

# 11074 – Programación I

División Computación – Departamento de Ciencias Básicas

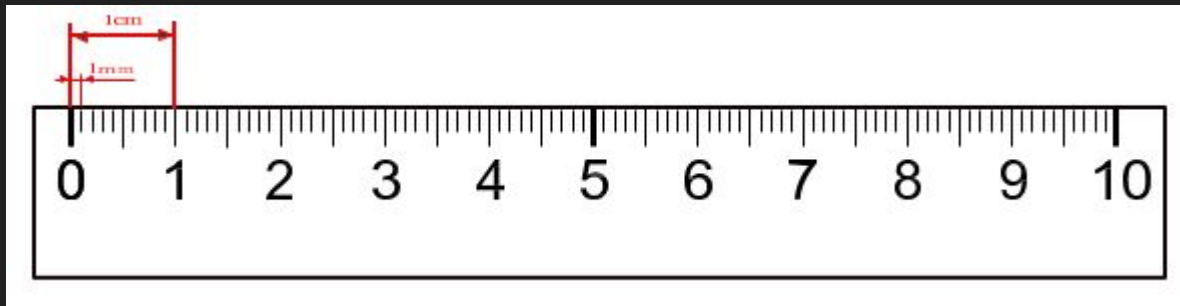


## EFICIENCIA COMPUTACIONAL



# Complejidad algorítmica: concepto

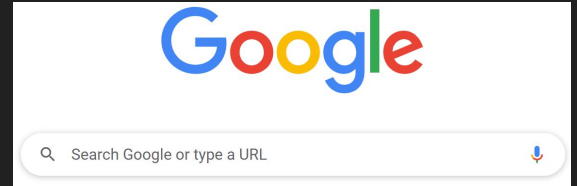
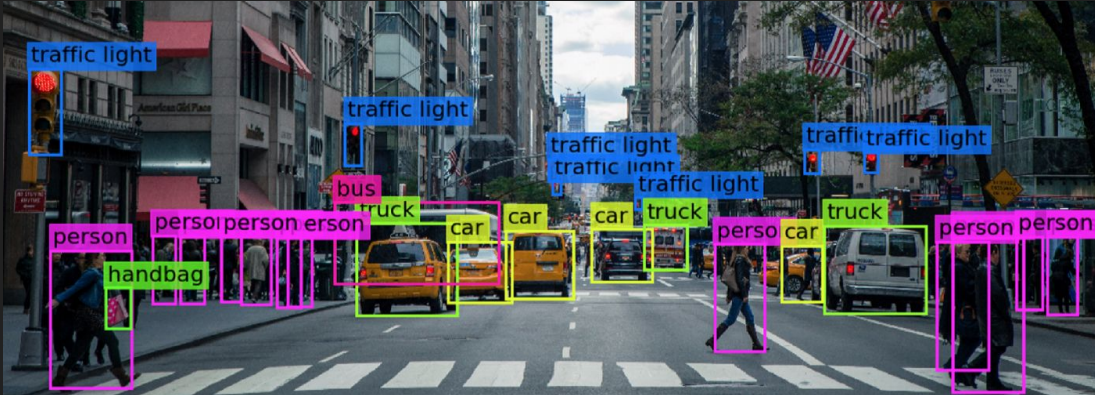
La complejidad algorítmica es la rama de la computación que estudia la **eficiencia** de los programas.



# Complejidad algorítmica: importancia

Las computadoras son cada vez más veloces pero...

- los conjuntos de datos cada vez son más grandes
- los problemas computacionales a resolver son cada vez más complejos
- hay problemas que necesitan resolverse en forma casi instantánea



4 millones de búsquedas por minuto

# ¿Qué medimos?

Diferentes recursos:

- Tiempo
- Memoria
- Procesadores

Se denomina **análisis de algoritmos** al proceso de calcular la complejidad computacional de los algoritmos: o sea el monto de tiempo, memoria, u otros recursos necesarios para ejecutarlos.

# Notación O grande

La notación **O grande** (big-O en inglés, O por orden) es una forma de cuantificar la tasa a la cual una cantidad crece.

Ejemplo:

- Un cuadrado de longitud de lado  $r$  tiene un área  $O(r^2)$ , ya que el área de un cuadrado es lado x lado.
- Un círculo de radio  $r$  tiene un área  $O(r^2)$ , ya que el área de un círculo es  $\pi r^2$

*Esto significa que ambas áreas tienen la misma tasa de crecimiento, no que son iguales!*

## Ejemplo de Big-O: Manufactura

- Estamos trabajando en una empresa que fabrica zapatillas. Cuesta cierto monto de dinero producir cada zapatilla, y hay además un costo para poner la fábrica.
- ¿Qué dato necesitamos para estimar el costo de producir 10 millones de zapatillas?

*Este término crece como función de  $n$*

*Este término no crece*

$$\begin{aligned}\text{Costo}(n) &= n \times \text{costoPorZapatilla} + \text{costoFabrica} \\ &= O(n)\end{aligned}$$

# Truco para calcular Big-O

Eliminamos los coeficientes de los términos de mayor orden, y eliminamos los términos de menor orden (incluyendo las constantes).

$$\text{Costo}(n) = \$2 \times n + \$500$$

$$= \cancel{\$2 \times} n + \cancel{\$500}$$

$$\text{Costo}(n) = O(n)$$

# Objetivo de Big-O

Big-O está diseñado para capturar **la tasa a la cual una cantidad crece**. No captura información acerca de:

- los coeficientes: el área de un cuadrado y de un círculo son ambos  $O(r^2)$ .
- los términos de menor orden.

A pesar de ello, es una **herramienta muy poderosa para predecir el crecimiento**.

Aunque no tengamos una fórmula exacta, saber a cuál tasa una cantidad escala, nos permite predecir su valor en el futuro.



# Midiendo la eficiencia temporal: timer



```
#include <stdio.h>
#include <time.h>      // para clock_t, clock(), CLOCKS_PER_SEC
#include <unistd.h>    // para sleep()
//Mide tiempo de ejecución de un programa en C con el clock de la computadora
int main(){
    //para almacenar el tiempo de ejecución del código
    double time_spent = 0.0;
    clock_t begin = clock();
    //hacer algunas cosas aquí
    sleep(3);
    clock_t end = clock();
    //calcula el tiempo transcurrido encontrando la diferencia (end - begin)
    //y dividiendo la diferencia por CLOCKS_PER_SEC para convertir a segundos
    time_spent += (double)(end - begin) / CLOCKS_PER_SEC;
    printf("El tiempo transcurrido es %f segundos\n", time_spent);
    return 0;
}
```

# Midiendo eficiencia con clock computadora

## *Ventajas:*

- Para programas complejos, es lo más sencillo y rápido.

## *Desventajas:*

- El tiempo de ejecución del programa depende de la computadora (las computadoras más rápidas tardarán menos)
- En una misma computadora el tiempo de ejecución varía en cada prueba, ya que el tiempo de ejecución depende de los otros procesos que está corriendo la computadora en ese momento
- El tiempo de ejecución varía para diferentes inputs, pero no podemos expresar una relación entre las entradas y el tiempo
- **Una medición individual de una ejecución del programa no puede predecir el tiempo de ejecución futuro del programa.**

# Midiendo eficiencia temporal

- Para poder independizar la eficiencia de un algoritmo del hardware de la computadora, pensamos que se ejecuta en una **computadora imaginaria**.
- Entonces, para medir la eficiencia temporal de un algoritmo, se cuentan la **cantidad de instrucciones del programa**.
- Se asume que **cada instrucción elemental tarda 1 unidad de tiempo**.
- En consecuencia, dados 2 algoritmos que resuelven la misma tarea, el algoritmo más rápido, es el que tiene menor cantidad de unidades de tiempo.

*La complejidad es un tema que verán con mayor profundidad en Programación 2.*

# Análisis de un algoritmo: ejemplo

Sea ut: *unidad de tiempo*. ¿Cuánto tarda la función vectorMin()?

```
int vectorMin(int* v, int n){  
    int minimoActual = v[0];           1 ut  
    for (int i = 1; i < n; i++){       1 ut + n ut + (n-1) ut  
        if (minimoActual > v[i])       (n-1) ut  
            minimoActual = v[i];      (como máximo) (n-1) ut  
    }  
    return minimoActual;               1 ut  
}
```

Máximo tiempo total:  $4n$

# Aplicando el Big-O en algoritmos

Queremos una tasa que represente lo siguiente: A medida que **crece el tamaño de la entrada** cuánto **crece el tiempo de ejecución**?

Máximo tiempo total de VectorMin():  $4n$ . Si duplicamos el tamaño de la entrada, se duplica el tiempo de ejecución.  $\Rightarrow$  El tamaño de la entrada y el tiempo de ejecución tienen una relación lineal  $\Rightarrow$  VectorMin() es  $O(n)$

¿Habrá otras características de la entrada, aparte de su tamaño, que influyan en la eficiencia de un algoritmo?

# Búsqueda lineal: eficiencia

*Peor caso:* el elemento  $x$  a buscar no está en el vector  $a$  o está en la última posición. El ciclo del algoritmo hará  $n$  iteraciones.

*Mejor caso:* el elemento  $x$  está en la primera posición. El ciclo hace 1 iteración.

Para medir la eficiencia de un algoritmo se considera usualmente al peor caso.

```
int buscar(int clave, int datos[], int n){
    int i;
    i = 0;
    while ( (i < n) && (datos[i] != clave) )
        i = i + 1;
    return i;
}
```

# Búsqueda binaria: eficiencia

*Mejor caso:* el elemento buscado está en el centro. Se hace una sola iteración.

*Peor caso:* El elemento buscado está en una esquina. Se hacen  $\log_2(n)$  iteraciones.

```
int buscar(int clave, int datos[], int n) {
    int inferior, superior, centro, resu, encontrado;
    inferior = 0;
    superior = n-1;
    resu= n;
    encontrado = 0;
    while ( (inferior <= superior) && !encontrado) {
        centro = (inferior + superior) / 2; /* Trunca hacia cero */
        if (clave == datos[centro]){
            resu = centro; /* Encontrado */
            encontrado = 1;}
        else {
            if (clave < datos[centro])
                superior = centro - 1; /* Buscar en la primera mitad */
            else
                inferior = centro + 1; /* Buscar en la segunda mitad */
        }
        /* Si no lo encuentra, retorna 'n' */
    }
    return resu;
}
```



# Categorías de complejidad

Si tenemos un algoritmo con 1000 elementos, y el  $O(\log n)$  corre en 10 milisegundos...

| <i>constante</i> | <i>logarítmica</i> | <i>lineal</i> | <i>n log n</i> | <i>cuadrática</i> | <i>polinomial</i> | <i>exponencial</i>         |
|------------------|--------------------|---------------|----------------|-------------------|-------------------|----------------------------|
| $O(1)$           | $O(\log n)$        | $O(n)$        | $O(n \log n)$  | $O(n^2)$          | $O(n^k) (k > 2)$  | $O(a^n) (a > 1)$           |
| 1 milisegundo    | 10 milisegundos    | 1 segundo     | 10 segundos    | 17 minutos        | 277 horas         | muerte oscura del universo |

**Hay casos en los que nos conviene ser  
ineficientes a propósito...**

# Ineficiencia algorítmica

Los algoritmos de encriptado están diseñados intencionalmente para ser muy lentos, y de un uso intensivo de recursos y de memoria, de tal forma que adivinar la clave sea muy costoso, aunque se intentase millones de veces por minuto.



THE ART OF  
COMPUTER PROGRAMMING

VOLUME 4 PRE-FASCICLE 6A

A (PARTIAL) DRAFT  
OF SECTION 7.2.2.2:  
SATISFIABILITY

DONALD E. KNUTH *Stanford University*

ADDISON-WESLEY  
February 17, 2003

How to read this:  
Please ignore these  
instructions; they're just  
here to remind you  
to ignore the text,  
and stay to other stuff!

KNUTH

THE ART OF  
COMPUTER PROGRAMMING

VOLUME 4 PRE-FASCICLE 5A

MATHEMATICAL  
PRELIMINARIES  
REDUX

DONALD E. KNUTH *Stanford University*

ADDISON-WESLEY  
April 1, 2001

How to read this:  
Please ignore these  
instructions; they're just  
here to remind you  
to ignore the text,  
and stay to other stuff!

Updated  
and  
Revised

THE CLASSIC  
EXTENDED A

The Art of  
Computer  
Programming

VOLUME 4A  
Combinatorial Algorithms  
Part 1

DONALD E.

Updated  
and  
Revised

THE CLASSIC  
NEWLY UPDATED

The Art of  
Computer  
Programming

VOLUME 3  
Sorting and Searching  
Second Edition

DONALD E.

Updated  
and  
Revised

THE CLASSIC  
NEWLY UPDATED

The Art of  
Computer  
Programming

VOLUME 2  
Seminumerical Algorithms  
Third Edition

DONALD E.

Updated  
and  
Revised

THE CLASSIC  
NEWLY UPDATED

The Art of  
Computer  
Programming

VOLUME 1  
Fundamental Algorithms  
MMIX  
A RISC Computer  
New Millennium Edition

DONALD E.

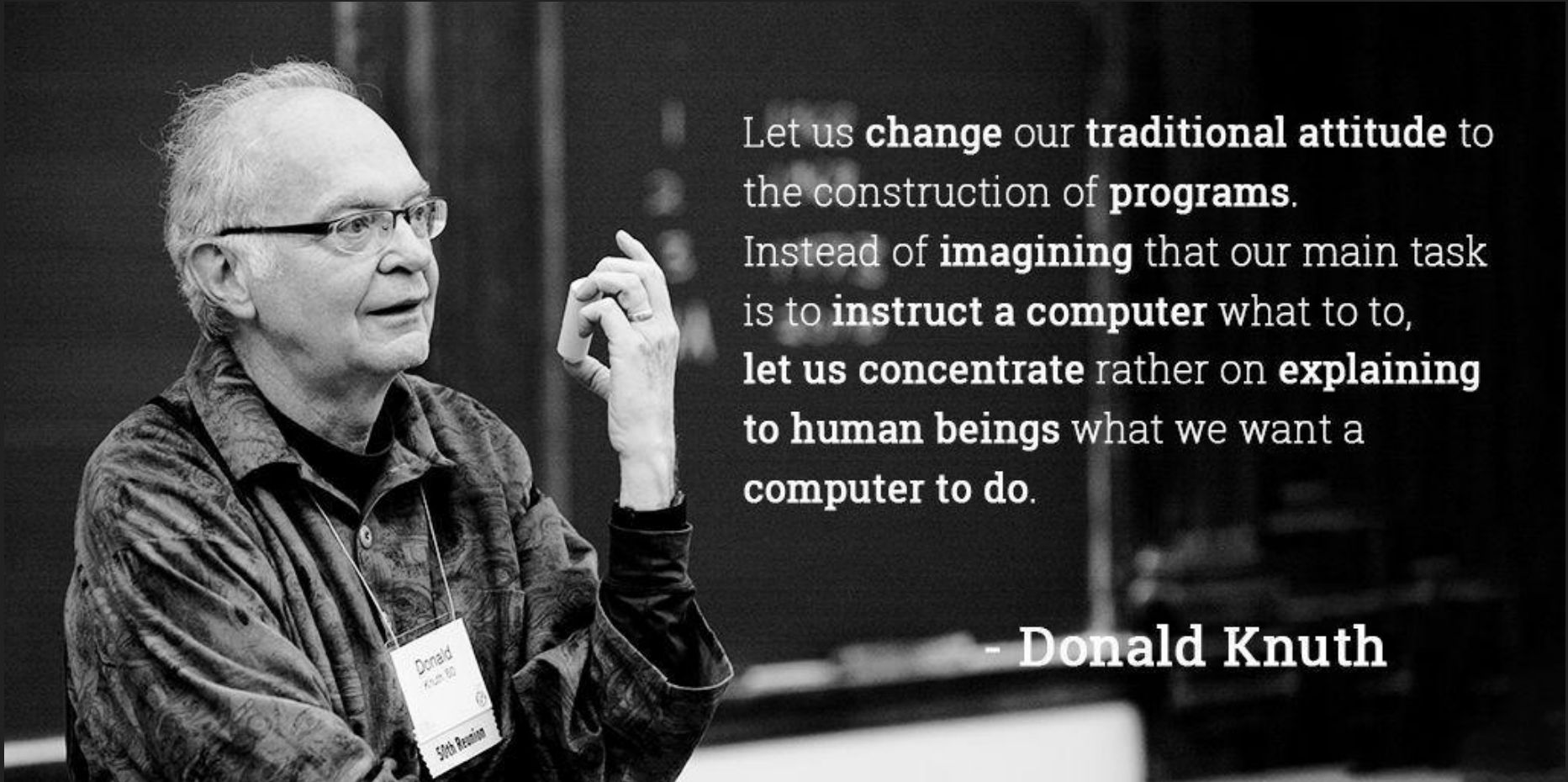
THE CLASSIC  
NEWLY UPDATED

The Art of  
Computer  
Programming

VOLUME 0  
Fundamental Algorithms  
First Edition

DONALD E.

## Donald Knuth: *Padre del análisis de algoritmos*

A black and white photograph of Donald Knuth. He is an older man with glasses, wearing a patterned button-down shirt over a dark t-shirt. He has a conference badge around his neck that reads "Donald Knuth '80" and "50th Reunion". He is gesturing with his right hand, pointing upwards with his index finger. The background is dark and out of focus, showing what appears to be a stage or lecture hall setting.

Let us **change** our **traditional attitude** to the construction of **programs**.

Instead of **imagining** that our main task is to **instruct a computer** what to do, **let us concentrate** rather on **explaining to human beings** what we want a computer to do.

- Donald Knuth