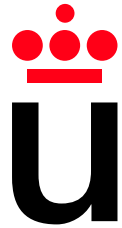
Transparencias

Diseño y Análisis de Algoritmos

Manuel Rubio Sánchez

13 de mayo de 2024



Universidad
Rey Juan Carlos

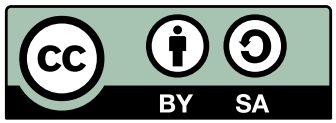
Copyright

©2024 Manuel Rubio Sánchez

Algunos derechos reservados.

Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en:

https://creativecommons.org/licenses/by-sa/4.0/deed.es.



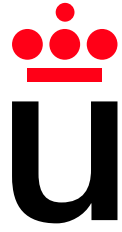
Índice de contenidos

Tema 1
Introducción a la algoritmia

Diseño y Análisis de Algoritmos

Manuel Rubio Sánchez

13 de mayo de 2024



Universidad
Rey Juan Carlos

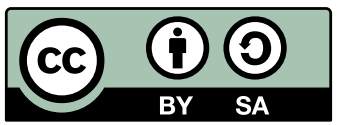
Copyright

©2024 Manuel Rubio Sánchez

Algunos derechos reservados.

Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en:

https://creativecommons.org/licenses/by-sa/4.0/deed.es.



Contenido

Problemas computacionales y algoritmos

Problema computacional

- Objeto matemático que representa una conjunto de preguntas que queremos resolver (con la ayuda de computadoras)
- Usa *precondiciones* y *postcondiciones* para describir las salidas o acciones adecuadas para unas entradas (instancias) válidas
- El enunciado especifica la relación entre las entradas y las salidas
- Una instancia del problema define las entradas necesarias para calcular la solución del problema
- Pueden verse como colecciones (a menudo infinitas) de instancias, junto con las soluciones a cada instancia

Problemas computacionales

Ejemplo: problema de ordenación

- Entradas: una secuencia de n números $\langle a_1, a_2, \ldots, a_n \rangle$
- Salidas: Una permutación (reordenación) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ de la secuencia de entrada tal que $a'_1 \leq a'_2 \leq \cdots \leq a'_n$
- Una posible instancia (entrada): $\langle 3, 14, 7, 1, 8, 12 \rangle$
- La solución (salida): $\langle 1, 3, 7, 8, 12, 14 \rangle$

Tipos de problemas computacionales

- Matemáticos o lógicos (funciones)
- De decisión (salida binaria/booleana)
- De ordenación
- De búsqueda
- Combinatorios
- Optimización
- Geométricos
- Sobre grafos
- Sobre cadenas

⋮

Algunos problemas pueden pertenecer a varias categorías

Algoritmo

- No hay una definición formal aceptada de "algoritmo"
- Procedimiento, conjunto de instrucciones, "receta", etc., que define una serie de pasos, bien definidos, tales que lleven a obtener las salidas adecuadas para un conjunto de datos de entrada
- La secuencia de pasos no tiene por qué ser determinista
- Un algoritmo es **correcto** si para cada instancia de entrada para con el resultado correcto
- En ese caso, decimos que el algoritmo *resuelve* el problema computacional
- Algoritmos incorrectos también pueden ser útiles, si el error que cometen puede ser controlado

Algoritmos según su implementación

- Iterativos/recursivos
  - Todo algoritmo iterativo se puede expresar de forma recursiva y viceversa
- Lógicos
  - Los algoritmos son procesos de deducción lógica controlada
- En serie, paralelos o distribuidos
  - Los algoritmos paralelos o distribuidos dividen los problemas en subproblemas que son tratados en diferentes procesadores o máquinas, para posteriormente combinar los resultados

Algoritmos según su implementación

- Deterministas o no deterministas
  - Deterministas: siempre toman la misma decisión en cada paso del algoritmo
  - No deterministas: utilizan algún grado de aleatoriedad para tomar decisiones. Emplean heurísticas, generalmente derivadas del conocimiento humano para tomar mejores decisiones
- Exactos o aproximados
  - Exactos: alcanzan la solución exacta
  - Aproximados: buscan una solución que se aproxime a la verdadera. Son útiles para problemas duros o intratables

Algoritmos según el paradigma de diseño

- Divide o vencerás
- Fuerza bruta o exhaustivos
  - *Backtraking* o "vuelta atrás"
- Algoritmos voraces
- Programación dinámica
  - En la asignatura "Algoritmos avanzados"

Algoritmos a partir de propiedades matemáticas

- **Algoritmo de Euclides**
  - Máximo común divisor de dos números naturales
  - Hay dos versiones:

$$mcd1(m,n) = \begin{cases} n & \text{si } m = 0 \\ mcd1(n,m) & \text{si } m > n \\ mcd1(m, n-m) & \text{si } (m \leq n) \text{ y } (m \neq 0) \end{cases}$$

$$mcd2(m,n) = \begin{cases} n & \text{si } m = 0 \\ mcd2(n\,\%m, m) & \text{si } m \neq 0 \end{cases}$$

Algoritmo de Euclides - Demostración *mcd*1

$$mcd1(m,n) = \begin{cases} n & \text{si } m = 0 \\ mcd1(n,m) & \text{si } m > n \\ mcd1(m,n-m) & \text{si } (m \leq n) \text{ y } (m \neq 0) \end{cases}$$

- Supongamos que $m \leq n$ (en caso contrario el propio algoritmo intercambia los parámetros)
- $m = az$, $n = bz$, con $a \leq b$
  - z son los factores primos (máximo común múltiplo)
  - a y b no comparten factores primos
- Hacemos $b = a + c$
- $n = bz = (a + c)z = (a_1 \cdots a_k + c_1 \cdots c_l)z$
  - No se puede sacar factor común. Si se pudiera sería otro factor de z.
  - a, c no comparten primos
- Por tanto, dado que $z = mcd1(az, bz) = mcd1(az, cz)$, y $c = b - a$ tenemos:

$$mcd1(m,n) = mcd1(az,bz) = mcd1(az,cz) = mcd1(m,n-m)$$

Algoritmo de Euclides - Demostración *mcd2*

$$mcd2(m,n) = \begin{cases} n & \text{si } m = 0 \\ mcd2(n\,\%m, m) & \text{si } m \neq 0 \end{cases}$$

- Supongamos que $m \leq n$ (en caso contrario el propio algoritmo intercambia los parámetros)
- $m = az$, $n = bz$, con $a \leq b$
  - z son los factores primos
  - a y b no comparten factores primos
- Hacemos $b = ac + d$, donde c y d son el cociente y el resto de b/a
- a y d no pueden compartir factores, de lo contrario serían también factores de b
- Por tanto:

$$z = mcd2(az, bz) = mcd2(dz, az) \Rightarrow mcd2(m,n) = mcd2(n\,\%m, m)$$

Introducción a las clases de complejidad computacional

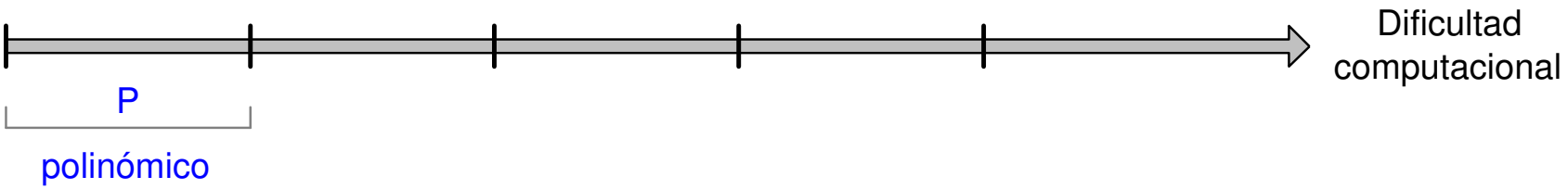
Eficiencia en tiempo y espacio

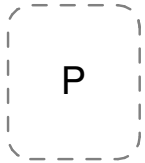
- Escogeremos algoritmos con menor *complejidad computacional*
  - Coste polinómico
    - 1 (constante), $\log n$, n, $n \log n$, $n^2 \ldots$
  - Coste mayor que polinómico
    - 2^n, $n!$, $n^n \ldots$
- Notación también usada para problemas computacionales
  - Generalmente, para cualquier instancia de un tamaño dado

Clases de complejidad

- Los problemas (de decisión) se pueden categorizar en *clases de complejidad* según su dificultad computacional
- Clases más importantes:
  - P: resolubles en tiempo polinómico
  - NP: resolubles en tiempo polinómico mediante un *algoritmo no-determinista*
  - NP-completos: Los más difíciles de NP
  - NP-duros: Tan difícil o más que el problema más difícil en NP
- Otras:
  - EXP: resolubles en tiempo exponencial
  - R: resolubles en tiempo finito

P: computable en tiempo polinómico

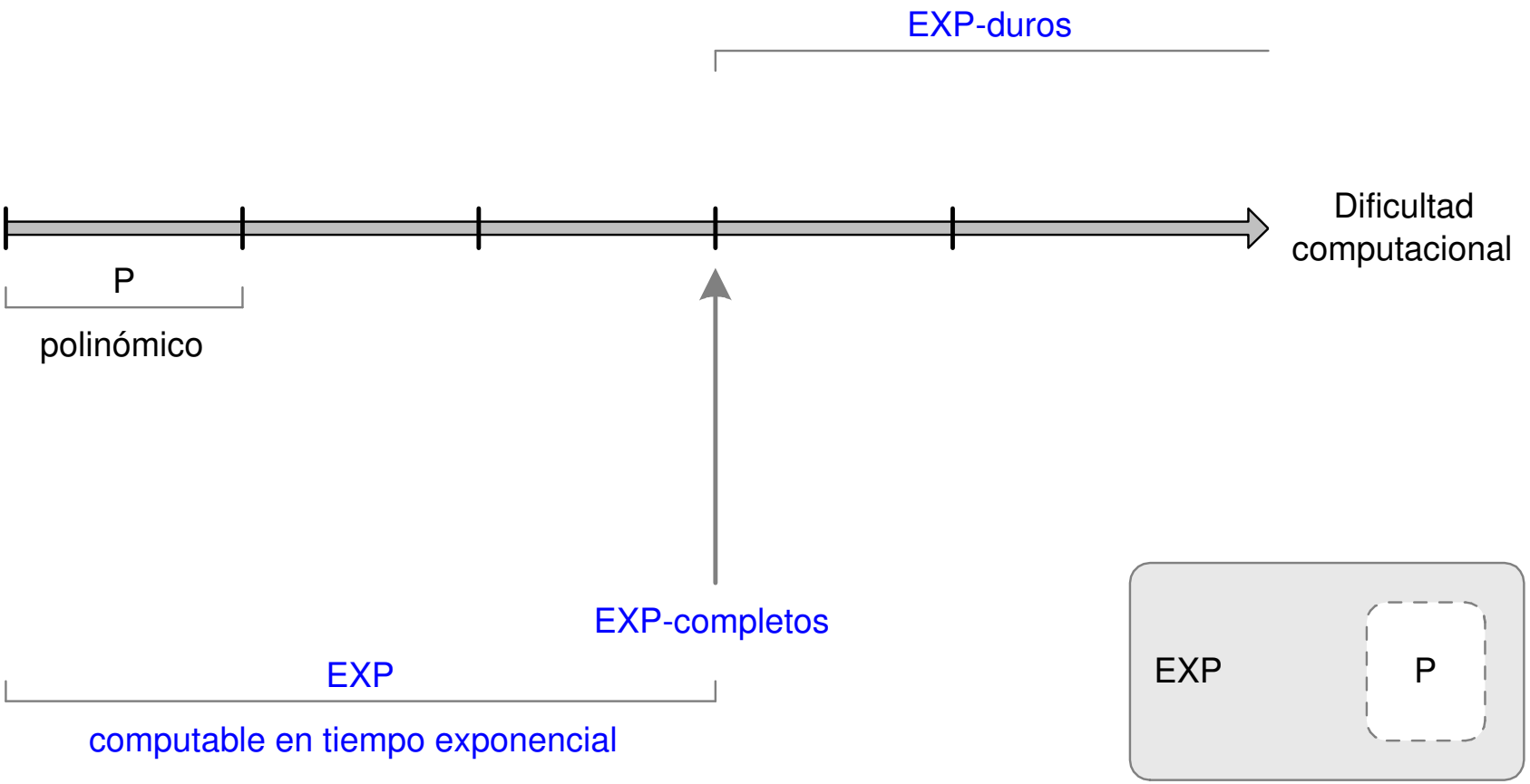




P: computable en tiempo polinómico

- $\mathcal{O}(p(n))$
  - $p(n)$: polinomio (de variable n)
- Ejemplos:
  - Búsqueda
  - Ordenación
  - Multiplicación de matrices
  - Sistemas de ecuaciones lineales
  - Algoritmo de Dijkstra (caminos más cortos)
  - Problemas de programación lineal
  - Problemas de optimización convexa

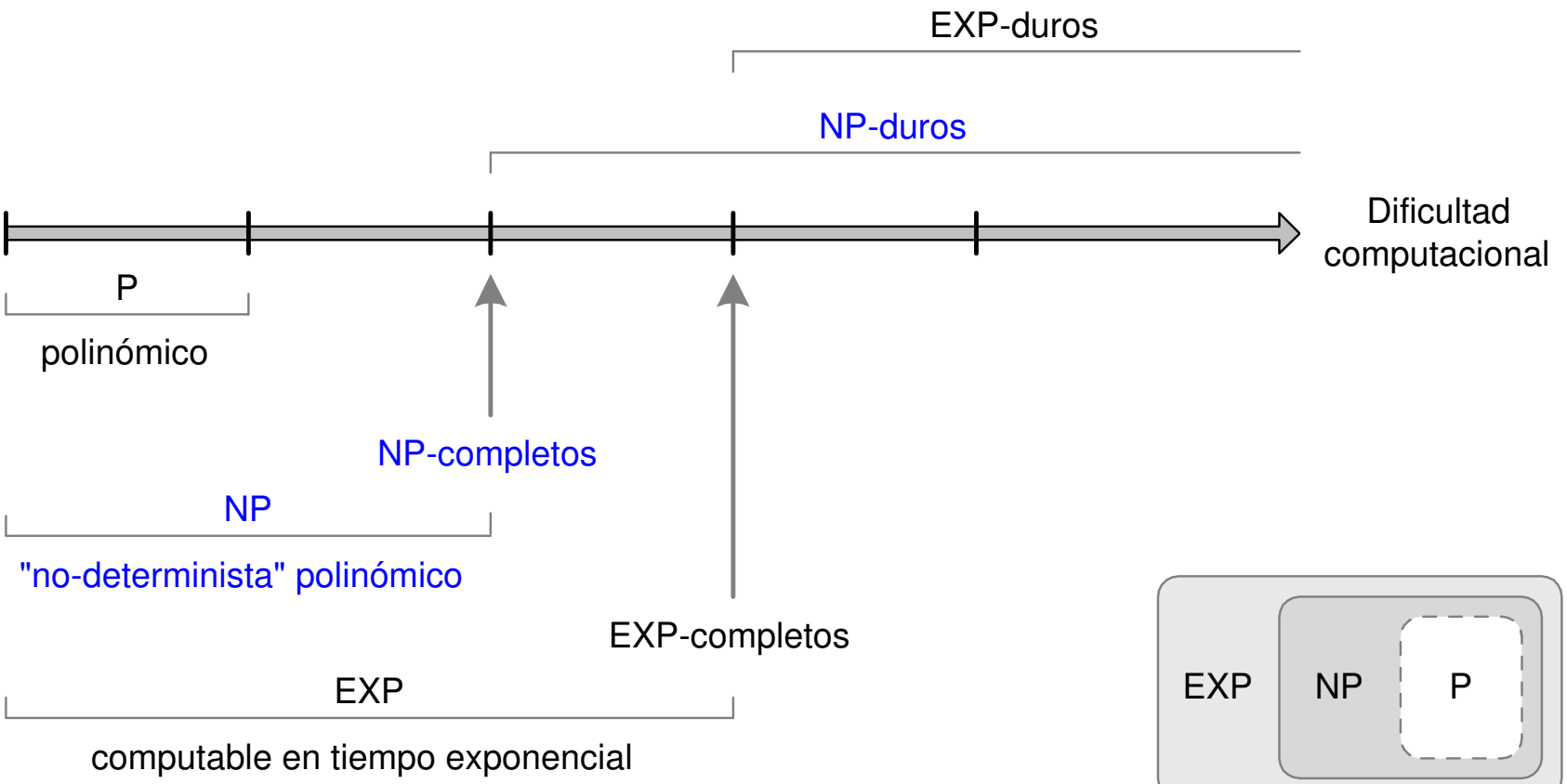
EXP: computable en tiempo exponencial



EXP: computable en tiempo exponencial

- $\mathcal{O}(2^{p(n)})$
  - $p(n)$: polinomio (de variable n)
- Ajedrez en tablero general $n \times n$. Juegan las blancas ¿Ganan?
  - $\in$ EXP, $\notin$ P
  - $\in$ **EXP-completo**
    - Tan difícil como cualquier otro problema $\in$ EXP
- Damas, Go (generales)
- **EXP-duro**: Tan difícil o más que el problema más difícil en EXP

NP: tiempo polinómico no determinista



¿P = NP?

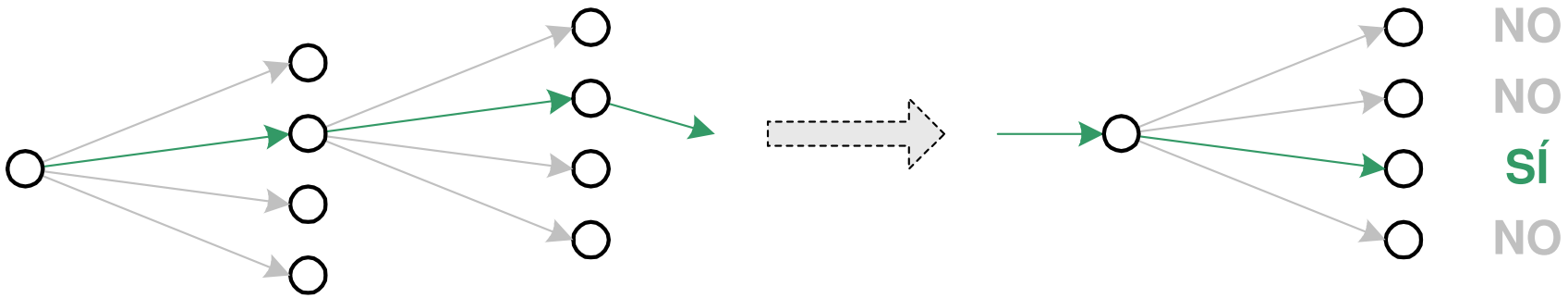
Problema del milenio

NP: tiempo polinómico no determinista

- *Nondeterministic polynomial time*
  - ¡¡¡NP no significa "tiempo no polinómico"!!!
- **NP**: Problemas de decisión que pueden ser resueltos en tiempo *polinómico* por un algoritmo *no determinista* "con suerte"
- **NP-completo**: Los problemas más difíciles de NP
- **NP-duro**: Tan difícil o más que el problema más difícil de NP

Algoritmo no determinista "con suerte"

- Toma decisiones aleatorias (adivina)
  - ¡¡Siempre acierta!!
- Si el resultado del problema de decisión es **SÍ** (o verdadero) las decisiones aleatorias conducen a obtener ese **SÍ**

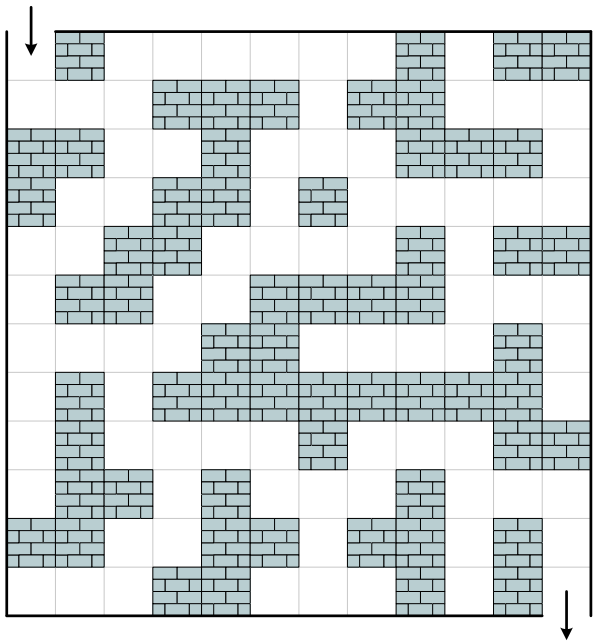


Ejemplo: búsqueda no determinista

```
1  Algoritmo busquedaNoDeterminista(A,n,elemento)  # O(1)
2  {
3      j = escojeIndice();  # No determinista O(1)
4
5      if(A[j]==elemento)
6      {
7          print(j);
8          exito();
9      }
10     else
11     {
12         print(-1);
13         fracaso();
14     }
15 }
```

Ejemplo de problema en NP (y P)

- Problema: ¿es posible encontrar un camino por un laberinto de tamaño $n \times n$?
  - Se puede resolver en tiempo polinómico por algoritmo no determinista con suerte $\Rightarrow$ el problema $\in$ NP
- ¿Se puede resolver también por un algoritmo determinista en tiempo polinómico?
  - En ese caso el problema $\in$ P
  - Recordad: P $\subseteq$ NP
  - Este problema sí está en P (Dijkstra)



NP: verificación de soluciones

- Definición equivalente de clase NP:
  - Un problema de decisión $\in$ NP si se puede verificar, en tiempo polinómico, si una solución conduce a un **SÍ**
- Problema del laberinto $n \times n$
  - Podemos verificar, en tiempo polinómico, que una secuencia de pasos (de longitud $\leq n^2$) es un camino a través del laberinto

Ejemplos de problemas en NP

- Tetris
  - Te dan una lista de piezas en orden
  - ¿Podrías sobrevivir?
  - Claramente se podría resolver en tiempo exponencial
- Factorización de un entero en producto de números primos
  - Fundamental en criptografía
- Problema (de decisión) del viajante de comercio
  - ¿Es posible obtener una solución con valor menor que una cierta cantidad?

Ejemplos de problemas en NP-completo

- Satisfacibilidad booleana (SAT)
  - Primer problema demostrado $\in$ NP-completo (Cook, 1971)
  - Ejemplo: ¿existen valores booleanos para x_1, x_2, x_3 y x_4 tal que $(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3 \vee \neg x_4)$ sea cierto?
- 21 problemas NP-completos de Karp (1972)
- Suma de subconjuntos
  - Dado un conjunto de enteros, ¿existe algún subconjunto cuya suma sea exactamente cero?
- Problemas de optimización
  - Problema del viajante de comercio
  - Problema de la mochila 0/1

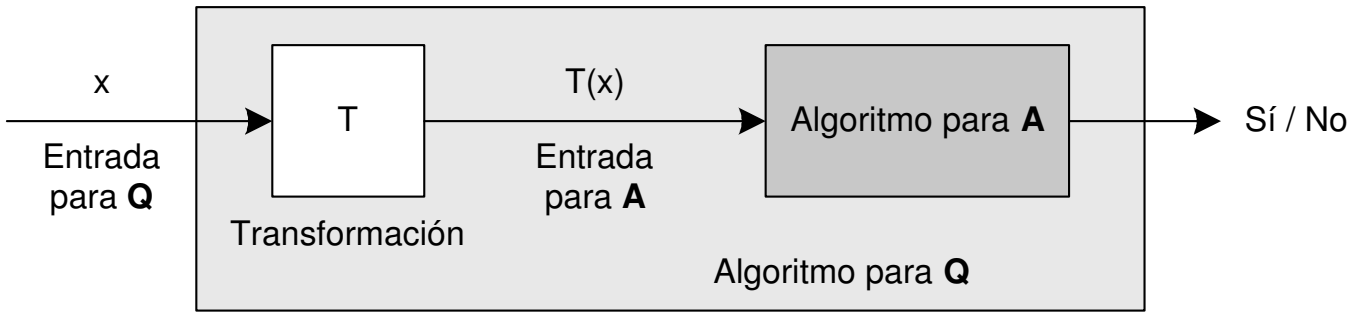
Importancia de conocer problemas NP-completos

- Parecen inocentes
  - A veces se parecen mucho a problemas para los que sí que existen algoritmos eficientes
- Aparecen frecuentemente
- Soluciones aproximadas
  - Un programador no experimentado puede perder mucho tiempo intentando diseñar un algoritmo eficiente sin éxito
  - Si demuestras que tu problema es NP-completo (o NP duro) puedes recurrir a soluciones aproximadas

Reducciones

- ¿Cómo demostrar que un problema **A** es NP-completo?
  1. Demostrar **A** $\in$ NP
  2. Demostrar que es NP-duro
     - Lo más habitual: "Reducir" un problema NP-completo **Q** a **A**
     - Si **Q** NP-completo hay que demostrar que **A** es tan difícil o más que **Q**

- Reducción
  - En general: transformar cada instancia de un problema **Q** a otra de un problema **A**
  - Al resolver **A** el resultado es el mismo que para **Q**



  - Cada problema NP-completo se puede reducir a los demás

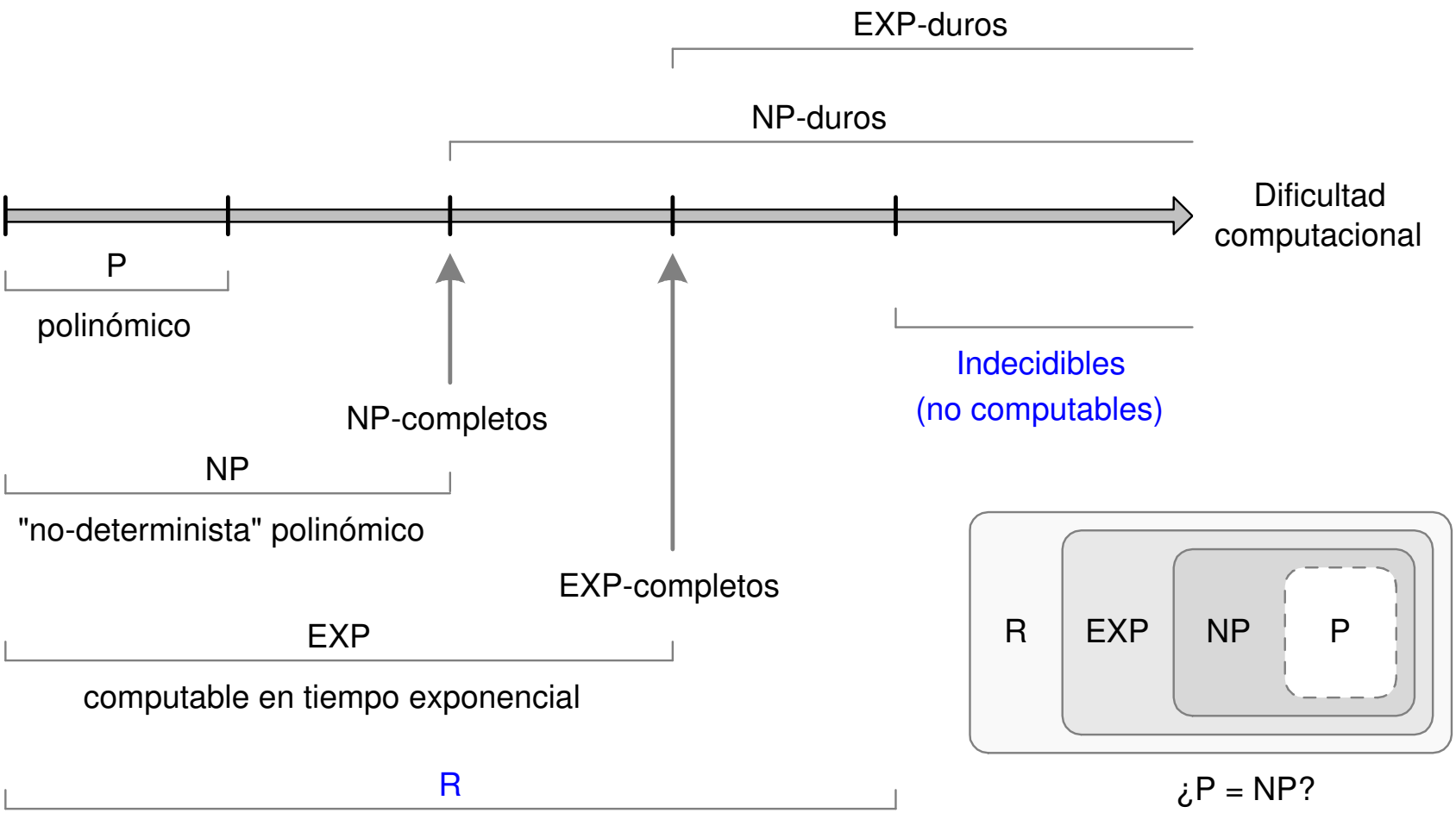
Reducciones

- Técnica de diseño de algoritmos
  - "Transforma y vencerás"
  - ¿Se puede transformar un problema en otro que sepamos resolver eficientemente?
- Ejemplos
  - Camino más corto por grafo no ponderado
    - Camino más corto por grafo ponderado (p.e., algoritmo de Dijkstra), dando el mismo peso a todas las aristas
  - Camino más corto considerando productos de pesos
    - Aplicar logaritmos a los pesos
  - Camino más largo
    - Multiplicar los pesos por -1

Conjetura ¿P = NP?

- Uno de los problemas matemáticos del milenio no resuelto
  - \$1.000.000 (*Clay Mathematics Institute*)
  - ¡¡Poca cantidad!!
  - Si existiera (se encontrara) un algoritmo polinómico para resolver un problema NP-completo $\Rightarrow$ P = NP
- Opinión mayoritaria: P $\neq$ NP
- P $\neq$ NP equivale a:
  - No puedes implementar la "suerte" al adivinar
  - Hay problemas para los que obtener una solución es más difícil que verificarla

R: computable en tiempo finito



Problemas no computables

- R: computable en tiempo finito
- R = "recursivo" (no en el sentido de función recursiva que se llama a sí misma)
- Problemas no computables $\equiv$ indecidibles $\equiv$ no "recursivos"
  - No existen algoritmos que los resuelvan
  - Ejemplo más conocido: problema de la parada
    - Determinar si *cualquier* programa va a parar en tiempo finito

Problema de la parada

```
1 def Termina(p, x):  # p es un programa, x sus entradas
2     # Supongamos que resuelve el problema de la parada
3     # Devuelve True si p(x) termina, y False en otro caso
```

```
1 def d(w):  # w es un programa
2     if Termina(w, w):
3         while True: pass # Bucle infinito
```

d(w) termina $\Leftrightarrow$ w(w) nunca termina

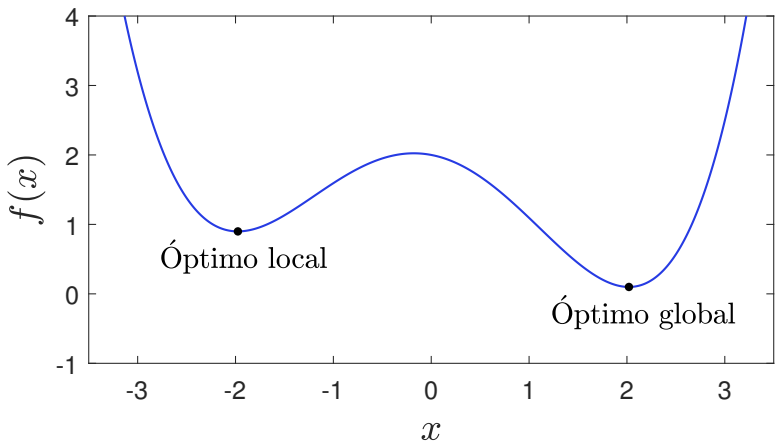
```
1 d(d):
2     if Termina(d, d):
3         while True: pass
```

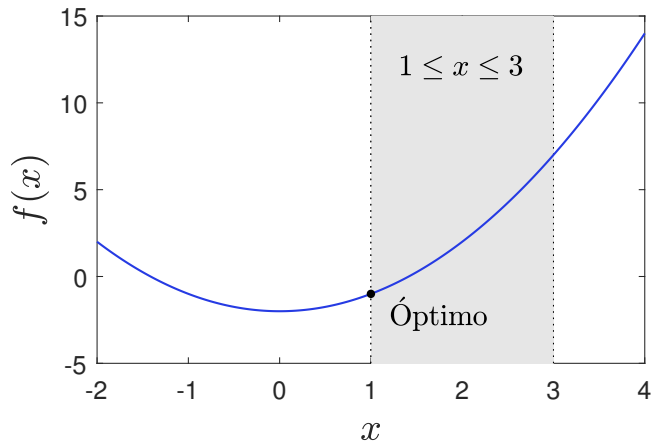
d(d) termina $\Leftrightarrow$ d(d) nunca termina ¡Contradicción!

Problemas de optimización

Problemas matemáticos de optimización

- Importante clase de problemas computacionales
- Maximizar/minimizar una *función objetivo*
  - Encontrar las variables ("parámetros") donde una función toma su mayor/menor valor
  - Las variables deben satisfacer *restricciones* que definen la *región factible* donde se encontrará la solución

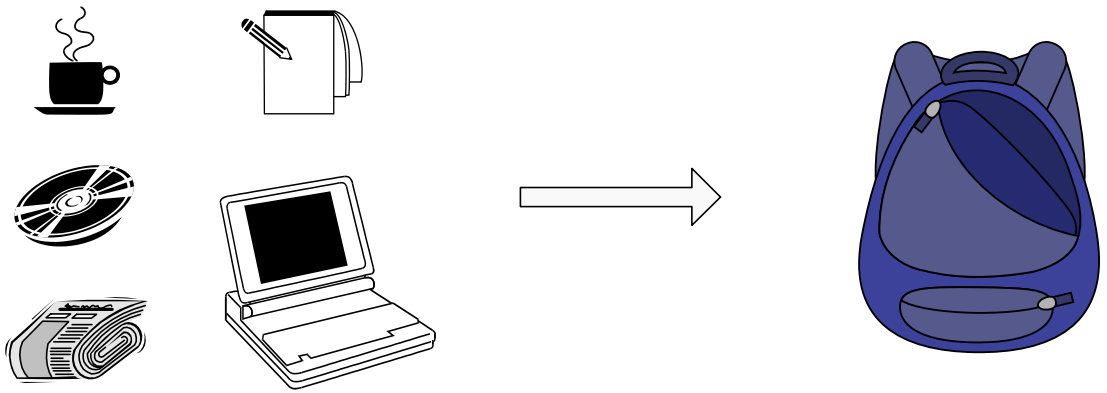




Ejemplo

Problema de la mochila 0/1

- Considérese un conjunto de n objetos, cada uno con un peso p_i y un valor v_i, para $i = 1, \ldots, n$, y una mochila con capacidad C, que es el máximo peso que puede soportar la mochila.
- Seleccionar el subconjunto de objetos que puedan introducirse en la mochila, sin sobrepasar la capacidad C, tal que la suma de sus valores sea máxima.



Notación matemática

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimiza}} \quad & f_0(\mathbf{x}) \\ \text{sujeto a} \quad & f_i(\mathbf{x}) \leq b_i, \quad i = 1, \ldots, m \end{aligned}$$

- f_0 es la *función objetivo*
- $\mathbf{x}$ representa varias variables
  - $\mathbf{x}$ no tiene por qué ser un escalar
- $f_i(\mathbf{x}) \leq b_i, \quad i = 1 \ldots, m$ son las *restricciones*
  - El propósito es hallar los valores de $\mathbf{x}$ que optimicen la función
  - Un problema de maximización se puede convertir fácilmente a uno de minimización: cambiando el signo de f_0

Ejemplo

Problema de la mochila 0/1

$$\begin{aligned} \underset{\mathbf{x}}{\text{maximiza}} \quad & \sum_{i=1}^{n} x_i v_i \\ \text{sujeto a} \quad & x_i \in \{0,1\}, \quad i = 1, \ldots, n \\ & \sum_{i=1}^{n} x_i p_i \leq C \end{aligned}$$

- Las variables x_i determinan si se introduce el objeto i en la mochila (x es un vector)
- C, p_i, v_i son datos del problema (indican una instancia concreta)
- Es un problema NP-completo

Problemas lineales

- $f_0, \ldots, f_m$ son **funciones lineales**:

$$f_i(\alpha \mathbf{x}_1 + \beta \mathbf{x}_2) = \alpha f_i(\mathbf{x}_1) + \beta f_i(\mathbf{x}_2)$$

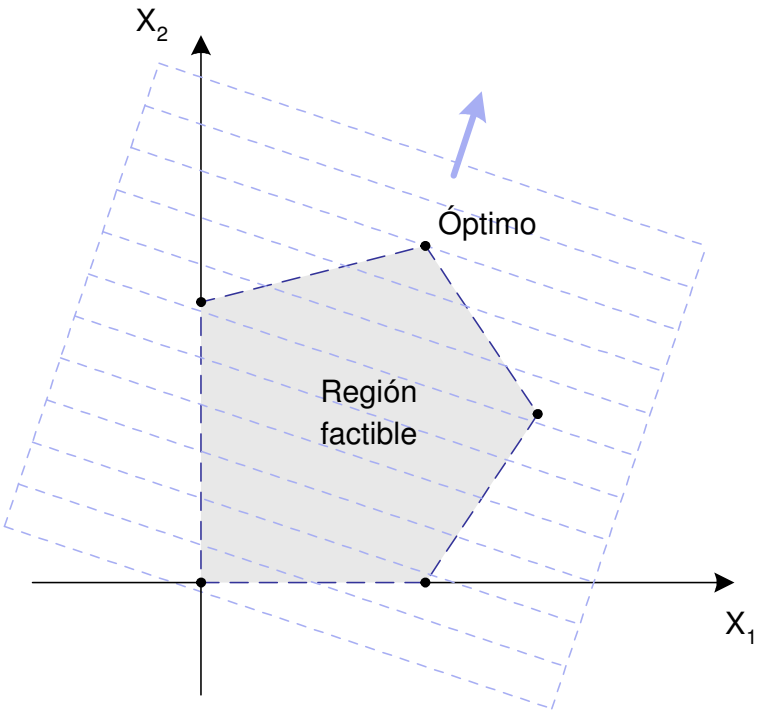
$$\forall\ \mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$$

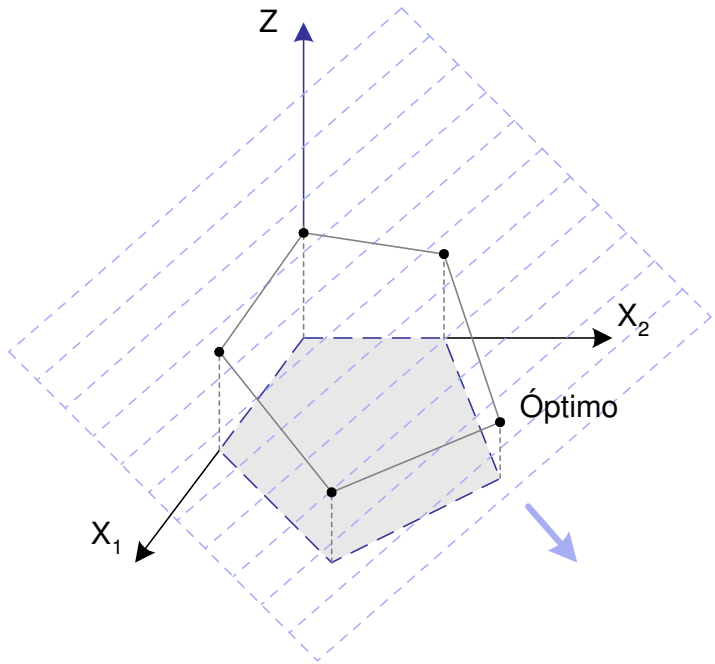
$$\forall\ \alpha, \beta \in \mathbb{R}$$

- No lineal, si no cumple lo anterior
- Los problemas lineales suelen llamarse "problemas de *programación lineal*"
- Sos estudiados en profundidad por la *Investigación Operativa*

Problemas lineales – interpretación geométrica

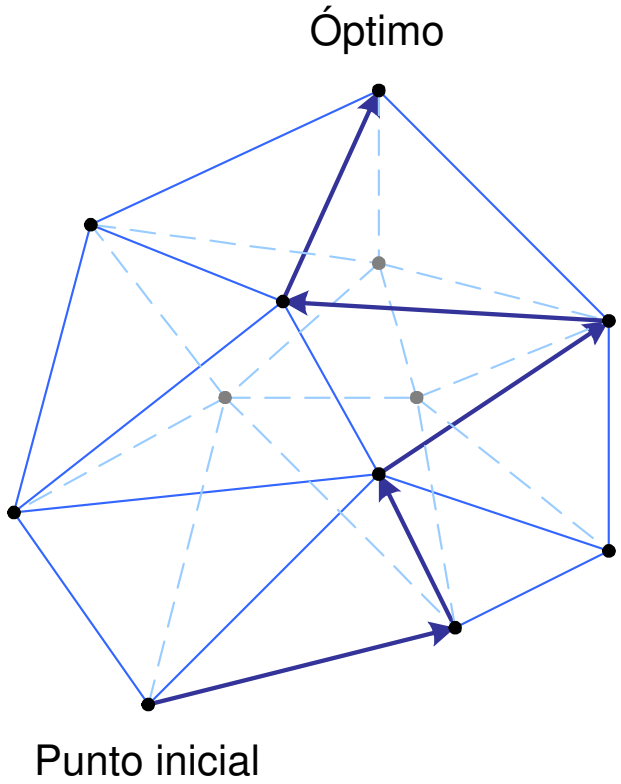
- Función objetivo: *hiperplano*
- Restricciones: también *hiperplanos*
  - Definen una región factible (donde debe estar la solución)





Problemas lineales – algoritmo Simplex

- Parte de un punto inicial
  - Vértice de región factible (poliedro convexo)
- Avanza por aristas hasta un vértice mejor
  - Hasta encontrar el óptimo
- Hay otros algoritmos para resolver problemas lineales



Problemas convexos

- $f_0, \ldots, f_m$ son **funciones convexas**:

$$f_i(\alpha \mathbf{x}_1 + \beta \mathbf{x}_2) \leq \alpha f_i(\mathbf{x}_1) + \beta f_i(\mathbf{x}_2)$$

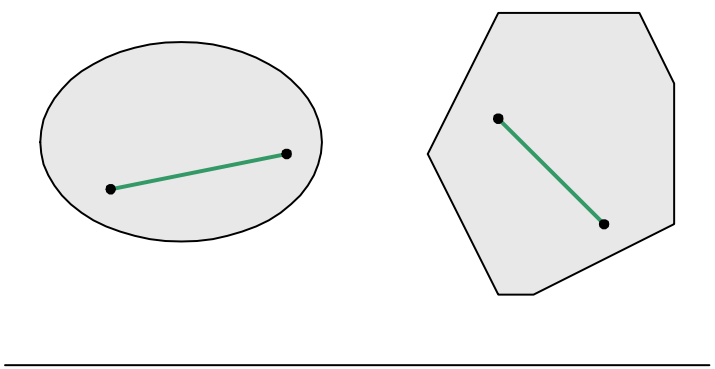
$$\forall\ \mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$$

$$\forall\ \alpha, \beta \in [0, 1],\ \text{donde}\ \ \alpha + \beta = 1$$

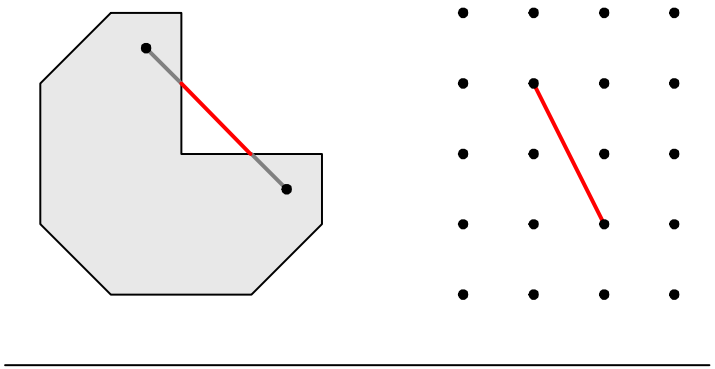
- Las restricciones definen un **conjunto convexo**
- No convexo, si no cumple lo anterior
- Los problemas lineales son convexos
- Se pueden resolver en **tiempo polinómico**

Conjuntos convexos

- Cualquier segmento entre dos puntos del conjunto debe estar completamente contenido en el conjunto

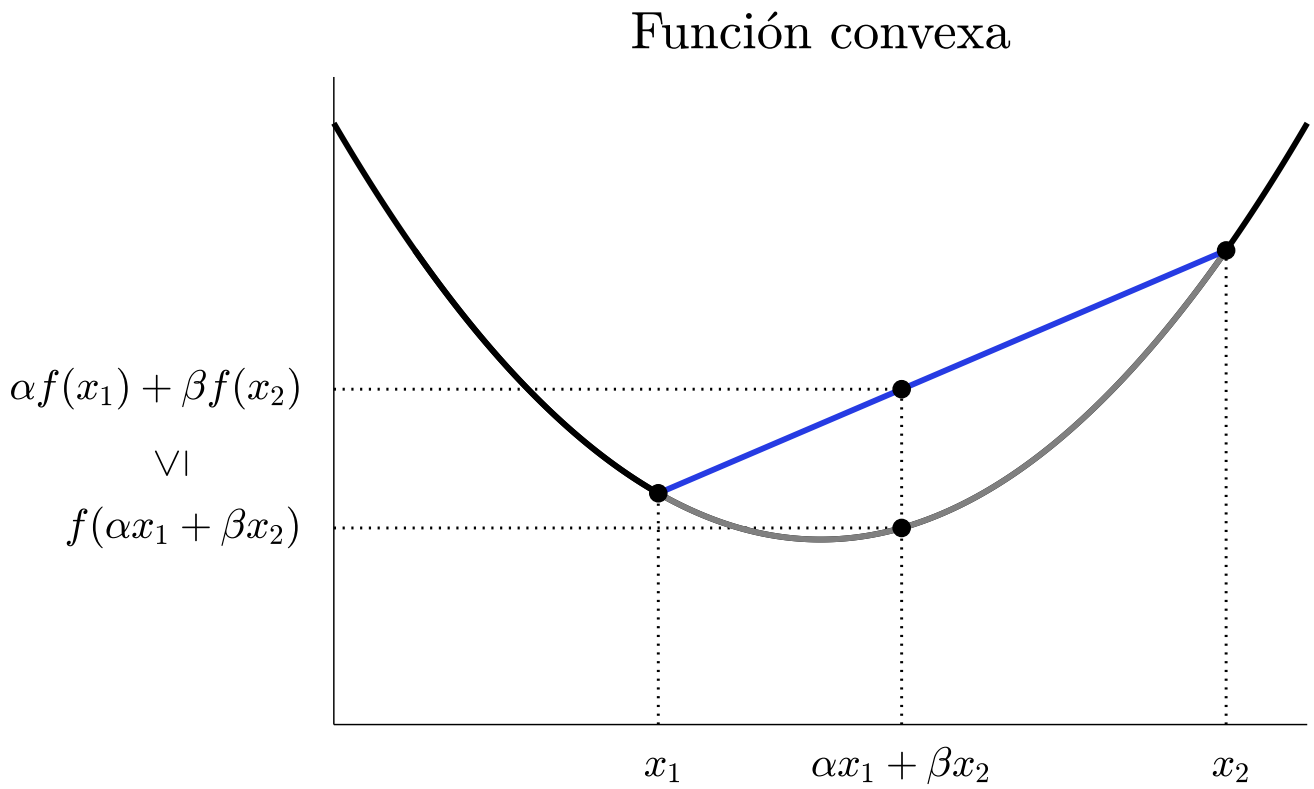


Conjuntos convexos



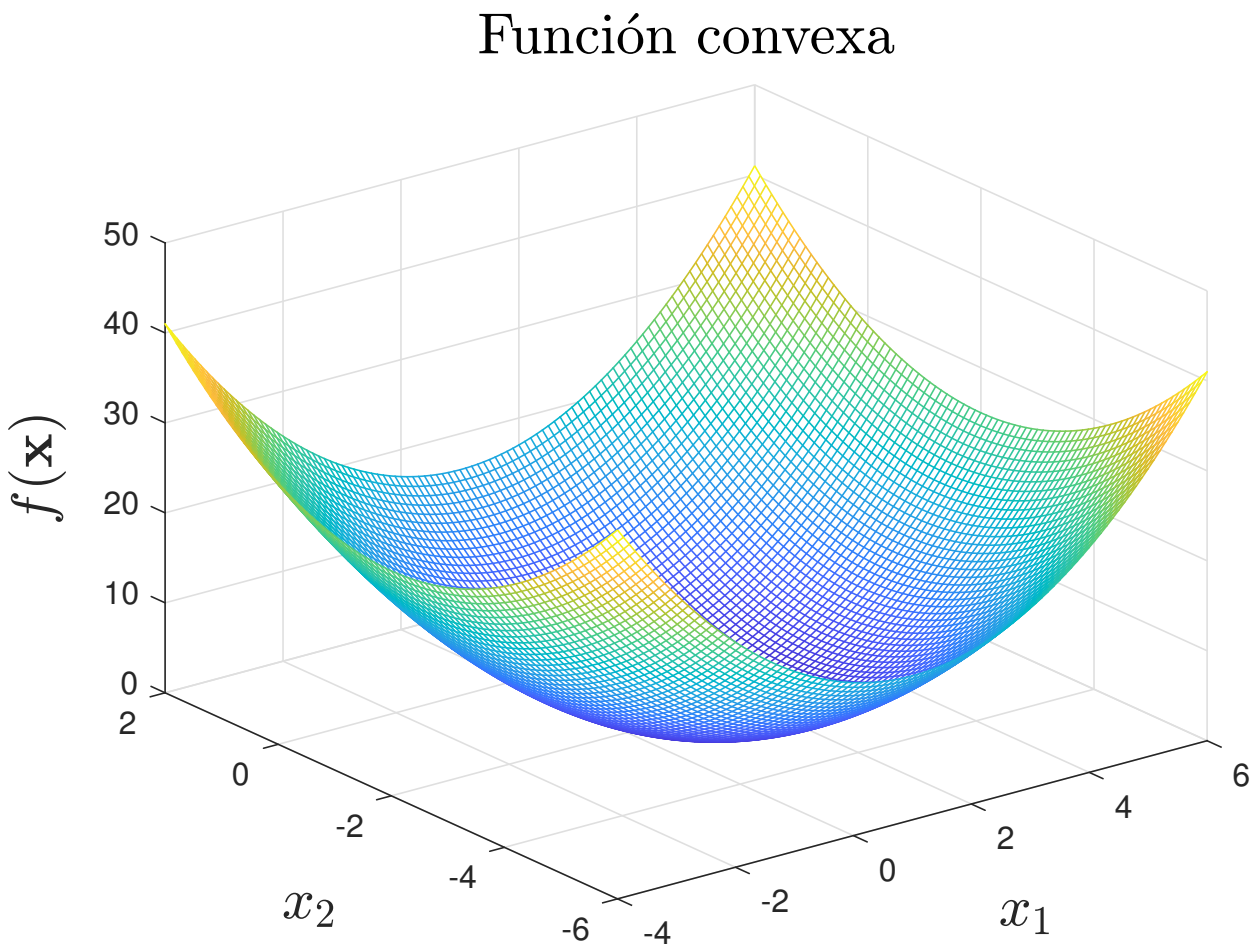
Conjuntos no convexos

Funciones convexas

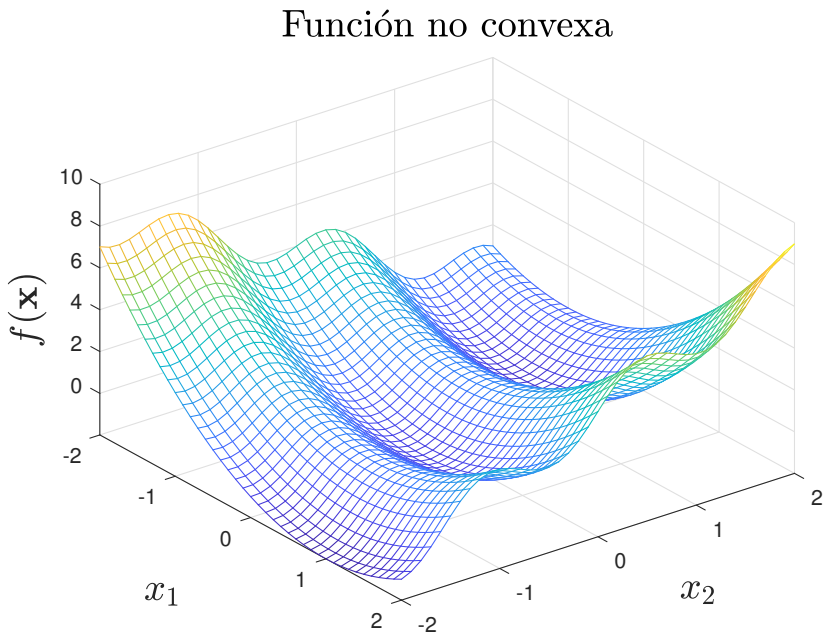


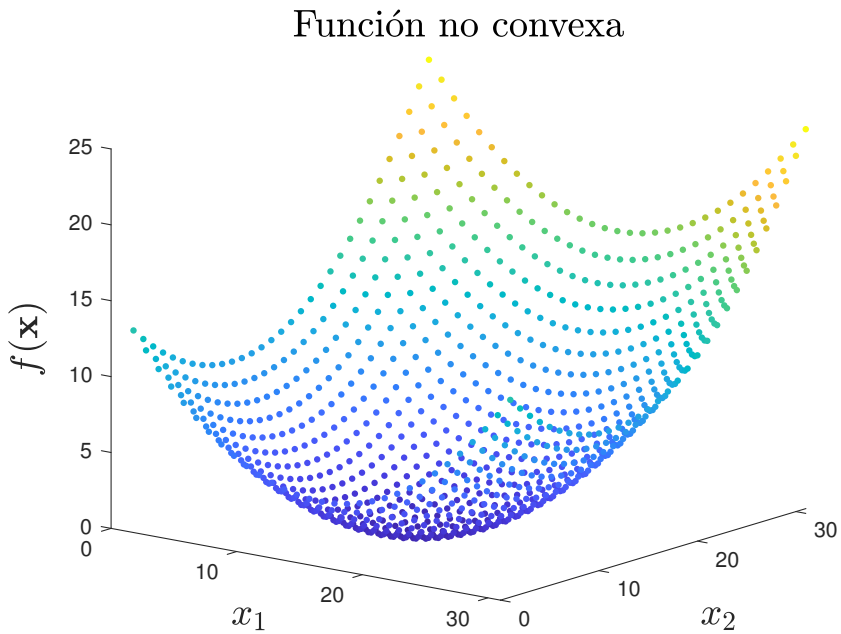
$$f_i(\alpha \mathbf{x}_1 + \beta \mathbf{x}_2) \leq \alpha f_i(\mathbf{x}_1) + \beta f_i(\mathbf{x}_2)$$

Funciones convexas



Funciones no convexas





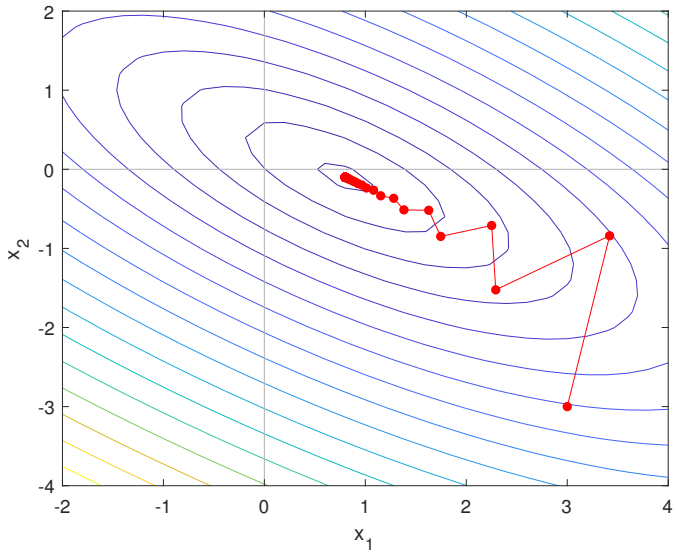
- Pueden tener múltiples óptimos locales
- Si el dominio es discreto no tiene sentido hablar de una función convexa

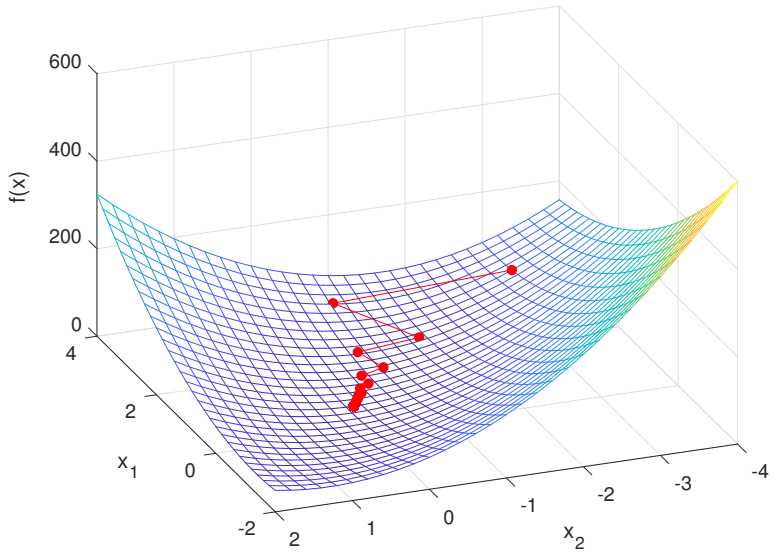
Funciones no convexas

- Tienen varios óptimos locales
- No siempre se pueden resolver en tiempo polinómico
- Para "resolverlos" se recurre a *heurísticas*, pero no garantizan llegar al óptimo global
  - Búsqueda aleatoria
  - Algoritmos genéticos
- Teorema "no hay comida gratis": ninguna heurística es mejor que todas en general

Descenso/ascenso de gradiente

- **Gradiente**: vector de derivadas parciales de una función
- Indica la dirección de mayor pendiente en un punto
- Algoritmo iterativo asciende o desciende poco a poco hasta llegar a un óptimo





Descenso/ascenso de gradiente – pseudocódigo

function AlgoritmoDescensoGradiente()

$\vec{x} \leftarrow \vec{x}_0$ ▷ Aproximación inicial
$\vec{g} \leftarrow$ calculaGradiente($\vec{x}$) ▷ Gradiente en $\mathbf{x}$

while $\|\vec{g}\| > \varepsilon$ **do** ▷ Iterar hasta que el módulo del ▷ gradiente sea muy pequeño

$\vec{x} \leftarrow \vec{x} - \alpha \cdot \vec{g}$ ▷ Actualizar aproximación en ▷ dirección del gradiente

$\vec{g} \leftarrow$ calculaGradiente($\vec{x}$) ▷ Actualizar gradiente

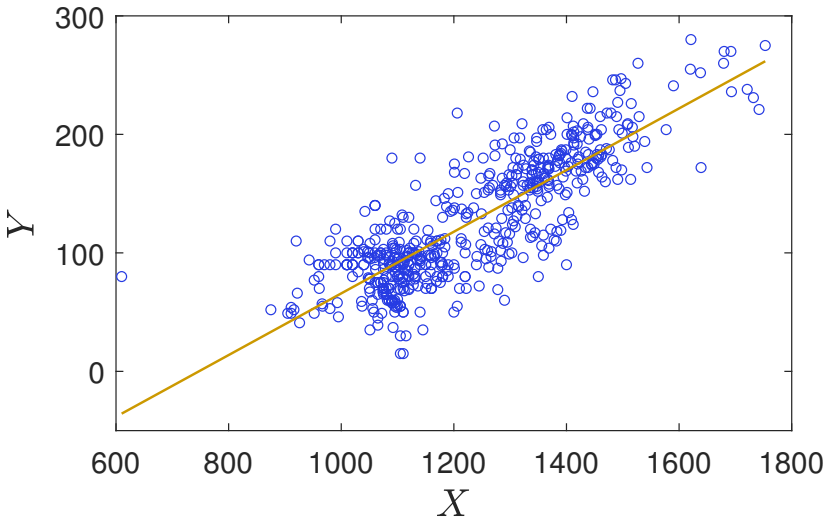
return $\vec{x}$ ▷ Devolver aproximación final

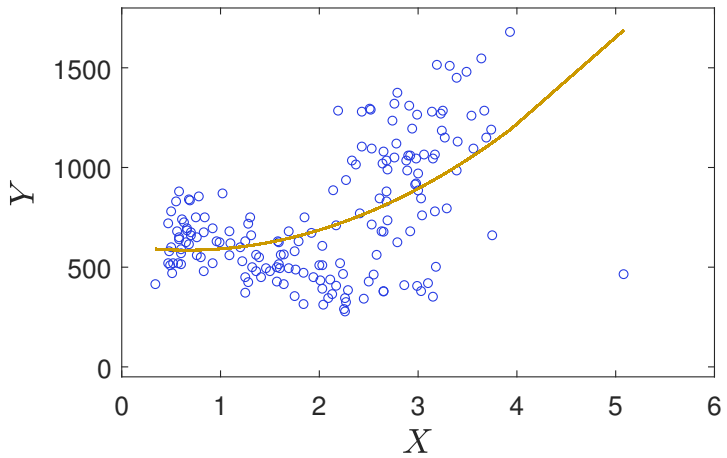
- Hay muchos más algoritmos (y mejores)
  - Puede no converger o ser muy lento

Introducción a problemas en Inteligencia Artificial, Aprendizaje Automático y Ciencia de Datos

Predicción - Regresión

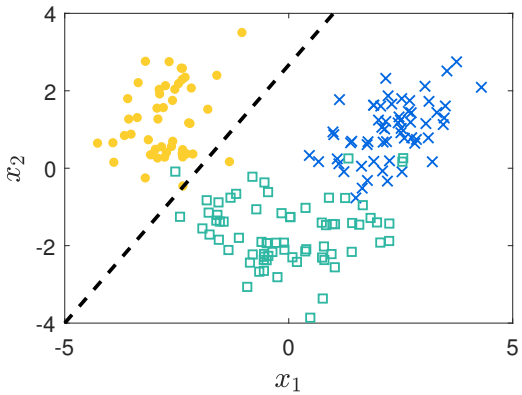
- Predices un valor real
- Generas un modelo matemático que se ajuste a los datos lo mejor posible
  - ¿Qué tipo de modelo?
  - ¿Mejor posible en qué sentido?

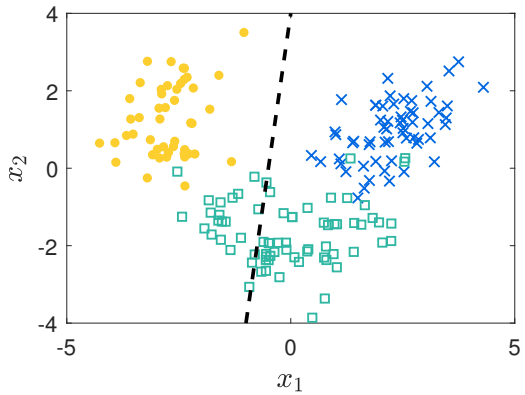


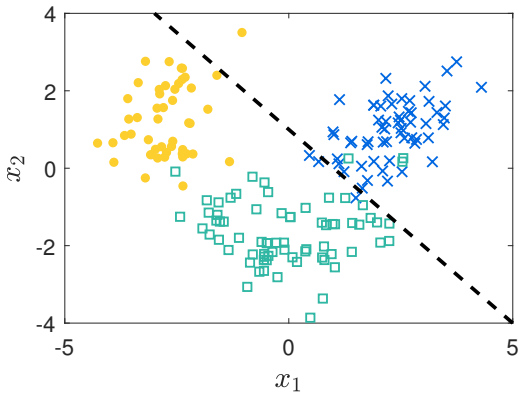


Predicción - Clasificación

- Predices una clase (categoría)
- Generas un modelo matemático que defina fronteras entre las clases en el espacio de los datos, para separar las clases lo mejor posible
  - ¿Qué tipo de modelo y fronteras?
  - ¿Mejor posible en qué sentido?







Predicción

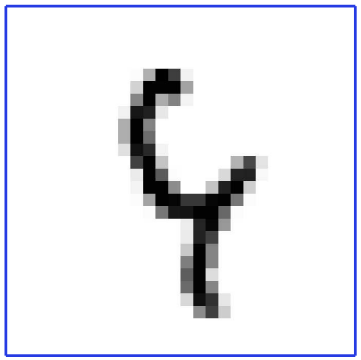
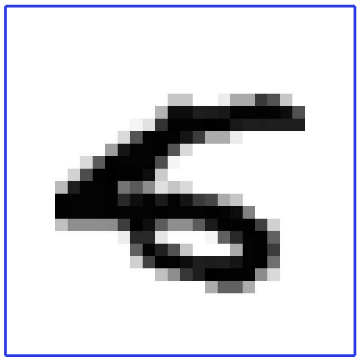
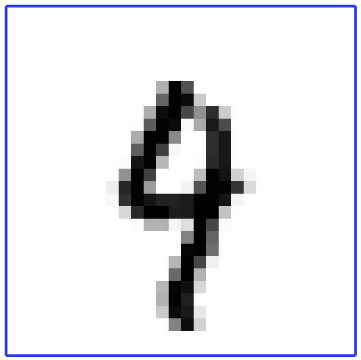
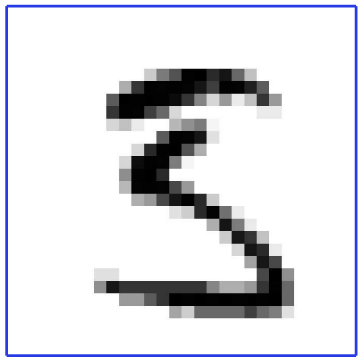
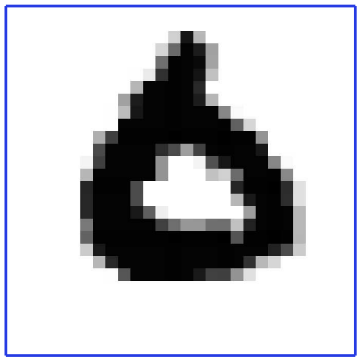
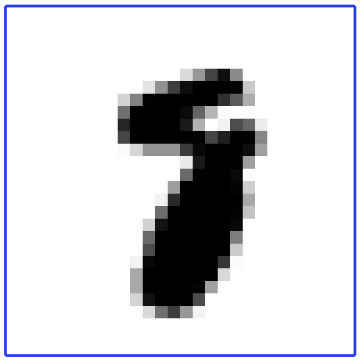
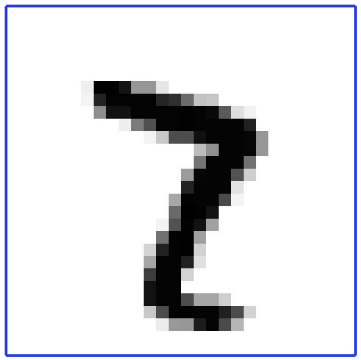
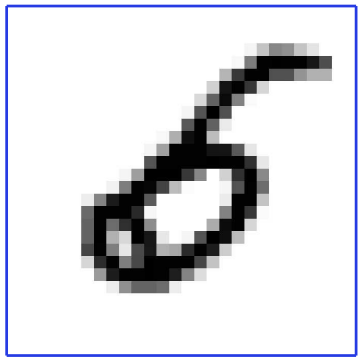
- Para estos "problemas" no hay una solución "correcta"
  - Los modelos se crean con unos datos disponibles (de entrenamiento)
  - El rendimiento a la hora de predecir depende de nuevos datos (de test)
- No se conoce la relación entre entradas y salidas completamente
  - No podemos hablar de un "problema computacional"
  - Especificarla (matemática o lógicamente) suele ser complicadísimo
  - No podremos elaborar un algoritmo

Aprendizaje automático

- Construir algoritmos a través de ejemplos
  - Caso concreto: reconocimiento de dígitos escritos a mano:

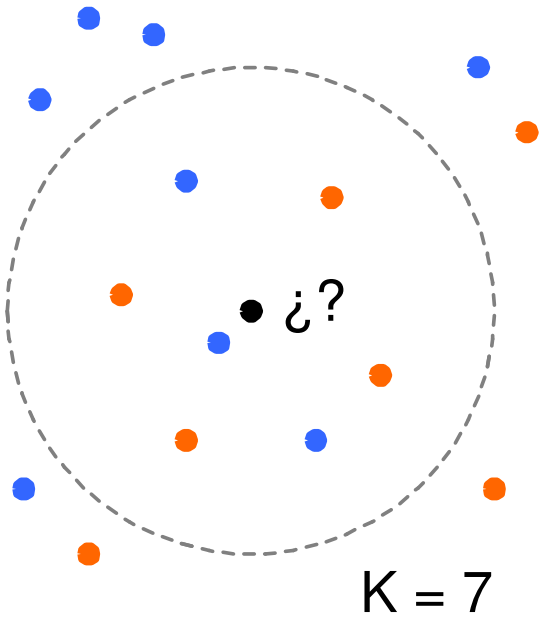
Base de datos MNIST

Dígitos difíciles

Técnicas: problemas computacionales concretos

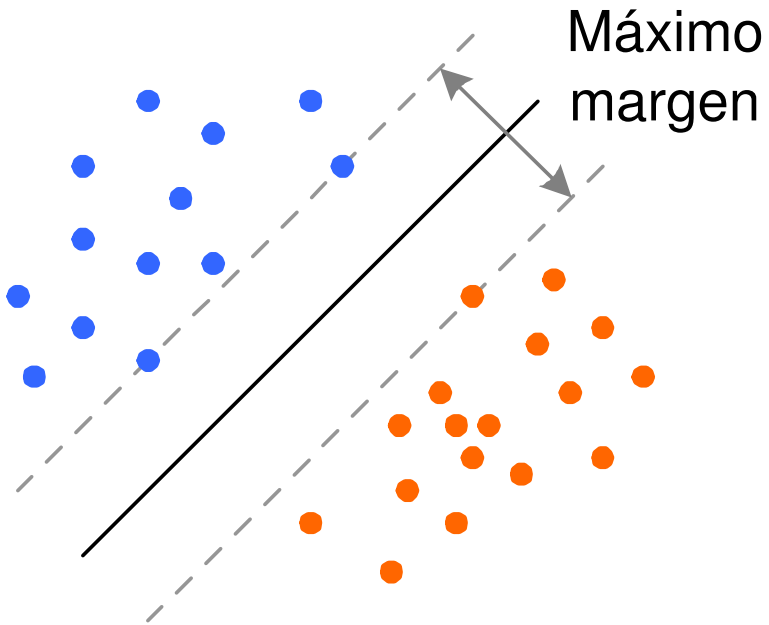
- Enfoque 1:



K vecinos más cercanos

K-NN (K - *nearest neighbors*)

- Enfoque 2:



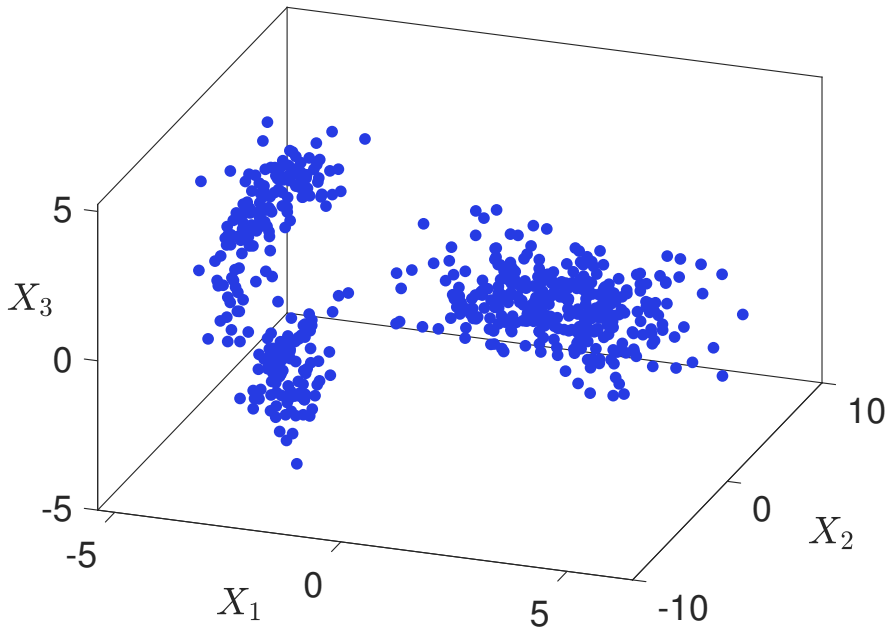
Máquinas de vectores soporte

SVM (*support vector machines*)

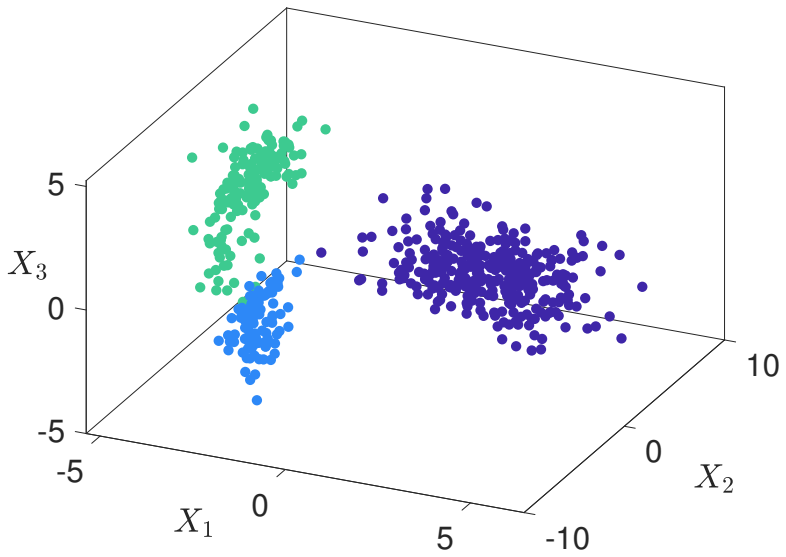
- Hay muchos más enfoques

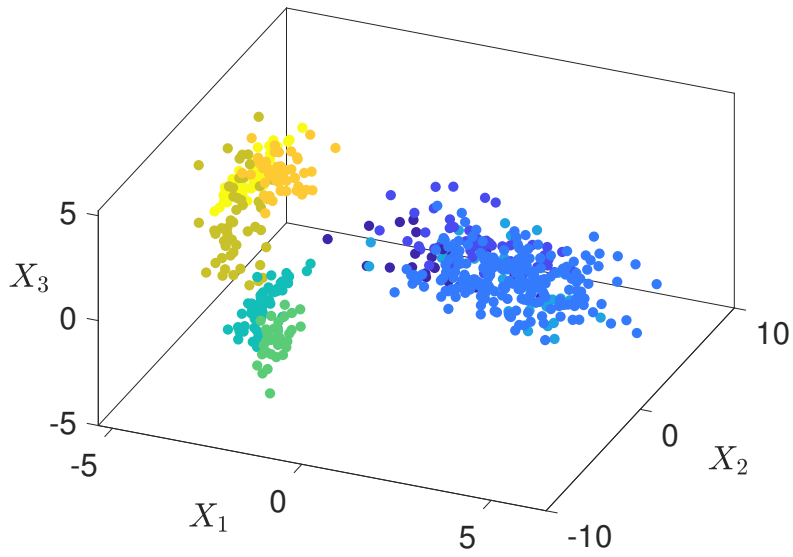
Clustering – agrupamiento

- La única información con la que contamos es la posición de los puntos (no hay información sobre clases)
- Queremos encontrar una agrupación o partición "razonable"



Posibles agrupaciones

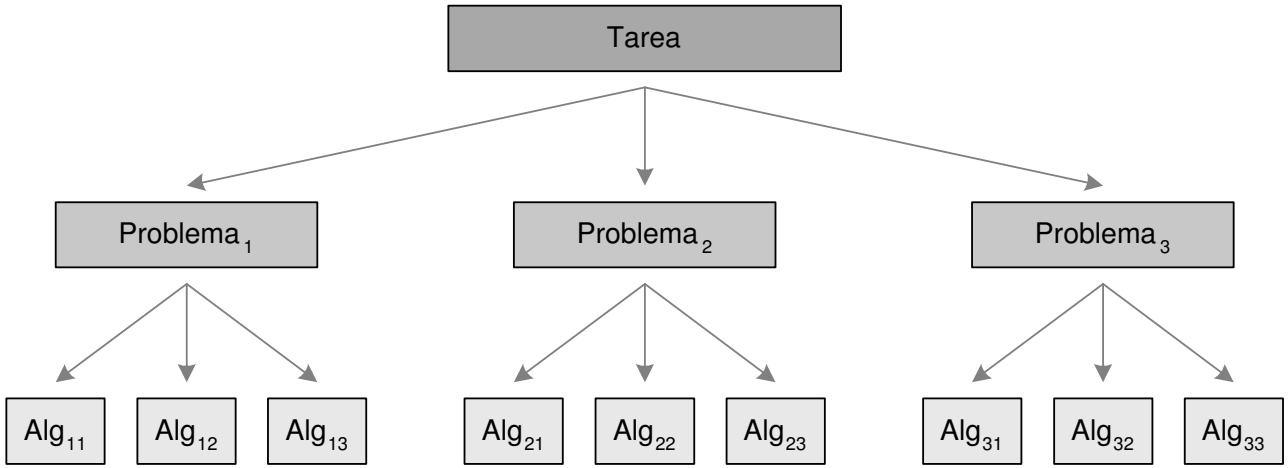




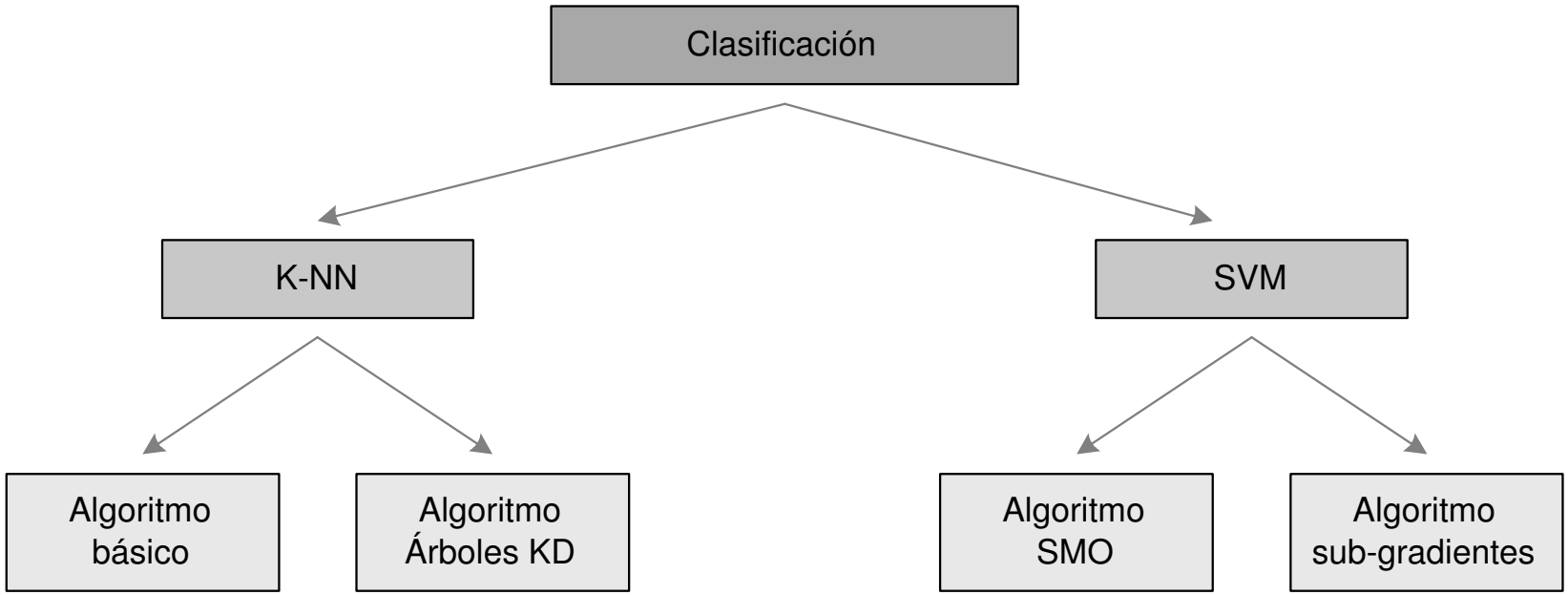
- ¿Cuál es la agrupación correcta? No se sabe...
  - La tarea no está claramente definida
  - Aunque existen medidas de bondad para evaluar la calidad de una agrupación

Tareas – Problemas – Algoritmos

- Existen "**tareas**" generales en ciencias e ingenierías que no se pueden definir como problemas computacionales
- Para abordarlas se plantean **problemas** computacionales concretos
- Los problemas computacionales se resuelven mediante **algoritmos**



Tareas – Problemas – Algoritmos: Clasificación



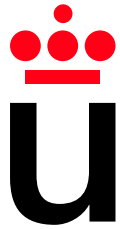
- Hay numerosos problemas y algoritmos alternativos
- Actualmente las redes neuronales artificiales profundas (*deep learning*) ofrecen el mejor rendimiento en muchas tareas

Tema 2.1
Preliminares matemáticos

Diseño y Análisis de Algoritmos

Manuel Rubio Sánchez

13 de mayo de 2024



Universidad
Rey Juan Carlos

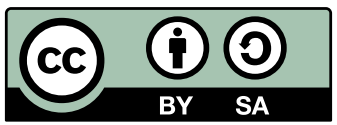
Copyright

©2024 Manuel Rubio Sánchez

Algunos derechos reservados.

Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en:

https://creativecommons.org/licenses/by-sa/4.0/deed.es.



Contenido

Fundamentos

Propiedades de potencias

- $b^1 = b$
- $b^0 = 1$
- $b^{-x} = 1/b^x$
- $b^x b^y = b^{x+y}$
- $(ab)^x = a^x b^x$
- $(b^x)^y = b^{xy} = (b^y)^x$

Propiedades de logaritmos

- Definición: $\log_b x = y \quad \Rightarrow \quad b^y = x$
  - $b > 0, b \neq 1, x > 0$

- $\log_b b = 1$
- $\log_b 1 = 0$
- $\log_b(x^y) = y \log_b x$
- $\log_b(b^x) = x$
- $b^{\log_b a} = a$
- $\log_b(xy) = \log_b(x) + \log_b(y)$
- $\log_b(x/y) = \log_b(x) - \log_b(y)$
- $\log_b x = \log_a x / \log_a b$
- $x^{\log_b y} = y^{\log_b x}$
- $\log_b a = 1/\log_a b$

Coeficientes binomiales

- El coeficiente binomial $\binom{n}{m}$, con $n \geq m \geq 0$, se puede definir de varias formas:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0 \text{ o } n = m \\ \frac{n!}{m!(n-m)!} & \text{en otro caso} \end{cases}$$

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0 \text{ o } n = m \\ \binom{n-1}{m-1} + \binom{n-1}{m} & \text{en otro caso} \end{cases}$$

  - Son enteros que aparecen en la expansión de $(1+x)^n$
  - Son el número de combinaciones de n elementos tomados de m en m (las formas de escoger m elementos de entre n, donde el orden no importa)

Vectores

- $\mathbf{x} = (x_1, \ldots, x_n)$, vector columna
- $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{x} \in \mathbb{C}^n$
- $\mathbf{0} = (0, 0, \ldots, 0)$
- $\mathbf{e}_j = (0, \ldots, \underbrace{1}_{j}, \ldots, 0)$
- *vector* se puede usar en términos más generales para denotar un elemento de un *espacio vectorial*, (por ejemplo, un vector podría ser un polinomio)

Matrices

- Matriz cero ($\mathbf{0}$)
- Matriz identidad ($\mathbf{I}$)
- Matriz diagonal
- Matriz triangular superior/inferior
- Matriz permutación
- Rango: rg($\mathbf{A}$), número de columnas o filas linealmente independientes de $\mathbf{A}$
- Traza: $\text{tr}(\mathbf{A}) = \sum_{i=1}^{n} a_{i,i}$, donde $\mathbf{A} \in \mathbb{R}^{n \times n}$
- Determinante ($|\mathbf{A}|$ o $\det(\mathbf{A})$)
  - $|\mathbf{ABC}| = |\mathbf{BCA}| = |\mathbf{CAB}|$

Matrices

- Matriz traspuesta ($\mathbf{A}^\top$)
  - $(\mathbf{ABC})^\top = \mathbf{C}^\top\mathbf{B}^\top\mathbf{A}^\top$
- Matriz simétrica: $\mathbf{A} = \mathbf{A}^\top$
- Matriz inversa: $\mathbf{A}^{-1}$, con $\mathbf{A}$ cuadrada, donde $\text{rg}(\mathbf{A}) = n$, $|\mathbf{A}| \neq 0$
- Matriz ortogonal: $\mathbf{Q}^\top\mathbf{Q} = \mathbf{Q}\mathbf{Q}^\top = \mathbf{I}$, con $|\det(\mathbf{Q})| = 1$
- Autovalores (λ) y autovectores ($\mathbf{v}$): $\mathbf{Av} = \lambda\mathbf{v}$, con $\mathbf{v} \neq 0$
  - Los autovalores son las raíces del *polinomio característico* $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$

Matrices - descomposiciones

- LU
  - $\mathbf{A} = \mathbf{PLU}$, con $\mathbf{P}$ matriz permutación, $\mathbf{L}$ triangular inferior y $\mathbf{U}$ triangular superior
- QR
  - $\mathbf{A} = \mathbf{QR}$, con $\mathbf{Q}$ ortogonal y $\mathbf{R}$ triangular superior
- Descomposición de autovalores
  - $\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^{\mathsf{T}}$, con $\mathbf{A}$ cuadrada e invertible, $\mathbf{V}$ es la matriz cuyas columnas son los autovectores de $\mathbf{A}$ y Λ es la matriz diagonal con los correspondientes autovalores
- Descomposición de valores singulares
  - $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^{\mathsf{T}}$, con $\mathbf{U}$ y $\mathbf{V}$ ortogonales y Σ diagonal

Derivadas

Función	Derivada
$f(x) = k$	$f'(x) = 0$
$f(x) = x$	$f'(x) = 1$
$f(x) = x^n$	$f'(x) = n \cdot x^{n-1}$
$f(x) = a^{g(x)}$	$f'(x) = g'(x) \cdot a^{g(x)} \cdot \ln(a)$
$f(x) = \log_a(g(x))$	$f'(x) = \frac{g'(x)}{g(x) \cdot \ln(a)}$
$f(x) = ag(x)$	$f'(x) = ag'(x)$
$f(x) = g(x) + h(x)$	$f'(x) = g'(x) + h'(x)$
$f(x) = g(x) \cdot h(x)$	$f'(x) = g'(x) \cdot h(x) + g(x) \cdot h'(x)$
$f(x) = \frac{g(x)}{h(x)}$	$f'(x) = \frac{g'(x) \cdot h(x) - g(x) \cdot h'(x)}{(h(x))^2}$
$f(x) = g(h(x))$	$f'(x) = g'(h(x)) \cdot h'(x)$

Derivadas de varias variables

- $f(x_1, x_2, \ldots, x_n)$, $f(\mathbf{x})$
  - Símbolos en negrita: vectores y matrices (habitualmente)

- Gradiente: vector de derivadas parciales

$$\nabla f(x_1, x_2, \ldots, x_n) = \nabla f(\mathbf{x}) = \frac{\partial f}{\partial \mathbf{x}} = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

- Fórmulas útiles

$$\frac{\partial \mathbf{x}^\mathsf{T}\mathbf{a}}{\partial \mathbf{x}} = \frac{\partial \mathbf{a}^\mathsf{T}\mathbf{x}}{\partial \mathbf{x}} = \mathbf{a} \qquad \frac{\partial \mathbf{x}^\mathsf{T}\mathbf{B}\mathbf{x}}{\partial \mathbf{x}} = (\mathbf{B} + \mathbf{B}^\mathsf{T})\mathbf{x}$$

Derivadas de varias variables

- Matriz Hessiana: matriz de derivadas parciales de segundo orden

$$\nabla^2 f(\mathbf{x}) = \begin{pmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_2^2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_n^2} \end{pmatrix}$$

Límites y la regla de L'Hopital

- $\lim_{n\to\infty} \frac{k}{n} = 0$
- $\lim_{n\to\infty} \frac{n}{k} = \infty$
- $\lim_{n\to\infty} \log_b n = \infty, \quad \text{si } b > 1$
- $\lim_{n\to\infty} n^a = \infty, \quad \text{si } a > 0$
- Regla de L'Hopital: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$
  - Tiene que existir el segundo límite (suele existir)
- Indeterminaciones: 1^∞, 0^0, ∞^0, $0/0$, ∞/∞, ∞^0, $+\infty - \infty$

Integrales

- Asumiendo $f(x) = F'(x)$ (en un intervalo $[a, b]$), entonces:
  - $\int f(x)\,dx = F(x) + C$
  - $\int_a^b f(x)\,dx = F(b) - F(a)$
- Técnicas de integración comunes
  - Cambio de variable (sustitución)
    - Sustitución trigonométrica
  - Descomposición en fracciones simples
  - Integración por partes: $\int u\,dv = uv - \int v\,du$

Integrales - Algunos ejemplos

- $\int dx = x$
- $\int x^n dx = \frac{x^{n+1}}{n+1}$
- $\int \frac{1}{x} dx = \ln(x)$
- $\int a^x dx = \frac{a^x}{\ln(a)}$
- $\int (f(x) + g(x))dx = \int f(x)dx + \int g(x)dx$
- $\int af(x)dx = a \int f(x)dx$

Geometría

- Producto escalar:

$$\vec{\mathbf{a}} \cdot \vec{\mathbf{b}} = \langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a}^\top \mathbf{b} = \sum_{i=1}^{n} a_i b_i = |\mathbf{a}| \cdot |\mathbf{b}| \cdot \cos(\alpha)$$

- Módulo o norma Euclídea:

$$|\mathbf{a}| = \|\mathbf{a}\|_2 = \sqrt{a_1^2 + \cdots + a_n^2} = \sqrt{\mathbf{a}^\top \mathbf{a}}.$$

- Otras normas:

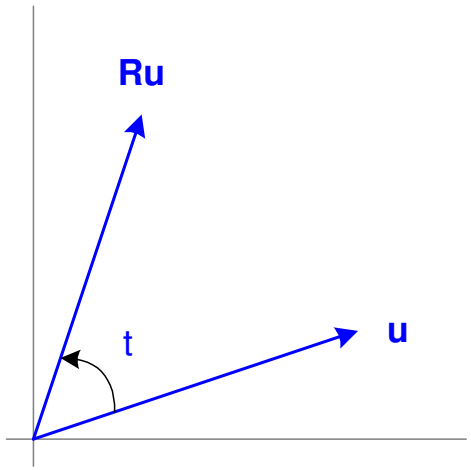
$$\|\mathbf{a}\|_1 = |a_1| + |a_2| + \cdots + |a_n|$$

$$\|\mathbf{a}\|_\infty = \max_{i=1,\ldots,n} \{a_1, a_2, \ldots, a_n\}$$

Geometría

- Matriz de rotación:

$$\mathbf{R} = \begin{bmatrix} \cos(t) & -\sin(t) \\ \sin(t) & \cos(t) \end{bmatrix}$$



- El vector $\mathbf{R} \cdot \mathbf{u}$ corresponde al vector $\mathbf{u}$ rotado t radianes en sentido contrario a las agujas del reloj

Sumatorios

Notación básica

- Definición:

$$\sum_{i=m}^{n} f(i) = f(m) + f(m+1) + \cdots + f(n-1) + f(n)$$

- No depende de la variable contadora:

$$\sum_{i=m}^{n} f(i) = \sum_{j=m}^{n} f(j)$$

- Suma en sentido contrario:

$$\sum_{i=m}^{n} f(i) = \sum_{i=m}^{n} f(n+m-i)$$

Propiedades básicas

- $\sum_{i=1}^{n} 1 = \underbrace{1 + 1 + \cdots + 1 + 1}_{n \text{ veces}} = n$
- $\sum_{i=1}^{n} k = \underbrace{k + k + \cdots + k + k}_{n \text{ veces}} = kn$
- $\sum_{i=m}^{n} k = \underbrace{k + k + \cdots + k + k}_{n-m+1 \text{ veces}} = k(n - m + 1)$

Propiedades básicas

- Operador lineal:
  - $\sum_{i=m}^{n} (f(i) + g(i)) = \sum_{i=m}^{n} f(i) + \sum_{i=m}^{n} g(i)$
  - $\sum_{i=m}^{n} kf(i) = k \sum_{i=m}^{n} f(i)$ Factor común
- Convención:
  - $\sum_{i=m}^{n} f(i) = 0$ si $m > n$

Suma de los primeros n naturales

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

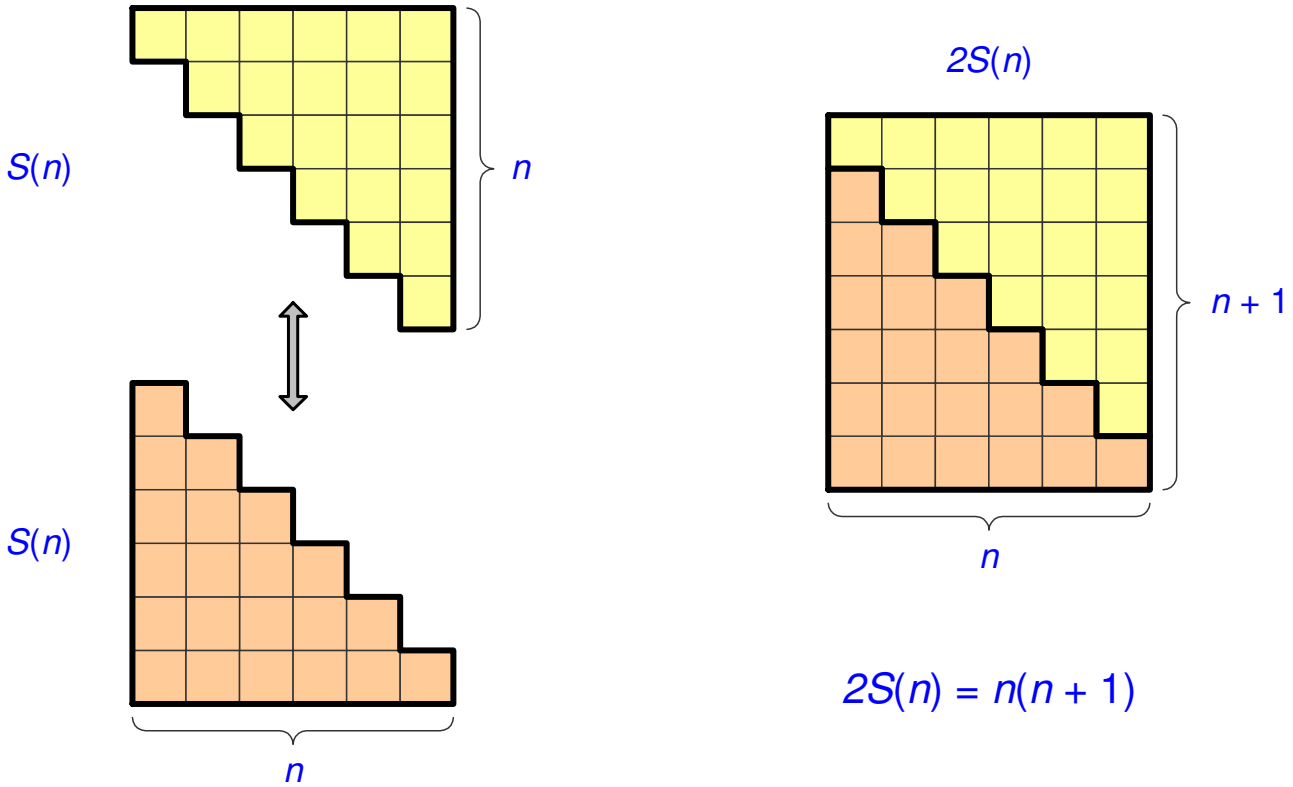
Demostración:

$$\begin{array}{rrcccccccccc}
 & S(n) & = & 1 & + & 2 & + & \cdots & + & (n-1) & + & n \\
+ & S(n) & = & n & + & (n-1) & + & \cdots & + & 2 & + & 1 \\
\hline
 & 2S(n) & = & (n+1) & + & (n+1) & + & \cdots & + & (n+1) & + & (n+1)
\end{array}$$

$$2S(n) = n(n+1) \quad \Rightarrow \quad \boxed{S(n) = \frac{n(n+1)}{2}}$$

Suma de los primeros n naturales

Demostración visual:



$2S(n) = n(n+1)$

$$S(n) = \frac{n(n+1)}{2}$$

Suma de los primeros n impares

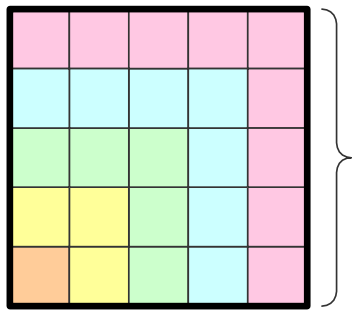
$$S(n) = \sum_{i=1}^{n}(2i-1) = 1+3+\cdots+(2n-3)+(2n-1) = n^2$$

Demostración:

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

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

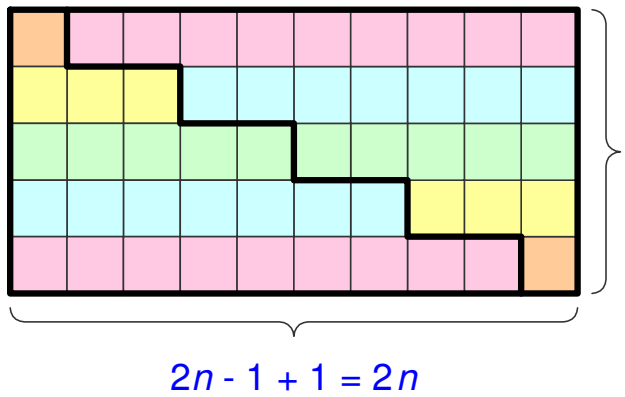
Demostración visual:



$$\Rightarrow \quad S(n) = n^2$$

Suma de los primeros n impares

Demostración visual alternativa:



$$2S(n) = 2n \cdot n \quad \Rightarrow \quad S(n) = n^2$$

Análogamente:

$$\begin{array}{rrcccccccccc}
 & S(n) & = & 1 & + & 3 & + & \cdots & + & (2n-3) & + & (2n-1) \\
+ & S(n) & = & (2n-1) & + & (2n-3) & + & \cdots & + & 3 & + & 1 \\
\hline
 & 2S(n) & = & 2n & + & 2n & + & \cdots & + & 2n & + & 2n
\end{array}$$

$$2S(n) = 2n(n) \quad \Rightarrow \quad \boxed{S(n) = n^2}$$

Sumas parciales de series aritméticas

- Progresión aritmética: $a_i = a_{i-1} + d$
  - $a_i = a_{i-1} + d = a_{i-2} + 2d = \cdots = a_0 + id$
  - $a_{m+k} + a_{n-k} = a_m + a_n$
- Fórmula general: $S(m,n) = \sum_{i=m}^{n} a_i = \frac{1}{2}(a_m + a_n)(n - m + 1)$

Demostración:

$$
\begin{array}{rrcccccccccc}
 & S(m,n) & = & a_m & + & a_{m+1} & + & \cdots & + & a_{n-1} & + & a_n \\
+ & S(m,n) & = & a_n & + & a_{n-1} & + & \cdots & + & a_{m+1} & + & a_m \\
\hline
 & 2S(m,n) & = & (a_m + a_n) & + & (a_m + a_n) & + & \cdots & + & (a_m + a_n) & + & (a_m + a_n)
\end{array}
$$

$$
2S(m,n) = (a_m + a_n)(n - m + 1) \quad \Rightarrow \quad \boxed{S(m,n) = \frac{1}{2}(a_m + a_n)(n - m + 1)}
$$

Suma de los primeros n cuadrados

$$S(n) = \sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + (n-1)^2 + n^2 = \frac{(2n+1)n(n+1)}{6}$$

- $\sum_{i=1}^{n} i$ (suma naturales) y $\sum_{i=1}^{n}(2i-1)$ (suma impares) son sumas parciales de progresiones aritméticas
- $\sum_{i=1}^{n} i^2$ **no** es una suma parcial de una progresión aritmética (ni geométrica)
- Sumas del tipo $\sum_{i=1}^{n} i^p$ son importantes ya que aparecen en análisis de algoritmos, tanto iterativos como recursivos

Suma de los primeros n cuadrados: demostración

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \ldots + n^2 = \frac{n(2n+1)(n+1)}{6}$$

Demostración 1 (descomponemos los cuadrados en sumas de impares):

$$\begin{array}{rcccccccccccl}
S(n) = \sum_{i=1}^{n} i^2 & = & 1 & + & 4 & + & 9 & + & 16 & + & \cdots & + & n^2 \\
\hline
 & & 1 & + & 1 & + & 1 & + & 1 & + & & + & 1 \quad = \quad 1(n) \\
 & & & + & 3 & + & 3 & + & 3 & + & & + & 3 \quad = \quad 3(n-1) \\
 & & & & & + & 5 & + & 5 & + & & + & 5 \quad = \quad 5(n-2) \\
 & & & & & & & + & 7 & + & & + & 7 \quad = \quad 7(n-3) \\
 & & & & & & & & & & & & \vdots \qquad\quad \vdots \\
 & & & & & & & & & & & + & 2n-1 \quad = \quad (2n-1)1 \\
\hline
\end{array}$$

Suma de los primeros n cuadrados: demostración

Sumamos los términos de la última columna:

$$S(n) = \sum_{i=1}^{n} i^2 = 1 \cdot n + 3 \cdot (n-1) + 5 \cdot (n-2) + \cdots + (2n-1) \cdot 1$$

$$= \sum_{i=1}^{n} (2i-1)(n-i+1) = \sum_{i=1}^{n} \left(2in - 2i^2 + 2i - n + i - 1\right)$$

$$= \sum_{i=1}^{n} (2n+3)i - 2\underbrace{\sum_{i=1}^{n} i^2}_{S(n)} - \sum_{i=1}^{n} n - \sum_{i=1}^{n} 1$$

$$= (2n+3)\sum_{i=1}^{n} i - 2S(n) - n^2 - n$$

Suma de los primeros n cuadrados: demostración

Continuando:

$$S(n) = \frac{(2n+3)n(n+1)}{2} - 2S(n) - n^2 - n$$

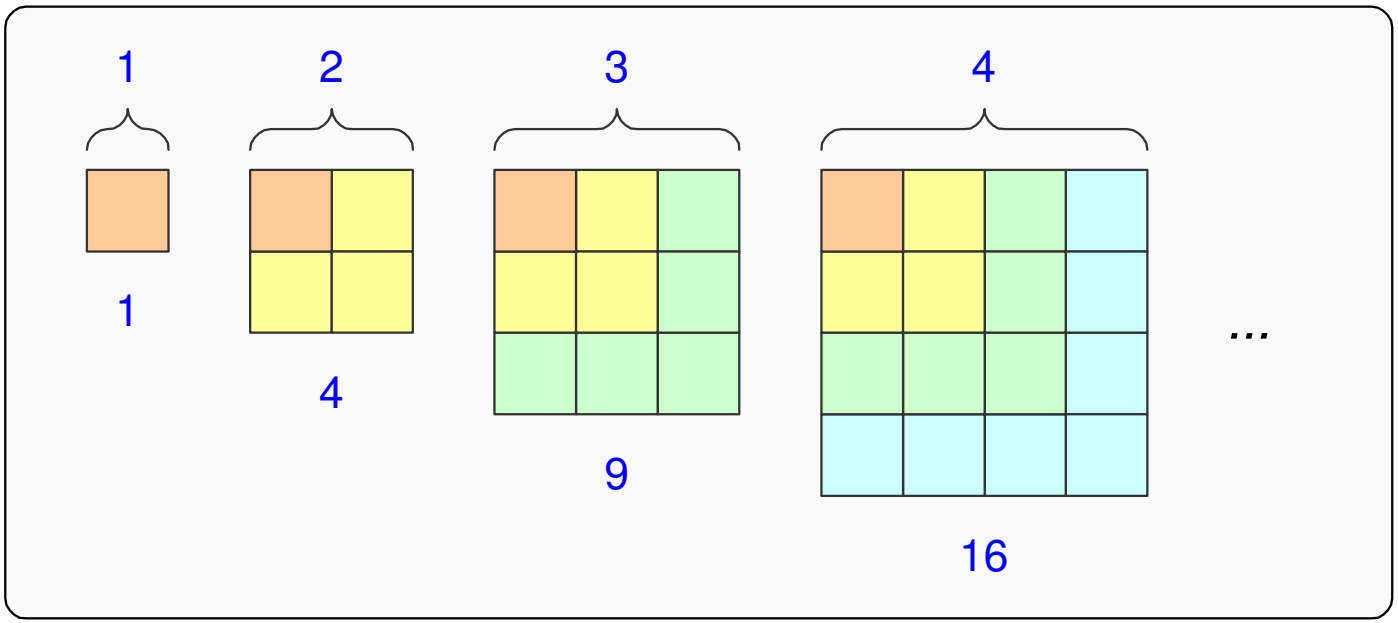
$$3S(n) = \frac{(2n+3)n(n+1)}{2} - n^2 - n$$

$$S(n) = \frac{n(2n^2+3n+1)}{6} = \frac{n \cdot 2(n+\frac{1}{2})(n+1)}{6}$$

$$\boxed{S(n) = \frac{n(2n+1)(n+1)}{6}}$$

Suma de los primeros n cuadrados: demostración

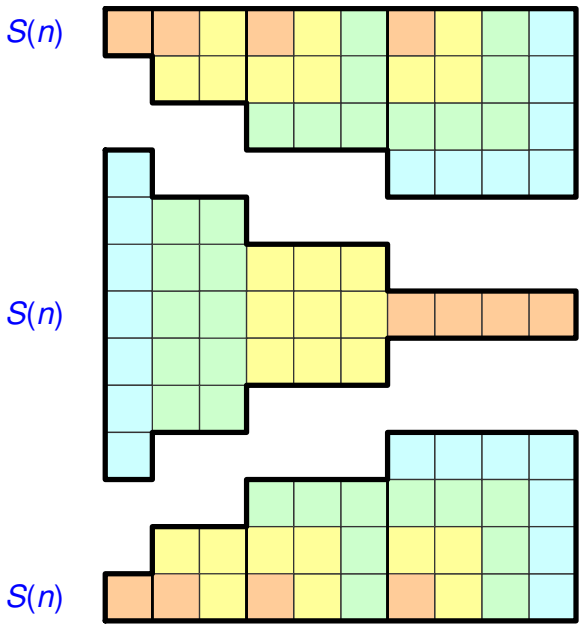
Usaremos esta descomposición de la suma de cuadrados:

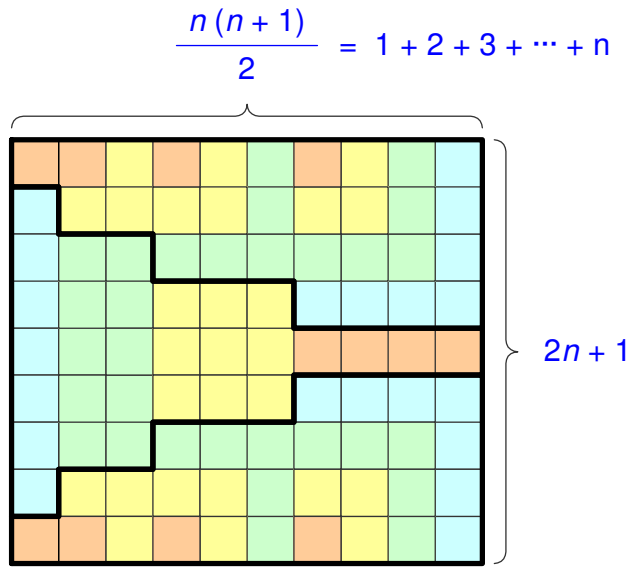


$S(n)$

Suma de los primeros n cuadrados: demostración

Demostración visual:





$$S(n) = \frac{n(2n+1)(n+1)}{6}$$

Suma de los primeros n cubos

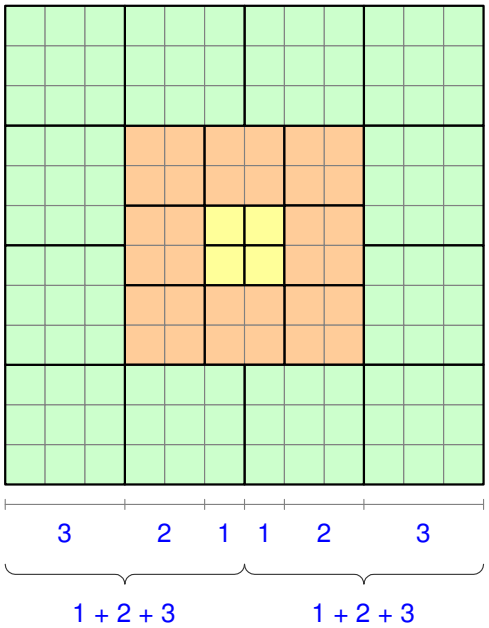
$$S(n) = \sum_{i=1}^{n} i^3 = \left(\sum_{i=1}^{n} i\right)^2 = \left(\frac{n(n+1)}{2}\right)^2$$

Se puede ver gracias a:

$$\begin{aligned}
1^3 &= 1 = 1^2 = (1)^2 \\
1^3 + 2^3 &= 9 = 3^2 = (1+2)^2 \\
1^3 + 2^3 + 3^3 &= 36 = 6^2 = (1+2+3)^2 \\
1^3 + 2^3 + 3^3 + 4^3 &= 100 = 10^2 = (1+2+3+4)^2
\end{aligned}$$

Suma de los primeros n cubos

Demostración visual:



$$\text{Área total} = \left[2(1+2+3+\cdots+n)\right]^2$$

$$\begin{aligned} 4 \cdot 1^2 &= 4 \cdot 1^3 \\ 8 \cdot 2^2 &= 4 \cdot 2^3 \\ 12 \cdot 3^2 &= 4 \cdot 3^3 \\ &\vdots \\ 4n \cdot n^2 &= 4 \cdot n^3 \\ \hline \text{Área total} &= 4 \cdot S(n) \end{aligned}$$

$$S(n) = 1^3 + 2^3 + 3^3 + \cdots + n^3 = (1+2+3+\cdots+n)^2 = \left(\frac{n(n+1)}{2}\right)^2$$

Sumatorios telescópicos

$$S(n) = \sum_{i=1}^{n} (a_i - a_{i-1}) = a_n - a_0$$

Demostración:

$$S(n) = -a_0 + \underbrace{a_1 - a_1}_{0} + \underbrace{a_2 - a_2}_{0} \cdots + \underbrace{a_{n-1} - a_{n-1}}_{0} + a_n = a_n - a_0$$

Ejemplo:

$$\sum_{i=1}^{n-1} \frac{1}{i(i+1)} = \sum_{i=1}^{n-1} \left(\frac{1}{i} - \frac{1}{i+1} \right) = +1 - \frac{1}{2} + \frac{1}{2} - \frac{1}{3} + \ldots + \frac{1}{n-1} - \frac{1}{n} = 1 - \frac{1}{n}$$

Hemos hecho: $a_i = -1/(i+1)$

$$\sum_{i=1}^{n-1} \frac{1}{i(i+1)} = \sum_{i=1}^{n-1} \left(+ \left(- \frac{1}{i+1} \right) - \left(- \frac{1}{i} \right) \right) = - \frac{1}{n} - (-1) = 1 - \frac{1}{n}$$

Suma de las primeras n potencias

- Método general para hallar $S(n) = \sum_{i=1}^{n} i^p$
  - Es necesario conocer $\sum_{i=1}^{n} i^k$ para $k < p$
- Ejemplo: $S_3 = \sum_{i=1}^{n} i^3$. Necesitamos:

$$S_0 = \sum_{i=1}^{n} i^0 = n$$

$$S_1 = \sum_{i=1}^{n} i^1 = \frac{n(n+1)}{2}$$

$$S_2 = \sum_{i=1}^{n} i^2 = \frac{(2n+1)n(n+1)}{6}$$

Suma de las primeras n potencias

- Se plantea el sumatorio telescópico: $S = \sum_{i=1}^{n} \left[(i+1)^{p+1} - i^{p+1}\right]$
- En el ejemplo ($p = 3$): $S = \sum_{i=1}^{n} \left[(i+1)^4 - i^4\right]$
  - Al ser telescópico: $S = \sum_{i=1}^{n} \left[(i+1)^4 - i^4\right] = (n+1)^4 - 1^4$
  - y desarrollando:

$$\begin{aligned} S &= \sum_{i=1}^{n} \left[i^4 + 4i^3 + 6i^2 + 4i + 1 - i^4\right] \\ &= \sum_{i=1}^{n} \left[4i^3 + 6i^2 + 4i + 1\right] \\ &= 4S_3 + 6S_2 + 4S_1 + S_0 \end{aligned}$$

Suma de las primeras n potencias

- Igualando ambas expresiones:

$$(n+1)^4 - 1^4 = 4S_3 + 6S_2 + 4S_1 + S_0$$

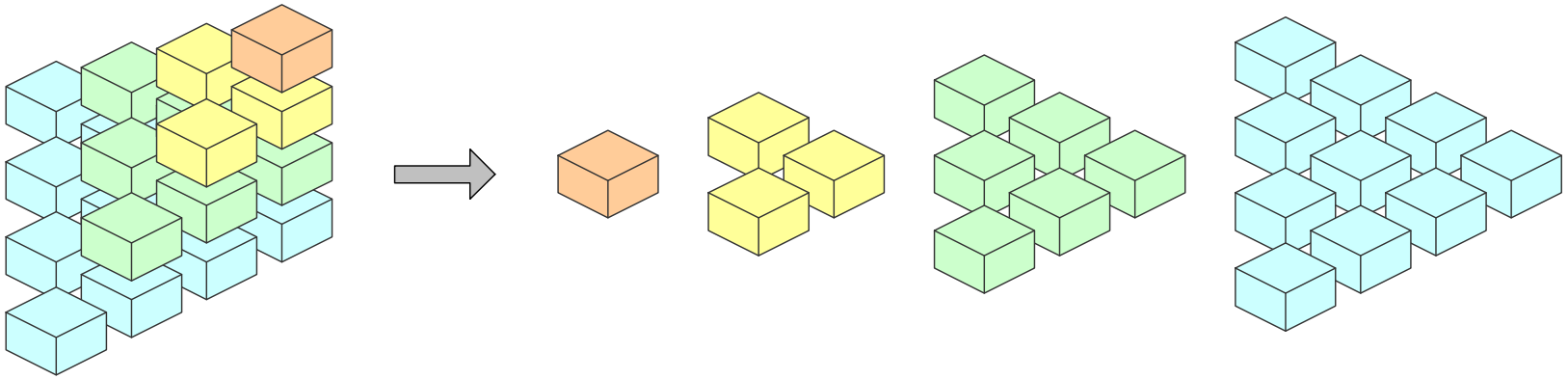
$$4S_3 = (n+1)^4 - 1 - 6S_2 - 4S_1 - S_0$$

- Desarrollando y sustituyendo S_0, S_1 y S_2:

$$\begin{aligned} 4S_3 &= n^4 + 4n^3 + 6n^2 + 4n + 1 - 1 - 2n^3 - 3n^2 - n - 2n^2 - 2n - n \\ 4S_3 &= n^4 + 2n^3 + n^2 \\ S_3 &= \frac{n^4 + 2n^3 + n^2}{4} = \left[\frac{n(n+1)}{2}\right]^2 = (1 + 2 + \cdots + n)^2 \end{aligned}$$

Ejercicio

- Contar el número de cajas para una pirámide de altura n



Solución

$$S(n) = 1 + (1+2) + (1+2+3) + \cdots + (1+2+3+\cdots+n)$$

- Agrupando los términos en cada paréntesis:

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

- Agrupando según el número de veces que se suma un determinado entero:

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

- La solución es:

$$\boxed{S(n) = \frac{n(n+1)(n+2)}{6}}$$

Sumas parciales de series geométricas

- Progresión aritmética: $a_i = a_{i-1} \cdot r$
  - $a_i = a_{i-1} r = a_{i-2} r^2 = \cdots = a_0 r^i$
- Fórmula general: $S(n) = \sum_{i=m}^{n} a_i = a_0 \sum_{i=m}^{n} r^i = a_0 \frac{r^{n+1} - r^m}{r - 1}$

Demostración (asumiendo $a_0 = 1$):

$$\begin{array}{rcccccccccccc} S(n) & = & r^m & + & r^{m+1} & + & r^{m+2} & + & \cdots & + & r^{n-1} & + & r^n \\ rS(n) & = & & + & r^{m+1} & + & r^{m+2} & + & \cdots & + & r^{n-1} & + & r^n & + & r^{n+1} \\ \hline rS(n) - S(n) & = & -r^m & & & & & & & & & & & + & r^{n+1} \end{array}$$

$$\boxed{S(n) = \sum_{i=m}^{n} r^i = \frac{r^{n+1} - r^m}{r - 1}}$$

Sumas parciales de series geométricas

- Ejemplo 1
  - $S(n) = \sum_{i=0}^{n-1} 2^i = 2^n - 1$

$$\begin{array}{rcl} S(n) & = & 2^0 + 2^1 + 2^2 + \cdots + 2^{n-1} \\ rS(n) & = & + 2^1 + 2^2 + \cdots + 2^{n-1} + 2^n \\ \hline 2S(n) - S(n) & = & -1 + 2^n \end{array}$$

$$\boxed{S(n) = 2^n - 1}$$

- Ejemplo 2
  - $S(n) = \sum_{i=0}^{n-1} \frac{1}{2^i} = \sum_{i=0}^{n-1} \left(\frac{1}{2}\right)^i = \frac{\left(\frac{1}{2}\right)^n - 1}{\frac{1}{2} - 1} = \frac{\frac{1-2^n}{2^n}}{-\frac{1}{2}} = 2\frac{2^n - 1}{2^n}$

Sumas parciales alternativa

$$S(n) = \sum_{i=1}^{n} i r^i = 1 \cdot r^1 + 2 \cdot r^2 + 3 \cdot r^3 + \cdots + (n-1) \cdot r^{n-1} + n \cdot r^n$$

- No corresponde a una progresión geométrica (ni aritmética)
- Se puede deducir su fórmula procediendo:
  1. Resta de sumatorios
  2. Aplicando derivadas
  3. Descomponiendo la suma en varias sumas parciales geométricas

Sumas parciales alternativa

- Método 1 (resta de sumatorios):

$$S(n) = \sum_{i=1}^{n} i r^i = 1 \cdot r^1 + 2 \cdot r^2 + 3 \cdot r^3 + \cdots + (n-1) \cdot r^{n-1} + n \cdot r^n$$

$$\begin{array}{rcccccccccccc} S(n) & = & 1 \cdot r^1 & + & 2 \cdot r^2 & + & 3 \cdot r^3 & + & \cdots & + & n \cdot r^n & & \\ rS(n) & = & & + & 1 \cdot r^2 & + & 2 \cdot r^3 & + & \cdots & + & (n-1) \cdot r^n & + & nr^{n+1} \\ \hline S(n) - rS(n) & = & r^1 & + & r^2 & + & r^3 & + & \cdots & + & r^n & - & nr^{n+1} \end{array}$$

$r^1 + r^2 + r^3 + \cdots + r^n$ es una suma geométrica con fórmula: $(r^{n+1} - r)/(r-1)$:

$$S(n) - rS(n) = \frac{r^{n+1} - r}{r-1} - nr^{n+1}$$

$$S(n) = \frac{r - r^{n+1}}{(r-1)^2} + \frac{nr^{n+1}}{r-1} = \frac{r - r^{n+1} + (r-1)nr^{n+1}}{(r-1)^2} = \frac{r + r^{n+1}(nr - n - 1)}{(r-1)^2}$$

Sumas parciales alternativa

- Método 2 (derivada de suma parcial geométrica):

$$S(n) = \sum_{i=1}^{n} i r^i = 1 \cdot r^1 + 2 \cdot r^2 + 3 \cdot r^3 + \cdots + (n-1) \cdot r^{n-1} + n \cdot r^n$$

Consideramos una suma parcial geométrica general y su derivada:

$$\begin{array}{ccccc} T & = & 1 + x + x^2 + \cdots + x^{m-1} & = & \dfrac{x^m - 1}{x - 1} \\ \downarrow & & \downarrow \text{ derivando} & & \downarrow \\ \dfrac{dT}{dx} & = & 1 + 2x + 3x^2 + \cdots + (m-1)x^{m-2} & = & \dfrac{mx^{m-1}(x-1) - 1 \cdot (x^m - 1)}{(x-1)^2} \end{array}$$

En este caso tenemos:

$$S(n) = x \frac{dT}{dx}\bigg|_{x=r,\ m=n+1} = r \frac{(n+1)r^n(r-1) - (r^{n+1} - 1)}{(r-1)^2} = \frac{r + r^{n+1}(nr - n - 1)}{(r-1)^2}$$

Sumas parciales alternativa

- Método 3 (descomposición):

$$S(n) = \sum_{i=1}^{n} i r^i = r + 2r^2 + 3r^3 + \cdots + nr^n$$

$$\begin{array}{rcl}
r + r^2 + r^3 + \cdots + r^n & = & (r^{n+1} - r)/(r-1) \\
+ r^2 + r^3 + \cdots + r^n & = & (r^{n+1} - r^2)/(r-1) \\
+ r^3 + \cdots + r^n & = & (r^{n+1} - r^3)/(r-1) \\
\vdots & & \vdots \\
+ r^n & = & (r^{n+1} - r^n)/(r-1)
\end{array}$$

En este caso tenemos:

$$S(n) = \frac{1}{r-1}\left[nr^{n+1} - \sum_{i=1}^{n} r^i\right] = \frac{1}{r-1}\left[nr^{n+1} - \frac{r^{n+1} - r}{r-1}\right]$$

$$= \frac{(r-1)nr^{n+1} - r^{n+1} + r}{(r-1)^2} = \frac{r + r^{n+1}(nr - n - 1)}{(r-1)^2}$$

Aproximación por integrales

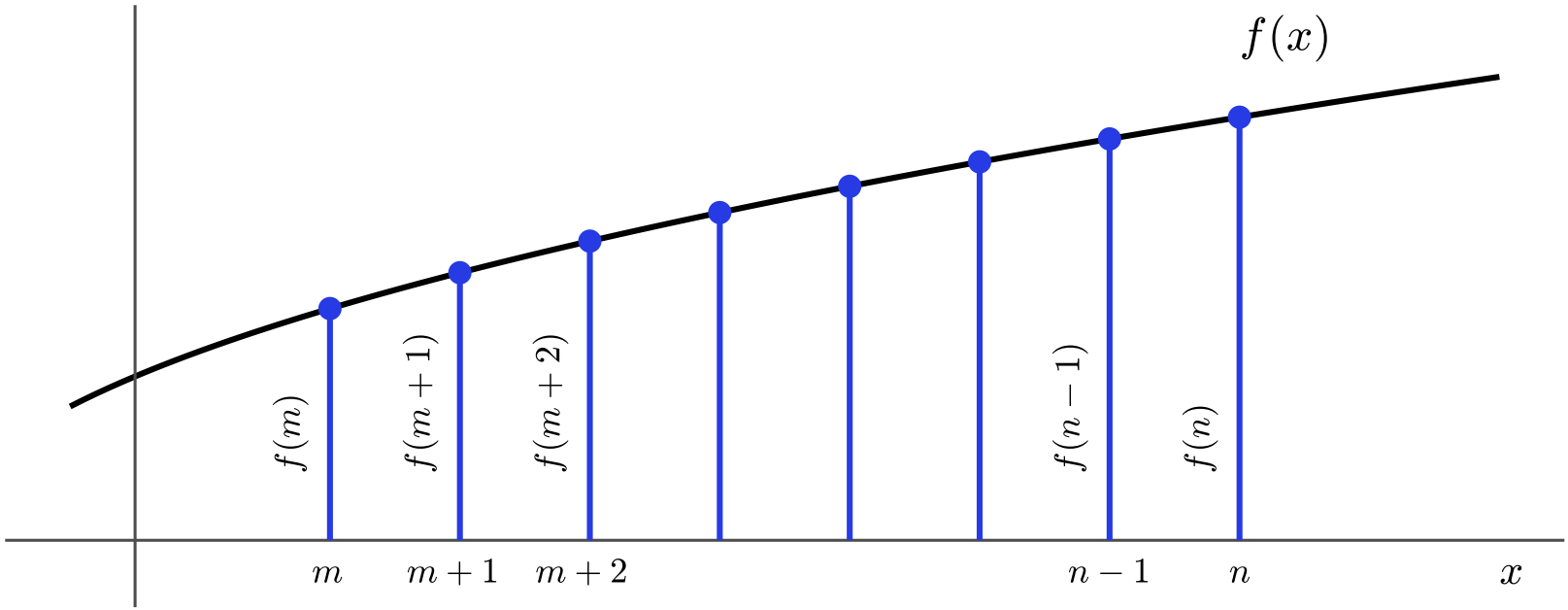
- Para muchos sumatorios no obtendremos una fórmula analítica
- Tendremos que recurrir a cotas
- Para $f(x)$ monótona creciente:

$$\int_{m-1}^{n} f(x)dx \quad \leq \quad \sum_{i=m}^{n} f(i) \quad \leq \quad \int_{m}^{n+1} f(x)dx$$

- Para $f(x)$ monótona decreciente:

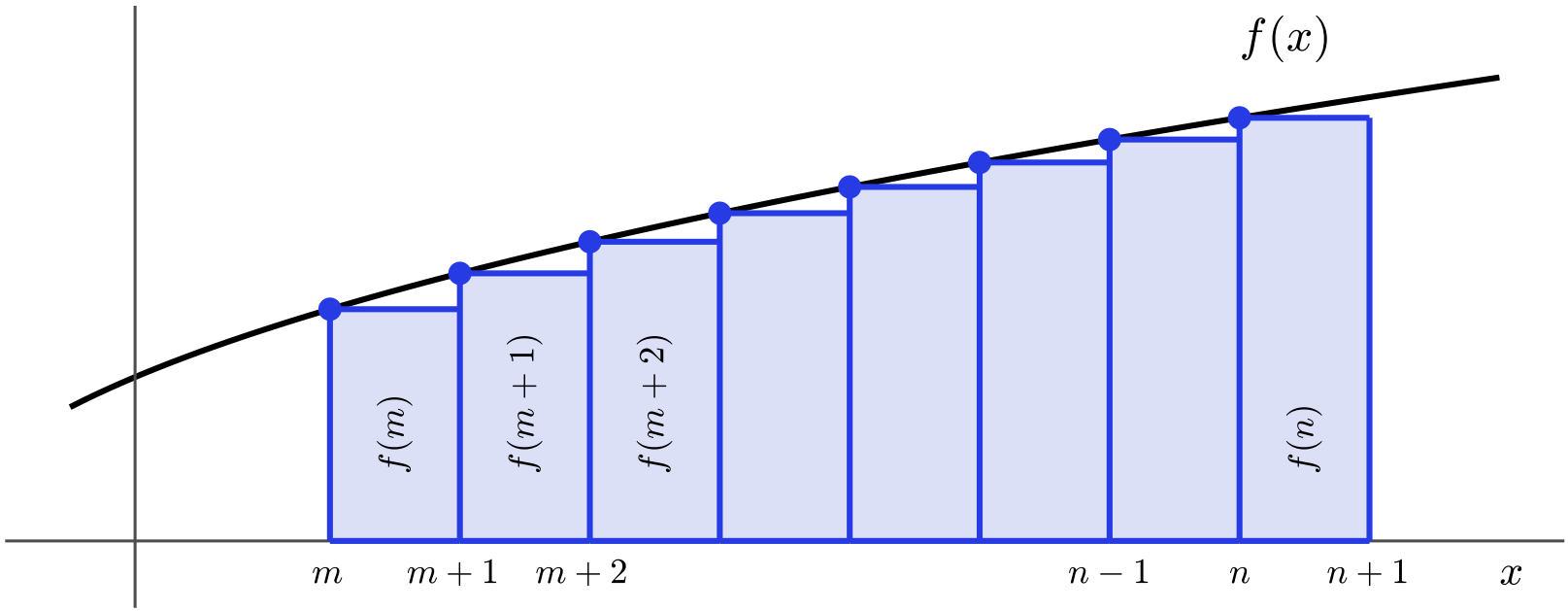
$$\int_{m}^{n+1} f(x)dx \quad \leq \quad \sum_{i=m}^{n} f(i) \quad \leq \quad \int_{m-1}^{n} f(x)dx$$

Función creciente



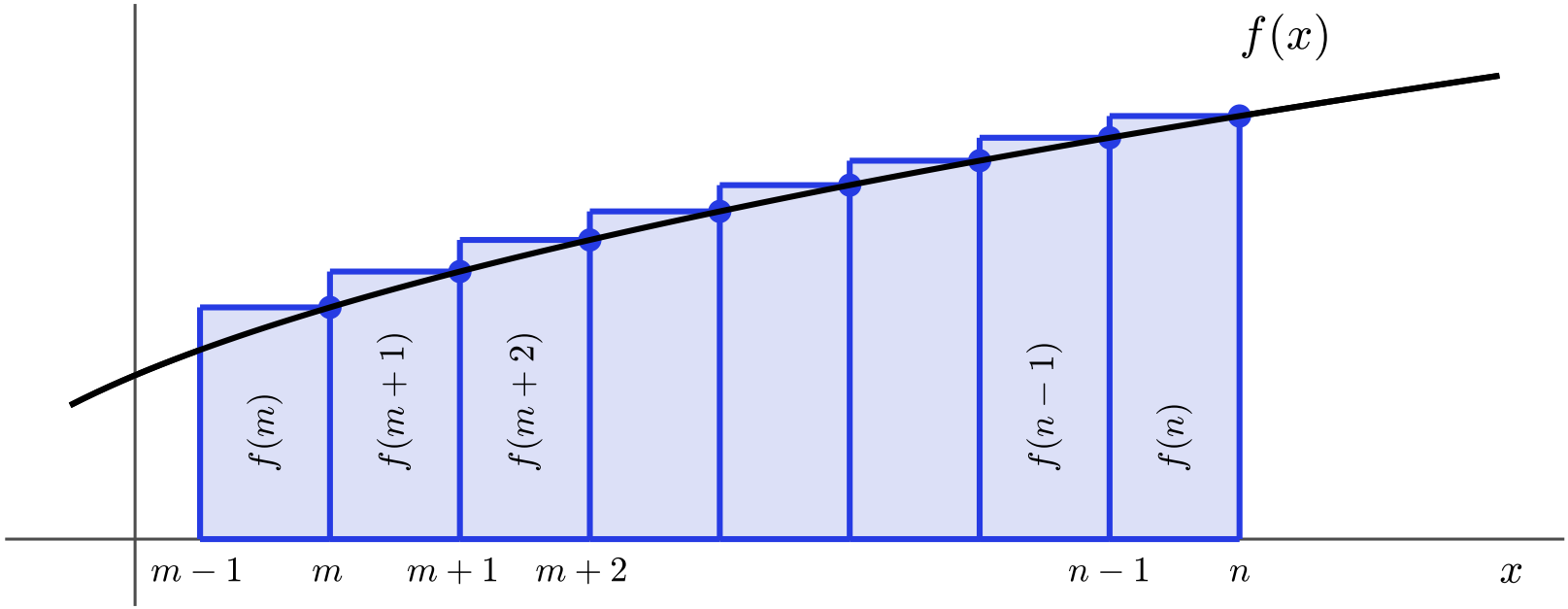
$$\sum_{i=m}^{n} f(i) \quad = \quad \text{suma de longitudes de segmentos}$$

Función creciente – Cota superior



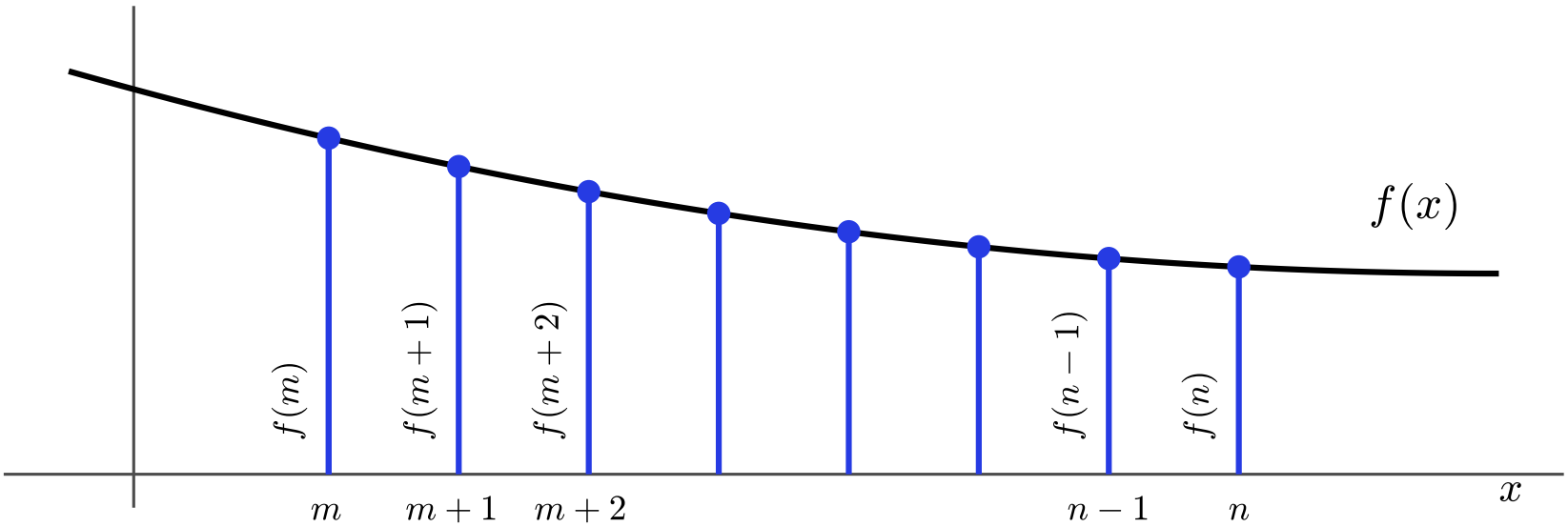
$$\sum_{i=m}^{n} f(i) \quad = \quad \text{suma de áreas} \quad \leq \quad \underbrace{\int_{m}^{n+1} f(x)dx}_{\text{Área debajo de la curva desde } m \text{ hasta } n+1}$$

Función creciente – Cota inferior



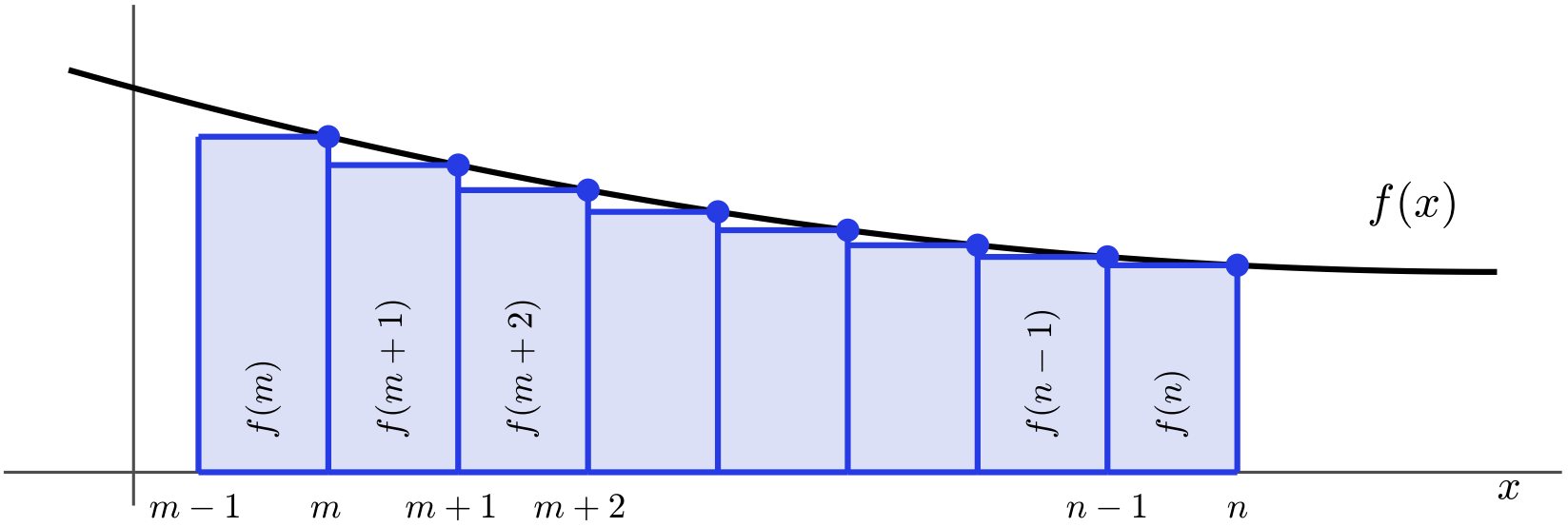
$$\underbrace{\int_{m-1}^{n} f(x)dx}_{\text{Área debajo de la curva desde } m-1 \text{ hasta } n} \leq \text{suma de áreas} = \sum_{i=m}^{n} f(i)$$

Función decreciente



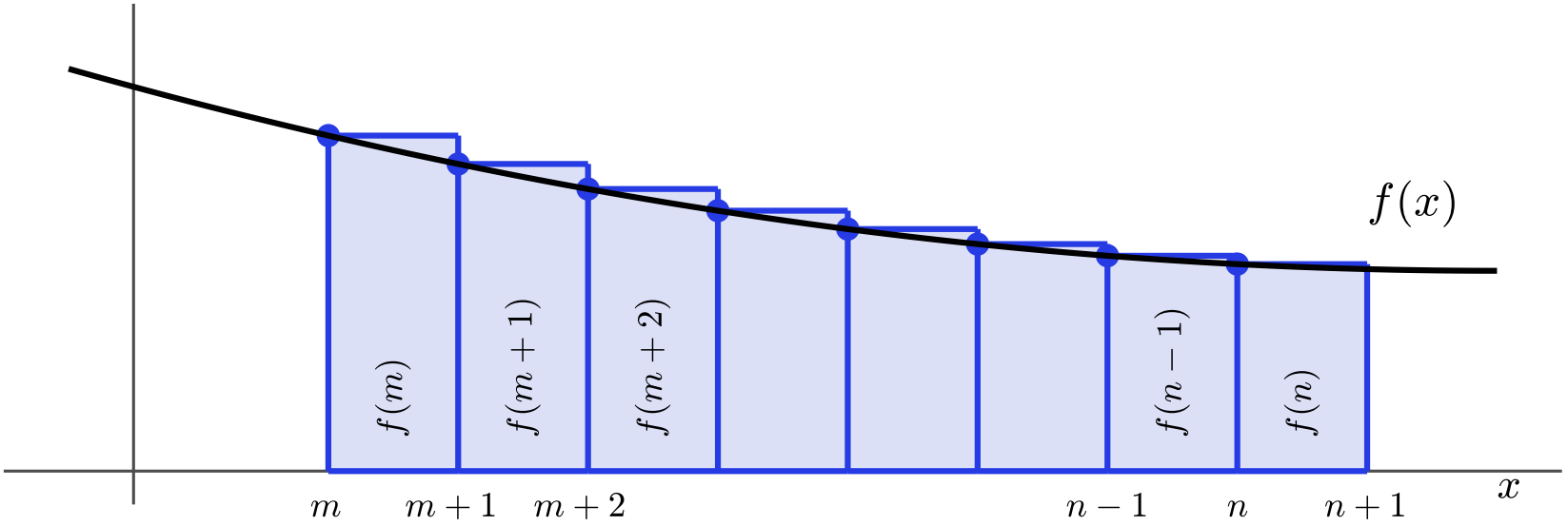
$$\sum_{i=m}^{n} f(i) \quad = \quad \text{suma de longitudes de segmentos}$$

Función decreciente – Cota superior



$$\sum_{i=m}^{n} f(i) \quad = \quad \text{suma de áreas} \quad \leq \quad \underbrace{\int_{m-1}^{n} f(x)dx}_{\text{Área debajo de la curva desde } m-1 \text{ hasta } n}$$

Función decreciente – Cota inferior



$$\underbrace{\int_{m}^{n+1} f(x)dx}_{\text{Área debajo de la curva desde } m \text{ hasta } n+1} \leq \text{suma de áreas} = \sum_{i=m}^{n} f(i)$$

Ejemplo: Suma parcial de la serie armónica

- Acotar $\sum_{i=1}^{n} \frac{1}{i} = 1 + \sum_{i=2}^{n} \frac{1}{i}$
- Cota superior

$$\sum_{i=2}^{n} \frac{1}{i} \leq \int_{1}^{n} \frac{1}{x} dx = \ln(n) \implies$$

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

- Cota inferior

$$\sum_{i=1}^{n} \frac{1}{i} \geq \int_{1}^{n+1} \frac{1}{x} dx = \ln(n+1)$$

Ejemplo: Suma parcial de la serie armónica

$$\ln(n+1) \;\leq\; S = \sum_{i=1}^{n} \frac{1}{i} = \ln n + \gamma + \mathcal{O}(1/n) \;\leq\; \ln(n) + 1$$

- $\gamma \simeq 0{,}577$ es la constante de Euler
- $\mathcal{O}(1/n)$ es un término muy pequeño
- La serie diverge: $\sum_{i=1}^{\infty} \frac{1}{i} = \infty$

Productos

Notación básica

- Definición:

$$\prod_{i=m}^{n} f(i) = f(m) \times f(m+1) \times \cdots \times f(n-1) \times f(n)$$

- Convención:
  - $\prod_{i=m}^{n} f(i) = 1$ si $m > n$

Productos o "productorios"

- Relación entre productos y sumatorios al utilizar **logaritmos**

$$\log\left(\prod_{i=1}^{n} a_i\right) = \sum_{i=1}^{n} \log a_i$$

- Se emplea mucho en probabilidad
  - Para simplificar cálculos
  - Para evitar desbordamientos (*underflows*), ya que al multiplicar muchas probabilidades (números entre 0 y 1) el resultado se hace muy pequeño
  - Al comparar probabilidades ($\log_b(x)$ es creciente para $b > 1$: no afecta desigualdades)

Truco "log-suma-exp" para evitar underflows

- Ejemplo: $P = \sum_{i}^{m} e^{a_i}$ (P muy pequeña)
- En muchos modelos probabilistas a_i son números negativos grandes, resultado de sumas de logaritmos de probabilidades (logaritmos de productos de probabilidades)
  - e^{a_i} causaría un error de *underflow* (por ejemplo, e^{-1000})
- Aplicamos logaritmos: $\log(P) = \log\left(\sum_{i=1}^{m} e^{a_i}\right) \approx \max_{i=1..m}\{a_i\}$
- Truco: $b = \max_{i=1..m}\{a_i\}$

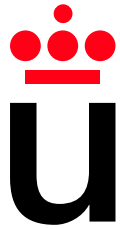
$$\log(P) = \log\left(e^{b}\sum_{i=1}^{m} e^{a_i-b}\right) = b + \log\left(\sum_{i=1}^{m} e^{a_i-b}\right)$$

Tema 2.2
Notaciones asintóticas

Diseño y Análisis de Algoritmos

Manuel Rubio Sánchez

13 de mayo de 2024



Universidad
Rey Juan Carlos

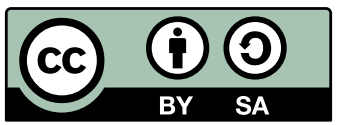
Copyright

©2024 Manuel Rubio Sánchez

Algunos derechos reservados.

Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en:

https://creativecommons.org/licenses/by-sa/4.0/deed.es.



Contenido

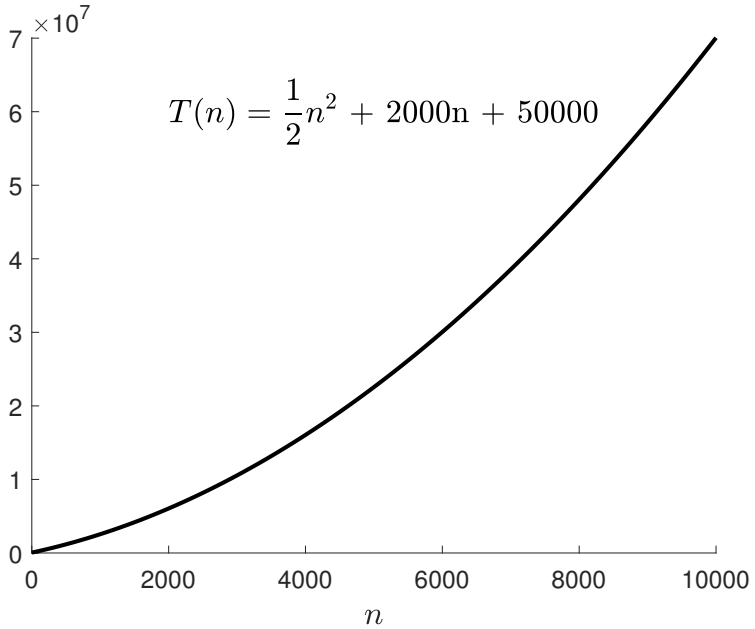
Introducción

Notaciones asintóticas

- Nos interesa cómo crece el tiempo de ejecución
  - Según aumenta el tamaño de la entrada
  - "En el límite", según el tamaño crece sin cota (hasta ∞)
- Eficiencia asintótica de algoritmos
  - Asumimos que las entradas son muy grandes
  - Nos interesa el "orden de crecimiento"
  - Las constantes y términos de orden inferior no son relevantes, al ser *dominados* por un término de orden superior
- El algoritmo con mejor coste o eficiencia asintótica suele ser la mejor elección
  - Salvo para entradas muy pequeñas

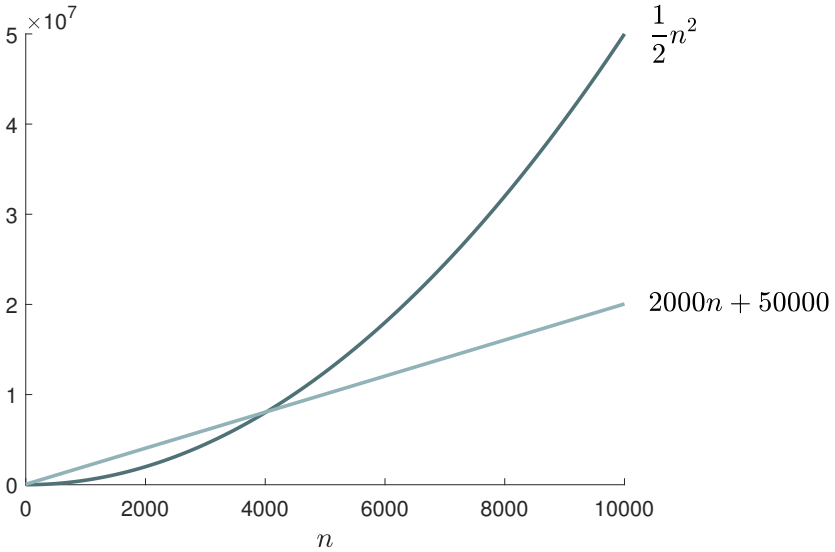
Tiempo de ejecución de un algoritmo

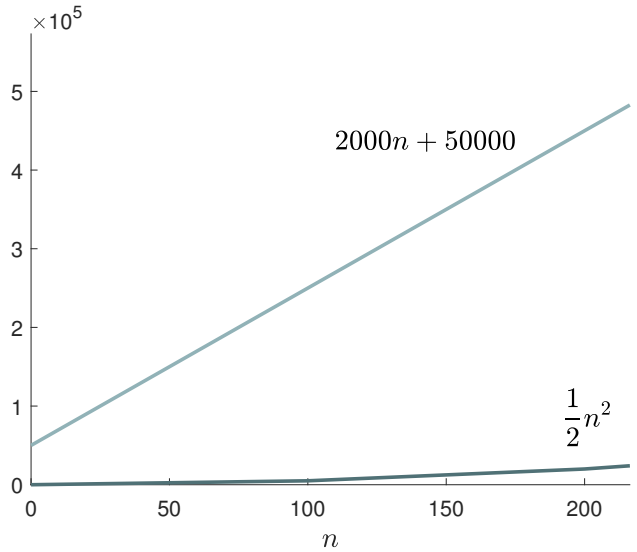
- Expresaremos el tiempo de ejecución mediante una fórmula (función) matemática
  - Generalmente la llamaremos $T(n)$
  - Es importante saber qué argumentos debe tomar dicha función
- Ejemplo:



- ¿Qué términos de $T(n)$ importan más para entender cómo crece?

Descomposición





- El término cuadrático determina el "orden" de crecimiento de la función
- Para valores pequeños de n todos los términos pueden ser relevantes

Términos de mayor orden

- El término que más nos importa es n^2
  - Es el término de **mayor orden**
  - Domina al resto de términos a medida que $n \to \infty$
  - Puede haber varios (si la función depende de más de un parámetro)
- Para valores pequeños de n todos los términos influyen
- Mediante la notación asintótica vamos a simplificar y a aislar los términos que más influyen cuando n toma valores muy grandes

Primeras nociones informales

- Supongamos que tenemos dos funciones $f(n)$ y $g(n)$
  - $f(n)$ es *asintóticamente menor* que $g(n)$ cuando:

$$f(n) < g(n) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

  - $f(n)$ es *asintóticamente mayor* que $g(n)$ cuando:

$$f(n) > g(n) \iff g(n) < f(n)$$

  - $f(n)$ es *asintóticamente igual* que $g(n)$ cuando:

$$f(n) = g(n) \iff f(n) \not< g(n) \quad \text{y} \quad g(n) \not< f(n)$$

Órdenes que más aparecen

- Considerados generalmente como "tratables"

$$1 \quad < \quad \log(n) \quad < \quad n \quad < \quad n\log(n) \quad < \quad n^2$$

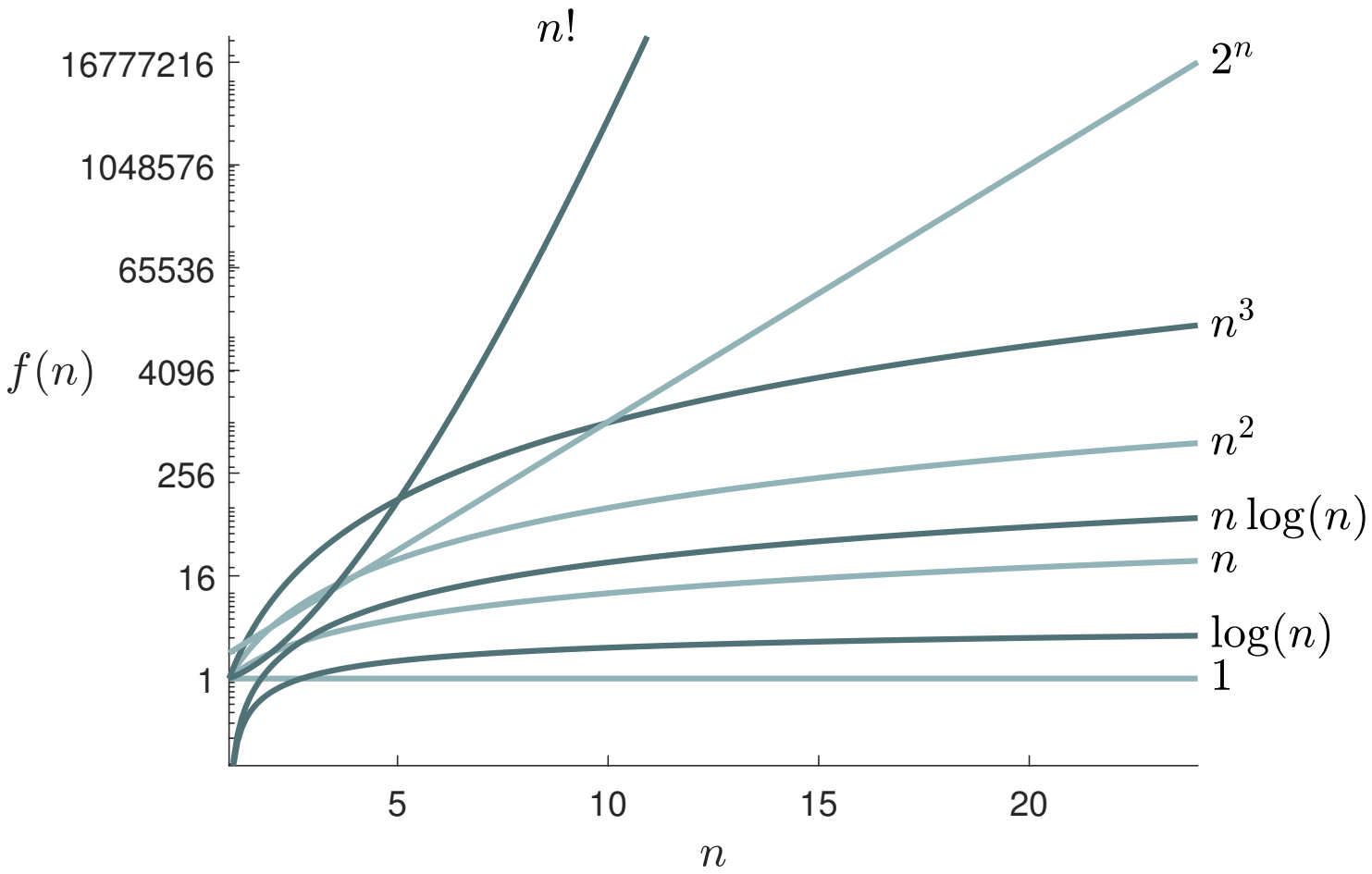
- Considerados generalmente como "intratables"

$$n^2 \quad < \quad n^3 \quad < \quad 2^n \quad < \quad n!$$

- n^2 se encuentra en el límite
  - En la práctica dependerá de n (tamaño de la entrada)

- $n^p < n^q$ para $p < q$
- $f(n) < f(n)\log(n)$
- $n^p < n^p \log(n) < n^{p+1}$
- $n^p < c^n$ para $c > 1$

Curvas de órdenes



Valores de funciones

1	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
1	0	**1**	0	1	1	2	1
1	1	**2**	2	4	8	4	2
1	2	**4**	8	16	64	16	24
1	3	**8**	24	64	512	256	40320
1	4	**16**	64	256	4096	65.536	$2{,}09 \cdot 10^{13}$
1	5	**32**	160	1024	32.768	4.295.967.296	$2{,}63 \cdot 10^{35}$

- Un orden exponencial es extremadamente costoso, incluso frente a ordenes polinómicos
- Un orden factorial es incluso más costoso que un orden exponencial

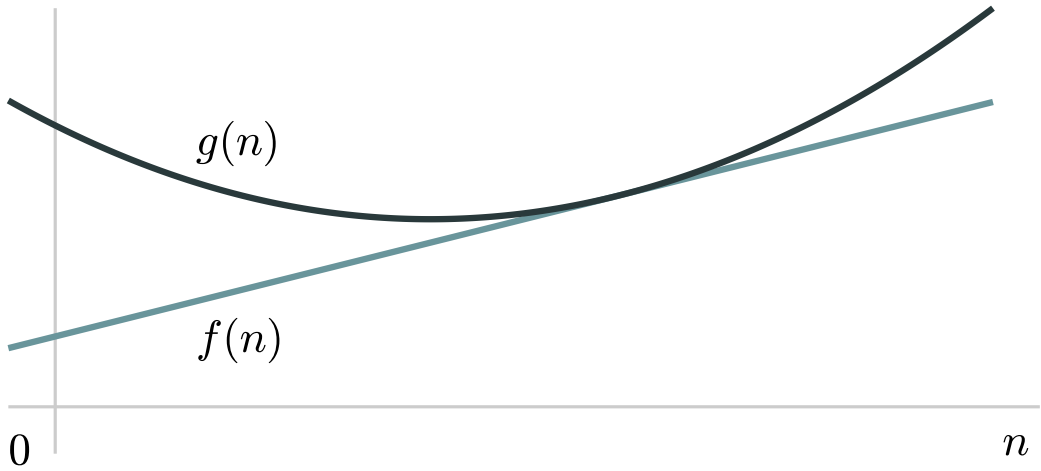
Notaciones asintóticas

Concepto básico de cota superior

- En general, $g(n)$ es cota superior de $f(n)$ cuando: $f(n) \leq g(n)$, $\forall n$.

- Así, dada una función $g(n)$, podemos definir el conjunto de todas las funciones para las que $g(n)$ es cota superior:

$$\Big\{ f(n) : f(n) \leq g(n), \forall n \Big\}$$

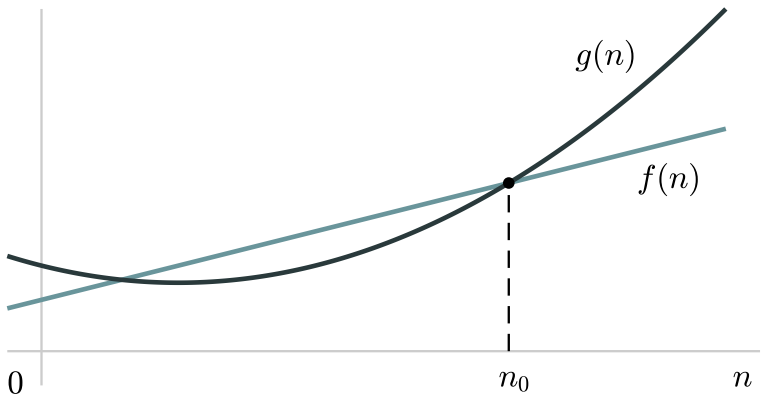


Concepto alternativo de cota superior

- *Definición alternativa*: $g(n)$ es "cota superior para valores grandes" de $f(n)$ si: $f(n) \leq g(n)$, en un intervalo $[n_0, \infty)$, para algún valor de n_0.

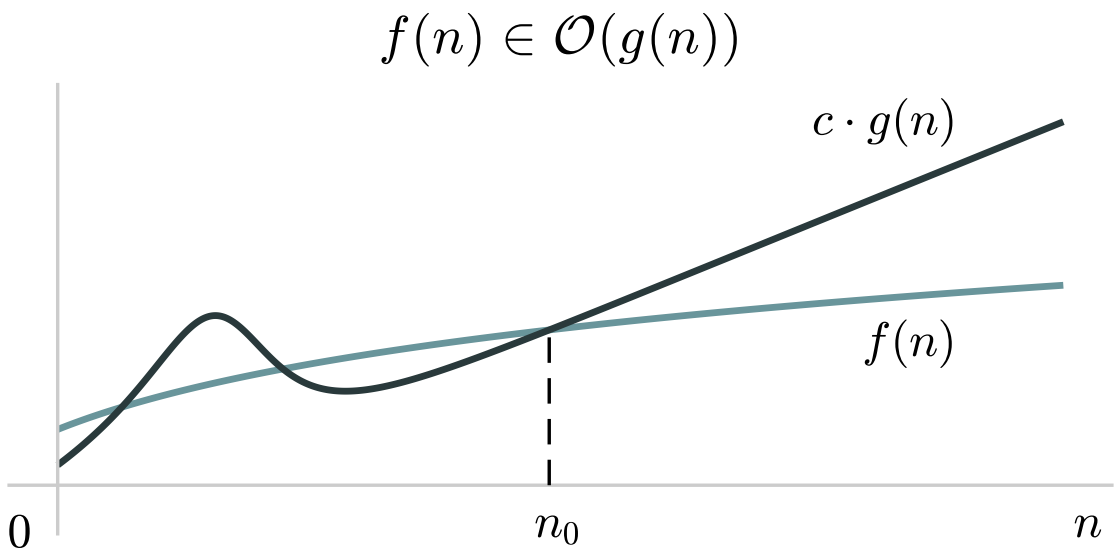
- Bajo esta nueva definición el conjunto de todas las funciones para las que $g(n)$ es cota superior para valores grandes sería :

$$\left\{ f(n) : \exists n_0 \;\middle|\; f(n) \leq g(n),\ \forall n \geq n_0 \right\}$$



Definición formal de cota superior $\mathcal{O}$

$$\mathcal{O}(g(n)) = \left\{ f(n) : \exists\, c > 0 \text{ y } n_0 > 0 \;\middle|\; 0 \leq f(n) \leq c{\cdot}g(n),\ \forall n \geq n_0 \right\}$$



- A partir de un n_0, $c \cdot g(n)$ siempre supera (o iguala) a $f(n)$
- $\mathcal{O}(g(n))$ es un conjunto de funciones

Cota superior $\mathcal{O}$

- Es la cota superior para valores grandes, pero donde
  - Se puede multiplicar $g(n)$ por cualquier constante
  - Es decir, no importan las constantes multiplicativas
- La definición no debe cumplirse para dos valores concretos de c y n_0, lo único que importa para que se cumpla es que **exista** alguna pareja de estos valores
  - Si se cumple la definición habrá infinitas parejas válidas
- La definición también considera:
  - $0 \leq f(n)$ (las funciones con las que trabajaremos miden cantidades no negativas: tiempos, operaciones, memoria, etc.)
  - $c > 0$ y $n_0 > 0$

Cota superior $\mathcal{O}$

- Ejemplos:
  - $2n + 5 \in \mathcal{O}(3n^2 - 8n)$
  - $2n + 5 \in \mathcal{O}(n + 10)$
  - $2n + 5 \in \mathcal{O}(n!)$
  - $2n + 5 \in \mathcal{O}(n) \Longleftarrow$ Querremos la cota superior más baja

- $f(n) \in \mathcal{O}(g(n))$ implica:
  - $f(n)$ es asintóticamente menor o igual que $g(n)$
  - $g(n)$ es una cota superior de $f(n)$ (asintóticamente)

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \text{constante finita}$$

Verificación formal de $f(n) \in \mathcal{O}(g(n))$

- Cuando $f(n) \in \mathcal{O}(g(n))$:
  1. Escoger una constante $c > 0$ adecuada
  2. Plantear la inecuación $f(n) \leq c \cdot g(n)$
  3. Encontrar los intervalos de n en los que se cumple $f(n) \leq c \cdot g(n)$
     - Si uno de los intervalos es del tipo $[n_0, \infty)$ podremos afirmar que $f(n) \in \mathcal{O}(g(n))$
     - c y n_0 constituirán una de las parejas de constantes buscadas
- Cuando $f(n) \notin \mathcal{O}(g(n))$:
  1. Plantear la inecuación $f(n) \leq c \cdot g(n)$
  2. Demostrar que, sea cual sea c, será imposible encontrar un intervalo del tipo $[n_0, \infty)$ en el que se verifique $f(n) \leq c \cdot g(n)$

Ejemplo. Demostrar que: $5n + 2 \in \mathcal{O}(n)$

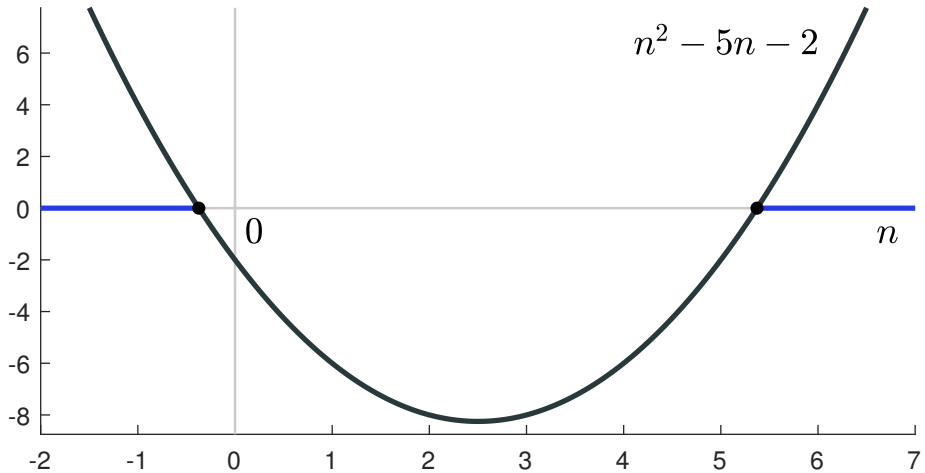
- Hay que encontrar $c > 0$ y $n_0 > 0$ tales que $5n + 2 \leq cn$, $\forall n \geq n_0$

- Para ello, seguimos los siguientes pasos:
  1. Elegimos una constante adecuada (por ejemplo $c = 6$)
     - Para que la parte derecha de la inecuación sea mayor que la izquierda
  2. Planteamos $5n + 2 \leq 6n$ (es decir, $5n + 2 \leq cn$)
  3. Determinamos cuándo se cumple la inecuación. Lo ideal es poder despejar la n, y en este caso se puede: $2 \leq 6n - 5n \Rightarrow 2 \leq n$. Por tanto, se cumple en el intervalo $[2, \infty)$, y podríamos escoger $n_0 = 2$ (cualquier valor mayor también sería válido).

- Al haber encontrado c y n_0 afirmamos que $5n + 2 \in \mathcal{O}(n)$

Ejemplo. Demostrar que: $5n + 2 \in \mathcal{O}(n^2)$

1. Elegir una constante adecuada (por ejemplo $c = 1$)
2. Plantear $5n + 2 \leq n^2$ (es decir, $5n + 2 \leq 1 \cdot n^2$)
3. Buscar qué valores de n hacen que se cumpla que $5n + 2 \leq n^2$
   - Ahora no podemos despejar n
   - Analizamos la desigualdad $n^2 - 5n - 2 \geq 0$



Ejemplo. Demostrar que: $5n + 2 \in \mathcal{O}(n^2)$

- continuación...
  - $n^2 - 5n - 2$ es una función cuadrática (convexa) con raíces en $-0,37$ y $5,37$
  - Por tanto, siempre será positiva para $n \geq 6$
  - Para $c = 1$ y $n_0 = 5{,}37$ se cumplen las condiciones de la definición y queda demostrado
    - n_0 también puede ser cualquier valor mayor que $5{,}37$, por ejemplo, $n_0 = 6$ o $n_0 = 1000$.
- Si escogemos $c = 5$, $n_0 = 2$ es suficiente
- Si escogemos $c = 8$, $n_0 = 1$ es suficiente

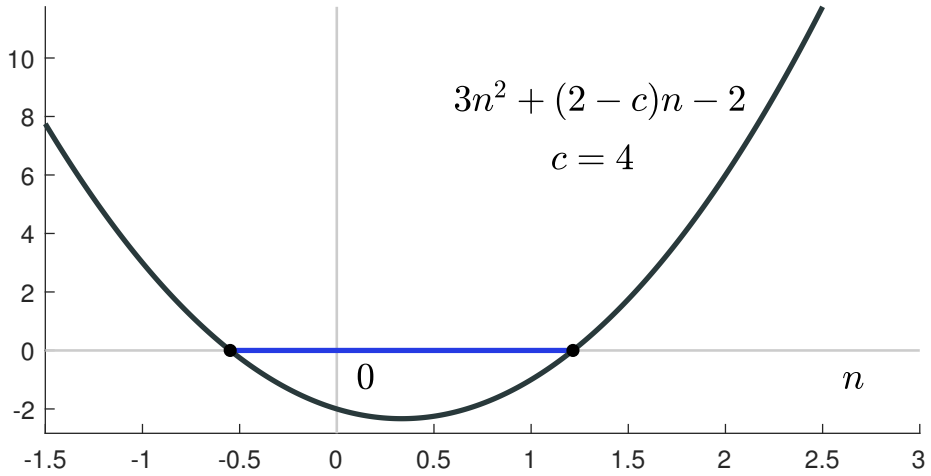
Ejemplo. Verificar si $3n^2 + 2n - 2 \in \mathcal{O}(n)$

- En este caso no vamos a poder encontrar las constantes (ya que $3n^2 + 2n - 2 \notin \mathcal{O}(n)$)

1. Consideramos $3n^2 + 2n - 2 \leq cn$
2. Cojamos la constante c que cojamos $3n^2 + 2n - 2 \leq cn$ no se va a verificar en un intervalo de tipo $[n_0, \infty)$
   - La inecuación se puede expresar como $3n^2 + (2 - c)n - 2 \leq 0$
   - La función $3n^2 + (2 - c)n - 2$ es una parábola convexa, que crece hasta el $+\infty$ según aumenta n, independientemente de c
   - Por tanto, no va a ser negativa **siempre** a partir de ningún n_0
   - Es imposible encontrar una pareja de constantes c y n_0. Por tanto, $3n^2 + 2n - 2 \notin \mathcal{O}(n)$).

Verificar si $3n^2 + 2n - 2 \in \mathcal{O}(n)$

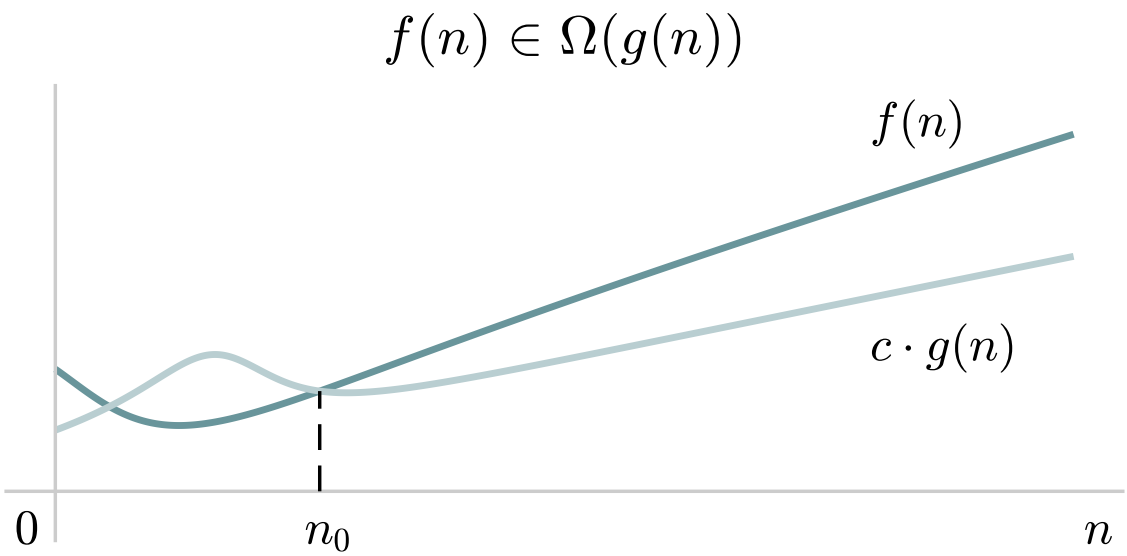
- Gráficamente:



- Sea cual sea c (en el ejemplo $c = 4$) la parábola siempre será convexa (con forma de "U"), ya que el coeficiente asociado a n^2 es positivo
- $3n^2 + (2 - c)n - 2 \leq 0$ se cumplirá, como mucho, en un intervalo finito, y nunca en uno de tipo $[n_0, \infty)$

Definición formal de cota inferior Ω

$$\Omega(g(n)) = \left\{ f(n) : \exists\, c > 0 \text{ y } n_0 > 0 \;\middle|\; 0 \leq c{\cdot}g(n) \leq f(n),\ \forall n \geq n_0 \right\}$$



- A partir de n_0, $f(n)$ siempre supera (o iguala) a $c \cdot g(n)$

Cota inferior Ω

- Ejemplos:
  - $2n + 5 \in \Omega(3 \log n)$
  - $2n + 5 \in \Omega(4n + 10)$
  - $2n + 5 \in \Omega(1)$
  - $2n + 5 \in \Omega(n) \Longleftarrow$ Querremos la cota inferior más alta

- $f(n) \in \Omega(g(n))$ implica:
  - $f(n)$ es asintóticamente mayor o igual que $g(n)$
  - $g(n)$ es una cota inferior de $f(n)$ (asintóticamente)

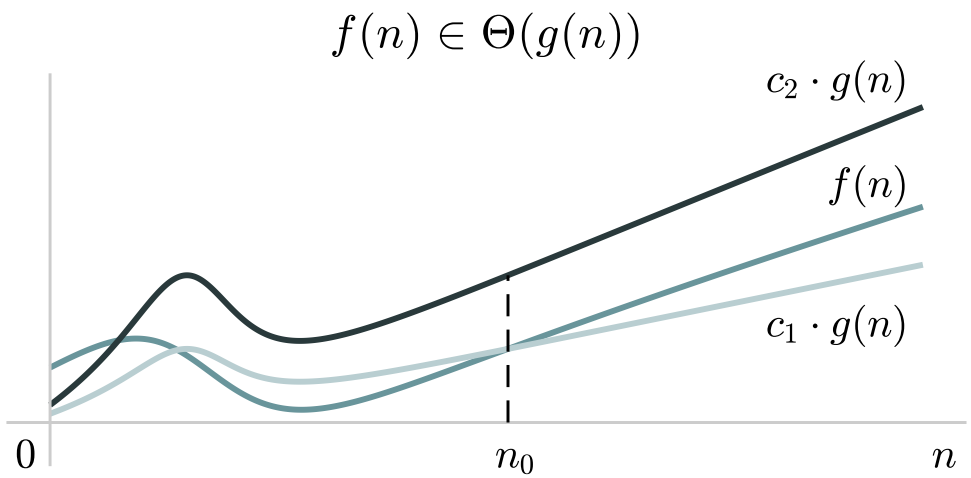
$$\lim_{n \to \infty} \frac{f(n)}{g(n)} > 0 \qquad \text{constante} > 0, \text{ o } \infty$$

Ejemplo. Verificar si $3n^2 + 2 \in \Omega(n)$

- Mismos pasos que para $\mathcal{O}$, pero usando $f(n) \geq c \cdot g(n)$
- Para $3n^2 + 2 \in \Omega(n)$:
  1. Elegimos una constante adecuada (por ejemplo $c = 5$)
  2. Planteamos $3n^2 + 2 \geq 5n$ (es decir, $3n^2 + 2 \geq cn$)
  3. Determinamos qué valores de n satisfacen la inecuación
     - Analizamos la desigualdad $3n^2 - 5n + 2 \geq 0$
     - $3n^2 - 5n + 2$ es una función cuadrática (convexa) con raíces en $2/3$ y 1
     - Por tanto, se cumple la inecuación en el intervalo $[1, \infty)$
     - Para $c = 5$ y $n_0 = 1$ se cumplen las condiciones de la definición y podemos afirmar que $3n^2 + 2 \in \Omega(n)$

Definición formal de cota ajustada Θ

$$\Theta(g(n)) = \Big\{ f(n) : \exists\ c_1 > 0,\ c_2 > 0 \text{ y } n_0 > 0 \ \Big|\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n),\ \forall n \geq n_0 \Big\}$$



- A partir de n_0, $f(n)$ siempre queda en medio de $c_1 g(n)$ y $c_2 g(n)$

Cota ajustada Θ

- Ejemplos:
  - $2n + 5 \in \Theta(8n + 10)$
  - $2n + 5 \in \Theta(n)$

- $f(n) \in \Theta(g(n))$ implica:
  - $f(n)$ es asintóticamente igual que $g(n)$
  - $g(n)$ es una cota ajustada de $f(n)$ (asintóticamente)

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \text{constante} > 0$$

$$f(n) \in \Theta(g(n)) \iff \begin{cases} f(n) \in \mathcal{O}(g(n)) \\ \text{y} \\ f(n) \in \Omega(g(n)) \end{cases}$$

Ejemplo. Demostrar $n^2/2 - 3n \in \Theta(n^2)$

- Tenemos que demostrar tanto $n^2/2 - 3n \in \mathcal{O}(n^2)$ como $n^2/2 - 3n \in \Omega(n^2)$
- Se busca que $c_1 n^2 \leq n^2/2 - 3n \leq c_2 n^2$
- Encontramos, por ejemplo: $c_1 = 1/14$ para $n \geq 7$ (Ω)
- Y, por ejemplo: $c_2 = 1/2$ para $n \geq 1$ ($\mathcal{O}$)
- Según la definición habríamos encontrado $c_1 = 1/14$, $c_2 = 1/2$, y $n_0 = 7$

Otras notaciones asintóticas: o y ω

- $f(n) \in o(g(n))$ (cota superior estricta)
  - El orden de $g(n)$ es mayor, estrictamente, que el de $f(n)$
  - $f(n) < g(n)$ (asintóticamente)
    - Si $f(n) \in \mathcal{O}(g(n))$ entonces $f(n) \leq g(n)$ (asintóticamente)

- $f(n) \in \omega(g(n))$ (cota inferior estricta)
  - El orden de $g(n)$ es menor, estrictamente, que el de $f(n)$
  - $f(n) > g(n)$ (asintóticamente)
    - Si $f(n) \in \Omega(g(n))$ entonces $f(n) \geq g(n)$ (asintóticamente)

Funciones de dos parámetros

- Hay problemas y algoritmos cuya complejidad computacional depende de varios parámetros
- Ejemplo: mezclar dos vectores ordenados de longitudes n y m
  - $T(n, m) \in \Theta(n + m)$
- Definición formal de $\mathcal{O}$ para funciones de dos parámetros:

$$\mathcal{O}(g(n,m)) = \Big\{ f(n,m) : \exists\ c > 0,\ n_0 > 0,\ \text{y}\ m_0 > 0 \ \Big|\ 0 \leq f(n,m) \leq c \cdot g(n,m),\ \forall n \geq n_0,\ \text{y}\ m \geq m_0 \Big\}$$

Funciones de dos parámetros

- En la práctica usaremos límites para determinar si un orden es mayor que otro

$$f(n,m) > g(n,m) \iff \begin{cases} \lim\limits_{n\to\infty} \dfrac{g(n,m)}{f(n,m)} = 0 \quad \text{y} \quad \lim\limits_{m\to\infty} \dfrac{g(n,m)}{f(n,m)} \neq \infty \\ \text{o} \\ \lim\limits_{m\to\infty} \dfrac{g(n,m)}{f(n,m)} = 0 \quad \text{y} \quad \lim\limits_{n\to\infty} \dfrac{g(n,m)}{f(n,m)} \neq \infty \end{cases}$$

Simplificación

- Simplificar $\Theta(3m^2n + m^3 + 10mn + 2n^2)$

1. Eliminar constantes

$$\Theta(m^2n + m^3 + mn + n^2)$$

2. Simplificar términos "contenidos" en otros
   - $m^2n > mn$, por tanto, se puede eliminar el término mn

$$\lim_{n\to\infty} \frac{mn}{m^2n} = \frac{1}{m} \neq \infty \quad \text{y} \quad \lim_{m\to\infty} \frac{mn}{m^2n} = 0$$

$$\boxed{\Theta(m^2n + m^3 + n^2)}$$

   - Si probamos las tres combinaciones de parejas de funciones que aparecen en la fórmula final veremos que ninguna es superior a otra

Aspectos adicionales

Aspectos adicionales

- Sea ρ alguna medida de complejidad computacional asintótica:
  - Las constantes no importan

$$\rho(kg(n)) = \rho(g(n))$$

  - Término de mayor orden de un polinomio

$$\rho(a_m x^m + a_{m-1} x^{m-1} + \ldots + a_1 x^1 + a_0) = \rho(x^m)$$

  - La base de los logaritmos no importa

$$\rho(\log_x g(n)) = \rho(\frac{\log_y g(n)}{\log_y x}) = \rho(\frac{1}{\log_y x} \log_y g(n)) = \rho(\log_y g(n)) = \rho(\log g(n))$$

Aspectos adicionales

- $\mathcal{O}$, Ω, y Θ definen **conjuntos**
  - Lo correcto es escribir $f(n) \in \mathcal{O}(g(n))$
  - A veces se escribe $f(n) = \mathcal{O}(g(n))$, aunque es un "abuso" de notación
  - Y lo mismo con Ω y Θ
- Las funciones ($f(n)$, $g(n)$, $f(n, m)$, etc.) son siempre positivas
- Es un error decir que si $f(n) \in \mathcal{O}(g(n))$, entonces $f(n)$ tarda al menos $g(n)$
  - Al contrario, tarda como mucho $g(n)$ ($g(n)$ es **cota superior**)
- Con estas definiciones las constantes no influyen: se proporcionan cotas hasta un factor constante multiplicativo
- Hay notaciones en las que tratan a los términos logarítmicos como irrelevantes también

Comentarios adicionales

- ¡Que un algoritmo tarde $\mathcal{O}(n^2)$, $\Omega(n^2)$, o $\Theta(n^2)$ para algunas entradas no quiere decir que tarde $\mathcal{O}(n^2)$, $\Omega(n^2)$, o $\Theta(n^2)$ para todas, o en general!
- Cuando hablamos de $\mathcal{O}$ normalmente lo hacemos en referencia al **peor caso**, que es cuando un algoritmo tarda más
  - En ese caso damos una cota **superior** del tiempo o número de operaciones
  - Es como una "garantía" de que el coste nunca va a superar la cota proporcionada
- Cuando se indica una cota, siempre hay que asociarla a un determinado tipo de entrada:
  - Caso mejor, peor, o medio

Comentarios adicionales

- Un algoritmo debe procesar todos los bits de todos los ($m = 2^n$) números de n bits. Para $n = 3$ procesa 24 bits):

0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

- Tamaño de entrada:
  - n bits $\Rightarrow \Theta(n \cdot 2^n)$ (¿intratable?)
  - m números $\Rightarrow \Theta(m \cdot \log(m))$ (¿tratable?)
- Ambas expresiones son idénticas

Ordenación

Ordenación en tiempo lineal

- Existen algoritmos de ordenación que tardan $\mathcal{O}(n)$, pero no son generales (no pueden ordenar cualquier tipo de datos)
- *Radix-sort*, *Bucket-sort*, *Counting-sort*...
- *Counting-sort*
  - Los elementos a ordenar son enteros y pertenecen al intervalo $[0, k]$
  - Si $k \in \mathcal{O}(n)$, entonces el algoritmos tarda $\Theta(n)$
  - Usa tres vectores:
    - A[1..n], es el vector de entrada
    - B[1..n], es el vector de salida
    - C[0..k], es un vector auxiliar

Idea del counting-sort

- Se recorre la secuencia A y se cuenta el número de veces que aparece cada entero

A	2	5	3	0	2	3	0	3	n = 8

C	2	0	2	3	0	1	k = 5
	0	1	2	3	4	5	

- Hay varias formas de obtener el vector ordenado

B	0	0	2	2	3	3	3	5

- A continuación se describe un algoritmo eficiente

Counting-sort

```
Counting-sort(A)

FOR i=0..k                          Θ(k)
  C[i] = 0

FOR j=1..length(A)                  Θ(n)
  C[A[j]]++

FOR j=1..k                          Θ(k)
  C[i] = C[i] + C[i-1]

FOR j=length(A)..1                  Θ(n)
  B[C[A[j]]] = A[j]
  C[A[j]]--
```

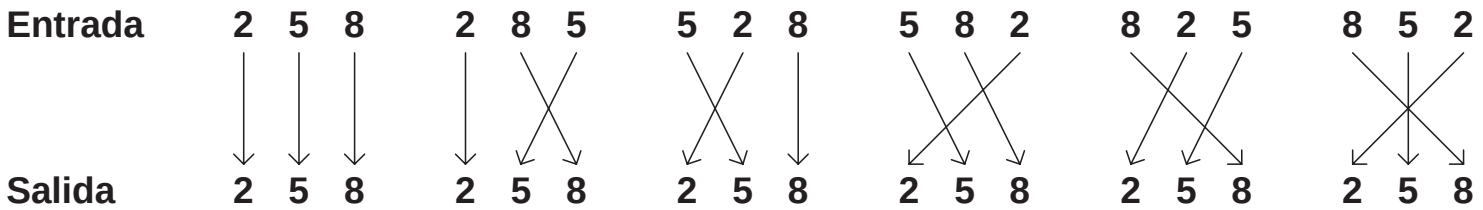
- $T(n) \in \Theta(n+k)$
- Si $k \in O(n)$, entonces $T(n) \in \Theta(n)$

Ordenación por comparación

- Algoritmos basados en ordenación por comparación:
  - Se basan en comparaciones del tipo $<$, $\leq$, $=$, $\geq$ y $>$
  - No se determina el orden de los elementos de otra manera
  - No necesitan conocer los valores de los elementos a ordenar
    - Sus valores y distribución son irrelevantes
    - Solo interesa el puesto en la ordenación (1º, 2º,...)
  - Son generales: pueden ordenar datos de cualquier tipo, siempre que existan funciones $<$, $\leq$, $=$, $\geq$ y $>$ para compararlos
    - Enteros, reales, cadenas de caracteres, etc.
- Se analizan en el caso mejor, peor, medio...
- Insert-sort, Bubble-sort, Select-sort, Merge-sort, Quicksort, Heap-sort...

Algoritmos de ordenación y permutaciones

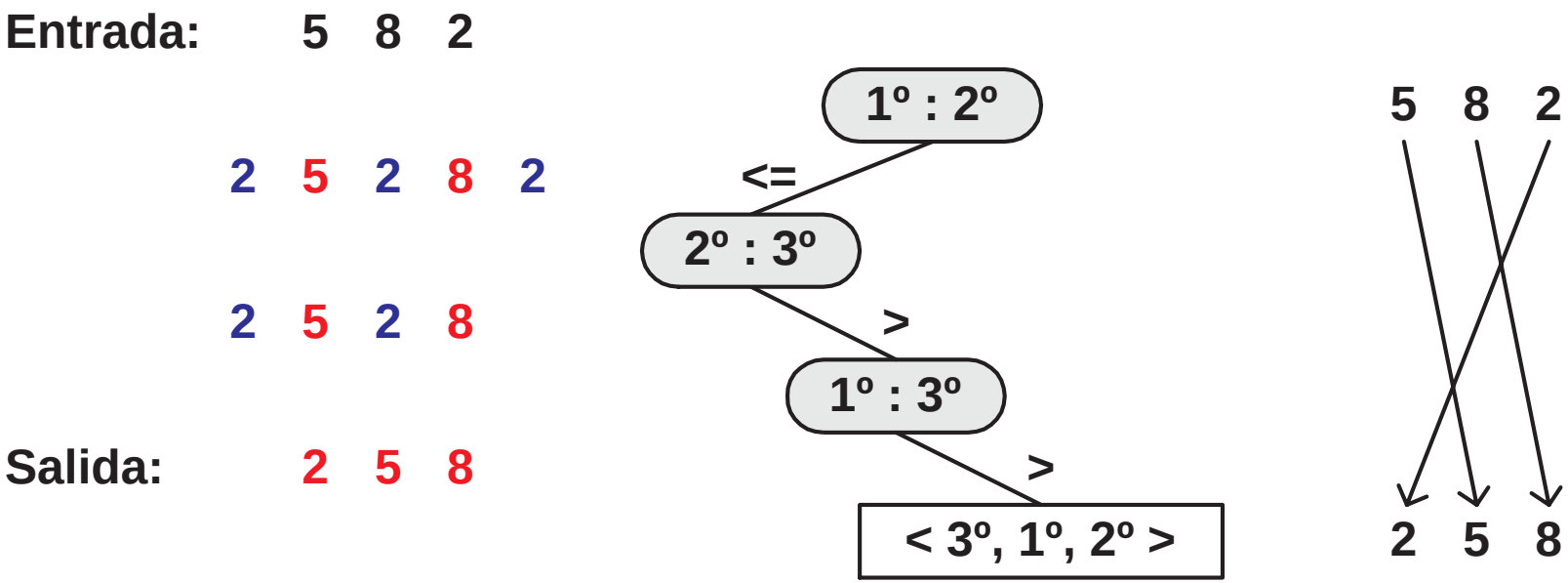
- Los algoritmos de ordenación por comparación tienen que ser capaces de generar cualquier permutación de un vector:



- Para un vector de n elementos, hay $n!$ posibles permutaciones

Algoritmos de ordenación y comparaciones

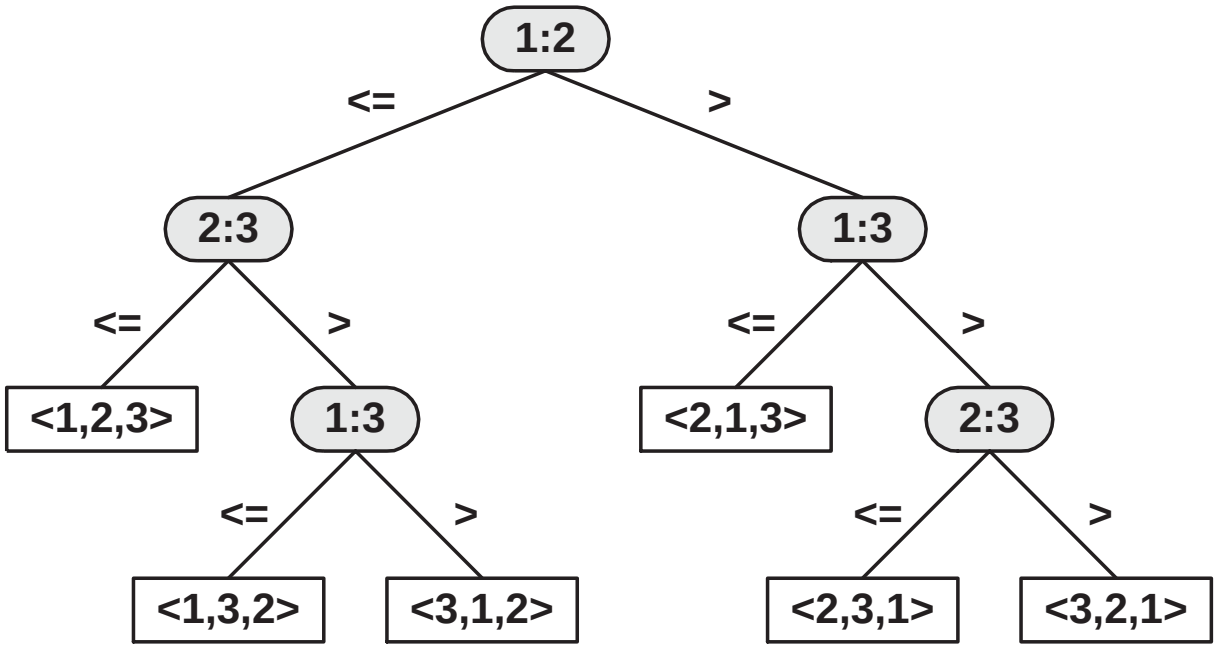
- Deben realizar varias comparaciones hasta hasta determinar la permutación correcta para cualquier entrada



- Incertidumbre
- Ordenados correctamente

Algoritmos de ordenación como árboles de decisión

- Los algoritmos de ordenación pueden verse de manera abstracta en términos de un árbol de decisión:



X:Y	Se comparan los elementos Xº e Yº de la secuencia original
<X,Y,Z>	Secuencia final <Xº, Yº, Zº>

Cota inferior para algoritmos de ordenación

- Cualquier permutación de los n elementos debe aparecer como hoja del árbol
- Hay $n!$ permutaciones posibles
- La profundidad o altura máxima de una hoja determina en nº de comparaciones en el peor caso
- Nos interesaría diseñar un algoritmo cuyo árbol tuviera la profundidad mínima
- Una cota inferior de la altura de árbol en el peor caso es una cota inferior del nº de comparaciones para cualquier algoritmo de ordenación por comparación

Cota inferior para algoritmos de ordenación

Teorema

Cualquier algoritmo de ordenación por comparaciones necesita $\Omega(n \log n)$ *comparaciones en el peor caso*

Demostración

Tenemos un árbol de decisión de altura h *con* l *hojas. Hay que demostrar que* $h \in \Omega(n \log n)$

- $n! \leq l$, *tiene que haber por lo menos* $n!$ *hojas*
- $l \leq 2^h$, *un árbol binario de altura* h *tiene como mucho* 2^h *hojas*

$$n! \leq l \leq 2^h \quad \Rightarrow \quad n! \leq 2^h$$

$$\log_2(n!) \leq \log_2(2^h) \quad \Rightarrow \quad h \geq \log_2(n!)$$

- *Como* $h \geq \log_2(n!)$, *nos interesa conocer el orden de complejidad* $\log(n!)$, *y nos basta con conocer una cota inferior*

Cota inferior para algoritmos de ordenación

Demostración

- *Queda por demostrar que* $\log(n!) \in \Omega(n\log(n))$. *Para ello usamos la definición formal. Antes transformamos* $\log(n!)$ *en un sumatorio:*

$$\log(n!) = \sum_{i=1}^{n} \log(i) = \sum_{i=2}^{n} \log(i) \geq cn\log(n)$$

- *Problema: ¿Cómo expresar* $\sum_{i=2}^{n} \log(i)$ *mediante una fórmula?*
- *Tenemos que recurrir a cotas:*

$$\sum_{i=2}^{n} \log(i) \geq \underbrace{g(n)}_{\substack{\text{Cota inferior de} \\ \sum_{i=2}^{n} \log(i)}} \geq cn\log(n)$$

- *Si demostramos* $g(n) \geq cn\log(n)$, *entonces* $\sum_{i=2}^{n} \log(i) \geq cn\log(n)$

Cota inferior para algoritmos de ordenación

Demostración

- *Cota inferior para* $\sum_{i=2}^{n} \log(i)$*:*
  - *Como* $\log(x)$ *es creciente:*

$$\sum_{i=2}^{n} \log(i) \geq \int_{1}^{n} \log(x)dx$$

- *Integral por partes (asumimos que la base del logaritmo es* e*)*

$$\int \underbrace{\log(x)}_{u}\underbrace{dx}_{dv} = \underbrace{x}_{v} \cdot \underbrace{\log(x)}_{u} - \int \underbrace{x}_{v}\underbrace{\frac{1}{x}dx}_{du}$$

$$= x\log(x) - \int dx = x\log(x) - x$$

Cota inferior para algoritmos de ordenación

Demostración

- *Cota inferior para* $\sum_{i=2}^{n} \log(i)$*:*

$$\sum_{i=2}^{n} \log(i) \geq \int_{1}^{n} \log(x)dx = \Big[x\log(x) - x\Big]_{1}^{n} = n\log(n) - n + 1$$

- *Volviendo a la demostración, buscamos una* c *tal que* $n\log(n) - n + 1 \geq cn\log(n)$ *se cumpla en un intervalo* $[n_0, \infty]$*:*
  1. *Cogemos* $c = 1/2$
  2. *Planteamos* $n\log(n) - n + 1 \geq \frac{1}{2}n\log(n)$
  3. *Hallamos un intervalo* $[n_0, \infty]$ *donde se satisfaga la inecuación*

Cota inferior para algoritmos de ordenación

Demostración

$$n\log(n) - n + 1 \geq \tfrac{1}{2} n\log(n)$$

$$\tfrac{1}{2} n\log(n) - n + 1 \geq 0$$

$$n(\tfrac{1}{2}\log(n) - 1) + 1 \geq 0$$

Se verifica si $\tfrac{1}{2}\log(n) - 1 \geq 0$

$$\log(n) \geq 2$$

$$n \geq e^2 \quad \Rightarrow \quad \textit{Se verifica en } [e^2, \infty)$$

- *Por tanto,* $n\log(n) - n + 1 \in \Omega(n\log(n))$
- *Como* $n\log(n) - n + 1$ *es cota inferior de* $\sum_{i=2}^{n} \log(i)$,
- *y* $h \geq \sum_{i=2}^{n} \log(i)$, *tenemos:*

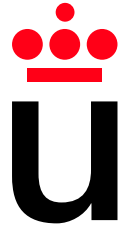
$$\boxed{h \in \Omega(n\log(n))}$$

Tema 3.1
Análisis de algoritmos iterativos

Diseño y Análisis de Algoritmos

Manuel Rubio Sánchez

13 de mayo de 2024



Universidad
Rey Juan Carlos

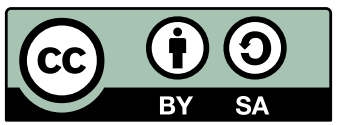
Copyright

©2024 Manuel Rubio Sánchez

Algunos derechos reservados.

Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en:

https://creativecommons.org/licenses/by-sa/4.0/deed.es.



Contenido

Introducción

Eficiencia de Algoritmos

- ¿Qué recursos necesitan los algoritmos?
  - Espacio (memoria)
  - Tiempo (número de operaciones)
  - Otros:
    - Ancho de banda

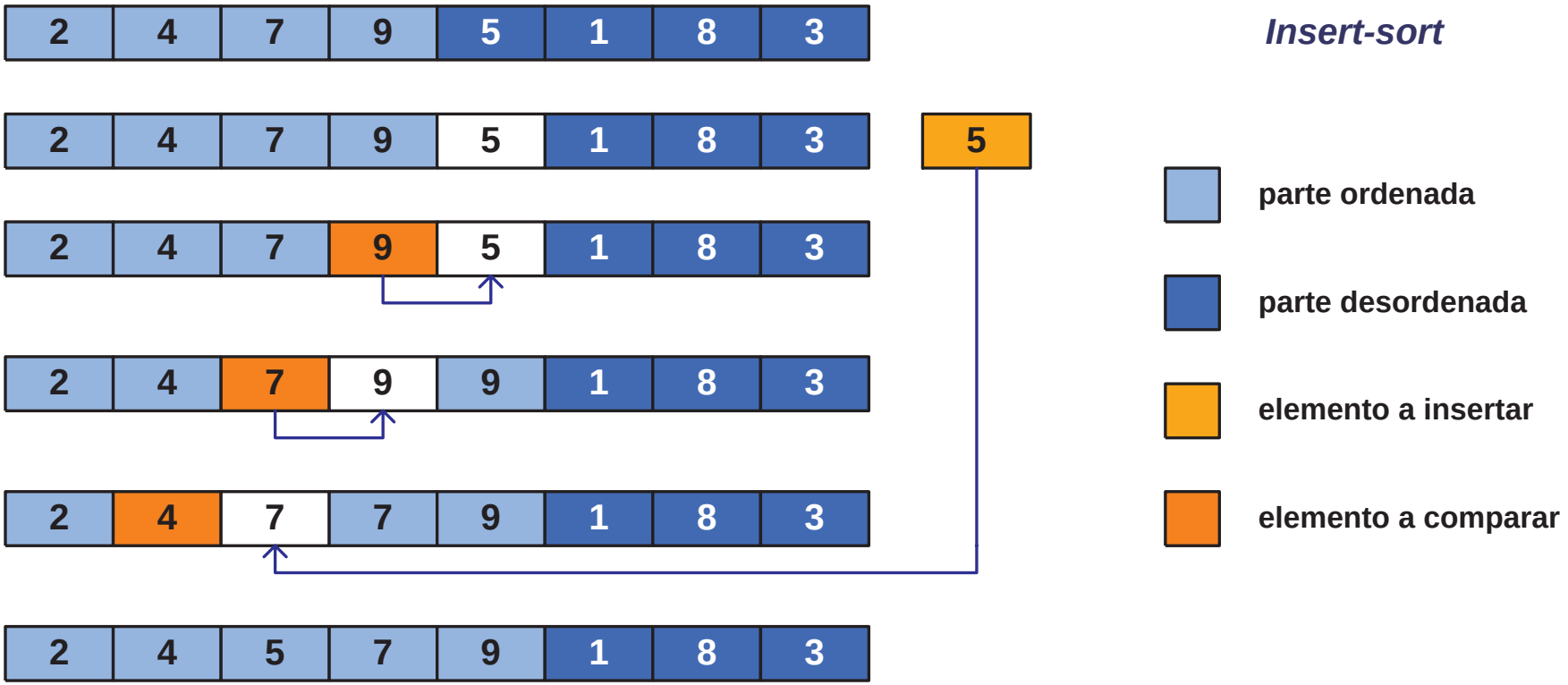
Eficiencia de Algoritmos

- Características del software
  - Amigabilidad
  - Buen estilo
  - Comentarios
  - Corrección
  - Escalabilidad
  - Funcionalidad
  - Mantenibilidad
  - Modularidad
  - Rendimiento
  - Robustez
  - Seguridad
  - Simplicidad
  - Tiempo de programación
  - ...
- La **eficiencia** ayuda a alcanzar algunas de estas características

Análisis de eficiencia de algoritmos iterativos

Un ejemplo: Insert-sort

- Algoritmo de ordenación por inserción directa (*insert-sort*)



Un ejemplo: Insert-sort

- Código y coste por línea $(n = \texttt{len(a)})$

		Coste	Nº de veces
0	`def insert_sort(a):`		
1	`    for j in range(1,len(a)):`	C_1	n
2	`        val = a[j]`	C_2	$n-1$
3	`        # Inserta a[j] en la lista`		
	`        # ordenada a[0:j-1]`	0	$n-1$
4	`        i = j-1`	C_4	$n-1$
5	`        while (i>=0) and (a[i]>val):`	C_5	$\sum_{j=1}^{n-1}(t_j+1)$
6	`            a[i+1] = a[i]`	C_6	$\sum_{j=1}^{n-1} t_j$
7	`            i = i-1`	C_7	$\sum_{j=1}^{n-1} t_j$
8	`        a[i+1] = val`	C_8	$n-1$

  - t_j = nº de veces que la condición del bucle de la línea 5 es verdadera (que se ejecuta el cuerpo del bucle), para un determinado valor de j

Un ejemplo: Insert-sort

- Ignoramos el coste concreto de cada operación básica
- Cada línea l tardará un determinado tiempo o coste, que denotamos por C_l
- La función de coste o tiempo, en función del tamaño del vector n es:

$$T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5 \sum_{j=1}^{n-1}(t_j + 1)$$

$$+ C_6 \sum_{j=1}^{n-1} t_j + C_7 \sum_{j=1}^{n-1} t_j + C_8(n-1)$$

- t_j depende del vector de entrada particular

Un ejemplo: Insert-sort

- Mejor caso (vector ordenado de menor a mayor)
- $t_j = 0$, para todo j

$$T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5(n-1) + C_8(n-1)$$

$$= (C_1 + C_2 + C_4 + C_5 + C_8)n - (C_2 + C_4 + C_5 + C_8)$$

$$= K_1 n + K_2 \quad \in \quad \Theta(n)$$

- El orden de $T(n)$ es lineal
- ¡Para el orden, el valor de las constantes no importa!

Un ejemplo: Insert-sort

- Peor caso (vector ordenado de mayor a menor)
- $t_j = j$ (valor máximo para cada j)
- En primer lugar observamos que:

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

$$\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$$

Un ejemplo: Insert-sort

- Sustituyendo:

$$T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5\left(\frac{n(n+1)}{2} - 1\right) +$$

$$C_6\left(\frac{n(n-1)}{2}\right) + C_7\left(\frac{n(n-1)}{2}\right) + C_8(n-1) =$$

$$= \left(\frac{C_5}{2} + \frac{C_6}{2} + \frac{C_7}{2}\right) n^2 + \left(C_1 + C_2 + C_4 + \frac{C_5}{2} - \frac{C_6}{2} - \frac{C_7}{2} + C_8\right) n -$$

$$(C_2 + C_4 + C_5 + C_8) = K_1 n^2 + K_2 n + K_3 \quad \in \quad \Theta(n^2)$$

- El orden de $T(n)$ es cuadrático

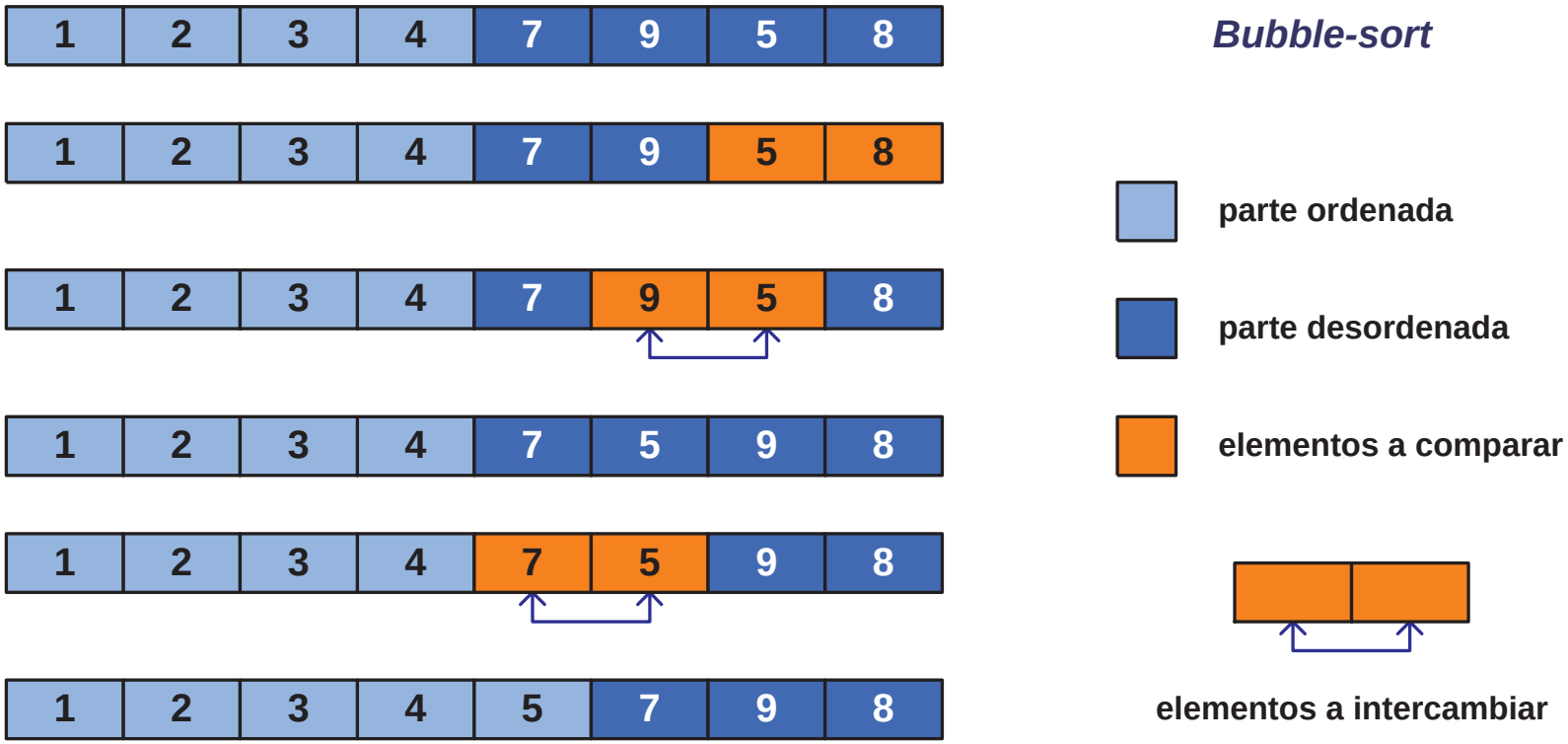
Caso medio frente a caso peor

- En ocasiones se puede calcular el caso medio, pero nos centraremos en el estudio de caso peor debido a:
  - El caso peor representa un cota superior para cualquier entrada, dándonos una garantía de que el algoritmo no tardará más
  - El caso peor suele ocurrir con frecuencia
    - En una búsqueda, ocurre cuando el elemento buscado no se encuentra
  - El tiempo medio suele ser tan malo como el peor en términos asintóticos
    - Si en el *insert-sort* hay que insertar el elemento hasta la posición $j/2$, el tiempo también sale cuadrático

Procedimiento general basado en sumatorios

Otro ejemplo: Bubble-sort (variante)

- Algoritmo de ordenación "burbuja" (*bubble-sort*)
  - Variante (tras cada iteración los elementos más pequeños quedan correctamente ordenados)



Otro ejemplo: Bubble-sort (variante)

```
1  def bubble_sort(a):
2
3      n = len(a)
4
5      for i in range(0,n-1):                    # i=0..n-2
6
7          for j in range(n-1, i, -1):           # j=n-1..i+1
8
9              if(a[j-1]>a[j]):
10
11                 aux = a[j-1]
12                 a[j-1] = a[j]
13                 a[j] = aux
```

Operaciones de un bucle de n iteraciones

- 1 inicialización
- n comparaciones
- Tiempo de ejecutar el cuerpo del bucle n veces
- n incrementos
- 1 última comparación para salir del bucle

$$T_{\text{bucle}} = 1_{\text{inic.}} + \sum^{n}(1_{\text{comp.}} + T_{\text{cuerpo}} + 1_{\text{incr.}}) + 1_{\text{última comp.}}$$

Tiempo en el mejor caso

- 1ª forma: simplemente contando operaciones
  - For 1:

$$1_{\text{inic.}} + (n-1)_{\text{incr.}} + (n-1)_{\text{comp.}} + 1_{\text{última comp.}} = 2n$$

  - For 2:

$$(n-1)_{\text{inic.}} + \big((n-1)+(n-2)+\cdots+1\big)_{\text{comp.}} + (n-1)_{\text{última comp.}}$$

$$\big((n-1)+(n-2)+\cdots+1\big)_{\text{decr.}} + \big((n-1)+(n-2)+\cdots+1\big)_{\text{comp. del IF}}$$

  - Sumando todo:

$$T_{\text{mejor}}(n) = 1+2n+2(n-1)+3\frac{n(n-1)}{2} = \frac{3}{2}n^2+\frac{5}{2}n-1 \quad \in \quad \Theta(n^2)$$

Tiempo en el mejor caso

- 2ª forma: usando la fórmula del bucle

$$T_{\text{mejor}}(n) = 1+1+\sum_{i=0}^{n-2}\left[1+1+\sum_{j=i+1}^{n-1}(1+1+1)+1+1\right]+1$$

  - For 1:

$$T_{\text{mejor}}(n) = 1_{\text{inic.}} + \sum_{i=0}^{n-2}[1_{\text{comp.}} + T_{\text{For 2}} + 1_{\text{incr.}}] + 1_{\text{última comp.}}$$

  - For 2:

$$T_{\text{mejor}}(n) = 1_{\text{inic.}} + \sum_{j=i+1}^{n-1}[1_{\text{comp.}} + 1_{\text{comp. IF}} + 1_{\text{incr.}}] + 1_{\text{última comp.}}$$

Tiempo en el mejor caso

- Simplificando:

$$T_{\text{mejor}}(n) = 3 + \sum_{i=0}^{n-2} \left[4 + \sum_{j=i+1}^{n-1} 3 \right]$$

- El sumatorio interno suma 3, $(n - i - 1)$ veces. Por tanto:

$$T_{\text{mejor}}(n) = 3 + 4(n-1) + \sum_{i=0}^{n-2} 3(n-i-1)$$

$$= 3 + 4n - 4 + 3n(n-1) - 3\frac{(n-1)(n-2)}{2} - 3(n-1)$$

$$T_{\text{mejor}}(n) = \frac{3}{2}n^2 + \frac{5}{2}n - 1 \quad \in \quad \Theta(n^2)$$

Ejemplo con tres bucles

- Calculad el número de operaciones ($T(n)$) que realiza el siguiente código:

```
1 def codigo(n):
2     for i in range(0,n):
3         for j in range(i+1,n+1):
4             if (condicion(i)):    # una operación
5                 for k in range(0,j):
6                     procesa(i,j,k);  # dos operaciones
```

  - Se considera que las inicializaciones, comparaciones, e incrementos siempre necesitan una sola operación.

Ejemplo con tres bucles

El código consta de 3 bucles, que podemos descomponer de la siguiente manera:

$$T(n) = 1 + \sum_{i=0}^{n-1}(1 + T_{\text{For 2}} + 1) + 1$$

$$T_{\text{For 2}} = 1 + \sum_{j=i+1}^{n}(1 + T_{\text{If}} + 1) + 1$$

$$T_{\text{If}} = \begin{cases} 1 & \text{en el mejor caso} \\ 1 + 1 + \sum_{k=0}^{j-1}(1 + 2 + 1) + 1 = 3 + 4j & \text{en el peor caso} \end{cases}$$

Ejemplo con tres bucles – mejor caso

Si $T_{\text{If}} = 1$, entonces sustituyendo en $T_{\text{For 2}}$ tenemos:

$$T_{\text{For 2}} = 2 + \sum_{j=i+1}^{n} 3 = 2 + 3(n-i) = 3n + 2 - 3i.$$

Sustituyendo en $T(n)$ obtenemos:

$$T(n) = 2 + \sum_{i=0}^{n-1}(3n + 4 - 3i) = 2 + 3n^2 + 4n - 3\sum_{i=0}^{n-1} i$$

$$= 2 + 3n^2 + 4n - 3\frac{(n-1)n}{2} = \frac{4 + 6n^2 + 8n - 3n^2 + 3n}{2}$$

$$= \frac{3n^2 + 11n + 4}{2} \in \Theta(n^2)$$

Ejemplo con tres bucles – peor caso

En el peor caso, sustituyendo T_{If} en $T_{\text{For 2}}$ tenemos:

$$T_{\text{For 2}} = 2 + \sum_{j=i+1}^{n} (5 + 4j) = 2 + 5(n - i) + 4 \sum_{j=i+1}^{n} j$$

$$= 2 + 5n - 5i + 4 \left[\sum_{j=1}^{n} j - \sum_{j=1}^{i} j \right]$$

$$= 2 + 5n - 5i + 4 \left[\frac{n(n+1)}{2} - \frac{i(i+1)}{2} \right]$$

Simplificando obtenemos:

$$T_{\text{For 2}} = 2 + 5n - 5i + 2n^2 + 2n - 2i^2 - 2i = 2n^2 + 7n + 2 - 7i - 2i^2$$

Ejemplo con tres bucles – peor caso

Sustituyendo en la expresión para el primer bucle tenemos:

$$T(n) = 2+\sum_{i=0}^{n-1}(2n^2+7n+4-7i-2i^2) = 2+2n^3+7n^2+4n-7\sum_{i=0}^{n-1} i-2\sum_{i=0}^{n-1} i^2$$

$$= 2 + 2n^3 + 7n^2 + 4n - 7\frac{(n-1)n}{2} - 2\frac{(n-1)n(2n-1)}{6}$$

$$= 2 + 2n^3 + 7n^2 + 4n - 7\frac{n^2}{2} + 7\frac{n}{2} - 2\frac{2n^3-3n^2+n}{6}$$

$$= \frac{12 + 12n^3 + 42n^2 + 24n - 21n^2 + 21n - 4n^3 + 6n^2 - 2n}{6}$$

Finalmente:

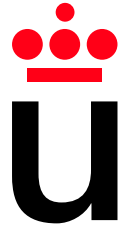
$$T(n) = \frac{8n^3 + 27n^2 + 43n + 12}{6} \in \Theta(n^3)$$

Tema 3.2
Análisis de algoritmos recursivos

Diseño y Análisis de Algoritmos

Manuel Rubio Sánchez

13 de mayo de 2024



Universidad
Rey Juan Carlos

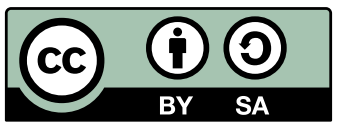
Copyright

©2024 Manuel Rubio Sánchez

Algunos derechos reservados.

Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en:

https://creativecommons.org/licenses/by-sa/4.0/deed.es.



Contenido

Introducción

Análisis de algoritmos recursivos

- La matemática necesaria para analizar algoritmos recursivos son las **relaciones de recurrencia**, también llamadas **ecuaciones en diferencias** o simplemente **recurrencias**
- Las recurrencias son expresiones matemáticas recursivas

$$T(n) = \begin{cases} 3 & \text{si } n = 0 \\ 5 + T(n-1) & \text{si } n > 0 \end{cases}$$

- La resolución de recurrencias consiste en proporcionar fórmulas no recursivas equivalentes

$$T(n) = 5n + 3$$

- Veremos dos formas de resolverlas:
  - Expansión de recurrencias
  - "Método general"

Análisis de eficiencia en tiempo: Expansión de recurrencias

Función potencia - versión 1

```
1 def pot1(b,e):  # e de tipo entero no negativo
2     if e==0:
3         return 1;
4     else:
5         return b*pot1(b,e-1)
```

$$T(n) = \begin{cases} 3 & \text{si } n = 0 \\ 5 + T(n-1) & \text{si } n > 0 \end{cases}$$

- n está relacionado con el tamaño del problema, que en este caso es el exponente e de la función
- Podemos pensar en el caso base se realizan 3 operaciones, y 5 en el recursivo (además del tiempo que llevaría realizar otra llamada con parámetro $n-1$)

Resolución por expansión de recurrencias

$$\begin{aligned} T(n) &= 5 + T(n-1) \\ &= 5 + 5 + T(n-2) = 5 \cdot 2 + T(n-2) \\ &= 5 + 5 + 5 + T(n-3) = 5 \cdot 3 + T(n-3) \\ &\vdots \\ &= 5i + T(n-i) \end{aligned}$$

- ¿Cuándo se llega al caso base $T(0)$?
  - Cuando $i = n$
- Sustituyendo:

$$T(n) = 5n + T(0) = 5n + 3 \quad \in \quad \Theta(n)$$

- Tiene sentido, ya que decrementamos n en cada llamada recursiva

Función potencia - versión 2

```
1 def pot2(b,e):  # e de tipo entero no negativo
2     if e==0:
3         return 1;
4     elif e%2==0:
5         return pot2(b*b,e//2)
6     else:
7         return b*pot2(b*b,e//2)
```

$$T(n) = \begin{cases} 3 & \text{si } n = 0 \\ 8 + T(n/2) & \text{si } n > 0 \text{ y } n \text{ es par} \\ 9 + T((n-1)/2) & \text{si } n > 0 \text{ y } n \text{ es impar} \end{cases}$$

- La función es difícil de analizar
- Pero podemos suponer que $n = 2^x$ es una potencia de dos
  - El resultado del análisis es el mismo

Resolución por expansión de recurrencias

- Asumimos que $n = 2^x$ es una potencia de dos (por tanto, par):

$$\begin{aligned} T(n) &= 8 + T(n/2) \\ &= 8 + 8 + T(n/4) = 8 \cdot 2 + T(n/2^2) \\ &= 8 + 8 + 8 + T(n/8) = 8 \cdot 3 + T(n/2^3) \\ &\vdots \\ &= 8i + T(n/2^i) \end{aligned}$$

- ¿Cuándo se llega al caso base $T(0)$?
  - $i \to \infty$
  - Pero eso no tiene sentido
  - El parámetro de la función T es entero

Resolución por expansión de recurrencias

- ¿Cuándo se llega al caso $T(1)$?
  - Cuando $n/2^i = 1$, es decir, cuando $i = \log_2 n$
- Sustituyendo:

$$T(n) = 8\log_2 n + T(1) = 8\log_2 n + 9 + T(0) = 8\log_2 n + 9 + 3 =$$

$$= 8\log_2 n + 12 \quad \in \quad \Theta(\log n)$$

- Tiene sentido, ya que dividimos n por 2 en cada llamada recursiva

Soluciones generales a relaciones de recurrencia

$$T(n) = \begin{cases} a & \text{si } n = 0 \\ b + T(n-1) & \text{si } n > 0 \end{cases}$$

$$T(n) = bn + a \quad \in \quad \Theta(n)$$

$$T(n) = \begin{cases} a & \text{si } n = 1 \\ b + T(n-1) & \text{si } n > 1 \end{cases}$$

$$T(n) = b(n-1) + a \quad \in \quad \Theta(n)$$

Soluciones generales a relaciones de recurrencia

$$T(n) = \begin{cases} a & \text{si } n = 1 \\ b + T(n/2) & \text{si } n > 1 \end{cases}$$

$$T(n) = b \log_2 n + a \quad \in \quad \Theta(\log n)$$

$$T(n) = \begin{cases} a & \text{si } n = 0 \\ b + T(n/2) & \text{si } n > 0 \end{cases}$$

$$T(n) = b \log_2 n + b + a \quad \in \quad \Theta(\log n)$$

Soluciones generales a relaciones de recurrencia

$$T(n) = \begin{cases} a & \text{si } n = 0 \\ bn + c + T(n-1) & \text{si } n > 0 \end{cases}$$

$$\begin{aligned}
T(n) &= bn + c + T(n-1) \\
&= bn + c + b(n-1) + c + T(n-2) = 2bn - b + 2c + T(n-2) \\
&= 2bn - b + 2c + b(n-2) + c + T(n-3) = \\
&= 3bn - b(1+2) + 3c + T(n-3) = \\
&= 3bn - b(1+2) + 3c + b(n-3) + c + T(n-4) = \\
&= 4bn - b(1+2+3) + 4c + T(n-4) = \\
&\vdots \\
&= ibn - b\sum_{j=1}^{i-1} j + ic + T(n-i) = ibn + ic - b\frac{i(i-1)}{2} + T(n-i)
\end{aligned}$$

Soluciones generales a relaciones de recurrencia

- Se alcanza $T(0)$ para $i = n$
- Sustituyendo:

$$T(n) = bn^2 - \frac{b}{2}n(n-1) + cn + a$$

$$T(n) = \frac{b}{2}n^2 + \left(c + \frac{b}{2}\right)n + a \quad \in \quad \Theta(n^2)$$

- Tiene sentido, ya que hacemos $n + (n-1) + (n-2) + \cdots + 1$ operaciones

Soluciones generales a relaciones de recurrencia

$$T(n) = \begin{cases} a & \text{si } n = 0 \\ bn + c + 2T(n/2) & \text{si } n > 0 \end{cases}$$

$$\begin{aligned}
T(n) &= bn + c + 2T(n/2) \\
&= bn + c + 2\left(b\frac{n}{2} + c + 2T(n/4)\right) \\
&= bn + c + 2\left[b\frac{n}{2} + c + 2\left(b\frac{n}{4} + c + 2T(n/8)\right)\right] \\
&= 3bn + c(1+2+4) + 2^3 T(n/2^3) = 3bn + c(2^3-1) + 2^3 T(n/2^3) \\
&\vdots \\
&= ibn + c(2^i - 1) + 2^i T(n/2^i)
\end{aligned}$$

Soluciones generales a relaciones de recurrencia

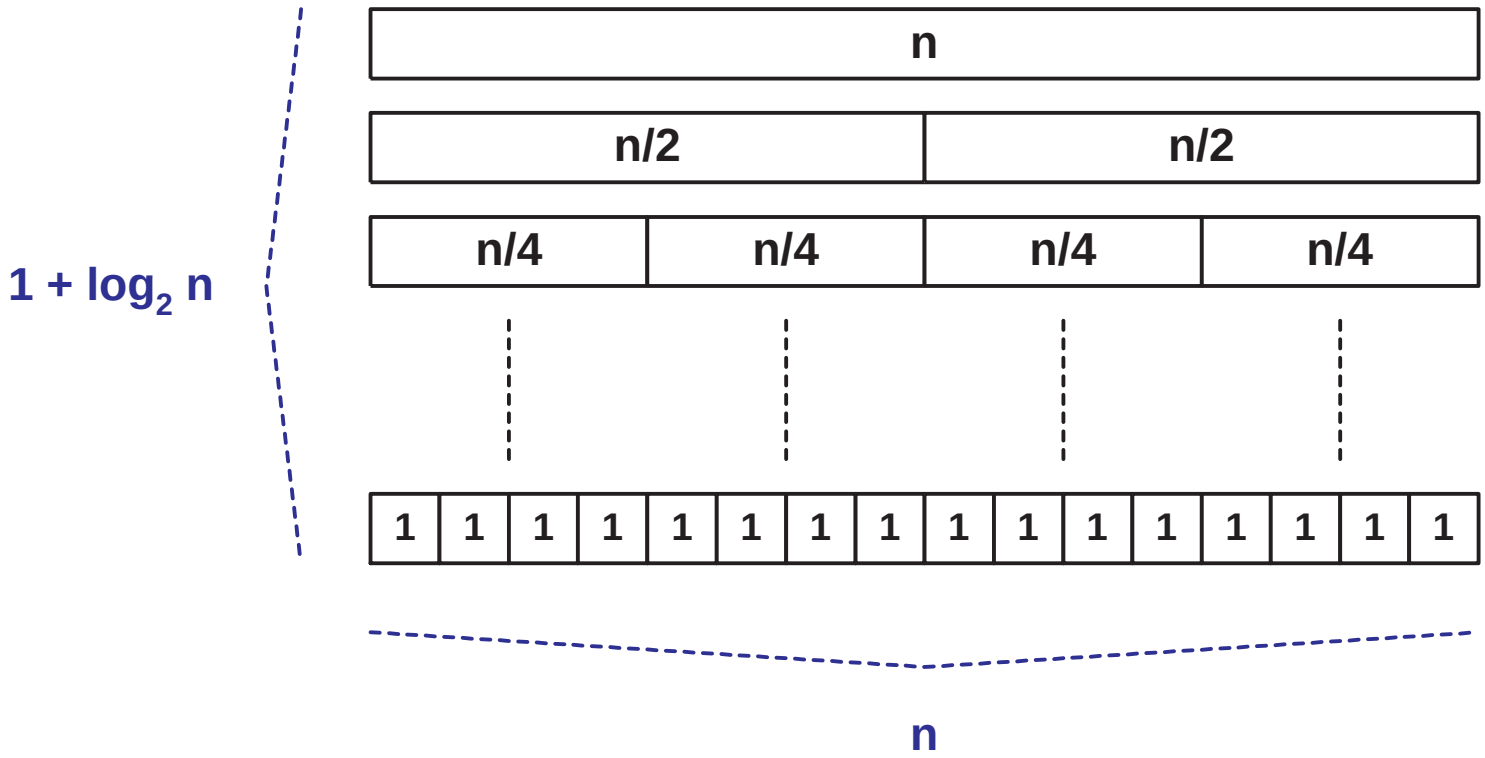
- Se alcanza $T(1)$ para $n/2^i = 1$, es decir, cuando $i = \log_2 n$
- Sustituyendo:

$$\begin{aligned} T(n) &= bn\log_2 n + c(n-1) + nT(1) \\ &= bn\log_2 n + c(n-1) + n(b + c + 2T(0)) \\ &= bn\log_2 n + cn - c + nb + nc + 2na \end{aligned}$$

$$T(n) = bn\log_2 n + (2c + b + 2a)n - c \quad \in \quad \Theta(n\log n)$$

Soluciones generales a relaciones de recurrencia

- Tiene sentido, ya que hacemos n operaciones $1 + \log_2 n$ veces



Teorema maestro - versión simple

- Fórmula útil para algoritmos "divide y vencerás":

$$T(n) = \begin{cases} d & \text{si } n = 1 \\ aT(n/b) + cn^k & \text{si } n > 1 \end{cases}$$

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } \frac{a}{b^k} < 1 \\ \Theta(n^k \log n) & \text{si } \frac{a}{b^k} = 1 \\ \Theta(n^{\log_b a}) & \text{si } \frac{a}{b^k} > 1 \end{cases}$$

Teorema maestro - demostración

$$
\begin{aligned}
T(n) &= aT(n/b) + cn^k \\
&= a\left[aT\left(\frac{n}{b^2}\right) + c\left(\frac{n}{b}\right)^k\right] + cn^k = a^2 T\left(\frac{n}{b^2}\right) + cn^k\left(1 + \frac{a}{b^k}\right) \\
&= a^2\left[aT\left(\frac{n}{b^3}\right) + c\left(\frac{n}{b^2}\right)^k\right] + cn^k\left(\frac{n}{b^2}\right) = a^3 T\left(\frac{n}{b^3}\right) + cn^k\left(1 + \frac{a}{b^k} + \frac{a^2}{b^{2k}}\right) \\
&\vdots \\
&= a^i T\left(\frac{n}{b^i}\right) + cn^k \sum_{j=0}^{i-1}\left(\frac{a}{b^k}\right)^j \qquad \text{Se alcanza } T(1) = c \text{ cuando: } i = \log_b n \\
&= ca^{\log_b n} + cn^k \sum_{j=0}^{\log_b n - 1}\left(\frac{a}{b^k}\right)^j = cn^k\left(\frac{a}{b^k}\right)^{\log_b n} + cn^k \sum_{j=0}^{\log_b n - 1}\left(\frac{a}{b^k}\right)^j
\end{aligned}
$$

Teorema maestro - demostración (logaritmos)

- La última igualdad se debe a:

$$cn^k \left(\frac{a}{b^k}\right)^{\log_b n} = cn^k \frac{a^{\log_b n}}{b^{k \log_b n}} = cn^k \frac{a^{\log_b n}}{n^k} = ca^{\log_b n}$$

- Más adelante también necesitaremos:

$$n^{log_b a} = a^{log_b n}$$

  Ya que:

$$\log_b n^{log_b a} = \log_b a^{log_b n} \quad = \quad \log_b a \cdot \log_b n$$

- Finalmente:

$$T(n) = cn^k \sum_{j=0}^{\log_b n} \left(\frac{a}{b^k}\right)^j$$

Y tendremos 3 casos según los valores de a, b y k

Teorema maestro - demostración

$$T(n) = cn^k \sum_{j=0}^{\log_b n} \left(\frac{a}{b^k}\right)^j$$

1. $a < b^k \Rightarrow T(n) \in \Theta(n^k)$

$$\sum_{i=0}^{\infty} r^i = \frac{1}{1-r} = \text{constante (no diverge), para } r < 1$$

2. $a = b^k \Rightarrow T(n) \in \Theta(n^k \log n)$

$$T(n) = cn^k(\log_b n + 1)$$

Teorema maestro - demostración

$$T(n) = cn^k \sum_{j=0}^{\log_b n} \left(\frac{a}{b^k}\right)^j$$

3. $a > b^k \Rightarrow T(n) \in \Theta(n^{\log_b a})$

$$T(n) = cn^k \frac{\left(\frac{a}{b^k}\right)^{\log_b n+1} - 1}{\left(\frac{a}{b^k}\right) - 1} = \frac{cn^k \left(\frac{a}{b^k}\right) \frac{a^{\log_b n}}{n^k} - cn^k}{K_1}$$

$$= \frac{K_2 a^{\log_b n} - cn^k}{K_1} = \frac{K_2 n^{\log_b a} - cn^k}{K_1}$$

Como $a > b^k \Rightarrow \log_b a > k \Rightarrow T(n) \in \Theta(n^{\log_b a})$

Teorema maestro - versión completa

- Dada una recurrencia del tipo:

$$T(n) = aT(n/b) + f(n)$$

donde $a > 0$, $b > 0$, y $f(n)$ es una función asintóticamente positiva, entonces se puede aplicar el **teorema maestro** en estos tres casos:

Teorema maestro - versión completa

1. Si $f(n) = \mathcal{O}(n^{log_b a - \epsilon})$ para alguna constante $\epsilon > 0$, entonces:

$$T(n) \in \Theta(n^{log_b a})$$

2. Si $f(n) = \Theta(n^{log_b a} log^k n)$ con[1] $k \geq 0$, entonces:

$$T(n) \in \Theta(n^{log_b a} log^{k+1} n)$$

3. Si $f(n) = \Omega(n^{log_b a + \epsilon})$ con $\epsilon > 0$, y $f(n)$ satisface la condición de regularidad ($af(n/b) \leq cf(n)$ para alguna constante $c < 1$ y para todo n lo suficientemente grande), entonces:

$$T(n) \in \Theta(f(n))$$

[1] k suele ser 0

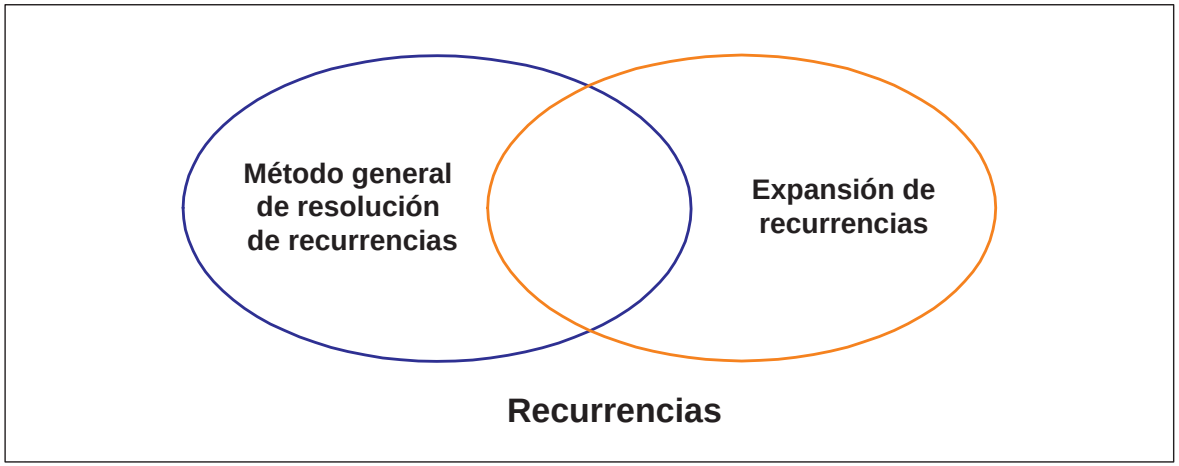
Errores frecuentes – Expansión de recurrencias

- No comprobar que la fórmula final sea análoga a la recurrencia inicial
  - También al aplicar el método general
- No realizar correctamente la primera expansión
  - ¡Muy frecuente!
- No simplificar
  - Los patrones serán mucho más complicados

Análisis de eficiencia en tiempo: Método general

Método general

- No siempre se puede aplicar la expansión de recurrencias
- Para muchas recurrencias no se conoce la forma de resolverlas
  - Sucede como con las integrales o ecuaciones diferenciales: sabemos como resolver un subconjunto de éstas, pero no todas
- Ahora veremos un método general con el que vamos a ampliar el conjunto de recurrencias que podemos resolver



Recurrencias homogéneas

- Dada la siguiente recurrencia homogénea (aparece un 0 en la parte derecha):

$$a_0 T(n) + a_1 T(n-1) + \cdots + a_k T(n-k) = 0$$

- Buscamos soluciones del tipo:

$$T(n) = C_1 P_1(n) r_1^n + \cdots + C_k P_k(n) r_k^n = \sum_{i=1}^{k} C_i P_i(n) r_i^n$$

- Realizando el cambio $x^{z-n+k} = T(z)$ obtenemos la ecuación característica asociada:

$$a_0 x^k + a_1 x^{k-1} + \cdots + a_{k-1} x + a_k = 0$$

Primer caso: raíces distintas

- Si todas las raíces del polinomio de la ecuación característica son distintas:

$$T(n) = C_1 r_1^n + \cdots + C_k r_k^n = \sum_{i=1}^{k} C_i r_i^n$$

  - Las constantes r_i van a ser las raíces de la ecuación característica
  - $P_i(n) = 1$, para todo i
  - Las constantes C_i se hallan a partir de las condiciones iniciales, resolviendo un sistema de ecuaciones

Ejemplo: Números de Fibonacci

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ T(n-1) + T(n-2) & \text{si } n > 1 \end{cases}$$

- La ecuación característica es $x^2 - x - 1 = 0$, cuyas raíces son:

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

- Por tanto, al ser distintas, la solución tiene la forma:

$$T(n) = C_1 r_1^n + C_2 r_2^n = C_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + C_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$$

Ejemplo: Números de Fibonacci

- El siguiente paso consiste en hallar las constantes, a partir de las condiciones iniciales (casos base de la recurrencia):

$$\left.\begin{array}{rcccl} C_1 + C_2 & = & 0 & = & T(0) \\ C_1\left(\frac{1+\sqrt{5}}{2}\right) + C_2\left(\frac{1-\sqrt{5}}{2}\right) & = & 1 & = & T(1) \end{array}\right\}$$

- Resolviendo el sistema obtenemos:

$$C_1 = \frac{1}{\sqrt{5}} \qquad C_2 = -\frac{1}{\sqrt{5}}$$

- Finalmente:

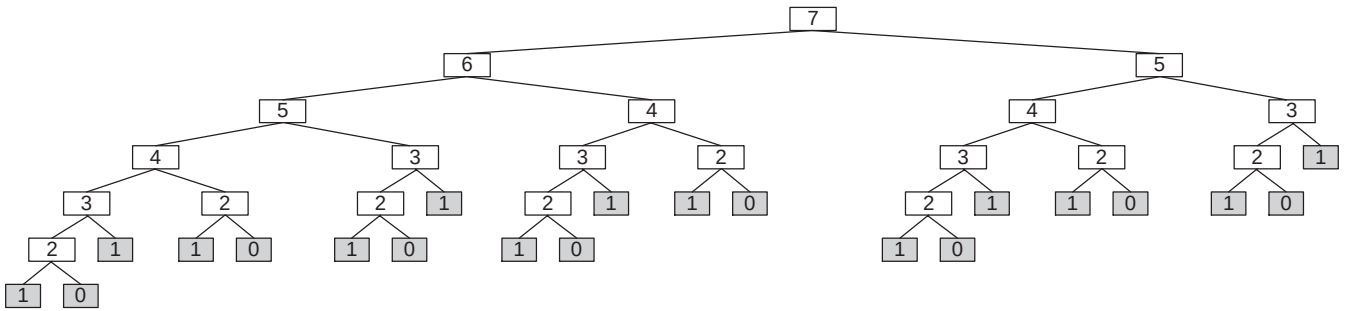
$$T(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Ejemplo: Números de Fibonacci

- El segundo término tiende a 0 según $n \to \infty$, por tanto:

$$T(n) \quad \in \quad \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

- El orden es exponencial
- El árbol de recursión es binario, pero está podado
- Para un árbol de recursión binario completo el orden es 2^n
- En este caso la base del exponente es $(1+\sqrt{5})/2 \approx 1{,}618 < 2$



Segundo caso: raíces con multiplicidad mayor que 1

- En general, el polinomio asociado a la ecuación característica puede tener raíces con multiplicidad 1 o mayor que 1

$$(x - r_1)^{m_1} \cdot (x - r_2)^{m_2} \cdots (x - r_k)^{m_k} = 0$$

- En este caso general, la solución tiene la forma:

$$T(n) = \sum_{i=1}^{m_1} C_{1i} n^{i-1} r_1^n + \sum_{i=1}^{m_2} C_{2i} n^{i-1} r_2^n + \cdots + \sum_{i=1}^{m_k} C_{ki} n^{i-1} r_k^n$$

Segundo caso: raíces con multiplicidad mayor que 1

- Ejemplo:

$$T(n) = 5T(n-1) - 8T(n-2) + 4T(n-3)$$

- Ecuación característica:

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

- El 2 es una raíz doble, por tanto:

$$T(n) = C_1 2^n + C_2 n 2^n + C_3 1^n$$

Recurrencias no homogéneas - una primera idea

- La parte de la derecha ya no es 0

$$T(n) - 2T(n-1) = 3^n \qquad (a)$$

- La convertimos en homogénea:

$$\begin{array}{rcll} T(n+1) - 2T(n) & = & 3^{n+1} & (1) \\ 3T(n) - 6T(n-1) & = & 3^{n+1} & (2) \\ \hline T(n+1) - 5T(n) + 6T(n-1) & = & 0 & (1)-(2) \end{array}$$

  - En (1) se incrementa n en (a), en (2) se multiplica (a) por 3

- Con $T(0) = 0$ y $T(1) = 3$

$$T(n) = 3 \cdot 3^n - 3 \cdot 2^n \quad \in \quad \Theta(3^n)$$

Recurrencias no homogéneas - caso simple

- Recurrencia, con **un solo** término a la derecha donde d es el orden del polinomio $P(n)$

$$a_0\,T(n) + a_1\,T(n-1) + \cdots + a_k\,T(n-k) = b^n P^d(n)$$

- Ecuación característica:

$$(a_0x^k + a_1x^{k-1} + \cdots + a_k)(x-b)^{d+1} = 0$$

- Ejemplo:

$$T(n) - 2\,T(n-1) = n \qquad b = 1 \quad P(n) = n \quad d = 1$$

$$(x-2)(x-1)^2 = 0 \quad \Longrightarrow \quad T(n) = C_1 2^n + C_2 1^n + C_3 n 1^n \in \Theta(2^n)$$

Recurrencias no homogéneas - caso general

- Recurrencia general:

$$a_0 T(n)+a_1 T(n-1)+\cdots+a_k T(n-k) = b_1^n P_1^{d_1}(n)+\cdots+b_s^n P_s^{d_s}(n)$$

- Ecuación característica:

$$(a_0 x^k + a_1 x^{k-1} + \cdots + a_k)(x - b_1)^{d_1+1} \cdots (x - b_s)^{d_s+1} = 0$$

Recurrencias no homogéneas - caso general

- Ejemplo:

$$T(n) = 2T(n-1) + n + 2^n \qquad \text{con } T(0) = 1$$

- Ecuación característica:

$$(x-2)(x-1)^2(x-2) = (x-2)^2(x-1)^2 = 0$$

- Solución (sin hallar las constantes):

$$T(n) = C_1 2^n + C_2 n 2^n + C_3 1^n + C_4 n 1^n$$

Recurrencias no homogéneas - caso general

- Dado $T(0) = 1$, y usando $T(n) = 2T(n-1) + n + 2^n$ tenemos que hallar $T(1)$, $T(2)$ y $T(3)$, para formar un sistema de 4 ecuaciones y 4 incógnitas (las constantes):

$$T(1) = 5 \qquad T(2) = 16 \qquad T(3) = 43$$

$$\left.\begin{array}{rcccl} C_1 + C_3 & = & 1 & = & T(0) \\ 2C_1 + 2C_2 + C_3 + C_4 & = & 5 & = & T(1) \\ 4C_1 + 8C_2 + C_3 + 2C_4 & = & 16 & = & T(2) \\ 8C_1 + 24C_2 + C_3 + 3C_4 & = & 43 & = & T(3) \end{array}\right\}$$

- Solución final ($C_1 = 3$, $C_2 = 1$, $C_3 = -2$ y $C_4 = -1$):

$$T(n) = 3 \cdot 2^n + n2^n - 2 - n \quad \in \quad \Theta(n2^n)$$

Método de cambio de variable

Recurrencia: $T(n) = 4T(n/2) + n \qquad T(1) = 1 \qquad T(2) = 6$

Cambio: $n = 2^k \quad \Rightarrow \quad T(2^k) = 4T(2^{k-1}) + 2^k$

$T(n) = T(2^k) = t(k) \quad \Rightarrow \quad \boxed{t(k) = 4t(k-1) + 2^k}$ Nueva recurrencia

Ecuación característica: $(x-4)(x-2) = 0$

$t(k) = C_1(4^k) + C_2(2^k) = C_1(2^k)^2 + C_2(2^k)$

Deshaciendo el cambio: $T(n) = C_1 n^2 + C_2 n$

Hallando las constantes: $T(n) = 2n^2 - n \quad \in \quad \Theta(n^2)$

Expansión de recurrencias (mismo ejemplo)

$$
\begin{aligned}
T(n) &= T(n) = 4T(n/2) + n \\
&= 4\left[4T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n = 4^2 T\left(\frac{n}{2^2}\right) + 2n + n \\
&= 4\left[4\left[4T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + \frac{n}{2}\right] + n = 4^3 T\left(\frac{n}{2^3}\right) + 4n + 2n + n \\
&= 4^i T\left(\frac{n}{2^i}\right) + n\sum_{j=0}^{i-1} 2^j = 4^i T\left(\frac{n}{2^i}\right) + n(2^i - 1)
\end{aligned}
$$

El caso base $T(1)$ se alcanza cuando $n = 2^i$, $n^2 = (2^i)^2 = (2^2)^i = 4^i$.
Sustituyendo:

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

Errores frecuentes – método general

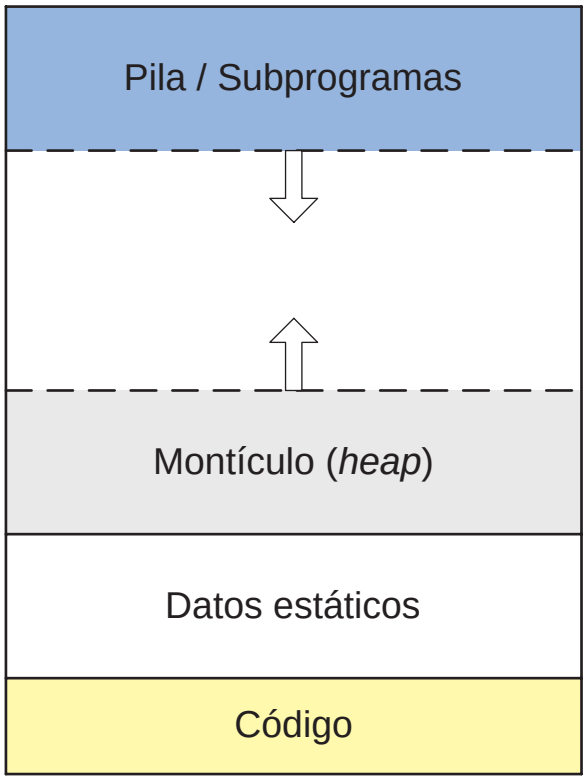
- Cambio de variable $n = 2^k$, y ecuación $T(2^k) = 4T(2^{k-1}) + 2^k$
  - Error: al cambiar $t(k) = T(2^k)$ hacer $t(k) = 4t(k-1) + k$
  - Lo correcto es: $t(k) = 4t(k-1) + 2^k$
- Si $t(k) - t(k-1) = 3^{2k}$ no tenemos una potencia del tipo b^k en el miembro derecho
  - Hay que transformarla: $3^{2k} = (3^2)^k = 9^k$
- A la hora de hallar el polinomio característico es necesario sacar factor común de las potencias en el miembro derecho
  - $t(k) - t(k-1) = 2 + k^3 + 2^k + k2^k$ es $t(k) - t(k-1) = (2 + k^3)1^k + (1 + k)2^k$
  - El polinomio característico sería $(x-1)(x-1)^4(x-2)^2 = (x-1)^5(x-2)^2$

Análisis de eficiencia en espacio

Modelo de memoria

- Código del algoritmo
- Datos estáticos
  - Variables globales
- Llamadas a subprogramas
  - Parámetros
  - Variables locales
- Memoria dinámica

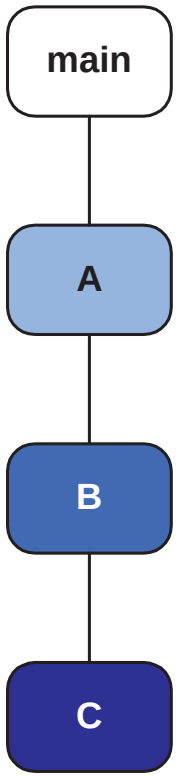
- Modelo (simple) de memoria



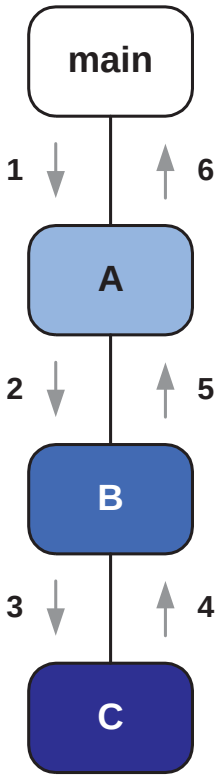
Árbol y proceso de llamadas a funciones

```
1  ...
2  A(..)
3  ...
4
5  def A(...)
6      ...
7      B(..)
8      ...
9
10 def B(...)
11     ...
12     C(..)
13     ...
14
15 def C(...)
16     ...
```

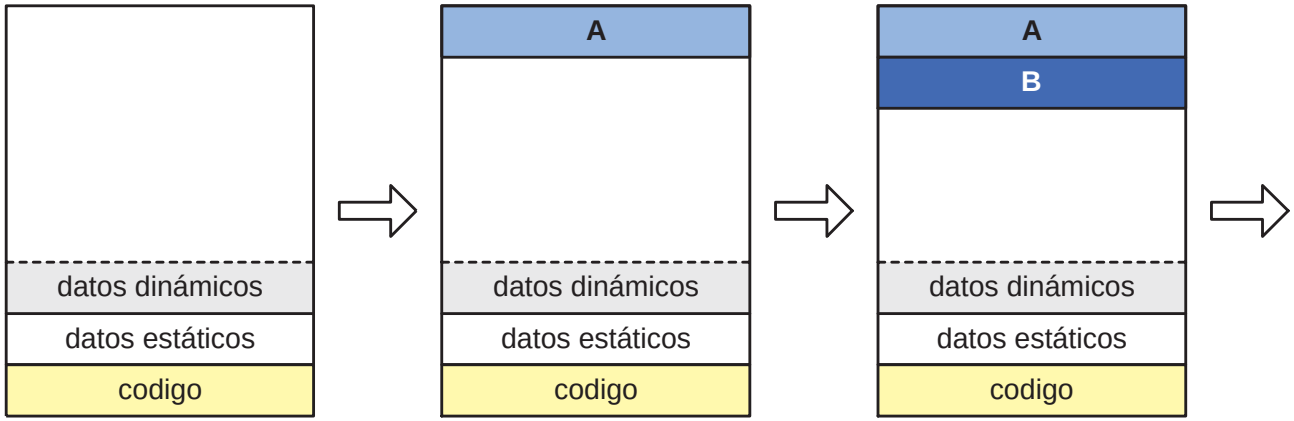
- Árbol de llamadas

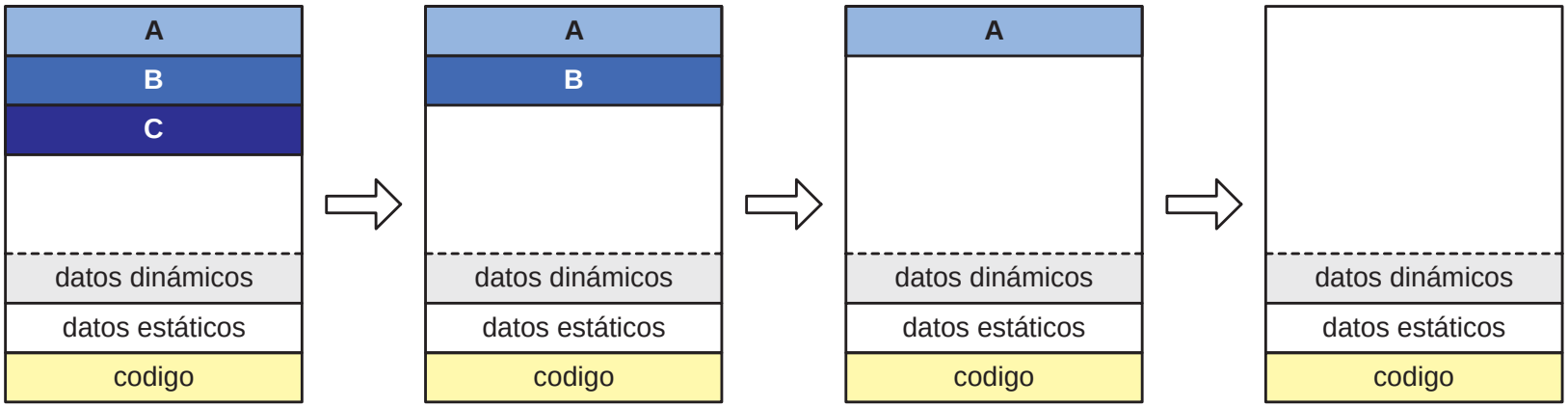


- Proceso de llamadas



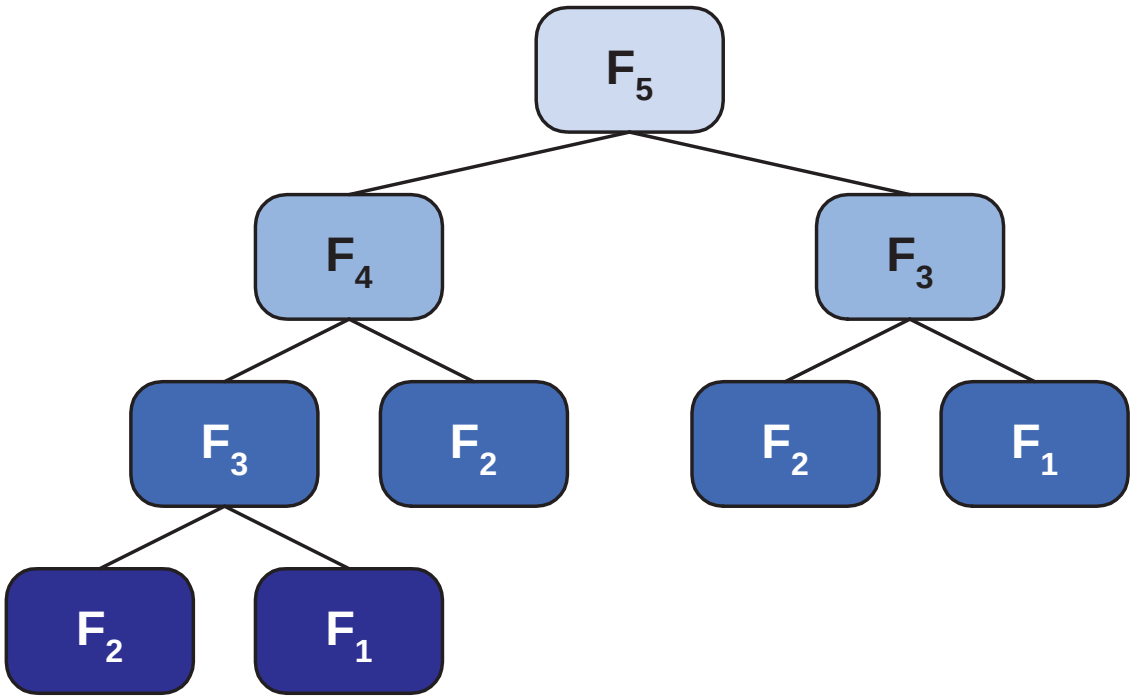
Estado de la pila



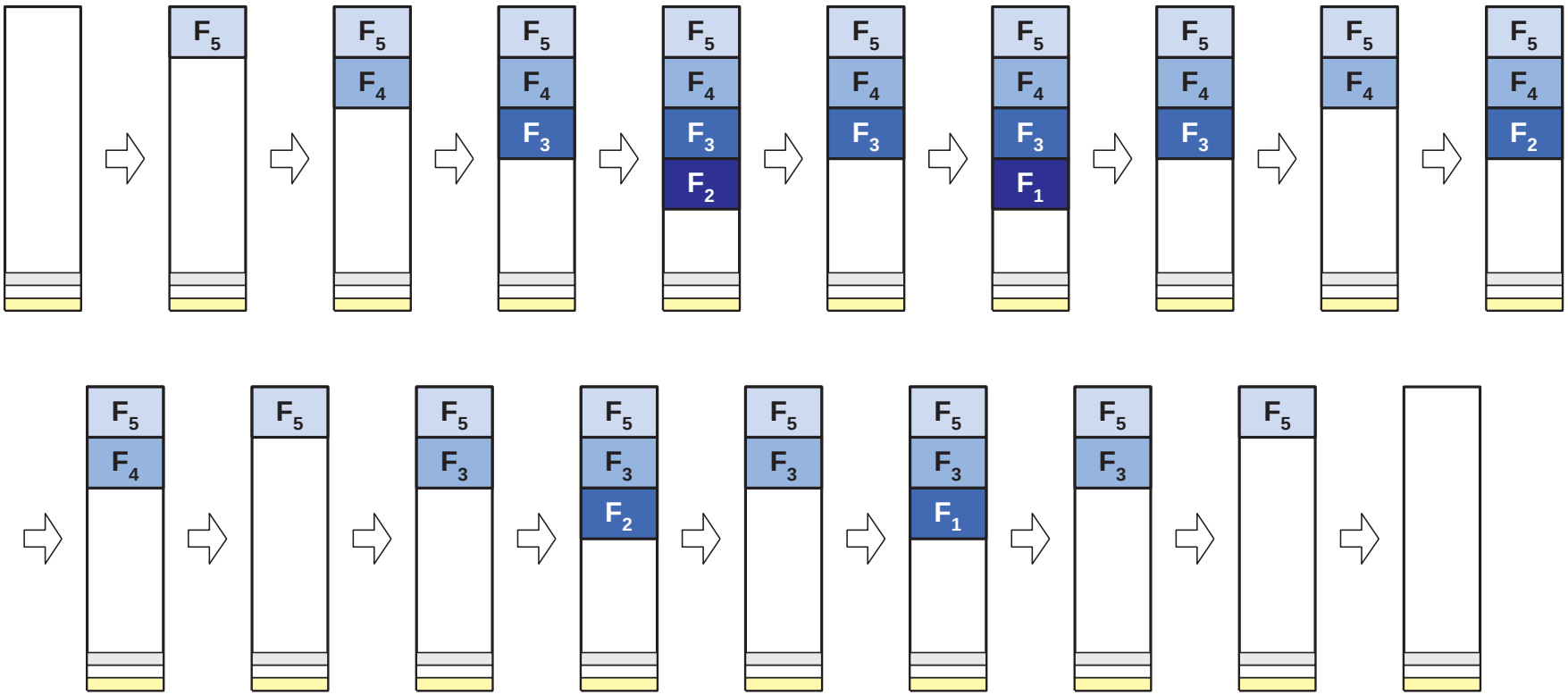


Árbol recursivo

- Caso recursivo: $F_{n+2} = F_{n+1} + F_n$
- Casos base $F_1 = F_2 = 1$



Estado de la pila



$$\text{Espacio}(n) \in \Theta(n)$$

Llamadas a subprogramas

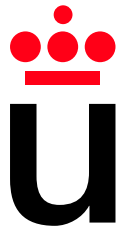
- La complejidad depende de la **profundidad del árbol recursivo** de llamadas
  - Para ilustrar esto se han coloreado los nodos de un mismo nivel en el árbol con el mismo color
- La complejidad también depende de los recursos que requiera cada subprograma (función/método/subrutina) ejecutado
  - Parámetros de los subprogramas
  - Variables locales a los subprogramas
  - Memoria dinámica reservada al ejecutar el subprograma

Tema 4
Introducción a la recursividad

Diseño y Análisis de Algoritmos

Manuel Rubio Sánchez

13 de mayo de 2024



Universidad
Rey Juan Carlos

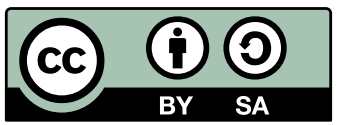
Copyright

©2024 Manuel Rubio Sánchez

Algunos derechos reservados.

Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en:

https://creativecommons.org/licenses/by-sa/4.0/deed.es.

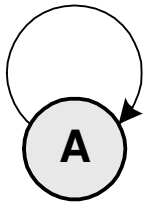


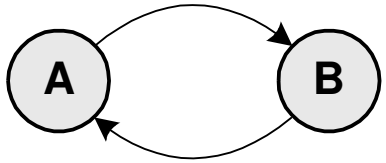
Contenido

Introducción

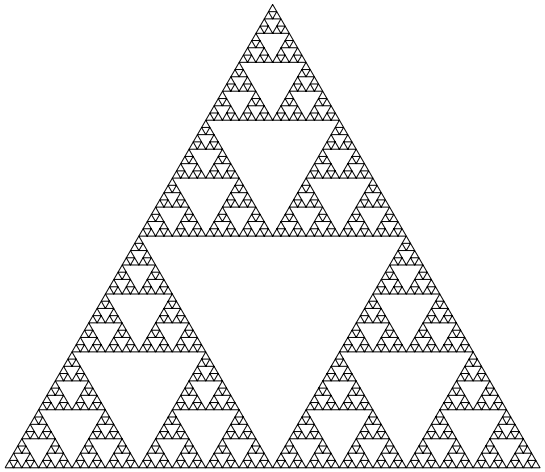
Recursividad

- Métodos/funciones/subprogramas que se invocan a sí mismos
  - Si **A** llama a **B**, y **B** llama a **A**: **A** y **B** también son recursivos





- La definición de conceptos en términos de sí mismos



Recursividad

- **Herramienta muy potente** para la resolución de problemas computacionales y matemáticos
  - Una forma de pensar al abordar problemas computacionales
  - Enfoque de diseño de algoritmos
- Alternativa a los bucles
  - Para ejecutar instrucciones de manera repetida
- Alternativa a algunas estructuras de datos
  - Pilas y colas

¿Se puede usar en un diccionario?

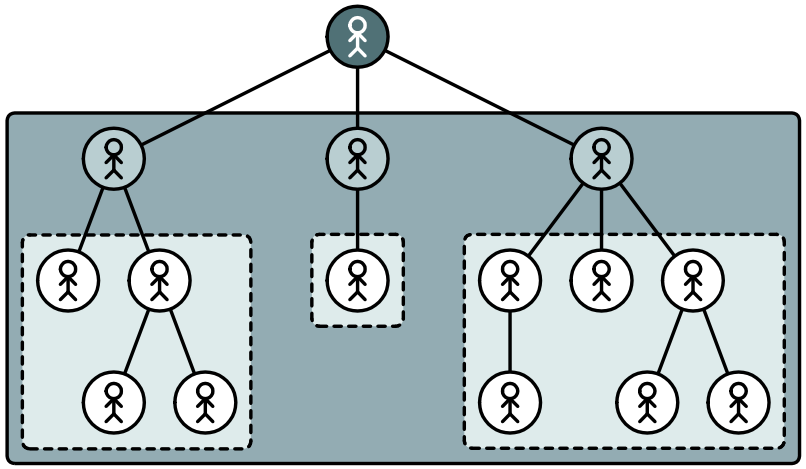
- Definición de **descendientes**, según el diccionario de la Real Academia Española:
  - *Descendientes*: "Hijos, nietos o personas que **descienden** de otra"
  - *Descender*: "**Proceder**, por natural propagación, de un mismo principio o persona común, que es la cabeza de la familia"
  - *Proceder*: "Dicho de una persona o de una cosa: Tener su origen en un determinado lugar, o **descender** de cierta persona, familia o cosa"

¿Se puede usar en un diccionario?

- Mejor definición: los hijos más los descendientes de los hijos

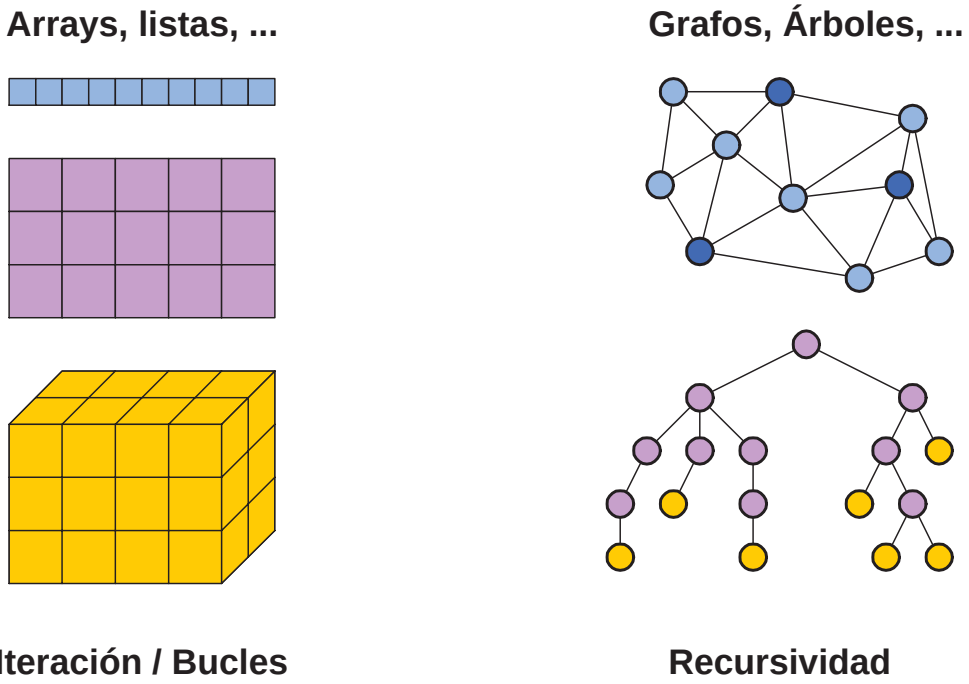
$$D(p) = \begin{cases} \varnothing & \text{si } H(p) = \varnothing \\ H(p) \cup D(H(p)) & \text{si } H(p) \neq \varnothing \end{cases}$$

H: hijos, D: descendientes, p: persona



¿Cuándo es útil aplicar la recursividad?

- Especialmente útil cuando la "estructura" del problema, algoritmo o los datos no es "lineal"



- Cuando el método realice dos o más llamadas a sí mismo
- Cuando sea más fácil comprender la versión recursiva

Conceptos clave en la recursividad

- **Descomposición/simplificación** de problemas
  - Debemos ser capaces de reconocer que para resolver un problema podemos valernos de **soluciones a "subproblemas" idénticos al original, pero más sencillos o de menor tamaño**

- **Inducción**
  - Construimos nuestra solución **suponiendo que ya sabemos la solución a estos problemas más simples**
    - No tenemos que calcular estas "subsoluciones" desde cero
    - Se obtienen a través de llamadas recursivas

Primer ejemplo: Factorial

$$n! = \prod_{i=1}^{n} i = \overbrace{\underbrace{1 \times 2 \times 3 \times \cdots \times (n-1)}_{\text{problema: } n!} \times n}^{\text{subproblema: } (n-1)!} = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{si } n > 0 \end{cases}$$

```
1 def factorial(n):
2     if n==0:
3         return 1
4     else:
5         return factorial(n-1) * n
```

El paradigma de programación declarativa

- En general, hay que pensar en **qué** se va a hacer mucho más que en **cómo** se va a hacer
- A diferencia del paradigma imperativo, evitaremos pensar en cómo se modifican los parámetros y variables a medida que se ejecuta un programa paso a paso
- Suponemos que sabemos **qué** se resuelve (el subproblema), pero no nos interesa saber **cómo**
- Salto de fe recursivo
  - Usas la función que estás programando, **asumiendo que funciona**, aunque todavía no la hayas terminado de implementar

Factorial: implementaciones "iterativas" (a evitar)

```
1 def fact1(n):
2     p = 1
3     while n>0:
4         p = p*n
5         n = n-1
6     return p
7
8 fact1(n)
```

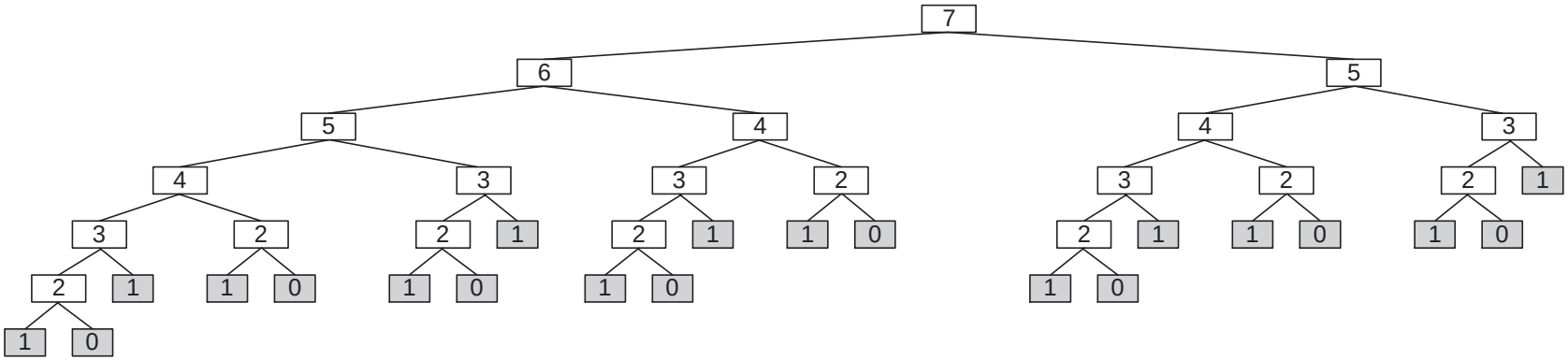
```
1 def fact2(n,p):
2     if n==0:
3         return p
4     else:
5         return fact2(n-1,p*n)
6
7
8 fact2(n,1)
```

iteración	n	p
0	4	1
1	3	4
2	2	12
3	1	24
4	0	24

- Aunque `fact2` sea recursivo (por cola), es equivalente a la versión iterativa `fact1`
- `fact2` surge de un enfoque *imperativo*, donde pensamos en los valores que toman variables y parámetros
- `fact2` no surge de pensar en subproblemas
- `fact2` no surge de pensar de manera "recursiva"

¿Árbol de recursión?

- Si podemos, evitaremos pensar en el *árbol de recursión*



  - Generalmente no ayuda a diseñar el algoritmo
  - Lo importante es definir la **regla recursiva**
    - La relación entre un nodo padre y los hijos, que se aplica en todo el árbol
    - En este caso es $F(n) = F(n-1) + F(n-2)$ (números de Fibonacci o similares)
  - No perdemos tiempo en pensar **cómo** se resolverán los subproblemas

Tipos de recursividad

- Lineal (no final)

$$f(n, A) = \begin{cases} I & \text{si } n = 1 \\ A \cdot f(n-1, A) & \text{si } n > 1 \end{cases} \qquad g(n) = f\left(n, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\right)_{1,2}$$

- Lineal final (por cola)

$$f(n, a, b) = \begin{cases} a & \text{si } n = 0 \\ f(n-1, a+b, a) & \text{si } n \geqslant 1 \end{cases} \qquad g(n) = f(n, 0, 1)$$

- Múltiple

$$f(n) = \begin{cases} 1 & \text{si } n = 1, 2 \\ 1 + \sum_{i=1}^{n-2} f(i) & \text{si } n \geqslant 3 \end{cases} \qquad g(n) = f(n)$$

- Mutua

$$B(i) = \begin{cases} 1 & \text{si } i = 1 \\ A(i-1) & \text{si } i \geqslant 2 \end{cases} \quad A(i) = \begin{cases} 0 & \text{si } i = 1 \\ A(i-1) + B(i-1) & \text{si } i \geqslant 2 \end{cases} \quad g(n) = B(n) + A(n)$$

- Anidada

$$f(n, y) = \begin{cases} 1 + y & \text{si } n = 1, 2 \\ f(n-1, y + f(n-2, 0)) & \text{si } n \geqslant 3 \end{cases} \qquad g(n) = f(n, 0)$$

Metodología para diseñar algoritmos recursivos

Metodología

1. Identificar el **tamaño del problema**
   - Lo que determina el número de operaciones a realizar
2. Establecer los **casos base**
   - Instancias sencillas del problema que no requieren llamadas recursivas
   - Suelen ser instancias de menor tamaño
3. Descomposición: escoger **instancias del problema de menor tamaño**
4. Inducción: establecer los **casos recursivos** a partir de las soluciones de las instancias seleccionadas en la fase de descomposición
5. Implementar y probar

Ejemplo: suma de los primeros naturales

$$S(n)^* = \sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-1) + n$$

- Paso 1 (tamaño del problema)
  - n
- Paso 2 (casos base)
  - $n = 0$. En ese caso: $S(0) = 0$
  - $n = 1$. En ese caso: $S(1) = 1$
    - Redundante si consideramos $S(0) = 0$

*Suponemos que no conocemos la fórmula $S(n) = n(n+1)/2$, que se calcula en tiempo constante $\Theta(1)$

Ejemplo: suma de los primeros naturales

- Paso 3 (descomposición)
  - Consideramos problemas de tamaño menor: $n-1$, $n-2$, $n/2$, $n/10$
  - Algunas descomposiciones no conducen a algoritmos sencillos o eficientes
  - Lo más sencillo en este caso es considerar $S(n-1)$

- Paso 4 (casos recursivos)

$$S(n) = \sum_{i=1}^{n} i = \overbrace{\underbrace{1+2+3+\ \cdots\ +(n-1)}_{\text{problema: } S(n)}+n}^{\text{subproblema: } S(n-1)} = \begin{cases} 0 & \text{si } n=0 \\ n+S(n-1) & \text{si } n>0 \end{cases}$$

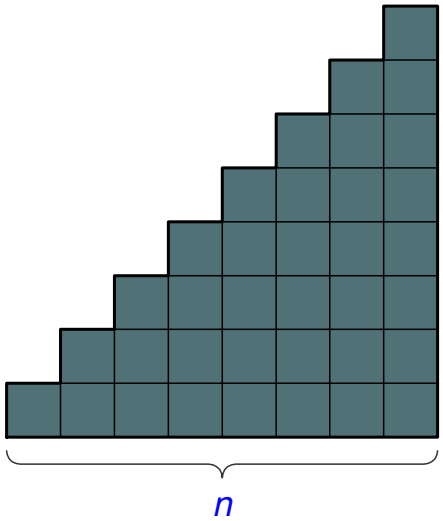
```
1 def suma(n):
2     if n==0:
3         return 0
4     else:
5         return suma(n-1) + n
```

$$T(n) = \begin{cases} 1 & \text{si } n=0 \\ T(n-1)+1 & \text{si } n>0 \end{cases}$$

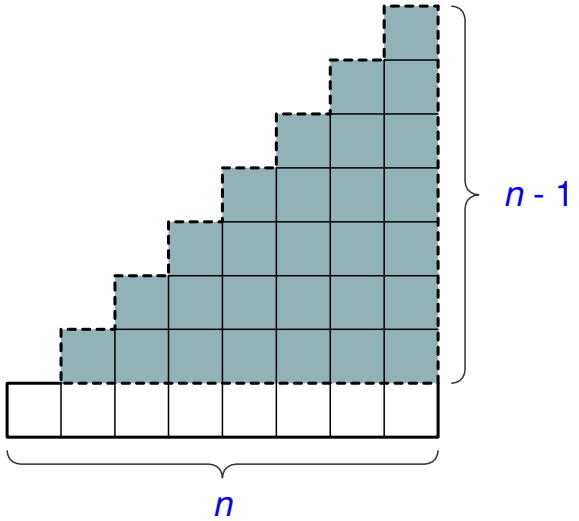
$$T(n) \in \Theta(n)$$

Gráficamente

- Paso 3 (descomposición)
  - Buena idea: usar diagramas para identificar el problema original y el o los subproblemas a considerar



Problema original

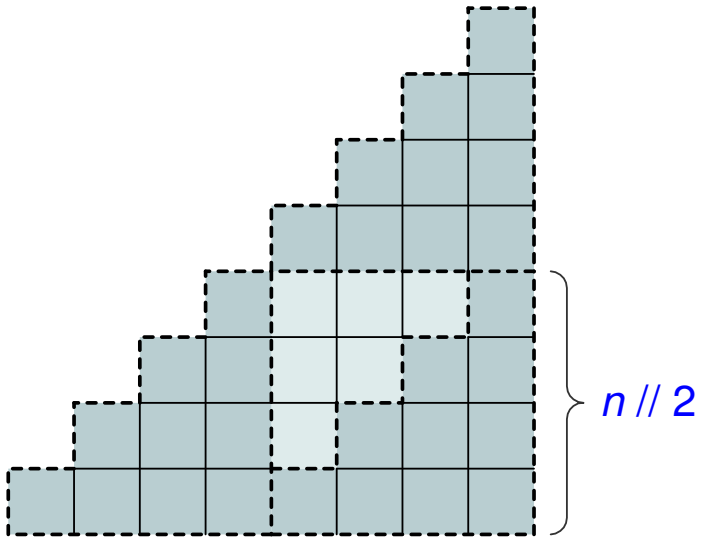


Subproblema

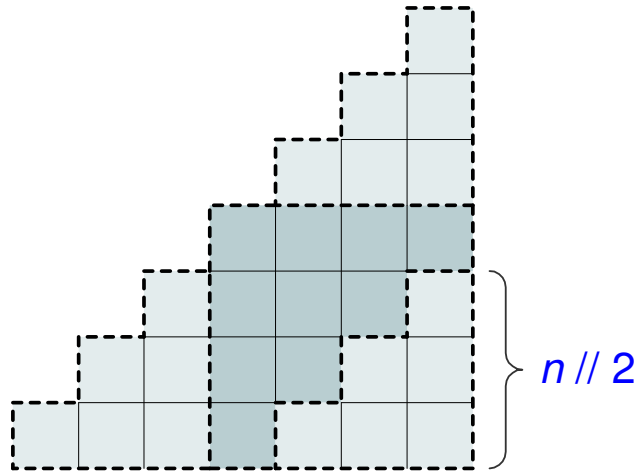
  - Problema $\equiv$ "número de cuadrados en una pirámide triangular de altura (y base) n"

Dividir el tamaño del problema por dos – I

- Paso 3 (descomposición)
  - Problemas de tamaño $n/2$ (aproximadamente)
  - $//$ indica división entera ($8//2 = 4$, $7//2 = 3$)
    - $x//y = \lfloor x/y \rfloor$



n par



n impar

Dividir el tamaño del problema por dos – I

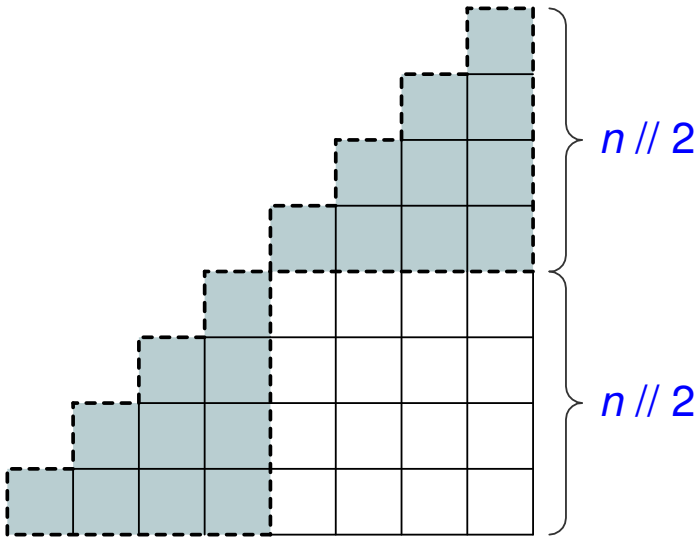
$$S(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ 3S(n/2) + S(n/2-1) & \text{si } n > 1 \text{ y } n \text{ par} \\ 3S((n-1)/2) + S((n+1)/2) & \text{si } n > 1 \text{ y } n \text{ impar} \end{cases}$$

```
1 def suma(n):
2     if n==0:
3         return 0
4     elif n==1:
5         return 1
6     elif n%2==0:
7         return 3*suma(n//2) + suma(n//2 - 1)
8     else:
9         return 3*suma(n//2) + suma(n//2 + 1)
```

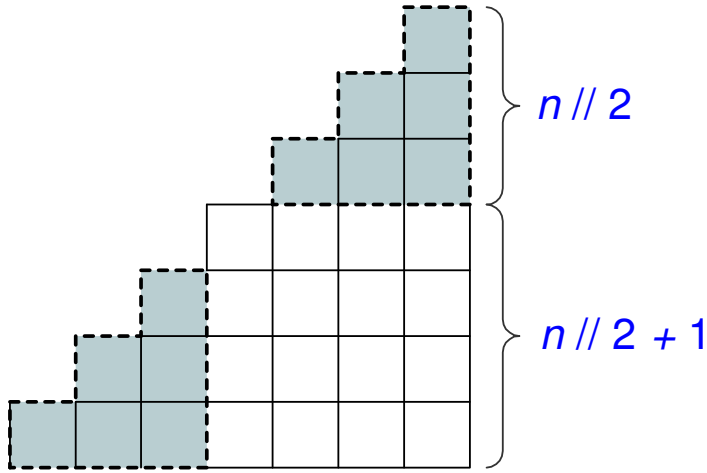
$T(n) = 2T(n/2) + 1$, con $T(0) = 1$ y $T(1) = 1 \quad \Rightarrow \quad T(n) \in \Theta(n)$

Dividir el tamaño del problema por dos – II

- Paso 3 (descomposición)
  - Problemas de tamaño $n/2$ (aproximadamente)



n par



n impar

Dividir el tamaño del problema por dos – II

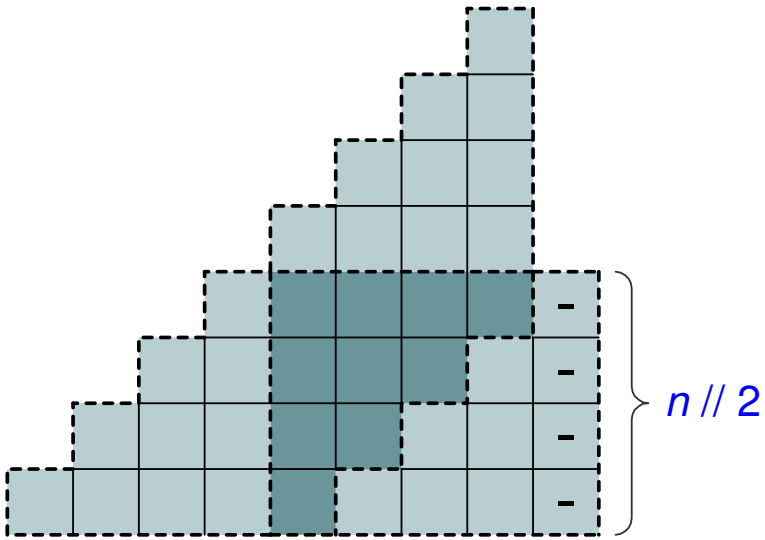
$$S(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ 2S(n/2) + (n/2)^2 & \text{si } n > 1 \text{ y } n \text{ par} \\ 2S((n-1)/2) + ((n+1)/2)^2 & \text{si } n > 1 \text{ y } n \text{ impar} \end{cases}$$

```
1 def suma(n):
2     if n==0:
3         return 0
4     elif n==1:
5         return 1
6     elif n%2==0:
7         return 2*suma(n//2) + (n//2)**2
8     else:
9         return 2*suma(n//2) + (n//2 + 1)**2
```

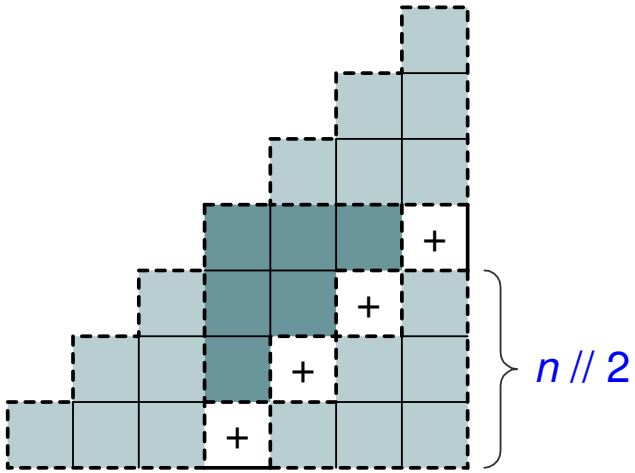
$T(n) = T(n/2) + 1$, con $T(0) = 1$ y $T(1) = 1 \quad \Rightarrow \quad T(n) \in \Theta(\log(n))$

Dividir el tamaño del problema por dos – III

- Paso 3 (descomposición)
  - Problemas de tamaño $n/2$ (aproximadamente)



n par



n impar

Dividir el tamaño del problema por dos – III

$$S(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ 4S(n/2) - (n/2) & \text{si } n > 1 \text{ y } n \text{ par} \\ 4S((n-1)/2) + (n+1)/2 & \text{si } n > 1 \text{ y } n \text{ impar} \end{cases}$$

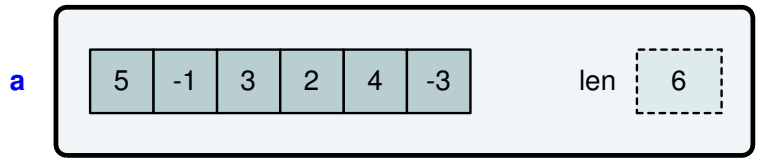
```
1 def suma(n):
2     if n==0:
3         return 0
4     elif n==1:
5         return 1
6     elif n%2==0:
7         return 4*suma(n//2) - n//2
8     else:
9         return 4*suma(n//2) + n//2 + 1
```

$T(n) = T(n/2) + 1$, con $T(0) = 1$ y $T(1) = 1 \quad \Rightarrow \quad T(n) \in \Theta(\log(n))$

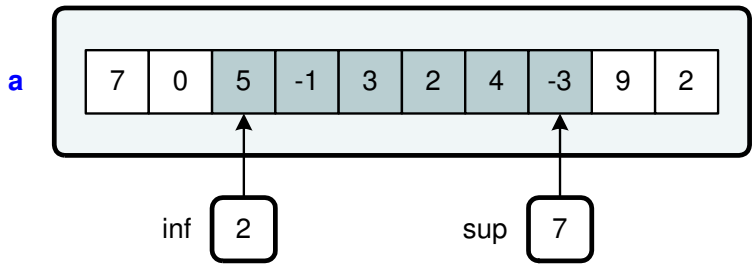
Arrays y listas

- Es importante conocer dos formas de implementar código relacionado con listas o arrays

1. Las funciones reciben la lista completa, pudiendo acceder a su longitud



2. Las funciones reciben la lista completa, pero se trabaja con una sublista de ésta, definida por un índice inferior y otro superior



   - Suele ser más eficiente, aunque también más compleja

Arrays y listas

Suma de los elementos de una lista

Dado una lista **a** de n números, hallar la suma de éstos:

$$s(\mathbf{a}) = \sum_{i=0}^{n-1} a_i$$

- Hay varias formas de emplear recursividad para resolver este problema, y varias implementaciones posibles

1. $s(\mathbf{a}) = \sum_{i=0}^{n-1} a_i = a_0 + \sum_{i=1}^{n-1} a_i = a_0 + s(\mathbf{a_{1..n-1}})$
2. $s(\mathbf{a}) = \sum_{i=0}^{n-1} a_i = \sum_{i=0}^{n-2} a_i + a_{n-1} = s(\mathbf{a_{0..n-2}}) + a_{n-1}$
3. $s(\mathbf{a}) = \sum_{i=0}^{n-1} a_i = \sum_{i=0}^{m} a_i + \sum_{i=m+1}^{n-1} a_i = s(\mathbf{a_{0..m}}) + s(\mathbf{a_{m+1..n-1}})$

Arrays y listas – Implementación tipo 1

```python
# s(a) => a[0] +  s(a[1:n])
def suma_lista_1(a):
    if len(a) == 0:
        return 0
    else:
        return a[0] + suma_lista_1(a[1:])


# s(a) => s(a[0:n-1]) + a[n-1]
def suma_lista_2(a):
    if len(a) == 0:
        return 0
    else:
        return suma_lista_2(a[:-1]) + a[-1]


# s(a) => s(a[0:n//2]) + s(a[n//2:n])
def suma_lista_3(a):
    if len(a) == 0:
        return 0
    elif len(a) == 1:
        return a[0]
    else:
        mitad = len(a)//2
        return suma_lista_3(a[:mitad]) + suma_lista_3(a[mitad:])
```

Arrays y listas – Implementación tipo 2

```
1  # s(a) => a[0] +  s(a[1:n])
2  def suma_lista_limites_1(a,inf,sup):
3      if inf>sup:
4          return 0
5      else:
6          return a[inf] + suma_lista_limites_1(a,inf+1,sup)
7
8
9  # s(a) => s(a[0:n-1]) + a[n-1]
10 def suma_lista_limites_2(a,inf,sup):
11     if inf>sup:
12         return 0
13     else:
14         return a[sup] + suma_lista_limites_2(a,inf,sup-1)
15
16
17 # s(a) => s(a[0:n//2]) + s(a[n//2:n])
18 def suma_lista_limites_3(a,inf,sup):
19     if inf>sup:
20         return 0
21     elif inf==sup:
22         return a[inf]  # or a[sup]
23     else:
24         mitad = (sup+inf)//2
25         return suma_lista_limites_3(a,inf,mitad) + suma_lista_limites_3(a,mitad+1,sup)
```

Problemas básicos

Suma de los dígitos de un número

Suma de los dígitos de un número entero no negativo

Dado un número $n \in \mathbb{N}$, hallar la suma de sus dígitos.

Ejemplo: Para $n = 3652$, se devuelve $s(3652) = 3 + 6 + 5 + 2 = 16$

- Paso 1 (tamaño):
  - El número de dígitos de n
- Paso 2 (casos base):
  - Instancia más pequeña: que el número solo contenga un dígito.
  - Si $n < 10$, $s(n) = n$
- Paso 3 (descomposición). Opciones:
  - Eliminar el dígito menos significativo $s(365)$ $\Leftarrow$ más sencillo
  - Eliminar el dígito más significativo $s(652)$
  - Dividir el número por la mitad $s(36)$, $s(52)$

Suma de los dígitos de un número

- Paso 3 (descomposición):
  - $s(n//10)$ es la suma de todos los dígitos de n, salvo el menos significativo. $s(365) = 14$
- Paso 4 (casos recursivos):
  - Si conocemos $s(n//10)$ solamente hace falta sumarle el dígito menos significativo de n: $(n\%10)$
  - En el ejemplo: $s(3652) = s(365) + 2$
  - Caso recursivo: $s(n) = s(n//10) + (n\%10)$
- Paso 5 (implementar):

```
1 def suma_digitos(n):
2     if n<10:
3         return n
4     else:
5         return suma_digitos(n//10) + n%10
```

Suma lenta

Suma lenta de dos números enteros no negativos

Dados $a, b \in \mathbb{N}$, hallar su suma. Solo se puede sumar o restar una unidad a los números.

Ejemplo: $s(a, b) = a + b$. Para $a = 3$ y $b = 5$, $s(3, 5) = 8$.

- Paso 1 (tamaño):
  - Hay varias opciones. Puede ser a, b o $\text{mín}(a, b)$. Empecemos tomando como tamaño a.
- Paso 2 (casos base):
  - Instancia más pequeña: $a = 0$. En ese caso $s(0, b) = b$
- Paso 3 (descomposición). Tenemos varias opciones válidas:
  - El subproblema $s(a - 1, b)$
  - El subproblema $s(a - 1, b + 1)$
  - El subproblema $s(a - 1, b - 1)$

Suma lenta – I

- Paso 3 (descomposición): $s(a-1,b) = a-1+b$
- Paso 4 (casos recursivos):
  - ¿Cómo conseguir $s(a,b)$ si ya sabemos $s(a-1,b)$? Solo hace falta sumarle 1
  - Caso recursivo: $s(a,b) = s(a-1,b)+1$
- Paso 5 (implementar):

```
1 def suma_lenta1(a,b):
2     if a==0:
3         return b
4     else:
5         return suma_lenta1(a-1,b) + 1
```

Suma lenta – II

- Paso 3 (descomposición): $s(a-1, b+1) = a-1+b+1 = a+b$
- Paso 4 (casos recursivos):
  - ¿Cómo conseguir $s(a, b)$ si ya sabemos $s(a-1, b+1)$? No hace falta hacer nada...
  - Caso recursivo: $s(a, b) = s(a-1, b+1)$
- Paso 5 (implementar):

```
1 def suma_lenta2(a,b):
2     if a==0:
3         return b
4     else:
5         return suma_lenta2(a-1,b+1)
```

Suma lenta – III

- Paso 3 (descomposición):
  $s(a-1, b-1) = a-1+b-1 = a+b-2$
- Paso 4 (casos recursivos):
  - ¿Cómo conseguir $s(a, b)$ si ya sabemos $s(a-1, b-1)$? Sumar 1 dos veces
  - Caso recursivo: $s(a, b) = s(a-1, b-1) + 1 + 1$
- Paso 5 (implementar):

```
1 def suma_lenta3(a,b):
2     if a==0:
3         return b
4     else:
5         return suma_lenta3(a-1,b-1) + 1 + 1
```

Suma lenta – IV

- Paso 1 (tamaño):
  - Ahora, tomemos $\text{mín}(a, b)$ como tamaño del problema.
- Paso 2 (casos base):
  - Instancia más pequeña: $a = 0$ o $b = 0$. Si $a = 0$ se retorna b, y si $b = 0$ se retorna a. Es decir, $s(0, b) = b$ y $s(a, 0) = a$
- Paso 3 (descomposición):
  - Consideramos el subproblema $s(a - 1, b - 1)$. Garantiza que se reduce el tamaño del problema.

Suma lenta – IV

- Paso 3 (descomposición):
  $s(a-1, b-1) = a-1+b-1 = a+b-2$
- Paso 4 (casos recursivos):
  - ¿Cómo conseguir $s(a, b)$ si ya sabemos $s(a-1, b-1)$? Sumar 1 dos veces
  - Caso recursivo: $s(a, b) = s(a-1, b-1) + 1 + 1$
- Paso 5 (implementar):

```
1 def suma_lenta4(a,b):
2     if a==0:
3         return b
4     elif b==0:
5         return a
6     else:
7         return suma_lenta4(a-1,b-1) + 1 + 1
```

Máximo de una lista

Máximo de una lista de números

Dada una lista $\mathbf{a}$, de longitud $n \geqslant 1$, hallar el máximo de sus elementos: $\text{máx}_{i=0..n-1}\{a_i\}$

Ejemplo: $m([4,2,7,5]) = 7$.

- Paso 1 (tamaño): n
- Paso 2 (casos base): Si $n = 1$, $s(\mathbf{a}) = a_0$
- Paso 3 (descomposición). Tenemos varias opciones válidas:
  - Eliminar el primer elemento de la lista (`a[1:]`) $m(\mathbf{a}_{1..n-1})$
  - Eliminar el último elemento de la lista (`a[:-1]`) $m(\mathbf{a}_{0..n-2})$
  - Dividir la lista por la mitad (`a[:mitad]` y `a[mitad:]`) $m(\mathbf{a}_{0..h-1})$ y $m(\mathbf{a}_{h..n-1})$, donde $h = n//2 = \lfloor n/2 \rfloor$.

Máximo de una lista – I

- Paso 3 (descomposición): $m(\mathbf{a}_{1..n-1})$
- Paso 4 (casos recursivos):
  - ¿Cómo conseguir $m(\mathbf{a})$ si ya sabemos $m(\mathbf{a}_{1..n-1})$? Será el mayor entre el encontrado en la sublista y el primer elemento de $\mathbf{a}$
  - Caso recursivo: $m(\mathbf{a}) = \max\{m(\mathbf{a}_{1..n-1}), a_0\}$
- Paso 5 (implementar):

```
1 def maximo1(a):
2     if len(a)==1:
3         return a[0]
4     else:
5         return max(maximo1(a[1:]),a[0])
```

Máximo de una lista – II

- Paso 3 (descomposición): $m(\mathbf{a}_{0..n-2})$
- Paso 4 (casos recursivos):
  - ¿Cómo conseguir $m(\mathbf{a})$ si ya sabemos $m(\mathbf{a}_{0..n-2})$? Será el mayor entre el encontrado en la sublista y el último elemento de $\mathbf{a}$
  - Caso recursivo: $m(\mathbf{a}) = \text{máx}\{m(\mathbf{a}_{0..n-2}), a_{n-1}\}$
- Paso 5 (implementar):

```
1 def maximo2(a):
2     n = len(a)
3     if n==1:
4         return a[0]
5     else:
6         return max(maximo2(a[:-1]),a[n-1])
```

Máximo de una lista – III

- Paso 3 (descomposición): $m(\mathbf{a}_{0..h-1})$ y $m(\mathbf{a}_{h..n-1})$
- Paso 4 (casos recursivos):
  - ¿Cómo conseguir $m(\mathbf{a})$ si ya sabemos $m(\mathbf{a}_{0..h-1})$ y $m(\mathbf{a}_{h..n-1})$? Será el mayor entre los máximos encontrados en ambas sublistas
  - Caso recursivo: $m(\mathbf{a}) = \text{máx}\{m(\mathbf{a}_{0..h-1}), m(\mathbf{a}_{h..n-1})\}$
- Paso 5 (implementar):

```
1 def maximo3(a):
2     n = len(a)
3     if n==1:
4         return a[0]
5     else:
6         mitad = n//2
7         return max(maximo3(a[:mitad]), maximo3(a[mitad:]))
```

Potencia con exponente no negativo

Potencia con exponente no negativo

Calcular b^n, donde $b \in \mathbb{R}$ y $n \in \mathbb{N}$ es un entero no negativo

Ejemplo: Si $f(b, n) = b^n$, $f(2, 5) = 2^5 = 32$

- Paso 1 (tamaño): n
- Paso 2 (casos base):
  - Si $n = 0$: $f(b, n) = 1$ (se asume que $0^0 = 1$)
- Paso 3 (descomposición): Opciones:
  - Considerar $f(b, n - 1)$
  - Considerar $f(b, n//2)$

Potencia con exponente no negativo – I

- Paso 4 (casos recursivos):
  - ¿Cómo conseguir b^n si conoces b^{n-1}?
  - Caso recursivo: $b^n = b \cdot b^{n-1}$
- Paso 5 (implementar):

```
1 def pot1(b,n):
2     if n==0:
3         return 1
4     else:
5         return b*pot1(b,n-1)
```

$$T(n) = \Theta(n)$$

Potencia con exponente no negativo – I

- Paso 4 (casos recursivos):
  - ¿Cómo conseguir b^n si conoces $b^{n//2}$?
    - Si n es par: $b^n = (b^{n//2})^2$
    - Si n es impar: $b^n = b \cdot (b^{n//2})^2$

- Paso 5 (implementar):

```
1 def pot2(b,n):
2     if n==0:
3         return 1
4     elif n%2==0:
5         p = pot2(b,n//2)
6         return p*p
7     else:
8         p = pot2(b,n//2)
9         return b*p*p
```

$$T(n) = \Theta(\log(n))$$

Escribir secuencia de caracteres – I

Escribir una secuencia de caracteres en orden

Dada una cadena de n caracteres **s**, escribir éstos verticalmente y en orden. Si **s** es la cadena vacía se escribirá un salto de línea.

Ejemplo: $f(\text{'hola'}) \rightarrow$ h↵o↵l↵a↵

- Paso 1 (tamaño): n
- Paso 2 (casos base):
  - Si $n = 0$: `print()`
  - Si $n = 1$: `print(s)`
- Paso 3 (descomposición):
  - Eliminar el primer elemento de **s** (`s[1:]`) $f1(\mathbf{s}_{1..n-1})$
  - Eliminar el último elemento de **s** (`s[:-1]`) $f2(\mathbf{s}_{0..n-2})$
  - Escribir ambas mitades por separado(`s[:mitad]` y `s[mitad:]`) $f3(\mathbf{s}_{0..h-1})$ y $f3(\mathbf{s}_{h..n-1})$, donde $h = n//2$.

Escribir secuencia de caracteres – II

Escribir una secuencia de caracteres en orden inverso

Dada una cadena de n caracteres **s**, escribir éstos verticalmente y en orden inverso. Si **s** es la cadena vacía se escribirá un salto de línea.

Ejemplo: $g(\text{'hola'}) \rightarrow$ a↵l↵o↵h↵

- Paso 1 (tamaño): n
- Paso 2 (casos base):
  - Si $n = 0$: `print()`
  - Si $n = 1$: `print(s)`
- Paso 3 (descomposición):
  - Eliminar el primer elemento de **s** (`s[1:]`) $g1(\mathbf{s}_{1..n-1})$
  - Eliminar el último elemento de **s** (`s[:-1]`) $g2(\mathbf{s}_{0..n-2})$
  - Escribir ambas mitades por separado(`s[:mitad]` y `s[mitad:]`) $g(\mathbf{s}_{0..h-1})$ y $g3(\mathbf{s}_{h..n-1})$, donde $h = n//2$.

Escribir secuencia de caracteres – I y II

- Pasos 4 y 5 (casos recursivos e implementación):

```
1  def f1(s):
2      n = len(s)
3      if n==0:
4          print()
5      elif n==1:
6          print(s)
7      else:
8          print(s[0])
9          f1(s[1:])
```

```
1  def f2(s):
2      n = len(s)
3      if n==0:
4          print()
5      elif n==1:
6          print(s)
7      else:
8          f2(s[:-1])
9          print(s[-1])
```

```
1  def f3(s):
2      n = len(s)
3      if n==0:
4          print()
5      elif n==1:
6          print(s)
7      else:
8          m = n//2
9          f3(s[:m])
10         f3(s[m:])
```

```
1  def g1(s):
2      n = len(s)
3      if n==0:
4          print()
5      elif n==1:
6          print(s)
7      else:
8          g1(s[1:])
9          print(s[0])
```

```
1  def g2(s):
2      n = len(s)
3      if n==0:
4          print()
5      elif n==1:
6          print(s)
7      else:
8          print(s[-1])
9          g2(s[:-1])
```

```
1  def g3(s):
2      n = len(s)
3      if n==0:
4          print()
5      elif n==1:
6          print(s)
7      else:
8          m = n//2
9          g3(s[m:])
10         g3(s[:m])
```

Sumatorio general

Calcular un sumatorio de una función general

Dada una función general $g(x)$ y unos límites m y n, hallar $f(m, n, g) = \sum_{i=m}^{n} g(i)$

- Paso 1 (tamaño): $n - m + 1$ (número de términos a sumar)
- Paso 2 (casos base):
  - Si $m > n$ (en ese caso el tamaño es 0 o negativo): $f(m, n, g) = 0$
  - Si $m = n$ (solo necesario si se divide el sumatorio en dos en la fase de descomposición): $f(m, n, g) = g(m)$
- Paso 3 (descomposición):
  - Eliminamos el primer elemento de la suma. Consideramos $f(m + 1, n, g)$: $\sum_{m}^{n} g(i) = g(m) + \sum_{m+1}^{n} g(i)$
  - Eliminamos el último elemento de la suma. Consideramos $f(m, n - 1, g)$: $\sum_{m}^{n} g(i) = \sum_{m}^{n-1} g(i) + g(n)$
  - Dividimos el sumatorio en dos (por la mitad). Consideramos $f(m, h, g)$ y $f(h + 1, n, g)$, donde $h = (n + m)//2 : \sum_{m}^{n} g(i) = \sum_{m}^{h} g(i) + \sum_{h+1}^{n} g(i)$

Sumatorio general

- Pasos 4 y 5 (casos recursivos e implementación):

```
1  def g(x):
2      return x*x;
3
4  def s1(m,n,g):
5      if m>n:
6          return 0
7      else:
8          return g(m) + s1(m+1,n,g)
9
10 def s2(m,n,g):
11     if m>n:
12         return 0
13     else:
14         return s2(m,n-1,g) + g(n)
15
16 def s3(m,n,g):
17     if m>n:
18         return 0
19     elif m==n:
20         return g(m)
21     else:
22         mitad = (n+m)//2
23         return s3(m,mitad,g) + s3(mitad+1,n,g)
```

¿Contiene dígito?

¿Contiene dígito?

Dado un entero no negativo n, y un dígito $d \in \{0, 1, \ldots, 9\}$, determinar si d es una de las cifras de n

Ejemplo: Si la función booleana es $f(n, d)$, $f(3082, 2) =$ `True`

- Paso 1 (tamaño): El número de dígitos de n
- Paso 2 (casos base):
  - Si n solo contiene un dígito el resultado es ¿$n = d$?
  - Si d es el último dígito de n (el menos significativo) el resultado es `True`
    - Se puede incorporar como caso base o en la regla recursiva
- Paso 3 (descomposición):
  - Consideramos $n//10$, que es n sin el último dígito.

¿Contiene dígito?

- Paso 4 (casos recursivos)
  - Si no estás en un caso base (d no es el último dígito de n) el resultado será $f(n//10, d)$ (si d es una de las cifras restantes de n)
- Paso 5 (implementación)

```
1  def contiene_digito_cola(n,d):
2      if n<10:
3          return n==d
4      elif n%10==d:
5          return True
6      else:
7          return contiene_digito_cola(n//10,d)
8
9  def contiene_digito_lineal(n,d):
10     if n<10:
11         return n==d
12     else:
13         return (n%10==d) or contiene_digito_lineal(n//10,d)
```

Problemas adicionales

Cambio de base

Cambio de base 10 a base b

Dado un número entero no negativo n, calcular su expresión en base b ($b \in [2, 9]$ es otro entero)

Ejemplo: $142_{10} = 1032_5$. Por tanto, $c(142, 5) = 1032$

- Paso 1 (tamaño): El número de dígitos de la salida
- Paso 2 (casos base):
  - Instancia más pequeña: que la salida solo tenga un dígito
    - $n < b$. En ese caso $c(n, b) = n$
- Paso 3 (descomposición).
  - ¿Cómo te acercas al caso base? Dividiendo n entre b. $c(n//b, b)$ tendrá un dígito menos que $c(n, b)$

Cambio de base

- Paso 4 (casos recursivos):
  - ¿Si conoces $c(n//b, b)$, cómo construyes $c(n, b)$?
  - Ejemplo. Para $n = 142$, $n//5 = 28$. Además, $c(142, 5) = 1032$ y $c(28, 5) = 103$
  - Tendríamos que multiplicar 103 por 10, y sumarle 2
  - La regla recursiva es: $c(n, b) = 10 \cdot c(n//b, b) + n\%b$

- Paso 5 (implementación):

```
1 def cambio_base(n,b):
2     if n<b:
3         return n
4     else:
5         return 10*cambio_base(n//b,b) + n%b
```

Evaluar un polinomio

Evaluar un polinomio

Sea $P(x, \mathbf{d}) = d_n x^n + d_{n-1} x^{n-1} + \cdots + d_1 x + d_0$ un polinomio de grado n, determinado por los coeficientes d_i, que se especifican en una lista $\mathbf{d}$ de longitud $n+1$. El problema consiste en evaluar el polinomio para un valor concreto de x.

Ejemplo: $P(3, [2, -4, 5]) = 5 \cdot (3)^2 - 4 \cdot (3)^1 + 2 = 35$

- Paso 1 (tamaño): n
- Paso 2 (casos base):
  - Instancia más pequeña:
    - $n = 0$. En ese caso $P(x, \mathbf{d}) = d_0$

Evaluar un polinomio – Algoritmo de Horner

- Paso 3 (descomposición)
  - Eliminamos el primer elemento de **d** (algoritmo de Horner)
    - También se puede eliminar el último, o partir el polinomio en dos

- Paso 4 (casos recursivos):
  - Asumimos que conocemos $P(x, \mathbf{d}_{1..n}) = d_n x^{n-1} + d_{n-1} x^{n-2} + \cdots + d_1$
  - La regla recursiva es: $P(x, \mathbf{d}) = d_0 + x \cdot P(x, \mathbf{d}_{1..n})$

- Paso 5 (implementación):

```
1 def horner(x,d):
2     if len(d)==1:
3         return d[0]
4     else:
5         return d[0] + x*horner(x,d[1:])
```

Búsqueda lineal

Búsqueda lineal

Dada una lista **a** de longitud n, no necesariamente ordenada, y un elemento x, devolver un índice de la lista en el que se encuentre x. Si x no está en la lista el método devolverá -1. El método será una función $f(\mathbf{a}, x)$.

- Paso 1 (tamaño): n
- Paso 2 (casos base):
  - $n = 0$. En ese caso el método devolverá -1.
  - Otros casos triviales son (no será necesario considerar ambos):
    - Si $x = a_{n-1}$ el método devolverá $n - 1$
    - Si $x = a_0$ el método devolverá 0
- Paso 3 (descomposición):
  - Eliminar el último elemento de la lista (veremos solo este caso)
  - Eliminar el primer elemento de la lista

Búsqueda lineal

- Paso 4 (casos recursivos):
  - Si hemos comprobado que $x \neq a_{n-1}$, entonces el resultado del método será el resultado de buscar a x en la sublista $\mathbf{a}_{0..n-2}$
  - En ese caso $f(\mathbf{a}, x) = f(\mathbf{a}_{0..n-2}, x)$
- Paso 5 (implementación):

```
1 def busqueda_lineal(a, x):
2     n = len(a)
3     if n==0:
4         return -1
5     elif a[n-1]==x:
6         return n-1
7     else:
8         return busqueda_lineal(a[:-1], x)
```

Búsqueda en lista ordenada

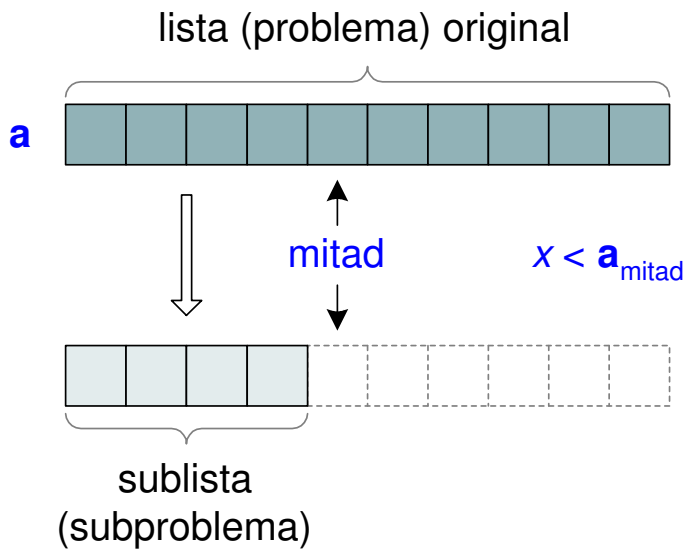
Búsqueda en lista ordenada

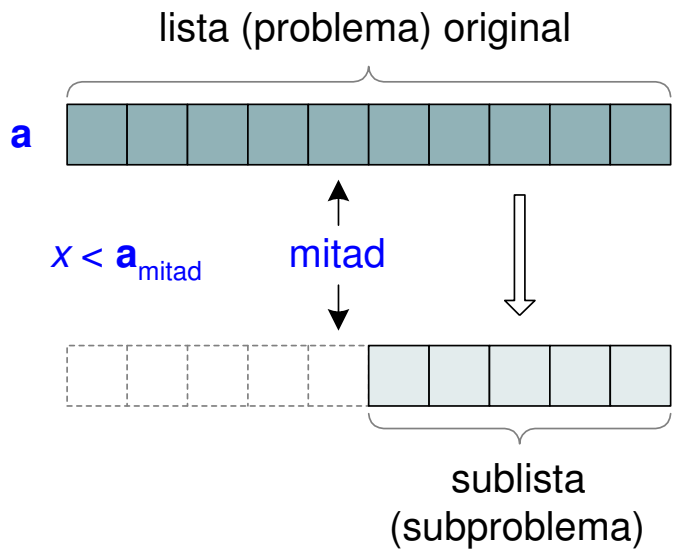
Dada una lista **a** de longitud n, ordenada de manera creciente, y un elemento x, devolver un índice de la lista en el que se encuentre x. Si x no está en la lista el método devolverá -1.

- Paso 1 (tamaño): n
- Paso 2 (casos base):
  - $n = 0$. La lista es vacía. En ese caso el método devolverá -1.
  - Si $x = a_m$ el método devolverá m. En el algoritmo que veremos (búsqueda binaria) m hará referencia a la mitad de la lista.
- Paso 3 (descomposición):
  - Descartar la búsqueda en la mitad superior o inferior de la lista. Esto conducirá a un algoritmo que correrá en tiempo $\Theta(\log(n))$

Búsqueda en lista ordenada

- Paso 4 (casos recursivos):
  - Si $x < a_m$ entonces x solo podrá estar en $\mathbf{a}_{0..m-1}$, por lo que la búsqueda debe continuar en esa primera mitad de la lista
  - Si $x > a_m$ entonces x solo podrá estar en $\mathbf{a}_{m+1..n-1}$, por lo que la búsqueda debe continuar en esa segunda mitad de la lista





Búsqueda en lista ordenada

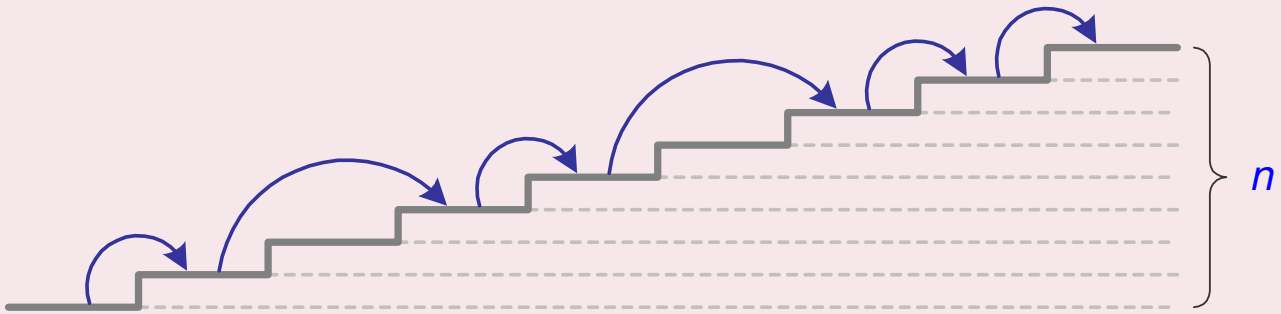
- Paso 5 (implementación):
  - Al tener que devolver un índice, es conveniente emplear parámetros que indiquen el principio (`inf`) y final (`sup`) de la sublista de **a** en la que se va a realizar la búsqueda
  - Se pasa toda la lista **a** (en realidad, un puntero a ella), pero la función debe trabajar solo con los índices entre `inf` y `sup`

```
1  def busqueda_binaria(a, x, inf, sup):
2      if inf>sup:
3          return -1
4      else:
5          mitad = (inf+sup)//2
6
7          if x==a[mitad]:
8              return mitad
9          elif x<a[mitad]:
10             return busqueda_binaria(a, x, inf,mitad-1)
11         else:
12             return busqueda_binaria(a, x, mitad+1, sup)
```

Formas de subir escaleras

Formas de subir escaleras

Halla el número de maneras en las que se puede subir unas escaleras de n peldaños, dando pasos de uno o dos peldaños.

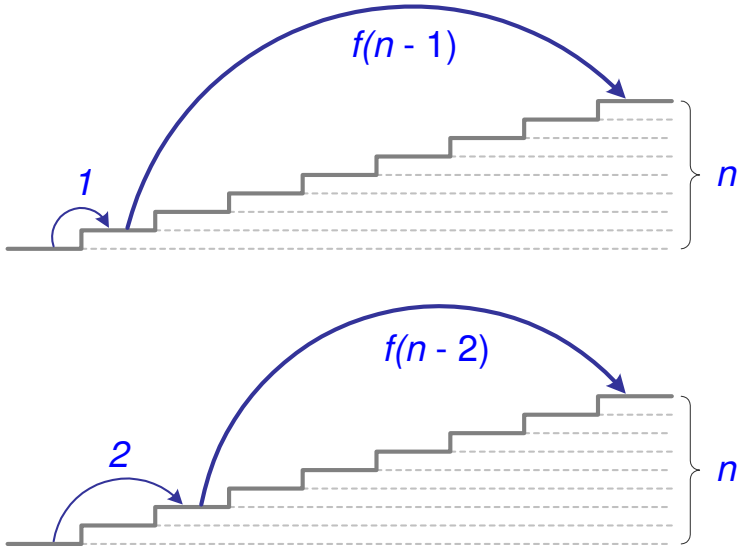


- Paso 1 (tamaño): n
- Paso 2 (casos base):
  - $n = 1$. Solo hay una forma de subir, $f(1) = 1$
  - $n = 2$. Hay dos formas de subir, $f(2) = 2$

Formas de subir escaleras

- Paso 3 (descomposición)
  - Suponemos que sabemos subir tanto $n-1$ como $n-2$ peldaños
  - Intentaremos construir $f(n)$ a partir de $f(n-1)$ y $f(n-2)$
- Paso 4 (casos recursivos):
  - En problemas combinatorios (de conteo) es recomendable agrupar los elementos a contar en diferentes categorías
  - En este caso hay dos tipos de formas de subir: las secuencias que comienzan subiendo un peldaño, y las que empiezan subiendo dos
    - El número de secuencias que empiezan subiendo un peldaño es $f(n-1)$
    - El número de secuencias que empiezan subiendo dos peldaños en un paso es $f(n-2)$

Formas de subir escaleras



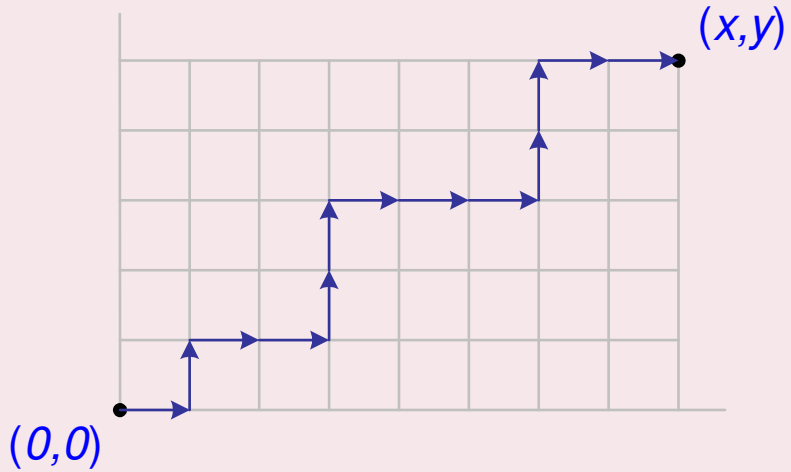
- Paso 5 (implementación):

```
1 def f(n):
2     if n==1:
3         return 1
4     elif n==2:
5         return 2
6     else:
7         return f(n-1) + f(n-2)
```

Caminos por Manhattan

Caminos por Manhattan

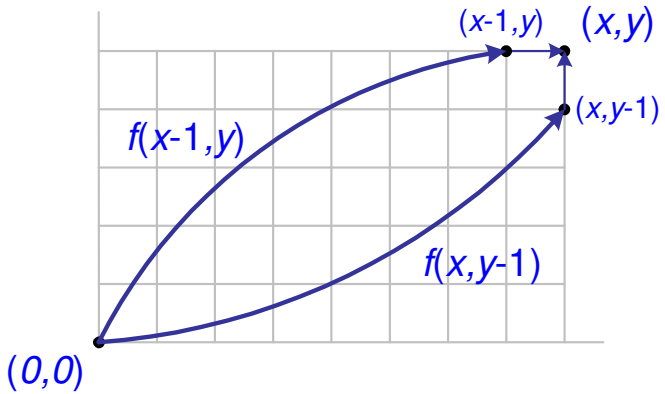
Hallar el número de formas de llegar desde el origen en el plano $(0, 0)$ hasta el punto (x, y), donde $x, y \in \mathbb{N}$, dando $x + y$ pasos, donde en cada uno solo se puede avanzar una unidad hacia arriba o hacia la derecha. Es decir, solo se puede incrementar en una unidad la coordenada x o la coordenada y en cada paso.



Caminos por Manhattan

- Paso 1 (tamaño): $x + y$
- Paso 2 (casos base):
  - $x = 0$ e $y = 0$ es un caso base, pero hay otro más general:
  - Si $x = 0$ o $y = 0$ solo habrá una forma de alcanzar (x, y)
    - Un camino totalmente horizontal o vertical
    - Si $x = 0$ e $y = 0$ se considera que hay un camino válido
- Paso 3 (descomposición)
  - Suponemos que sabemos llegar al punto $(x - 1, y)$ y al $(x, y - 1)$
  - Es decir, aprovechamos las soluciones $f(x - 1, y)$ y $f(x, y - 1)$
- Paso 4 (casos recursivos)
  - Desde $(x - 1, y)$ o $(x, y - 1)$ solo hay una manera de llegar a (x, y). Por tanto: $f(x, y) = f(x - 1, y) + f(x, y - 1)$

Caminos por Manhattan



- Paso 5 (implementación):

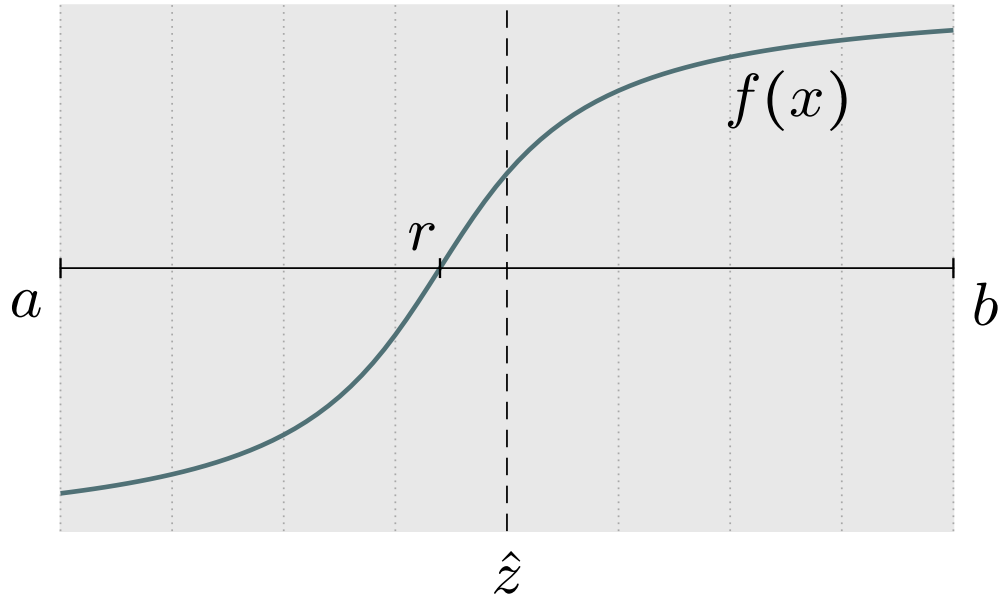
```
1 def f(x,y):
2     if (x==0) or (y==0):
3         return 1
4     else:
5         return f(x-1,y) + f(x,y-1)
```

$$f(x,y) = \frac{(x+y)!}{x! \cdot y!} = \binom{x+y}{y} = \binom{x+y}{x}$$

Encontrar un 0 de una función: bisección

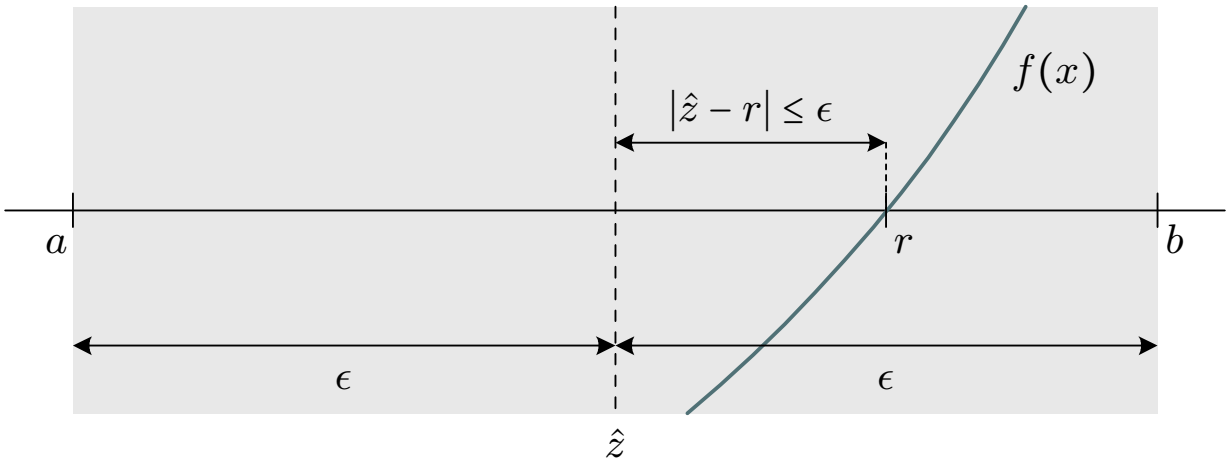
Encontrar un 0 de una función

Sea $f(x)$ una función continua en un intervalo [a,b]. Se sabe que $\text{signo}(f(a)) \neq \text{signo}(f(b))$, por lo que la función tendrá al menos un cero (r). Se pide hallar uno de estos ceros con una precisión de ϵ (la diferencia con respecto al cero real será ϵ como mucho). Por tanto, el resultado será una estimación $\hat{z}$. La función a implementar será $\zeta(a, b, f, \epsilon)$.



Encontrar un 0 de una función: bisección

- Paso 1 (tamaño): $b - a$, que, en función de ϵ, determinará el número de veces que se reducirá el intervalo de búsqueda por 2.
- Paso 2 (casos base):
  - Si $f(\hat{z}) = 0$ se podrá parar: se ha encontrado un cero exacto ($\hat{z} = r$)
  - Si $b - a < 2\epsilon$, la diferencia entre el cero real (r) y nuestra estimación del cero ($\hat{z}$) será menor o igual a ϵ y podremos parar

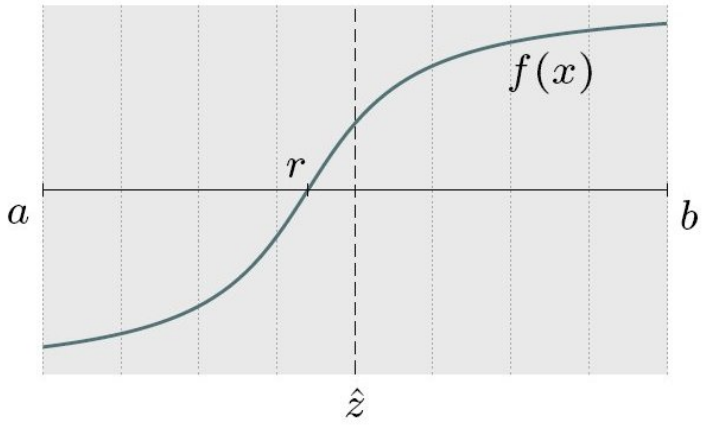


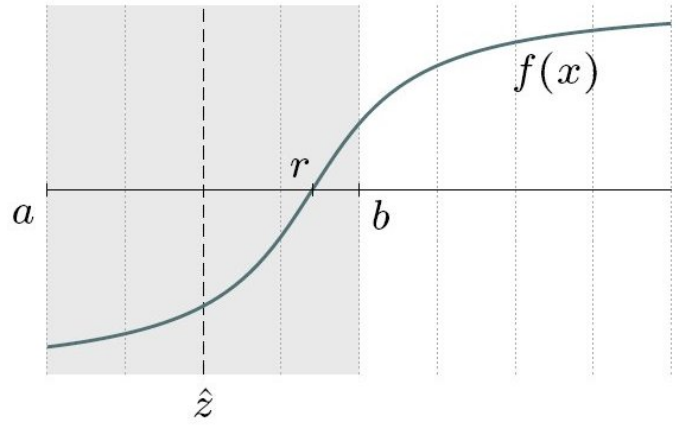
Encontrar un 0 de una función: bisección

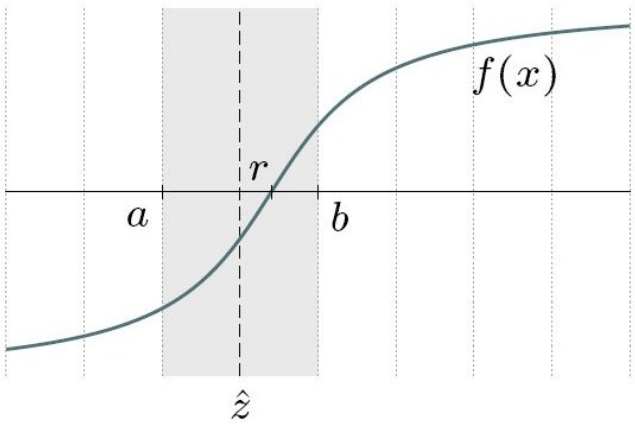
- Paso 3 (descomposición)
  - Reducimos el ancho del intervalo, dividiéndolo por dos
  - Tomando $\hat{z} = (a + b)/2$, $\hat{z}$ pasará a ser un extremo de un nuevo intervalo, de manera que los signos de la función en sus extremos sean diferentes
- Paso 4 (casos recursivos):
  - Inicialmente $\hat{z} = (a + b)/2$
  - Si $\text{signo}(f(a)) \neq \text{signo}(f(\hat{z}))$ entonces el resultado será $\zeta(a, \hat{z}, f, \epsilon)$
  - En caso contrario será $\zeta(\hat{z}, b, f, \epsilon)$

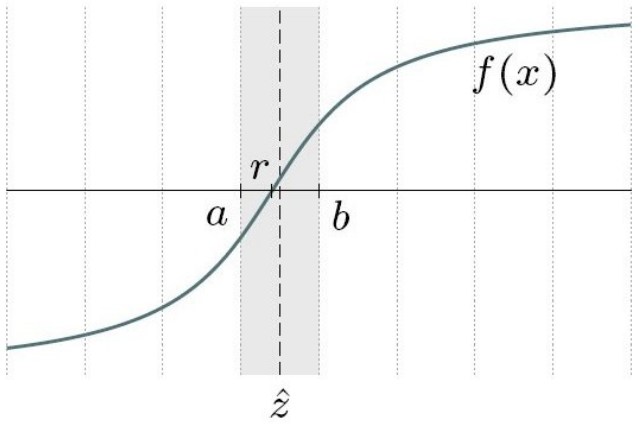
Encontrar un 0 de una función: bisección

- Proceso:









Encontrar un 0 de una función: bisección

- Paso 5 (implementación):

```
1  def f(x):
2      return x*x-2
3
4  def biseccion(a, b, f, epsilon):
5      z = (a+b)/2
6
7      if f(z)==0 or b-a<=2*epsilon:
8          return z
9      elif (f(a)>0 and f(z)<0) or (f(a)<0 and f(z)>0):
10         return biseccion(a, z, f, epsilon)
11     else:
12         return biseccion(z, b, f, epsilon)
```

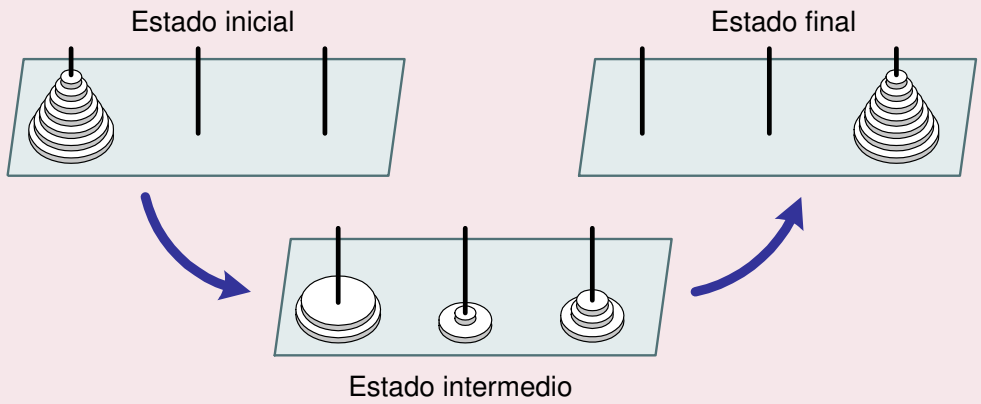
Torres de Hanoi

Torres de Hanoi

En este puzle se dispone de n discos de diferente tamaño (se puede asumir que el i-ésimo disco tiene radio i), insertados en una torre, en orden decreciente de tamaño (estado inicial). El objetivo consiste en determinar la secuencia de movimientos de discos individuales para ubicar a todos los discos en otra torre (estado final), usando una tercera como auxiliar, con las siguientes reglas:

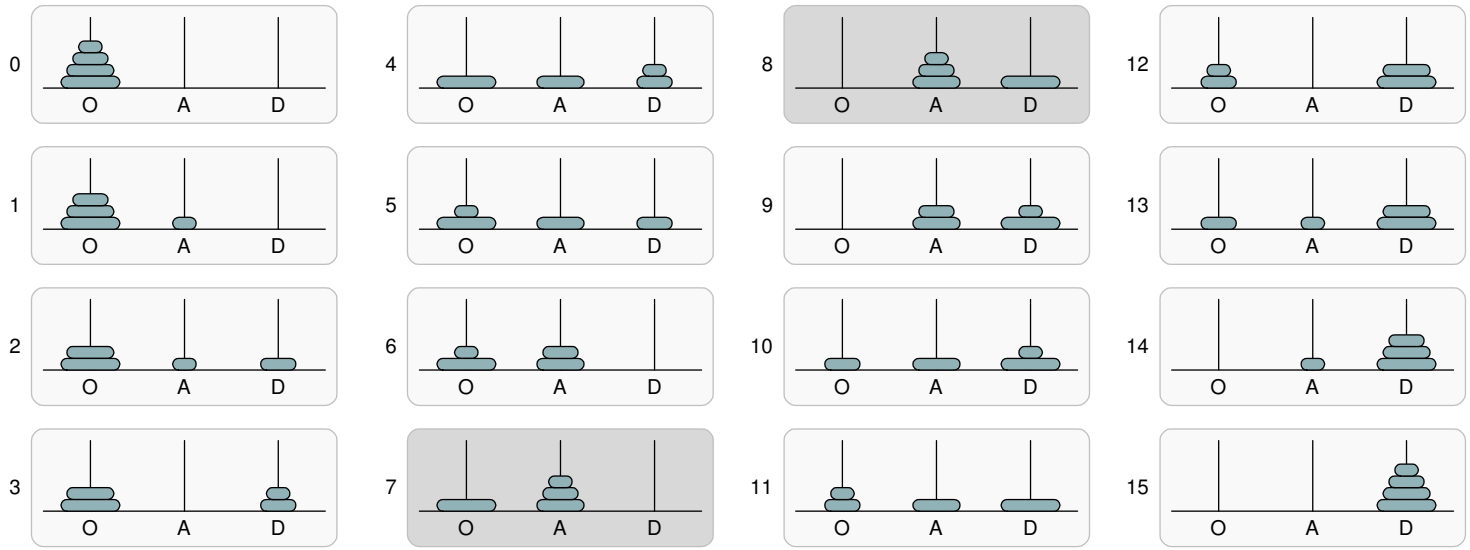
1. Solo se puede mover un disco a la vez
2. No puede haber un disco de mayor tamaño encima de otro de menor tamaño

Cada movimiento se imprimirá por pantalla como "Mover disco i desde t_1 a t_2", donde t_1 y t_2 podrán ser la torre origen, auxiliar o destino.



Torres de Hanoi

- Proceso, para $n = 4$ se puede resolver en 15 movimientos:
  - O: origen. A: auxiliar. D: destino



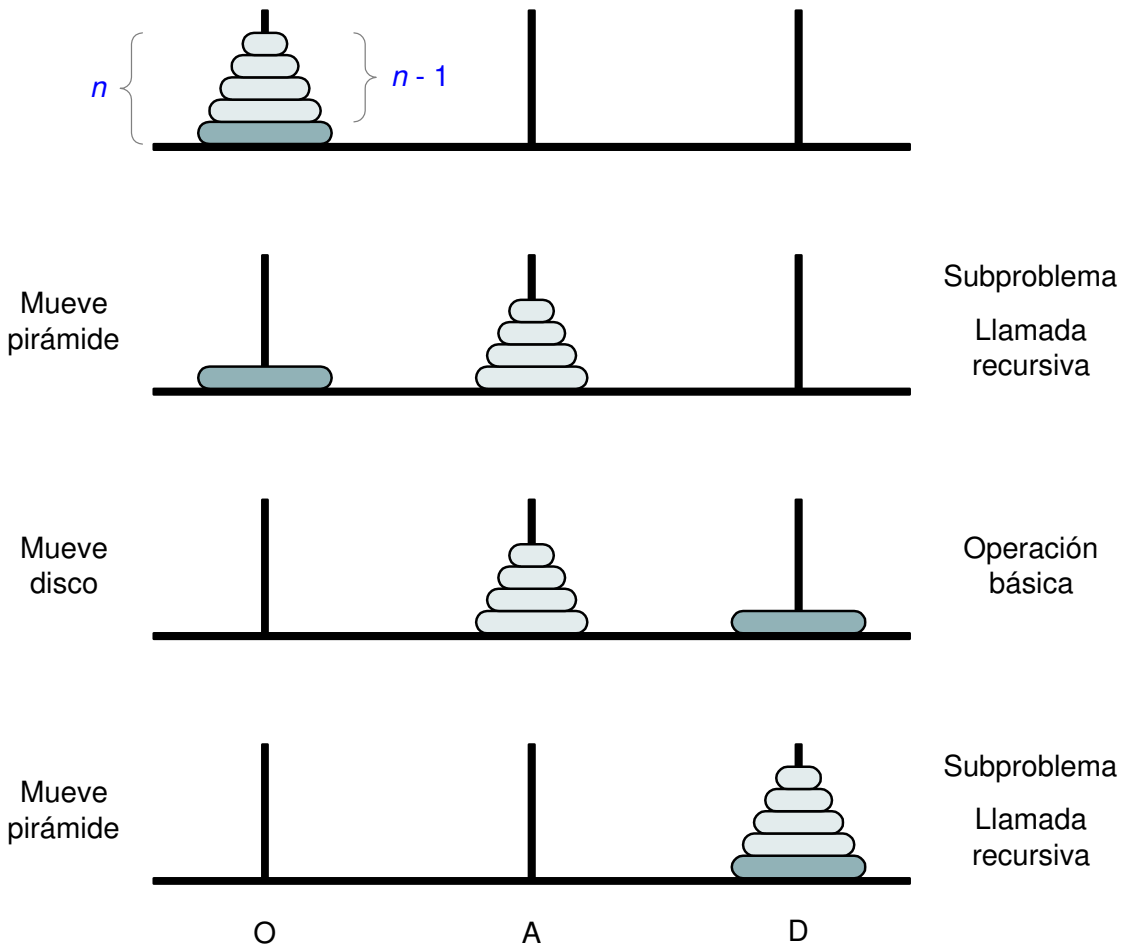
- No suele ser útil analizar todas las operaciones
  - Las etapas 7 y 8 son especialmente interesantes. ¿Por qué?
- Debemos centrarnos en subproblemas y sus soluciones

Torres de Hanoi

- Paso 1 (tamaño): n
- Paso 2 (casos base):
  - Si $n = 1$: mover el disco 1 desde la torre origen a la destino
  - También: Si $n = 0$: no se realiza ningún movimiento
- Paso 3 (descomposición):
  - Consideramos subproblemas de tamaño $n - 1$
  - Implicación clave: suponemos que sabemos mover toda una pirámide de $n - 1$ discos, desde cualquier torre a otra

Torres de Hanoi

- Paso 4 (casos recursivos):



Torres de Hanoi

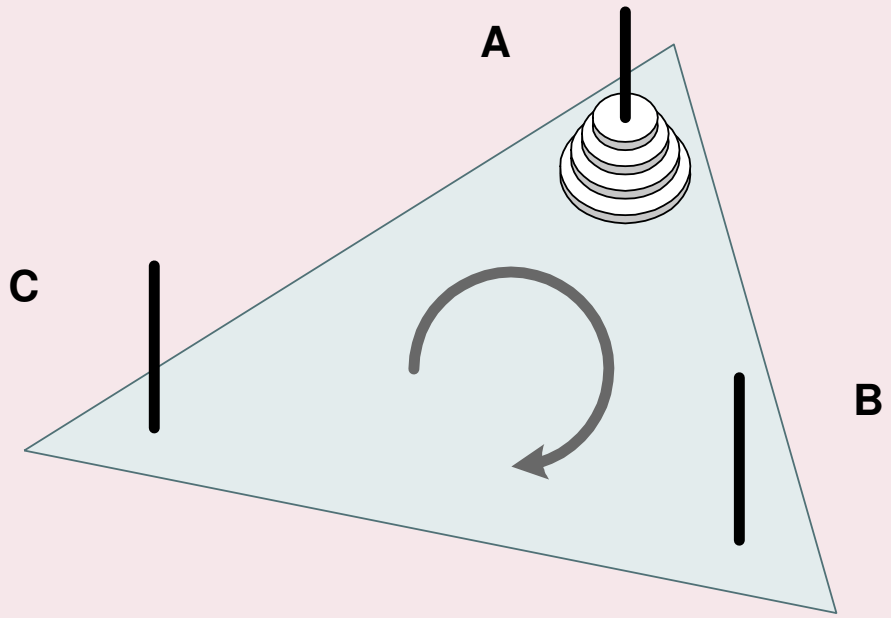
- Paso 5 (implementación):

```
1  def torres_Hanoi(n,o,d,a):
2      if n==1:
3          print('Mueve disco',n,'desde torre',o,'a torre',d)
4      else:
5          torres_Hanoi(n-1,o,a,d)
6          print('Mueve disco',n,'desde torre',o,'a torre',d)
7          torres_Hanoi(n-1,a,d,o)
8
9
10 # Código alternativo
11 def torres_Hanoi_alt(n,o,d,a):
12     if n>0:
13         torres_Hanoi_alt(n-1,o,a,d)
14         print('Mueve disco',n,'desde torre',o,'a torre',d)
15         torres_Hanoi_alt(n-1,a,d,o)
```

Torres de Hanoi cíclicas

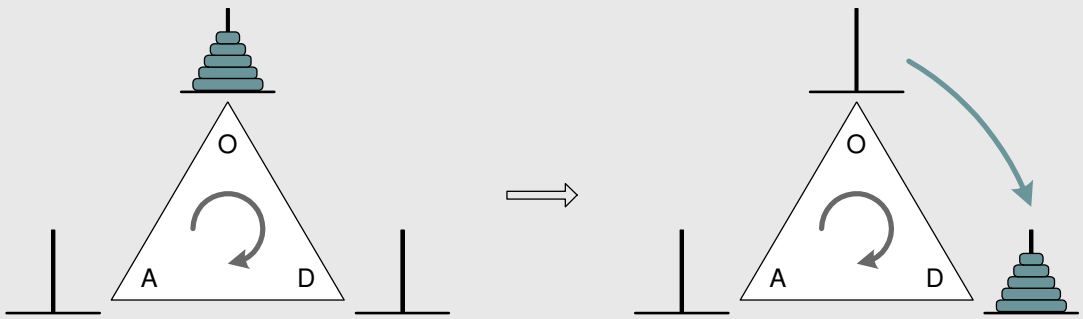
Torres de Hanoi cíclicas

Este puzle is idéntico al de las torres de Hanoi original, pero añade una restricción sobre el movimiento de los discos. Si las torres se ubican en los vértices de un triángulo, los discos solo podrán pasar de una torre a otra en el sentido de las agujas del reloj.

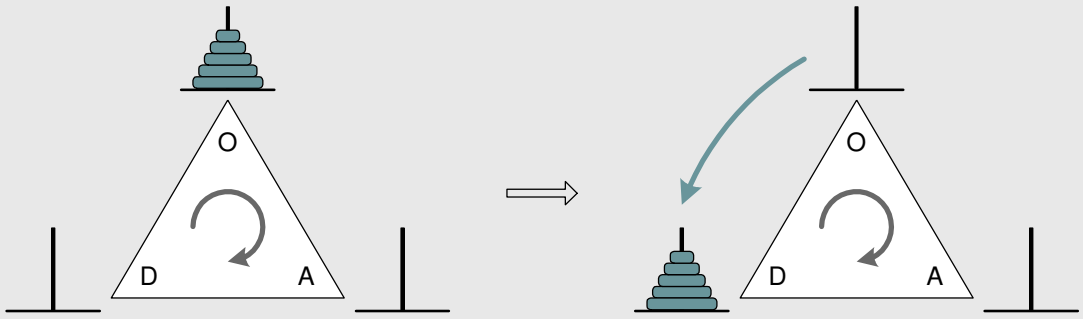


Torres de Hanoi cíclicas

- Realmente tenemos dos problemas distintos:



PROBLEMA 1: Mover *n* discos en el sentido de las agujas del reloj



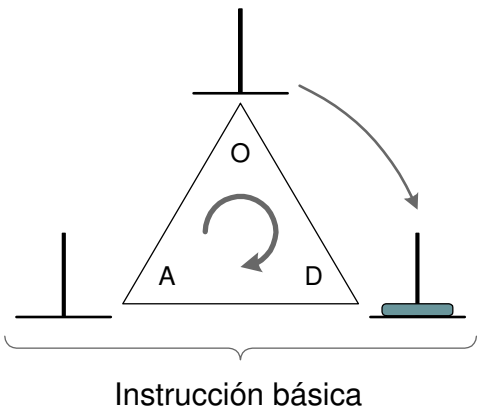
PROBLEMA 2: Mover *n* discos en el sentido contrario al de las agujas del reloj

Torres de Hanoi cíclicas

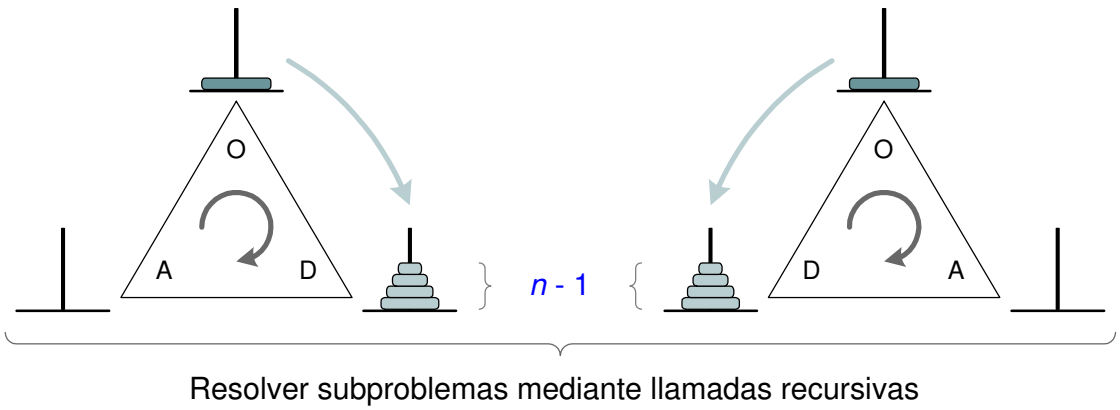
- Paso 1 (tamaño): n
- Paso 2 (casos base):
  - Si $n = 1$ ("reloj"): mover el disco 1 desde la torre origen a la destino
  - Si $n = 1$ ("antireloj"): mover el disco 1 desde la torre origen a la auxiliar, y luego a la destino
  - También: Si $n = 0$: no se realiza ningún movimiento
- Paso 3 (descomposición):
  - Consideramos subproblemas de tamaño $n - 1$, para ambos problemas
  - Implicación clave: suponemos que sabemos mover toda una pirámide de $n - 1$ discos, en cualquiera de los dos sentidos

Torres de Hanoi cíclicas

- Paso 4 (casos recursivos). Operaciones que podemos realizar:
  - Instrucción básica
    - Mover un disco
  - Llamadas recursivas
    - Mover $n-1$ discos en el sentido de las agujas del reloj
    - Mover $n-1$ discos en el sentido contrario al de las agujas del reloj



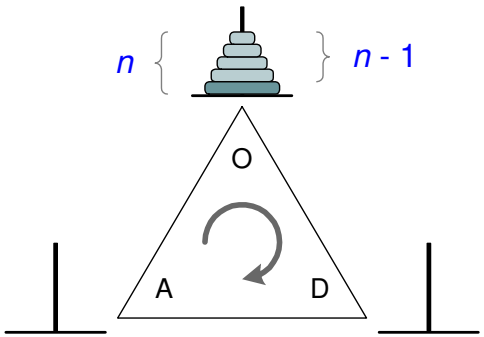
Instrucción básica



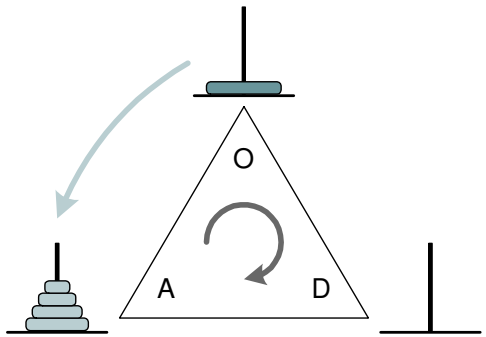
Resolver subproblemas mediante llamadas recursivas

Torres de Hanoi cíclicas

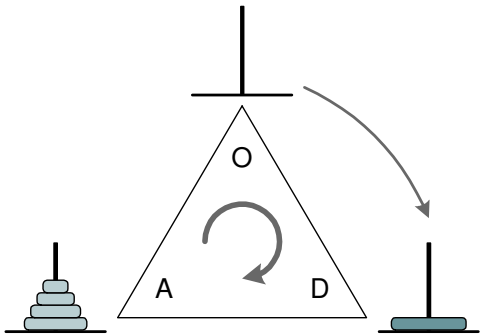
- Paso 4 (casos recursivos). Sentido "reloj"



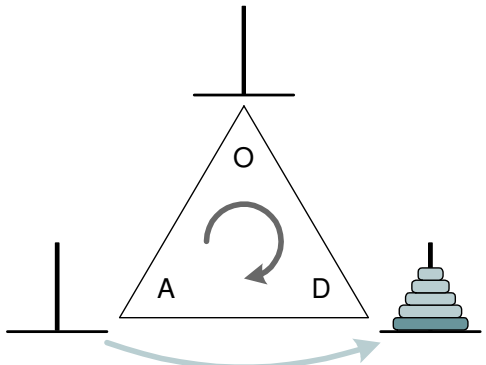
Inicio



Paso 1



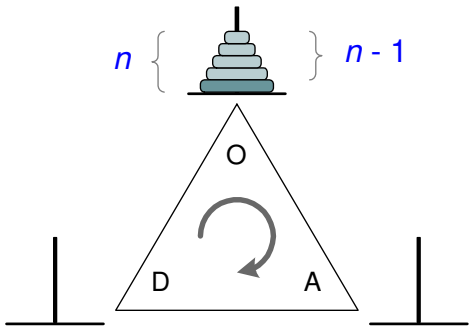
Paso 2



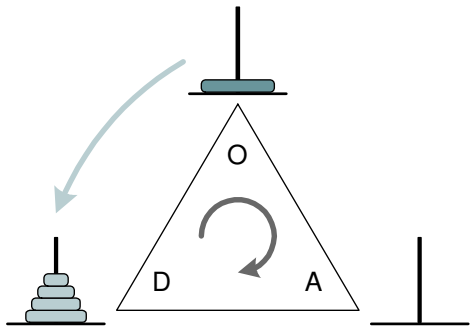
Paso 3

Torres de Hanoi cíclicas

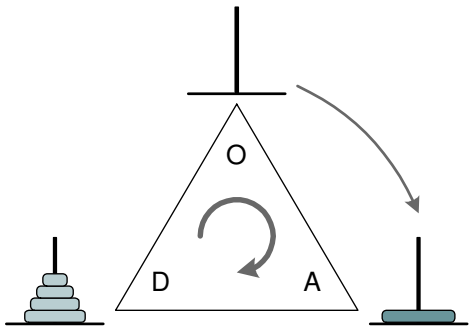
- Paso 4 (casos recursivos). Sentido "antireloj"



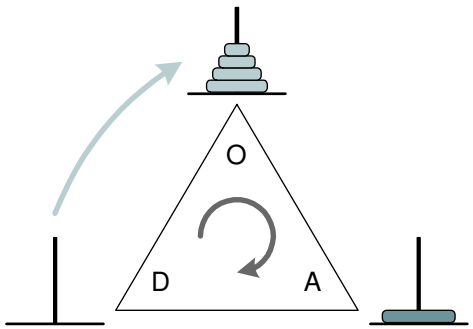
Inicio



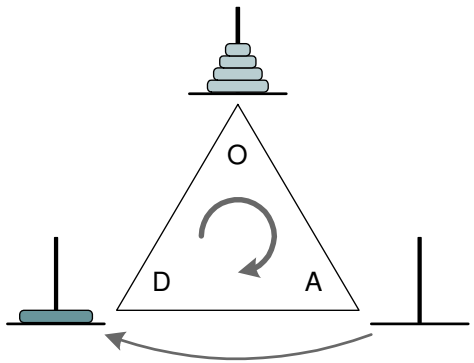
Paso 1



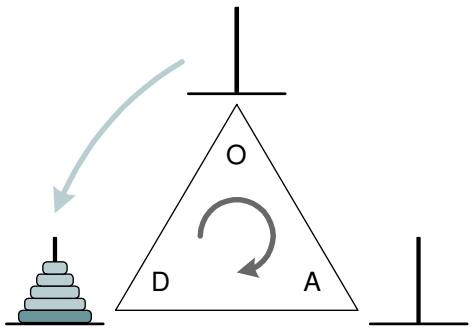
Paso 2



Paso 3



Paso 4



Paso 5

Torres de Hanoi cíclicas

- Paso 5 (implementación):

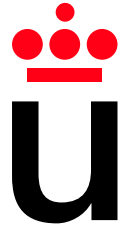
```
1  def reloj(n,o,d,a):
2      if n>0:
3          antireloj(n-1,o,a,d)
4          print('Mueve disco',n,'desde torre',o,'a torre',d)
5          antireloj(n-1,a,d,o)
6
7
8  def antireloj(n,o,d,a):
9      if n>0:
10         antireloj(n-1,o,d,a)
11         print('Mueve disco',n,'desde torre',o,'a torre',d)
12         reloj(n-1,d,o,a)
13         print('Mueve disco',n,'desde torre',o,'a torre',d)
14         antireloj(n-1,o,d,a)
```

Tema 5
Divide y vencerás

Diseño y Análisis de Algoritmos

Manuel Rubio Sánchez

13 de mayo de 2024



Universidad
Rey Juan Carlos

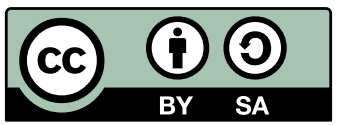
Copyright

©2024 Manuel Rubio Sánchez

Algunos derechos reservados.

Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en:

https://creativecommons.org/licenses/by-sa/4.0/deed.es.



Contenido

Introducción

Divide y vencerás

- Uno de los más importantes paradigmas de diseño algorítmico
- Se basa en la resolución de un problema dividiéndolo en **dos o más** subproblemas de igual tipo o similar
- El proceso continúa hasta que éstos llegan a ser lo suficientemente sencillos como para que se resuelvan directamente
- Al final, las soluciones a cada uno de los subproblemas se combinan para dar una solución al problema original
- Muchos algoritmos basados en esta estrategia son recursivos
  - También hay algoritmos iterativos de divide y vencerás

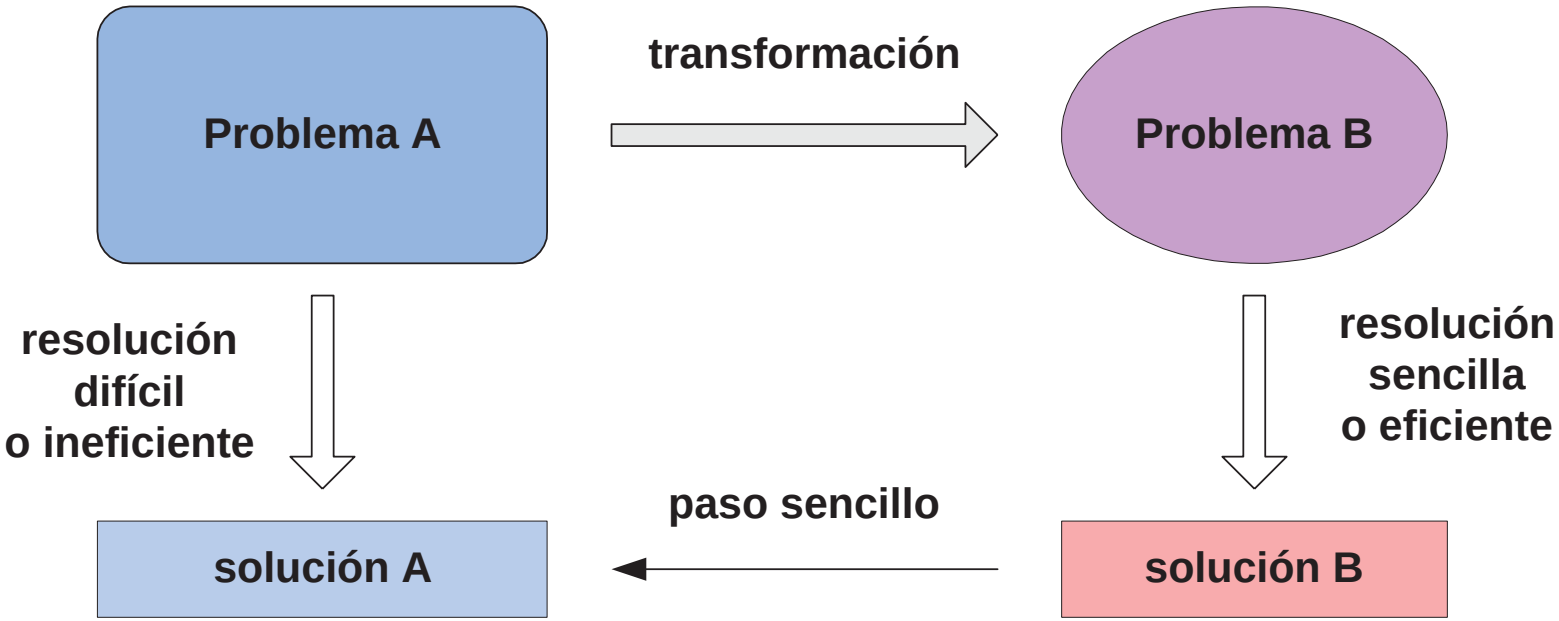
Estrategias de diseño de algoritmos

- **Transforma y vencerás**
  - Reducciones (vistas en el Tema 1)
- **Decrementa y vencerás**
  - Se reduce el tamaño del problema (p.e., $n-1$, $n//2$, $n//10$, etc.)
  - Una sola llamada recursiva en cada caso recursivo
  - Se requiere la solución de un solo subproblema
  - Ejemplos vistos en el Tema 4
- **Divide y vencerás**
  - Se reduce el tamaño del problema (p.e., $n//2$, $n//4$, etc.)
  - Más de una llamada en algún caso recursivo
  - Se requiere la solución de varios subproblemas

Transforma y vencerás

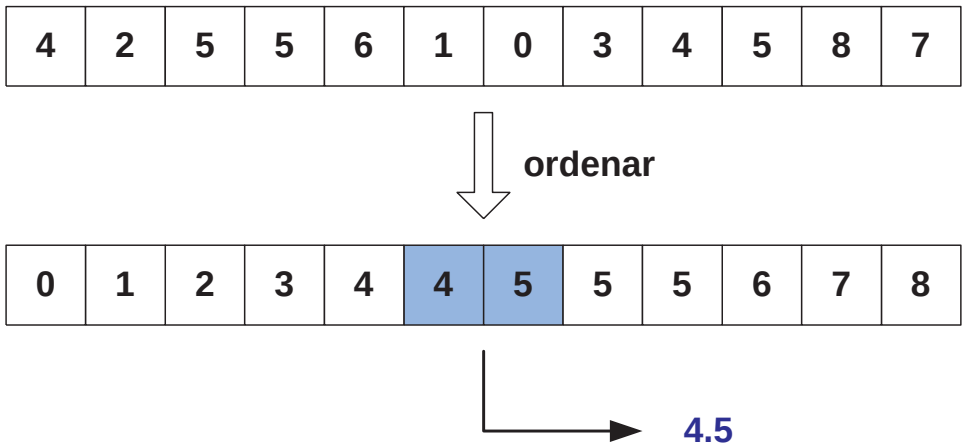
Transforma y vencerás

- Esta estrategia también se denomina **reducción**
- Resolver un problema transformándolo en otro más simple, conocido, o para el que existan algoritmos eficientes



Mediana de un array v desordenado

1. Ordenar el array **v** $\longrightarrow \Theta(n \log n)$
2. Devolver:
   - $v[n//2]$ si n es impar
   - $\dfrac{v[(n//2)-1]+v[n//2]}{2}$ si n es par



- Hay algoritmos más eficientes (p.e., "Quickselect")

Estimación del valor de π

- Planteamos un problema alternativo
- Buscamos una **aproximación del perímetro p de un círculo** de radio R

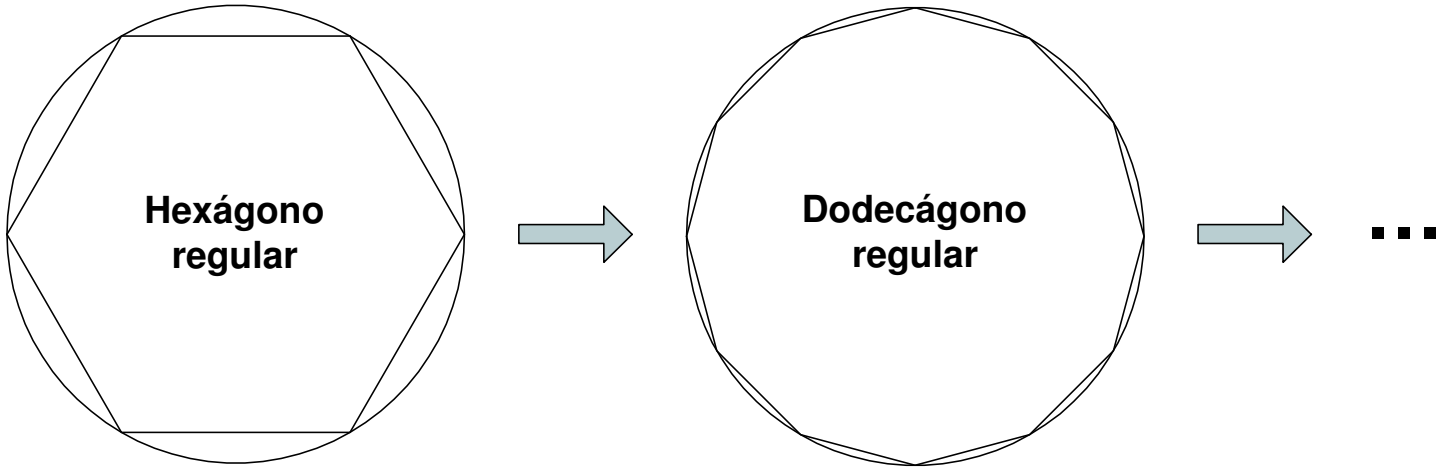
$$p = 2\pi R \quad \Rightarrow \quad \pi = \frac{p}{2R}$$

- Aproximamos un círculo con un polígono regular inscrito o circunscrito al círculo
- El perímetro del polígono inscrito p_i nos dará una cota inferior de p
- El perímetro del polígono circunscrito p_c nos dará una cota superior de p

$$\frac{p_i}{2R} \leqslant \pi \leqslant \frac{p_c}{2R}$$

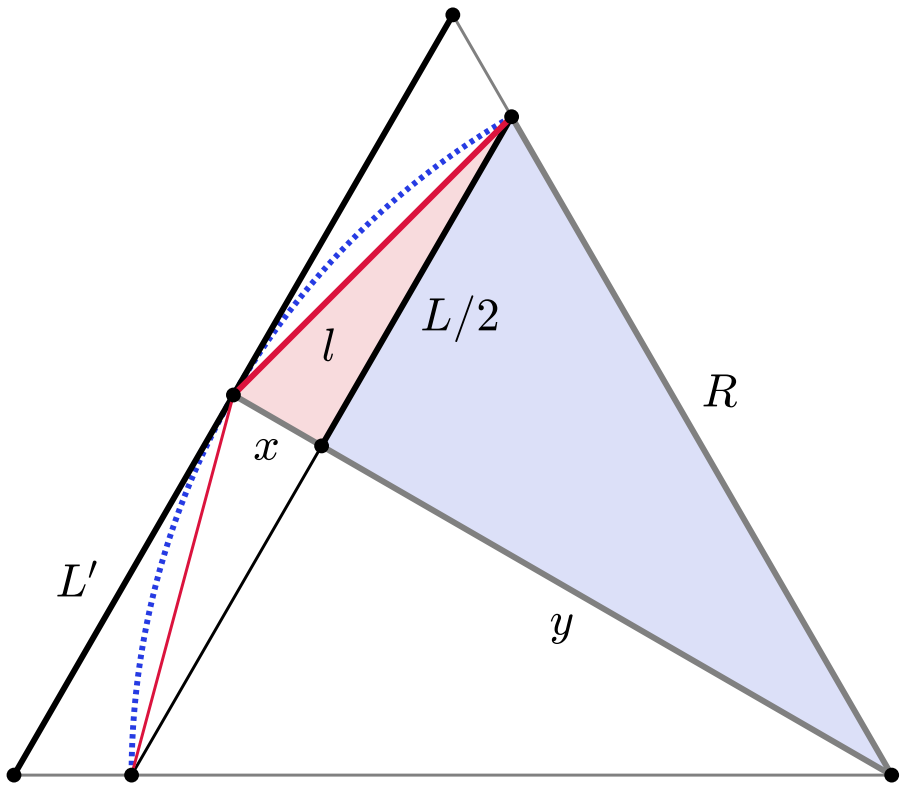
Estimación del valor de π

- Comenzamos por un polígono regular inscrito (por ejemplo, un hexágono), cuyo perímetro será $6R$
- A continuación, doblamos el número de lados, obteniendo una mejor aproximación



- El perímetro de un polígono con el doble de lados se puede obtener fácilmente a partir del primero
- Los perímetros de los polígonos circunscritos se pueden obtener de los perímetros de los polígonos inscritos

Estimación del valor de π



$$R^2 = y^2 + \left(\frac{L}{2}\right)^2$$

$$R = x + y$$

$$l^2 = x^2 + \left(\frac{L}{2}\right)^2$$

$$\frac{L'}{R} = \frac{l}{y}$$

Estimación del valor de π

- Si el lado de un polígono inscrito mide L, se puede hallar la longitud un lado del polígono l con el doble de lados

$$R^2 = y^2 + \left(\frac{L}{2}\right)^2 \quad \Rightarrow \quad y = \sqrt{R^2 - \left(\frac{L}{2}\right)^2} \qquad x = R - y = R - \sqrt{R^2 - \left(\frac{L}{2}\right)^2}$$

$$l = \sqrt{\left(\frac{L}{2}\right)^2 + R^2 + R^2 - \left(\frac{L}{2}\right)^2 - 2R\sqrt{R^2 - \left(\frac{L}{2}\right)^2}} = R\sqrt{2 - \sqrt{4 - \left(\frac{L}{R}\right)^2}}$$

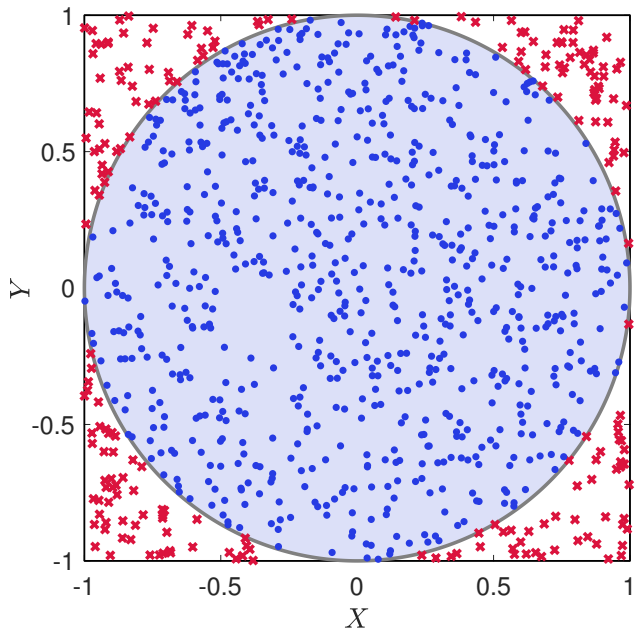
- El lado L' de un polígono circunscrito se puede hallar a partir del lado L de un polígono inscrito

$$\frac{L'}{R} = \frac{L}{y} \quad \Rightarrow \quad L' = \frac{LR}{y}$$

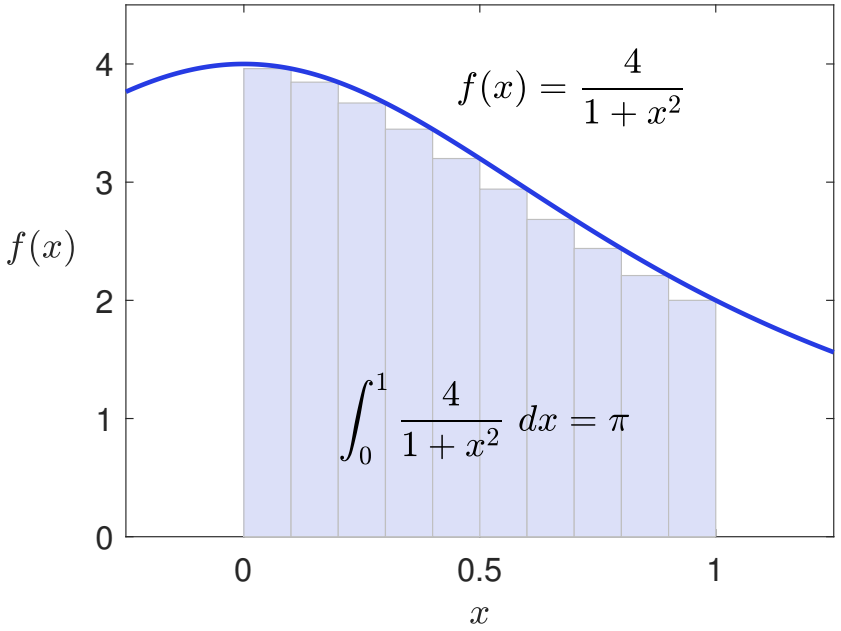
- la estimación de π se obtiene a partir de los perímetros hallados:

$$\frac{L \cdot \text{número de lados}}{2R} \leqslant \pi \leqslant \frac{L' \cdot \text{número de lados}}{2R}$$

Otros métodos para estimar del valor de π



Simulación Monte Carlo



Integración numérica

- Series

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots \qquad \text{para } x = 1 \quad \Rightarrow \quad \frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

Raíz cuadrada

- Deseamos hallar $\sqrt{a}$
- Transformamos el problema en el de **hallar un cero de una función**

$$x = \sqrt{a} \quad \rightarrow \quad x^2 = a \quad \rightarrow \quad x^2 - a = 0$$

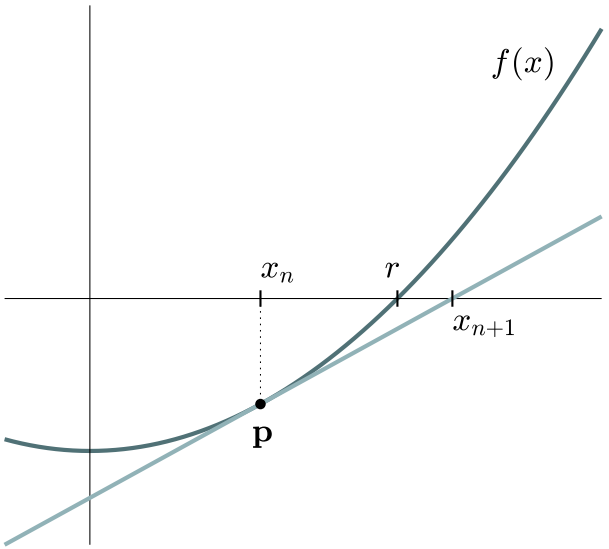
- Por tanto, querremos hallar un cero de la función $y = x^2 - a$
- Como x será generalmente un número real, nos podemos conformar con una aproximación
- Hay varios métodos para resolver este nuevo problema, por ejemplo:
  - Método de Newton-Raphson (que veremos a continuación)
  - Método de bipartición (de "decrementa y vencerás")

Método de Newton-Raphson

- Se trata de un método iterativo, que parte de un valor inicial x_0, y aplica la siguiente regla para hallar un cero (r) de la función $f(x)$:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

- En cada iteración se busca la recta tangente a $f(x)$ que pasa por el punto $\mathbf{p} = (x_n, f(x_n))$
- El corte de la recta con el eje de abscisas constituye el nuevo punto x_{n+1}



Método de Newton-Raphson

- El método surge del desarrollo de $f(x)$ en serie de Taylor, para un entorno del punto x_n:

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + (x - x_n)^2 \frac{f''(x_n)}{2!} + \ldots$$

- Si se trunca el desarrollo a partir del término de grado 2, se obtiene la recta tangente (que pasa por el punto $(x_n, f(x_n))$)
- Posteriormente evaluamos en x_{n+1}:

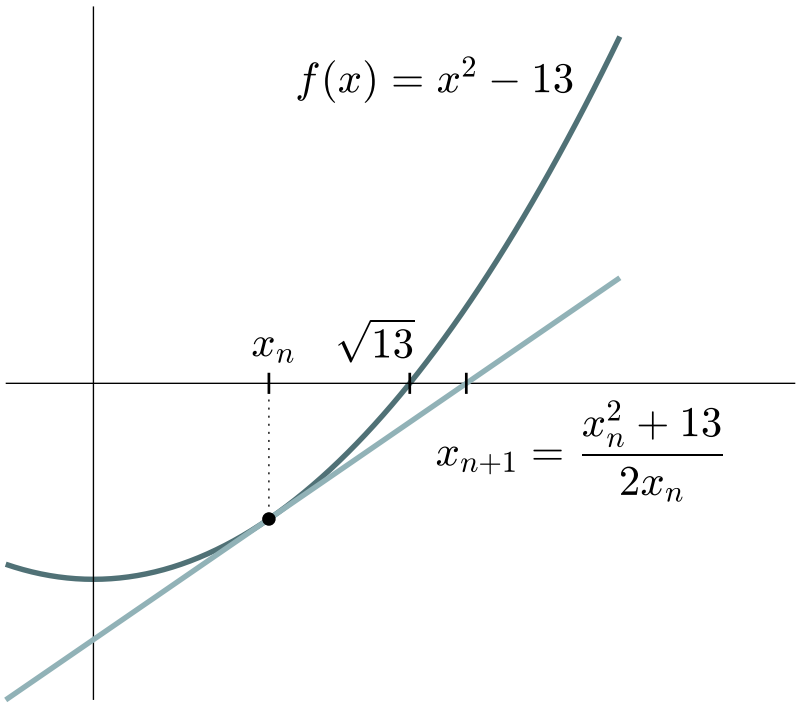
$$f(x_{n+1}) = f(x_n) + f'(x_n)(x_{n+1} - x_n)$$

- Si asumimos que x_{n+1} tiende hacia el cero de la función, podemos sustituir $f(x_{n+1}) = 0$, obteniéndose el algoritmo

Raíz cuadrada - método de Newton-Raphson

- Debemos aplicar la regla para $f(x) = x^2 - a$, con $f'(x) = 2x$

$$x_{n+1} = x_n - \frac{x_n^2 - a}{2x_n} = \frac{2x_n^2}{2x_n} - \frac{x_n^2 - a}{2x_n} = \frac{x_n^2 + a}{2x_n}$$



x_0	13,000000000000000000
x_1	7,000000000000000000
x_2	4,428571428571428825
x_3	3,682027649769585409
x_4	3,606345489465518472
x_5	3,605551362917601476
x_6	3,605551275463990457
x_7	3,605551275463989124
x_8	3,605551275463989568
x_9	3,605551275463989124
x_{10}	3,605551275463989568
x_{11}	3,605551275463989124
$\sqrt{13}$	3,605551275463989124

Factorización LU

- Supongamos que queremos resolver un sistema de ecuaciones $\mathbf{Ax} = \mathbf{b}$, donde $\mathbf{A}$ es una matriz cuadrada e invertible de dimensiones $n \times n$
- Una solución: $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$
  - $\mathbf{A}^{-1}$ se calcula en $\mathcal{O}(n^3)$, pero no suele ser buena idea por:
    - Errores de precisión
    - $\mathbf{A}^{-1}$ puede ser densa incluso cuando $\mathbf{A}$ es dispersa
- Alternativa: **factorización LU**, que también se calcula en $\mathcal{O}(n^3)$
- $\mathbf{A} = \mathbf{LU}$, donde $\mathbf{L}$ es triangular inferior, y $\mathbf{U}$ triangular superior (del mismo tamaño). Para una matrix de 3×3 tendríamos:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

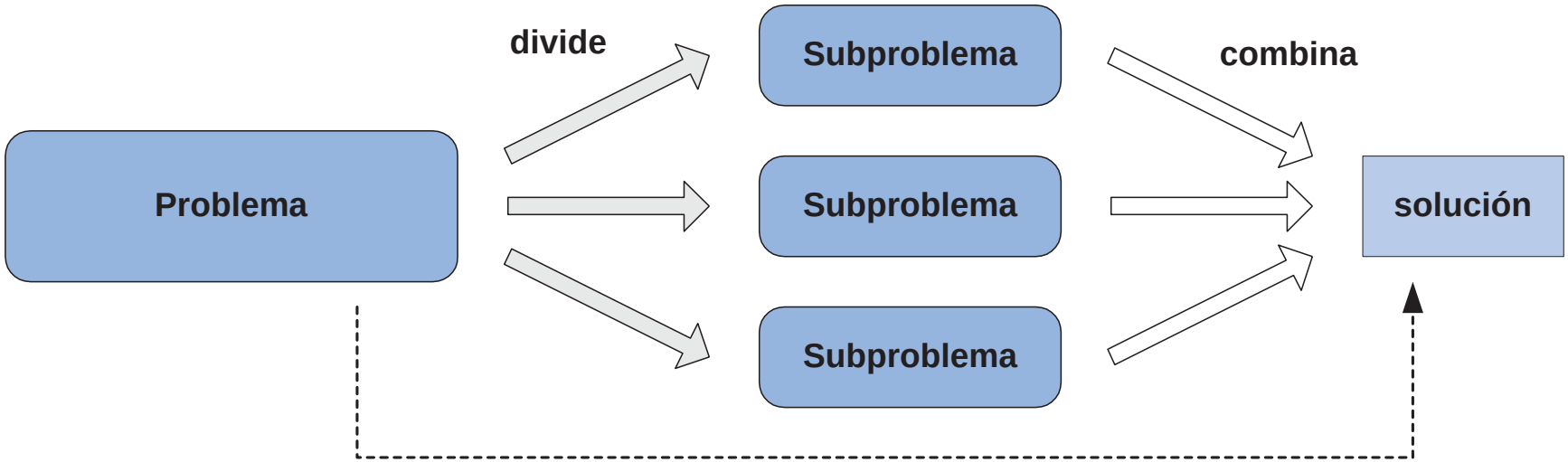
- Para resolver el sistema $\mathbf{Ax} = \mathbf{b}$ ahora se van a resolver dos, pero más sencillos:

$$\mathbf{Ax} = \mathbf{LUx} = \mathbf{b} \quad \Rightarrow \quad \boxed{\mathbf{Ly} = \mathbf{b}}, \quad \boxed{\mathbf{Ux} = \mathbf{y}}$$

Divide y vencerás

Divide y vencerás

- La descomposición de un problema genera varios subproblemas
- Las soluciones a los subproblemas se combinan para generar la solución al problema original, la cual puede requerir alguna operación adicional



Divide y vencerás – Ventajas

- Base de numerosos algoritmos
  - Transformada Rápida de Fourier (FFT)
    - Posiblemente el algoritmo más importante en la actualidad
  - Problemas geométricos
    - Par de puntos más próximos en el plano
    - Cierre convexo
  - Skyline (en el libro, apartado 6.7)
  - Otros que veremos en este tema...
- Aumento de eficiencia en términos de coste computacional
- Paralelismo (subproblemas tratados en diferentes procesadores)
- Uso eficiente de jerarquías de memoria (cache)

Máximo de una lista a de n números

1. Tamaño: n
2. Caso base:
   - Si $n = 1$ se devuelve a_0
3. Descomposición:
   - $x =$ máximo de la primera mitad de **a**
   - $y =$ máximo de la segunda mitad de **a**
4. Combinación:
   - máx$\{x, y\}$

```
1 def maximo(a):  # Ya visto en el Tema 4
2     n = len(a)
3     if n==1:
4         return a[0]
5     else:
6         mitad = n//2
7         return max(maximo(a[:mitad]), maximo(a[mitad:]))
```

Máximo de una lista a de n números

- ¿Podríamos conseguir un algoritmo más eficiente que una búsqueda lineal, que tarda $\mathcal{O}(n)$?
  - Claramente, no
  - Sea cual sea nuestra estrategia tenemos que procesar todos los elementos de la lista
  - Este problema tiene una cota inferior $\Omega(n)$

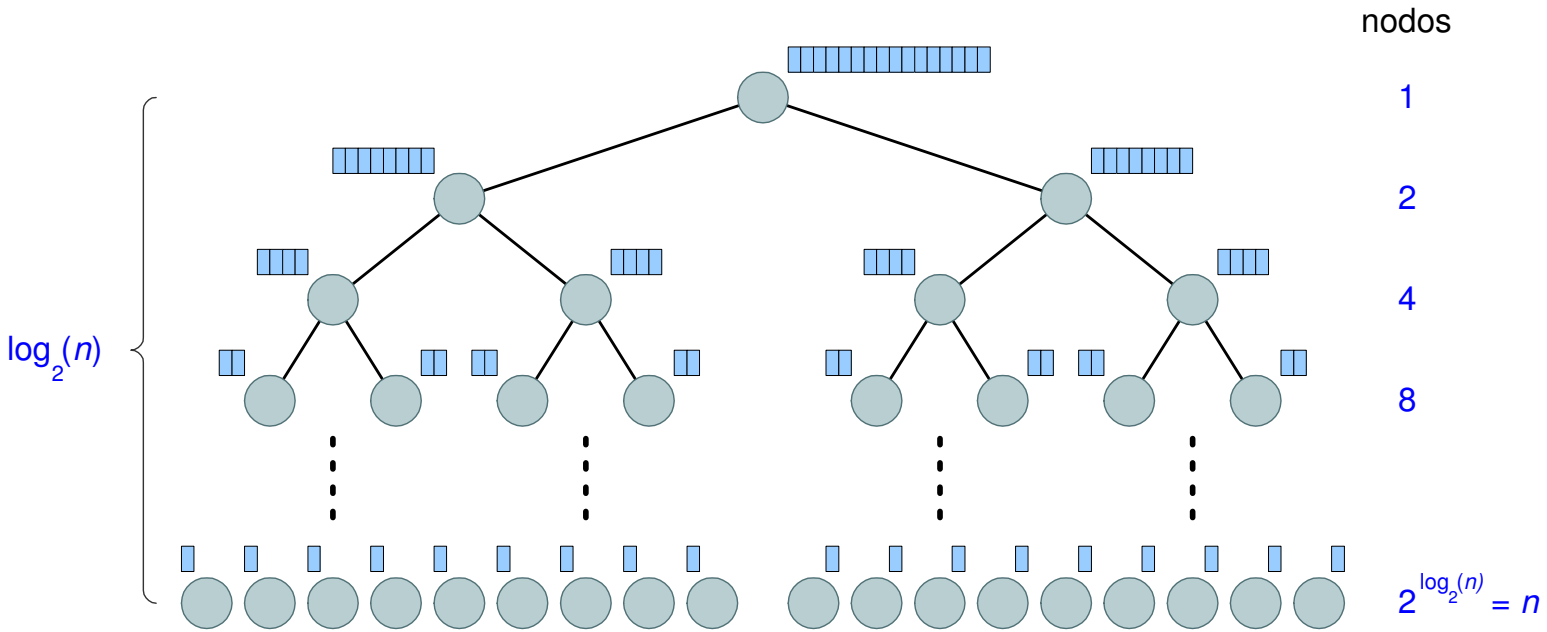
$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 1 + 2T(n/2) & \text{si } n > 1 \end{cases}$$

$$T(n) = 2n - 1 \in \Theta(n)$$

  - Coste lineal a pesar de dividir el problema por dos, ya que tenemos que resolver dos problemas

Máximo de una lista a de n números

- El árbol de recursión tiene del orden de 2^h nodos ($2 \cdot 2^h - 1$ si es completo), donde h es la altura del árbol, pero en este problema $h = \log_2(n)$
  - La función exponencial se cancela con la logarítmica
  - El árbol tiene $2n - 1$ nodos, y cada uno requiere $\Theta(1)$ operaciones
  - $T(n) \in \Theta(n)$



Problema de ordenación

- Insert-sort
- Select-sort
- Merge-sort (divide y vencerás)
- Quicksort (divide y vencerás)
  - Quickselect (algoritmo de búsqueda - no de ordenación)

Problema de ordenación

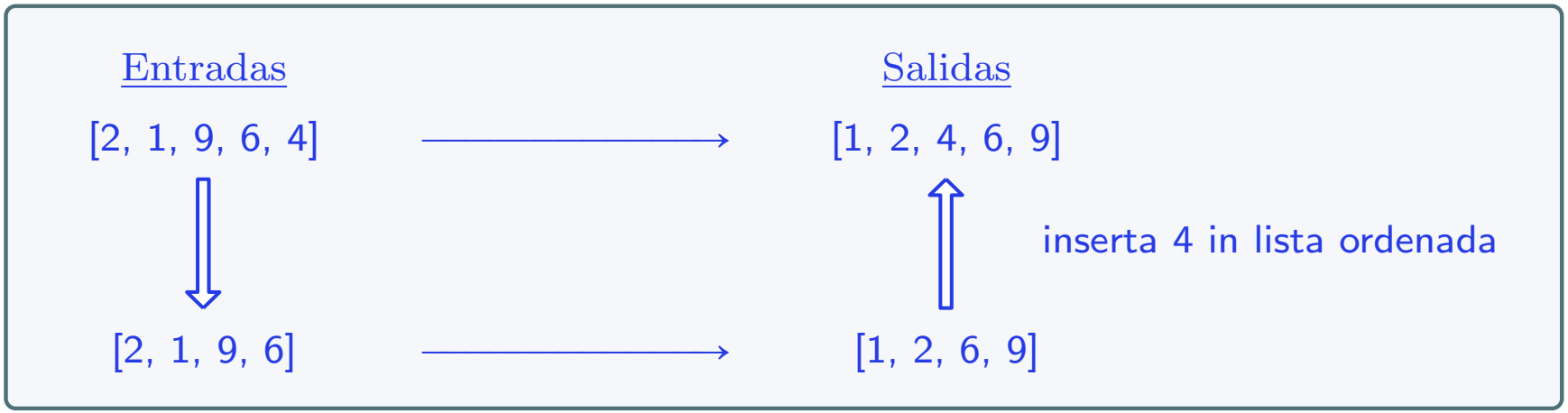
Problema de ordenación

- Entradas: una lista **a** de n números $[a_0, a_1, \ldots, a_{n-1}]$
- Salidas: Una permutación (reordenación) $[a'_0, a'_1, \ldots, a'_{n-1}]$ de los elementos de **a** tal que $a'_1 \leqslant a'_2 \leqslant \cdots \leqslant a'_n$

- Tamaño del problema: n
- Casos base:
  - Si $n \leqslant 1$ se devuelve la lista de entrada
    - n podría ser 0, en cuyo caso se devolvería una lista vacía
- La descomposición que escojamos determinará el algoritmo a desarrollar

Insert-sort

- Descomposición: Eliminar el último elemento de la lista de entrada

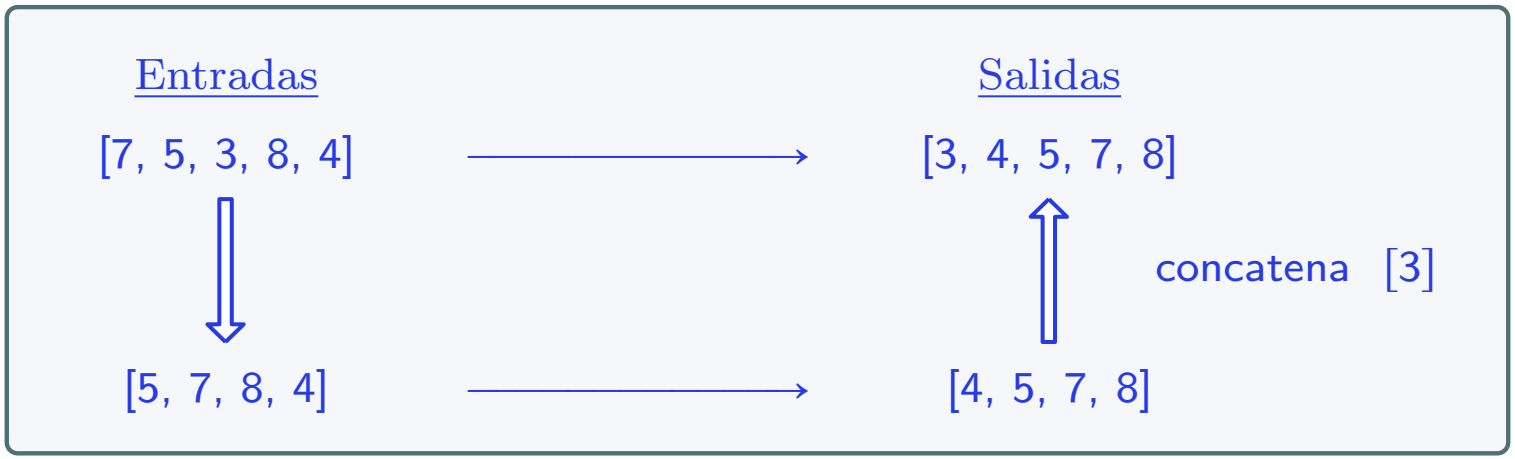



```
1 def insert_sort_rec(a):
2     if len(a) <= 1:
3         return a
4     else:
5         return inserta_en_lista_ordenada(
6             insert_sort_rec(a[:-1]), a[-1])
```

- Ejercicio para casa: `inserta_en_lista_ordenada()`

Select-sort

- Descomposición: Eliminar el menor elemento de la lista de entrada




```
1 def select_sort_rec(a):
2     if len(a) <= 1:
3         return a
4     else:
5         m = min(a)
6         a.remove(m)
7         return [m] + select_sort_rec(a)
```

Select-sort

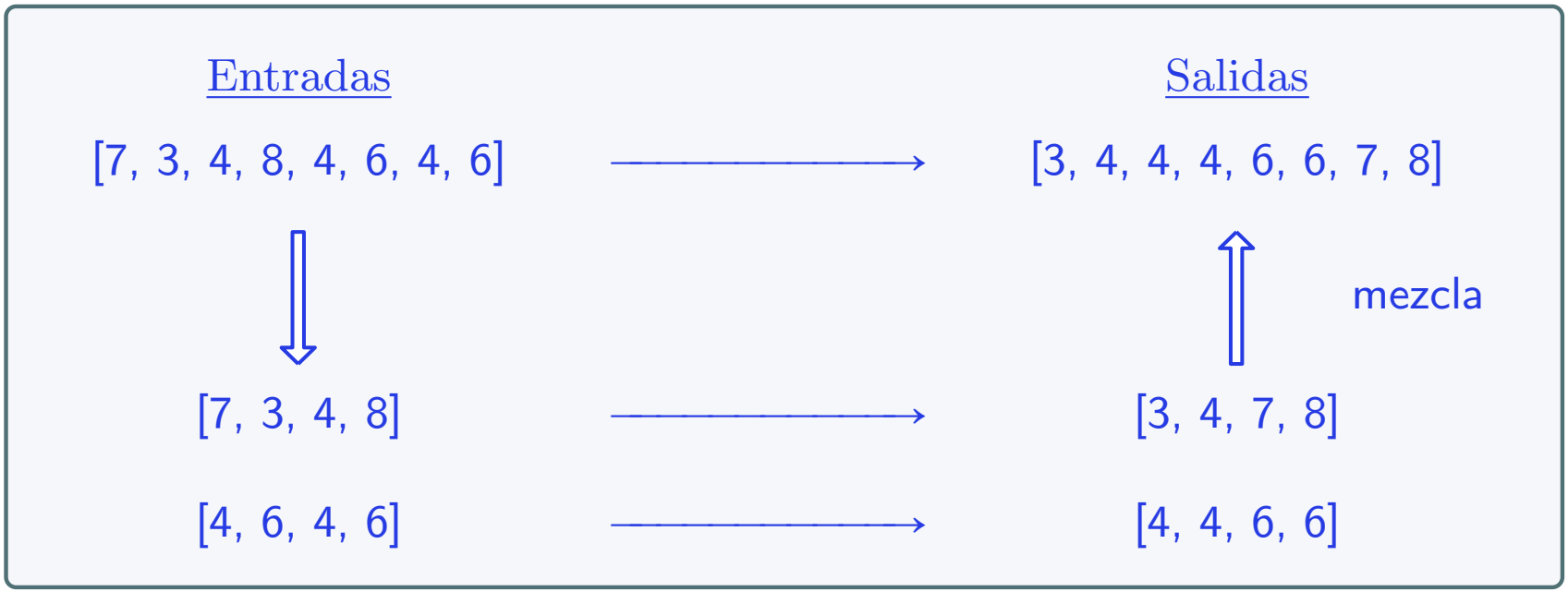
- Alternativas que no modifican la lista de entrada

```
1  def select_sort_rec(a):
2      if len(a) <= 1:
3          return a
4      else:
5          b = list(a)
6          min_index = b.index(min(b))
7          aux = b[min_index]
8          b[min_index] = b[0]
9          b[0] = aux
10         return [aux] + select_sort_rec(b[1:])
```

```
1  def select_sort_rec(a):
2      if len(a) <= 1:
3          return a
4      else:
5          b = list(a)
6          m = min(b)
7          b.remove(m)
8          return [m] + select_sort_rec(b)
```

Merge-sort

- Descomposición: Dividir la lista de entrada en dos mitades



Merge-sort

```
1  def merge_sort(a):  # Versión out-of-place
2      n = len(a)
3      if n <= 1:
4          return a
5      else:
6          mitad = n//2
7          a1 = merge_sort(a[:mitad])
8          a2 = merge_sort(a[mitad:])
9          return mezcla(a1, a2)
10
11 # Versión recursiva sencilla. Es preferible una versión iterativa
12 # Las listas a and b están ordenadas de menor a mayor
13 def mezcla(a, b):
14     if a == []:
15         return b
16     elif b == []:
17         return a
18     else:
19         if a[0] < b[0]:
20             return [a[0]] + mezcla(a[1:], b)
21         else:
22             return [b[0]] + mezcla(a, b[1:])
```

Merge-sort – Versión in-place más eficiente

```
1  # Versión in-place, más eficiente
2  def merge_sort_limites(a,ini,fin):
3      if ini<fin:
4          mitad = (ini+fin)//2
5          merge_sort_limites(a,ini,mitad)
6          merge_sort_limites(a,mitad+1,fin)
7          a[ini:fin+1] = mezcla_limites(a,ini,mitad,mitad+1,fin)
8
9
10 def mezcla_limites(a,aI,aF,bI,bF):
11     if aI>aF:
12         return a[bI:bF+1]
13     elif bI>bF:
14         return a[aI:aF+1]
15     else:
16         if a[aI]<a[bI]:
17             return [a[aI]] + mezcla_limites(a,aI+1,aF,bI,bF)
18         else:
19             return [a[bI]] + mezcla_limites(a,aI,aF,bI+1,bF)
```

Merge-sort – Coste computacional

- Coste en tiempo:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ n + 2T(n/2) & \text{si } n > 1 \end{cases} = n\log_2(n) + n \in \Theta(n\log(n))$$

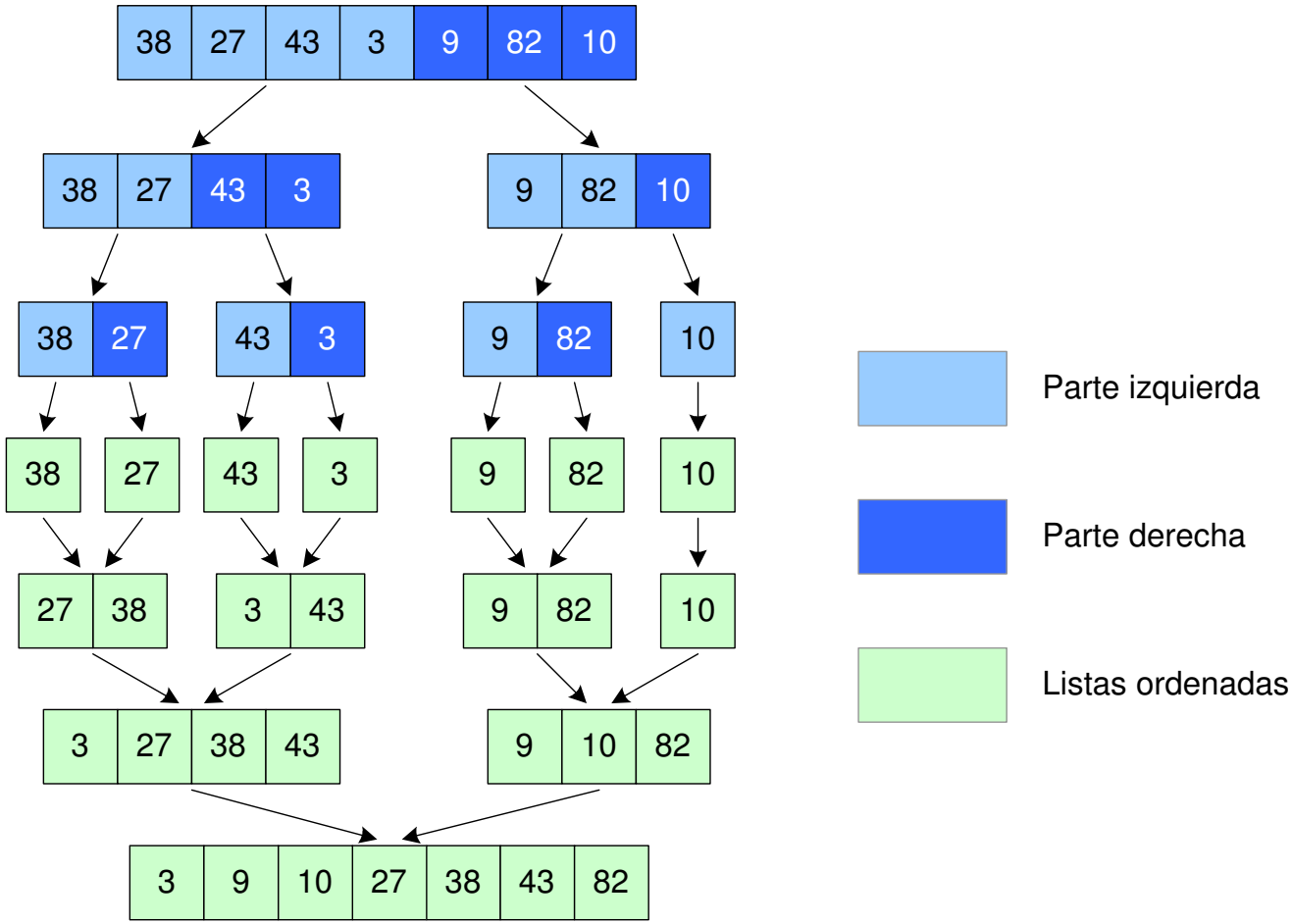
  - $\Theta(n \log n)$ en todos los casos: mejor, peor, y medio

- ¿Cuál es el tiempo si no se divide el vector por la mitad, sino según un $10\,\%$ y $90\,\%$?
  - También $\Theta(n \log n)$

- Memoria auxiliar
  - "*In-place*": la ordenación se realiza en el propio vector
  - "*Out-of-place*": se requiere un vector auxiliar de tamaño n. Es decir, la ordenación no se realiza en el propio vector

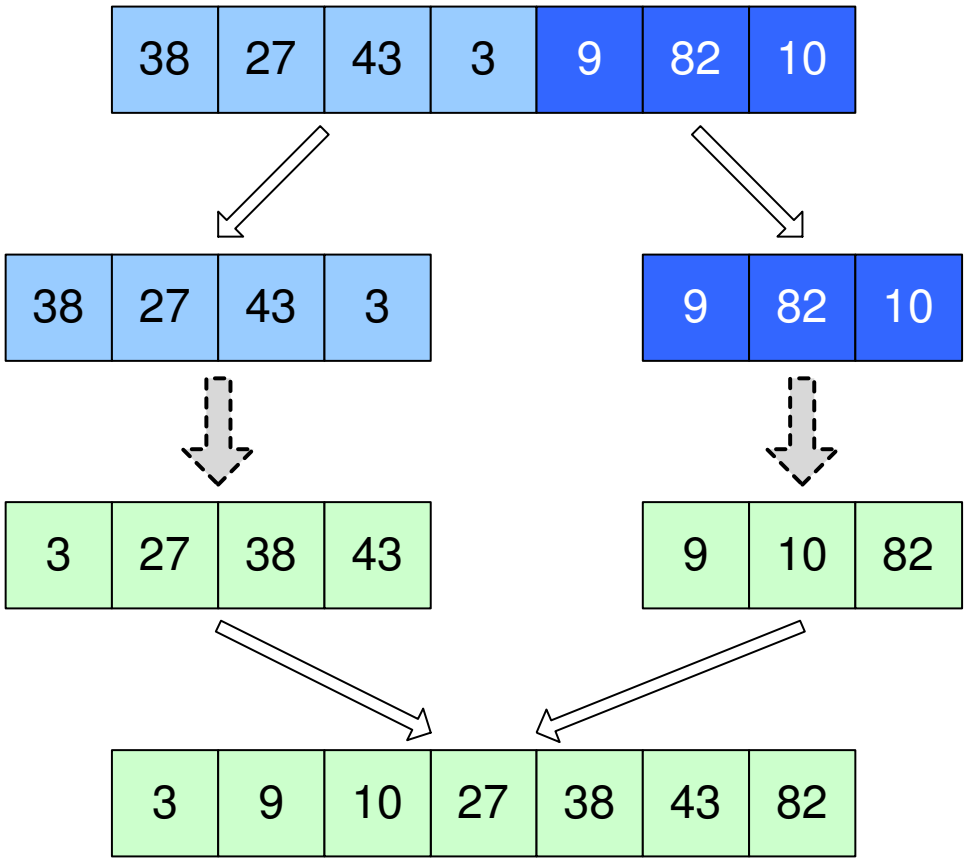
Merge-sort

- Paso a paso



Merge-sort

- Para diseñar el código, es mejor pensar en este esquema



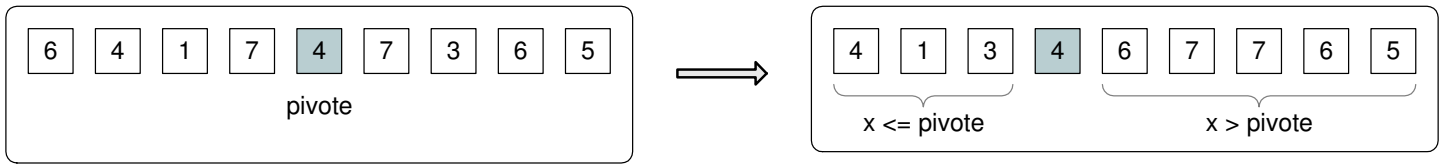
Descomponer el problema

Obtener soluciones de subproblemas (automáticamente) a través de llamadas recursivas

Combinar soluciones

Quicksort

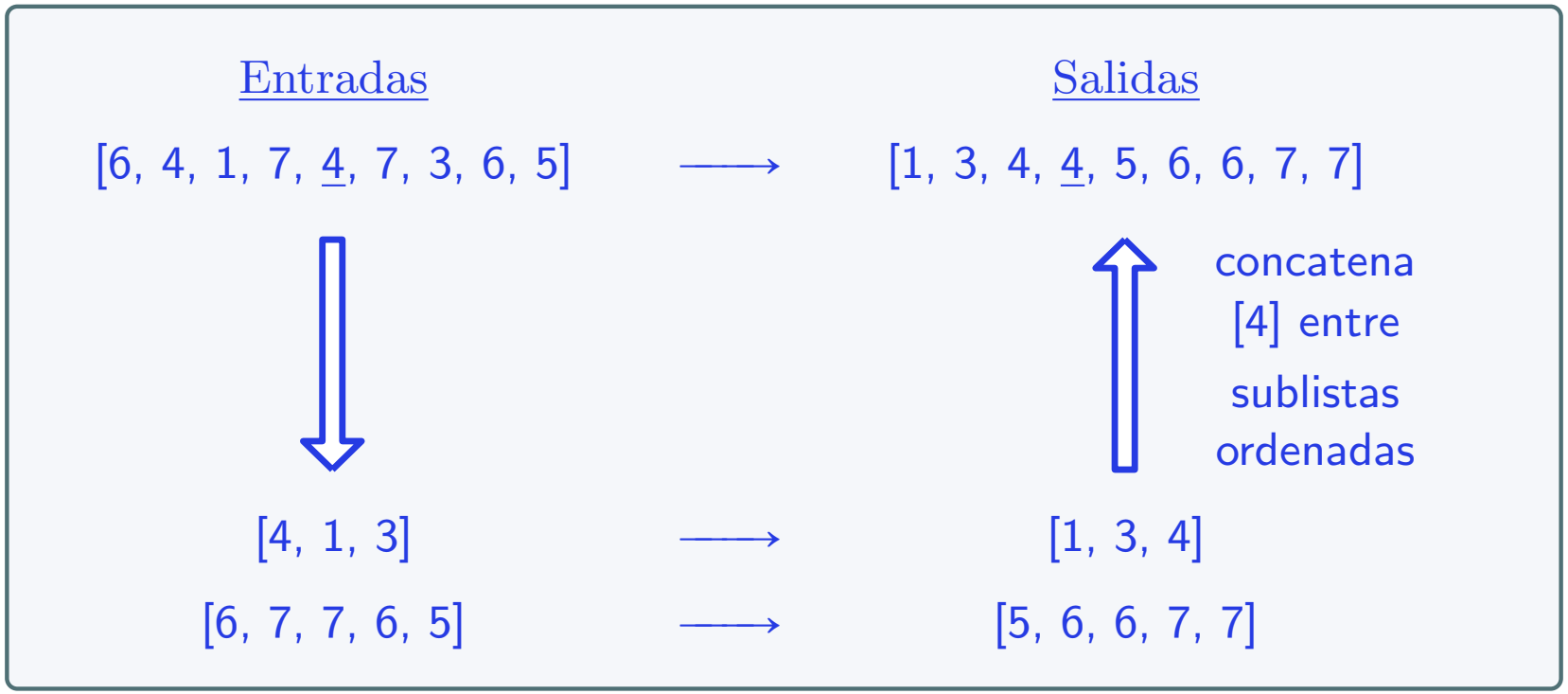
- Descomposición
  - Se escoge un elemento "pivote"
  - Se divide la lista en tres partes:
    - Parte izquierda: elementos menores o iguales que el pivote
    - El propio pivote
    - Parte derecha: elementos mayores que el pivote



  - Posteriormente se ordenan las partes izquierda y derecha
- Combinación
  - Trivial: la lista ya está ordenada, no hay que hacer nada más

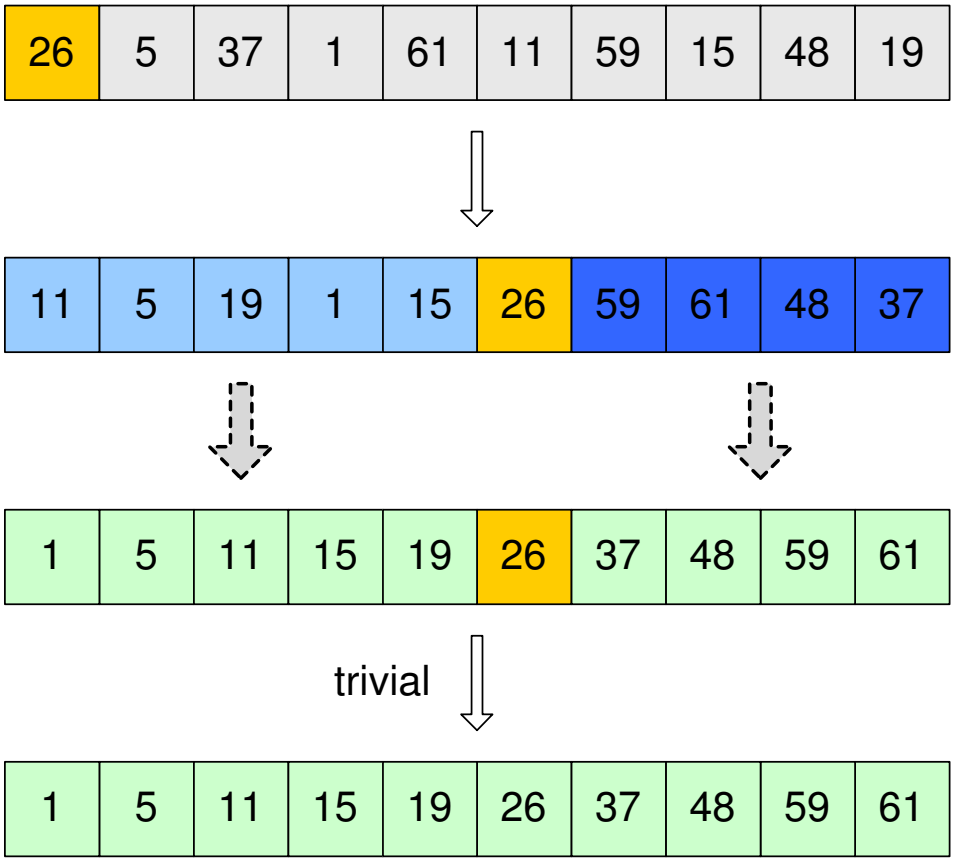
Quicksort

- Descomposición: Elementos menores o iguales, y mayores, que un pivote (el primero, el central, uno al azar...)



Quicksort

- Diagrama alternativo:



Descomponer el problema

Obtener soluciones de subproblemas (automáticamente) a través de llamadas recursivas

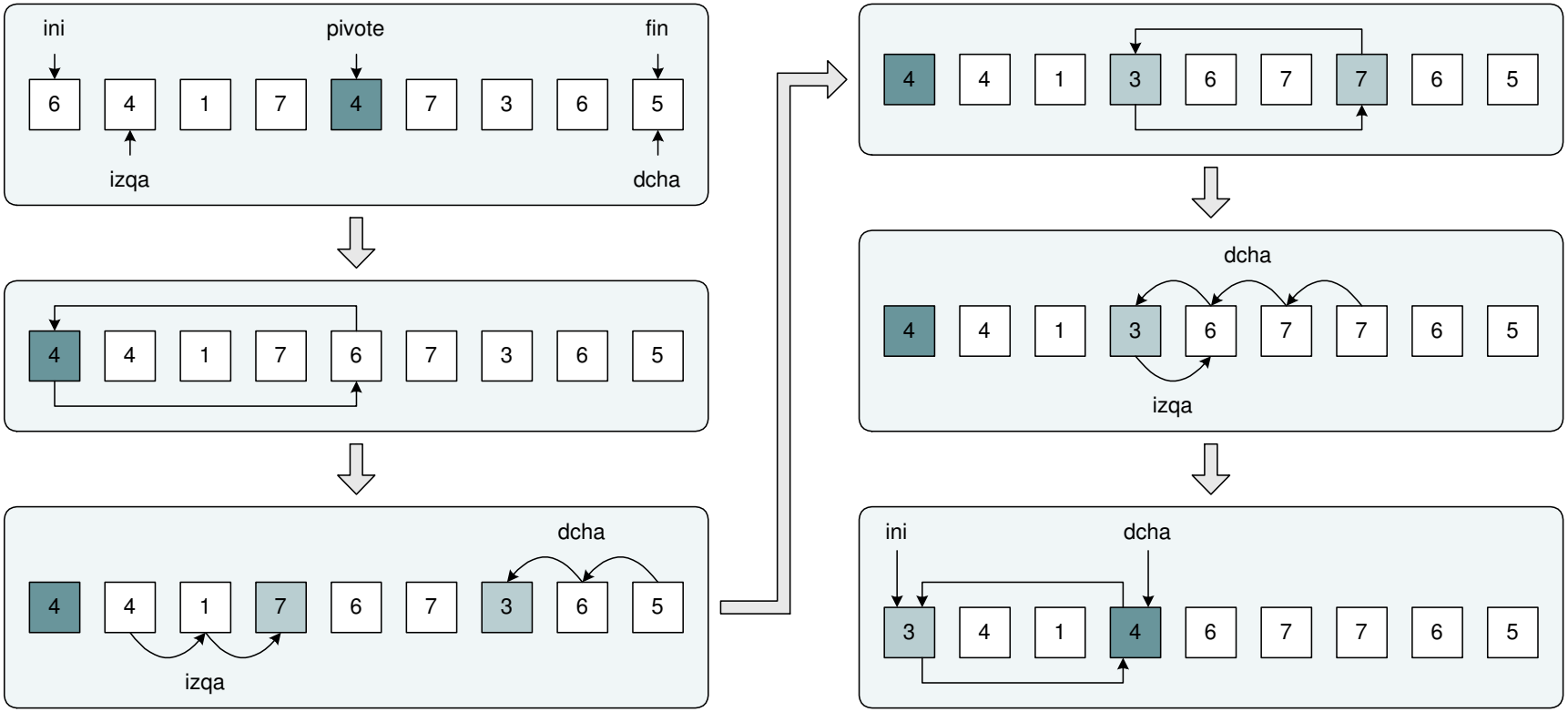
Combinar soluciones

Quicksort

```
1 def quick_sort_inplace(a, ini, fin):
2     if ini < fin:
3         indice_pivote = particion_Hoare(a, ini, fin)
4 
5         quick_sort_inplace(a, ini, indice_pivote - 1)
6         quick_sort_inplace(a, indice_pivote + 1, fin)
```

- Partición de Hoare: algoritmo iterativo para realizar la descomposición
  - Se usa en otros algoritmos (quickselect)
- Después de las llamadas recursivas la lista ya queda ordenada

Partición de Hoare – Idea



Partición de Hoare – Código

```
1  def particion_Hoare(a,ini,fin):
2      if fin>=0:
3          mitad = (ini+fin)//2
4          pivote = a[mitad]
5          a[mitad] = a[ini]
6          a[ini] = pivote
7
8          izqa  = ini+1
9          dcha = fin
10
11         ha_terminado = False
12         while not ha_terminado:
13
14             while izqa<=fin and a[izqa]<=pivote:
15                 izqa = izqa+1
16
17             while a[dcha]>pivote:
18                 dcha = dcha-1
19
20             if izqa<dcha:
21                 aux = a[izqa];
22                 a[izqa] = a[dcha];
23                 a[dcha] = aux;
24
25             ha_terminado = izqa>dcha
26
27         a[ini] = a[dcha]
28         a[dcha] = pivote
29
30         return dcha
```

Quicksort

- Caso mejor (el elemento pivote está en la mitad)

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases} \quad \Rightarrow \quad \Theta(n \log n)$$

- Caso peor (un subvector queda siempre vacío)

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ T(1) + T(n-1) + \Theta(n) & \text{si } n > 1 \end{cases} \quad \Rightarrow \quad \Theta(n^2)$$

Quicksort

- Algoritmo de ordenación rápido en la práctica
  - Caso peor: $\Theta(n^2)$
  - Caso mejor: $\Theta(n \log n)$
  - Caso medio: $\Theta(n \log n)$
- Suele superar a otros algoritmos $\Theta(n \log n)$ al utilizar mejor las jerarquías de memoria
- Memoria auxiliar
  - $\mathcal{O}(n)$
  - Puede ser "out-of-place" o "in-place" (la ordenación no se realiza en el propio vector)

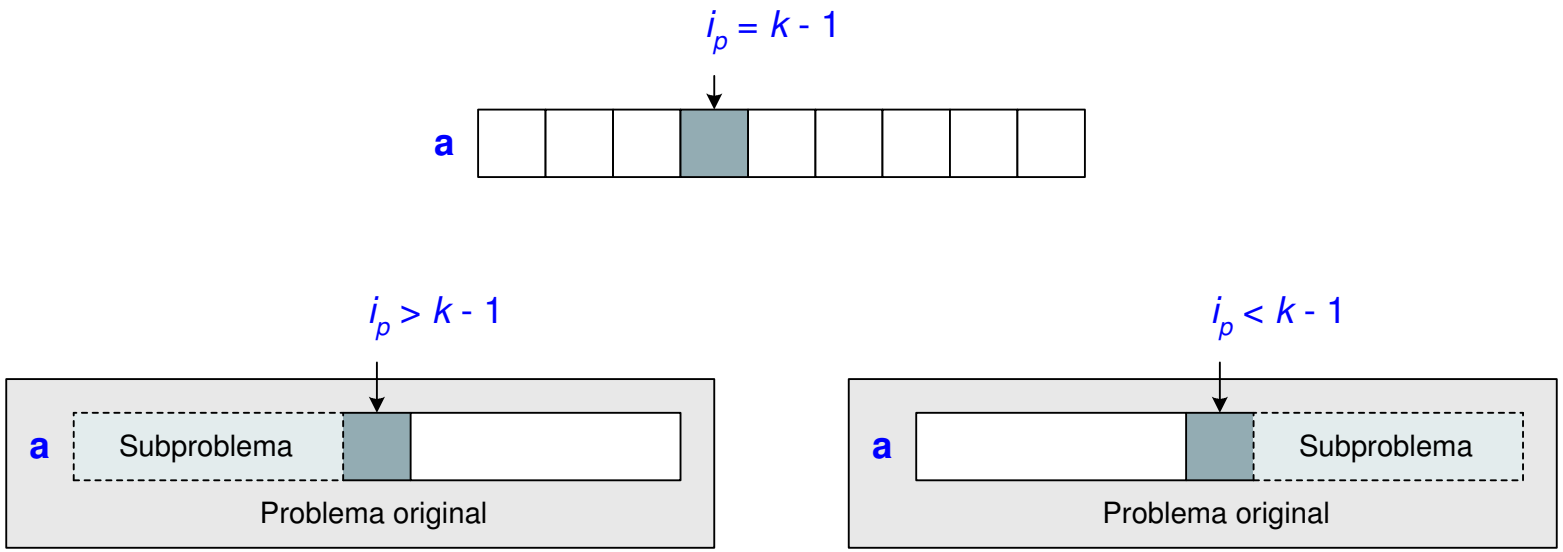
Búsqueda del k-ésimo elemento más pequeño

Búsqueda del k-ésimo elemento más pequeño

Dada una lista **a** de n números, encontrar el k-ésimo elemento más pequeño.
Ejemplo: Para $a = [4, 3, 7, 7, 2, 5, 1, 8, 9, 2, 1]$, y $k = 5$, el algoritmo devuelve 3, ya que es el $5^{\underline{o}}$ elemento de la lista ordenada $[1, 1, 2, 2, 3, 4, 5, 7, 7, 8, 9]$.

- Caso general de la búsqueda de la mediana
- Veremos el algoritmo **Quickselect**
  - Basado en la partición de Hoare
  - Solo realiza una llamada recursiva
  - Estrictamente, no es un algoritmo de divide y vencerás

Búsqueda del k-ésimo elemento más pequeño



- Aplica la partición de Hoare, ubicando el pivote donde quedaría en caso de ordenar la lista
- Si el pivote está en el índice $k - 1$ se ha encontrado el k-ésimo elemento más pequeño
- De lo contrario se sigue buscando, pero solo en la mitad donde se puede encontrar el elemento buscado

Quickselect

```
1  def quickselect(a,ini,fin,k):
2      if ini==fin:
3          return a[ini]
4      else:
5          indice_pivote = particion_Hoare(a, ini, fin)
6
7          if indice_pivote==k-1:
8              return a[indice_pivote]
9          elif indice_pivote<k-1:
10             return quickselect(a,indice_pivote+1,fin,k)
11         else:
12             return quickselect(a,ini,indice_pivote-1,k)
```

- Coste medio: $\mathcal{O}(n)$
- Coste en el peor caso: $\mathcal{O}(n^2)$

Multiplicación

- Multiplicación de números naturales en binario
  - Algoritmo de Karatsuba
- Multiplicación de matrices
  - Descomposiciones recursivas
  - Algoritmo de Strassen

Multiplicación rápida de números en binario

Multiplicación de números en binario

Dados dos números enteros no negativos x e y, expresados en binario mediante b_x y b_y bits, respectivamente, hallar el producto xy.

- Tamaño del problema: $\text{mín}(x, y)$
- Casos base:
  - Si $x = 0$ o $y = 0$ (es decir, si $\text{mín}(x, y) = 0$) el resultado es 0
  - Si $x = 1$ el resultado es y
  - Si $y = 1$ el resultado es x

Multiplicación rápida de números en binario

- Descomposición:

$$x = a \cdot 2^m + b$$
$$y = c \cdot 2^m + d$$

$$m = \text{mín}\left(\left\lfloor \frac{b_x}{2} \right\rfloor, \left\lfloor \frac{b_y}{2} \right\rfloor\right)$$

- Ejemplo. Para $x = 594$ e $y = 69$:

$$x = 1001010010_2 = \underbrace{1001010}_{a=74}\ \underbrace{010}_{b=2} = 74 \cdot 2^3 + 2$$

$$y = 1000101_2 = \underbrace{1000}_{c=8}\ \underbrace{101}_{d=5} = 8 \cdot 2^3 + 5$$

  - $b_x = 10$, $b_y = 7$, $m = 3$, $a = 74$, $b = 2$, $c = 8$, y $d = 5$

Multiplicación rápida de números en binario

- Regla recursiva básica (lenta):

$$xy = (a \cdot 2^m + b)(c \cdot 2^m + d) = ac2^{2m} + (ad + bc)2^m + bd$$

  - 4 llamadas recursivas (4 "subproductos")
  - Si x e y tienen n bits el coste en tiempo es:

$$T(n) = \begin{cases} 1 & \text{si } n \leqslant 1 \\ 4\,T(n/2) + en + f & \text{si } n > 1 \end{cases}$$

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

Multiplicación rápida de números en binario

- Regla recursiva rápida (algoritmo de Karatsuba):

$$xy = ac2^{2m} + [(a+b)(c+d) - ac - bd]\,2^m + bd$$

  - 3 llamadas recursivas (3 "subproductos")
  - Si x e y tienen n bits el coste en tiempo es:

$$T(n) = \begin{cases} 1 & \text{si } n \leqslant 1 \\ 3T(n/2) + en + f & \text{si } n > 1 \end{cases}$$

$$T(n) \in \Theta(n^{log_2 3}) = \Theta(n^{1,585\ldots})$$

Multiplicación rápida de números en binario

```
1  def numero_de_bits(n):
2      if n < 2:
3          return 1
4      else:
5          return 1 + numero_de_bits(n >> 1)
6
7  def multiplicacion_karatsuba(x, y):
8      if x == 0 or y == 0:
9          return 0
10     elif x == 1:
11         return y
12     elif y == 1:
13         return x
14     else:
15         n_bits_x = numero_de_bits(x)
16         n_bits_y = numero_de_bits(y)
17
18         m = min(n_bits_x // 2, n_bits_y // 2)
19
20         a = x >> m              # << y >> son operaciones de desplazamiento de bits
21         b = x - (a << m)        # (z << m) equivale a z*(2**m)
22         c = y >> m              # (z >> m) equivale a z//(2**m)
23         d = y - (c << m)
24
25         ac = multiplicacion_karatsuba(a, c)
26         bd = multiplicacion_karatsuba(b, d)
27         t = multiplicacion_karatsuba(a + b, c + d) - ac - bd
28
29         return (ac << (2 * m)) + (t << m) + bd
```

Multiplicación de matrices

- La división de una matriz en bloques tiene varias ventajas:
  - Simplifica operaciones algebraicas
  - Se puede aprovechar para construir algoritmos eficientes que hagan buen uso de la memoria caché
- Para realizar una multiplicación hay varias formas de dividir la matriz

$$A \cdot B = \left(\begin{array}{c} A_1 \\ \hline A_2 \end{array}\right) \cdot \left(\begin{array}{c|c} B_1 & B_2 \end{array}\right) = \begin{pmatrix} A_1 B_1 & A_1 B_2 \\ A_2 B_1 & A_2 B_2 \end{pmatrix}$$

$$A \cdot B = \left(\begin{array}{c|c} A_1 & A_2 \end{array}\right) \cdot \left(\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right) = \left(\begin{array}{c} A_1 B_1 + A_2 B_2 \end{array}\right)$$

$$A \cdot B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

Multiplicación de matrices - complejidad

- Sea A una matriz de $p \times q$, y B una matriz de $q \times r$
- El producto $C = AB$ requiere $\mathcal{O}(pqr)$ operaciones
- Considérese que las matrices son cuadradas de dimensión $n \times n$, y la siguiente descomposición:

$$A \cdot B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 8T(n/2) + 4\Theta(n^2) & \text{si } n > 1 \end{cases}$$

  - 8 multiplicaciones (de matrices de $n/2 \times n/2$)
  - 4 sumas (de matrices de $n/2 \times n/2$)
  - Por el teorema maestro: $T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$

Multiplicación de matrices – Librería numpy

```
1  import numpy as np
2
3  def mult_mat(A, B):
4      p = A.shape[0]; q = A.shape[1]; r = B.shape[1]
5
6      if p == 0 or q == 0 or r == 0:
7          return np.zeros((p, r))
8      elif p == 1 and q == 1 and r == 1:
9          return np.matrix([[A[0, 0] * B[0, 0]]])
10     else:
11         A11 = A[0:p // 2, 0:q // 2]; A21 = A[p // 2:p, 0:q // 2]
12         A12 = A[0:p // 2, q // 2:q]; A22 = A[p // 2:p, q // 2:q]
13
14         B11 = B[0:q // 2, 0:r // 2]; B21 = B[q // 2:q, 0:r // 2]
15         B12 = B[0:q // 2, r // 2:r]; B22 = B[q // 2:q, r // 2:r]
16
17         C11 = mult_mat(A11, B11) + mult_mat(A12, B21)
18         C12 = mult_mat(A11, B12) + mult_mat(A12, B22)
19         C21 = mult_mat(A21, B11) + mult_mat(A22, B21)
20         C22 = mult_mat(A21, B12) + mult_mat(A22, B22)
21
22         return np.vstack([np.hstack([C11, C12]),
23                           np.hstack([C21, C22])])
24
25 A = np.matrix([[2, 3, 1, -3], [4, -2, 1, 2]])
26 B = np.matrix([[2, 3, 1], [4, -1, -5], [0, -6, 3], [1, -1, 1]])
27 print(mult_mat(A, B))
```

Multiplicación de matrices - algoritmo de Strassen

- La operación básica más costosa es la multiplicación
- El algoritmo de Strassen consigue reducir el número de multiplicaciones
- Considérese la siguiente descomposición:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22})B_{11}$$
$$M_3 = A_{11}(B_{12} - B_{22})$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12})B_{22}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 - M_2 + M_3 + M_6$$

Multiplicación de matrices - algoritmo de Strassen

- Con esa factorización el algoritmo requiere
  - 7 multiplicaciones
  - 18 sumas

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 7T(n/2) + 18\Theta(n^2) & \text{si } n > 1 \end{cases}$$

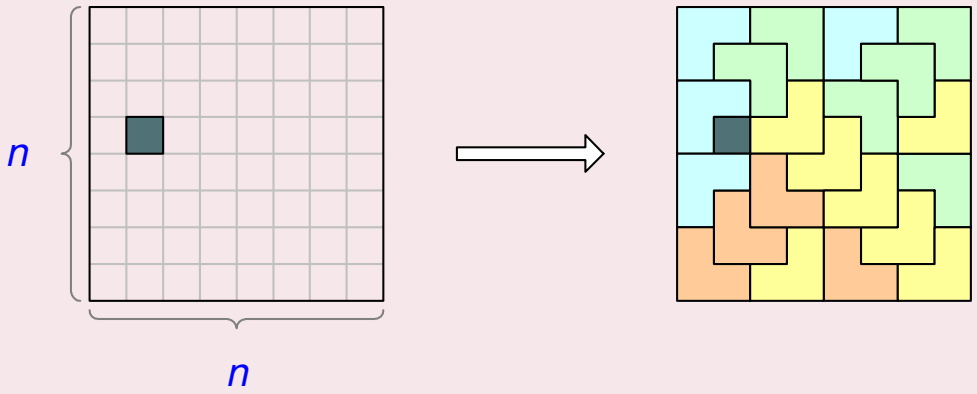
  - Por el teorema maestro: $T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2,807})$

- Solo es más eficiente que una multiplicación ordinaria si el tamaño de las matrices es muy elevado
- Actualmente hay un algoritmos en $\mathcal{O}(n^{2,376})$ (Coppersmith - Winograd)
- En la práctica la mejor estrategia para acelerar el producto de dos matrices consiste en aprovechar la estructura jerárquica de la memoria

Embaldosado con L-triominós

Embaldosado con L-triominós

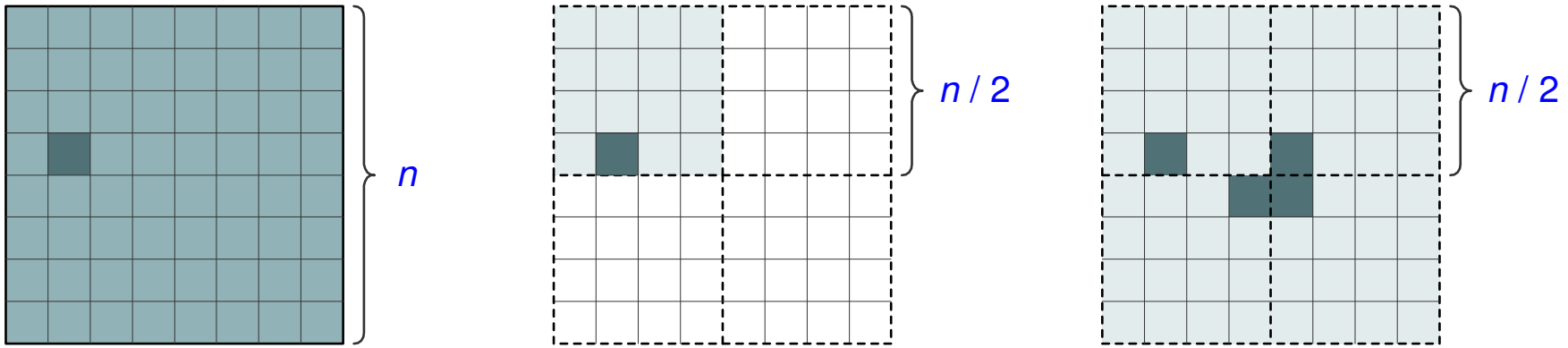
Se desea cubrir una cuadrícula de dimensiones $n \times n$, donde $n = 2^k$ con $k \geqslant 1$ es una potencia de dos, con L-triominós, donde solo un cuadrado no quede cubierto.



- Casos base:

Embaldosado con L-trominós

- Descomposición y casos recursivos:



- Ubicar un trominó en el centro
- Realizar 4 llamadas recursivas con cuadrículas de lado $n/2$
- Códigos 6.11, 6.12 y 6.13 en el libro

Permutación de inversión de bits

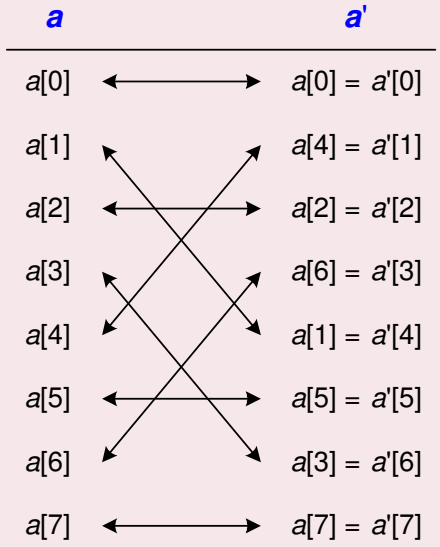
Permutación de inversión de bits

Dada una lista **a** de 2^p números ($p \in \mathbb{N}^+$), devuelva otra lista $\mathbf{a}'$ que sea una permutación concreta de los elementos de **a**. En particular, $a'_j = a_i$ (y $a'_i = a_j$) donde j es el número que resulta de invertir los bits de i, asumiendo que i y j son numeros binarios de p bits.

relación entre i y j, para $p = 3$

i	b_2	b_1	b_0		b_0	b_1	b_2	j
0	0	0	0		0	0	0	0
1	0	0	1		1	0	0	4
2	0	1	0		0	1	0	2
3	0	1	1	⟺	1	1	0	6
4	1	0	0		0	0	1	1
5	1	0	1		1	0	1	5
6	1	1	0		0	1	1	3
7	1	1	1		1	1	1	7

permutación (intercambios) para $p = 3$



Permutación de inversión de bits

- Ejemplo 1:
  - Para $p = 5$
  - $i = (b_4 b_3 b_2 b_1 b_0)_2 \quad \Leftrightarrow \quad j = (b_0 b_1 b_2 b_3 b_4)_2$
  - $a_{26} = a'_{11}$ y $a_{11} = a'_{26}$
    - Ya que $26 = 11010_2$ y $11 = 01011_2$
- Ejemplo 2:
  - Para $p = 3$
  - $\mathbf{a} = [3, 5, 7, 2, 4, 8, 1, 6] \quad \Leftrightarrow \quad \mathbf{a}' = [3, 4, 7, 1, 5, 8, 2, 6]$
    - Ya que para $p = 3$ se intercambian los elementos de las posiciones 1 y 4, y los de las posiciones 3 y 6
- Notas:
  - El problema de inversión de bits es parte de implementaciones de la FFT (transformada rápida de Fourier)
  - Hay algoritmos más eficientes que los que se describen a continuación

Permutación de inversión de bits

- Tamaño: p
- Caso base:
  - Si $p = 1$ se devuelve la propia lista **a**
  - $p = 1 \Leftrightarrow n = 2$ (n: longitud de **a**)
- Descomposiciones:
  - Veremos dos que conducen a diferentes algoritmos
  - Veremos una tercera versión "iterativa"
  - Para $p = 2$ $(n = 4)$ se intercambian los dos elementos centrales:

relación entre i y j, para $p = 2$

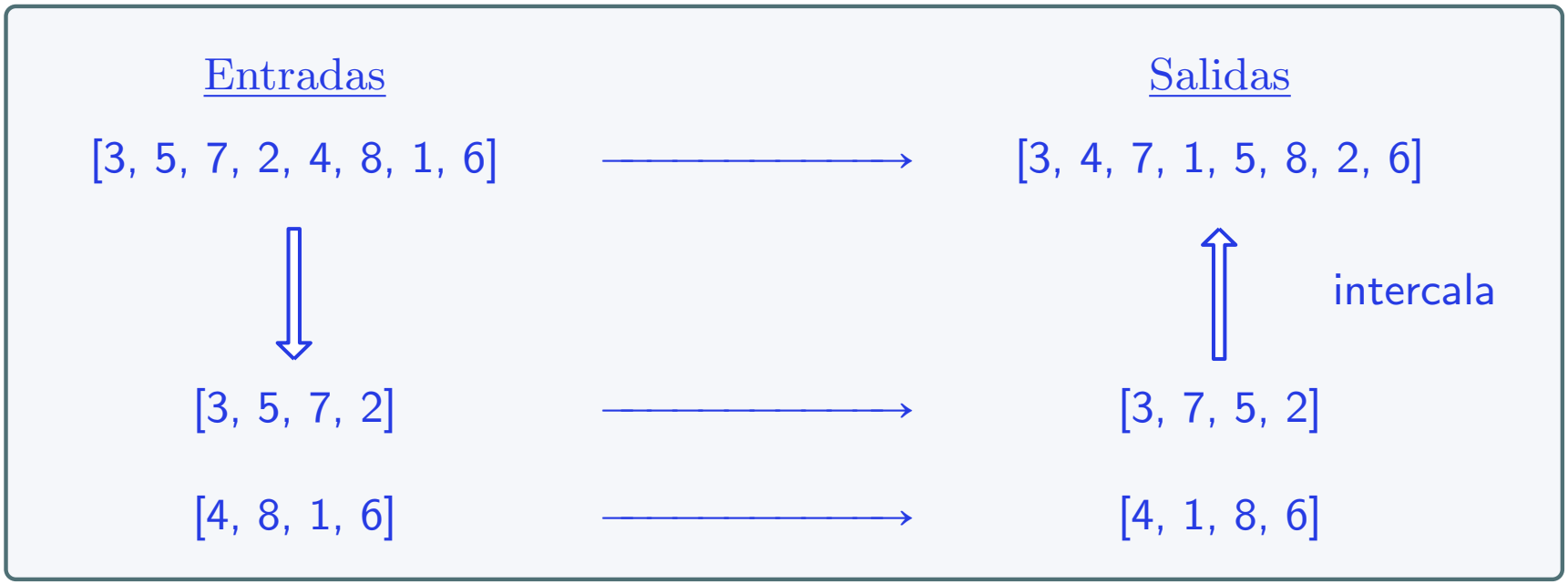
i	b_1	b_0		b_0	b_1	j
0	0	0	⟺	0	0	0
1	0	1		1	0	2
2	1	0		0	1	1
3	1	1		1	1	3

permutación para $p = 2$

a		**a'**
a[0]	⟷	a[0] = a'[0]
a[1]	⟋⟍	a[2] = a'[1]
a[2]	⟍⟋	a[1] = a'[2]
a[3]	⟷	a[3] = a'[3]

Permutación de inversión de bits – Solución I

- Descomposición: Dividir la lista de entrada en dos mitades

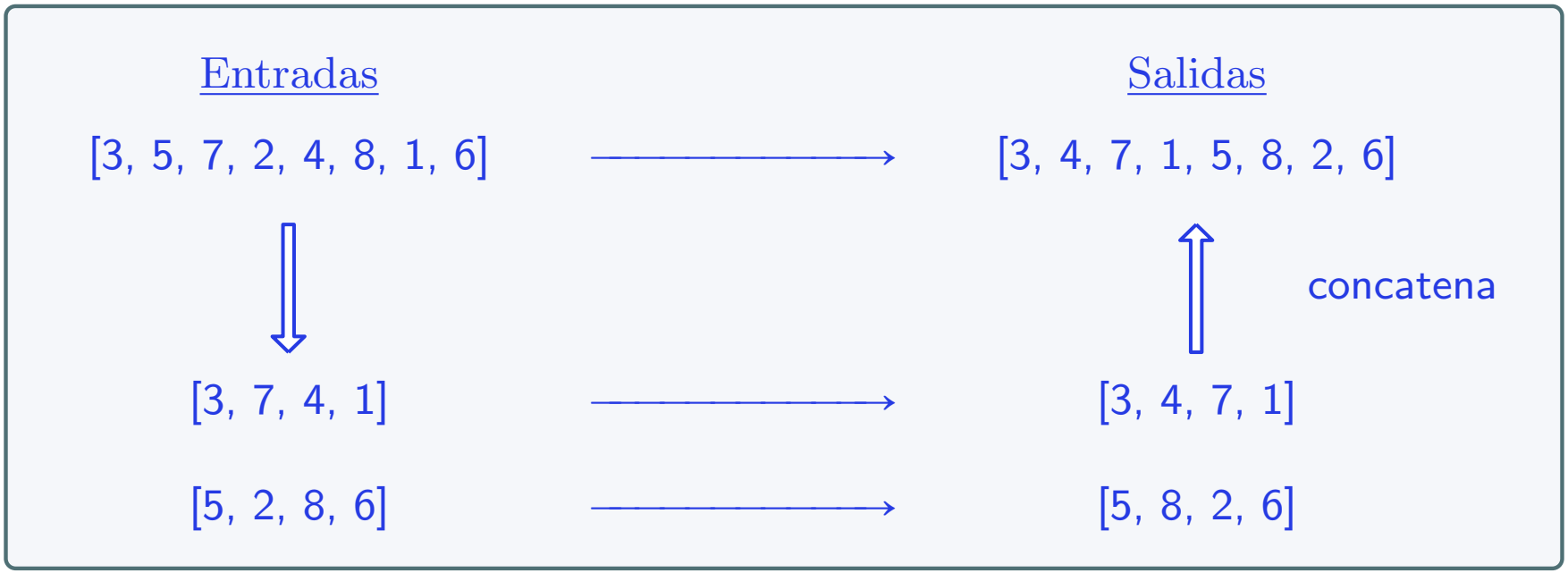


Permutación de inversión de bits – Solución I

```
1  def intercala(a1,a2):
2      if len(a1)==1:
3          return [a1[0],a2[0]]
4      else:
5          return [a1[0]] + [a2[0]] + intercala(a1[1:],a2[1:])
6
7
8  def invbitperm1(a):
9      if len(a)<=2:
10         return a
11     else:
12         mitad = len(a)//2
13         a1 = invbitperm1(a[:mitad])
14         a2 = invbitperm1(a[mitad:])
15         return intercala(a1,a2)
```

Permutación de inversión de bits – Solución II

- Descomposición: Listas de índices pares e impares

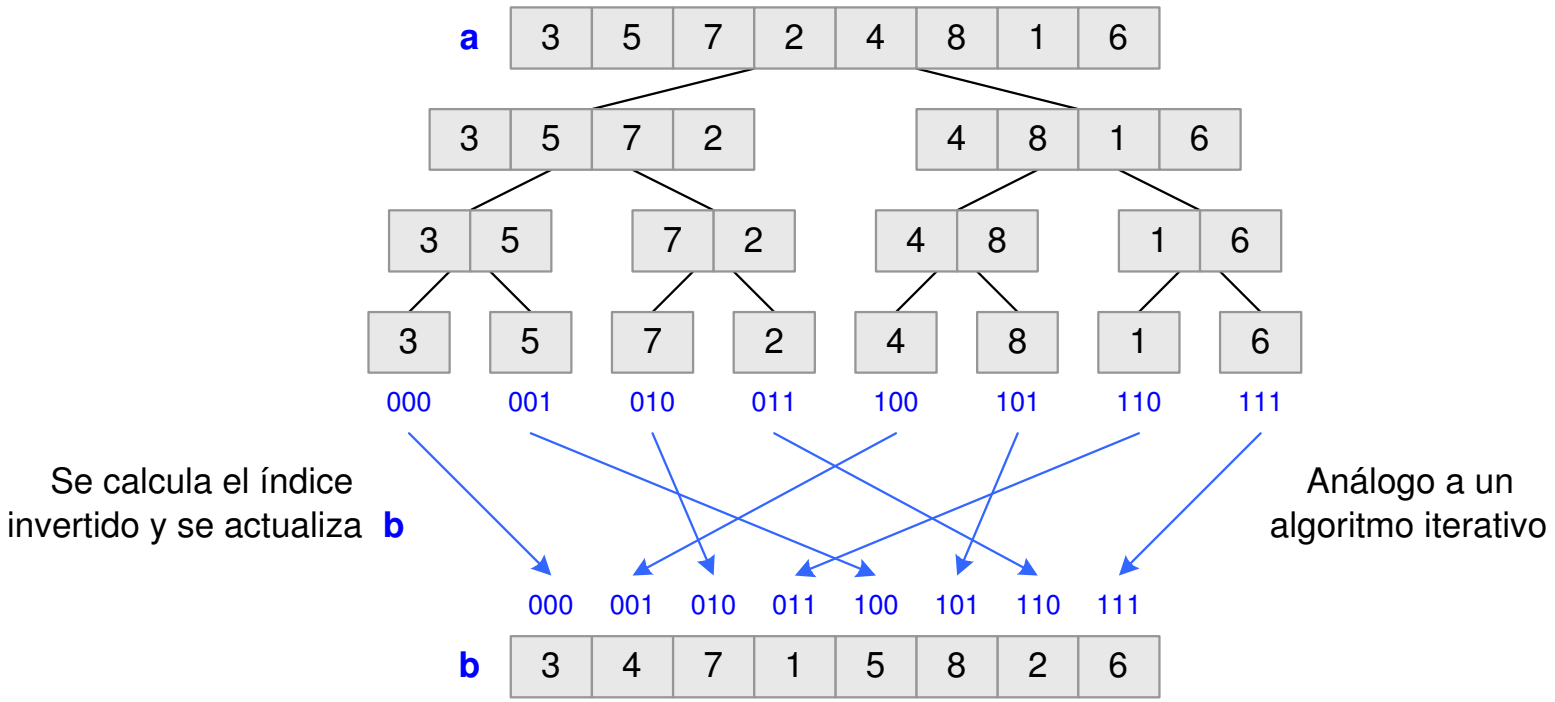


Permutación de inversión de bits – Solución II

```
1  def pares_impares(a):
2      if a==[]:
3          return ([],[])
4      else:
5          (a1,a2) = pares_impares(a[2:])
6          return ([a[0]] + a1, [a[1]] + a2)
7
8
9  def invbitperm2(a):
10     if len(a)<=2:
11         return a
12     else:
13         (a1,a2) = pares_impares(a)
14         return invbitperm2(a1) + invbitperm2(a2)
```

Permutación de inversión de bits – Solución III

- Descomposición: Dividir la lista en dos mitades
- Caso base: Si la lista tiene un elemento, se calcula el índice invertido, y se actualiza una segunda lista que almacene la solución



Permutación de inversión de bits – Solución III

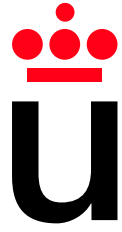
```
1  def indice_invertido(x,p):
2      if p==1:
3          return x
4      else:
5          return indice_invertido(x//2,p-1) + (x%2)*2**(p-1)
6
7
8  def invbitperm3(a,ini,fin,b,p):
9      if ini==fin:
10         i = indice_invertido(ini,p)
11         b[i] = a[ini]
12     else:
13         mitad = (ini+fin)//2
14         invbitperm3(a,ini,mitad,b,p)
15         invbitperm3(a,mitad+1,fin,b,p)
```

Tema 6
Vuelta atrás – Backtracking

Diseño y Análisis de Algoritmos

Manuel Rubio Sánchez

13 de mayo de 2024



Universidad
Rey Juan Carlos

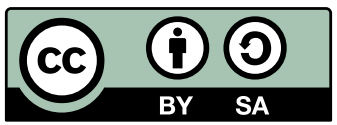
Copyright

©2024 Manuel Rubio Sánchez

Algunos derechos reservados.

Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en:

https://creativecommons.org/licenses/by-sa/4.0/deed.es.



Contenido

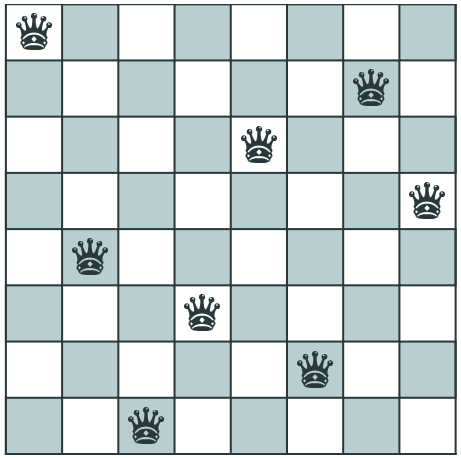
Introducción

Backtracking - Vuelta atrás

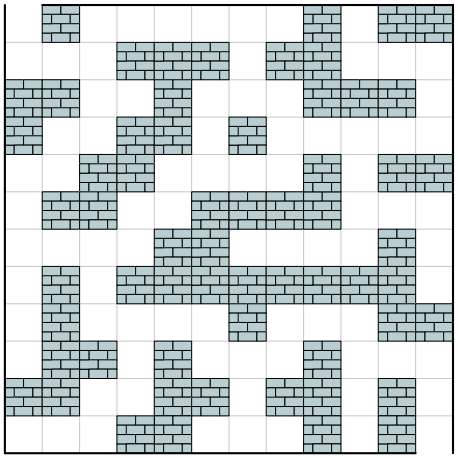
- Estrategia para encontrar soluciones a problemas con restricciones definidos sobre espacios discretos (de elevado tamaño)
- Construye **soluciones parciales** progresivamente, las cuales deben cumplir las restricciones del problema
- El recorrido (en profundidad) tiene éxito si, procediendo de esta forma, se puede definir por completo una solución (en una hoja de un árbol de recursión)
  - Puede detenerse al encontrar una solución o seguir hasta encontrar todas
- Si en alguna etapa la solución parcial construida hasta el momento no se puede completar, se **vuelve atrás** deshaciendo la solución parcial, hasta un punto donde puede seguir explorando posibles soluciones
- Es un método de "fuerza bruta" pero "inteligente"

Ejemplos

N reinas



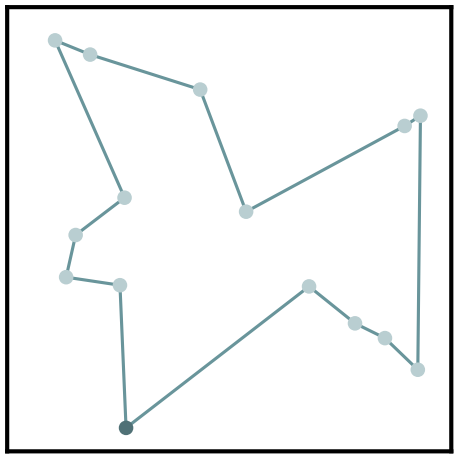
Laberinto



Sudoku

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

Problema del viajante (TSP)

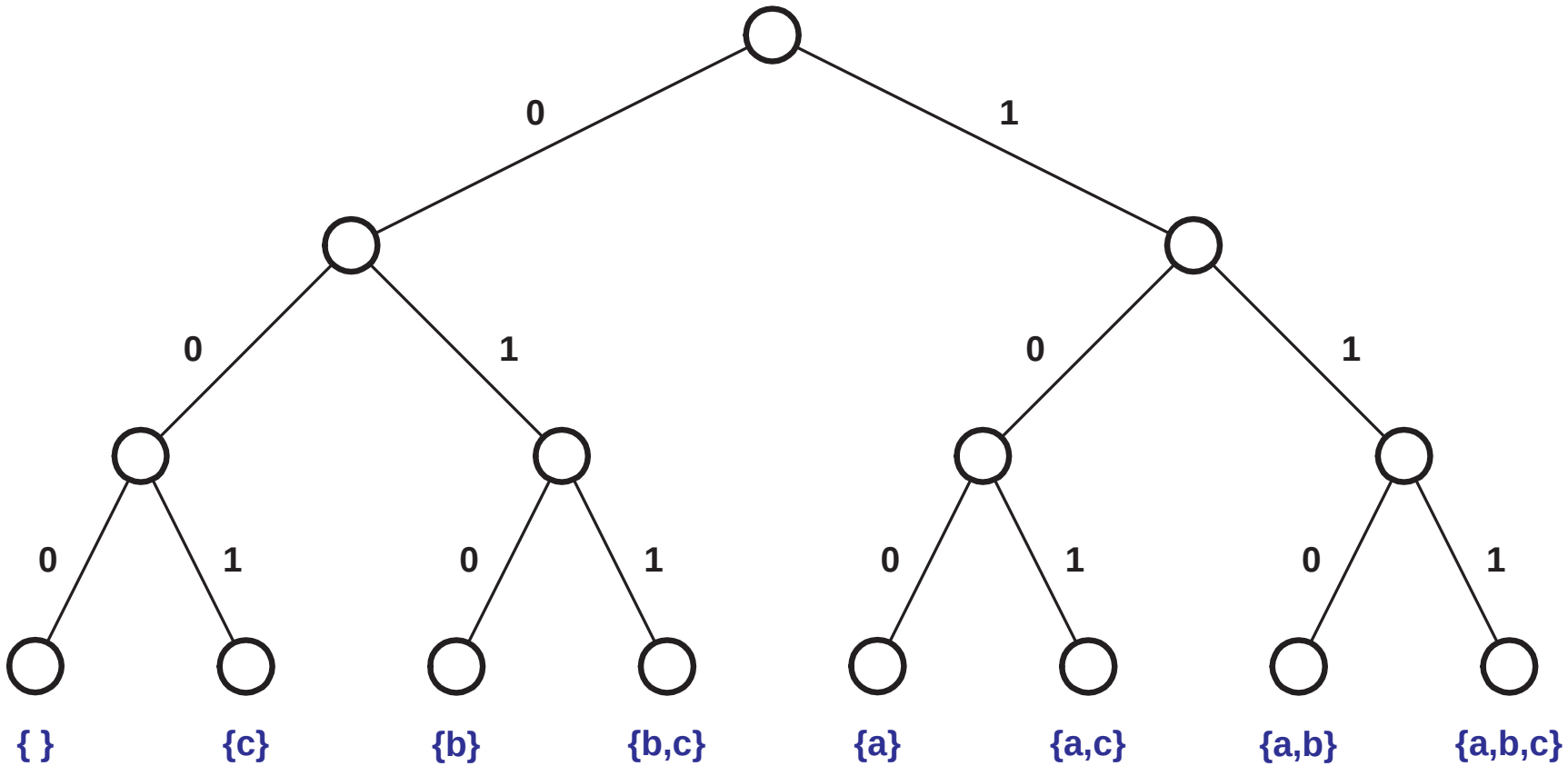


Subconjuntos y permutaciones

Subconjuntos de $\{a, b, c\}$

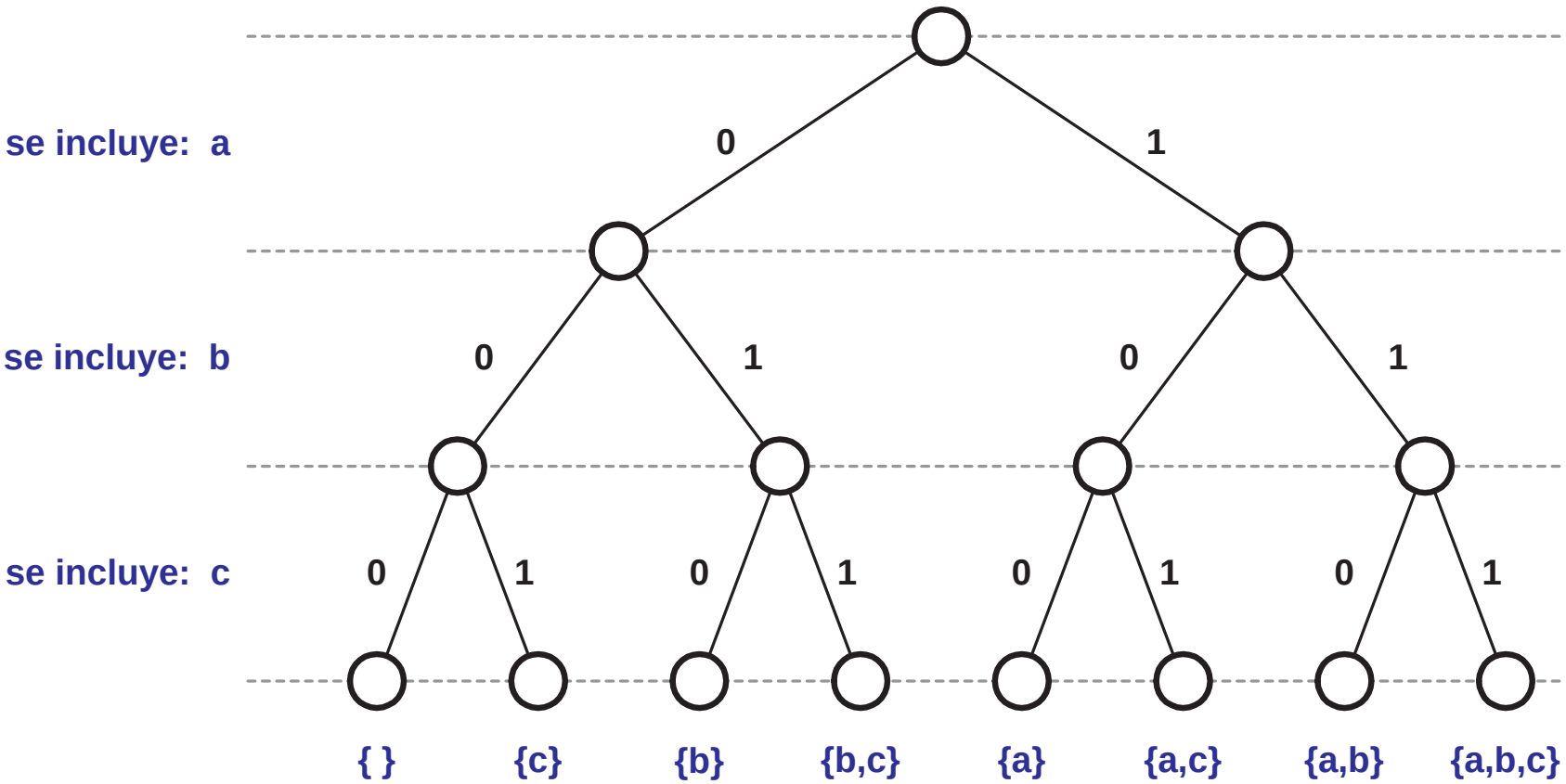
Solución - árbol binario

- Árbol recursivo del algoritmo ("árbol de búsqueda")



Subconjuntos de $\{a, b, c\}$

Solución - árbol binario



Subconjuntos de $\{a, b, c\}$

Solución - árbol binario

- Árbol binario
- Al descender hacia las hojas se decide si los elementos estarán presentes o no en el subconjunto creado
- Solución parcial: lista de valores binarios/booleanos
  - $[1, 0, 1] = \{a, c\}$
- Cada hoja representa un subconjunto de los 2^n posibles
  - n es la cardinalidad del conjunto inicial (en este caso $n = 3$)

Implementación – I

Solución - árbol binario

```
1  def genera_subconjuntos_wrapper(elementos):
2      sol = [None] * (len(elementos))
3      genera_subconjuntos(0,sol,elementos)
4
5
6  def imprime_subconjunto(sol,elementos):
7      ha_imprimido_alguno = False
8      print('{',end='')
9      for i in range(0,len(sol)):
10         if sol[i]==1:
11             if ha_imprimido_alguno:
12                 print(',',elementos[i],sep='',end='')
13             else:
14                 print(elementos[i],sep='',end='')
15                 ha_imprimido_alguno = True
16     print('}')
```

Implementación – II

Solución - árbol binario

```
1  def genera_subconjuntos(i,sol,elementos):
2
3      if i==len(elementos):  # Caso base: solución parcial completa
4
5          imprime_subconjunto(sol,elementos)  # Imprimir solución
6                                              # completa
7      else:
8          for k in range(0,2): # Genera candidatos a incluir en
9                               # solución parcial
10
11             sol[i] = k  # Incluye candidato k en solución parcial
12
13             # Expande solución parcial a partir del índice i+1
14             genera_subconjuntos(i+1,sol,elementos)
```

- `i`: índice del elemento a considerar (nivel del árbol)
- `elementos`: lista inicial de elementos
- `len(elementos)`: número de elementos (profundidad del árbol)
- `sol`: solución parcial (valores binarios que definen los elementos presentes en un subconjunto)

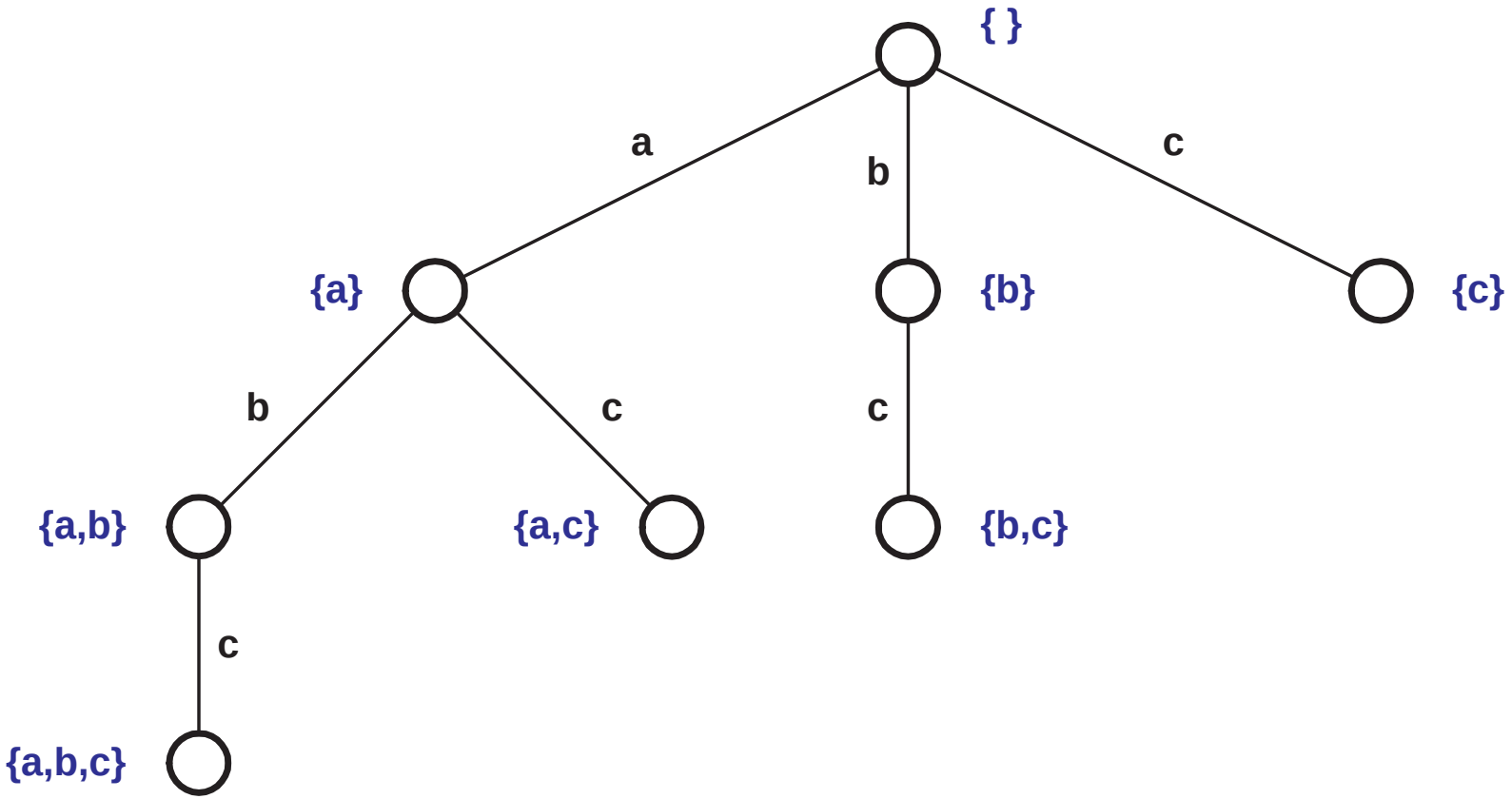
Consideraciones

Solución - árbol binario

- Código "sencillo"
  - Claro
  - ¡Hay que aprenderse estos códigos/esquemas generales!
  - Los algoritmos de backtracking que veremos tendrán una estructura similar, aunque también más elementos
- Buen esquema si los subconjuntos a generar tienen cardinalidades diferentes
- Se reserva memoria para toda la solución parcial en la función wrapper
  - Importante de cara a la eficiencia
  - Es preferible que usar una estructura de datos que crezca y decrezca dinámicamente

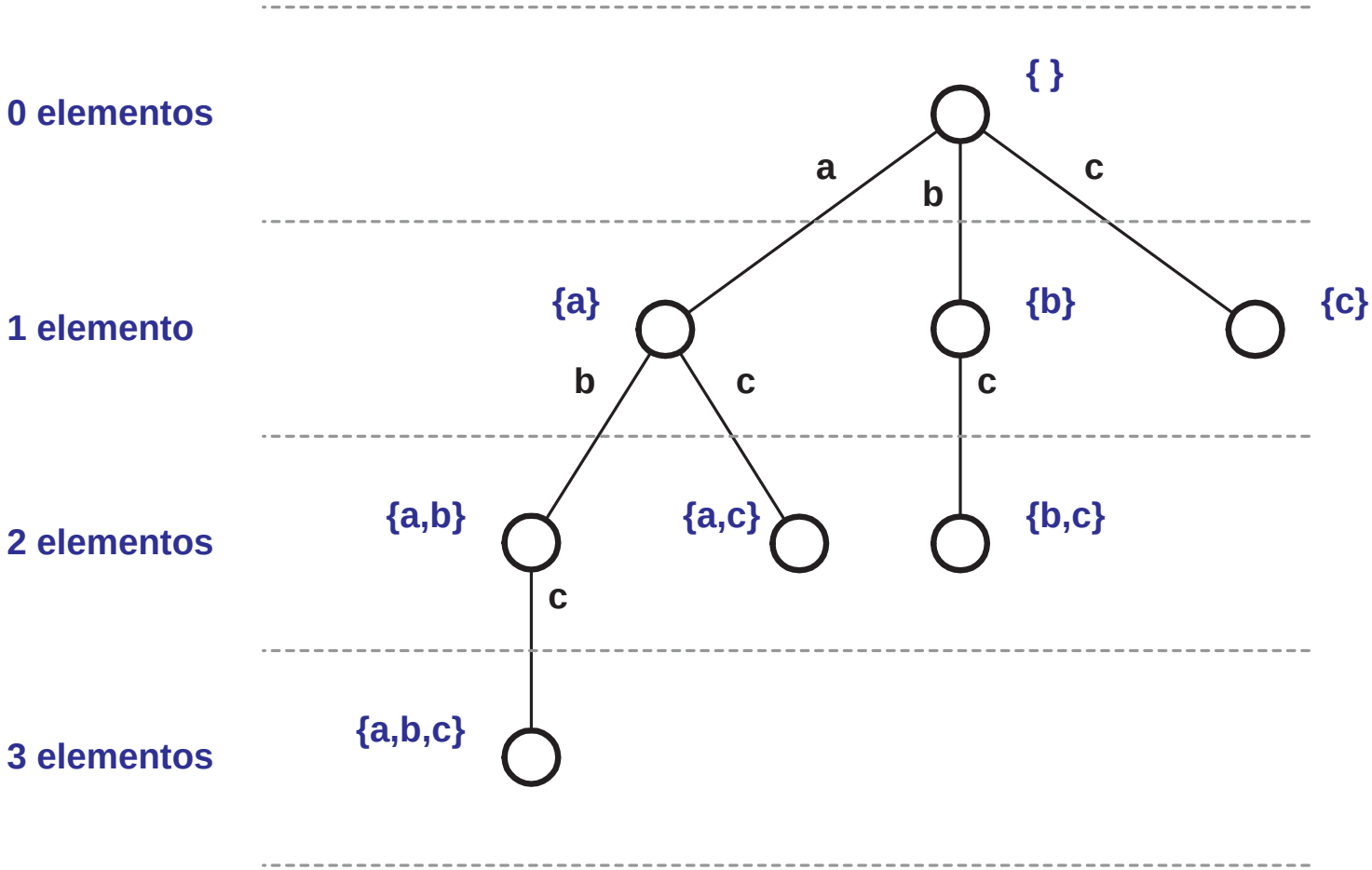
Subconjuntos de $\{a, b, c\}$

Solución - subconjunto en cada nodo



Subconjuntos de $\{a, b, c\}$

Solución - subconjunto en cada nodo



- La etiqueta de la rama indica qué elemento a insertar

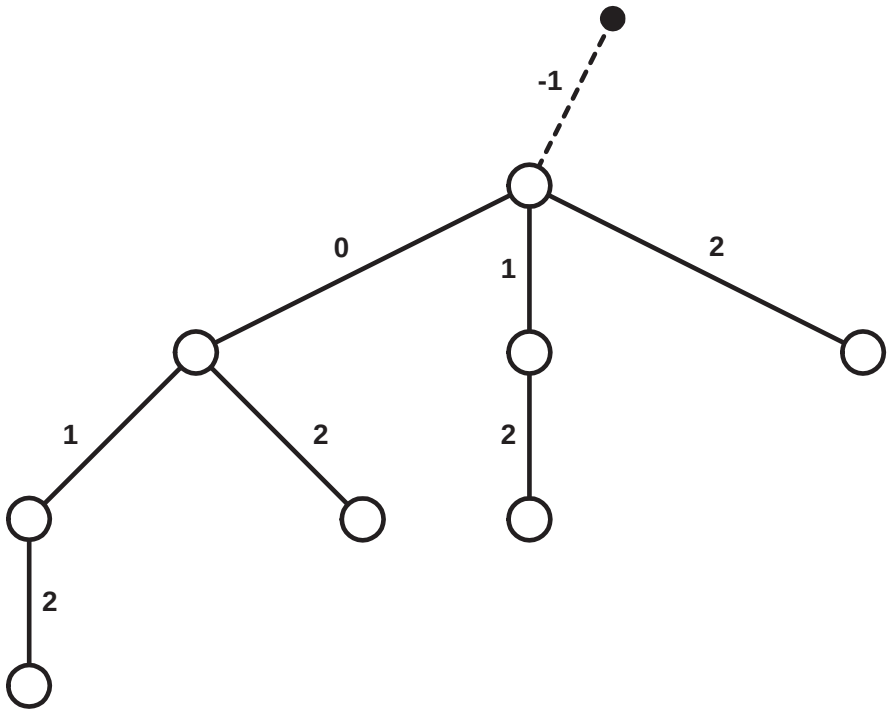
Subconjuntos de $\{a, b, c\}$

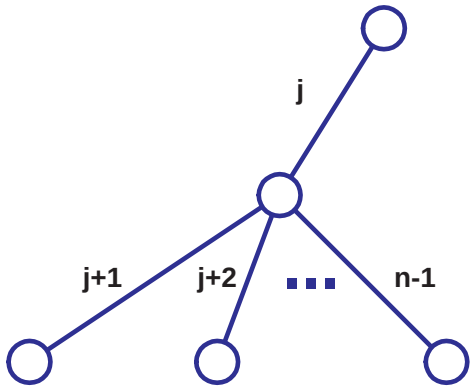
Solución - subconjunto en cada nodo

- El árbol de recursión no es binario
- En el nivel i las soluciones tienen i elementos
- La solución parcial es una lista con los índices de los elementos que se van incluyendo
  - $[0, 2] = \{a, c\}$
- Se obtiene un subconjunto en cada nodo (no solo en las hojas)
- Más eficiente si se buscan subconjuntos de la misma cardinalidad

Subconjuntos de $\{a, b, c\}$

Solución - subconjunto en cada nodo





- Ahora es necesario llevar dos índices: `i` y `j`
  - `i`: Índice de la solución parcial donde se incluirá un candidato
  - `j`: Índice del último candidato incluido

Implementación – I

Solución - subconjunto en cada nodo

```
1  def genera_subconjuntos_alt_wrapper(elementos):
2
3      # reserva memoria para la solución parcial
4      sol = [None] * (len(elementos))
5
6      # Llamada a función recursiva. j inicializada a -1 indica
7      # que no se ha incluido ningún candidato en la solución
8      # parcial todavía. La siguiente llamada intentará incluir
9      # en los subconjuntos todos los elementos, empezando por
10     # el del índice 0 = j+1
11     genera_subconjuntos_alt(0,-1,sol,elementos)
12
13
14 def imprime_subconjunto_alt(sol,n_elementos,elementos):
15     print('{',end='')
16     for i in range(0,n_elementos-1):
17         print(elementos[sol[i]],',',sep='',end='')
18     if n_elementos>0:
19         print(elementos[sol[n_elementos-1]],sep='',end='')
20     print('}')
```

Implementación – II

Solución - subconjunto en cada nodo

```
1  def genera_subconjuntos_alt(i,j,sol,elementos):
2
3      # Imprimir solución completa
4      imprime_subconjunto_alt(sol,i,elementos)
5
6      for k in range(j+1,len(elementos)):  # genera candidatos
7
8          sol[i] = k  # Incluye candidato k en solución parcial
9
10         # Expande solución parcial a partir del índice i+1
11         # sabiendo que k es el último candidato incluido en sol
12         genera_subconjuntos_alt(i+1,k,sol,elementos)
```

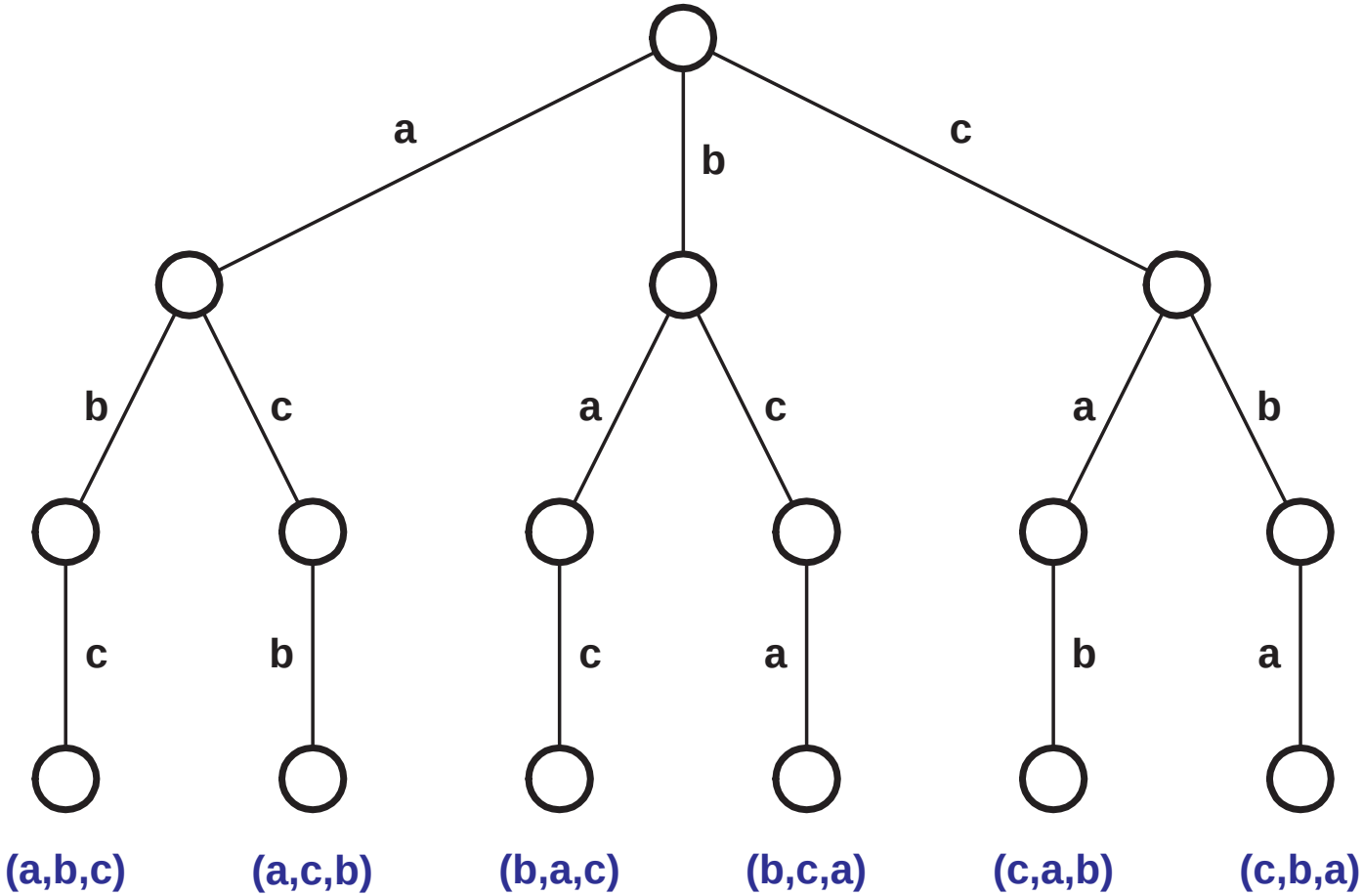
- `len(elementos)`: número de elementos (profundidad máxima del árbol)
- `i`: índice en la solución parcial (`sol`) del candidato a incluir
- `j`: índice del último candidato incluido en `sol`
- `sol`: solución parcial (índices de la lista `elementos`)
- `elementos`: conjunto (lista) original

Implementación – III

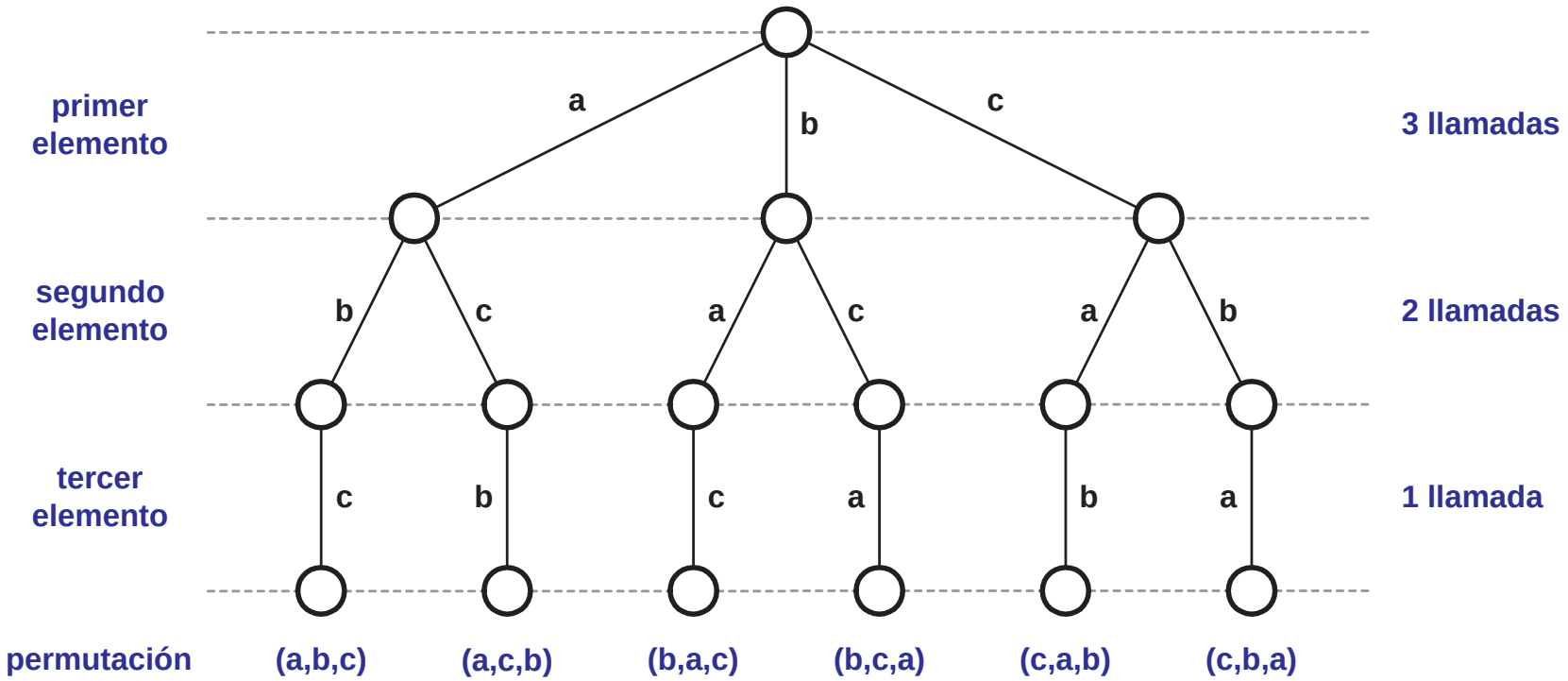
Solución - subconjunto en cada nodo (sin parámetro j)

```
1  def genera_subconjuntos_alt2(i,sol,elementos):
2
3      imprime_subconjunto_alt(sol,i,elementos)
4
5      if i==0:
6          k_ini = 0
7      else:
8          k_ini = sol[i-1]+1
9
10     for k in range(k_ini,len(elementos)):
11         sol[i] = k
12         genera_subconjuntos_alt2(i+1,sol,elementos)
```

Permutaciones de $\{a, b, c\}$



Permutaciones de $\{a, b, c\}$



Permutaciones de $\{a, b, c\}$

- ¿Cuántas llamadas recursivas se hacen en cada nivel?
  - 3, 2, 1 (se podrían controlar con bucles)
  - Pero lo más fácil y eficiente es generar siempre 3 posibles llamadas, y usar una lista de valores binarios o booleanos para ver si realmente se debe realizar la llamada recursiva
  - La lista de booleanos se puede pasar por valor o referencia
    - Referencia: habrá que deshacer los cambios al retroceder
    - Valor: no será necesario deshacer cambios
- El nivel de la llamada indica la posición del nuevo elemento a añadir
- Al llegar al último nivel (a las hojas) tenemos completada la permutación

Implementación – I

```
1  def genera_permutaciones_wrapper(elementos):
2
3      # Reserva memoria para la solución parcial y para lista
4      # que indica si un elemento ya se ha incluido en la
5      # solución parcial (si forma parte de la permutación)
6      sol = [None] * (len(elementos))
7      libres = [True] * (len(elementos))
8
9      # Llamada al método recursivo
10     genera_permutaciones(0,libres,sol,elementos)
11
12
13 def imprime_permutacion(sol):
14     for i in range(0,len(sol)):
15         print(sol[i],' ',end='')
16     print()
```

Implementación – II

```
1  def genera_permutaciones(i,libres,sol,elementos):
2      n = len(elementos)
3      if i==n:  # Caso base
4          imprime_permutacion(sol)  # Imprimir solución completa
5      else:
6          # Genera candidatos
7          for k in range(0,n):
8
9              # Comprueba validez del candidato
10             if libres[k]:
11
12                 # Incluye candidato k en solución parcial
13                 sol[i] = elementos[k]
14
15                 # El candidato k deja de estar libre
16                 libres[k] = False
17
18                 # Expande solución parcial a partir del índice i+1
19                 genera_permutaciones(i+1,libres,sol,elementos)
20
21                 # El candidato k vuelve a estar libre
22                 libres[k] = True
```

Consideraciones

- Parámetros:
  - `i`: índice en la solución parcial donde incluir un nuevo elemento
  - `libres`: lista booleana que indica si un elemento ya se ha incluido en la solución parcial
  - `sol`: solución (permutación) parcial construida
  - `elementos`: elementos a permutar
- ¡Hay que aprenderse estos códigos/esquemas generales!
- `sol` también podría contener índices a los elementos
- El uso de la lista `libres` es clave de cara a la eficiencia
  - Se calcula si un elemento está libre en $\mathcal{O}(1)$
  - Si no se usara se necesitaría un bucle para comprobarlo, cuyo coste sería $\mathcal{O}(n)$, donde n es el número de elementos a permutar
- La reserva de memoria también influye en la eficiencia
  - Se debe evitar concatenar elementos a `sol` (o borrarlos)

Permutaciones ineficiente

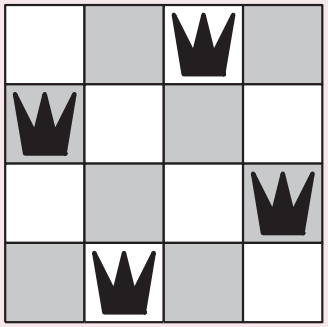
```
1  def es_factible(elemento,sol,i):
2      esta_elemento = False
3      j = 0
4      while (j<i) and not esta_elemento:
5          esta_elemento = (elemento==sol[j])
6          j = j + 1
7
8      return not esta_elemento
9
10
11 def genera_permutaciones(i,sol,elementos):
12     if i==len(elementos):
13         imprime_permutacion(sol)
14     else:
15         for k in range(0,len(elementos)):
16             if es_factible(elementos[k],sol,i):
17                 sol[i] = elementos[k]
18                 genera_permutaciones(i+1,sol,elementos)
```

Problema de las N reinas

N reinas

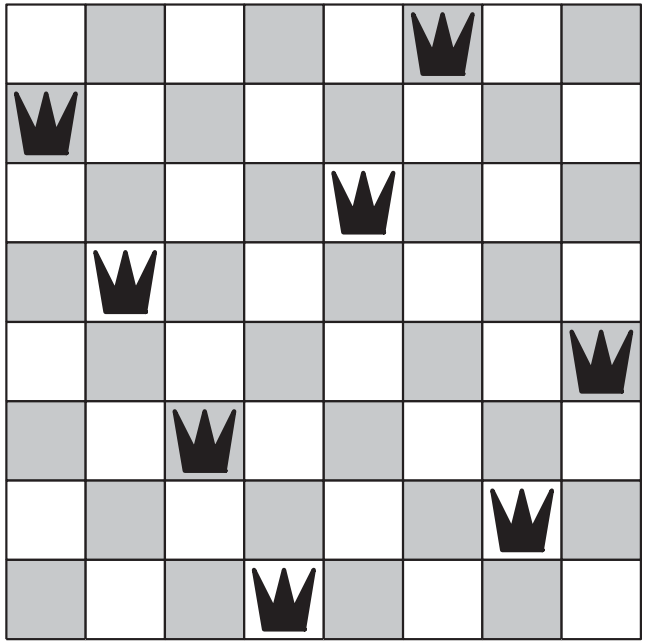
Problema de las N reinas

- Dado un "tablero ajedrez" de $n \times n$ celdas, se pide ubicar n reinas de modo que no se amenacen
  - No pueden estar en la misma fila
  - No pueden estar en la misma columna
  - No pueden estar en la misma diagonal (principal o secundaria)



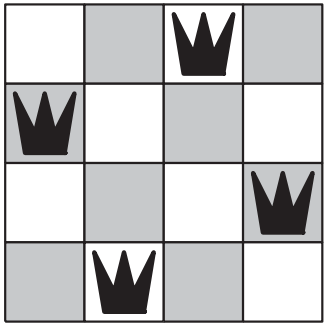
- Ejemplo más famoso de problema que puede resolverse aplicando *backtracking*

8 reinas



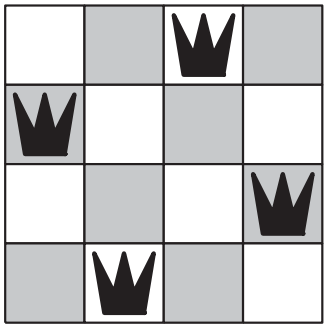
- Para $n = 8$ hay 92 soluciones posibles
- Aunque 12 únicas
  - Las demás pueden obtenerse aplicando simetrías, rotaciones y traslaciones
- El problema puede solicitar encontrar una solución o todas

N reinas



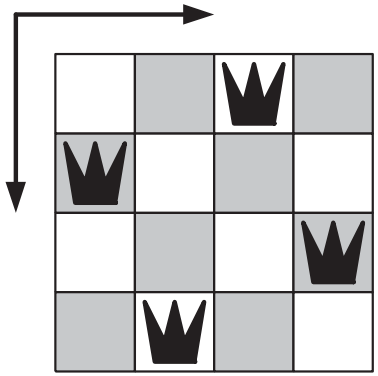
- Solución obvia pero absurda:
  - Probar las 2^{n^2} formas de colocar reinas en el tablero
    - $1{,}84 \cdot 10^{19}$ para $n = 8$
    - 65536 para $n = 4$
- Pero los conjuntos solución solo deben contener n elementos
  - Esto reduciría nuestro espacio de búsqueda a $\binom{n^2}{n}$
    - $4{,}42 \cdot 10^{10}$ para $n = 8$
    - 1820 para $n = 4$

N reinas



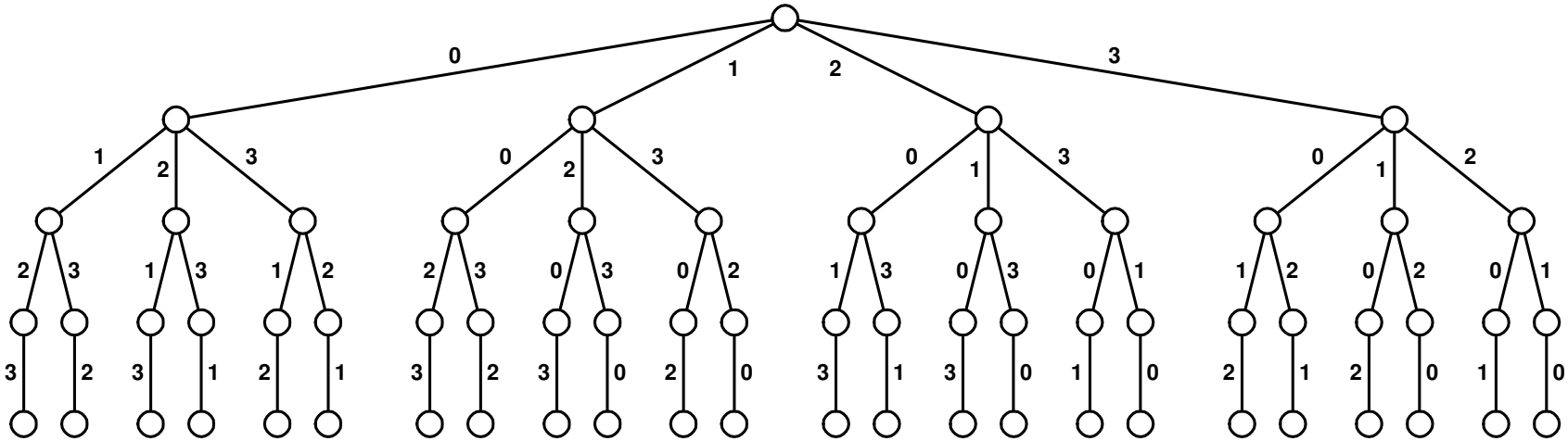
- Además solo puede haber una reina por cada columna:
  - Esto reduce las posibilidades a n^n (hay n formas de colocar una reina en una columna, y hay n columnas)
    - 16777216 para $n = 8$
    - 256 para $n = 4$
- Pero además, no puede haber dos reinas en la misma fila
  - Esto convierte nuestro problema en la búsqueda de una permutación con $n!$ posibilidades (lo cual sigue siendo elevado)
    - 40320 para $n = 8$
    - 24 para $n = 4$

N reinas



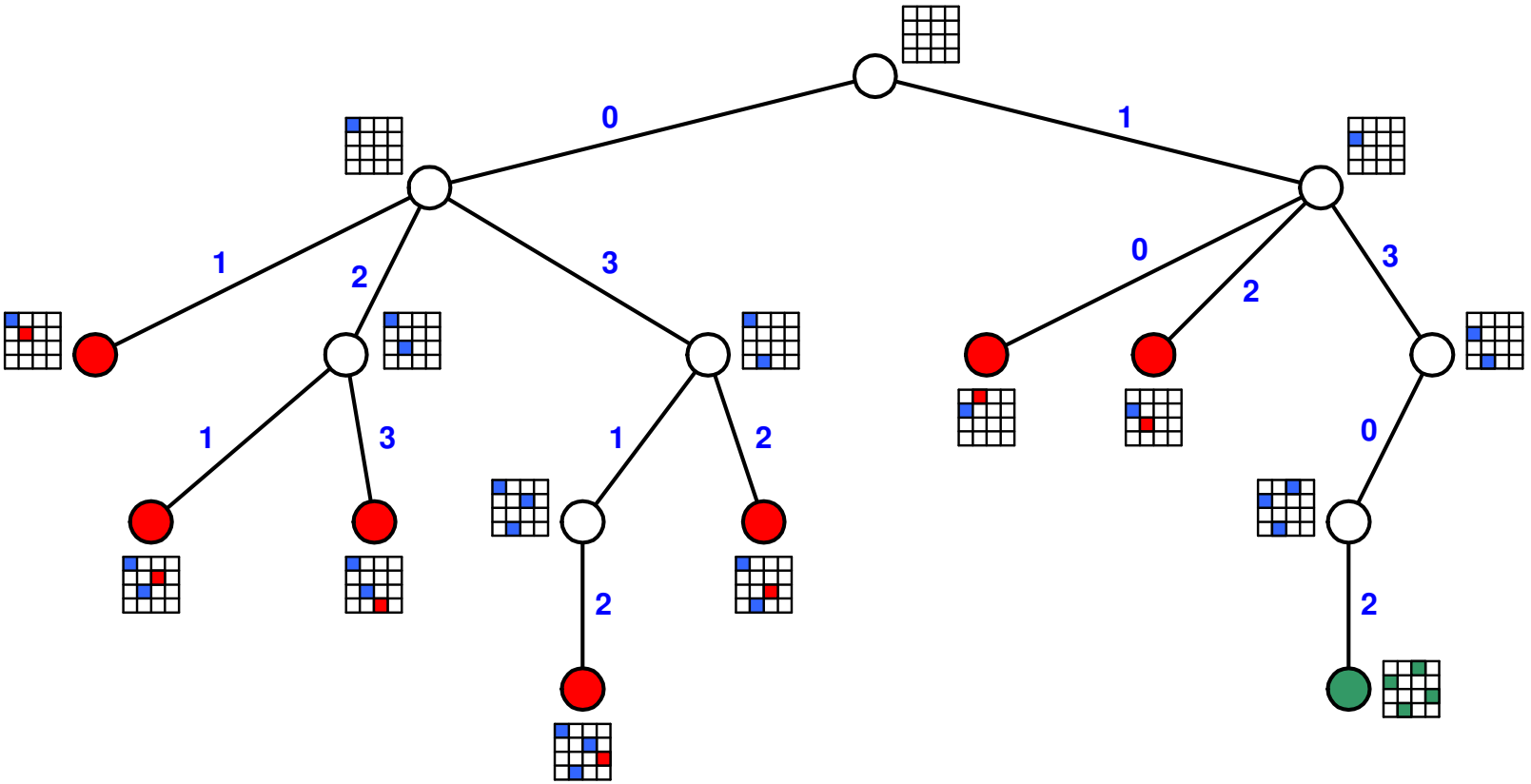
- Formato de la solución:
  - (fila de la columna 0, fila de la columna 1,. . . ,fila de la columna $n-1$)
  - $(1, 3, 0, 2)$ en la figura
  - Todas las filas son diferentes y tienen que estar representadas
  - Tendremos que buscar las **permutaciones** válidas

Árbol de búsqueda para $N = 4$



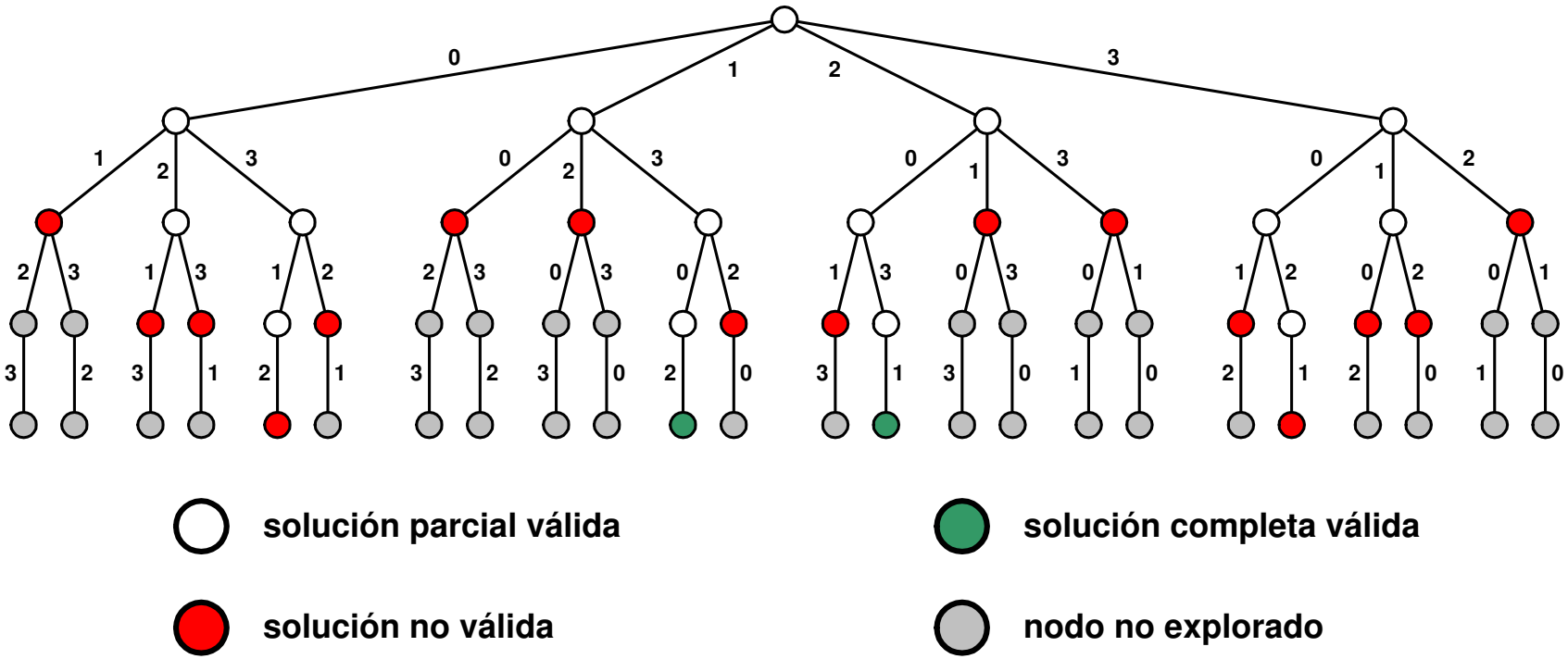
- Podemos emplear un algoritmo para buscar permutaciones
  - Al llegar a una hoja se "han colocado" las cuatro reinas y podemos probar si la solución es válida
  - Pero, podemos comprobar si la solución parcial puede llegar a ser solución final antes de llegar a una hoja
    - Podaríamos el árbol ahorrando cálculos

Árbol de búsqueda para $N = 4$

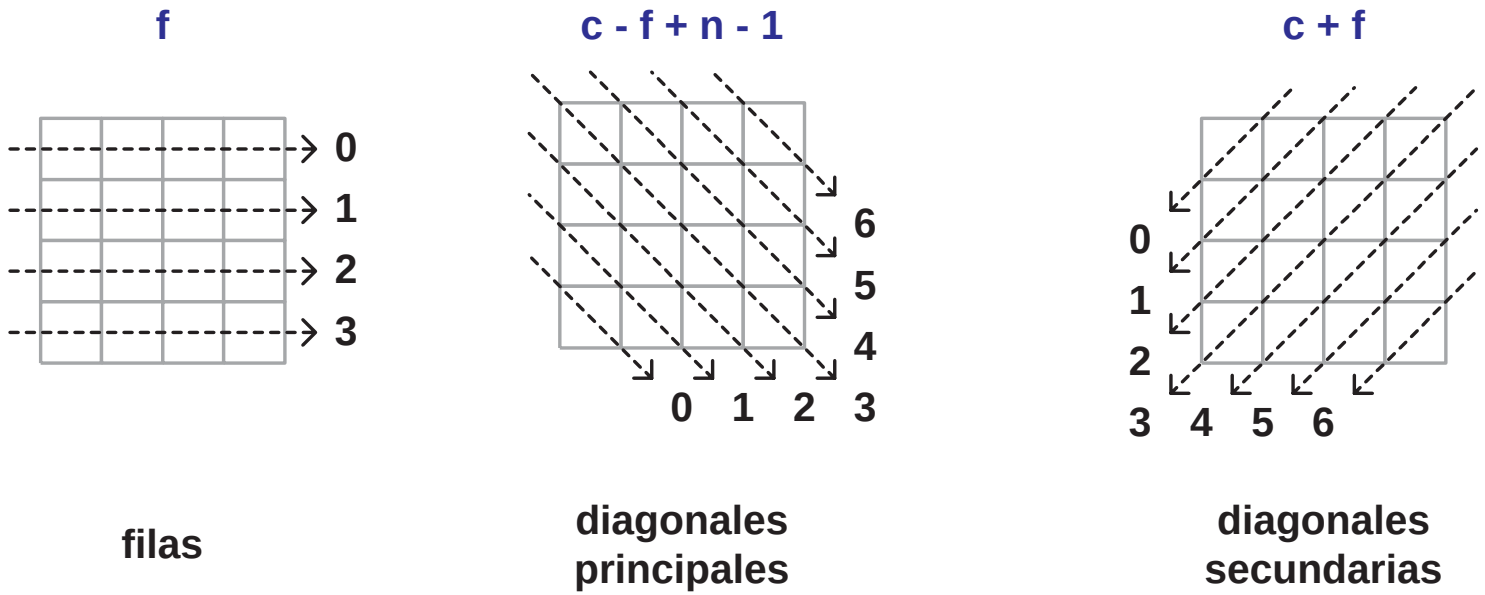


- Solo se muestra el árbol hasta encontrar la primera solución válida

Árbol de búsqueda podado para $N = 4$



Implementación



- Para verificar que una solución parcial es válida usamos:
  - `filas_libres`: filas libres
  - `pdiags_libres`: diagonales principales libres ($columna - fila + n - 1$)
  - `sdiags_libres`: diagonales secundarias libres ($columna + fila$)
- `sol`: solución parcial

Implementación – I

```
1  def nreinas_todas_wrapper(n):
2
3      # reserva de memoria para estructuras de datos booleanas
4      filas_libres = [True] * n
5      pdiags_libres = [True] * (2*n-1)
6      sdiags_libres = [True] * (2*n-1)
7
8      # reserva de memoria para la solución parcial
9      sol = [None] * n
10
11     # llamada a función recursiva
12     nreinas_todas(0,filas_libres,pdiags_libres,sdiags_libres,sol)
```

Implementación – II

```
1  def nreinas_todas(i,filas_libres,pdiags_libres,sdiags_libres,sol):
2      n = len(sol)
3      if i==n: # Comprueba si la solución parcial es completa
4          print(sol)
5      else:
6          # Genera candidatos
7          for k in range(0,n):
8              # Comprueba restricciones
9              if filas_libres[k] and pdiags_libres[i-k+n-1] and sdiags_libres[i+k]:
10
11                 # Introduce candidato k en la solución parcial
12                 sol[i] = k
13
14                 # Actualiza estructuras booleanas
15                 filas_libres[k] = False
16                 pdiags_libres[i-k+n-1] = False
17                 sdiags_libres[i+k] = False
18
19                 # Intenta expandir la solución parcial
20                 nreinas_todas(i+1,filas_libres,pdiags_libres,sdiags_libres,sol)
21
22                 # Vuelve a marcar la fila y las diagonales como libres
23                 filas_libres[k] = True
24                 pdiags_libres[i-k+n-1] = True
25                 sdiags_libres[i+k] = True
```

Implementación – III

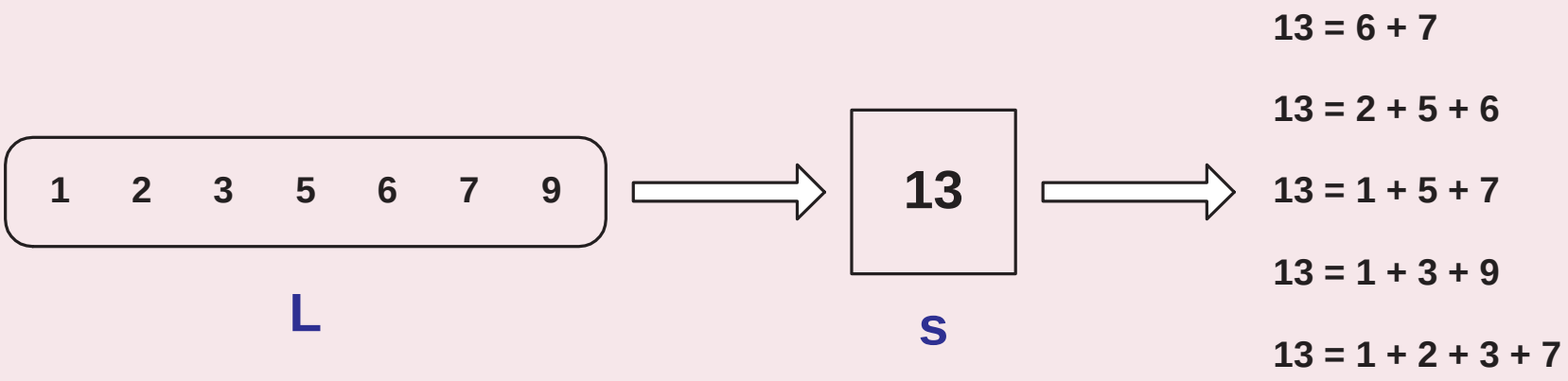
- Líneas 3 y 4: si se ha llegado a una solución válida se imprime
- Línea 7: se generan los candidatos
- Línea 9: se comprueba la validez del candidato
- Línea 12: se incluye el candidato en la solución parcial
- Líneas 15–17: se marcan como ocupadas las correspondientes filas y diagonales
- Línea 20: nueva llamada recursiva para extender la solución parcial
- Líneas 23–25: Se marcan como libres la fila y diagonales del candidato para que puedan usarse en futuros pasos

Otros problemas

Suma de subconjuntos

Problema de la suma de subconjuntos

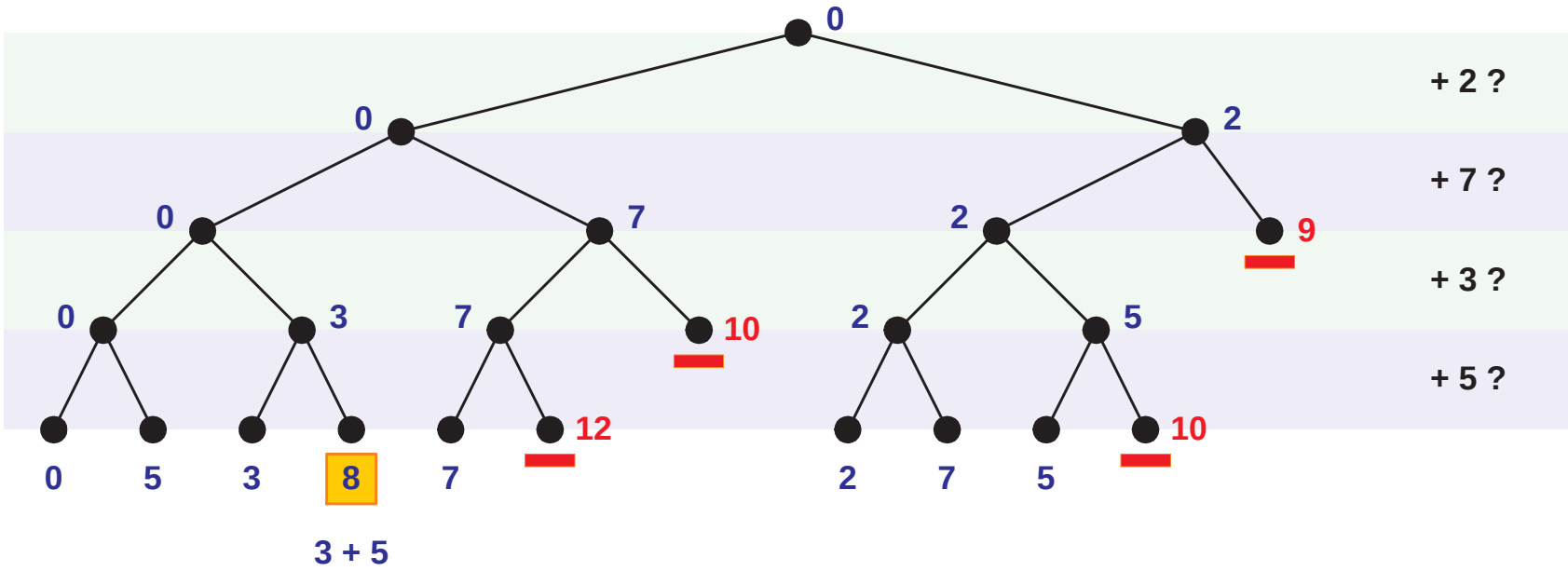
Dada una lista L de números no negativos (que pueden repetirse) y otro número s, determinar los subconjuntos de L cuyos elementos suman s



- Las soluciones son subconjuntos de tamaño variable
  - Usaremos el esquema binario para generar subconjuntos

Suma de subconjuntos

- Ejemplo: $L = [2, 7, 3, 5]$, $s = 8$
- Se usará un parámetro que almacene la suma parcial (suma de los elementos de la solución parcial)
  - ¡Con este parámetro mejora la eficiencia!
  - Se podrá obtener la suma parcial en $\Theta(1)$, en lugar de $\mathcal{O}(n)$ si no se usa
- Restricción adicional: podar el árbol si la suma parcial es mayor que s



Implementación – I

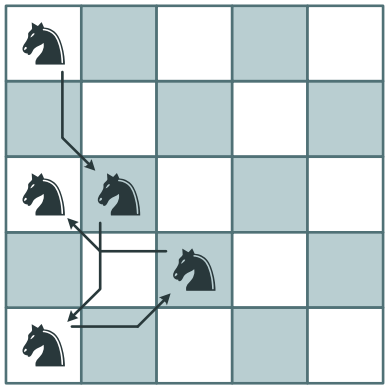
```
1  def suma_subconjuntos_wrapper(L,s):
2
3      # Reserva memoria para solución parcial
4      sol = [0] * (len(L))
5
6      suma_subconjuntos(0,sol,0,L,s)
7
8
9  def imprime_subconjunto(sol,elementos,j):
10     ha_imprimido_alguno = False
11     print('{',end='')
12     for i in range(0,j):
13         if sol[i]==1:
14             if ha_imprimido_alguno:
15                 print(',',elementos[i],sep='',end='')
16             else:
17                 print(elementos[i],sep='',end='')
18                 ha_imprimido_alguno = True
19     print('}')
```

Implementación – II

```
1  def suma_subconjuntos(i,sol,suma_parcial,L,s):
2      # Caso base
3      if suma_parcial==s:
4          imprime_subconjunto(sol,L,i)
5      elif i<len(L):
6          # Genera candidatos
7          for k in range(0,2): # Esquema binario
8
9              # Comprueba si se puede podar el árbol recursivo
10             if suma_parcial + k*L[i]<=s:
11
12                 # Expande la solución parcial
13                 sol[i] = k
14
15                 # Actualiza la solución parcial
16                 suma_parcial = suma_parcial + k*L[i]
17
18                 # Intenta expandir la solución parcial
19                 suma_subconjuntos(i+1,sol,suma_parcial,L,s)
```

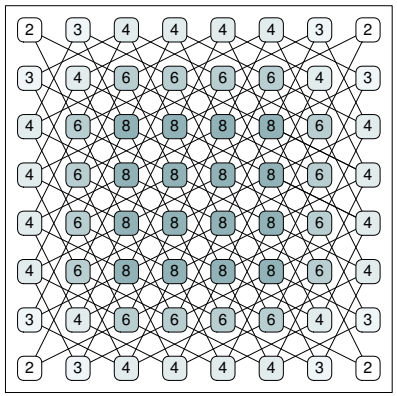
Ruta del caballo de ajedrez

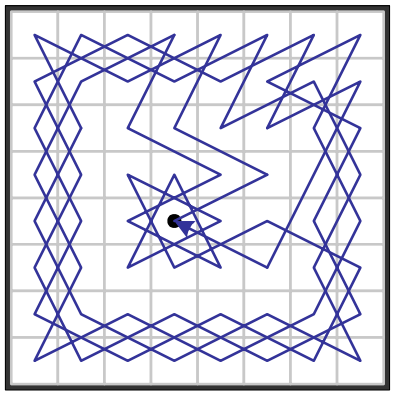
- Identificar una ruta de una caballo de ajedrez en un tablero de $n \times n$ de manera que salte a todas las casillas sin repetir ninguna
  - No es requisito que tenga que volver al punto de partida



1	14	9	20	23
10	19	22	15	8
5	2	13	24	21
18	11	4	7	16
3	6	17	12	25

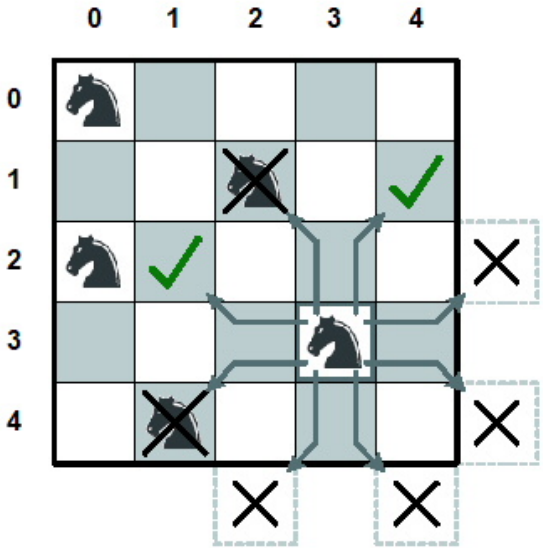
- Variante: **ciclo Hamiltoniano** (camino cerrado en un grafo)

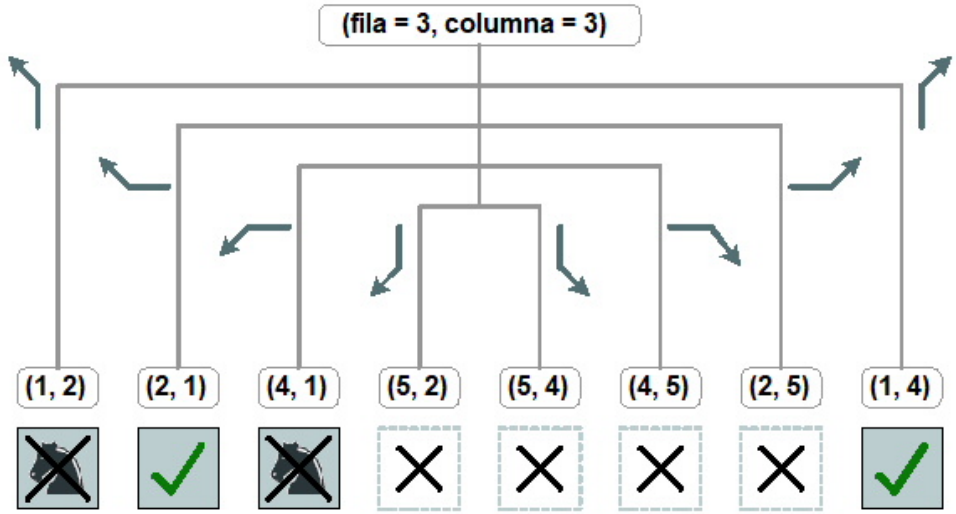




Ruta del caballo de ajedrez

Árbol de búsqueda





- Se podría encontrar una permutación de n^2 elementos (tablero de $n \times n$)
  - Las restricciones serían muy complejas
- Para tamaños mayores que 6×6 tarda en hallar una solución

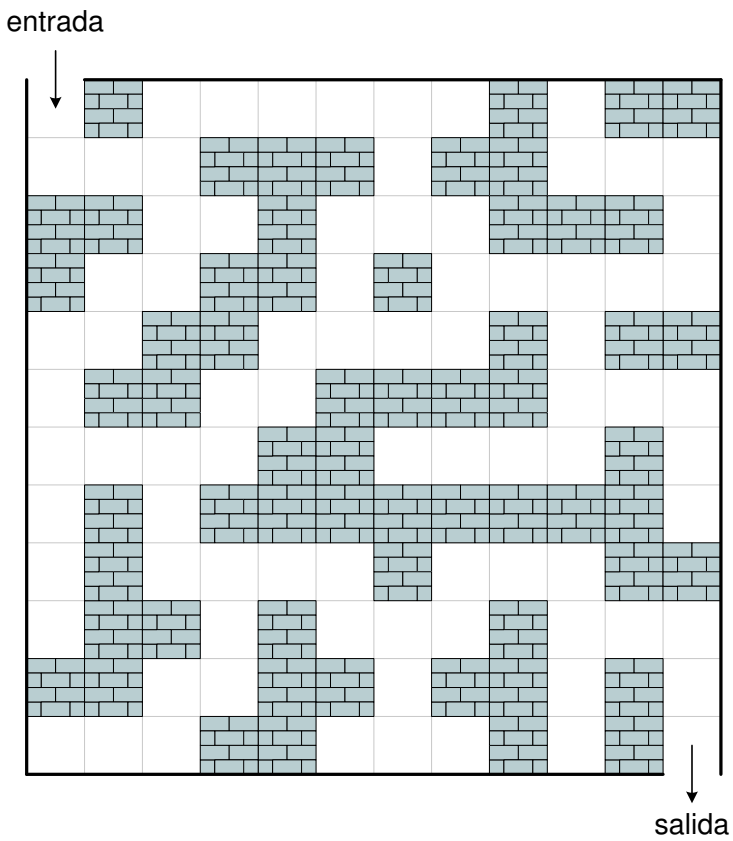
Implementación – I

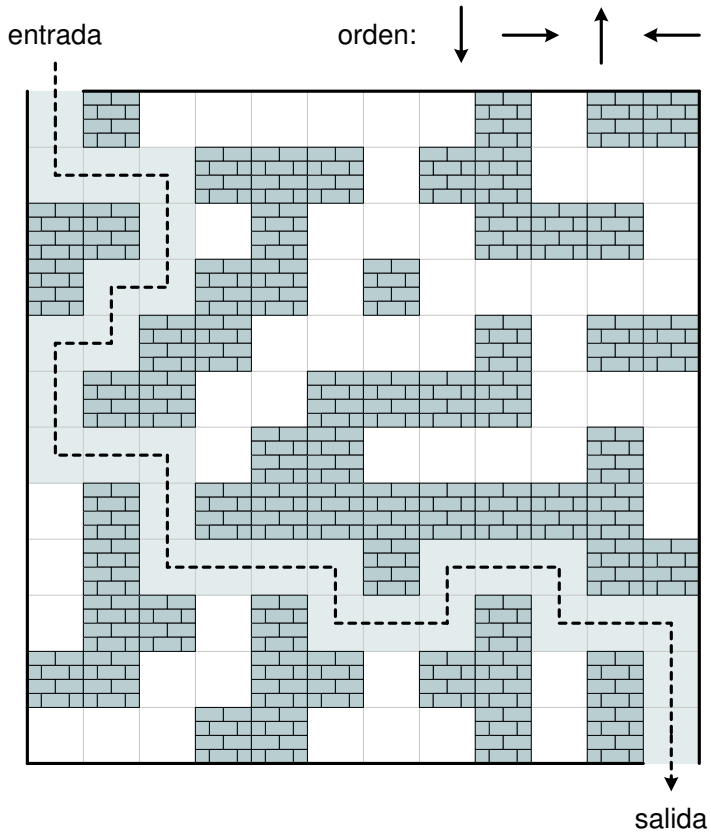
```
1  def busca_ruta_caballo_ajedrez_wrapper(n,fila_inicial,col_inicial):
2      # lista de incrementos a las coordenadas de una casilla
3      # para especificar un salto de caballo
4      incr = [(1,-2),(2,-1),(2,1),(1,2), \
5              (-1,2),(-2,1),(-2,-1),(-1,-2)]
6
7      # Inicializar una lista de listas a 0 (solución parcial)
8      B = [None]*n
9      for i in range(0,n):
10         B[i] = [0]*n
11
12     B[fila_inicial][col_inicial] = 1  # ubicamos un primer caballo
13
14     # Si encontramos una solución la imprimimos
15     if busca_ruta_caballo_ajedrez(2,fila_inicial,col_inicial,B,incr):
16         imprime_matriz(B)
17
18 def imprime_matriz(B):
19     n = len(B)
20     for i in range(0,n):
21         for j in range(0,n):
22             print("%5d" % (B[i][j]),end='')
23         print()
24
25 busca_ruta_caballo_ajedrez_wrapper(5,0,0)
```

Implementación – II

```
1  def busca_ruta_caballo_ajedrez(i,fila,col,B,incr):
2      if i>len(B)**2:  # Comprobar si se ha completado una ruta
3          return True   # Solución encontrada
4      else:
5          sol_encontrada = False
6          k=0
7          while not sol_encontrada and k<8:  # Genera candidatos
8              # Nueva casilla candidata
9              nueva_col = col + incr[k][0]
10             nueva_fila = fila + incr[k][1]
11
12             # Comprueba validez de la candidata
13             if nueva_fila>=0 and nueva_fila<len(B) and \
14                nueva_col>=0 and nueva_col<len(B[0]) and \
15                B[nueva_fila][nueva_col]==0:
16
17                 # Añade casilla a la ruta (a la solución parcial)
18                 B[nueva_fila][nueva_col] = i
19
20                 # Intenta expandir la ruta empezando desde la nueva casilla
21                 sol_encontrada = busca_ruta_caballo_ajedrez(i+1,nueva_fila, \
22                                                             nueva_col,B,incr)
23
24                 # Marca la nueva casilla como vacía si no se encuentra solución
25                 if not sol_encontrada:
26                     B[nueva_fila][nueva_col] = 0
27
28             k=k+1
29         return sol_encontrada
```

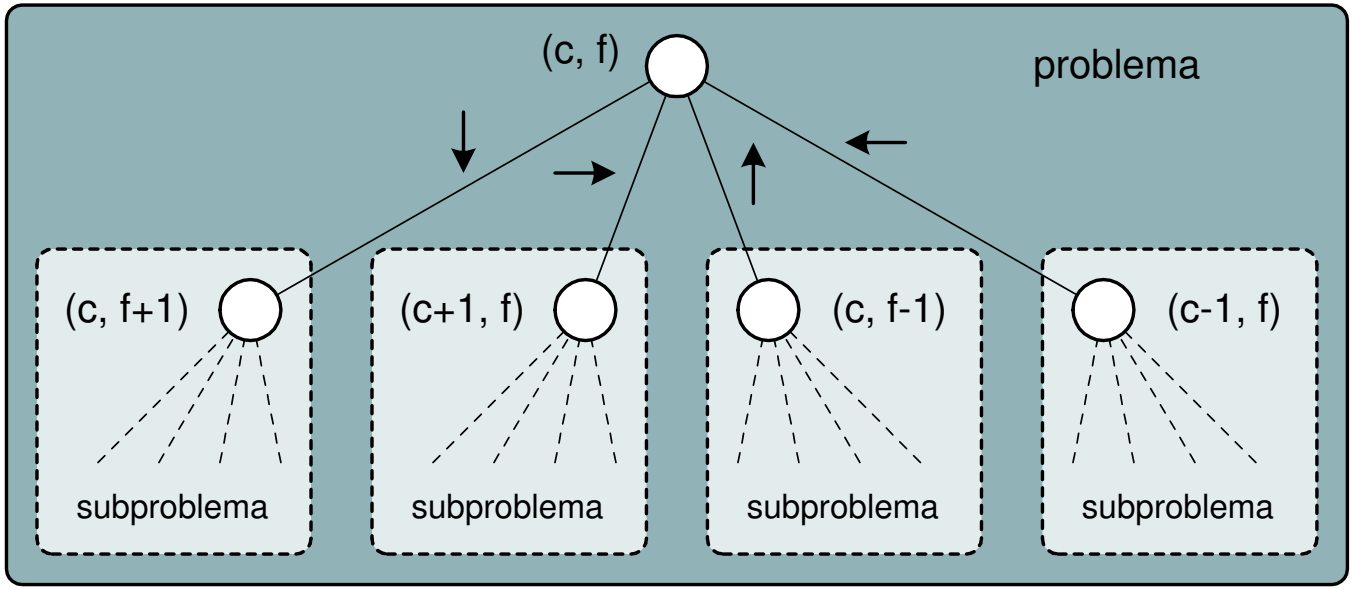
Laberinto





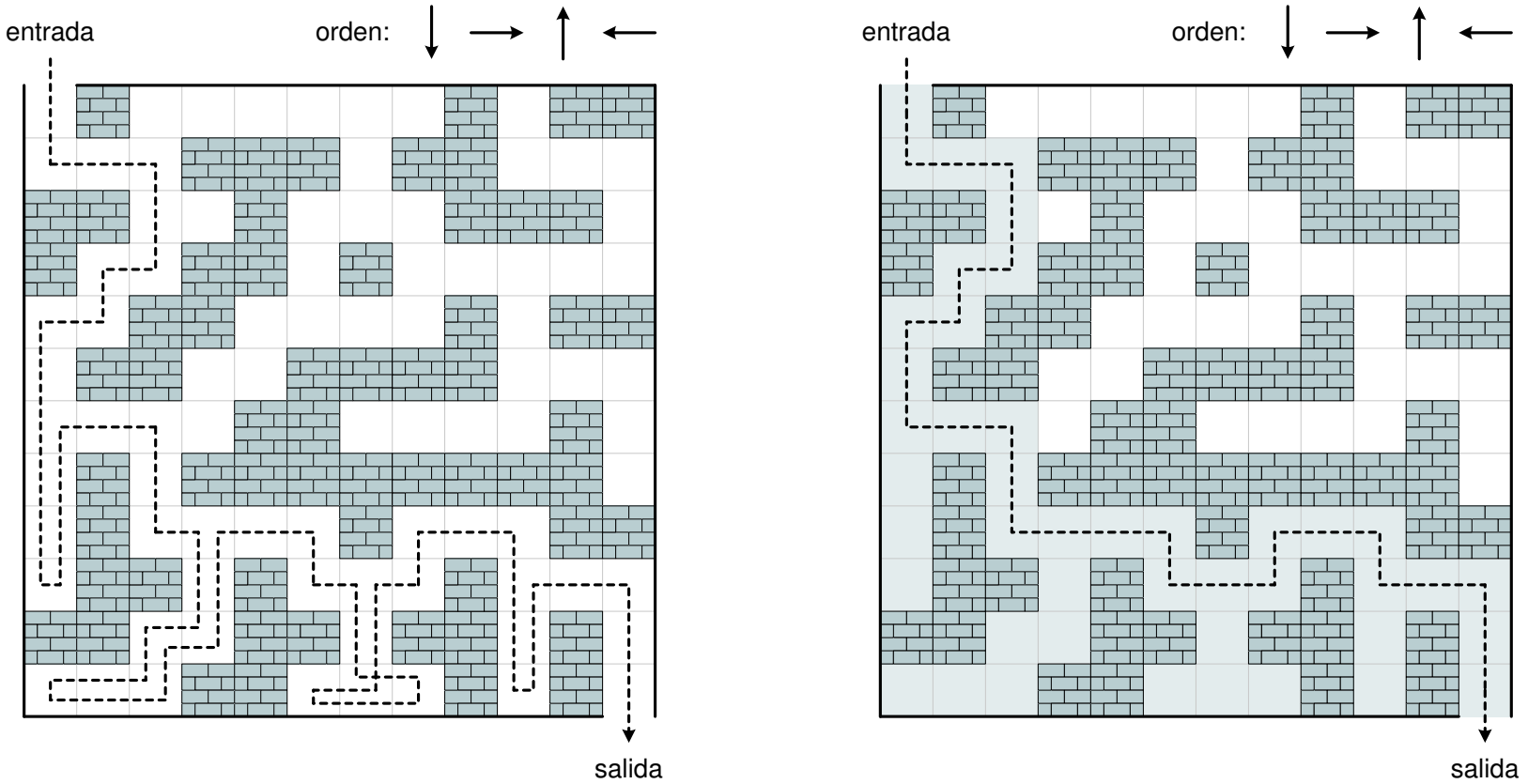
Laberinto

Árbol de búsqueda



- No se debe hallar una permutación (el camino puede tener duración variable)
- No se debe hallar un subconjunto de celdas (el orden importa)
- Usaremos un esquema diferente

Laberinto


entrada
orden:
salida
entrada
orden:
salida
celda vacía
pared
celda explorada
camino solución
camino explorado

Implementación – I

```
1  def lee_laberinto_de_fichero(nombre_fichero):
2      fichero = open(nombre_fichero,'r')
3      M=[]
4      for line in fichero.readlines():
5          M.append( [ x[0] for x in line.split(' ') ] )
6      fichero.close()
7      return M
8
9  def encuentra_camino_wrapper(M,fila_entrada,col_entrada,
       fila_salida,col_salida):
10     incr = [(0,1),(1,0),(0,-1),(-1,0)]
11
12     M[fila_entrada][col_entrada] = 'P'  # Camino (path)
13     return encuentra_camino(fila_entrada,col_entrada,M, \
14                             incr,fila_salida,col_salida)
15
16 M = lee_laberinto_de_fichero('laberinto_01.txt')
17 # Entrada esquina arriba-izquierda, salida abajo-derecha
18 if encuentra_camino_wrapper(M,0,0,len(M)-1,len(M[0])-1):
19     dibuja_laberinto(M,0,0,len(M)-1,len(M[0])-1)
```

Fichero de entrada (laberinto_01.txt)

```
1  E W E E E E E E W E W W
2  E E E W W W E W W E E E
3  W W E E W E E E W W W E
4  W E E W W E W E E E E E
5  E E W W E E E E W E W W
6  E W W E E W W W W E E E
7  E E E E W W E E E E W E
8  E W E W W W W W W W W E
9  E W E E E E W E E E W W
10 E W W E W E E E W E E E
11 W W E E W W E W W E W E
12 E E E W W E E E W E W E
```

Implementación – II

```
1  def encuentra_camino(fila,col,M,incr,fila_salida,col_salida):
2      if fila==fila_salida and col==col_salida:
3          return True  # Caso base. Se encontró una solución
4      else:
5          sol_encontrada = False
6          # Genera candidatos
7          k=0
8          while not sol_encontrada and k<4:
9              # Nueva celda candidata
10             nueva_col = col + incr[k][0]
11             nueva_fila = fila + incr[k][1]
12
13             # Comprueba validez de la nueva celda candidata
14             if nueva_fila>=0 and nueva_fila<len(M) and \
15                 nueva_col>=0 and nueva_col<len(M[0]) and \
16                 M[nueva_fila][nueva_col]=='E':
17
18                 # Añade al camino ((usamos M como solución parcial)
19                 M[nueva_fila][nueva_col] = 'P'
20
21                 # Intenta expandir el camino empezando en la nueva celda
22                 sol_encontrada = encuentra_camino(nueva_fila,nueva_col, \
23                                                   M,incr,fila_salida,col_salida)
24
25                 # Marca la nueva celda como vacía si no se ha encontrado solución
26                 if not sol_encontrada:
27                     M[nueva_fila][nueva_col] = 'E'
28
29             k=k+1
30         return sol_encontrada
```

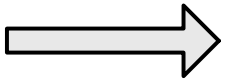
Implementación – III

```
1  gris = (0.75,0.75,0.75); amarillo = (0.75,0.75,0); turquesa = (0,0.75,0.75)
2
3  import matplotlib.pyplot as plt; from matplotlib.patches import Rectangle
4
5  def dibuja_laberinto(M,fila_entrada,col_entrada,fila_salida,col_salida):
6      nfilas = len(M); ncols = len(M[0])
7      fig = plt.figure(); fig.patch.set_facecolor('white'); ax = plt.gca()
8
9      if fila_entrada is not None and col_entrada is not None:
10         ax.add_patch(Rectangle((col_entrada,nfilas-fila_entrada), \
11                 1,-1,linewidth=0,facecolor=turquesa,fill=True))
12     if fila_salida is not None and col_salida is not None:
13         ax.add_patch(Rectangle((col_salida,nfilas-fila_salida), \
14                 1,-1,linewidth=0,facecolor=amarillo,fill=True))
15
16     for fila in range(0,nfilas):
17         for col in range(0,ncols):
18             if M[fila][col]=='W':
19                 ax.add_patch(Rectangle((col,nfilas-fila),1,-1, \
20                                 linewidth=0,facecolor=gris))
21             elif M[fila][col]=='P':
22                 circ=plt.Circle((col+0.5,nfilas-fila-0.5), \
23                     radius=0.15, color='black', fill=True)
24                 ax.add_patch(circ)
25
26     ax.add_patch(Rectangle((0,0),ncols,nfilas,edgecolor='black',fill=False))
27     plt.axis('equal'); plt.axis('off'); plt.show()
```

Sudoku

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

Sudoku inicial



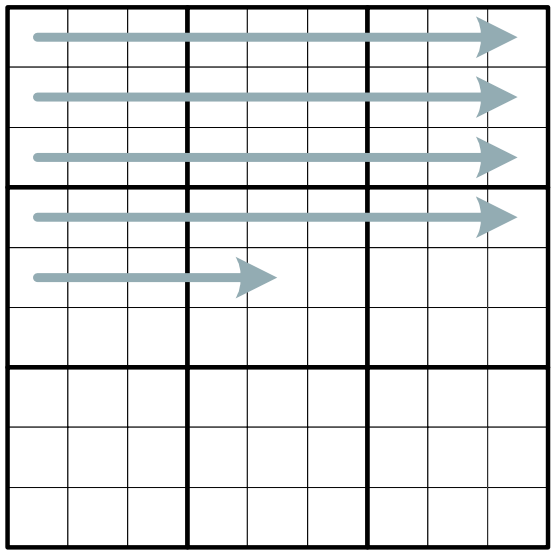
9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

Solución

- Rellenar el tablero con números
  - No puede haber dos mismos números en una fila, columna o caja de 3×3

Sudoku

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



Progreso por filas

- Solución parcial: el propio tablero
- Expandimos la solución parcial (por ejemplo) por filas

Implementación – I

```
1  def lee_sudoku(nombre_fichero):
2      fichero = open(nombre_fichero,'r')
3      S=[[None]*9]*9
4      i=0
5      for line in fichero.readlines():
6          S[i] = [int(x) for x in line.split(' ')]
7          i=i+1
8
9      fichero.close()
10     return S
11
12 def imprime_sudoku(S):
13     for s in S:
14         print(*s)
15
16 # Función auxiliar para expandir la solución parcial fila a fila
17 def avanza(fila,col):
18     if col==8:
19         return (fila+1,0)
20     else:
21         return (fila,col+1)
22
23 S = lee_sudoku('sudoku01_input.txt')
24 if(resuelve_sudoku(0,0,S)):
25     imprime_sudoku(S)
```

Fichero de entrada (sudoku01_input.txt)

```
0 6 0 1 0 4 0 5 0
0 0 8 3 0 5 6 0 0
2 0 0 0 0 0 0 0 1
8 0 0 4 0 7 0 0 6
0 0 6 0 0 0 3 0 0
7 0 0 9 0 1 0 0 4
5 0 0 0 0 0 0 0 2
0 0 7 2 0 6 9 0 0
0 4 0 5 0 8 0 7 0
```

Implementación – II

```
1  def resuelve_sudoku(i,j,S):
2      if i==9:  # Comprueba si la solución parcial está completa
3          return True  # Solución encontrada
4      else:
5          # Comprueba si S[i][j] contiene un número inicial fijo
6          if S[i][j]!=0:
7              (i_nueva,j_nueva) = avanza(i,j)
8              return resuelve_sudoku(i_nueva,j_nueva,S)
9          else:
10             sol_encontrada = False
11             k=1
12             # Genera candidatos
13             while k<10 and not sol_encontrada:
14                 # Comprueba validez del candidato
15                 if es_candidato_valido(i,j,k,S):  # ¡¡INEFICIENTE!!
16                     # Incluye dígito en celda (i,j)
17                     S[i][j] = k
18
19                     (i_nueva,j_nueva) = avanza(i,j)
20                     sol_encontrada =  resuelve_sudoku(i_nueva,j_nueva,S)
21
22                 k=k+1
23
24             # Si no hay solución se marca la celda (i,j) como vacía
25             if not sol_encontrada:
26                 S[i][j] = 0
27
28             # Devolver si se ha encontrado una solución
29             return sol_encontrada
```

Implementación – III

```
1  # Comprueba si el dígito en la celda (fila,col) is válido
2  def es_candidato_valido(fila,col,digito,S):
3      # Comprueba conflicto en fila
4      for k in range(0,9):
5          if k!=col and digito==S[fila][k]:
6              return False
7
8      # Comprueba conflicto en columna
9      for k in range(0,9):
10         if k!=fila and digito==S[k][col]:
11             return False
12
13     # Comprueba conflicto en caja
14     caja_fila= math.floor(fila/3)
15     caja_col = math.floor(col/3)
16     for k in range(0,3):
17         for m in range(0,3):
18             if fila!=3*caja_fila+k and col!=3*caja_col+m:
19                 if digito==S[3*caja_fila+k][3*caja_col+m]:
20                     return False
21
22     return True
```

Sudoku – Aceleración

- Matrices booleanas para detectar conflictos en $\Theta(1)$

Índice de fila / Dígito

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

Índice de columna / Dígito

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

Índice de caja / Dígito

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

Sudoku inicial

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

Caja 0	Caja 1	Caja 2
Caja 3	Caja 4	Caja 5
Caja 6	Caja 7	Caja 8

- Ejercicio para casa/práctica

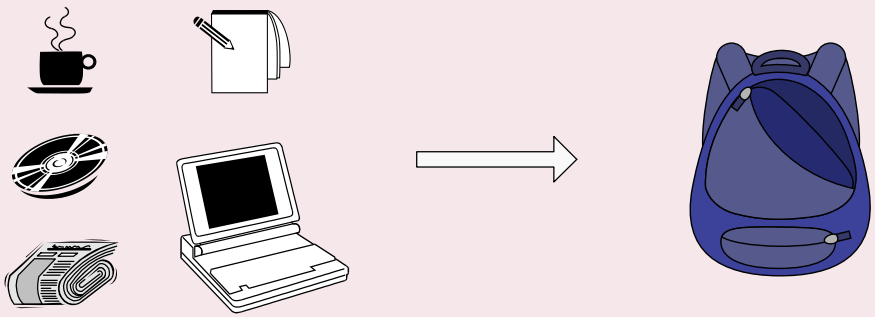
Problema de la mochila 0-1

Problema de la mochila 0-1

Dado un conjunto de n objetos, cada uno con un peso p_i y un valor v_i, $i = 1, \ldots, n$, y una mochila con capacidad C. Maximizar la suma de los valores asociados a los objetos que se introducen en la mochila, sin sobrepasar la capacidad C, sabiendo que los objetos **NO** pueden partirse en fracciones más pequeñas:

$$\underset{x}{\textit{maximizar}} \quad \sum_{i=1}^{n} x_i v_i \qquad \text{sujeto a} \quad x_i \in \{0,1\} \qquad i = 1, \ldots, n$$

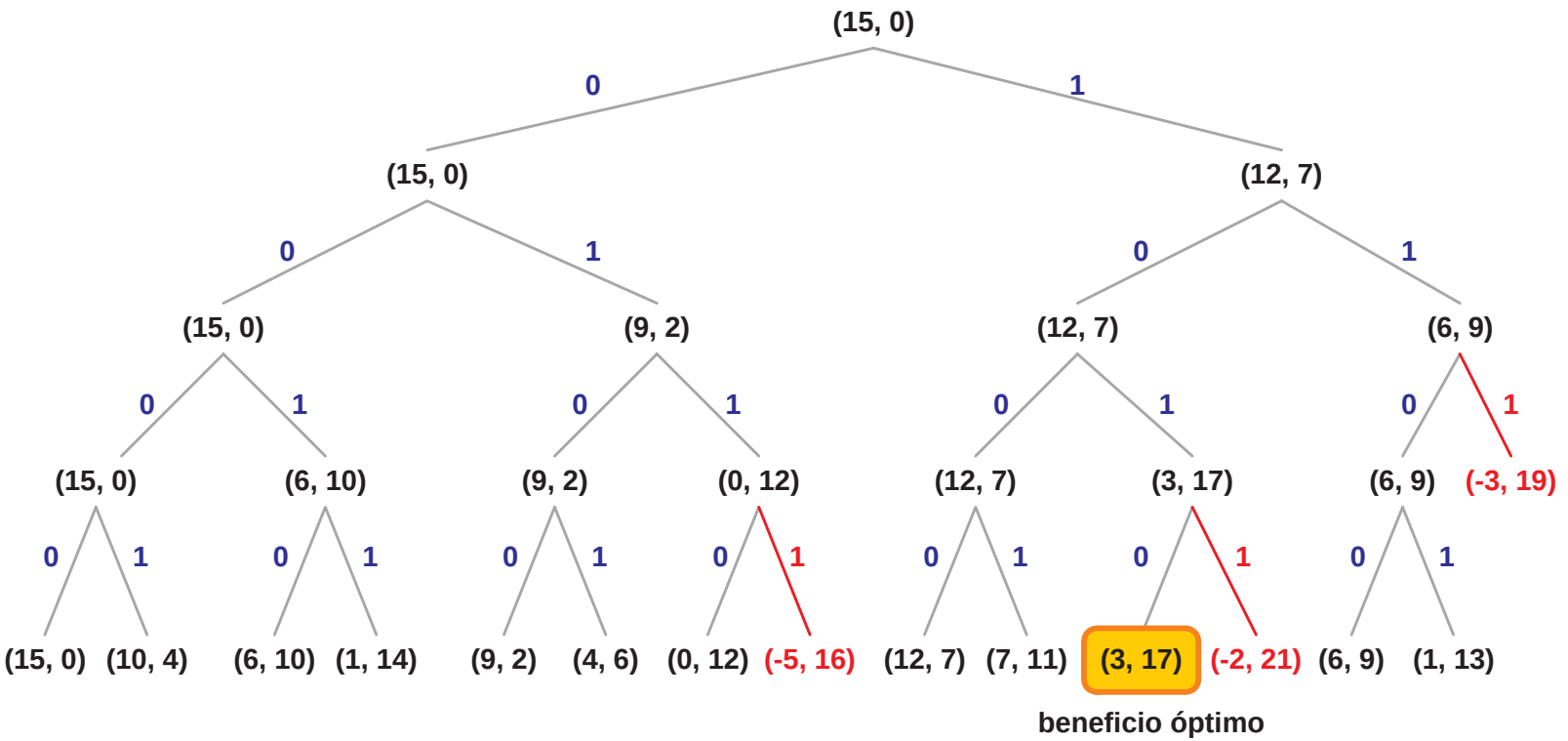
$$\sum_{i=1}^{n} x_i p_i \leqslant C$$

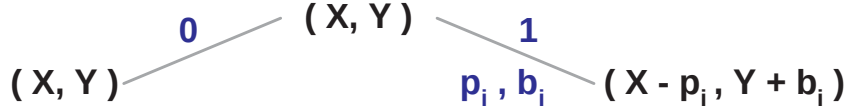


Mochila 0-1

Árbol de búsqueda

- pesos $= (3, 6, 9, 5)$, beneficios $= (7, 2, 10, 4)$, capacidad $= 15$





(X, Y) = (capacidad restante, beneficio parcial)

(X, Y) capacidad sobrepasada

Implementación – I

```
1  def mochila_01_wrapper(p,v,C):
2      sol = [0] * (len(p))
3      sol_opt = [0] * (len(p))
4
5      v_total = mochila_01(0,C,0,sol,sol_opt,-1,p,v,C)
6
7      return v_total
8
9
10 p = [3,6,9,5]   # pesos
11 v = [7,2,10,4]   # valores
12 C = 15   # capacidad
13 print(mochila_01_wrapper(p,v,C))
```

Implementación – II

```
1  def mochila_01(i,C_restante,v_actual,sol,sol_opt,v_opt,p,v,C):
2      if i==len(sol):  # Se han analizado todos los objetos
3          if v_actual>v_opt:  # Comprueba si se ha encontrado una solución mejor
4              # Actualiza el mejor valor y solución
5              v_opt = v_actual
6              for k in range(0,len(sol)):
7                  sol_opt[k] = sol[k]
8      else:
9          for k in range(0,2):  #  Genera candidatos
10
11             # Comprueba si se puede podar el árbol de recursión
12             if k*p[i]<=C_restante:
13
14                 # Expande la solución parcial
15                 sol[i]=k
16
17                 # Actualiza la capacidad restante y el valor parcial
18                 nueva_C_restante = C_restante - k*p[i]
19                 nuevo_v_actual = v_actual + k*v[i]
20
21                 # Intenta expandir la solución parcial
22                 v_opt = mochila_01(i+1,nueva_C_restante, \
23                     nuevo_v_actual,sol,sol_opt,v_opt,p,v,C)
24
25     # devuelve el mejor valor encontrado hasta el momento
26     return v_opt
```

Introducción a la ramificación y poda

Ramificación y poda

- Hasta ahora podábamos el árbol de recursión si una solución parcial no cumplía los requisitos del problema para llegar a ser una solución completa

- En **problemas de optimización** se pueden realizar podas adicionales:

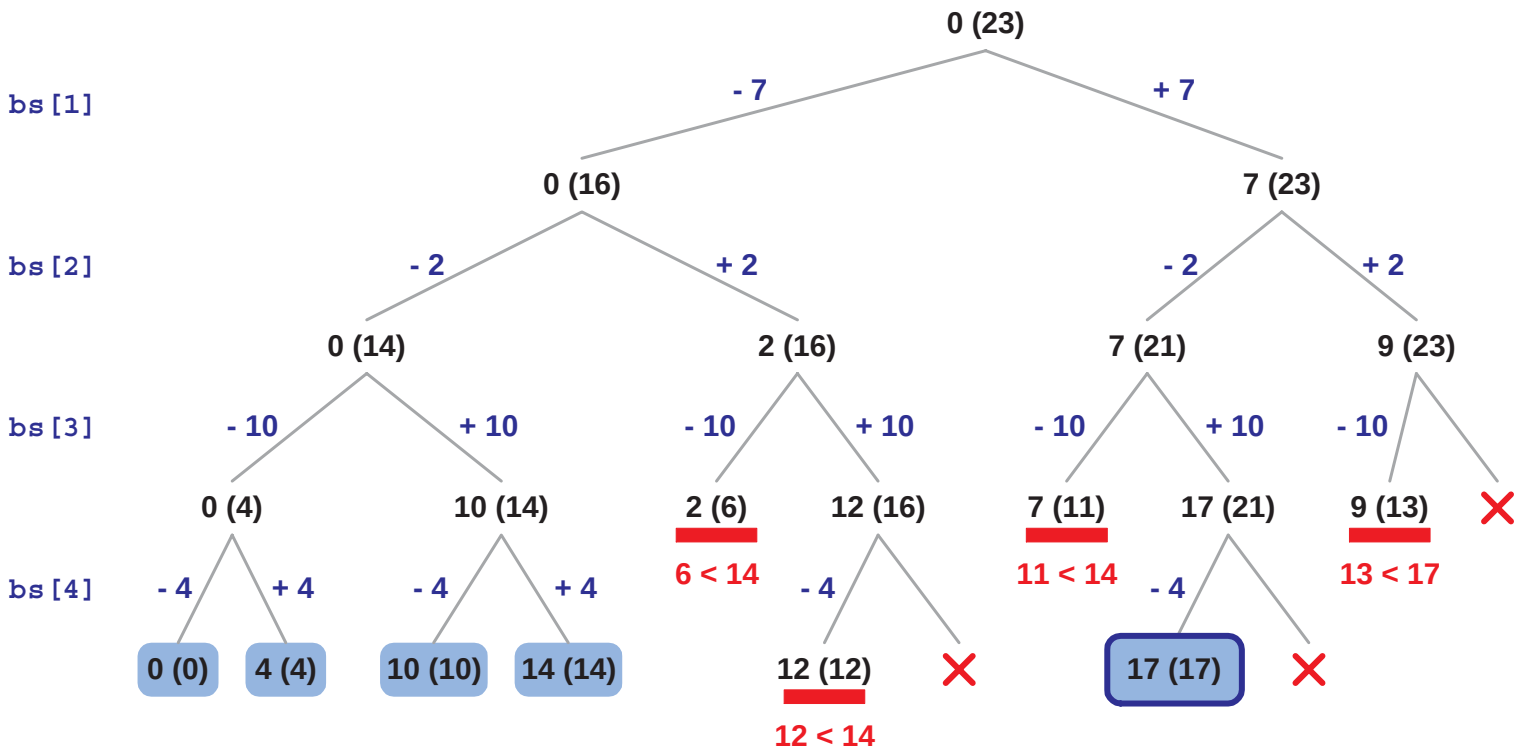
  Habiendo conseguido hallar una solución válida con valor óptimo v, podemos descartar soluciones que nunca puedan alcanzar dicho valor óptimo

- Es un método exacto de resolución de problemas de optimización

- Se aplica a problemas duros (desde un punto de vista computacional), para acelerar la búsqueda del óptimo global

Ramificación y poda: Mochila 0-1

- pesos $= (3, 6, 9, 5)$, beneficios $= (7, 2, 10, 4)$, capacidad $= 15$



Implementación – I

```
1  def mochila_01_bnb_wrapper(p,v,C):
2      sol = [0] * (len(p))
3      sol_opt = [0] * (len(p))
4
5      v_total = mochila_01_bnb(0,C,0,sum(v),sol,sol_opt,-1,p,v,C)
6
7      return v_total
8
9
10 p = [3,6,9,5]   # pesos
11 v = [7,2,10,4]   # valores
12 C = 15   # capacidad
13 print(mochila_01_bnb_wrapper(p,v,C))
```

Implementación – II

```
1  def mochila_01_bnb(i,C_restante,v_actual,max_v,sol,sol_opt,v_opt,p,v,C):
2      if i==len(sol):  # Se han analizado todos los objetos
3          if v_actual>v_opt:  # Comprueba si se ha encontrado una solución mejor
4              # Actualiza el mejor valor y solución
5              v_opt = v_actual
6              for k in range(0,len(sol)):
7                  sol_opt[k] = sol[k]
8      else:
9          for k in range(0,2):  #  Genera candidatos
10
11             # Comprueba si se puede podar el árbol de recursión por capacidad
12             if k*p[i]<=C_restante:
13
14                 # Actualiza el máximo valor alcanzable
15                 nuevo_max_v = max_v - (1-k)*v[i]
16
17                 # Comprueba si se puede podar el árbol de recursión por valor
18                 if nuevo_max_v>v_opt:
19
20                     sol[i]=k  # Expande la solución parcial
21
22                     # Actualiza la capacidad restante y el valor parcial
23                     nueva_C_restante = C_restante - k*p[i]
24                     nuevo_v_actual = v_actual + k*v[i]
25
26                     # Intenta expandir la solución parcial
27                     v_opt = mochila_01_bnb(i+1,nueva_C_restante, \
28                         nuevo_v_actual,nuevo_max_v,sol,sol_opt,v_opt,p,v,C)
29
30     return v_opt  # devuelve el mejor valor encontrado hasta el momento
```

Aspectos a considerar al usar backtracking

- ¿Las soluciones son permutaciones, subconjuntos u otra entidad?
  - Si son permutaciones o subconjuntos: usad los esquemas vistos. Son claros y eficientes
  - Estructura de la solución parcial
- ¿Se pueden acelerar cálculos usando estructuras de datos?
- ¿Se busca una solución o todas?
  - Si solo se busca una, paramos en cuanto la encontremos (`while` + variable booleana (`solucion_encontrada`))
  - En problemas de optimización se busca solo una (la mejor), pero se analizan todas
- En problemas de optimización
  - Añadimos dos parámetros: mejor solución y mejor valor
  - ¿Se puede podar el árbol de recursión por valor?

Secuencias de paréntesis válidas – I

```
1  def parentesis(i,n_abiertos,sol,n):
2      if i==2*n:
3          return 1
4      else:
5          nsols = 0
6
7          for k in range(0,2):
8              n_abiertos = n_abiertos + (1-k)
9
10             if n_abiertos>=i-n_abiertos+1 and n_abiertos<=n:
11                 sol[i]=k
12                 nsols = nsols + parentesis(i+1,n_abiertos,sol,n)
13
14             n_abiertos = n_abiertos - (1-k)
15
16         return nsols
17
18
19 def parentesis_wrapper(n):
20     sol = [None] * (2*n)
21     return parentesis(0,0,sol,n)
```

Secuencias de paréntesis válidas – II

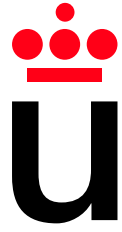
```
1  def parentesis_alt(i,sol,n):
2      if i==n:
3          # print(sol)
4
5          return 1
6      else:
7          nsols = 0
8
9          if i==0:
10             k_ini = 0
11         else:
12             k_ini = sol[i-1]+1
13
14         for k in range(k_ini,2*i+1):
15             sol[i]=k
16             nsols = nsols + parentesis_alt(i+1,sol,n)
17
18         return nsols
19
20
21 def parentesis_wrapper_alt(n):
22     sol = [None] * n
23     return parentesis_alt(0,sol,n)
```

Tema 7
Algortimos voraces

Diseño y Análisis de Algoritmos

Manuel Rubio Sánchez

13 de mayo de 2024



Universidad
Rey Juan Carlos

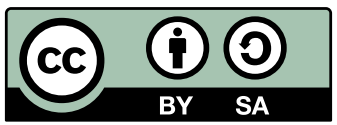
Copyright

©2024 Manuel Rubio Sánchez

Algunos derechos reservados.

Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en:

https://creativecommons.org/licenses/by-sa/4.0/deed.es.



Contenido

Introducción

Algoritmos voraces

- Los algoritmos voraces se suelen aplicar a problemas de optimización
  - Maximizar o minimizar una función objetivo
- Suelen ser rápidos y fáciles de implementar
- Exploran soluciones "locales"
- No siempre garantizan la solución óptima

Algoritmos voraces

- Son algoritmos que siguen una heurística mediante la cual toman **decisiones óptimas locales** en cada paso, de manera muy eficiente, con la esperanza de poder encontrar un óptimo global tras una serie de pasos
- Se aplican, sobre todo, a problemas duros, desde un punto de vista computacional
  - Ejemplo: problema del viajante (NP-completo)
  - Heurística: "escoge la ciudad más próxima no visitada aún"
- Para ciertos problemas se puede demostrar que algunas estrategias voraces sí que logran hallar un óptimo global de manera eficiente

Ventajas y desventajas

- Ventajas
  - Son fáciles de implementar
  - Producen soluciones eficientes
  - A veces encuentran la solución óptima
- Desventajas
  - No todos los problemas de optimización son resolubles con algoritmos voraces
  - La búsqueda de un óptimo local no implica encontrar un óptimo global
  - Dificultad de encontrar la función de selección que garantice la elección óptima

Elementos

- **Conjunto de candidatos** C: la solución se construirá con un subconjunto de estos candidatos
- **Función de selección**: selecciona el candidato "local" más idóneo
- **Función de factibilidad**: comprueba si un candidato es factible
- **Función objetivo**: determina el valor de la solución (función a optimizar)
- **Función solución**: determina si el subconjunto de candidatos ha alcanzado una solución

Esquema general

- La técnica voraz funciona por pasos:
  - Partimos de una solución vacía y de un conjunto de candidatos a formar parte de la solución
  - En cada paso se intenta añadir el mejor de los candidatos restantes a la solución parcial
  - Una vez tomada la decisión, no se puede deshacer
  - Si la solución ampliada es válida $\Rightarrow$ candidato incorporado
  - Si la solución ampliada no es válida $\Rightarrow$ candidato desechado
- El algoritmo acabará cuando el conjunto de elementos seleccionados constituya una solución o cuando no queden elementos sin considerar

Esquema general en pseudocódigo

función voraz(C) //C es el conjunto de candidatos//

$S \leftarrow \emptyset$
mientras $C \neq \emptyset$ y no solución(S) hacer
    $x \leftarrow$ seleccionar(C)
    $C \leftarrow C \backslash \{x\}$
    si factible($S \cup \{x\}$) entonces
        $S \leftarrow S \cup \{x\}$

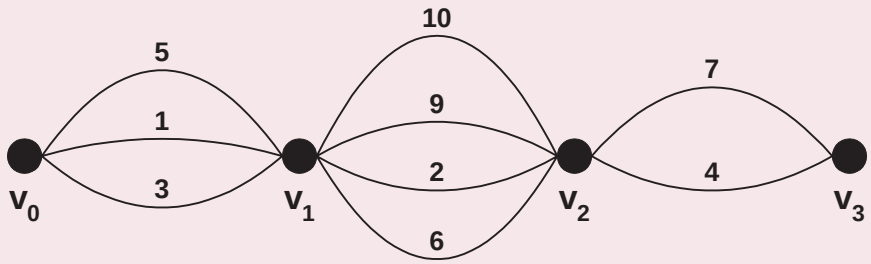
si solución(S) entonces
    devolver S //S es una solución//
si no
    devolver $\emptyset$ //No hay soluciones//

Ejemplos básicos

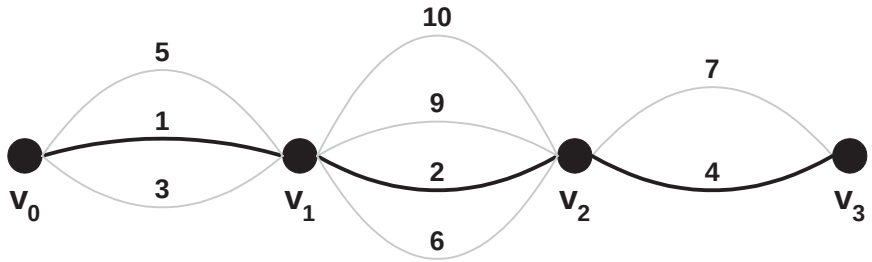
Camino más corto - 1

Camino más corto - 1

Encontrar el camino más corto de v_0 a v_n, donde solo hay caminos entre vértices adyacentes (v_{i-1} y v_i, para $i = 1, \ldots, n$).



- ¿Función de selección?
  - Método voraz: en cada paso se coge el arco de menor longitud

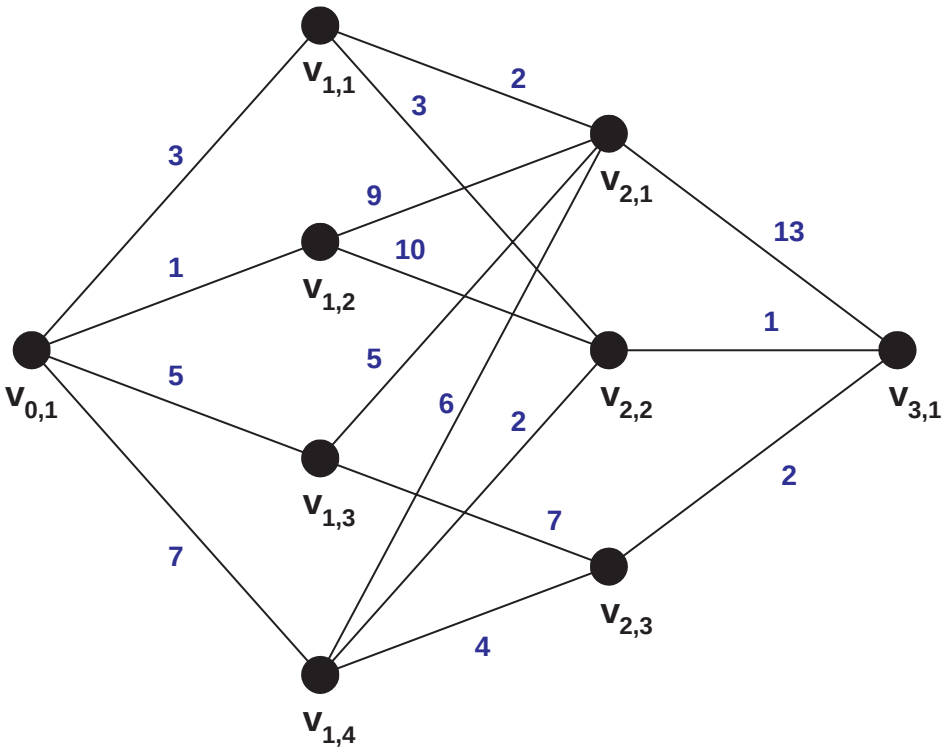


- Solución optima: $1 + 2 + 4 = 7$
- Se demuestra que la estrategia es óptima (por contradicción)

Camino más corto - 2

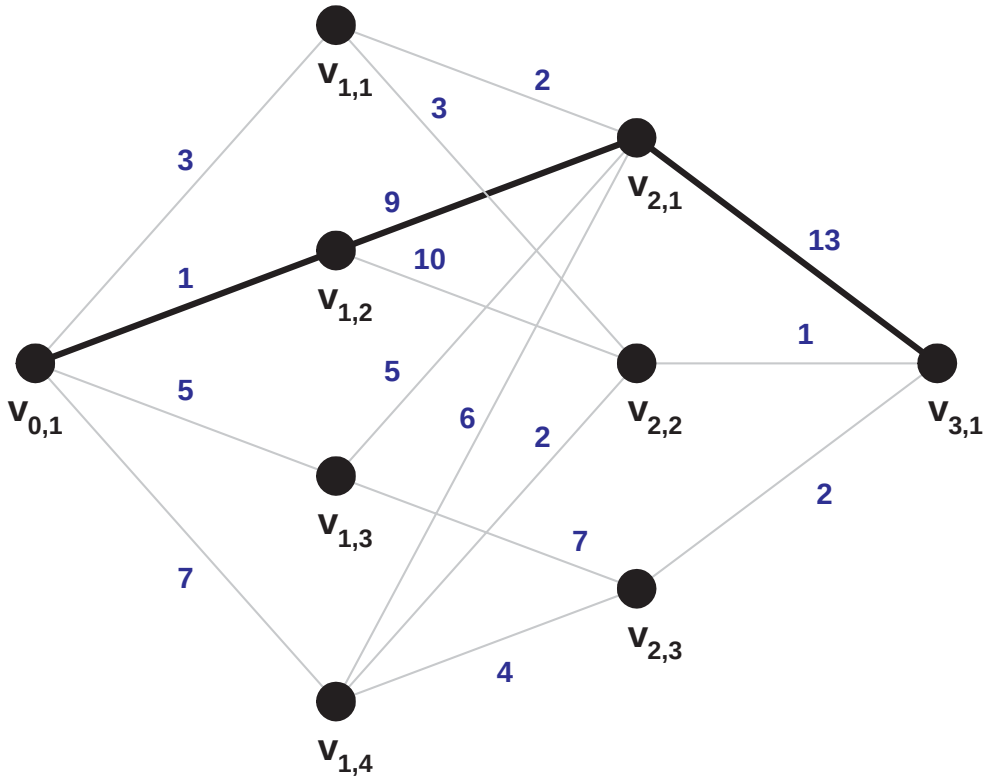
Camino más corto - 2

Encontrar el camino más corto de $v_{0,1}$ a $v_{n,1}$, donde solo hay caminos entre vértices de etapas adyacentes ($v_{i-1,j}$ y $v_{i,k}$, para $i = 1, \ldots, n$, y $\forall j, k$, donde puede haber cualquier número de vértices en las etapas 1, 2,. . . ,$n-1$).



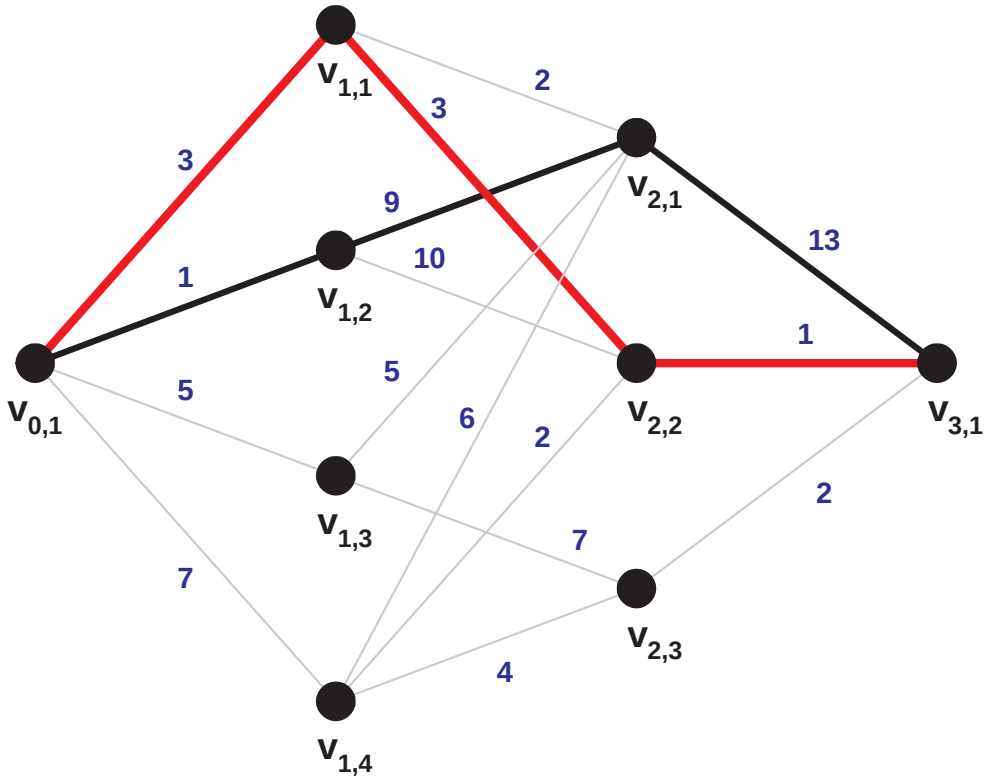
Camino más corto - 2

- ¿Función de selección?
  - Método voraz: en cada paso se coge el arco de menor longitud



- Longitud de la solución voraz: $1 + 9 + 13 = 23$
  $(v_{0,1}, v_{1,2}, v_{2,1}, v_{3,1})$

Camino más corto - 2



- Se demuestra que la estrategia no es óptima (contraejemplo)
- Longitud de la solución optima: $3 + 3 + 1 = 7$ $(v_{0,1}, v_{1,1}, v_{2,2}, v_{3,1})$

Problemas clásicos

Cambio de monedas

Problema del cambio de monedas

Se dispone de n monedas de euro con valores de 1,2,5,10,20 y 50 céntimos de euro, 1 y 2 euros.

- Dada una cantidad X de euros, devolver dicha cantidad con el menor número posible de monedas

- Ejemplo: devolver 2.24€
  - Solución: 4 (2 + 0,20 + 0,02 + 0,02)

Cambio de monedas

- Conjunto de candidatos:
  - Todos los tipos de monedas
- Función solución:
  - Conjunto de monedas que suman X
- Función de factibilidad:
  - Si $\sum_{i=1}^{8} v_i n_i > X$, el conjunto obtenido no podrá ser solución
    - $n_i =$ número de monedas de tipo i
    - $v_i =$ valor de una moneda de tipo i
- Función objetivo:
  - Minimizar la cardinalidad de las soluciones posibles
- Función de selección:
  - Moneda de valor más alto posible, que no supere el valor que queda por devolver

Cambio de monedas

- ¿Es óptima la función de selección propuesta?

Problema del cambio de monedas

Se dispone de n monedas de euro con valores de 1,2,5,10,**12**,20 y 50 céntimos de euro, 1 y 2 euros.

- Dada una cantidad X de euros, devolver dicha cantidad con el menor número posible de monedas

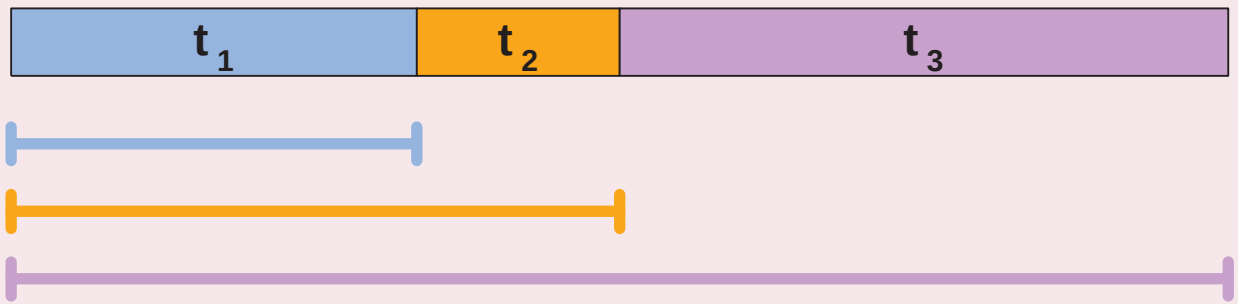
- Ejemplo: devolver 2.24€
  - Solución voraz: 4 $(2 + 0{,}20 + 0{,}02 + 0{,}02)$
  - Solución óptima: 3 $(2 + 0{,}12 + 0{,}12)$

Minimización del tiempo en el sistema

Minimización del tiempo en el sistema

Considérese un servidor que tiene que dar servicio a n clientes, donde t_i, con $i = 1, \ldots, n$, es el tiempo requerido por el cliente i. Suponiendo que todos los clientes llegan al mismo tiempo al servidor pero solo uno puede usarlo, minimizar el tiempo T en el sistema para los n clientes:

$$T = \sum_{i=1}^{n} (\text{tiempo en el sistema para el cliente } i)$$



Ejemplo

- Supongamos que tenemos 3 clientes con $t_1 = 5$, $t_2 = 10$, y $t_3 = 3$, hay varias posibilidades según el orden en el que sean tratados en el servidor

Orden	T	
123:	$5 + (5 + 10) + (5 + 10 + 3) = 38$	
132:	$5 + (5 + 3) + (5 + 3 + 10) = 31$	
213:	$10 + (10 + 5) + (10 + 5 + 3) = 43$	$\leftarrow$ peor planificación
231:	$10 + (10 + 3) + (10 + 3 + 5) = 41$	
312:	$3 + (3 + 5) + (3 + 5 + 10) = 29$	$\leftarrow$ mejor planificación
321:	$3 + (3 + 10) + (3 + 10 + 5) = 34$	

Estrategia voraz

- Dar servicio en orden creciente de tiempo t_i
- Es una estrategia óptima:
  - Sea $P = \langle p_1, p_2, \ldots, p_n \rangle$ una permutación de los clientes (de enteros de 1 a n)
  - Sea $s_i = t_{p_i}$, entonces

$$\begin{aligned} T(P) &= s_1 + (s_1 + s_2) + (s_1 + s_2 + s_3) + \ldots \\ &= ns_1 + (n-1)s_2 + (n-2)s_3 + \ldots \\ &= \sum_{k=1}^{n} (n - k + 1)s_k \end{aligned}$$

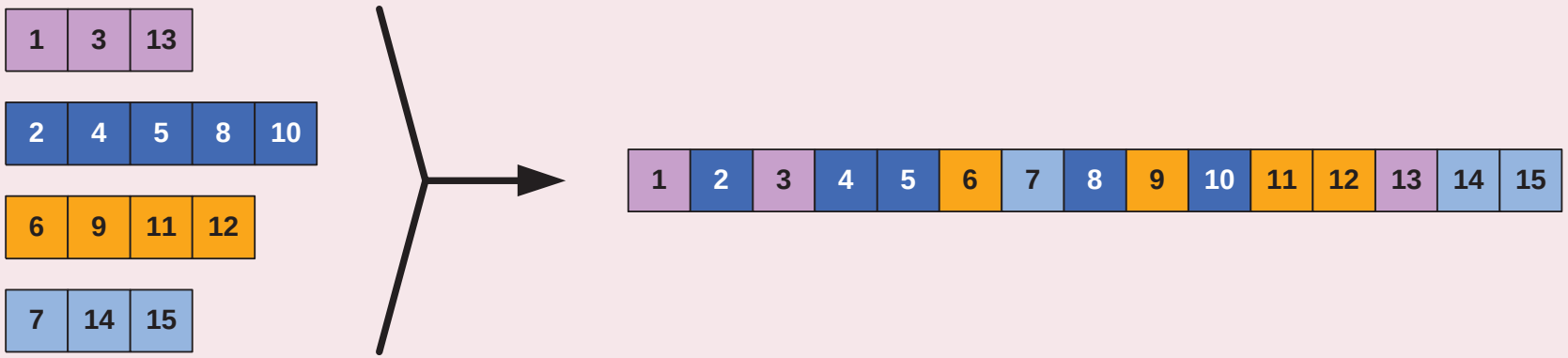
  - Haciendo s_1 lo menor posible, luego s_2, etc. conseguimos minimizar T

Mezcla de listas

Mezcla de listas

Sean m listas ordenadas, cada una con n_i elementos ($i = 1, \ldots, m$)

- ¿Cuál es la sucesión óptima del proceso de mezcla, mezclando listas dos a dos, para mezclar todas las listas de manera ordenada con el menor número de comparaciones?

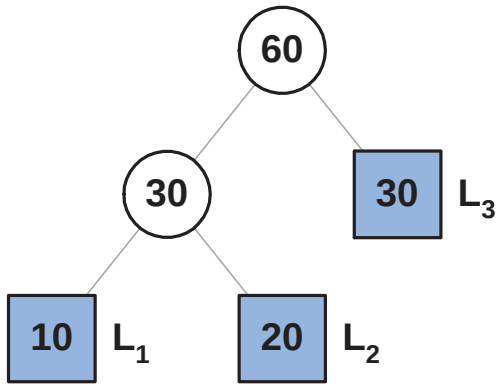


Ejemplo

- Sean tres listas L_1, L_2, y L_3, con longitudes 10, 20, y 30, respectivamente

$L_1 + L_2 = 30$	$L_1 + L_3 = 40$	$L_2 + L_3 = 50$
$30 + L_3 = 60$	$40 + L_2 = 60$	$50 + L_1 = 60$
$90 \Leftarrow$	100	110

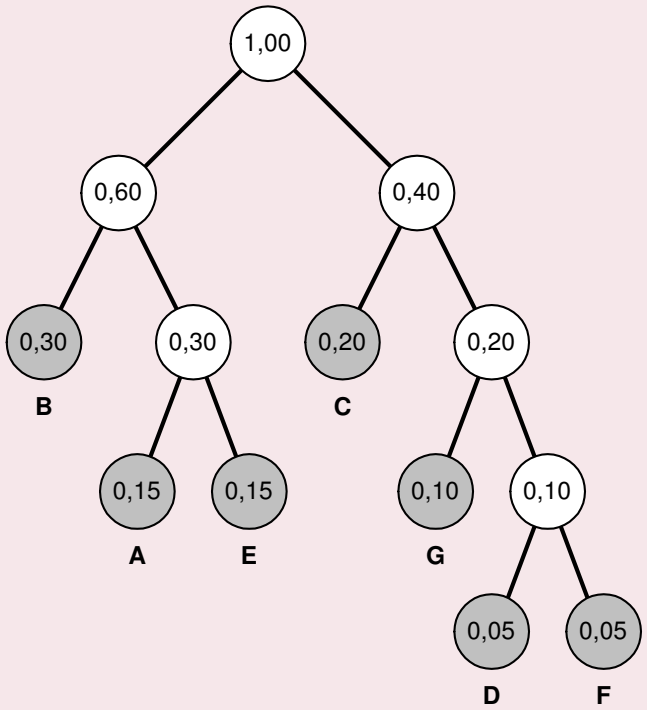
- Solución voraz: escoger en cada momento los dos lotes de menor tamaño
  - Construye un árbol binario de abajo hacia arriba (árbol de mezclas)



Códigos de Huffman

Códigos de Huffman

Dado un conjunto de n objetos y sus respectivas frecuencias de aparición f_i, $i = 1, \ldots, n$, obtener una codificación binaria para los objetos que minimice el promedio de bits necesarios para representarlos.



Códigos de Huffman

- La codificación Huffman es un método usado para compresión de datos
- Se codifica una serie de objetos mediante una secuencia de bits según su frecuencia de aparición
- Los objetos con mayor frecuencia se codifican con menos bits
- La codificación Huffman minimiza el promedio de bits necesarios para representar los objetos

Ejemplo

- Tenemos un texto de 100000 caracteres
- Solo aparecen 6 caracteres con las siguientes frecuencias absolutas de aparición

carácter	a	b	c	d	e	f
frecuencia	45000	13000	12000	16000	9000	5000

- Solución 1: Usar 3 bits por cada carácter $\Rightarrow$ 300000 bits
- Solución 2: Usar códigos de longitud variable en el que los más frecuentes tienen el código más corto $\Rightarrow$ 224000 bits
  - **Objetivo**: crear una codificación de longitud variable para minimizar el número de bits, en promedio, necesarios para representar un texto

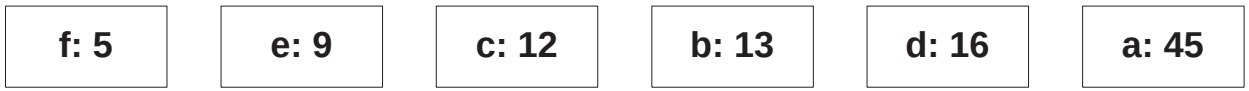
Estrategia voraz

- **Asignar códigos más largos a los caracteres con menor frecuencia de aparición**

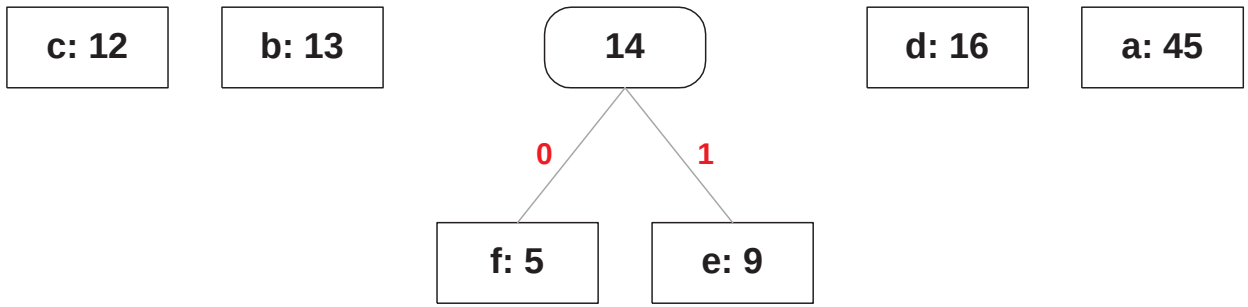
- Utiliza la tabla de frecuencias de aparición de cada carácter
- Construye un árbol binario de longitud variable de abajo hacia arriba
- Utiliza una cola Q de árboles con prioridades
- Inicialmente Q contiene un árbol por cada carácter
- En cada paso, se "mezclan" los dos árboles de Q que tienen menos frecuencia dando lugar a un nuevo árbol

Estrategia voraz

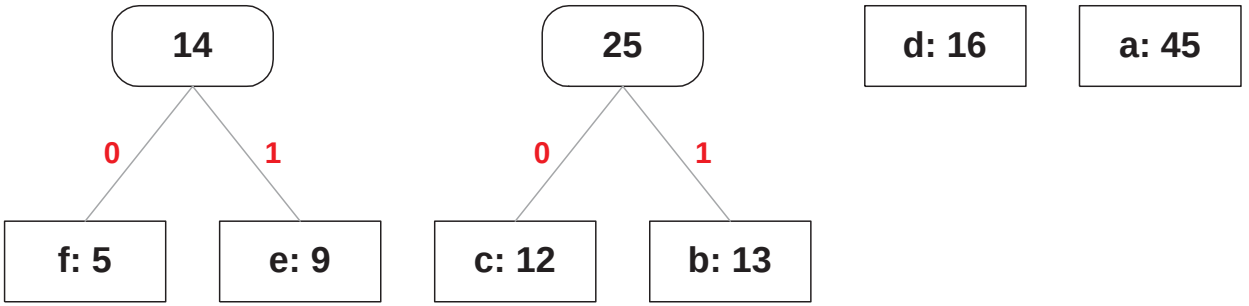
- **Fase 1**
  - Orden creciente de frecuencias (en miles)

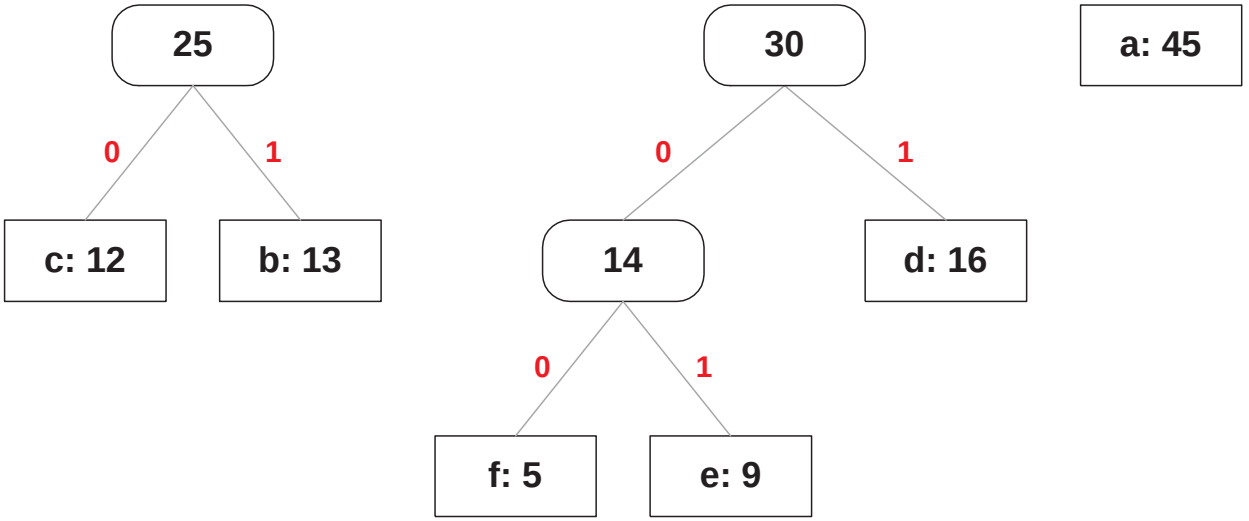


- **Fase 2 y posteriores**
  - Fusionar árboles hasta obtener un sólo árbol manteniendo la ordenación creciente

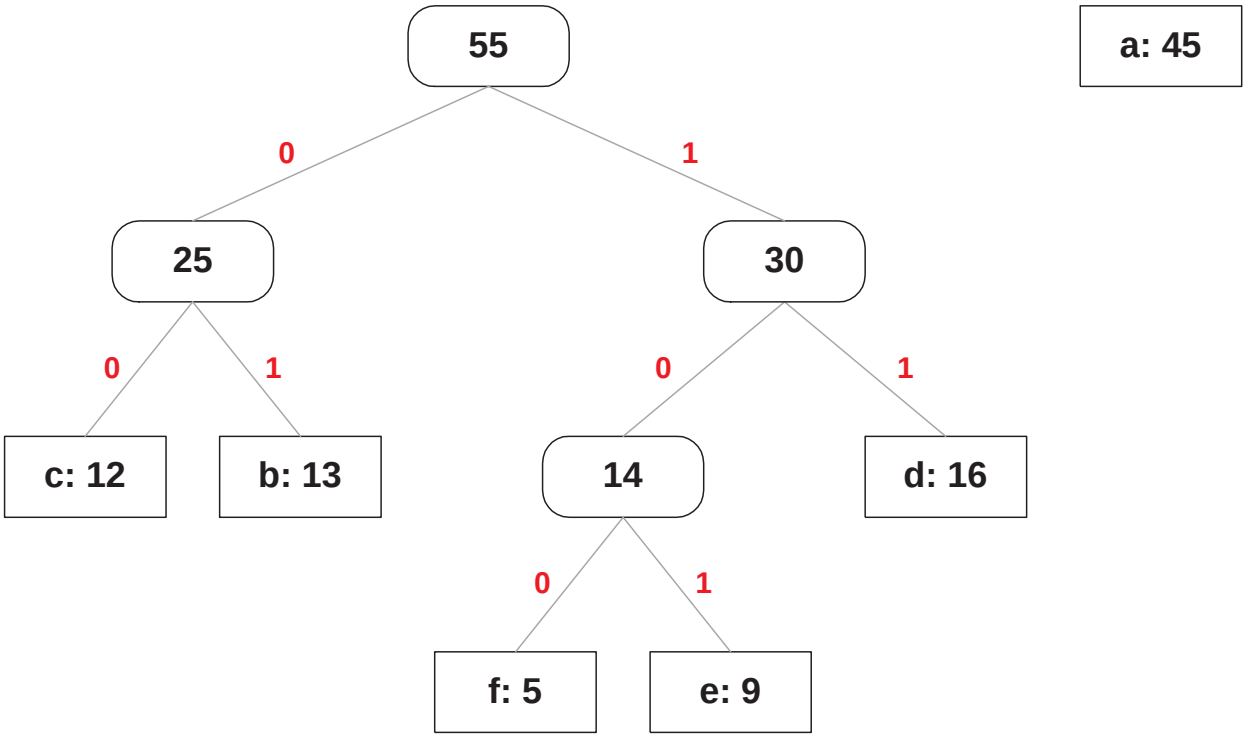


Estrategia voraz

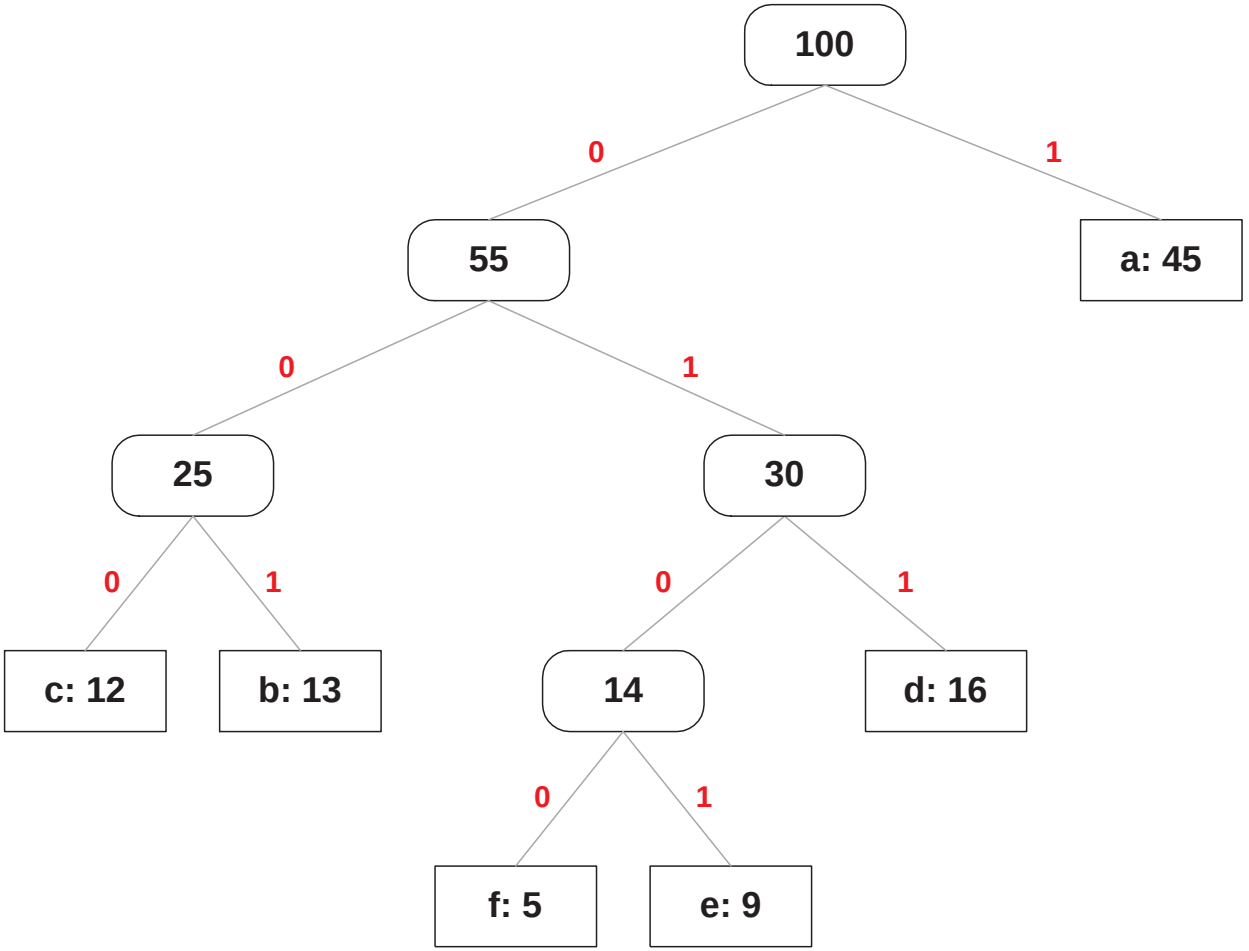




Estrategia voraz



Estrategia voraz



Codificación final

- Para hallar la codificación de un carácter se usa el camino desde la raíz del árbol hasta la hoja que representa el carácter
- La secuencia de bits la determina la rama (izquierda: 0, derecha: 1) por la que se avanza hasta la hoja
- Codificación final:

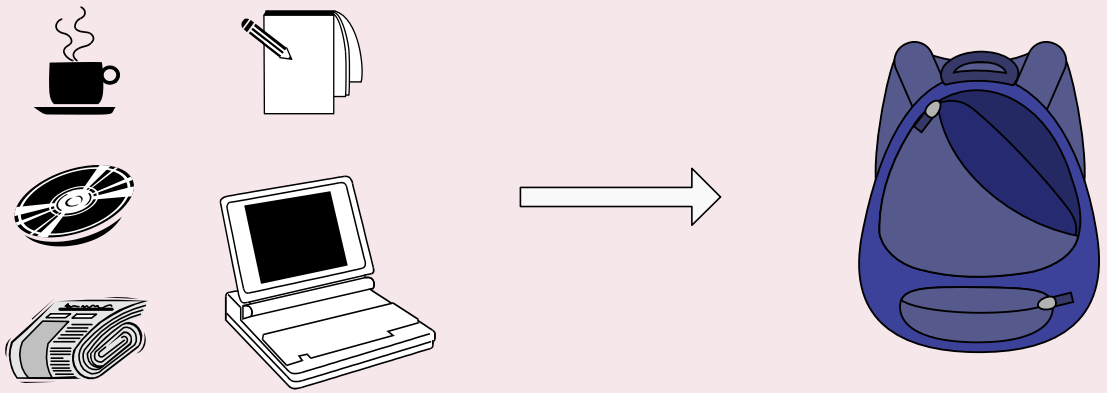
carácter	código
a	1
d	011
e	0101
f	0100
b	001
c	000

Problema de la mochila

Problema de la mochila – 1

Problema de la mochila – Versión 1

Se tiene un conjunto de n objetos, cada uno con un peso p_i, y una mochila con capacidad C.



- Maximizar el **número de objetos** que se pueden introducir en la mochila sin sobrepasar la capacidad C

Problema de la mochila – 1

Problema de la mochila – Versión 1 – Formulación matemática

Dado un conjunto de n objetos, cada uno con un peso p_i, $i = 1, \ldots, n$, y una mochila con capacidad C. Maximizar el **número de objetos** que se pueden introducir en la mochila sin sobrepasar la capacidad C:

$$\begin{array}{ll} \underset{\mathbf{x}}{\textit{maximizar}} & \displaystyle\sum_{i=1}^{n} x_i \\ \text{sujeto a} & x_i \in \{0,1\} \qquad i = 1, \ldots, n \\ & \displaystyle\sum_{i=1}^{n} x_i p_i \leqslant C \end{array}$$

- Las variables x_i determinan si se introduce el objeto i en la mochila
- Es un problema de programación lineal entera

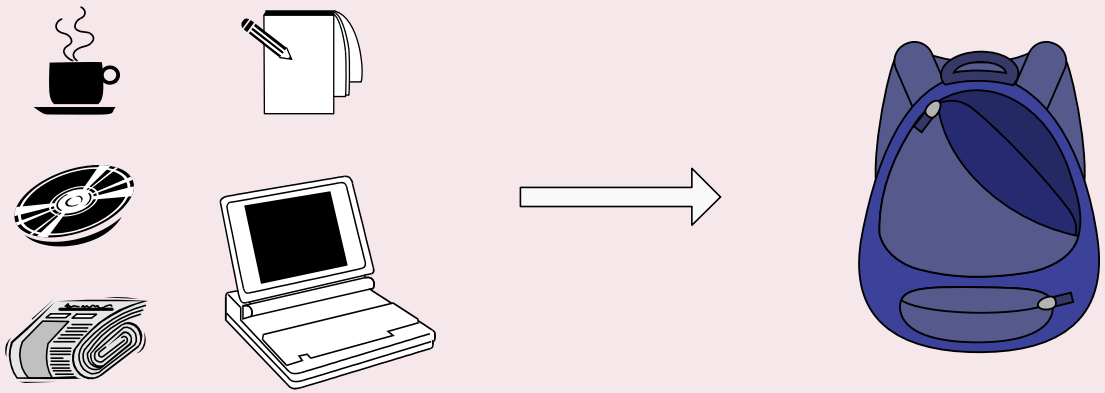
Problema de la mochila – 1

- Ejemplo: $C = 15$, $p = (9, 6, 5)$
- ¿Función de selección?
  - Peso decreciente
    - Solución: 2 $(9 + 6)$
  - Peso creciente
    - Solución: 2 $(5 + 6)$
- Ejemplo: $C = 15$, $p = (9, 5, 6, 4)$
- ¿Función de selección?
  - Peso decreciente
    - Solución: 2 $(9 + 6)$
  - Peso creciente
    - Solución: 3 $(4 + 5 + 6)$
- La estrategia voraz escogiendo candidatos en orden creciente de peso es mejor (para este problema es óptima)

Problema de la mochila – 2

Problema de la mochila – Versión 2

Se tiene un conjunto de n objetos, cada uno con un peso p_i, y una mochila con capacidad C.



- Maximizar el **peso de los objetos** que se introducen en la mochila sin sobrepasar la capacidad C

Problema de la mochila – 2

Problema de la mochila - Versión 2 - Formulación matemática

Dado un conjunto de n objetos, cada uno con un peso p_i, $i = 1, \ldots, n$, y una mochila con capacidad C. Maximizar el **peso de los objetos** que se introducen en la mochila sin sobrepasar la capacidad C:

$$\begin{array}{ll} \underset{\mathbf{x}}{\textit{maximizar}} & \displaystyle\sum_{i=1}^{n} x_i p_i \\ \text{sujeto a} & x_i \in \{0,1\} \qquad i = 1, \ldots, n \\ & \displaystyle\sum_{i=1}^{n} x_i p_i \leqslant C \end{array}$$

- Las variables x_i determinan si se introduce el objeto i en la mochila
- Es un problema de programación lineal entera

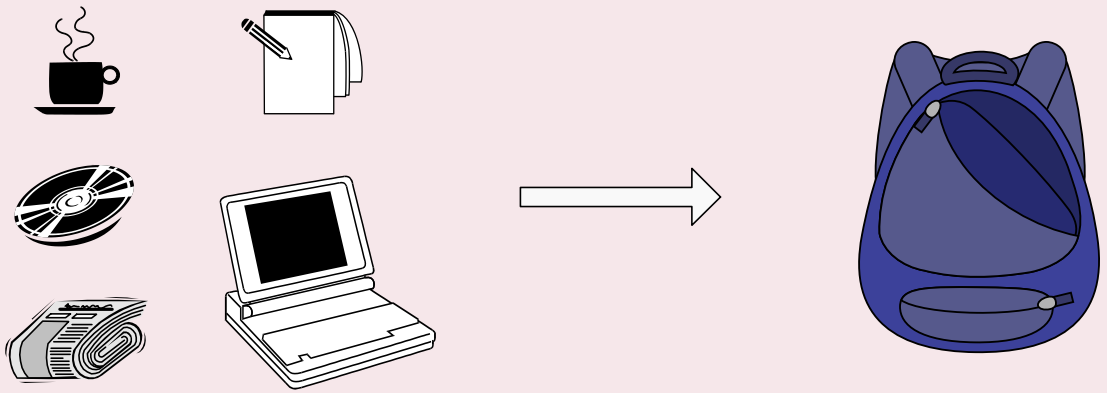
Problema de la mochila – 2

- Ejemplo: $C = 15$, $p = (9, 6, 5)$
- ¿Función de selección?
  - Peso decreciente
    - Solución: 15 $(9 + 6)$
  - Peso creciente
    - Solución: 11 $(5 + 6)$
- Ejemplo: $C = 15$, $p = (10, 7, 6)$
- ¿Función de selección?
  - Peso decreciente
    - Solución: 10 (10)
  - Peso creciente
    - Solución: 13 $(6 + 7)$
- Ninguna de las estrategias de selección es óptima, ni mejor que la otra

Problema de la mochila – 3

Problema de la mochila – Versión 3

Se tiene un conjunto de n objetos, cada uno con un peso p_i y un valor v_i, y una mochila con capacidad C. Los objetos pueden partirse en fracciones más pequeñas.



- Maximizar la **suma de los valores asociados a los objetos** que se introducen en la mochila, sin sobrepasar la capacidad C

Problema de la mochila – 3

Problema de la mochila – Versión 3 – Formulación matemática

Dado un conjunto de n objetos, cada uno con un peso p_i y un valor v_i, $i = 1, \ldots, n$, y una mochila con capacidad C. Maximizar la suma de los valores asociados a los objetos que se introducen en la mochila, sin sobrepasar la capacidad C, sabiendo que los objetos pueden partirse en fracciones más pequeñas:

$$\begin{array}{ll} \underset{\mathbf{x}}{\textit{maximizar}} & \displaystyle\sum_{i=1}^{n} x_i v_i \\ \text{sujeto a} & 0 \leqslant x_i \leqslant 1 \qquad i = 1, \ldots, n \\ & \displaystyle\sum_{i=1}^{n} x_i p_i \leqslant C \end{array}$$

- Las variables x_i determinan la fracción del objeto i que se introduce en la mochila
- Es un problema de programación lineal

Problema de la mochila – 3

- Conjunto de candidatos:
  - Todos los objetos
- Función de factibilidad:
  - $\sum_{i=1}^{n} x_i p_i \leqslant C$
- Función objetivo:
  - Maximizar $\sum_{i=1}^{n} x_i v_i$
- Funciones de selección posibles:
  - Seleccionar el objeto con mayor valor
  - Seleccionar el objeto con menor peso restante
  - Seleccionar el objeto cuyo valor por unidad de peso sea el mayor posible
- Función solución:
  - Cualquier conjunto de elementos es válido si no se ha sobrepasado C

Problema de la mochila – 3

- Ejemplo: $C = 100$, $n = 5$, con:

i	1	2	3	4	5
p_i	10	20	30	40	50
v_i	20	30	66	40	60
v_i/p_i	2,0	1,5	2,2	1,0	1,2

seleccionar x_i	x_i					valor total
maximizar v_i	0	0	1	0,5	1	146
minimizar p_i	1	1	1	1	0	156
maximizar v_i/p_i	1	1	1	0	0,8	164

- Función de selección óptima es: maximizar v_i/p_i

Problema de la mochila – 4

Problema de la mochila – Versión 4 – Formulación matemática

Dado un conjunto de n objetos, cada uno con un peso p_i y un valor v_i, $i = 1, \ldots, n$, y una mochila con capacidad C. Maximizar la suma de los valores asociados a los objetos que se introducen en la mochila, sin sobrepasar la capacidad C, sabiendo que los objetos **NO** pueden partirse en fracciones más pequeñas:

$$\begin{array}{ll} \underset{\mathbf{x}}{\textit{maximizar}} & \displaystyle\sum_{i=1}^{n} x_i v_i \\ \text{sujeto a} & x_i \in \{0,1\} \qquad i = 1, \ldots, n \\ & \displaystyle\sum_{i=1}^{n} x_i p_i \leqslant C \end{array}$$

- Las variables x_i determinan si se introduce el objeto i en la mochila
- Es un problema NP-completo (Problema de la mochila 0-1)

Problemas sobre grafos

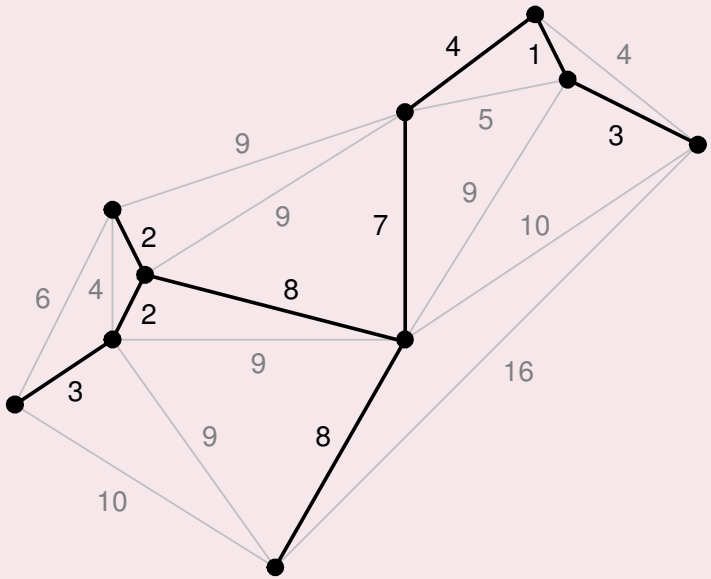
Árbol de recubrimiento de coste mínimo

- Árbol libre: es un grafo no dirigido conexo acíclico:
  - Todo árbol libre con n vértices tiene n-1 aristas
  - Si se añade una arista se introduce un ciclo
  - Si se borra una arista quedan vértices no conectados
  - Cualquier par de vértices está unido por un único camino simple
- Árbol de recubrimiento de un grafo no dirigido y ponderado no negativamente: es cualquier subgrafo que contenga todos los vértices y que sea un árbol libre
- **Árbol de recubrimiento de coste mínimo**: es un árbol de recubrimiento y no hay ningún otro árbol de recubrimiento cuya suma de los pesos de las aristas sea menor. Lo denotamos $arm(G)$, donde G es un grafo no dirigido, conexo, y ponderado no negativamente.

Árbol de recubrimiento de coste mínimo

Árbol de recubrimiento de coste mínimo

Dado un grafo $G = \langle V, E \rangle$ no dirigido, conexo, y ponderado no negativamente, hallar $arm(G)$.



- Valor 38 en la figura
- No tiene por qué ser único

Propiedad fundamental

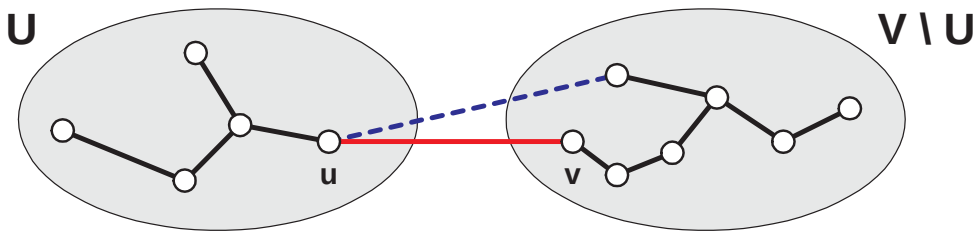
- Sea $G = \langle V, E \rangle$ un grafo no dirigido, conexo, y ponderado no negativamente,

$$G \in \{f : V \times V \longrightarrow E\}$$

- Sea U un conjunto de vértices $U \subset V$, $U \neq \emptyset$

  Si $\langle u, v \rangle$ es la arista más pequeña de G tal que $u \in U$, y $v \in V \backslash U$, entonces existe algún árbol de recubrimiento de coste mínimo de G que la contiene

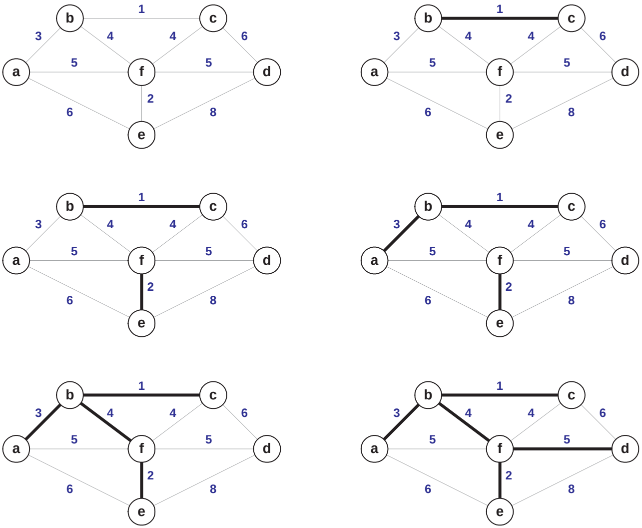
  - Demostración por reducción al absurdo (contradicción)



Algoritmo de Kruskal

- Seleccionar la arista más corta en cada etapa y construir el árbol
- Se basa en la propiedad de los árboles de recubrimiento de coste mínimo: partiendo del árbol vacío, se selecciona en cada paso la arista de menor peso que no provoque ciclo sin requerir ninguna otra condición sobre sus extremos
- Es una estrategia óptima

Algoritmo de Kruskal paso a paso



Algoritmo de Kruskal - Pseudocódigo

Kruskal(G)
    // Entrada: $G = \langle V, E \rangle$, es un grafo no dirigido, conexo,
        // y ponderado no negativamente
    // Salida: E_T = conjunto de aristas que forman un $arm(G)$

    ordenar E en orden no decreciente según los pesos asociados a las aristas:
        $w(e_{i_1}) \leqslant \ldots \leqslant w(e_{i_{|E|}})$
    $E_T \leftarrow \varnothing$
    $cont \leftarrow 0$ // nº de aristas de $arm(G)$
    $k \leftarrow 0$ // nº de aristas procesadas

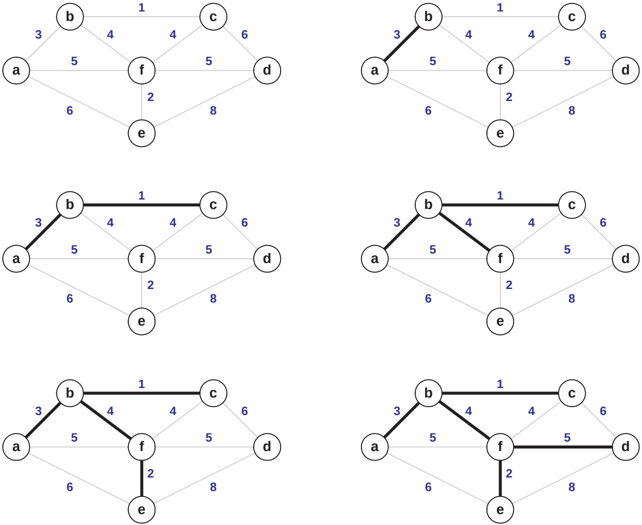
    while $cont < |V| - 1$
        $k \leftarrow k + 1$
        si $E_T \cup \{e_{i_k}\}$ no contiene ciclos
            $E_T \leftarrow E_T \cup \{e_{i_k}\}$
            $cont \leftarrow cont + 1$

    devolver E_T

Algoritmo de Prim

- Seleccionar un nodo y construir el árbol a partir de él
- Aplica reiteradamente la propiedad de los árboles de recubrimiento de coste mínimo incorporando a cada paso una arista
- Se usa un conjunto U de vértices tratados y se selecciona en cada paso la arista mínima que une un vértice de U con otro de su complementario
- Es una estrategia óptima

Algoritmo de Prim paso a paso



Algoritmo de Prim - Pseudocódigo

Prim(G)
    // Entrada: $G = \langle V, E \rangle$, es un grafo no dirigido, conexo
        // (puede usar pesos negativos)
    // Salida: E_T = conjunto de aristas que forman un $arm(G)$

    $V_T \leftarrow \{v_0\}$ // v_0 puede ser cualquier vértice
    $E_T \leftarrow \varnothing$

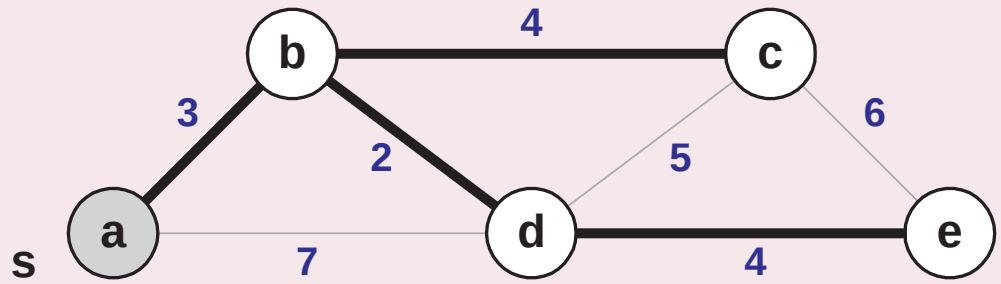
    desde $i \leftarrow 1$ **hasta** $|V| - 1$ **hacer**
        encontrar una arista $e^* = (v^*, u^*)$ de peso mínimo entre todas
            las aristas (v, u) tales que $v \in V_T$ y $u \in V \backslash V_T$
        $V_T \leftarrow V_T \cup \{u^*\}$
        $E_T \leftarrow E_T \cup \{e^*\}$

    devolver E_T

Caminos mínimos desde un nodo

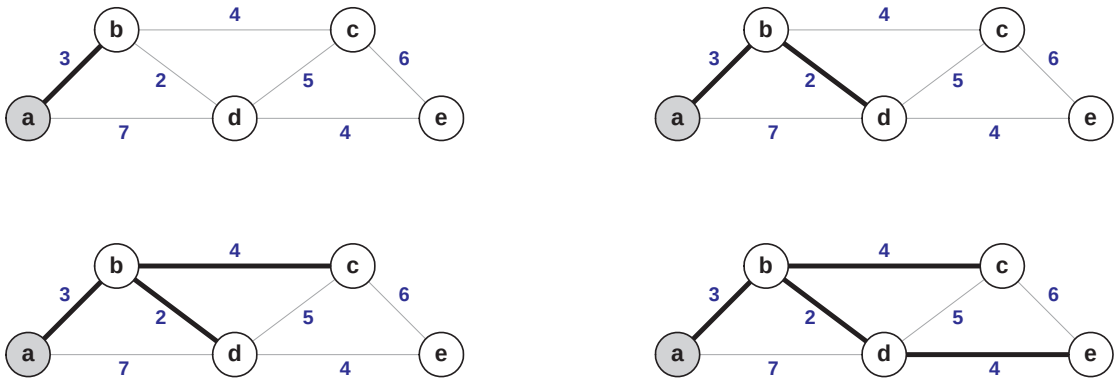
Caminos mínimos desde un nodo

Dado un grafo $G = \langle V, E \rangle$ conexo y ponderado no negativamente, y un vértice $s \in V$, hallar la longitud de los caminos mínimos desde s al resto de vértices.



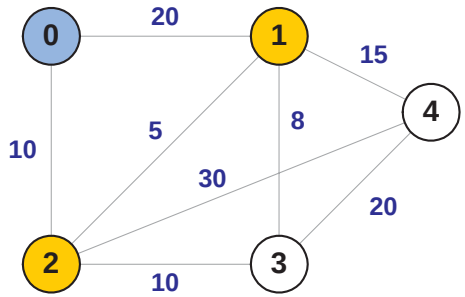
- Sirve para grafos dirigidos y no dirigidos (si es no dirigido se puede sustituir cada arista por dos con el mismo peso, y en "direcciones" contrarias)
- Veremos el **algoritmo de Dijkstra**

Algoritmo de Dijkstra paso a paso

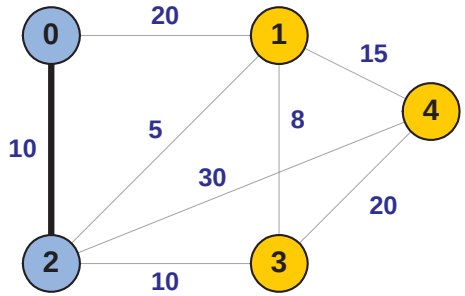


- Inserta vértices en un árbol T progresivamente, según calcula nuevas longitudes mínimas
  - El árbol T es conceptual (es una lista de nodos)
- Para un nuevo vértice tiene en cuenta no sólo la longitud de la nueva arista, sino también todo el camino hasta dicho vértice
  - Es la diferencia con respecto al algoritmo de Prim
- En cada paso inserta en T el vértice con menor longitud desde el origen (estrategia voraz óptima)
- Tras insertar un nuevo vértice v_{sig}, debe recalcular distancias de nodos conectados a v_{sig} y T

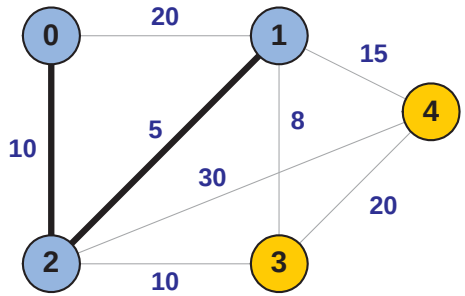
Algoritmo de Dijkstra en detalle



i	Camino	d_i	$\in T$
0	0	0	✓
1	0, 1	20	✗
2	0, 2	10	✗
3	0, 3	∞	✗
4	0, 4	∞	✗

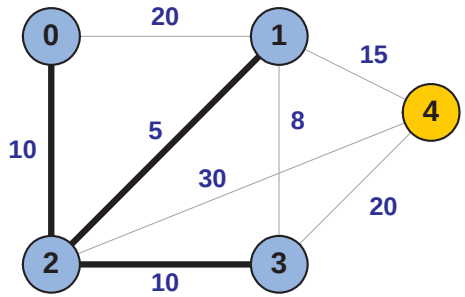


i	Camino	d_i	$\in T$
0	0	0	✓
1	0, 2, 1	15	✗
2	0, 2	10	✓
3	0, 2, 3	20	✗
4	0, 2, 4	40	✗

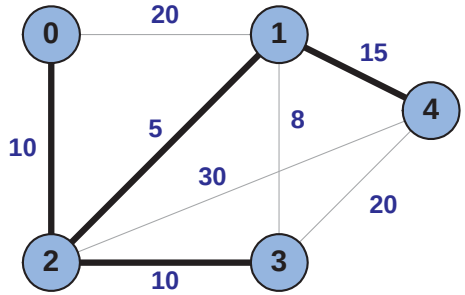


i	Camino	d_i	$\in T$
0	0	0	✓
1	0, 2, 1	15	✓
2	0, 2	10	✓
3	0, 2, 3	20	✗
4	0, 2, 1, 4	30	✗

Algoritmo de Dijkstra en detalle



i	Camino	d_i	$\in T$
0	0	0	✓
1	0, 2, 1	15	✓
2	0, 2	10	✓
3	0, 2, 3	20	✓
4	0, 2, 1, 4	30	✗



i	Camino	d_i	$\in T$
0	0	0	✓
1	0, 2, 1	15	✓
2	0, 2	10	✓
3	0, 2, 3	20	✓
4	0, 2, 1, 4	30	✓

Nota: el ejemplo usa un grafo no dirigido

Algoritmo de Dijkstra - pseudocódigo

Dijkstra(G, s, t)
    // Entrada: $G = \langle V, E \rangle$, es un grafo no dirigido, conexo,
        // y ponderado no negativamente con pesos w entre aristas
    // Salida: Distancias mínima d desde el vértice s hasta el t,
        // y todas las distancias mínimas a vértices que son menores que d

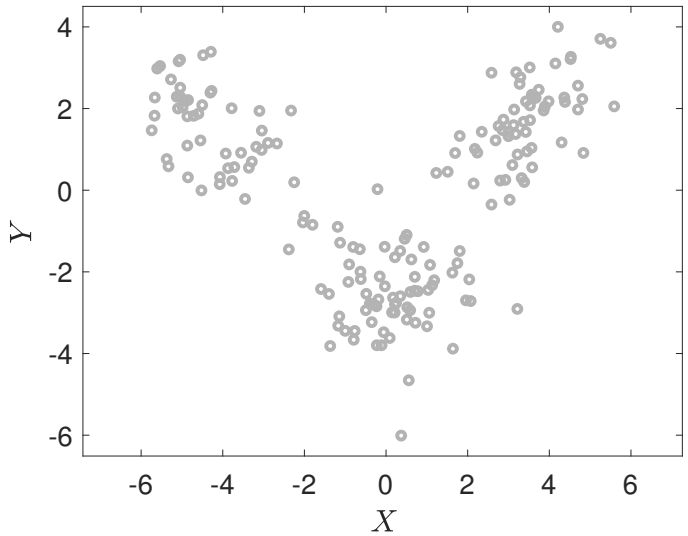
    $T \leftarrow \{s\}$
    desde $i \leftarrow 1$ **hasta** $|V|$ **hacer** $d_i \leftarrow \infty$
    para cada arista (s, v) **hacer** $d_v = w(s, v)$
    $ultimo \leftarrow s$
    mientras ($ultimo \neq t$) **hacer**
        seleccionar $v_{sig} \notin T$ // el vértice desconocido que minimiza d
        para cada arista (v_{sig}, x) **hacer**
            $d_x \leftarrow \text{mín}[d_x, d_{v_{sig}} + w(v_{sig}, x)]$
        $ultimo \leftarrow v_{sig}$
        $T \leftarrow T \cup \{v_{sig}\}$

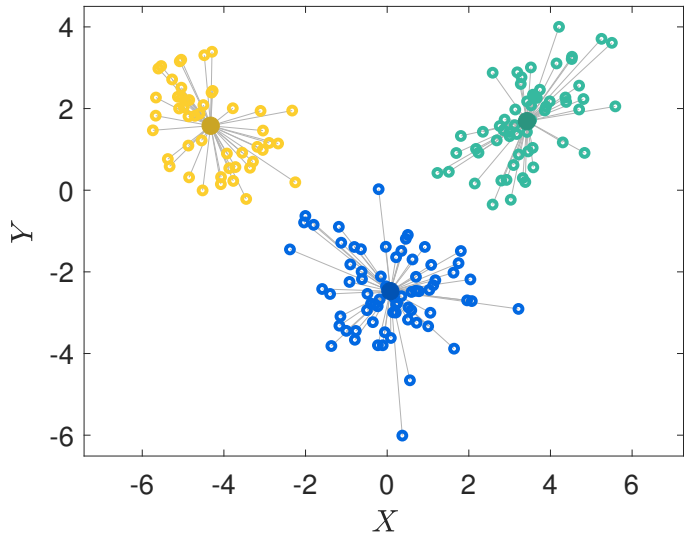
    devolver d

K-medias

K-medias

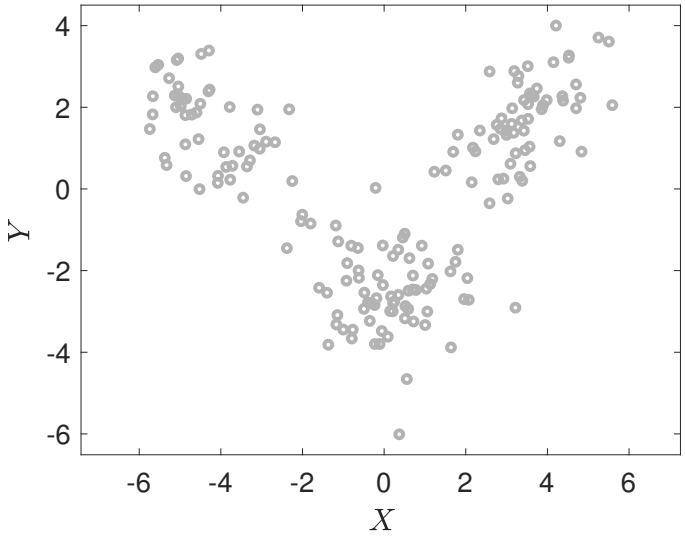
- *Clustering*: identificar grupos de puntos (datos) y asignar a cada punto uno de esos grupos
- El algoritmo K-medias procura encontrar los K mejores representantes ("centroides", que también son puntos) de los datos, donde cada centroide representa un grupo
  - El representante de un dato es el más cercano a éste
- K-medias intenta resolver el problema cuyo objetivo es minimizar la suma de distancias Euclideas (al cuadrado) entre cada dato y su centroide



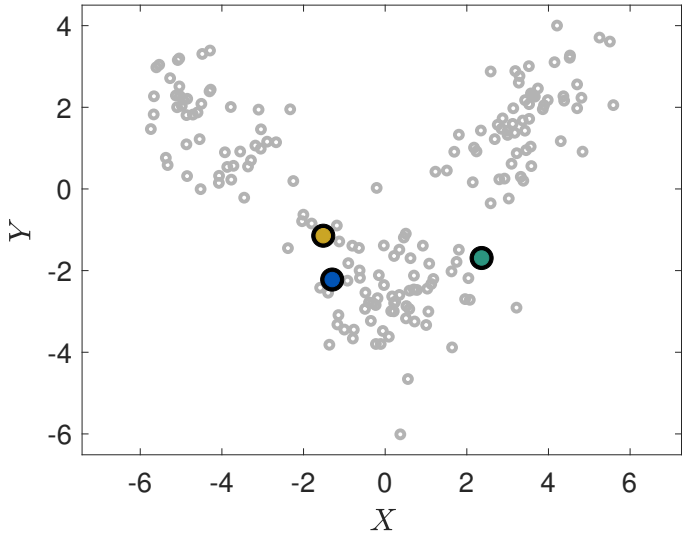


K-medias – Ejemplo de ejecución

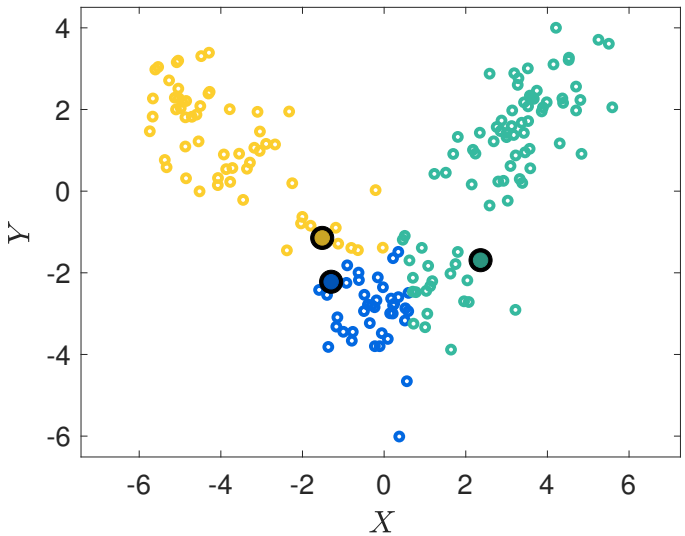
(1)



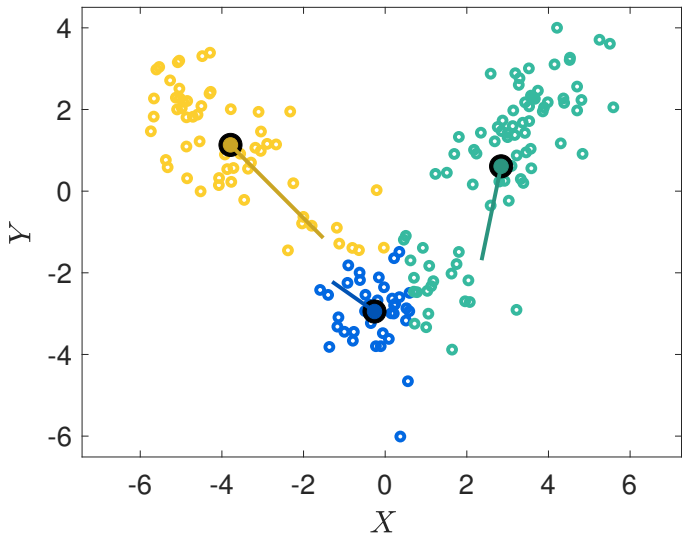
(2)



(3)

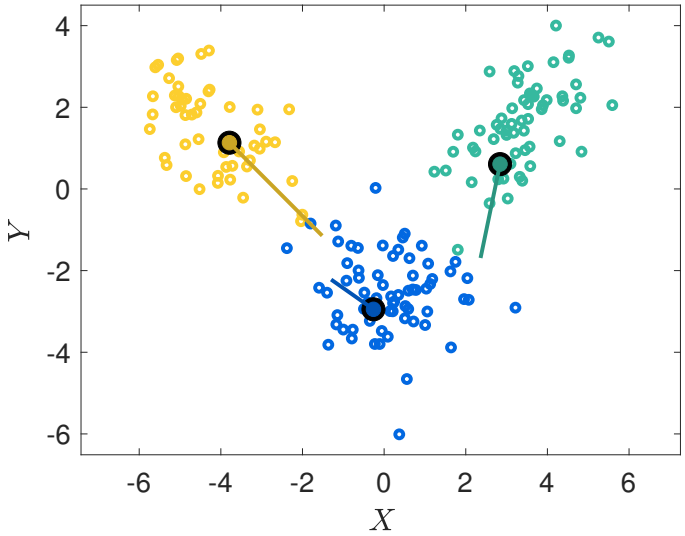


(4)

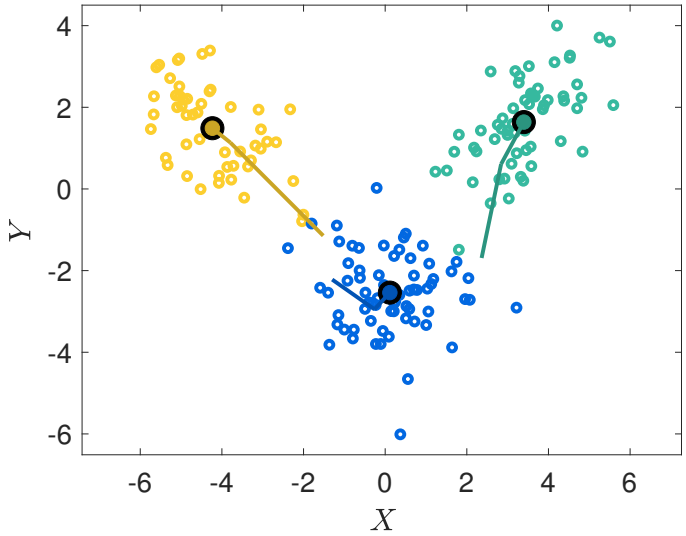


K-medias – Ejemplo de ejecución

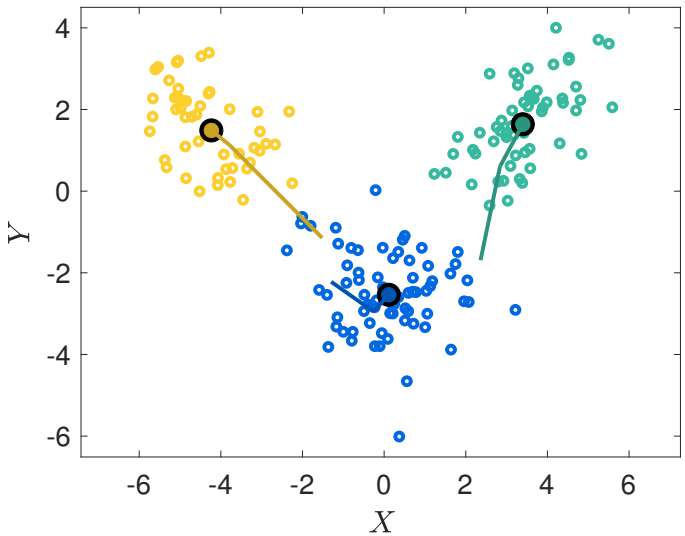
(5)



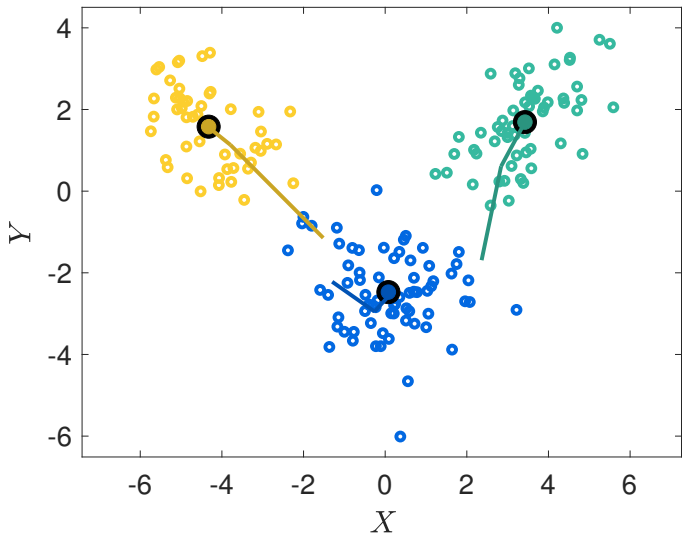
(6)



(7)



(8)



K-medias – pseudocódigo

- El problema de optimización asociado no es convexo, ni se existe una fórmula analítica para la solución
  - Recurrimos a un algoritmo iterativo

- Pseudocódigo del algoritmo **K-medias**:

K-medias(D, K)

Inicializar cada μ_k (valores aleatorios, datos concretos, etc.)

repetir:

(A) Escoge las mejores asignaciones a_{ik} para unos centroides μ_k fijos

(B) Escoge los mejores centroides μ_k para unas asignaciones a_{ik} fijas

hasta convergencia

K-medias – Dos pasos

- (A): Asignar $\mathbf{x}_i$ a su centroide μ_k más cercano (el grupo sería C_k)

$$a_{ik} = \begin{cases} 1 & \text{si } k = \arg\min_l \|\mathbf{x}_i - \mu_l\|^2 \\ 0 & \text{en caso contrario} \end{cases}$$

- (B): Asignar a μ_k la media de los puntos asignados a él (a C_k)

$$\mu_k = \frac{1}{n_k} \sum_{\substack{i:\ \mathbf{x}_i \text{ se} \\ \text{asigna a } C_k \\ (\text{es decir: } a_{ik}=1)}} \mathbf{x}_i$$

$$n_k = \sum_{i=1}^{N} a_{ik} \qquad (\text{número de puntos } \mathbf{x}_i \text{ asignados a } C_k)$$

K-medias

Problema de clustering que resuelve el K-medias

- Sea $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ un conjunto de "datos", donde $\mathbf{x}_i \in \mathbb{R}^d$, que deseamos agrupar en K grupos: $\{C_1, \ldots, C_K\}$
- Cada grupo está representado por un "centroide". Denotamos estos centroides mediante: $\mu_1, \ldots, \mu_K \in \mathbb{R}^d$
- El problema (no convexo) consiste en minimizar:

$$\mathcal{L} = \sum_{k=1}^{K} \sum_{\substack{i:\ \mathbf{x}_i \text{ se} \\ \text{asigna a } C_k}} \|\mathbf{x}_i - \mu_k\|^2 = \sum_{k=1}^{K} \sum_{i=1}^{N} a_{ik} \|\mathbf{x}_i - \mu_k\|^2$$

$$a_{ik} = \begin{cases} 1 & \text{si } \mathbf{x}_i \text{ está asignado a } C_k \\ 0 & \text{en caso contrario} \end{cases}$$

K-medias

- Se garantiza que converge a un óptimo local
  - Convergencia: los centroides no varían tras una iteración
  - También se puede parar tras T pasos
- No siempre encuentra el óptimo global
  - Se suele ejecutar varias veces con diferentes inicializaciones
- Asume que los clusters son "esféricos"
  - Poco realista
  - Hay alternativas mejores de clustering
- El problema es NP-duro
- Se puede interpretar como un caso particular del algoritmo EM (*expectation-maximization*) de teoría de probabilidad