

Algoritmos y Estructuras de Datos III  
Primer cuatrimestre 2013

Algoritmos y Estructuras de Datos III  
Primer cuatrimestre 2013

(bienvenidos!)

# Programa

## 1. Algoritmos:

- ▶ Definición de algoritmo. Máquina RAM. Complejidad. Algoritmos de tiempo polinomial y no polinomial. Límite inferior.
- ▶ Técnicas de diseño de algoritmos: *divide and conquer*, *backtracking*, algoritmos golosos, programación dinámica.
- ▶ Algoritmos aproximados y algoritmos heurísticos.

# Programa

## 2. Grafos:

- ▶ Definiciones básicas. Adyacencia, grado de un nodo, isomorfismos, caminos, conexión, etc.
- ▶ Grafos eulerianos y hamiltonianos.
- ▶ Grafos bipartitos.
- ▶ Árboles: caracterización, árboles orientados, árbol generador.
- ▶ Planaridad. Coloreo. Número cromático.
- ▶ Matching, conjunto independiente, recubrimiento. Recubrimiento de aristas y vértices.

## 3. Algoritmos en grafos y aplicaciones:

- ▶ Representación de un grafo en la computadora: matrices de incidencia y adyacencia, listas.
- ▶ Algoritmos de búsqueda en grafos: BFS, DFS, A\*.
- ▶ Mínimo árbol generador, algoritmos de Prim y Kruskal.
- ▶ Algoritmos para encontrar el camino mínimo en un grafo: Dijkstra, Ford, Floyd, Dantzig.
- ▶ Planificación de procesos: PERT/CPM.
- ▶ Algoritmos para determinar si un grafo es planar. Algoritmos para coloreo de grafos.
- ▶ Algoritmos para encontrar el flujo máximo en una red: Ford y Fulkerson.
- ▶ Matching: algoritmos para correspondencias máximas en grafos bipartitos. Otras aplicaciones.

## 4. Complejidad computacional:

- ▶ Problemas tratables e intratables. Problemas de decisión. P y NP. Máquinas de Turing no determinísticas. Problemas NP-completos. Relación entre P y NP.
- ▶ Problemas de grafos NP-completos: coloreo de grafos, grafos hamiltonianos, recubrimiento mínimo de las aristas, corte máximo, etc.

# Bibliografía

1. G. Brassard and P. Bratley, *Fundamental of Algorithmics*, Prentice-Hall, 1996.
2. F. Harary, *Graph theory*, Addison-Wesley, 1969.
3. J. Gross and J. Yellen, *Graph theory and its applications*, CRC Press, 1999.
4. R. Ahuja, T. Magnanti and J. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, 1993.
5. M. Garey and D. Johnson, *Computers and intractability: a guide to the theory of NP- Completeness*, W. Freeman and Co., 1979.

# Algoritmos

- ▶ ¿Qué es un algoritmo?
- ▶ ¿Qué es un buen algoritmo?
- ▶ Dados dos algoritmos para resolver un mismo problema, ¿cuál es mejor?
- ▶ ¿Cuándo un problema está bien resuelto?



# Complejidad computacional

**Definición informal:** La *complejidad* de un algoritmo es una función que representa el tiempo de ejecución en función del tamaño de la entrada del algoritmo.

- ▶ Complejidad en el **peor caso**.
- ▶ Complejidad en el **caso promedio**

# Complejidad computacional

**Definición informal:** La *complejidad* de un algoritmo es una función que representa el tiempo de ejecución en función del tamaño de la entrada del algoritmo.

- ▶ Complejidad en el **peor caso**.
- ▶ Complejidad en el **caso promedio** (dijo “promedio”?).

# Complejidad computacional

**Definición informal:** La *complejidad* de un algoritmo es una función que representa el tiempo de ejecución en función del tamaño de la entrada del algoritmo.

- ▶ Complejidad en el **peor caso**.
- ▶ Complejidad en el **caso promedio** (dijo “promedio”?).

**Definición formal?**

# Máquina RAM (modelo de cómputo)

**Definición:** Máquina de registros + registro acumulador + direccionamiento indirecto.

**Motivación:** Modelar computadoras en las que la memoria es suficiente y donde los enteros involucrados en los cálculos entran en una palabra.

# Máquina RAM (modelo de cómputo)

**Definición:** Máquina de registros + registro acumulador + direccionamiento indirecto.

**Motivación:** Modelar computadoras en las que la memoria es suficiente y donde los enteros involucrados en los cálculos entran en una palabra.

- ▶ **Unidad de entrada:** Sucesión de celdas numeradas, cada una con un entero de tamaño arbitrario.
- ▶ **Memoria:** Sucesión de celdas numeradas, cada una puede almacenar un entero de tamaño arbitrario.
- ▶ **Programa no almacenado** en memoria (aún así es una máquina programable!).

# Máquina RAM - Instrucciones

- ▶ LOAD operando - Carga un valor en el acumulador
- ▶ STORE operando - Carga el acumulador en un registro
- ▶ ADD operando - Suma el operando al acumulador
- ▶ SUB operando - Resta el operando al acumulador
- ▶ MULT operando - Multiplica el operando por el acumulador
- ▶ DIV operando - Divide el acumulador por el operando
- ▶ READ operando - Lee un nuevo dato de entrada → operando
- ▶ WRITE operando - Escribe el operando a la salida
- ▶ JUMP label - Salto incondicional
- ▶ JGTZ label - Salta si el acumulador es positivo
- ▶ JZERO label - Salta si el acumulador es cero
- ▶ HALT - Termina el programa

# Máquina RAM - Operandos

- ▶ **LOAD** = *a*: Carga en el acumulador el entero *a*.
- ▶ **LOAD** *i*: Carga en el acumulador el contenido del registro *i*.
- ▶ **LOAD** \**i*: Carga en el acumulador el contenido del registro indexado por el valor del registro *i*.

# Complejidad en la Máquina RAM

- ▶ Asumimos que cada instrucción tiene un **tiempo de ejecución** asociado.
- ▶ **Tiempo de ejecución de un algoritmo  $A$ :**  
 $T_A(I)$  = suma de los tiempos de ejecución de las instrucciones realizadas por el algoritmo con la *instancia*  $I$ .
- ▶ **Complejidad de un algoritmo  $A$ :**  
 $f_A(n) = \max_{I: |I|=n} T_A(I)$



# Complejidad en la Máquina RAM

- ▶ Asumimos que cada instrucción tiene un **tiempo de ejecución** asociado.
- ▶ **Tiempo de ejecución de un algoritmo  $A$ :**  
 $T_A(I)$  = suma de los tiempos de ejecución de las instrucciones realizadas por el algoritmo con la *instancia*  $I$ .
- ▶ **Complejidad de un algoritmo  $A$ :**  
 $f_A(n) = \max_{I: |I|=n} T_A(I)$  (pero debemos definir  $|I|$ !).

# Tamaño de una instancia

**Definición (incompleta):** Dada una instancia  $I$ , se define  $|I|$  como el número de símbolos de un alfabeto finito necesarios para codificar  $I$ .

- ▶ Depende del **alfabeto** y de la **base**!
- ▶ Para almacenar  $n \in \mathbb{N}$ , se necesitan  $\lceil \log_2(n+1) \rceil$  dígitos binarios.
- ▶ en una base  $b$  cualquiera, se necesitan  $\lceil \log_b(n+1) \rceil$  dígitos.
- ▶ Si  $a$  y  $b \neq 1$  entonces  $\log_b N = \frac{\log_a N}{\log_a b} = \frac{1}{\log_a b} \times \log_a N$ .

# Tamaño de una instancia

- ▶ **Modelo uniforme:** Asumimos que los valores numéricos dentro de la instancia están acotados de antemano.
- ▶ **Modelo logarítmico:** Medimos el tamaño en bits de cada entero por separado, y no se asume una cota superior de antemano.

# Notación O

Dadas dos funciones  $f, g : \mathbb{N} \rightarrow \mathbb{R}$ , decimos que:

- ▶  $f(n) = O(g(n))$  si existen  $c \in \mathbb{R}_+$  y  $n_0 \in \mathbb{N}$  tales que  $f(n) \leq c g(n)$  para todo  $n \geq n_0$ .
- ▶  $f(n) = \Omega(g(n))$  si existen  $c \in \mathbb{R}_+$  y  $n_0 \in \mathbb{N}$  tales que  $f(n) \geq c g(n)$  para todo  $n \geq n_0$ .
- ▶  $f(n) = \Theta(g(n))$  si  $f = O(g(n))$  y  $f = \Omega(g(n))$ .

# Ejemplos

- ▶ Búsqueda secuencial:  $O(n)$ .
- ▶ Búsqueda binaria:  $O(\log(n))$ .

# Ejemplos

- ▶ Búsqueda secuencial:  $O(n)$ .
- ▶ Búsqueda binaria:  $O(\log(n))$ .
- ▶ Ordenar un arreglo (bubblesort):  $O(n^2)$ .
- ▶ Ordenar un arreglo (quicksort):  $O(n^2)$  en el peor caso (!).
- ▶ Ordenar un arreglo (heapsort):  $O(n \log(n))$ .

# Ejemplos

- ▶ Búsqueda secuencial:  $O(n)$ .
- ▶ Búsqueda binaria:  $O(\log(n))$ .
- ▶ Ordenar un arreglo (bubblesort):  $O(n^2)$ .
- ▶ Ordenar un arreglo (quicksort):  $O(n^2)$  en el peor caso (!).
- ▶ Ordenar un arreglo (heapsort):  $O(n \log(n))$ .

Es interesante notar que  $O(n \log(n))$  es la complejidad **óptima** para algoritmos de ordenamiento basados en comparaciones (cómo se demuestra?).

## Ejemplo dramático: Cálculo del determinante de una matriz

**Recordemos:** Si  $A \in \mathbb{R}^{n \times n}$ , se define su *determinante* por

$$\det(A) = \sum_{j=1}^n (-1)^{j+1} a_{ij} \det(A_{ij}),$$

donde  $i \in \{1, \dots, n\}$  y  $A_{ij}$  es la submatriz de  $A$  obtenida al eliminar la fila  $i$  y la columna  $j$ .



## Ejemplo dramático: Cálculo del determinante de una matriz

**Recordemos:** Si  $A \in \mathbb{R}^{n \times n}$ , se define su *determinante* por

$$\det(A) = \sum_{j=1}^n (-1)^{j+1} a_{ij} \det(A_{ij}),$$

donde  $i \in \{1, \dots, n\}$  y  $A_{ij}$  es la submatriz de  $A$  obtenida al eliminar la fila  $i$  y la columna  $j$ .

**Complejidad:**  $f(n) = \begin{cases} n f(n-1) + O(n) & \text{si } n > 1 \\ O(1) & \text{si } n = 1 \end{cases}$

## Ejemplo dramático: Cálculo del determinante de una matriz

**Recordemos:** Si  $A \in \mathbb{R}^{n \times n}$ , se define su *determinante* por

$$\det(A) = \sum_{j=1}^n (-1)^{j+1} a_{ij} \det(A_{ij}),$$

donde  $i \in \{1, \dots, n\}$  y  $A_{ij}$  es la submatriz de  $A$  obtenida al eliminar la fila  $i$  y la columna  $j$ .

**Complejidad:**  $f(n) = \begin{cases} n f(n-1) + O(n) & \text{si } n > 1 \\ O(1) & \text{si } n = 1 \end{cases}$   
 $= O(n!)$  (oops!).

## Ejemplo dramático: Cálculo del determinante de una matriz

**Algoritmo alternativo:** Obtener la *descomposición LU*, escribiendo  $PA = LU$ . Entonces,

$$\det(A) = \det(P^{-1}) \det(L) \det(U),$$

y todos los determinantes del lado derecho son sencillos de calcular.

## Ejemplo dramático: Cálculo del determinante de una matriz

**Algoritmo alternativo:** Obtener la *descomposición LU*, escribiendo  $PA = LU$ . Entonces,

$$\det(A) = \det(P^{-1}) \det(L) \det(U),$$

y todos los determinantes del lado derecho son sencillos de calcular.

**Complejidad:**  $f(n) = O(n^3) + 3O(n) = O(n^3)$  (ta-daaa!).

# Problemas bien resueltos

**Definición:** Decimos que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para el problema.

## Problemas bien resueltos

**Definición:** Decimos que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para el problema.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
--	----------	----------	----------	----------	----------

## Problemas bien resueltos

**Definición:** Decimos que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para el problema.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$O(n)$	0.01 ms	0.02 ms	0.03 ms	0.04 ms	0.05 ms

## Problemas bien resueltos

**Definición:** Decimos que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para el problema.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$O(n)$	0.01 ms	0.02 ms	0.03 ms	0.04 ms	0.05 ms
$O(n^2)$	0.10 ms	0.40 ms	0.90 ms	0.16 ms	0.25 ms



## Problemas bien resueltos

**Definición:** Decimos que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para el problema.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$O(n)$	0.01 ms	0.02 ms	0.03 ms	0.04 ms	0.05 ms
$O(n^2)$	0.10 ms	0.40 ms	0.90 ms	0.16 ms	0.25 ms
$O(n^3)$	1.00 ms	8.00 ms	2.70 ms	6.40 ms	0.12 sg

## Problemas bien resueltos

**Definición:** Decimos que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para el problema.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$O(n)$	0.01 ms	0.02 ms	0.03 ms	0.04 ms	0.05 ms
$O(n^2)$	0.10 ms	0.40 ms	0.90 ms	0.16 ms	0.25 ms
$O(n^3)$	1.00 ms	8.00 ms	2.70 ms	6.40 ms	0.12 sg
$O(n^5)$	0.10 sg	3.20 sg	24.30 sg	1.70 min	5.20 min

## Problemas bien resueltos

**Definición:** Decimos que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para el problema.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$O(n)$	0.01 ms	0.02 ms	0.03 ms	0.04 ms	0.05 ms
$O(n^2)$	0.10 ms	0.40 ms	0.90 ms	0.16 ms	0.25 ms
$O(n^3)$	1.00 ms	8.00 ms	2.70 ms	6.40 ms	0.12 sg
$O(n^5)$	0.10 sg	3.20 sg	24.30 sg	1.70 min	5.20 min
$O(2^n)$	1.00 ms	1.00 sg	17.90 min	12 días	35 años

## Problemas bien resueltos

**Definición:** Decimos que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para el problema.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$O(n)$	0.01 ms	0.02 ms	0.03 ms	0.04 ms	0.05 ms
$O(n^2)$	0.10 ms	0.40 ms	0.90 ms	0.16 ms	0.25 ms
$O(n^3)$	1.00 ms	8.00 ms	2.70 ms	6.40 ms	0.12 sg
$O(n^5)$	0.10 sg	3.20 sg	24.30 sg	1.70 min	5.20 min
$O(2^n)$	1.00 ms	1.00 sg	17.90 min	12 días	35 años
$O(3^n)$	0.59 sg	58 min	6 años	3855 siglos	$2 \times 10^8$ siglos!

## Problemas bien resueltos

**Conclusión:** Los algoritmos polinomiales se consideran satisfactorios (cuanto menor sea el grado, mejor), y los algoritmos supra-polinomiales se consideran no satisfactorios.

# Problemas bien resueltos

**Conclusión:** Los algoritmos polinomiales se consideran satisfactorios (cuanto menor sea el grado, mejor), y los algoritmos supra-polinomiales se consideran no satisfactorios.

- ▶ Si los tamaños de instancia son pequeños, ¿es tan malo un algoritmo exponencial?
- ▶ ¿Cómo se comparan  $O(n^{85})$  con  $O(1,001^n)$ ?
- ▶ ¿Puede pasar que un algoritmo de peor caso exponencial sea eficiente en la práctica? ¿Puede pasar que en la práctica sea *el mejor*?
- ▶ ¿Qué pasa si no encuentro un algoritmo polinomial?

# Técnicas de diseño de algoritmos

- ▶ Algoritmos golosos
- ▶ *Divide and conquer* (dividir y conquistar)
- ▶ Recursividad
- ▶ Programación dinámica
- ▶ *Backtracking* (búsqueda con retroceso)
- ▶ Algoritmos probabilísticos

# Algoritmos golosos

**Idea:** Construir una solución seleccionando en cada paso la mejor alternativa, sin considerar (o haciéndolo débilmente) las implicancias de esta selección.



# Algoritmos golosos

**Idea:** Construir una solución seleccionando en cada paso la mejor alternativa, sin considerar (o haciéndolo débilmente) las implicancias de esta selección.

- ▶ Habitualmente, proporcionan **heurísticas** sencillas para **problemas de optimización**.
- ▶ En general permiten construir soluciones razonables, pero sub-óptimas.
- ▶ Sin embargo, en ocasiones nos pueden dar interesantes sorpresas!

## Ejemplo: El problema de la mochila

### Datos de entrada:

- ▶ Capacidad  $C \in \mathbb{R}_+$  de la mochila (peso máximo).
- ▶ Cantidad  $n \in \mathbb{N}$  de objetos.
- ▶ Peso  $p_i \in \mathbb{R}_+$  del objeto  $i$ , para  $i = 1, \dots, n$ .
- ▶ Beneficio  $b_i \in \mathbb{R}_+$  del objeto  $i$ , para  $i = 1, \dots, n$ .

**Problema:** Determinar qué objetos debemos incluir en la mochila sin excedernos del peso máximo  $C$ , de modo tal de **maximizar** el beneficio total entre los objetos seleccionados.

## Ejemplo: El problema de la mochila

**Algoritmo(s) goloso(s):** Mientras no se haya excedido el peso de la mochila, agregar a la mochila el objeto  $i$  que ...

- ▶ ... tenga mayor beneficio  $b_i$ .
- ▶ ... tenga menor peso  $p_i$ .
- ▶ ... maximice  $b_i/p_i$ .

## Ejemplo: El problema de la mochila

**Algoritmo(s) goloso(s):** Mientras no se haya excedido el peso de la mochila, agregar a la mochila el objeto  $i$  que ...

- ▶ ... tenga mayor beneficio  $b_i$ .
- ▶ ... tenga menor peso  $p_i$ .
- ▶ ... maximice  $b_i/p_i$ .

¿Qué podemos decir en cuanto a la **calidad** de las soluciones obtenidas por estos algoritmos? ¿Qué podemos decir en cuanto a su **complejidad**?

## Ejemplo: Tiempo de espera total en un sistema

**Problema:** Un servidor tiene  $n$  clientes para atender, y los puede atender en cualquier orden. Para  $i = 1, \dots, n$ , el tiempo necesario para atender al cliente  $i$  es  $t_i \in \mathbb{R}_+$ . El objetivo es determinar en qué orden se deben atender los clientes para minimizar la suma de los tiempos de espera de los clientes.

## Ejemplo: Tiempo de espera total en un sistema

**Problema:** Un servidor tiene  $n$  clientes para atender, y los puede atender en cualquier orden. Para  $i = 1, \dots, n$ , el tiempo necesario para atender al cliente  $i$  es  $t_i \in \mathbb{R}_+$ . El objetivo es determinar en qué orden se deben atender los clientes para minimizar **la suma de los tiempos de espera** de los clientes.

Si  $I = (i_1, i_2, \dots, i_n)$  es una permutación de los clientes que representa el orden de atención, entonces la suma de los tiempos de espera es

$$\begin{aligned} T &= t_{i_1} + (t_{i_1} + t_{i_2}) + (t_{i_1} + t_{i_2} + t_{i_3}) + \dots \\ &= \sum_{k=1}^n (n - k + 1) t_{i_k}. \end{aligned}$$

## Ejemplo: Tiempo de espera total en un sistema

**Algoritmo goloso:** En cada paso, atender al cliente pendiente que tenga menor tiempo de atención.

- ▶ Retorna una permutación  $I_{\text{GOL}} = (i_1, \dots, i_n)$  tal que  $t_{i_j} \leq t_{i_{j+1}}$  para  $j = 1, \dots, n - 1$ .
- ▶ ¿Cuál es la **complejidad** de este algoritmo?

## Ejemplo: Tiempo de espera total en un sistema

**Algoritmo goloso:** En cada paso, atender al cliente pendiente que tenga menor tiempo de atención.

- ▶ Retorna una permutación  $I_{\text{GOL}} = (i_1, \dots, i_n)$  tal que  $t_{i_j} \leq t_{i_{j+1}}$  para  $j = 1, \dots, n - 1$ .
- ▶ ¿Cuál es la **complejidad** de este algoritmo?
- ▶ Este algoritmo proporciona la **solución óptima**! (cómo se demuestra?)



# Divide and conquer

- ▶ Si la instancia  $I$  de entrada es pequeña, entonces utilizar un algoritmo ad hoc para el problema.
- ▶ En caso contrario:
  - ▶ **Dividir**  $I$  en sub-instancias  $I_1, I_2, \dots, I_k$  más pequeñas.
  - ▶ Resolver **recursivamente** las  $k$  sub-instancias.
  - ▶ **Combinar** las soluciones para las  $k$  sub-instancias para obtener una solución para la instancia original  $I$ .

## Ejemplo: Mergesort

Algoritmo *divide and conquer* para ordenar un arreglo  $A$  de  $n$  elementos (von Neumann, 1945).

- ▶ Si  $n$  es pequeño, ordenar por cualquier método sencillo.
- ▶ Si  $n$  es grande:
  - ▶  $A_1 :=$  primera mitad de  $A$ .
  - ▶  $A_2 :=$  segunda mitad de  $A$ .
  - ▶ Ordenar recursivamente  $A_1$  y  $A_2$  por separado.
  - ▶ Combinar  $A_1$  y  $A_2$  para obtener los elementos de  $A$  ordenados (apareo de arreglos).

## Ejemplo: Mergesort

Algoritmo *divide and conquer* para ordenar un arreglo  $A$  de  $n$  elementos (von Neumann, 1945).

- ▶ Si  $n$  es pequeño, ordenar por cualquier método sencillo.
- ▶ Si  $n$  es grande:
  - ▶  $A_1 :=$  primera mitad de  $A$ .
  - ▶  $A_2 :=$  segunda mitad de  $A$ .
  - ▶ Ordenar recursivamente  $A_1$  y  $A_2$  por separado.
  - ▶ Combinar  $A_1$  y  $A_2$  para obtener los elementos de  $A$  ordenados (apareo de arreglos).

Este algoritmo contiene todos los elementos típicos de la técnica *divide and conquer*.

## Ejemplo: Mergesort

Algoritmo *divide and conquer* para ordenar un arreglo  $A$  de  $n$  elementos (von Neumann, 1945).

- ▶ Si  $n$  es pequeño, ordenar por cualquier método sencillo.
- ▶ Si  $n$  es grande:
  - ▶  $A_1 :=$  primera mitad de  $A$ .
  - ▶  $A_2 :=$  segunda mitad de  $A$ .
  - ▶ Ordenar recursivamente  $A_1$  y  $A_2$  por separado.
  - ▶ Combinar  $A_1$  y  $A_2$  para obtener los elementos de  $A$  ordenados (apareo de arreglos).

Este algoritmo contiene todos los elementos típicos de la técnica *divide and conquer*.

## Ejemplo: Multiplicación de Strassen

- ▶ Sean  $A, B \in \mathbb{R}^{n \times n}$ . El algoritmo estándar para calcular  $AB$  tiene una complejidad de  $\Theta(n^3)$ .
- ▶ Durante muchos años se pensaba que esta complejidad era **óptima**.

## Ejemplo: Multiplicación de Strassen

- ▶ Sean  $A, B \in \mathbb{R}^{n \times n}$ . El algoritmo estándar para calcular  $AB$  tiene una complejidad de  $\Theta(n^3)$ .
- ▶ Durante muchos años se pensaba que esta complejidad era **óptima**.
- ▶ Sin embargo, Strassen (1969) pateó el tablero. Particionamos:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$

## Ejemplo: Multiplicación de Strassen

Definimos:

$$M_1 = (A_{21} + A_{22} - A_{11})(B_{22} - B_{12} + B_{11})$$

$$M_2 = A_{11}B_{11}$$

$$M_3 = A_{12}B_{21}$$

$$M_4 = (A_{11} - A_{21})(B_{22} - B_{12})$$

$$M_5 = (A_{21} + A_{22})(B_{12} - B_{11})$$

$$M_6 = (A_{12} - A_{21} + A_{11} - A_{22})B_{22}$$

$$M_7 = A_{22}(B_{11} + B_{22} - B_{12} - B_{21}).$$

## Ejemplo: Multiplicación de Strassen

Definimos:

$$M_1 = (A_{21} + A_{22} - A_{11})(B_{22} - B_{12} + B_{11})$$

$$M_2 = A_{11}B_{11}$$

$$M_3 = A_{12}B_{21}$$

$$M_4 = (A_{11} - A_{21})(B_{22} - B_{12})$$

$$M_5 = (A_{21} + A_{22})(B_{12} - B_{11})$$

$$M_6 = (A_{12} - A_{21} + A_{11} - A_{22})B_{22}$$

$$M_7 = A_{22}(B_{11} + B_{22} - B_{12} - B_{21}).$$

Entonces,

$$AB = \begin{pmatrix} M_2 + M_3 & M_1 + M_2 + M_5 + M_6 \\ M_1 + M_2 + M_4 - M_7 & M_1 + M_2 + M_4 + M_5 \end{pmatrix}.$$



## Ejemplo: Multiplicación de Strassen

- ▶ Este algoritmo permite calcular el producto  $AB$  en tiempo  $O(n^{\log_2(7)}) = O(n^{2,81})$  (!).
- ▶ Requiere 7 multiplicaciones de matrices de tamaño  $n/2 \times n/2$ , en comparación con las 8 multiplicaciones del algoritmo estándar.
- ▶ La cantidad de sumas (y restas) de matrices es mucho mayor.

## Ejemplo: Multiplicación de Strassen

- ▶ Este algoritmo permite calcular el producto  $AB$  en tiempo  $O(n^{\log_2(7)}) = O(n^{2,81})$  (!).
- ▶ Requiere 7 multiplicaciones de matrices de tamaño  $n/2 \times n/2$ , en comparación con las 8 multiplicaciones del algoritmo estándar.
- ▶ La cantidad de sumas (y restas) de matrices es mucho mayor.
- ▶ El algoritmo asintóticamente más eficiente conocido a la fecha tiene una complejidad de  $O(n^{2,376})$  (Coppersmith y Winograd, 1987).

# Backtracking

**Idea:** Técnica para recorrer sistemáticamente todas las posibles configuraciones de un espacio asociado a soluciones candidatos de un problema computacional. Se puede pensar este espacio tiene forma de árboles dirigidos (o grafos dirigidos en general pero sin ciclos).

# Backtracking

**Idea:** Técnica para recorrer sistemáticamente todas las posibles configuraciones de un espacio asociado a soluciones candidatas de un problema computacional. Se puede pensar este espacio tiene forma de árboles dirigidos (o grafos dirigidos en general pero sin ciclos).

- ▶ Habitualmente, utiliza un **vector**  $a = (a_1, a_2, \dots, a_k)$  para representar una solución candidata, cada  $a_i$  pertenece un dominio/conjunto ordenado y finito  $S_i$ .
- ▶ Se extienden las soluciones candidatas agregando un elemento más al final del vector  $a$ , las nuevas soluciones candidatas son sucesores de la anterior.

# Backtracking: Esquema General

$BT(a, k)$

1. Si  $a$  es solución
2.    entonces solución :=  $a$
3.    debe\_finalizar? := verdadero
4.    sino para cada  $a' \in \text{Sucesores}(a, k)$
5.         $BT(a', k + 1)$
6.    Si debe\_finalizar?
7.        entonces retornar solución

# Backtracking: Esquema General

$BT(a, k)$

1. Si  $a$  es solución
2.    entonces solución :=  $a$
3.    debe\_finalizar? := verdadero
4.    sino para cada  $a' \in \text{Sucesores}(a, k)$
5.         $BT(a', k + 1)$
6.    Si debe\_finalizar?
7.        entonces retornar solución

- ▶ solución es una variable global que guarda la solución final.
- ▶ debe\_finalizar? es una variable booleana global que indica que se encontró o no la solución final, inicialmente tiene valor falso.

## Ejemplo: Problema de las 8 reinas

Ubicar 8 reinas en el tablero de ajedrez ( $8 \times 8$ ) sin que ninguna "amenace" a otra.

## Ejemplo: Problema de las 8 reinas

Ubicar 8 reinas en el tablero de ajedrez ( $8 \times 8$ ) sin que ninguna "amenace" a otra.

- ▶ ¿Cuántas combinaciones del tablero hay que considerar?



## Ejemplo: Problema de las 8 reinas

Ubicar 8 reinas en el tablero de ajedrez ( $8 \times 8$ ) sin que ninguna "amenace" a otra.

- ▶ ¿Cuántas combinaciones del tablero hay que considerar?

$$\binom{64}{8} = 442616536$$

- ▶ Sabemos que cada fila debe tener exactamente una reina. Entonces  $a_i$  es la posición (columna que está la reina de la fila  $i$ ) o sea podemos usar el vector  $a = (a_1, \dots, a_8)$  representa una solución candidata.

## Ejemplo: Problema de las 8 reinas

Ubicar 8 reinas en el tablero de ajedrez ( $8 \times 8$ ) sin que ninguna "amenace" a otra.

- ▶ ¿Cuántas combinaciones del tablero hay que considerar?

$$\binom{64}{8} = 442616536$$

- ▶ Sabemos que cada fila debe tener exactamente una reina. Entonces  $a_i$  es la posición (columna que está la reina de la fila  $i$ ) o sea podemos usar el vector  $a = (a_1, \dots, a_8)$  representa una solución candidata.

Tenemos ahora  $8^8 = 16777216$  combinaciones.

## Ejemplo: Problema de las 8 reinas

- ▶ Es más, una misma columna debe tener exactamente una reina!

## Ejemplo: Problema de las 8 reinas

- Es más, una misma columna debe tener exactamente una reina!

Se reduce a  $8! = 40320$  combinaciones.

## Ejemplo: Problema de las 8 reinas

- ▶ Es más, una misma columna debe tener exactamente una reina!

Se reduce a  $8! = 40320$  combinaciones.

- ▶ ¿Cómo chequear un vector  $a$  es una solución?

## Ejemplo: Problema de las 8 reinas

- Es más, una misma columna debe tener exactamente una reina!

Se reduce a  $8! = 40320$  combinaciones.

- ¿Cómo chequear un vector  $a$  es una solución?

$$a_i - a_j \notin \{i - j, 0, j - 1\}, \forall i, j \in \{1, \dots, 8\} \text{ e } i \neq j.$$

## Ejemplo: Problema de las 8 reinas

- Es más, una misma columna debe tener exactamente una reina!

Se reduce a  $8! = 40320$  combinaciones.

- ¿Cómo chequear un vector  $a$  es una solución?

$$a_i - a_j \notin \{i - j, 0, j - 1\}, \forall i, j \in \{1, \dots, 8\} \text{ e } i \neq j.$$

- Ahora estamos en condición de implementar un algoritmo para resolver el problema!

## Ejemplo: Problema de las 8 reinas

- Es más, una misma columna debe tener exactamente una reina!

Se reduce a  $8! = 40320$  combinaciones.

- ¿Cómo chequear un vector  $a$  es una solución?

$$a_i - a_j \notin \{i - j, 0, j - 1\}, \forall i, j \in \{1, \dots, 8\} \text{ e } i \neq j.$$

- Ahora estamos en condición de implementar un algoritmo para resolver el problema!
- ¿Cómo generalizar para el problema de  $n$  reinas?



# Algoritmos probabilísticos

- ▶ Cuando un algoritmo tiene que hacer una elección a veces es preferible elegir al azar en vez de gastar mucho tiempo tratando de ver cual es la mejor elección.
- ▶ Algoritmos al azar para problemas numéricos: siempre da una respuesta aproximada. + tiempo proceso  $\Rightarrow$  + precisión (por ejemplo cálculo de integral)
- ▶ Algoritmos de Monte Carlo: siempre da una respuesta exacta no garantizada. + tiempo proceso  $\Rightarrow$  + probabilidad de acertar (por ejemplo determinar la existencia de un elemento mayor en un arreglo).

# Algoritmos probabilísticos

- ▶ Algoritmos Las Vegas: la respuesta siempre es correcta pero puede no darla. + tiempo proceso  $\Rightarrow$  + probabilidad de obtener la respuesta (por ejemplo problema de 8 reinas)
- ▶ Algoritmos Sherwood : randomiza un algoritmo determinístico donde hay una gran diferencia entre el peor caso y caso promedio. Elimina la diferencia entre buenas y malas instancias (quicksort).

# Heurísticas

- ▶ Dado un problema  $\Pi$  , un algoritmo heurístico es un algoritmo que intenta obtener soluciones para el problema que intenta resolver pero no necesariamente lo hace en todos los casos.
- ▶ Sea  $\Pi$  un problema de optimización,  $I$  una instancia del problema,  $x^*(I)$  el valor óptimo de la función a optimizar en dicha instancia. Un algoritmo heurístico obtiene una solución con un valor que se espera sea cercano a ese óptimo pero no necesariamente el óptimo.
- ▶ Si  $H$  es un algoritmo heurístico para un problema de optimización llamamos  $x^H(I)$  al valor que devuelve la heurística.

# Algoritmos aproximados

Decimos que  $H$  es un algoritmo  $\epsilon$  – aproximado para el problema  $\Pi$  si para algún  $\epsilon > 0$

$$|x^H(I) - x^*(I)| = \epsilon |x^*(I)|$$

# Algoritmos con certificados

- ▶ ¿Cómo podemos saber si la respuesta o el resultado de un algoritmo es correcto o no?
- ▶ Algoritmos que ordenan un arreglo
- ▶ Algoritmos que multiplican matrices
- ▶ Algoritmos que determinan un número natural es o no compuesto

# Algoritmos con certificados

- ▶ ¿Cómo podemos saber si la respuesta o el resultado de un algoritmo es correcto o no?
- ▶ Algoritmos que ordenan un arreglo
- ▶ Algoritmos que multiplican matrices
- ▶ Algoritmos que determinan un número natural es o no compuesto
- ▶ Certificados y algoritmos de verificación o autenticación