

# **Introducción a la Bioinformática**

## **Conceptos elementales de Computación**

### **Algoritmos**

**Fernán Agüero**

**Instituto de Investigaciones Biotecnológicas**

**UNSAM**

- Es la disciplina que estudia como resolver problemas con computadoras
- Deriva de las Matemáticas
  - Resolución de problemas
  - Algoritmos
- Qué es un Algoritmo?
  - Informalmente: *“una serie de reglas que definen en forma precisa una secuencia de operaciones”*
  - Formalmente: *“un método, expresado como una lista finita de instrucciones bien definidas para calcular una función”*

# Qué es un algoritmo?

Comenzando en un **estado o "input"** inicial

Las instrucciones describen **cómputos**

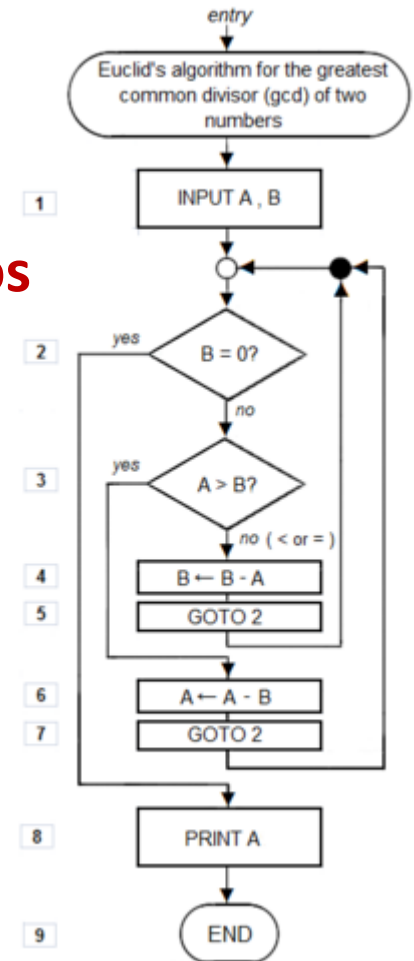
Que al ser ejecutados

Procederán por una serie de **estados bien definidos**

Eventualmente produciendo un **"output"**

Terminando en un estado final

<http://en.wikipedia.org/wiki/Algorithm>



# Qué es un algoritmo?

## Un algoritmo puede ser especificado

- En inglés, español, francés, etc.
- En un lenguaje formal: matemático, o de programación
  - C, Java, Perl, Python
- En forma de un diseño de hardware

## A qué se parece?

- A un protocolo
- A una receta de cocina

## Un programa es una implementación de un algoritmo!

### Ejemplo:

BLAST, algoritmo para realizar búsquedas de similitud entre secuencias.

**Hay varias implementaciones: NCBI (C, C++), WashU, etc.**

# Problemas computacionales

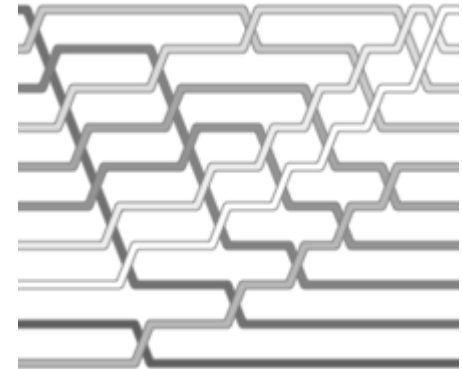
- Vamos a ver varios ejemplos a lo largo del curso!
- Tienen que cumplir con las siguientes condiciones
  - Tiene que estar bien definido
  - Tiene que tener una solución
- “Un problema es una colección *infinita* de instancias, junto con una solución para cada una de esas instancias”
- Ejemplo: ordenar números en forma creciente
- **Input:** una secuencia de  $n$  números ( $a_1, a_2, \dots, a_n$ )
- **Output:** una permutación de la secuencia original ( $a'_1, a'_2, \dots, a'_n$ ) tal que  $a'_1 > a'_2 > \dots > a'_n$
- Una instancia del problema:
  - **Input:** (31, 41, 59, 26, 41, 58)
  - **Output:** (26, 31, 41, 41, 58, 59)

Hay muchas maneras de ordenar números!

Algoritmos para ordenar:

- **Insertion sort**
- **Merge sort**
- **Selection sort**
- **Bubble sort**
- ...
- **Cuál es el mejor?**

Hay que analizar el algoritmo!



# Wikipedia es un libro de algoritmos!

ikipedia.org/wiki/Insertion\_sort

Apture Editor Altmetric it Scoop.it! Spotify Web Player Open Access Button

Other bookmarks

## Insertion sort

From Wikipedia, the free encyclopedia

**Insertion sort** is a simple [sorting algorithm](#) that builds the final [sorted array](#) (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as [quicksort](#), [heapsort](#), or [merge sort](#). However, insertion sort provides several advantages:

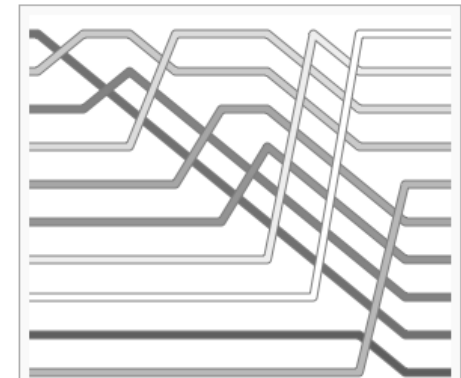
- Simple implementation
- Efficient for (quite) small data sets
- [Adaptive](#) (i.e., efficient) for data sets that are already substantially sorted: the [time complexity](#) is  $O(n + d)$ , where  $d$  is the number of [inversions](#)
- More efficient in practice than most other simple quadratic (i.e.,  $O(n^2)$ ) algorithms such as [selection sort](#) or [bubble sort](#); the best case (nearly sorted input) is  $O(n)$
- [Stable](#); i.e., does not change the relative order of elements with equal keys
- [In-place](#); i.e., only requires a constant amount  $O(1)$  of additional memory space
- [Online](#); i.e., can sort a list as it receives it

When people manually sort something (for example, a deck of playing cards), most use a method that is similar to insertion sort.<sup>[1]</sup>

### Contents [\[hide\]](#)

- 1 Algorithm
- 2 Best, worst, and average cases
- 3 Relation to other sorting algorithms
- 4 Variants
  - 4.1 List insertion sort code in C
- 5 References
- 6 External links

### Insertion sort



Graphical illustration of insertion sort

<b>Class</b>	<a href="#">Sorting algorithm</a>
<b>Data structure</b>	<a href="#">Array</a>
<b>Worst case performance</b>	$O(n^2)$ comparisons, swaps
<b>Best case performance</b>	$O(n)$ comparisons, $O(1)$ swaps
<b>Average case performance</b>	$O(n^2)$ comparisons, swaps
<b>Worst case space</b>	$O(n)$ total, $O(1)$ auxiliary complexity

## Corrección

Un algoritmo es “**correcto**” si para cada instancia de un input, termina con el output correcto (la solución).

**Hay algoritmos incorrectos!**

- Aproximados
- Heurísticos



## Eficiencia

Una medida posible de eficiencia es la **“velocidad”**: cuánto demora un algoritmo en llegar al output/solución.

Pero la **“velocidad”** es engañosa (depende del hardware).

Tal vez la mejor definición sea:

“Cuántos recursos utiliza el algoritmo para realizar su tarea”

**Recursos = tiempo, espacio de almacenamiento, cantidad de memoria.**

- Dado un conjunto de números
  - Existe algún subconjunto cuya suma sea **N**?
    - Ejemplo:
      - **Conjunto:** 1 2 4 5 8 9 10 11 23 76 89
      - **N** = 119

- **Evaluar**

- **Viabilidad**

- Existen impedimentos teóricos?

- **Solución (algoritmo)**

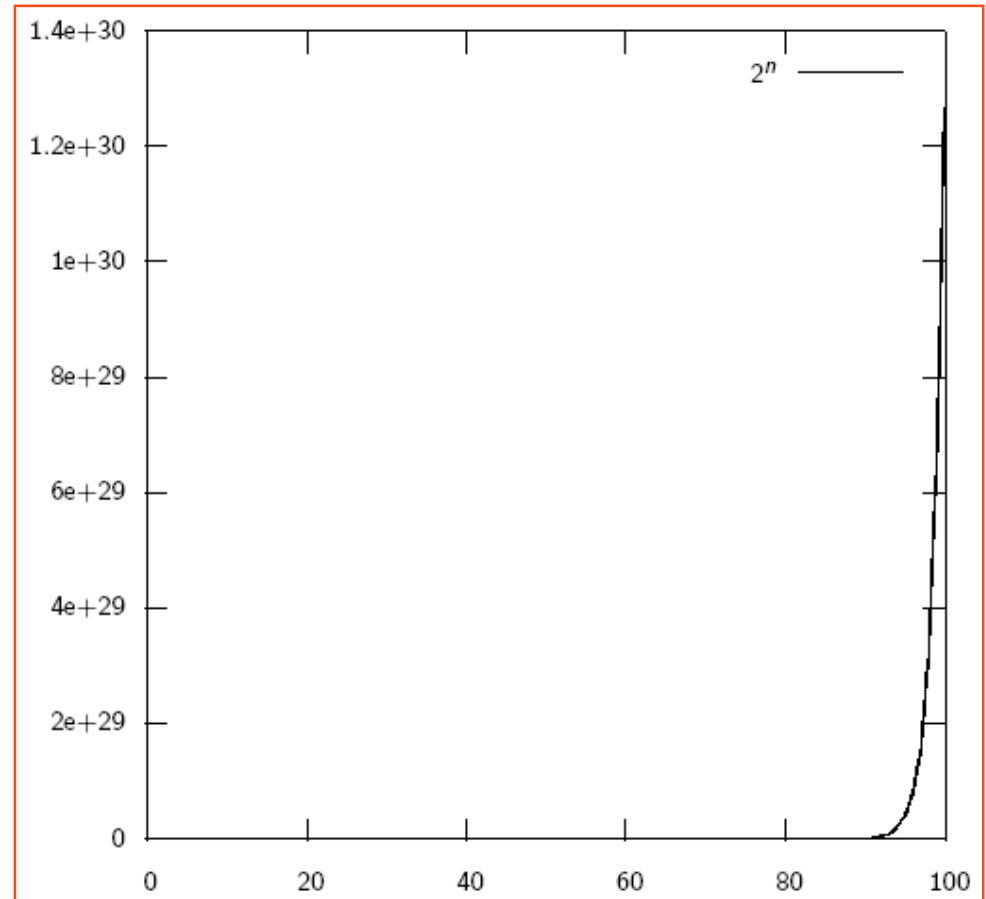
- Tomar subconjuntos de S y evaluar la suma
    - Responder SI/NO

- **Análisis del algoritmo**

- Eficiencia: cómo escala con el input
    - Problema = hay  $2^S$  subconjuntos de S!
      - Si cada subconjunto se verifica en 0.0001 segundo
      - Para 100 elementos tardaríamos
      - $2^{100} * 0.0001s / 60 * 60 * 24 * 360 * 100 = 6.34 \cdot 10^{86}$  siglos

# Complejidad

- Métrica para comparar algoritmos
- Relación entre cantidad de datos de entrada (input) y tiempo/espacio necesario para resolver el problema
- Normalmente se especifica para
  - $\theta(n)$ , mejor caso
  - $\Omega(n)$ , caso promedio
  - $O(n)$ , peor caso



- **Constante**
  - $O(1)$
- **Lineal**
  - $O(n)$
- **Polinomial**
  - $O(n^2)$
- **Exponencial**
  - $O(2^n)$

- Pueden existir varias maneras de resolver un problema
- La elección final puede depender del tiempo que estamos dispuestos a esperar para obtener una respuesta
- Algoritmo de la división
  - $10 / 5 = 2$
  - $133 / 4 = 33,25$
  - $10 / 3 = 3,33333...$
- Opciones del algoritmo
  - Termina cuando no hay más resto
  - Termina cuando llega a colocar la coma
  - Termina cuando llega al n-ésimo dígito.

- **Algoritmo exacto**
  - Da la solución exacta (óptima)
- **Algoritmo aproximado**
  - Da una solución que se encuentra a una distancia calculable (el error) de la solución óptima.
- **Heurística**
  - Da una solución aceptable sin error calculable.

- **Abundan en biología**
  - Soluciones ‘casi’ óptimas
  - Costo computacional razonable
  - No garantiza factibilidad del resultado ni da idea de a qué distancia se encuentra uno del resultado óptimo
  - Pero se pueden hacer análisis estadísticos sobre los datos
- **Algoritmos genéticos**
  - Optimizan una función objetivo
  - Generalmente al azar, utilizando mecanismos similares a los que usa la evolución



- **Inteligencia artificial, Machine learning**
  - Se entrenan programas sobre un set de datos “de entrenamiento”
  - El programa entrenado se enfrenta con un set de datos desconocido
    - **Hidden Markov Models**
    - **Neural networks**
- **Ejemplos en biología:**
  - **Reconocimiento de patrones**
    - SignalP, NetOGlyc, DGPI
    - Identificación de genes
    - Búsquedas de similitud de alta sensibilidad

Introduction to Algorithms (2009). Cormen, Leiserson, Rivest, Stein. 3rd Edition, MIT Press.