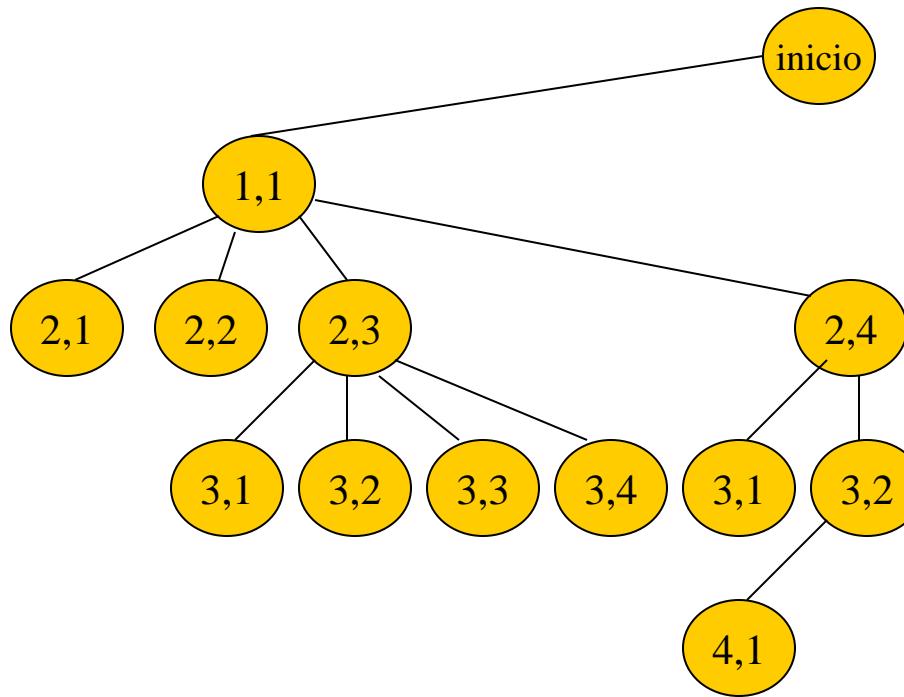
**NO se cumple criterio
(misma columna)**

Ejemplo... para $n = 4$



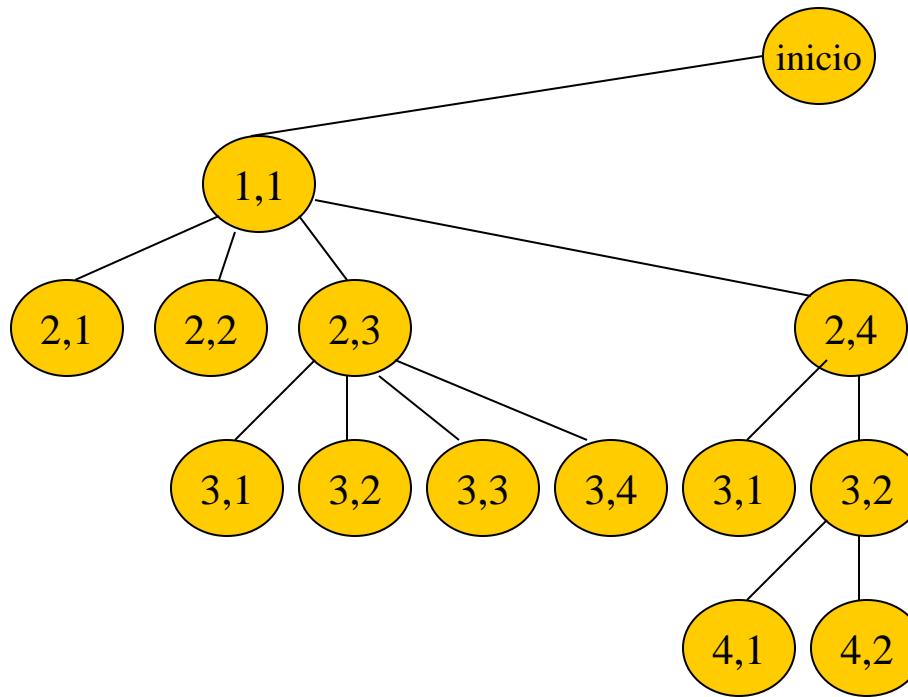
OK... adelante con la
búsqueda!

Ejemplo... para $n = 4$



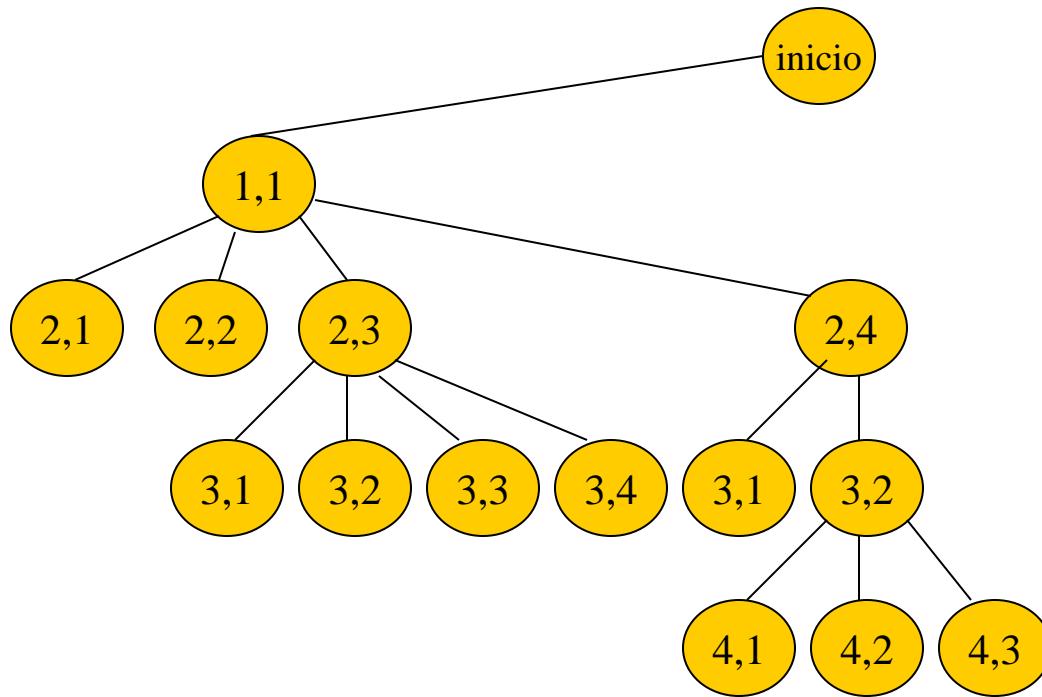
**NO se cumple criterio
(misma columna)**

Ejemplo... para $n = 4$



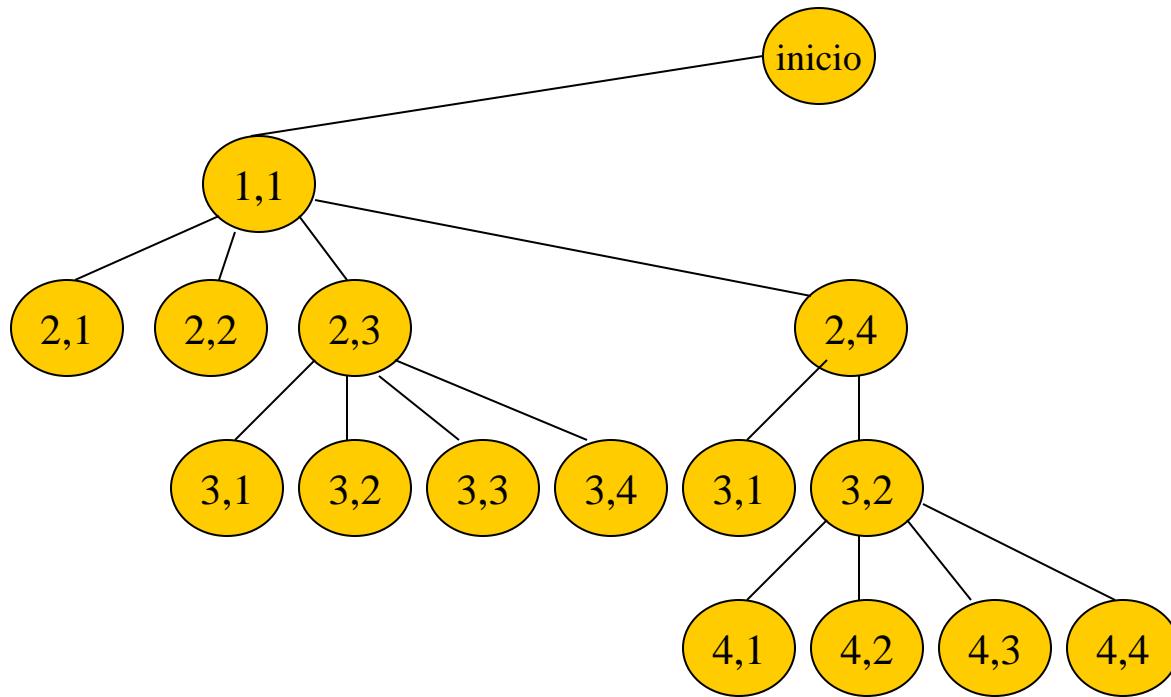
**NO se cumple criterio
(misma columna)**

Ejemplo... para $n = 4$



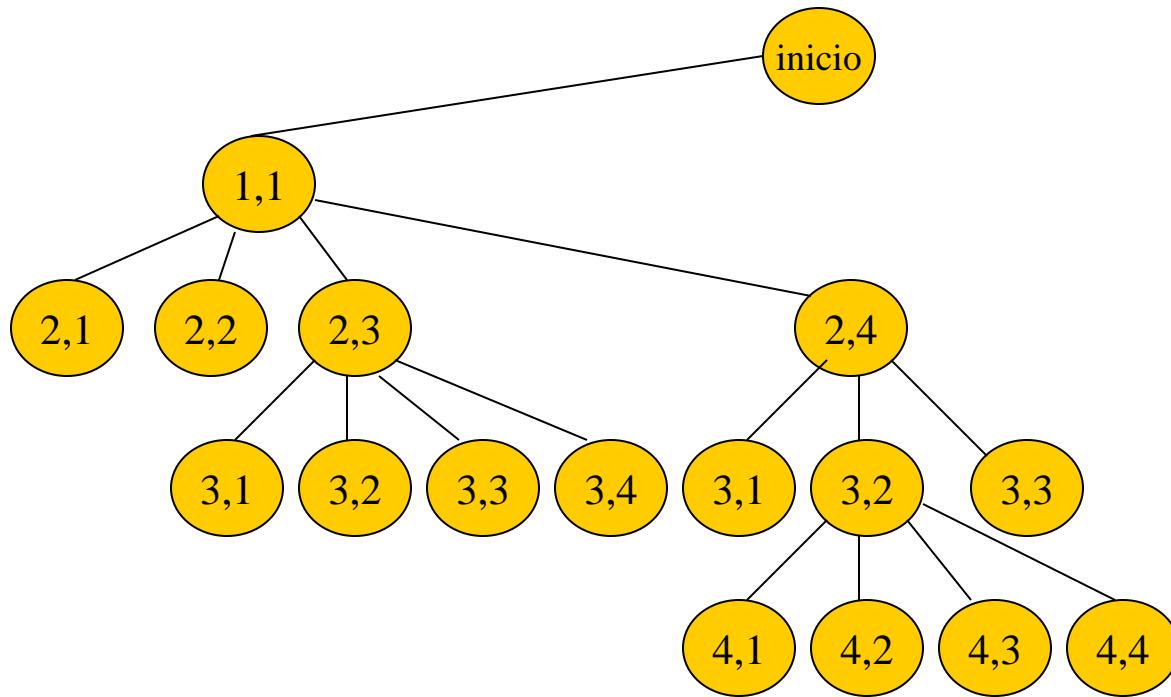
**NO se cumple criterio
(misma diagonal 3,2)**

Ejemplo... para $n = 4$



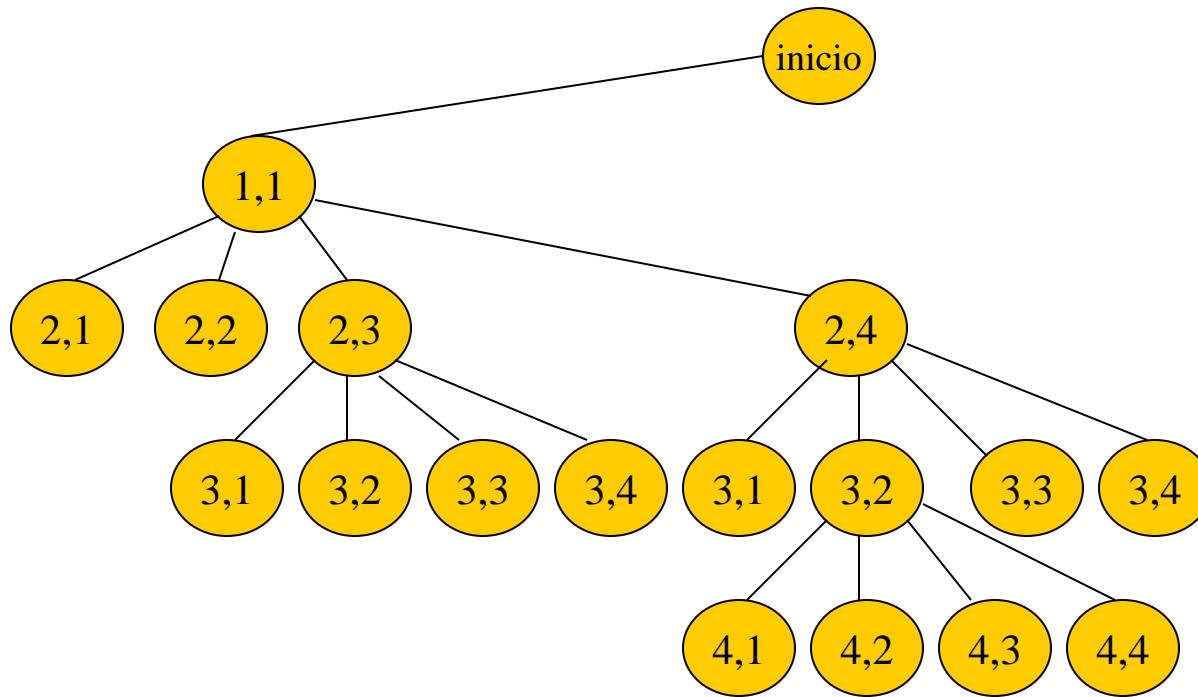
**NO se cumple criterio
(misma columna)**

Ejemplo... para $n = 4$



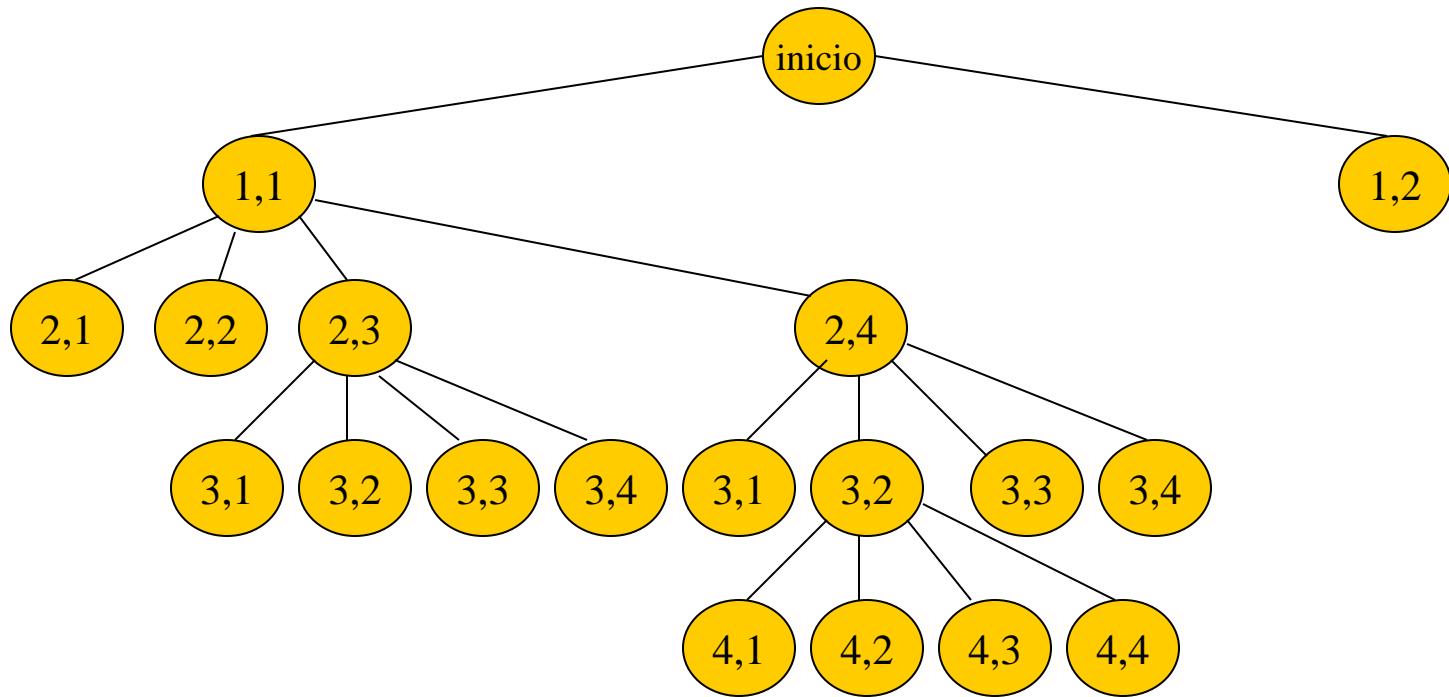
**NO se cumple criterio
(misma diagonal)**

Ejemplo... para $n = 4$



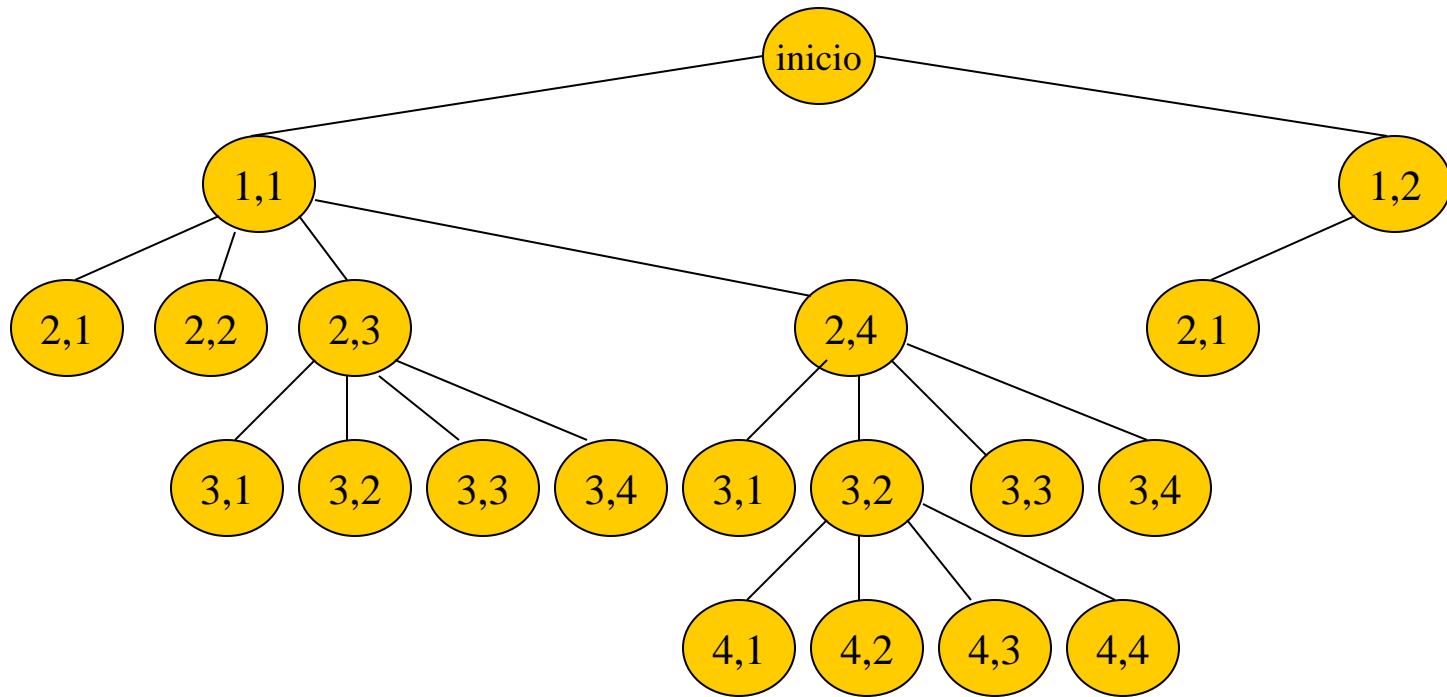
**NO se cumple criterio
(misma columna)**

Ejemplo... para $n = 4$



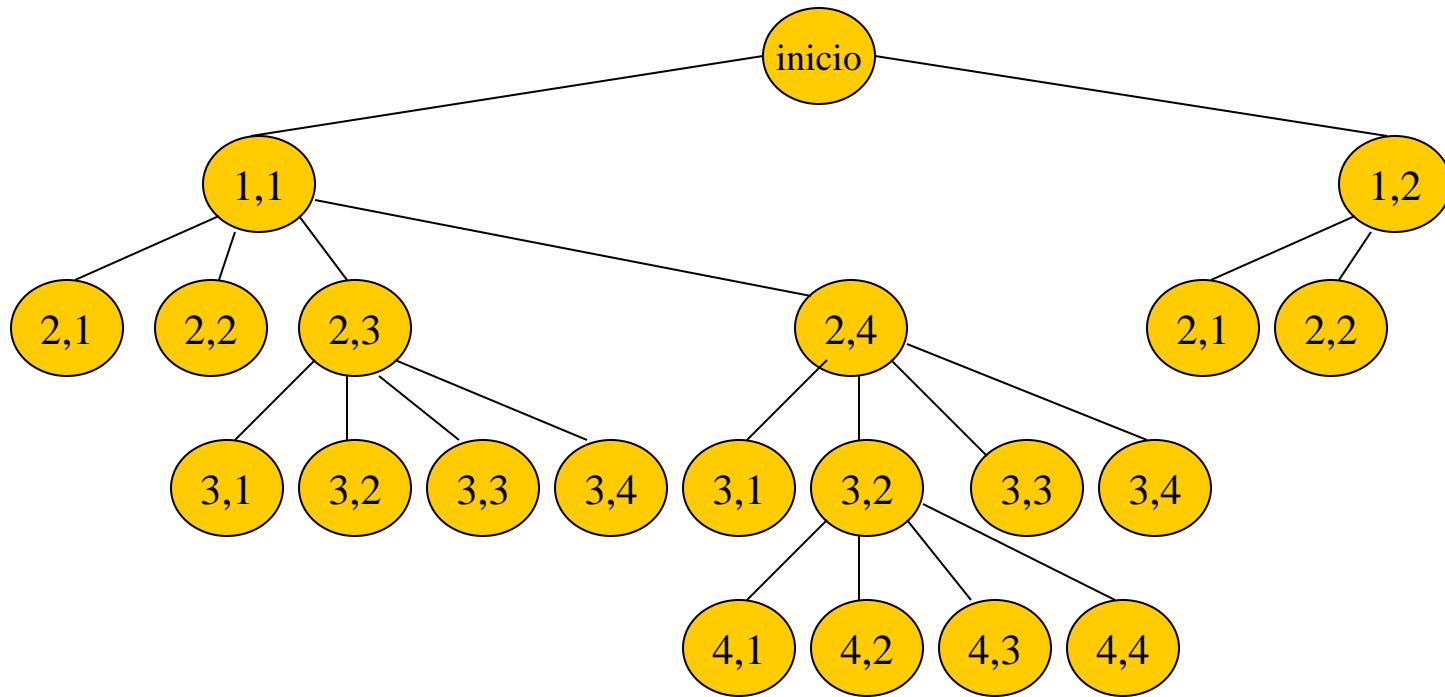
OK... adelante con la
búsqueda!

Ejemplo... para $n = 4$



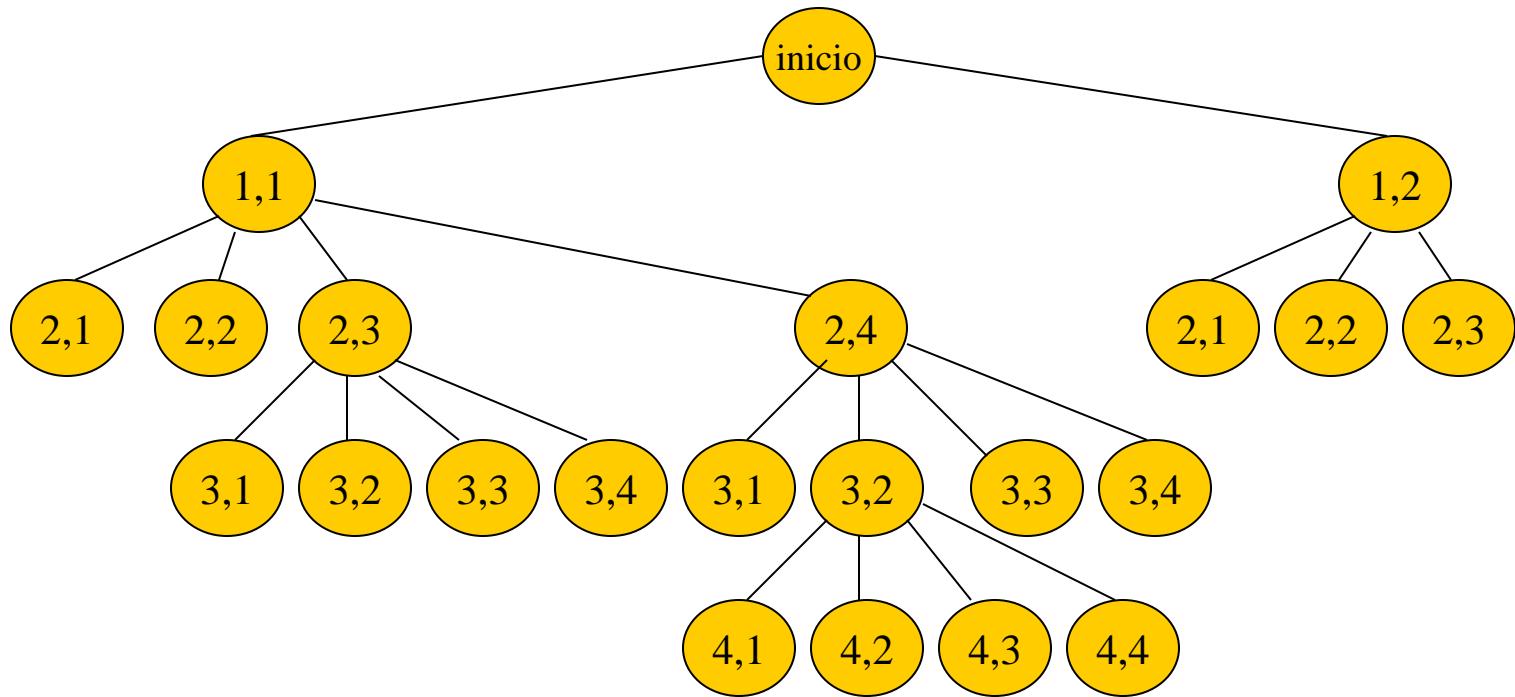
**NO cumple criterio
(misma diagonal)**

Ejemplo... para $n = 4$



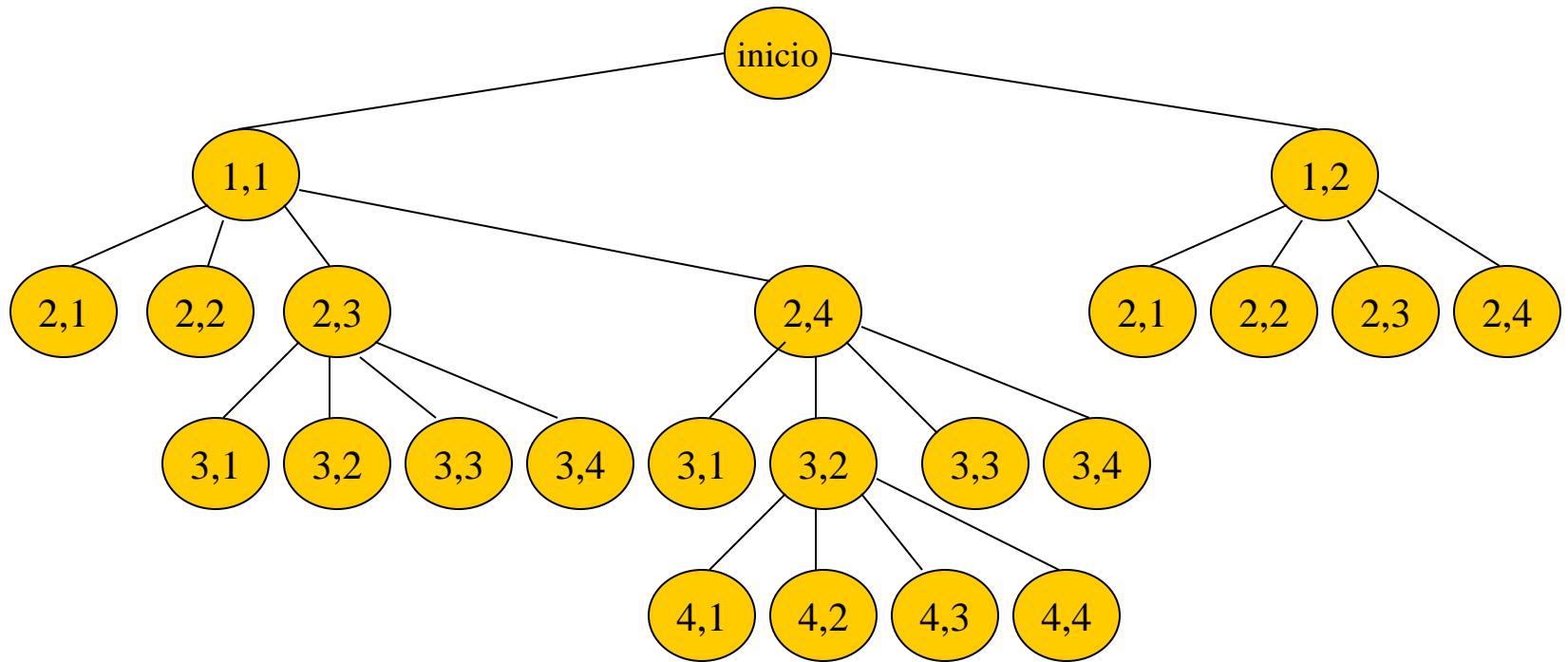
**NO cumple criterio
(misma columna)**

Ejemplo... para $n = 4$



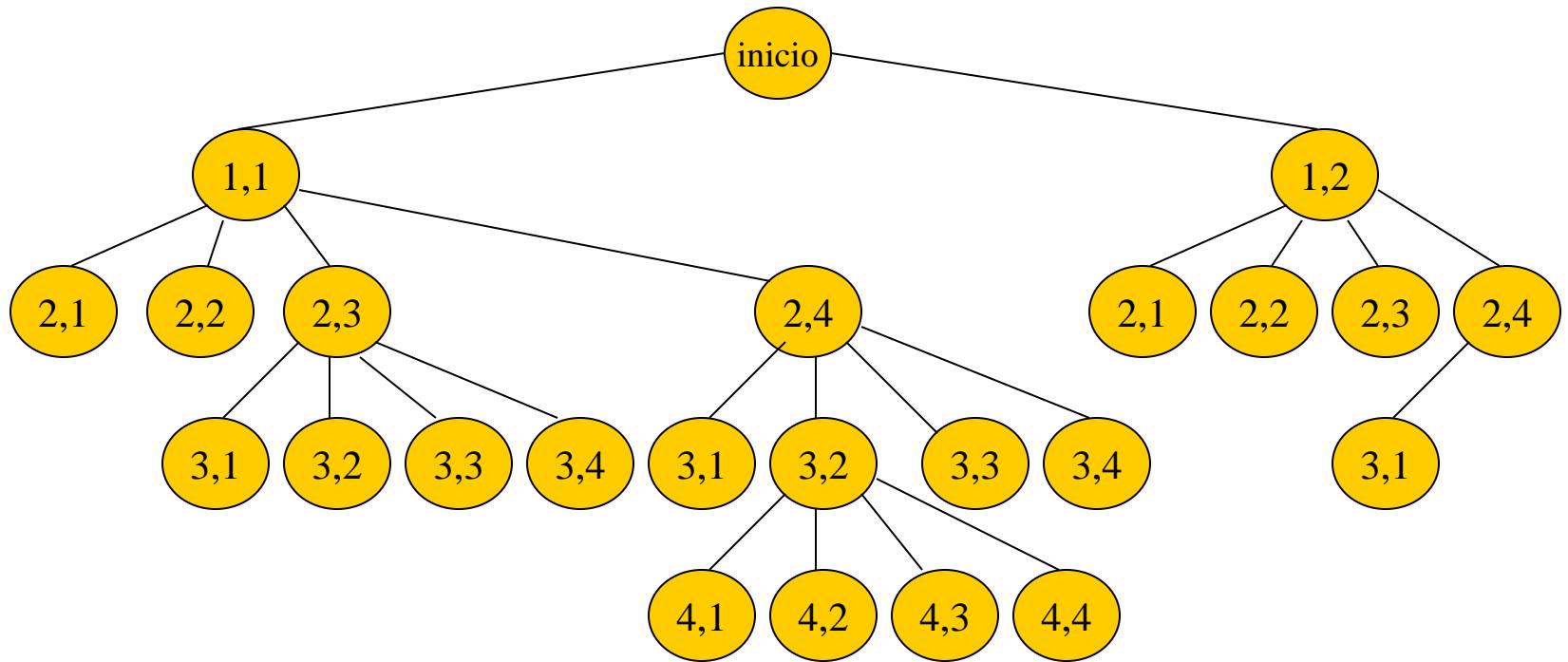
**NO cumple criterio
(misma diagonal)**

Ejemplo... para $n = 4$



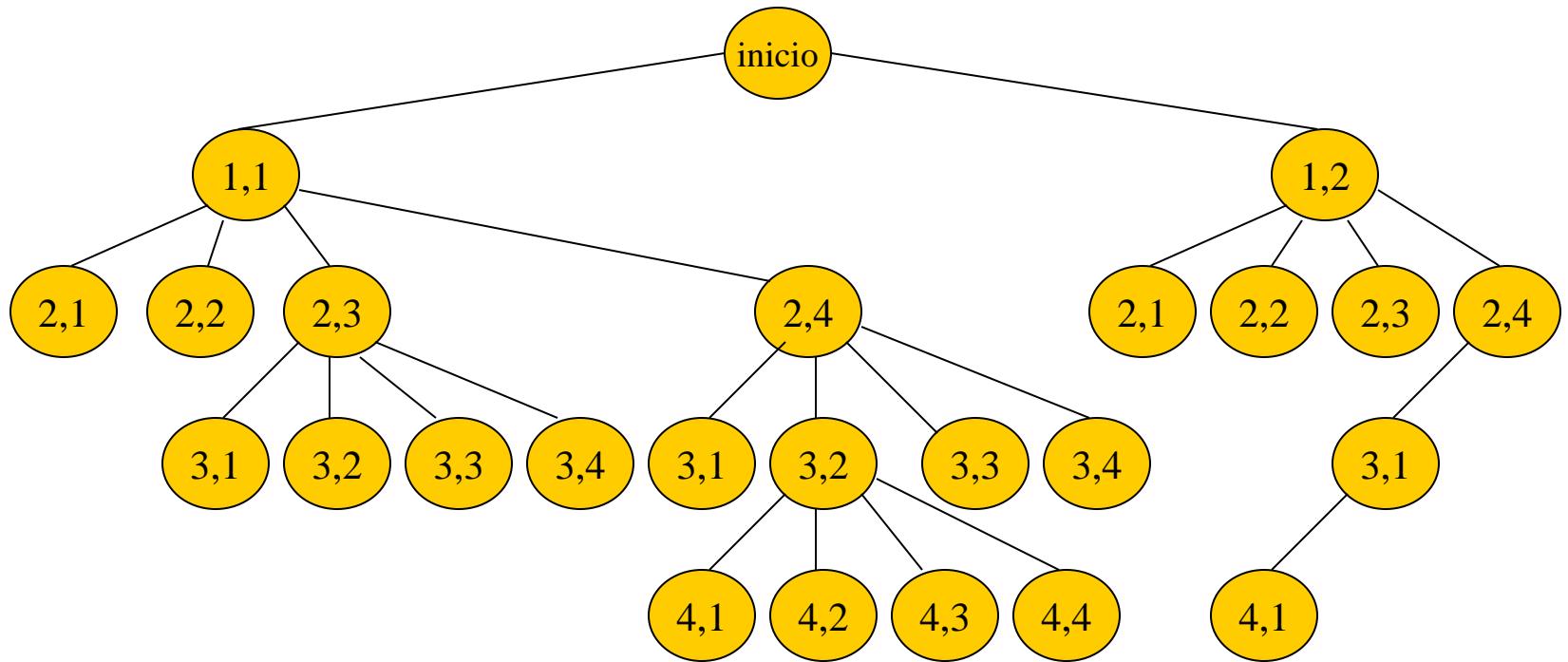
OK... adelante con la
búsqueda!

Ejemplo... para $n = 4$



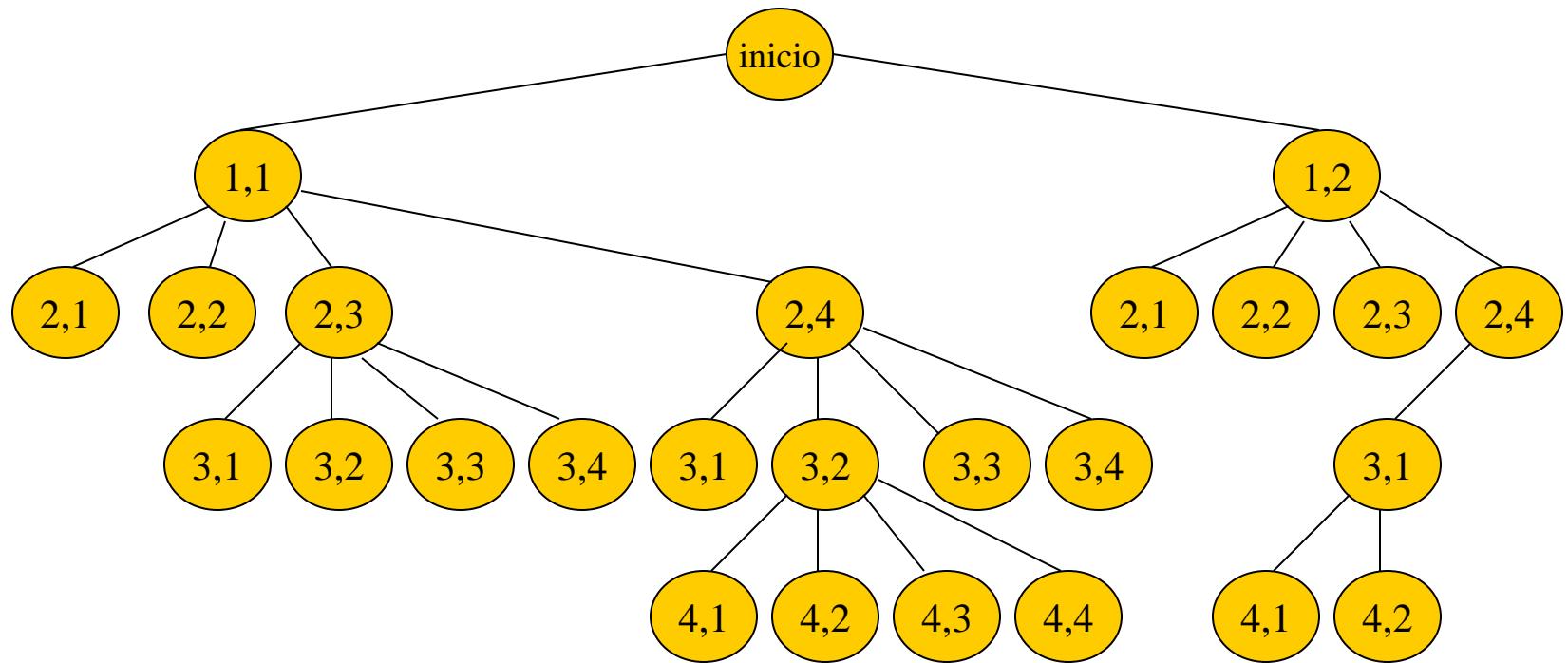
OK... adelante con la
búsqueda!

Ejemplo... para $n = 4$



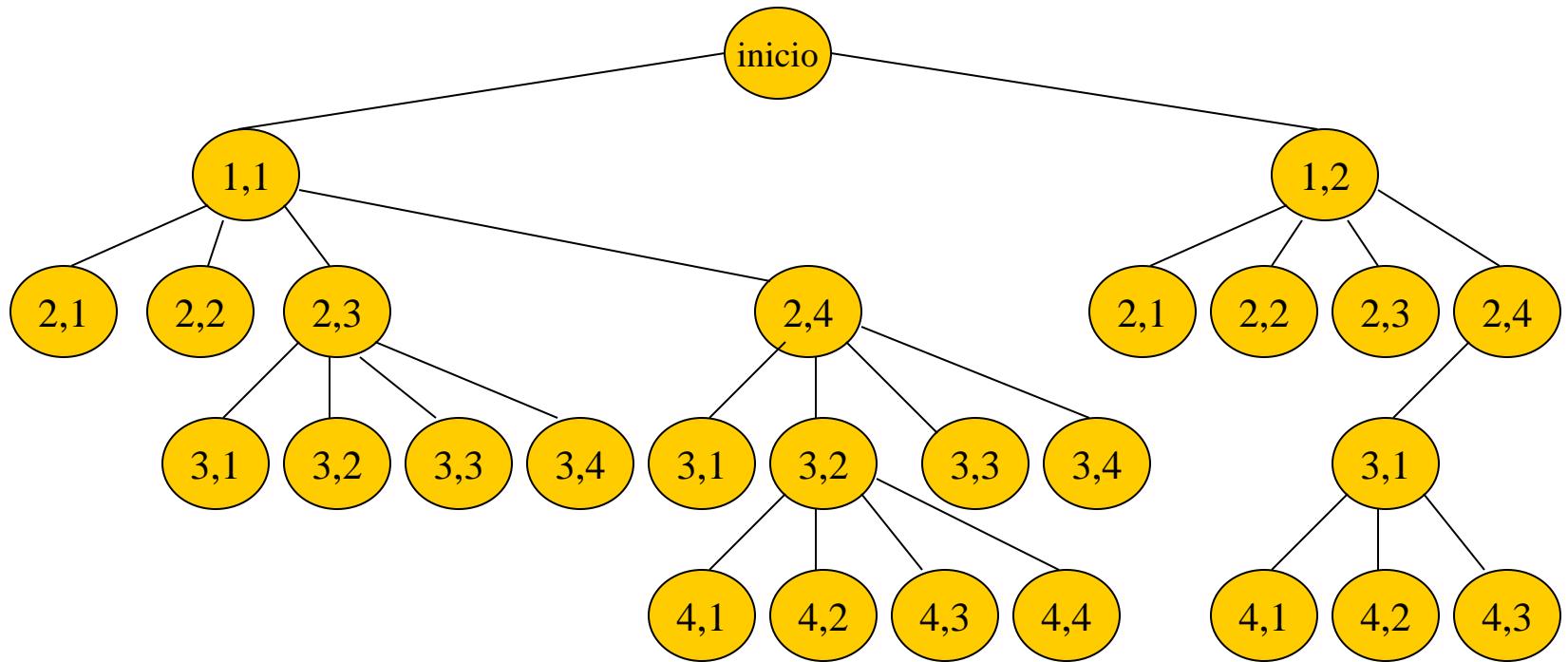
**NO cumple el criterio
(misma columna)**

Ejemplo... para $n = 4$



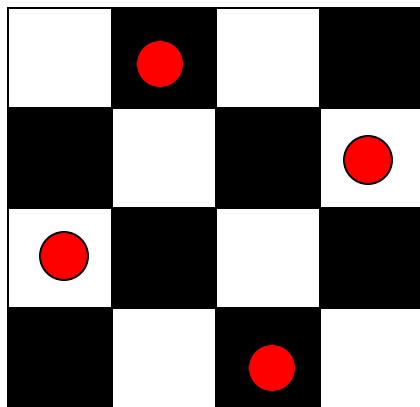
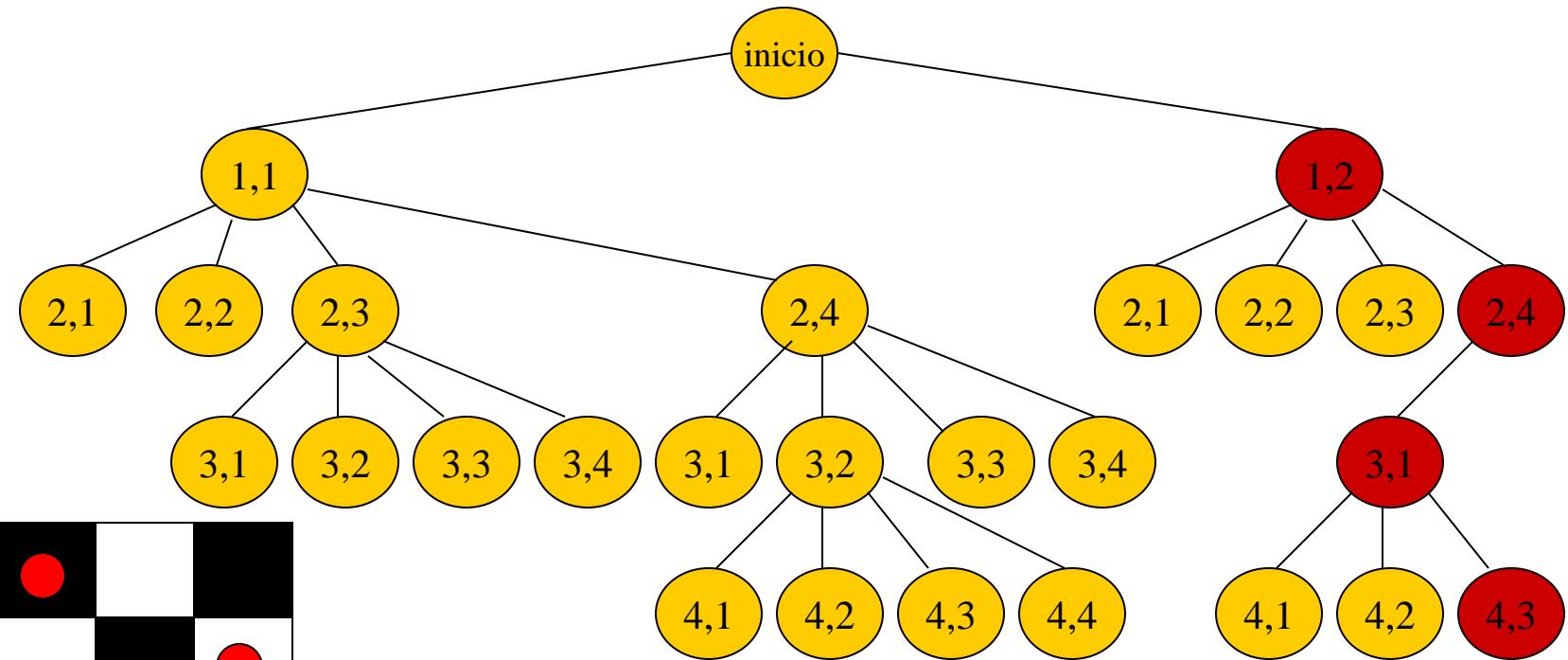
**NO cumple el criterio
(misma columna)**

Ejemplo... para $n = 4$



¡¡OK... se encontró solución !!

Ejemplo... para $n = 4$



Se comprobaron 27 nodos... Sin backtracking se hubieran comprobado 155 nodos

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 nos 4 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 solo 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 $B_{k-1} = \text{true}$ }

$X(k) = 1$

{4} If $s + W(k) = M$

{5} Then For $i = 1$ to k print $X(j)$
 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 $s + W(k+1) \leq M$

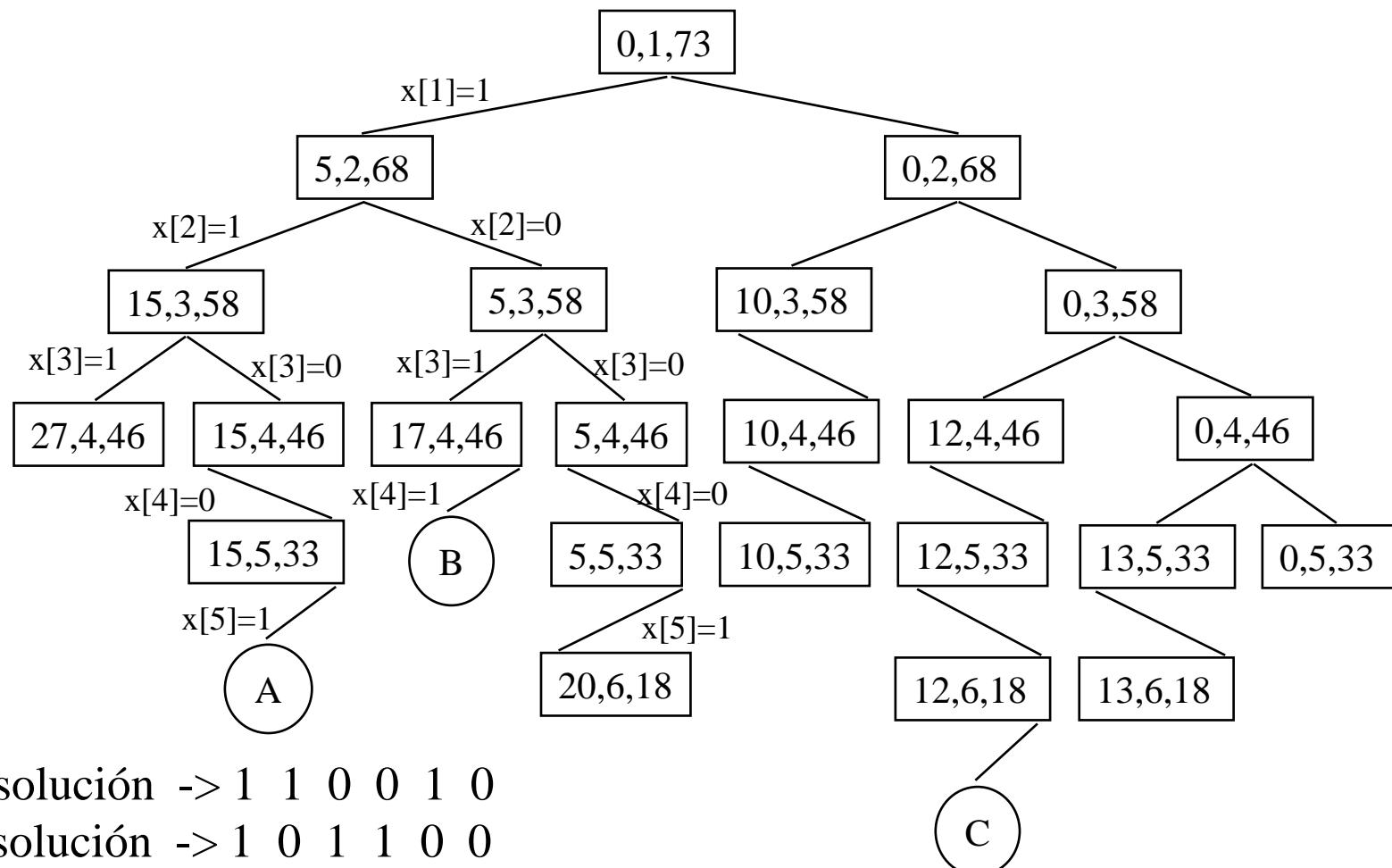
 Then $X(k) = 0$

 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 $M = 30$.



A, 1^a solución -> 1 1 0 0 1 0

B, 2^a solución -> 1 0 1 1 0 0

C, 3^a solución -> 0 0 1 0 0 1

Ciclos hamiltonianos

- 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 se visita dos veces.
- Estamos interesados en construir un algoritmo backtracking que determine todos los ciclos Hamiltonianos de G , que puede ser dirigido o no.
- El vector backtracking solución (x_1, \dots, x_n) se define de modo que x_i represente el i -esimo vértice visitado en el ciclo propuesto.

Ciclos hamiltonianos

- 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 este conectado por una arista a $X(k-1)$.
- $X(n)$ solo puede ser el único vértice restante y debe estar conectado a $X(n-1)$ y a $X(1)$.

Ciclos hamiltonianos

Algoritmo Hamiltoniano (k)

while

$x[k] = \text{SiguienteValor}(k)$

 if ($x[k] = 0$) then return

 if ($k = N$) then print solucion

 else Hamiltoniano (k+1)

endWhile

Utilizando este algoritmo podemos particularizar el esquema backtracking recursivo que vimos para encontrar todos los ciclos hamiltonianos

Algoritmo SiguienteValor (k)

while

 value = $(x[k]+1) \bmod (N+1)$

 if (value = 0) then return value

 if ($G[x[k-1], value]$)

 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



Algorítmica

Capítulo 5: Algoritmos para la Exploración de Grafos.

Tema 15: El método Branch and Bound*

■ Método general Branch and Bound

- Resolución del Problema de la Mochila.
- Resolución del Problema del Viajante de Comercio.
- Descripción del Problema de los 15

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 (Programación Matemática) y para jugar juegos
- Lo crearon A.H. Land y A.G. Doig en 1960, pero el nombre se lo dieron Little, Murty, Sweeney y Karel. Fue R. J. Dakin quien le dió la versión actual en 1965

ECONOMETRICA
VOLUME 28 July, 1960 NUMBER 3

AN AUTOMATIC METHOD OF SOLVING DISCRETE
PROGRAMMING PROBLEMS

BY A. H. LAND AND A. G. DOIG

In the classical linear programming problem the behaviour of continuous, nonnegative variables subject to a system of linear inequalities is investigated. One possible generalization of this problem is to relax the continuity condition on the variables. This paper presents a simple numerical algorithm for the solution of programming problems in which some or all of the variables can take only discrete values. The algorithm requires no special techniques beyond those used in ordinary linear programming, and lends itself to automatic computing. Its use is illustrated on two numerical examples.

I. INTRODUCTION

THERE IS A growing literature [1, 3, 5, 6] about optimization problems which could be formulated as linear programming problems with additional constraints that some or all of the variables may take only integral values. This form of linear programming arises whenever there are indivisibilities. It is not meaningful, for instance, to schedule 3–7/10 flights between two cities, or to undertake only 1/4 of the necessary setting up operation for running a job through a machine shop. Yet it is basic to linear programming that the variables are free to take on any positive value,¹ and this sort of answer is very likely to turn up.

In some cases, notably those which can be expressed as transport problems, the linear programming solution will itself yield discrete values of the variables. In other cases the percentage change in the maximand² from common sense rounding of the variables is sufficiently small to be neglected. But there remain many problems where the discrete variable constraints are significant and costly.

Until recently there was no general automatic routine for solving such problems, as opposed to procedures for proving the optimality of conjectured solutions, and the work reported here is intended to fill the gap. About the time of its completion an alternative method was proposed by Gomory [5] and subsequently extended by Beale [1]. Gomory's method

¹ Or more generally, any value within a bounded interval.

² We shall speak throughout of maximisation, but of course an exactly analogous argument applies to minimisation.

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 bactracking).
- Puede ser vista como una **generalización** (o mejora) de la técnica de Backtracking.
- La principal novedad es que habrá una **estrategia de ramificación**.
- Se tratará como un aspecto importante las **técnicas de poda**, para eliminar nodos que no lleven a soluciones óptimas.
- 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)

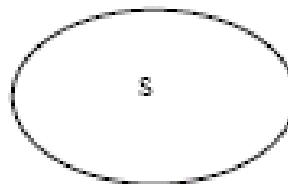
■ En consecuencia:

- En Backtracking los únicos nodos vivos son los que están en el camino de la raíz al nodo en curso.
- En BB puede haber más nodos vivos (se almacenan en una estructura de datos auxiliar: **lista de nodos vivos (LNV)**).

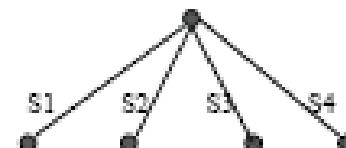
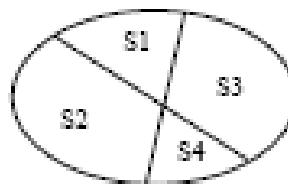
■ Además

- 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

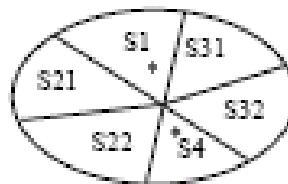
Branch and bound



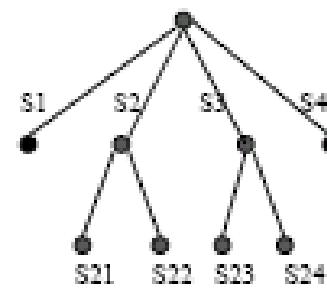
(a)



(b)



(c)



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 raíz (el problema original con su región factible completa)
- Entonces se aplican procedimientos de cotas inferiores y superiores al problema raíz.
- Si las cotas cumplen las condiciones que se hayan establecido, habremos encontrado la solución óptima y el procedimiento termina.
- Si ese no fuera el caso, entonces la región factible se divide en dos o más regiones, dando lugar a distintos subproblemas.
- Esos subproblemas partitionan la región factible. La búsqueda se desarrolla en cada una de esas regiones.
- El método (algoritmo) se aplica recursivamente a los subproblemas.

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 superior es menor 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 superiores de los problemas no resueltos

Branch and bound

■ Para cada nodo i tendremos:

- Una **Cota superior (CS(i))** y una **Cota inferior (CI(i))** del beneficio (o coste) óptimo que podemos alcanzar a partir de ese nodo.
 - Determinan cuándo se puede realizar una poda.
- Una **Estimación del Beneficio (o coste)** óptimo que se puede encontrar a partir de ese nodo. Puede ser una media de las anteriores.
 - Ayuda a decidir qué parte del árbol evaluar primero.
- Una **Estrategia de Poda**
 - Suponemos un problema de maximización
 - Hemos recorrido varios nodos 1..n, estimando para cada uno la cota superior **CS (j)** e inferior **CI (j)**, respectivamente, para j entre 1 y n.
 - Hay dos casos

Branch and bound

- **CASO 1.** Si a partir de un nodo siempre podemos obtener alguna solución válida, entonces **podar un nodo i si:**

$CS(i) \leq CI(j)$, para algún nodo j generado.

- **Ejemplo:** problema de la mochila, utilizando un árbol binario.
 - A partir de **a**, podemos encontrar un beneficio máximo de $CS(a)=4$.
 - A partir de **b**, tenemos garantizado un beneficio mínimo de $CI(b)=5$.
 - Podemos podar el nodo **a**, sin perder ninguna solución óptima.
- **CASO 2.** Si a partir de un nodo puede que no lleguemos a una solución válida, entonces **podar un nodo i si:**

$CS(i) \leq \text{Beneficio}(j)$, para algún j, solución final (factible).
- **Ejemplo:** problema de las 8 reinas. A partir de una solución parcial, no está garantizado que exista una solución

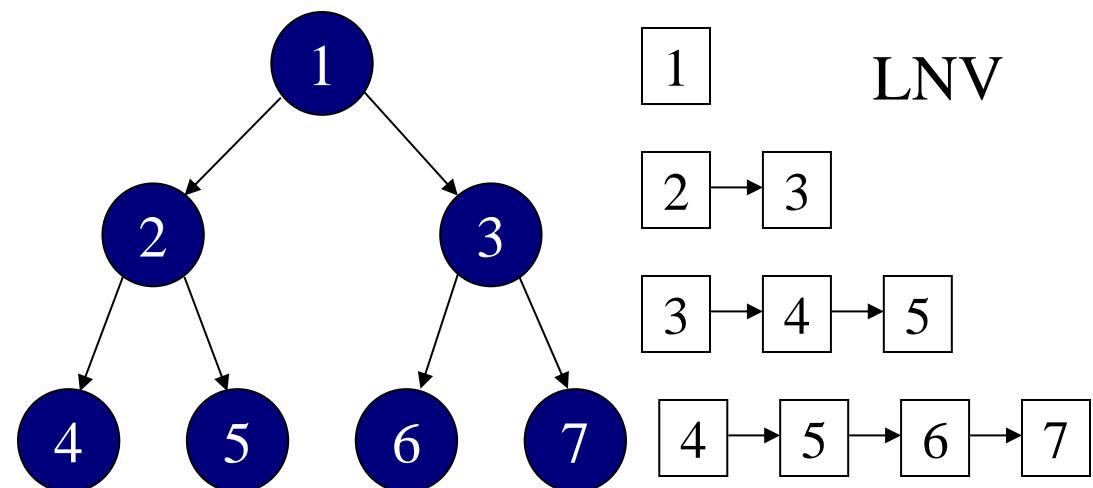
Branch and bound

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 (LNV)**
- La 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.
- 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

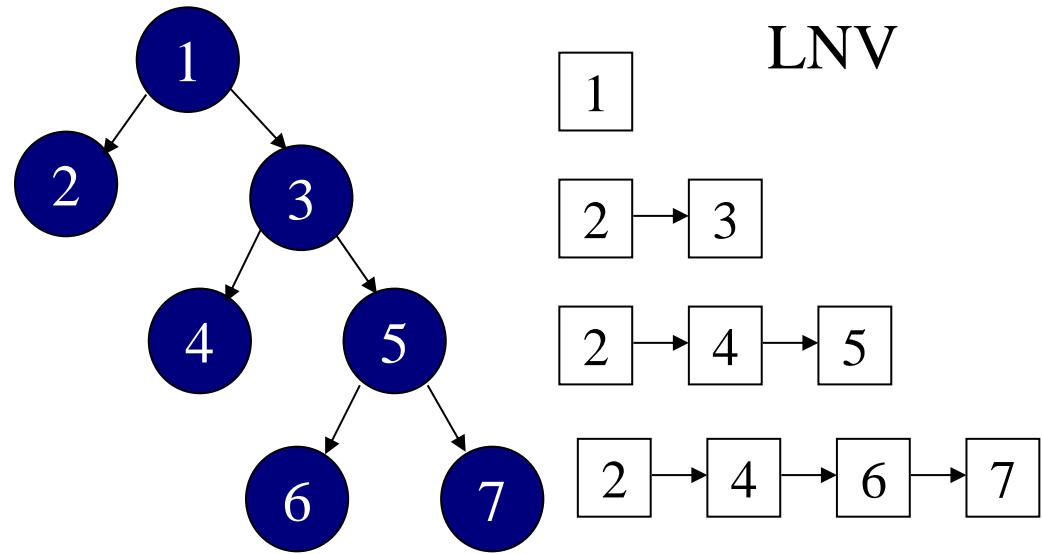


Branch and bound

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.

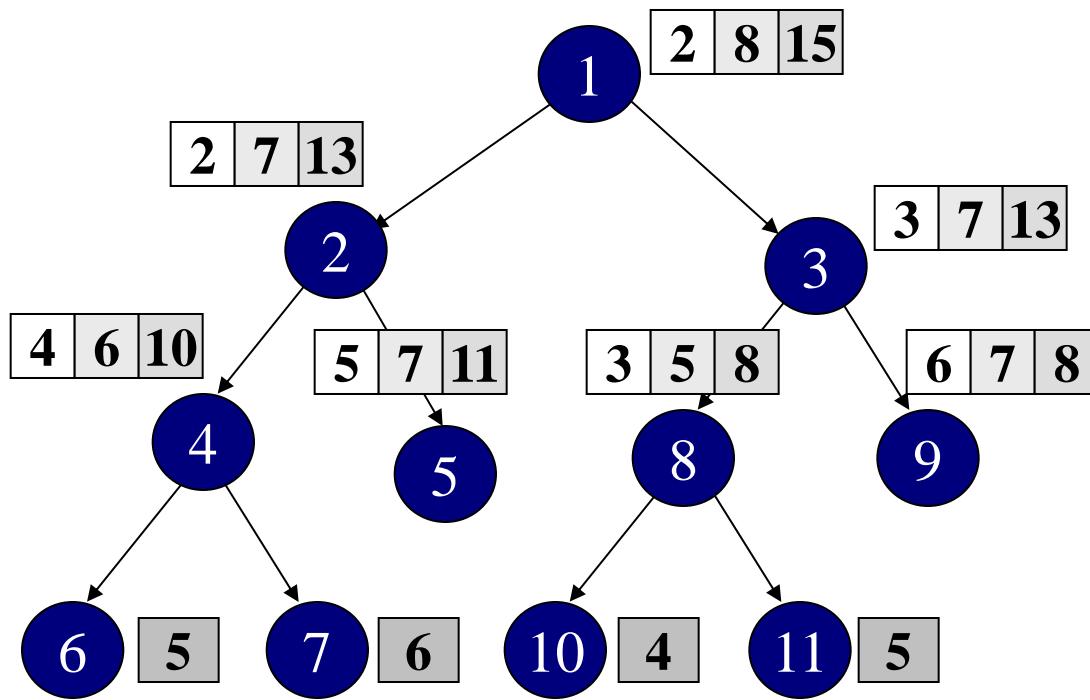
Branch and bound

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 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** = 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 .



C	LNV
15	[1]
13	[2] → [3]
10	[4] → [3] → [5]
5	[3] → [5]
5	[8] → [5]
4	[5]

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, o si tenemos el caso 2 (a partir de un nodo puede que no exista ninguna solución)?
- ¿Cómo será la poda, para cada uno de los casos anteriores?
- ¿Qué pasa si para un nodo **i** tenemos que $CI(i) = CS(i)$?
- ¿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:= CS (NodoRaiz);

 s:= \emptyset ;

 Mientras LNV $\neq \emptyset$ 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:= min (C, **Coste** (y));

 Sino si y no es solución final y (**CI**(y) < C) entonces

 LNV:= LNV + {y};

 C:= min (C, **CS** (y));

 FinPara;

 FinMientras;

BB: Análisis de los 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 lleva la LNV)
- ¿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 de calcularlas.

Ejemplo, 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) :
 $(s_1, \dots, s_k, s_k+1), (s_1, \dots, s_k, s_k+2), \dots, (s_1, \dots, s_k, n)$

Ejemplo, El Problema de la Mochila 0/1

■ Cálculo de cotas:

- **Cota inferior:** Beneficio que se obtendría incluyendo solo los objetos incluidos hasta ese nodo.
- Estimación del beneficio:** A la solución actual, sumar el beneficio de incluir los objetos enteros que quepan, utilizando greedy. 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: $CS = CI + \lfloor \text{MochilaGreedy}(3, 4) \rfloor = 5 + \lfloor 4 + 5/4 \rfloor = 10$

Ejemplo, 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**)

Ejemplo, El Problema de la Mochila 0/1

```
Mochila01RyP (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;
```

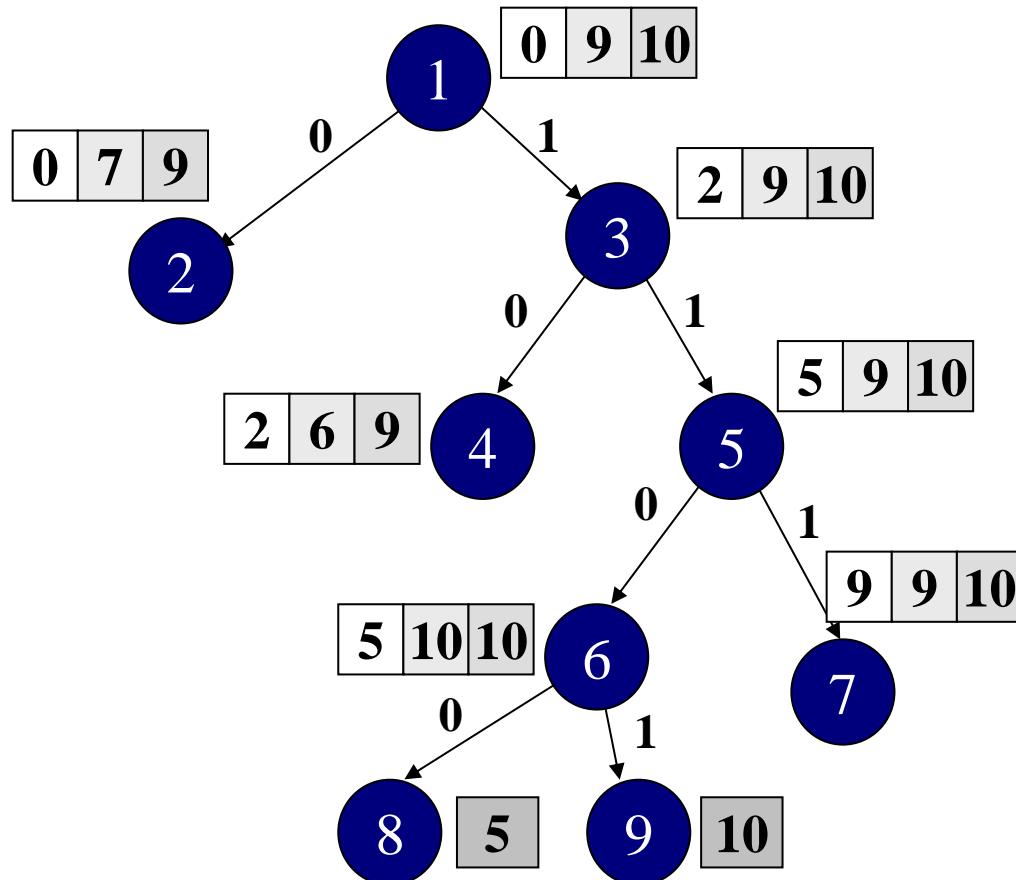
Ejemplo, El Problema de la Mochila 0/1

```
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;
```

Ejemplo, El Problema de la Mochila 0/1

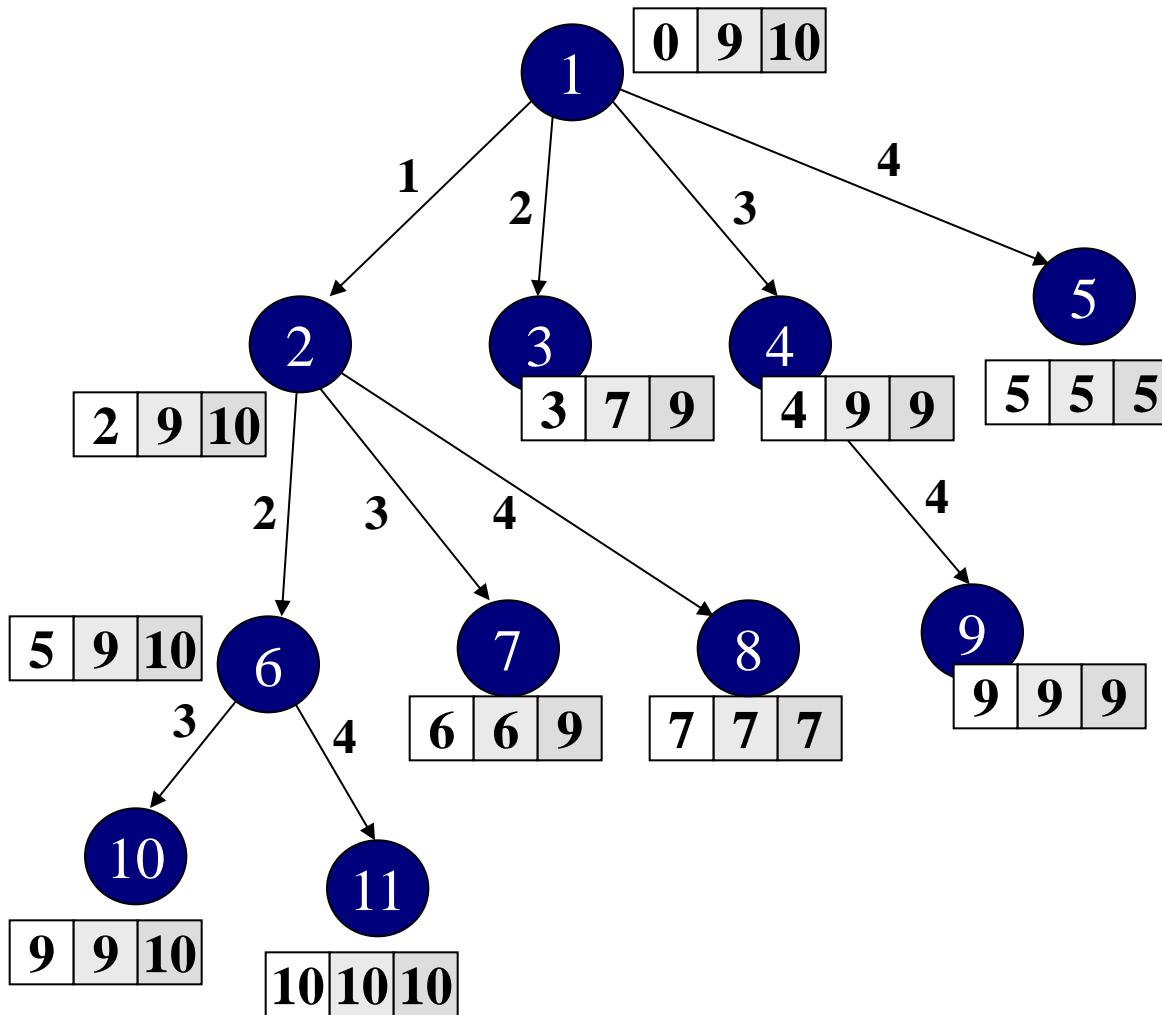
- Ejemplo. $n = 4$, $M = 7$, $v = (2, 3, 4, 5)$, $w = (1, 2, 3, 4)$



C	LNV
0	1
2	3 → 2
5	5 → 2 → 4
9	6 → 7 → 2 → 4
10	7 → 2 → 4
10	2 → 4
10	4

Ejemplo, 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	$\boxed{1}$
5	$\boxed{2} \rightarrow \boxed{4} \rightarrow \boxed{3}$
7	$\boxed{4} \rightarrow \boxed{6} \rightarrow \boxed{3} \rightarrow \boxed{7}$
9	$\boxed{6} \rightarrow \boxed{3} \rightarrow \boxed{7}$
10	$\boxed{3} \rightarrow \boxed{7}$
10	$\boxed{7}$

El Problema del Viajante de Comercio

- Este problema tiene un algoritmo exacto (lo veremos mas adelante) pero se le puede adaptar Branch and bound sin dificultad
- **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

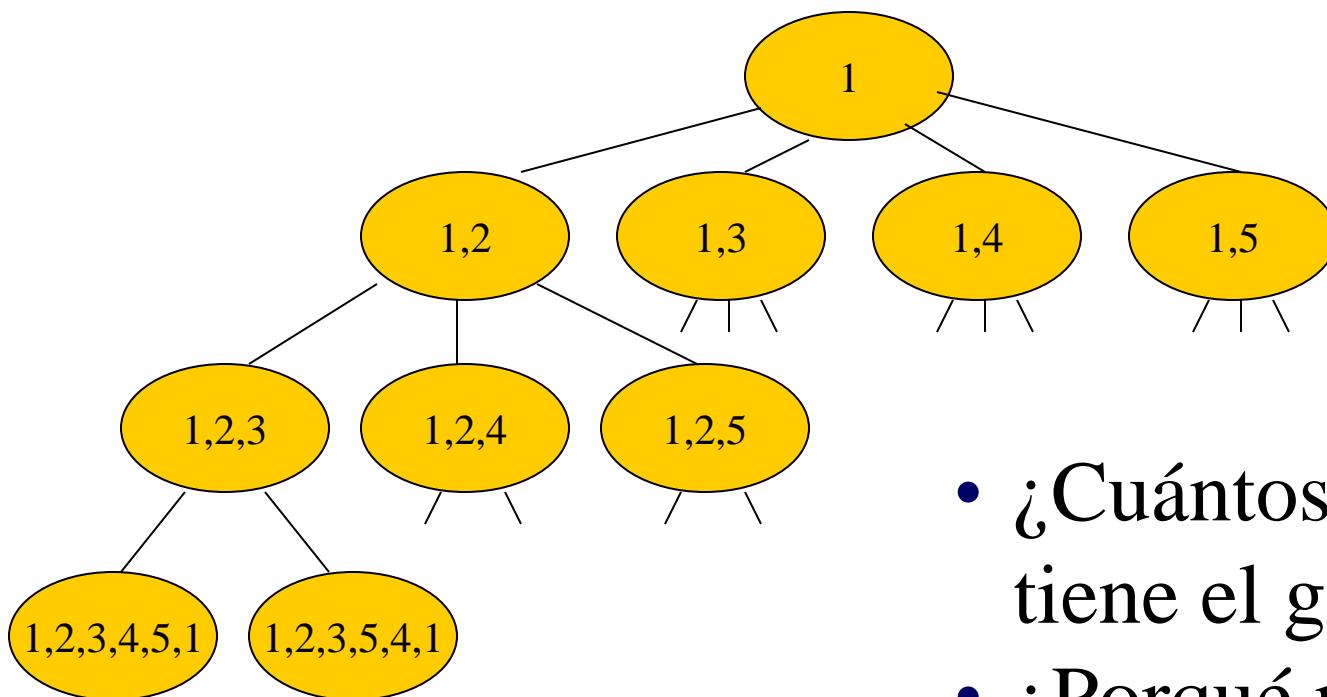
■ Formalización:

- Sean $G = (V, A)$ un grafo orientado, $V = \{1, 2, \dots, n\}$,
 - $D[i, j]$ 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

- 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?
- ¿Porqué no se requiere el último nivel en el árbol?

Análisis del problema con Branch and Bound

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

Estimación del costo posible a acumular

- Si se sabe cuáles son los vértices que faltan por visitar...
- Cada vértice que falta tiene arcos de salida hacia otros vértices...
- El mejor costo, será el del arco que tenga el valor menor...
- Esta información se puede obtener del renglón correspondiente al vértice en la matriz de adyacencias (excluyendo a los valores cero)...
- La sumatoria de los mejores arcos de cada vértice que falte, 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

$$\longrightarrow \text{Mínimo} = 4$$

$$\longrightarrow \text{Mínimo} = 7$$

$$\longrightarrow \text{Mínimo} = 4$$

$$\longrightarrow \text{Mínimo} = 2$$

$$\longrightarrow \text{Mínimo} = 4$$

TOTAL = 21

Ejemplo

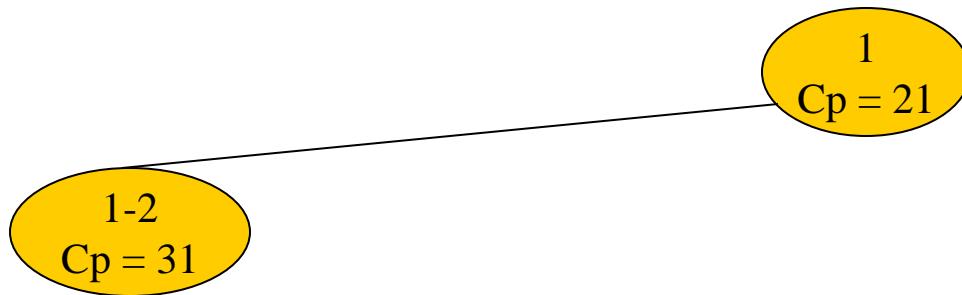
$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$

$$\begin{matrix} 1 \\ \text{Cp} = 21 \end{matrix}$$

Costo mínimo = ∞

Ejemplo

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



Costo mínimo = ∞

Cálculo del Costo posible:

Acumulado de 1-2 : **14**

Más mínimo de 2-3, 2-4 y 2-5: **7**

Más mínimo de 3-1, 3-4 y 3-5: **4**

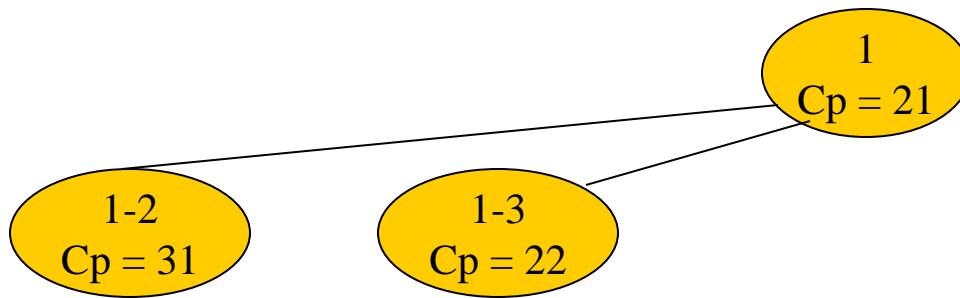
Más mínimo de 4-1, 4-3 y 4-5: **2**

Más mínimo de 5-1, 5-3 y 5-4: **4**

TOTAL = 31

Ejemplo

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



Costo mínimo = ∞

Cálculo del Costo posible:

Acumulado de 1-3 : 4

Más mínimo de 3-2, 3-4 y 3-5: 5

Más mínimo de 2-1, 2-4 y 2-5: 7

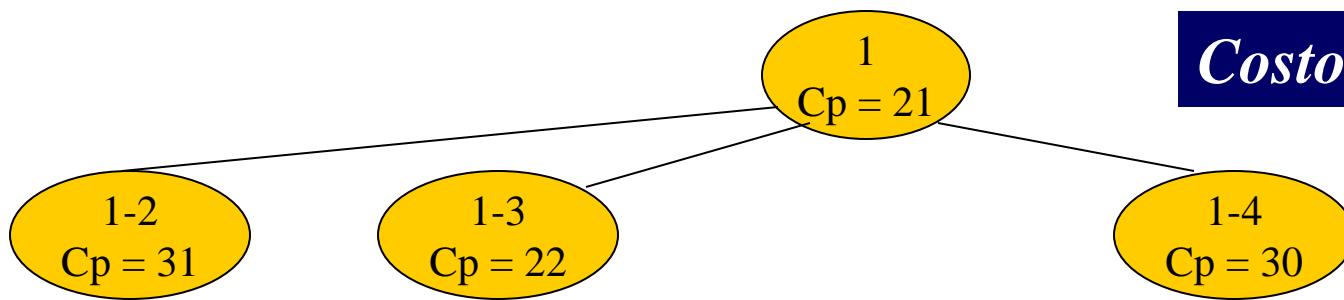
Más mínimo de 4-1, 4-3 y 4-5: 2

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 mínimo = ∞

Cálculo del Costo posible:

Acumulado de 1-4 : **10**

Más mínimo de 4-2, 4-3 y 4-5: **2**

Más mínimo de 3-1, 3-2 y 3-5: **4**

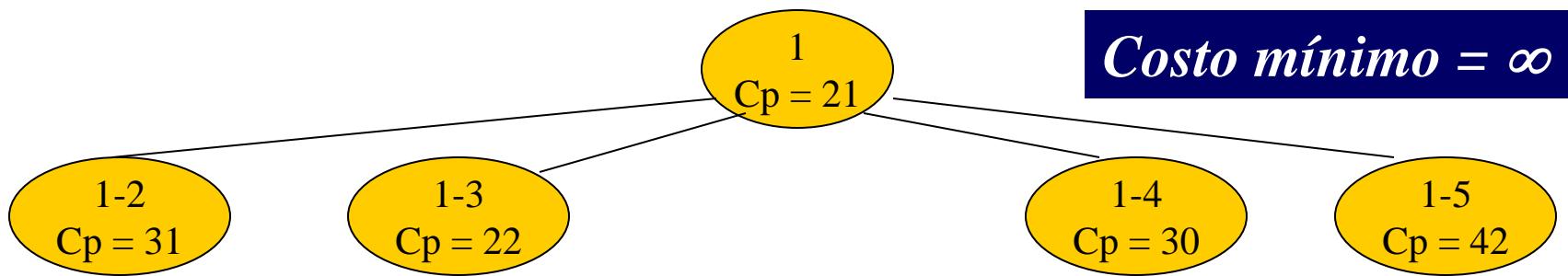
Más mínimo de 2-1, 2-3 y 2-5: **7**

Más mínimo de 5-1, 5-2 y 5-3: **7**

TOTAL = 30

Ejemplo

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



¿Cuál es el mejor nodo para expandir?

Cálculo del Costo posible:

Acumulado de 1-5 : 20

Más mínimo de 5-2, 5-3 y 5-4: 4

Más mínimo de 4-1, 4-2 y 4-3: 7

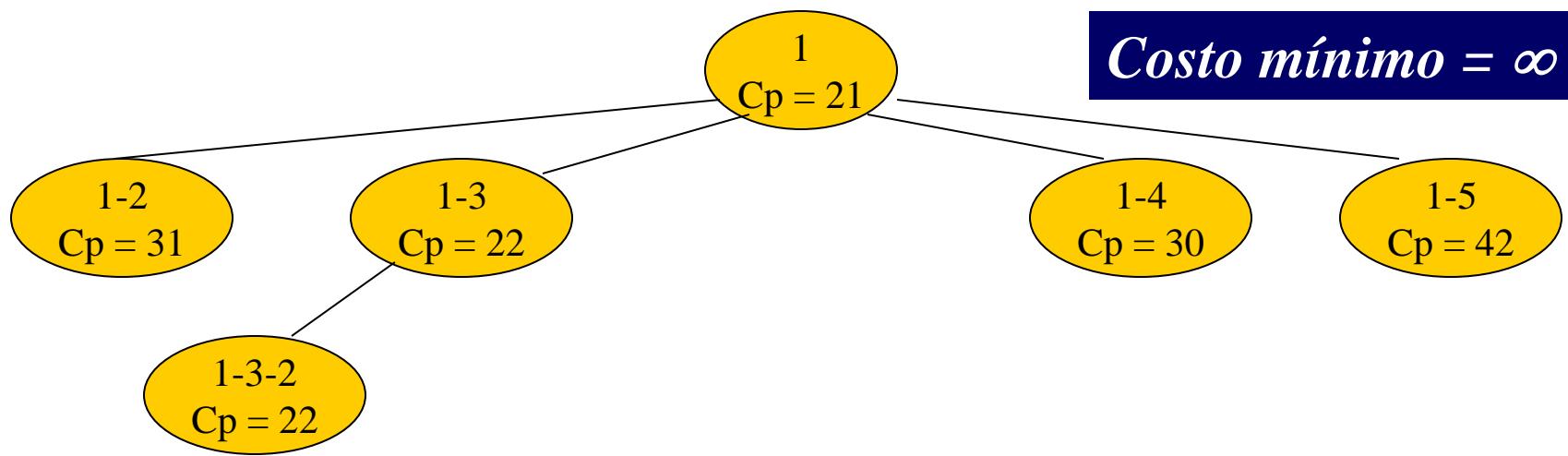
Más mínimo de 3-1, 3-2 y 3-4: 4

Más mínimo de 2-1, 2-3 y 2-4: 7

TOTAL = 42

Ejemplo

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



Cálculo del Costo posible:

Acumulado de 1-3-2 : 9

Más mínimo de 2-4 y 2-5: 7

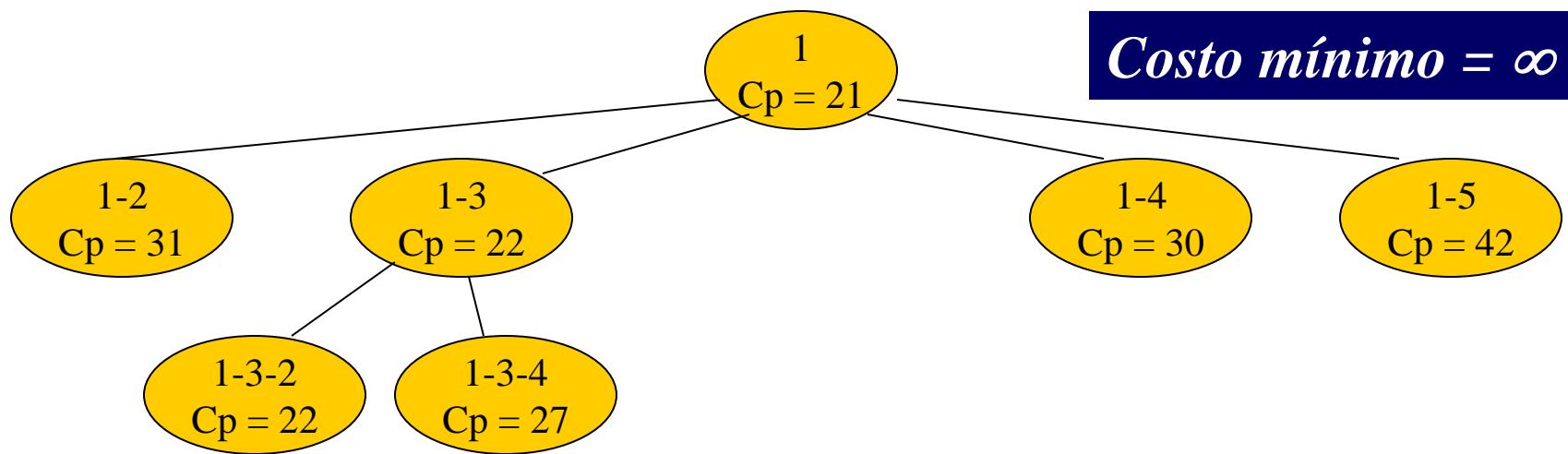
Más mínimo de 4-1 y 4-5: 2

Más mínimo de 5-1 y 5-4: 4

TOTAL = 22

Ejemplo

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



Cálculo del Costo posible:

Acumulado de 1-3-4 : 11

Más mínimo de 4-2 y 4-5: 2

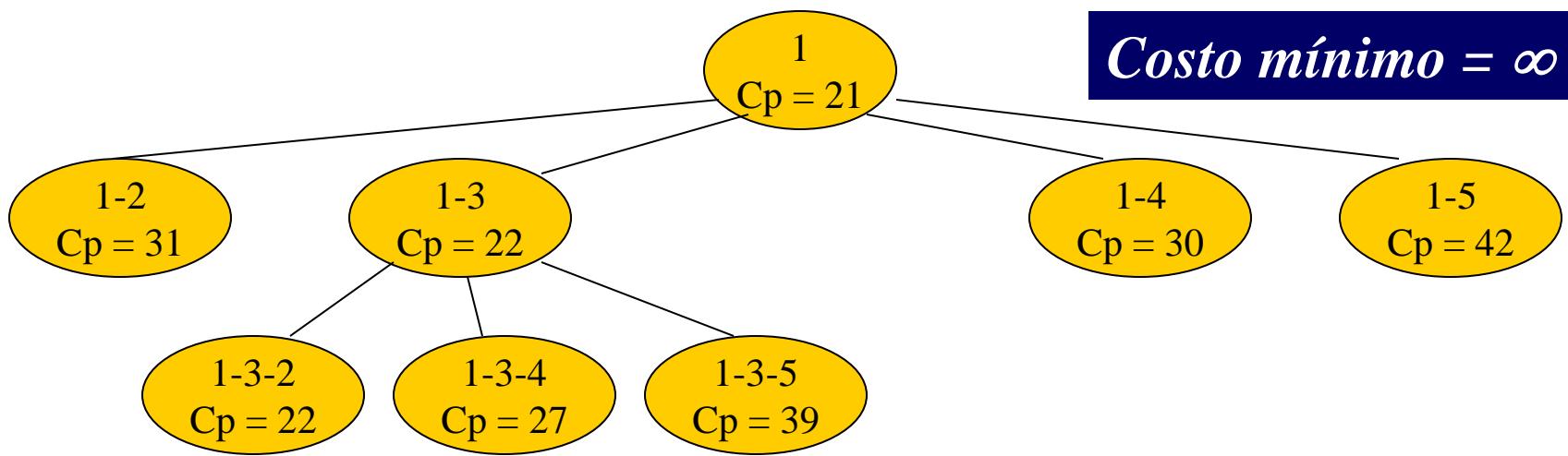
Más mínimo de 2-1 y 2-5: 7

Más mínimo de 5-1 y 5-2: 7

TOTAL = 27

Ejemplo

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



¿Cuál es el mejor nodo para expandir?

Cálculo del Costo posible:

Acumulado de 1-3-5 : **20**

Más mínimo de 5-2 y 5-4: **4**

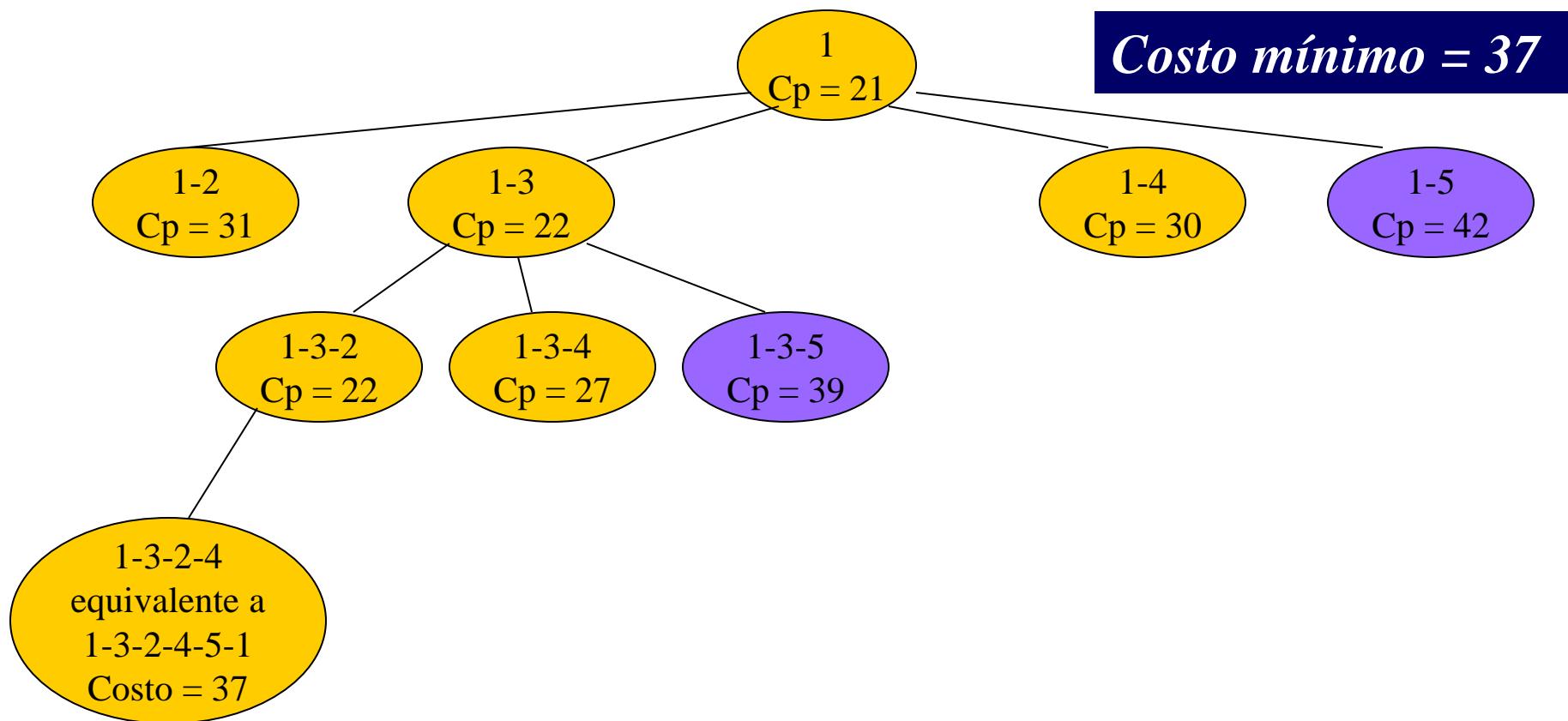
Más mínimo de 2-1 y 2-4: **8**

Más mínimo de 4-1 y 4-2: **7**

TOTAL = 39

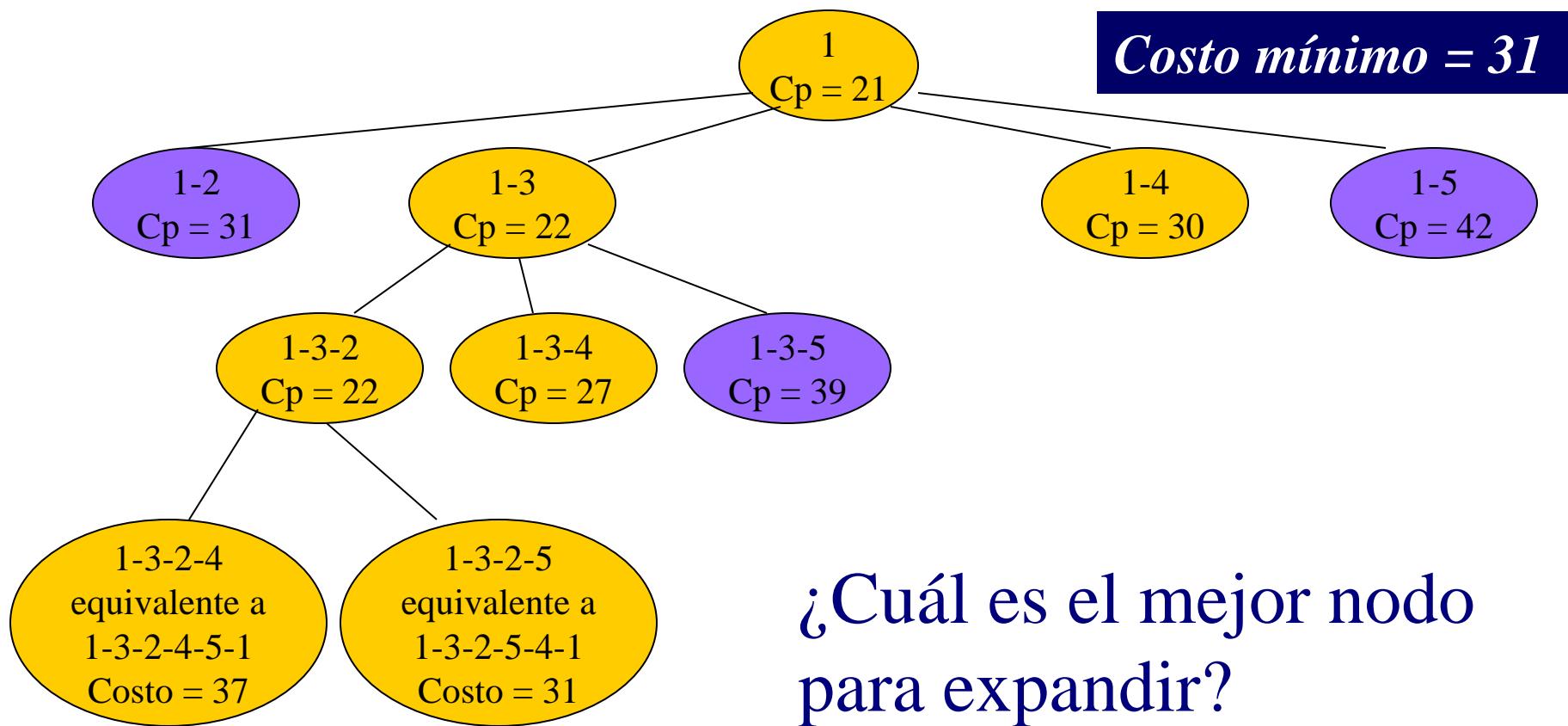
Ejemplo

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



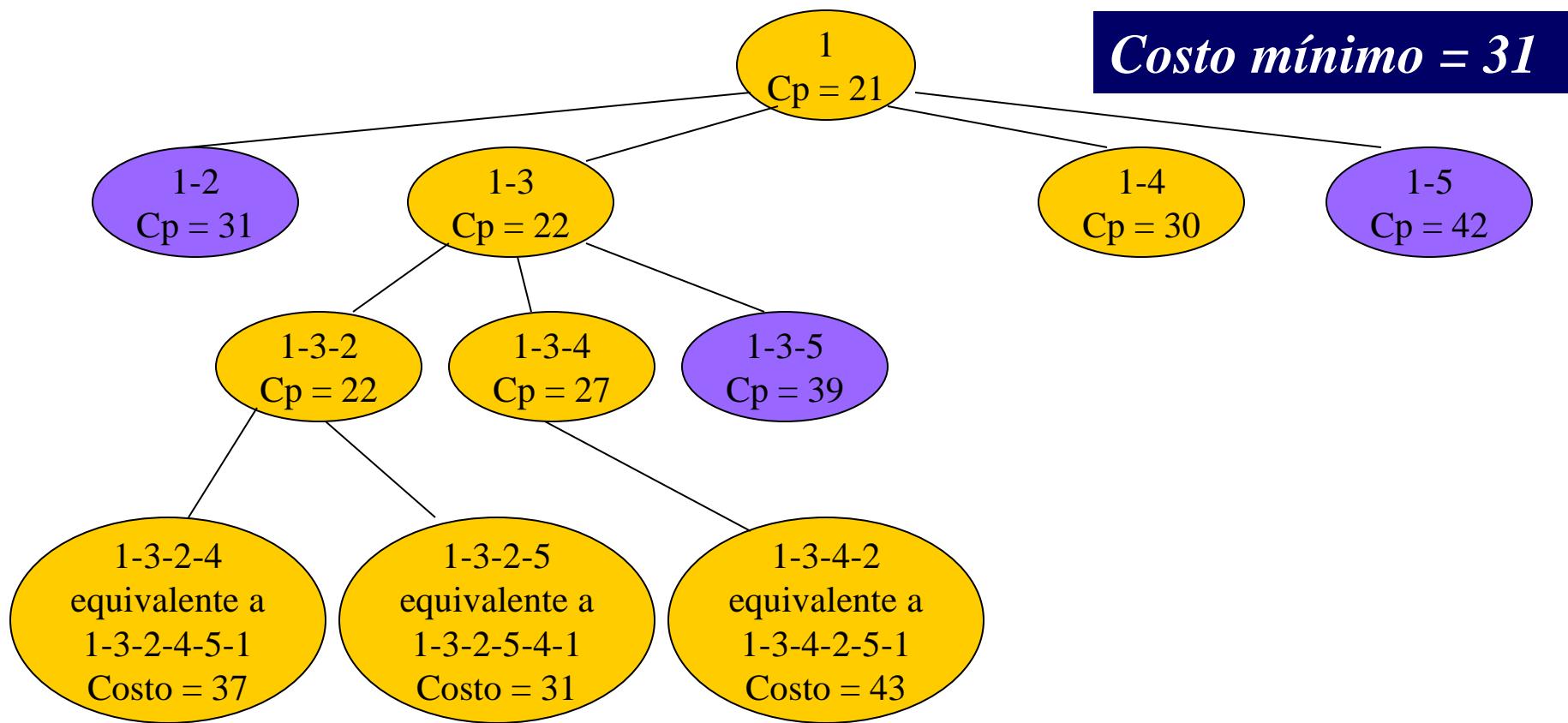
Ejemplo

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



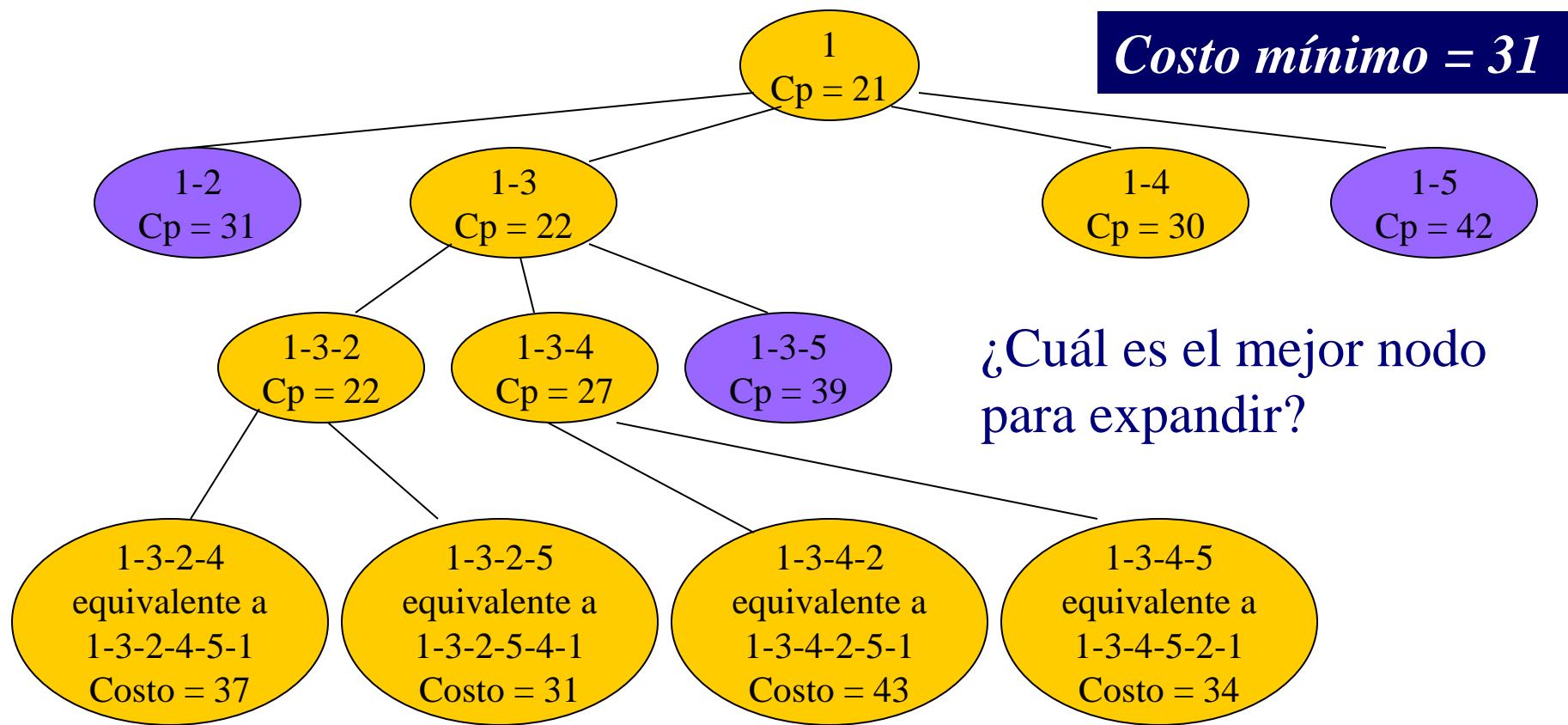
Ejemplo

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



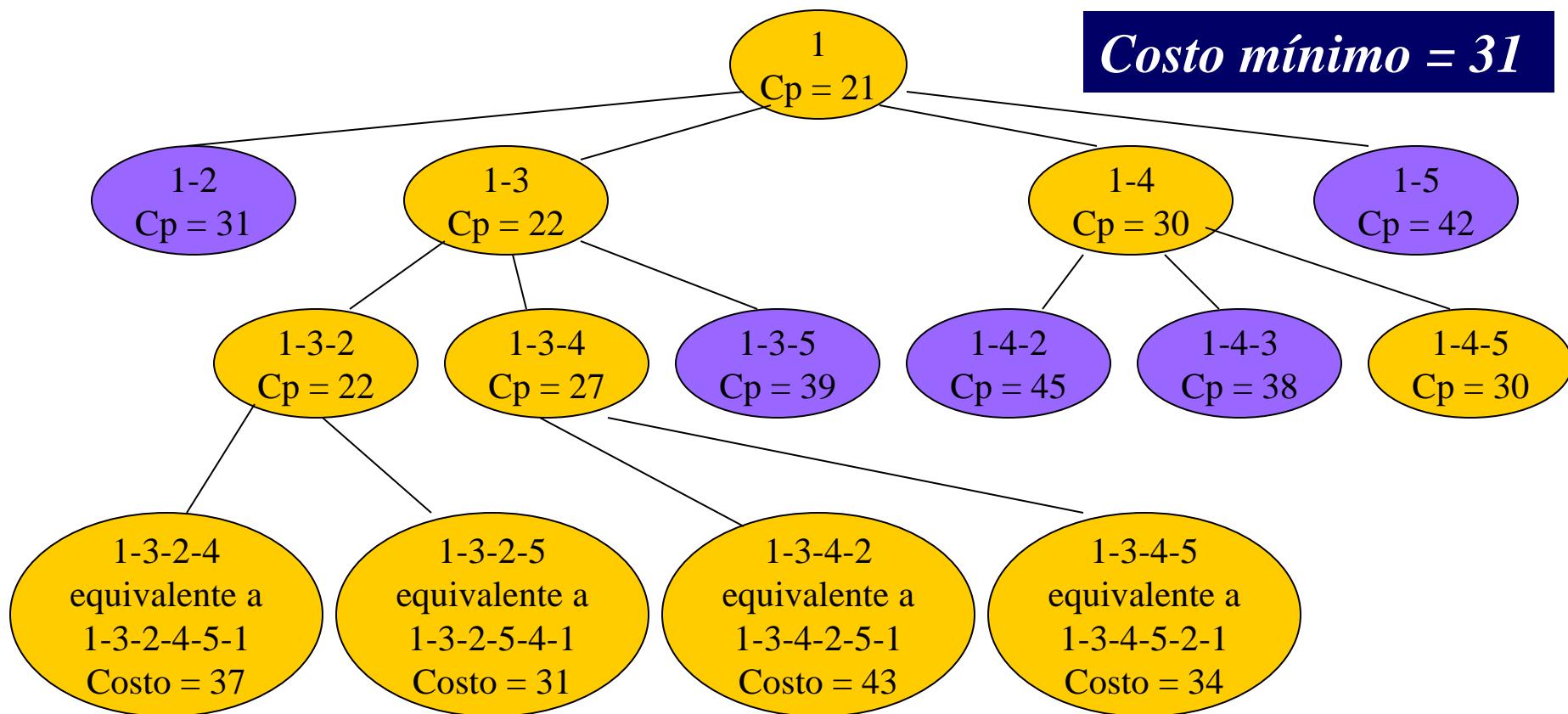
Ejemplo

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



Ejemplo

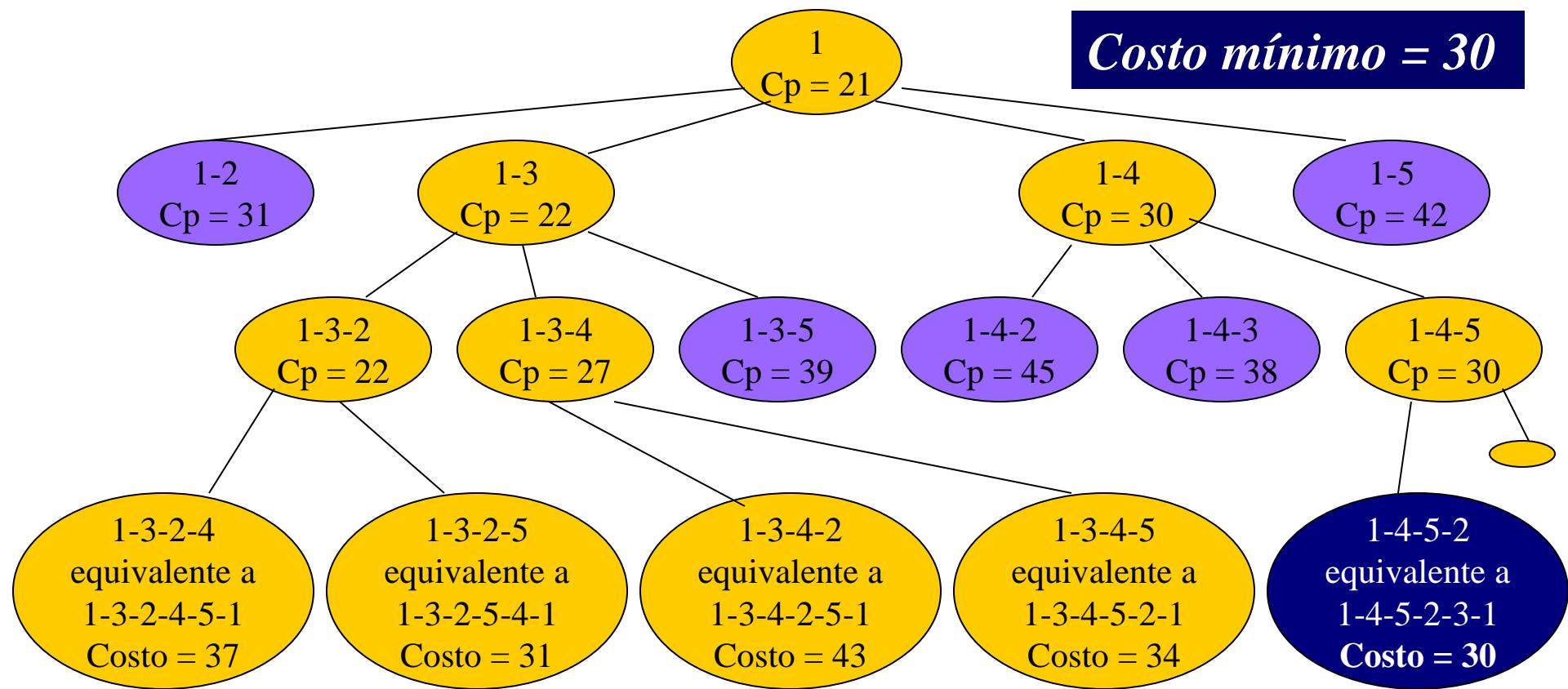
0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



¿Cuál es el mejor nodo para expandir?

Ejemplo

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



Conclusión final

El Problema del Viajante de Comercio

- 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, FANS, TS, ...

El juego del 15

Samuel Loyd: El juego del 15 o “taken”.

Problema publicado en un periódico de Nueva York en 1878 y por cuya solución se ofrecieron 1000 dólares.

- El problema original:

1	2	3	4
5	6	7	8
9	10	11	12
13	15	14	

Problema de Lloyd



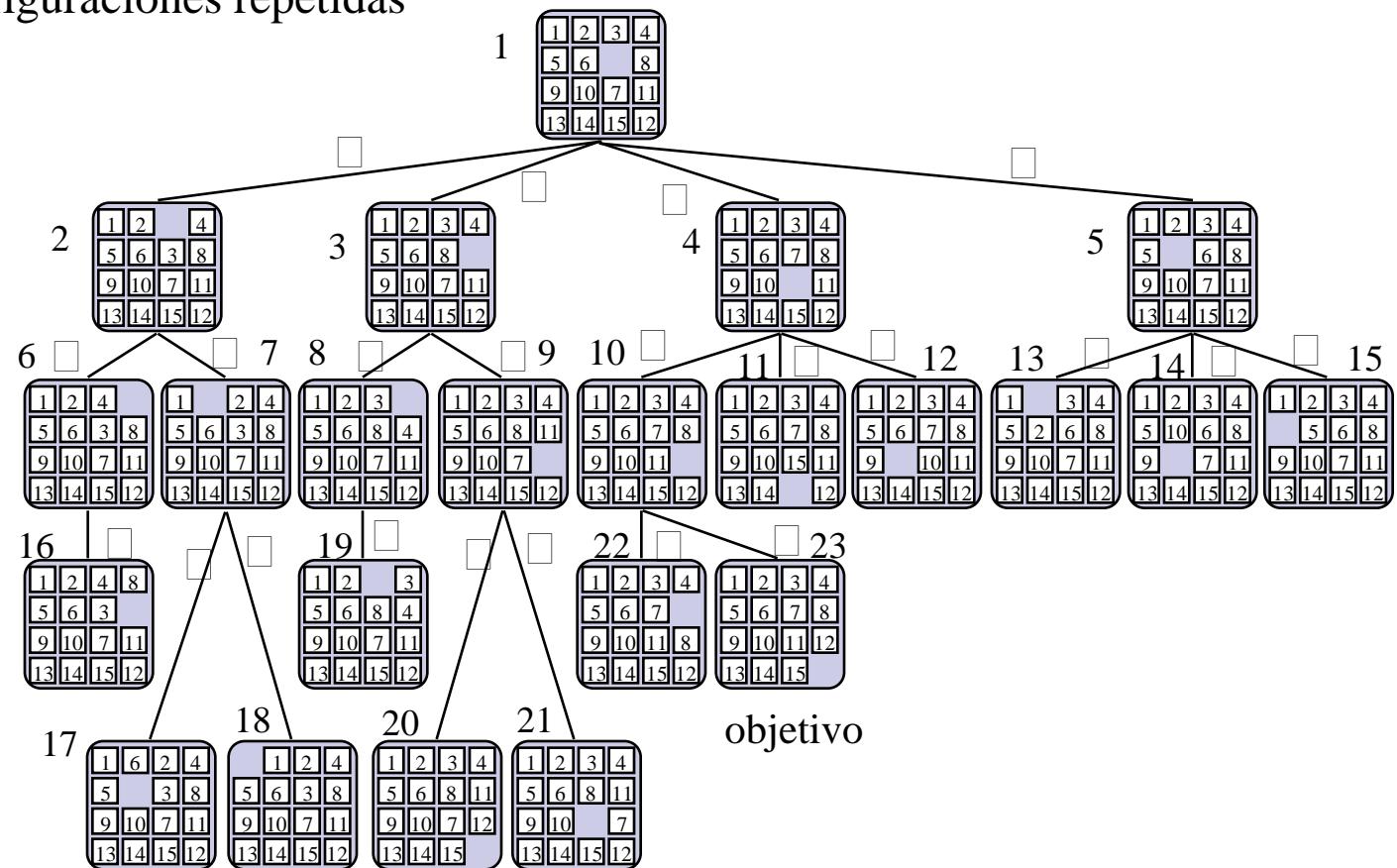
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

El objetivo

- Decisión: encontrar una secuencia de movimientos que lleven al objetivo
- Optimización: encontrar la secuencia de movimientos más corta

El juego del 15

- Configuración: permutación de $(1, 2, \dots, 16)$
- Solución: secuencia de configuraciones que Empiezan en el estado inicial y acaban en el final.
De cada configuración se puede pasar a la siguiente.
No hay configuraciones repetidas



El juego del 15

- **Problema muy difícil para backtracking**
 - El árbol de búsqueda es potencialmente muy profundo ($16!$ niveles), aunque puede haber soluciones muy cerca de la raíz.
- **Se puede resolver con branch and bound (aunque hay métodos mejores): Algoritmo A***
- **Funciones de prioridad:**
 - número de fichas mal colocadas (puede engañar)
 - suma, para cada ficha, de la distancia a la posición donde le tocaría estar
 - El 8-puzzle, introducción a la I.A. clásica

Algorítmica

Capítulo 6. Otras Metodologías Algorítmicas.

Tema 16. Algoritmos de Precomputación

-Algoritmos numéricos

- Evaluación de polinomios, Multiplicación de matrices, Sistemas de ecuaciones lineales**

Eficiencia de Algoritmos Numéricos

- Los programas numéricos suelen hacer cálculos muy concretos un gran número de veces
- Pequeñas, casi insignificantes, mejoras pueden producir importantes ahorros de tiempo debido a la gran cantidad de veces que se hace un cierto cálculo
- La eficiencia algorítmica se mide en términos de exactitud.

Preprocesamiento

- Sea I el conjunto de los casos de un problema, y supongamos que cada caso $i \in I$ consiste en dos componentes $j \in J$ y $k \in K$ (es decir $I \subseteq J \times K$)
- Un algoritmo de preprocesamiento para este problema es un algoritmo A que acepta como input algún elemento $j \in J$ y produce como output otro algoritmo B_j
- Ese algoritmo B_j debe ser tal que si $k \in K$ y $(j, k) \in I$, entonces la aplicación de B_j en k da la solución del caso (j, k) del problema original.

Ejemplo

- Sea J un conjunto de gramáticas para una familia de lenguajes de programación (C, Fortran, Cobol, Pascal ...) y K un conjunto de programas
- El problema general es saber si un programa dado es sintácticamente correcto en alguno de los lenguajes dados
- Aquí I es el conjunto de casos del tipo: ¿Es válido el programa k en el lenguaje que define la gramática $j \in J$?
- Un posible algoritmo de preprocessamiento para este ejemplo es un generador de compiladores:
 - Aplicado a la gramática $j \in J$ genera un compilador B_j para el lenguaje en cuestión
 - Por tanto para saber si $k \in K$ es un programa en el lenguaje j , simplemente aplicamos el compilador B_j a K

Preprocesamiento

- Sea:
 - $a(j)$ = tiempo para producir B_j dado j
 - $b_j(k)$ = tiempo para aplicar B_j a k
 - $t(j,k)$ = tiempo para resolver (j,k) directamente
- Generalmente $b_j(k) \leq t(j,k) \leq a(j) + b_j(k)$
- No interesa el preprocesamiento si
$$b_j(k) > t(j,k)$$

Utilidad del Preprocesamiento

- Suele ser útil en dos situaciones:
 - Emergencias: Necesitamos ser capaces de resolver cualquier caso muy rápidamente
 - Hay que resolver una serie de casos para un mismo valor de j : $(j, k_1), (j, k_2), \dots, (j, k_n)$. El tiempo consumido en resolver todos los casos si trabajamos sin preprocesamiento es

$$t_1 = \sum_{i=1..n} t(j, k_i)$$

y

$$t_2 = a(j) + \sum_{i=1..n} b_j(k_i)$$

si trabajamos con preprocesamiento

Cuando n es suficientemente grande, t_2 suele ser menor que t_1

- Lo estudiaremos asociado a problemas de tipo numérico en los que siempre intervienen unos mismos coeficientes.

Eficiencia en Algoritmos Numéricos

- Contaremos adiciones y multiplicaciones
- Como normalmente las adiciones son mucho mas rápidas que las multiplicaciones, la reducción del número de multiplicaciones, a costa del aumento de las adiciones, puede producir mejoras
- Nuestros análisis se basarán en la potencia mayor de un polinómio o en el tamaño de las matrices con las que estemos trabajando

Cálculo de Polinomios

- Usaremos la forma general de un polinomio

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

- en la que los valores de los coeficientes se suponen conocidos y constantes.
- Queremos evaluar (repetidamente) ese polinomio.
- El valor de x será el input y el output será el valor del polinomio usando ese valor de x

Algoritmo de Evaluación Estándar

```
result = a[0] + a[1]*x
```

```
xPower = x
```

```
for i = 2 to n do
```

```
    xPower = xPower * x
```

```
    result = result + a[i]*xPower
```

```
end for
```

```
return result
```

- El análisis fácil: Antes del lazo, hay una multiplicación y una adición. El lazo se hace $N-1$ veces y hay dos multiplicaciones y una adición. Por tanto hay un total de $2N-1$ multiplicaciones y N adiciones

Evaluación con el Método de Horner

- Se basa en la factorización de un polinomio
- Nuestra ecuación general puede factorizarse como

$$p(x) = \{L [(a_n x + a_{n-1}) * x + a_{n-2}] * x + L + a_2 \} * x + a_1 * x + a_0$$

- Por ejemplo, el polinomio

$$p(x) = x^3 - 5x^2 + 7x - 4$$

se factorizaría como

$$p(x) = [(x - 5) * x + 7] * x - 4$$

Evaluación con el Método de Horner

```
result = a[n]
```

```
for i = n - 1 down to 0 do
```

```
    result = result * x
```

```
    result = result + a[i]
```

```
end for
```

```
return result
```

- El análisis es sencillo: el lazo for se hace N veces y conlleva una multiplicación y una adición. Así que hay un total de N multiplicaciones y N adiciones. Por tanto nos ahorraremos $N-1$ multiplicaciones sobre el algoritmo estándar.

Preprocesamiento de Coeficientes

- Usa la factorización de un polinomio, considerada a partir de polinomios de grado mitad del original
- Por ejemplo, cuando el algoritmo estándar tuviera que hacer 255 multiplicaciones para calcular x^{256} , nosotros podríamos considerar el cuadrado de x y el del resultado, con lo que ahorraríamos mucho tiempo para obtener el mismo resultado
- Suponemos polinomios mónicos ($a_n=1$), con la mayor potencia siendo de valor uno menos que cierta potencia de 2.
- Así, si nuestro polinomio tiene como mayor potencia 2^k-1 , lo podemos factorizar como:

$$p(x) = (x^j + b)^* q(x) + r(x)$$

donde $j = 2^{k-1}$

Preprocesamiento de Coeficientes

- Si miramos el coeficiente del término $j - 1$ del polinomio

tomamos $b = \frac{p(x)}{a_{j-1} - 1} = (x^j + b)^* q(x) + r(x)$ y entonces $q(x)$ y $r(x)$ también serán mónicos, con lo que el proceso podrá aplicarse recursivamente sobre ellos también.

- Eso llevará a reducir el número de operaciones a efectuar.

Preprocesamiento de Coeficientes: Ejemplo

- Sea

$$p(x) = x^3 - 5x^2 + 7x - 4$$

Como la mayor potencia es $3 = 2^2 - 1$, entonces j sería $2^1 = 2$, y b valdría $a_1 - 1 = 6$.

- Así, nuestro factor es $x^2 + 6$. Entonces dividimos $p(x)$ por este polinomio para encontrar $q(x)$ y $r(x)$, y se obtiene

$$q(x) = x - 5, r(x) = x + 26$$

- Y por tanto $p(x) = (x^2 + 6)(x-5) + (x + 26)$

Análisis

- Analizamos el preprocesamiento de coeficientes desarrollando una ecuación de recurrencia para el número de multiplicaciones y adiciones.
- En nuestra factorización, partimos el polinomio en otros dos mas pequeños, haciendo una multiplicación mas y dos sumas adicionales
- Sea $M(k)$ el número de multiplicaciones requeridas para evaluar el polinomio de grado $N = 2^k - 1$.
- Sea $A(k) = M(k) - k + 1$ el número de multiplicaciones requeridas si no contamos las usadas en el cálculo de $x^2, x^4, \dots, x^{(n+1)/2}$
- Se obtiene la siguiente ecuación recurrente,

$$A(0) = 0 \text{ si } k = 1$$

$$A(k) = 2A(k-1) + 1 \quad \text{si } k \geq 2$$

Análisis

- Resolviendo esta ecuación obtenemos
$$A(k) = 2^{k-1} - 1, \text{ cuando } k \geq 1,$$
- y así
$$M(k) = 2^{k-1} + k - 2$$
- En otras palabras, $(N-3)/2 + \log(N+1)$ multiplicaciones son suficientes para evaluar un polinomio de grado $N = 2^k - 1$.

Comparación de los tres Algoritmos

- En el ejemplo que hemos visto:
 - **Algoritmo Estándar:**
 - 5 multiplicaciones y 3 adiciones
 - **Método de Horner**
 - 3 multiplicaciones y 3 adiciones
 - **Preprocesamiento de Coeficientes**
 - 2 multiplicaciones y 4 adiciones
- Para un polinomio de grado N:
 - **Algoritmo Estándar:**
 - $2N-1$ multiplicaciones y N adiciones
 - **Método de Horner**
 - N multiplicaciones y N adiciones
 - **Preprocesamiento de coeficientes**
 - $N/2 + \lg N$ multiplicaciones y $(3N-1)/2$ adiciones

Multiplicación de Matrices

- Dos matrices se pueden multiplicar si el número de columnas de la primera es igual al número de filas de la segunda
- Cada fila de la primera matriz se multiplica por cada columna de la segunda matriz
- El valor en la casilla i, j de la matriz es el resultado de la suma de los productos correspondientes a la fila i de la primera matriz por la columna j de la segunda

Multiplicación de Matrices Estándar

```
for i = 1 to a do
```

```
    for j = 1 to c do
```

$$R_{i,j} = G_{i,1} * H_{1,j}$$

```
        for k = 2 to b
```

$$R_{i,j} = R_{i,j} + G_{i,k} * H_{k,j}$$

```
        end for k
```

```
    end for j
```

```
end for I
```

- Por tanto, como sabemos, para multiplicar dos matrices $G(axb)$ y $H(bxc)$ el algoritmo hace $a*b*c$ multiplicaciones y $a*(b-1)*c$ adiciones

Multiplicación de Matrices de Winograd

- Podemos considerar cada fila y columna como vectores:
 $V = (v_1, v_2, v_3, v_4)$ y $W = (w_1, w_2, w_3, w_4)$
 - Cada elemento del resultado será el producto de dos vectores:

$$V \bullet W = v_1 * w_1 + v_2 * w_2 + v_3 * w_3 + v_4 * w_4$$

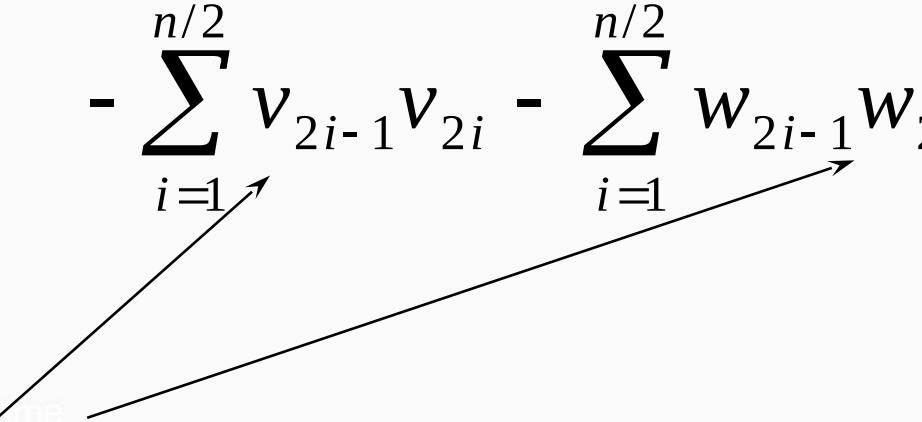
- Cada uno de esos productos se puede factorizar:

$$\begin{aligned} V \bullet W &= (v_1 + w_2) * (v_2 + w_1) + (v_3 + w_4) * (v_4 + w_3) \\ &\quad - v_1 * v_2 - v_3 * v_4 \end{aligned}$$

- Aunque esto parec \bar{e} m \acute{a} s trabajoso, las dos l \acute{u} ltimas l \acute{e} neas se pueden hacer solo una vez para cada fila de la primera matriz y cada columna de la segunda matriz.

Multiplicación de Matrices de Winograd

- Entonces para multiplicar una fila por una columna, tendríamos

$$V \bullet W = \sum_{i=1}^{n/2} (v_{2i-1} + w_{2i})(v_{2i} + w_{2i-1})$$
$$- \sum_{i=1}^{n/2} v_{2i-1}v_{2i} - \sum_{i=1}^{n/2} w_{2i-1}w_{2i}$$


- Donde estos dos valores se calculan una vez, pero se usan muchas veces.

Algoritmo de Winograd

Etapa de Preprocesamiento

```
d = b/2
// calcular los factores de las filas de la primera matriz (G)
for i = 1 to a do
    rowFactor[i] = Gi,1 * Gi,2
    for j = 2 to d do
        rowFactor[i] = rowFactor[i] + Gi,2j-1 * Gi,2j
    end for j
end for I
// calcular los factores de las columnas de la segunda matriz (H)
for i = 1 to c do
    columnFactor[i] = H1,i * H2,i
    for j = 2 to d do
        columnFactor[i] = columnFactor[i] + H2j-1,i * H2j,i
    end for j
end for i
```

Algoritmo de Winograd

Etapa de Cálculo

```
for i = 1 to a do
    for j = 1 to c do
        Ri,j = -rowFactor[i] - columnFactor[j]
        for k = 1 to d do
            Ri,j = Ri,j + (Gi,2k-1 + H2k,j)*(Gi,2k + H2k-1,j)
        end for k
    end for j
end for i
// sumas de terminos para la dimension compartida impar
if (2 * (b / 2) ≠ b) then
    for i = 1 to a do
        for j = 1 to c do
            Ri,j = Ri,j + Gi,b * Hb,j
    end for j
end for i
end if
```

Análisis del Algoritmo de Winograd

- Etapa de Preprocesamiento:
 - El lazo “for j” se hace $d-1$ veces, realizando una multiplicación y una adición
 - El lazo “for i” se hace a ($\circ c$) veces, llevando a cabo d multiplicaciones y $d-1$ adiciones
- Etapa de calculo para una dimensión par compartida (b) de las matrices:
 - El lazo “for k” se ejecuta d veces y hace una multiplicación y tres adiciones
 - El lazo “for j” se ejecuta c veces y hace d multiplicaciones y $3d + 1$ adiciones
 - El lazo “for i” se ejecuta a veces y hace $c \cdot d$ multiplicaciones y $c \cdot (3d + 1)$ adiciones

Análisis del Algoritmo de Winograd

- El algoritmo completo hace:

$a*d + c*d + a*c*d$ multiplicaciones

$a*(d-1) + c*(d-1) + a*c*(3d+1)$ adiciones

cuando b es par y $d = b/2$

Algoritmo de Strassen para la Multiplicación de Matrices

- Analizado ya con la técnica Divide y Vencerás:
- Usa siete formulas para multiplicar dos matrices 2×2
- No tiene en cuenta la commutatividad de los productos de elementos
- Puede aplicarse recursivamente:
 - Dos matrices 4×4 se pueden multiplicar considerando cada una de ellas como una matriz 2×2 de matrices 2×2

Análisis del Algoritmo de Strassen

- Esas formulas requieren 7 multiplicaciones y 18 adiciones para multiplicar dos matrices 2x2
- El ahorro real se da cuando se aplica recursivamente, haciendo aproximadamente $N^{2.81}$ multiplicaciones y $6N^{2.81}-6N^2$ adiciones
- Aunque no se usa en la práctica, el método de Strassen es importante porque fue el primer algoritmo que bajó de $O(N^3)$

Análisis del Algoritmo de Strassen

- Si suponemos que multiplicamos dos matrices de dimensiones idénticas NxN, la comparativa de los tres algoritmos es la de esta tabla

Algoritmo	Multiplicaciones	Adiciones
Estándar	N^3	$N^3 - N^2$
Wnograd	$(N^3 + 2N^2)/2$	$(3N^3 + 4N^2 - 4N)/2$
Strassen	$N^{2.81}$	$6N^{2.81} - 6N^2$

Ecuaciones Lineales

- Un sistema de ecuaciones lineales es un conjunto de N ecuaciones en N incógnitas.
- Los coeficientes (los valores a) son constantes y los resultados (los términos independientes) suelen ser los input de cada problema.
- Buscamos los valores x que satisfacen estas ecuaciones y producen los resultados indicados.
- Por ejemplo,

$$2x_1 - 4x_2 + 6x_3 = 14$$

$$6x_1 - 6x_2 + 6x_3 = 24$$

$$4x_1 + 2x_2 + 2x_3 = 18$$

Solución de Ecuaciones Lineales

- Un primer método de solución consiste en sustituir una ecuación en la otra
 - Por ejemplo, resolveríamos la primera ecuación para x_1 y entonces la sustituiríamos en el resto de las ecuaciones
- Esta sustitución reduce el número de ecuaciones y de incógnitas
- Iterando llegamos a una incógnita y una ecuación, de donde podremos ir recuperando los valores del resto
- Pero si hay muchas ecuaciones, el proceso es lento, produce errores y es difícil de implementar.

Algoritmo de Gauss-Jordan

- Este método se basa en la idea anterior
- Almacenamos las constantes en una matriz con N filas y N+1 columnas
- En nuestro ejemplo tendríamos:

$$\begin{array}{cccc} 2 & -4 & 6 & 14 \end{array}$$
$$\begin{array}{cccc} 6 & -6 & 6 & 24 \end{array}$$
$$\begin{array}{cccc} 4 & 2 & 2 & 18 \end{array}$$

Algoritmo de Gauss-Jordan

- Manipulamos algebraicamente las filas hasta obtener la matriz identidad en las primeras N columnas, con lo que los valores de las incógnitas se encontrarán en la última columna:

$$\begin{bmatrix} 1 & 0 & 0 & x_1 \\ 0 & 1 & 0 & x_2 \\ 0 & 0 & 1 & x_3 \end{bmatrix}$$

Algoritmo de Gauss-Jordan

- En cada paso, tomamos una nueva fila y la dividimos por el primer elemento que no es cero
- Restamos múltiplos de esta fila a todas las demás para obtener todo ceros en esta columna, menos en esta fila
- Cuando esto lo hemos hecho N veces, cada fila tendrá un valor 1 y la última columna presentará los valores de las incógnitas

Ejemplo

- Consideremos de nuevo el ejemplo:

$$\left[\begin{array}{cccc} 2 & -4 & 6 & 14 \\ 6 & -6 & 6 & 24 \\ 4 & 2 & 2 & 18 \end{array} \right]$$

- Comenzamos dividiendo la primera fila por 2, y entonces la restamos 6 veces de la segunda fila y cuatro de la tercera

Ejemplo

$$\begin{bmatrix} 1 & -2 & 3 & 7 \\ 0 & 6 & -12 & -18 \\ 0 & 10 & -10 & -10 \end{bmatrix}$$

- Ahora dividimos la segunda fila por 6, y entonces la sustraemos -2 veces de la primera y 10 veces de la tercera

Ejemplo

$$\begin{bmatrix} 1 & 0 & -1 & 1 \\ 0 & 1 & -2 & -3 \\ 0 & 0 & 10 & 20 \end{bmatrix}$$

- Ahora, dividimos la tercera fila por 10, y la restamos -1 veces de la primera y -2 veces de la segunda

Ejemplo

- Así logramos:

$$\begin{bmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{bmatrix}$$

- Y tenemos que $x_1 = 3$, $x_2 = 1$, y $x_3 = 2$, los valores que aparecen en el término independiente

Dificultades

- En la práctica, los redondeos producen resultados inexactos
- Si la matriz es singular, una fila será múltiplo exacto de alguna otra, dando lugar a un error al dividir por cero
- Hay mejores algoritmos para el problema, pero son propios de Análisis Numérico
- El cálculo de la inversa de la matriz de los coeficientes es un típico ejemplo de preprocessamiento: Regla de Cramer.

Fin de Curso

- Los objetivos del curso han sido
 - Dominar los métodos de cálculo de la eficiencia teórica de los algoritmos
 - Conocer en profundidad las técnicas de diseño de algoritmos y
 - Saber asociar a un problema el mejor algoritmo para su resolución
- Solo quedan dos aspectos que comentar:
 - El futuro a corto plazo (como será el examen), y
 - El futuro a medio plazo (como evaluaré el examen)

Últimas recomendaciones

- Pondré 6 preguntas:
 - Una será un ejercicio numérico que se puntuará sobre 10 y que, ponderado por 0.1 se sumará a la nota de prácticas (40% de prácticas).
 - Las otras 5 incluirán notaciones, greedy, DV, exploración de grafos y PD, y cada una puntuará sobre 10. La nota de teoría será la de la media de estas 5 preguntas ponderada por 0.6.
 - La nota final, será la suma de la nota de teoría (sobre 6) con las dos de prácticas (sobre 3 y sobre 1)
- Evaluación única: Las 6 preguntas del examen (cada una puntuada sobre 10)

Últimas recomendaciones

- El examen durará por lo menos 2 horas y media. El tiempo no será un problema
- No traigan dispositivos móviles o digitales. En concreto: calculadoras, teléfonos o cualquier dispositivo digital está prohibido en el examen.
- No se podrán usar apuntes o libros. Ahorrense su traslado
- Para preparar el examen,
 - No memoricen.
 - Estudien sobre libros y no sobre transparencias
 - Razonen el por qué de las cosas.
 - Relacionen métodos, conceptos
- Escriban claro, expresense bien, presenten bien el examen, repásenlo antes de entregarlo,

Y por último...

- Aprovechen las oportunidades que brindan los programas de intercambio
 - Erasmus (Europa)
 - LATAM, Canada, ...
 - Cooperación: CICODE
- Inicien contactos formativos sobre emprendimiento
- Contrato tácito
 - Formación continua. Mantengase en contacto con la Escuela y con sus profesores.

Algorítmica

Capítulo 6. Otras Metodologías Algorítmicas.
Tema 17. Algoritmos de Transformación Algebraica
-La Transformada de Fourier

Parecía una locura

- Diseñar un algoritmo que es una unidad de magnitud mas rápido que otro, es un importante logro.
- Cuando esa mejora esta asociada a algún proceso que tiene muchas aplicaciones, lo conseguido tiene gran repercusión en el terreno científico y tecnológico.
- Este es el caso de la transformada de Fourier.
- Ninguna mejora en ningún algoritmo ha tenido un impacto mayor que el provocado por este método.
- ¿Por qué?. ¿Cuál es su origen?. ¿Tiene alguna repercusión en el contexto de la Informática?



¿Como se representan las señales?

- Una serie de Taylor series representa cualquier función empleando polinomios

$$\begin{aligned}f(x) &= f(\alpha) + f'(\alpha)(x - \alpha) + \frac{f''(\alpha)}{2!} \\&\quad (x - \alpha)^2 + \frac{f^{(3)}(\alpha)}{3!} (x - \alpha)^3 + \dots + \frac{f^{(n)}(\alpha)}{n!} (x - \alpha)^n + \dots\end{aligned}$$

- Pero los polinomios dan problemas porque son inestables y porque fisicamente no son fácilmente comprensibles.
- Nos resulta mas intuitivo y sencillo hablar sobre “señales” en terminos de sus “frecuencias” (como de rápido, cada cuanto cambian las señales, etc.)

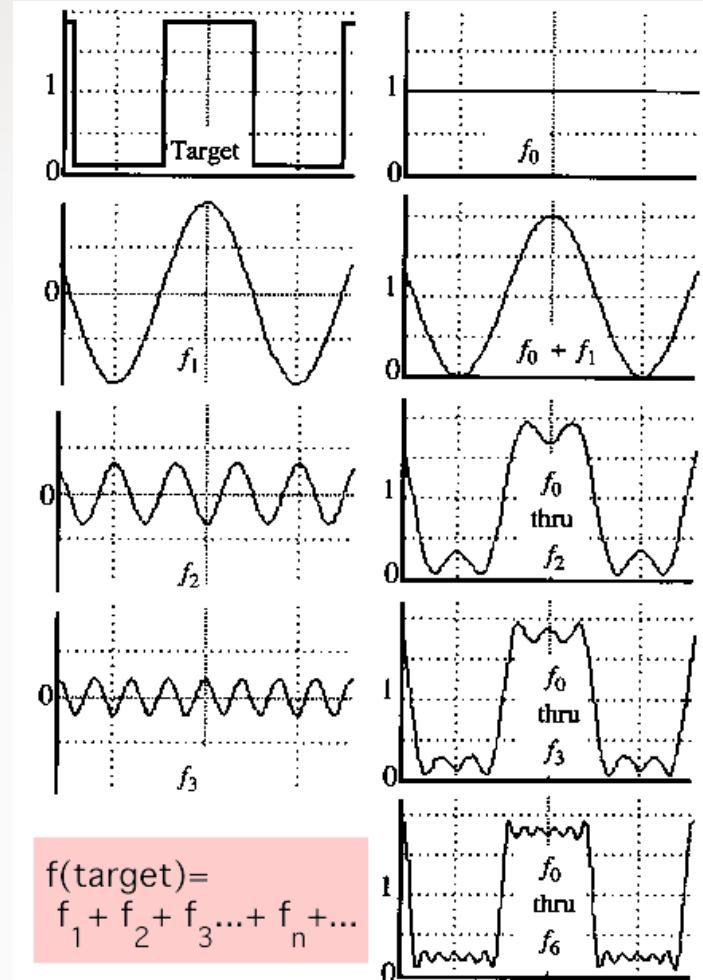
Jean Baptiste Joseph Fourier (1768-1830)

- En 1807 Fourier tuvo una idea:
- Cualquier función periódica podía reescribirse como la suma ponderada de senos y cosenos de diferentes frecuencias
- Difícil de creer ¿verdad?
 - No se lo creyeron Lagrange, Laplace, Poisson y muchos otros “popes” de la época.
 - Su trabajo no se tradujo al inglés hasta 1878 (¡71 años después!)
- Pero es verdad
 - Se hace con la **Serie de Fourier**
 - Posiblemente la mas poderosa herramienta de las Ingenierías



Una suma de sinusoides

- A partir de la herramienta básica:
 $A \sin(\omega x + \phi)$
- Como en el mundo de los bloques:
- se suman tantos de ellos como sean necesarios (se construye la serie) para representar la señal $f(x)$ que queremos reproducir



La transformada de Fourier

- Buscamos interpretar (conocer) la frecuencia ω de nuestra señal. Para ello reparametrizamos la señal, en lugar de por medio de x , por medio de ω



- Para cualquier ω entre 0 e infinito, $F(\omega)$ nos da la amplitud A y la fase ϕ de $A \sin(\omega x + \phi)$
- Esto es así gracias a que,

$$F(\omega) = R(\omega) + iI(\omega)$$

$$A = \pm \sqrt{R(\omega)^2 + I(\omega)^2} \quad \phi = \tan^{-1} \frac{I(\omega)}{R(\omega)}$$

Your company name



La Transformada Rápida de Fourier

- La transformada de Fourier de una función continua $a(t)$ esta dada por

$$A(f) = \int_{-\infty}^{\infty} a(t)e^{2\pi i f t} dt$$

- mientras que la transformada inversa esta dada por

$$a(t) = 1/(2\pi) \int_{-\infty}^{\infty} A(f)e^{-2\pi i f t} df.$$

- La i en las dos anteriores ecuaciones representa la raiz cuadrada de -1 y la constante e es la base de los logaritmos naturales.
- A menudo la variable t se entiende que es el tiempo, mientras que f se toma como la frecuencia, de modo que, como hemos explicado antes,
- La transformada de Fourier expresa una función del tiempo en una de la frecuencia.

La Transformada Rápida de Fourier

- Asociada a esta transformada de Fourier continua, hay una versión discreta, que se llama **Transformada de Fourier Discreta**, y que trabaja sobre puntos muestrales de $a(t)$, a_0 , a_1 , ... a_{N-1} .
- La transformada discreta de Fourier se define por

$$A_j = \sum_{0 \leq k \leq N-1} a_k e^{2\pi i j k / N}, \quad 0 \leq j \leq N - 1$$

- siendo la inversa

$$a_k = (1/N) \sum_{0 \leq j \leq N-1} A_j e^{-2\pi i j k / N}, \quad 0 \leq k \leq N - 1$$

- En el caso discreto se da un conjunto de N puntos muestrales y se obtiene un nuevo conjunto de N puntos.

La Transformada Rápida de Fourier

- Llegado a este punto nos interesa destacar la estrecha conexión que existe entre la Transformada Discreta de Fourier y la evaluación de polinomios.
- En efecto, supongamos el polinomio

$$a(x) = a_{N-1}x^{N-1} + a_{N-2}x^{N-2} + \dots + a_1x + a_0$$

- entonces el elemento Fourier A_j es el valor de $a(x)$ en $x = w^j$, donde $w = e^{2\pi i / N}$. De manera similar, para la transformada inversa de Fourier, si imaginamos el polinomio con los coeficientes de Fourier

$$A(x) = A_{N-1}x^{N-1} + A_{N-2}x^{N-2} + \dots + A_1x + A_0$$

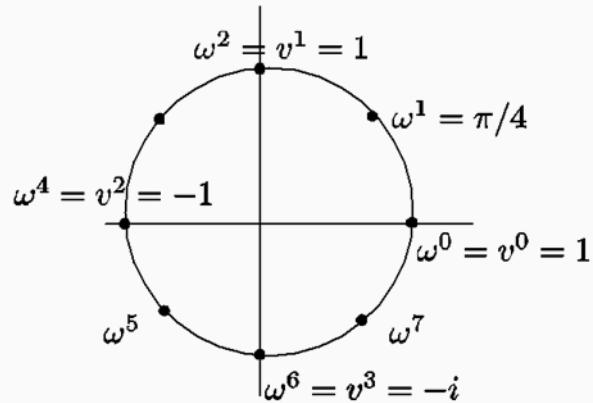
- entonces cada a_k es el valor de $A(x)$ en $x = (w^{-1})^k$ donde $w = e^{2\pi i / N}$.
- Por tanto la Transformada de Fourier Discreta se corresponde exactamente con la evaluación de un polinomio en N los puntos w^0, w^1, \dots, w^{N-1} .

La Transformada Rápida de Fourier

- Recordemos que un polinomio de grado N podemos evaluarlo en N puntos usando $O(N^2)$ operaciones.
- Aplicamos el algoritmo de Horner una vez en cada punto.
- La Transformada Rápida de Fourier (TRF) es un algoritmo para calcular esos N valores usando solo $O(N \log N)$ operaciones.
- Este algoritmo fue popularizado por Cooley y Tukey en 1965, y luego explotado por el mismo Cooley.
- Conviene fijarse en que la Transformada de Fourier puede ser mas rápida que el algoritmo de Horner porque los puntos de evaluación no son arbitrarios, sino mas bien muy especiales. De hecho son las N potencias w^j para $0 \leq j \leq N-1$, donde $w = e^{2\pi i/N}$.
- El punto w es una raiz primitiva N -ésima de la unidad en el plano complejo

Formalidades

- **Definición.** Un elemento w en un anillo commutativo es llamado una raíz primitiva N -ésima de la unidad si
 - (i) $w \neq 1$
 - (ii) $w^N = 1$
 - (iii) $\sum_{p=0..N-1} w^{jp} = 0, 1 \leq j \leq N-1$
- De manera más intuitiva, una raíz N -ésima de la unidad es un número complejo z tal que $z^n = 1$. Exactamente hay N raíces primitivas de la unidad, $w^k, k = 0, 1, \dots, N-1$



Formalidades

- Las dos siguientes propiedades de las raíces N-ésimas sirven para entender mejor el algoritmo de la TRF.
 - 1) Sea $N = 2n$ y supongamos que w es una raíz primitiva N-ésima de la unidad. Entonces
$$- w^j = w^{j+n}$$
 - 2) Sea $N = 2n$ y w una raíz primitiva N-ésima de la unidad. Entonces w^2 es una raíz primitiva N-ésima de la unidad.
- A partir de estas propiedades se puede concluir que si w^j , $0 < j \leq N-1$ son las raíces primitivas N-ésimas de la unidad, $N = 2n$, entonces w^{2j} , $0 < j \leq n-1$ son n raíces primitivas de la unidad.
- A partir de estas dos propiedades podemos obtener un algoritmo Divide y Vencerás para la Transformada de Fourier.
- La complejidad del algoritmo será $O(N \log N)$, mas rápido por tanto que el que proporciona el algoritmo de evaluación polinómica convencional $O(N^2)$.

Algoritmo para el cálculo de la TRF

- Consideremos de nuevo que los coeficientes que queremos transformar son a_{N-1}, \dots, a_0 y sea

$$a(x) = a_{N-1}x^{N-1} + \dots + a_1x + a_0$$

- Dividimos $a(x)$ en dos partes, una conteniendo los exponentes numerados impar, y la otra con los pares.

$$\begin{aligned} a(x) = & a_{N-1}x^{N-1} + a_{N-3}x^{N-3} + \dots + a_1x + \\ & + a_{N-2}x^{N-2} + \dots + a_2x^2 + a_0 \end{aligned}$$

- Tomando $y = x^2$ podemos reescribir $a(x)$ como la suma de dos polinomios

$$\begin{aligned} a(x) = & (a_{N-1}y^{n-1} + a_{N-3}y^{n-2} + \dots + a_1)x + (a_{N-2}y^{n-1} + a_{N-4}y^{n-2} + \dots + a_0) = \\ = & c(y) \cdot x + b(y) \end{aligned}$$

Algoritmo para el cálculo de la TRF

- $a(x) = (a_{N-1}y^{n-1} + a_{N-3}y^{n-2} + \dots + a_1)x + (a_{N-2}y^{n-1} + a_{N-4}y^{n-2} + \dots + a_0) =$
 $= c(y) \cdot x + b(y)$
- Recordemos que los valores de la TRF son $a(w^j)$, $0 \leq j \leq N-1$ y que los valores de $a(x)$ en los puntos w^j , $0 \leq j \leq n-1$ son ahora expresables como,

$$a(w^j) = c(w^{2j}) w^j + b(w^{2j})$$

$$a(w^{j+n}) = -c(w^{2j}) w^j + b(w^{2j})$$

- Con estas dos formulas un problema de tamaño N se resuelve transformándolo en 2 problemas idénticos de tamaño $n = N/2$.
- Estos subproblemas son la evaluación de $b(y)$ y $c(y)$, cada uno de grado $n-1$, en los puntos $(w^2)^j$, $0 \leq j \leq n-1$, que son raíces primitivas n -ésimas.

Algoritmo para el cálculo de la TRF

- Esto es un ejemplo de Divide y Vencerás, y por tanto podemos volver a aplicar esa metodología mientras sigamos teniendo un número par de puntos.
- Esto nos lleva a elegir siempre N como una potencia de 2, $N = 2^m$, ya que así podemos continuar desarrollando este procedimiento de división hasta que alcancemos un problema trivial, es decir, hasta que lleguemos a tener que evaluar un polinomio constante.
- Así, el algoritmo Divide y Vencerás para el cálculo de la TRF tiene una eficiencia $O(N \log N)$ obtenida a partir de la recurrencia,

$$T(N) = 2T(N/2) + cN$$

Algoritmo para el cálculo de la TRF

- Un algoritmo recursivo DV para el cálculo de la TRF (Horowitz y Sahni, 1978) es este:

```
procedure FFT(N, a(x), w, A)
    //N =  $2^m$ , a(x) =  $a_{N-1}x^{N-1} + \dots + a_0$ , w is a //
    //primitive N-th root of unity A(0:N - 1) is set to //
    //the values a(wj),  $0 \leq j \leq N - 1$ . //
    integer N real A(0:N - 1), B(0:(N/2) - 1), C(0:(N/2) - 1),
           WP(-1:(N/2) - 1)
    if N = 1 then A(0) ← a0
    else n ← N/2
        b(x) ←  $a_{N-2}x^{n-1} + \dots + a_2x + a_0$  //divide the coefficients //
        c(x) ←  $a_{N-1}x^{n-1} + \dots + a_3x + a_1$  //into 2 sets //
        call FFT(n, b(x), w2, B) //apply this algorithm again //
        call FFT(n, c(x), w2, C) //and again //
        WP(-1) ← 1/w
        for j = 0 to n - 1 do
            WP(j) ← w*WP(j - 1)
            A(j) ← B(j) + WP(j)*C(j)
            A(j + n) ← B(j) - WP(j)*C(j)
        repeat
    endif
end FFT
```

Una última nota

- Consideraremos el problema de la multiplicación de polinomios.
- La técnica de la transformada llama a evaluar $A(x)$ y $B(x)$ en $2N + 1$ puntos, calculando los $2N + 1$ productos $A(x_i) \cdot B(x_i)$ y obteniendo el producto $A(x)B(x)$.
- Sabremos que la evaluación de N puntos requiere $O(N^2)$ operaciones, de modo que parece que no ganamos nada con esta técnica de la transformada.
- Pero lo que acabamos de ver es que si los puntos los elegimos de manera que sean las $N = 2^m$ potencias distintas de una raíz primitiva N -ésima de la unidad, entonces la evaluación, y la interpolación, pueden hacerse con $O(N \log N)$ operaciones.
- Por tanto usando el algoritmo de la TRF podemos multiplicar dos polinomios de grados N con $O(N \log N)$ operaciones.