Algoritmos y Estructuras de Datos

PARTE II: ALGORÍTMICA (o ALGORITMIA)

Tema 0. Introducción

0.1. Definición y propiedades.

0.2. Análisis y diseño de algoritmos.

0.3. Heurísticas para una buena programación.

0.1. Definición y propiedades.

- **Algoritmo:**

  Conjunto de reglas para resolver un problema.
- **Propiedades**
  - **Definibilidad:** El conjunto debe estar bien definido, sin dejar dudas en su interpretación.
  - **Finitud:** Debe tener un número finito de pasos que se ejecuten en un tiempo finito.

0.1. Definición y propiedades.

- **Algoritmos deterministas:** Para los mismos datos de entrada se producen los mismos datos de salida.
- **Algoritmos no deterministas:** Para los mismos datos de entrada pueden producirse diferentes de salida.

- **ALGORITMIA:** Ciencia que estudia técnicas para construir algoritmos eficientes y técnicas para medir la eficacia de los algoritmos.
- **Objetivo**: Dado un problema concreto encontrar la mejor forma de resolverlo.

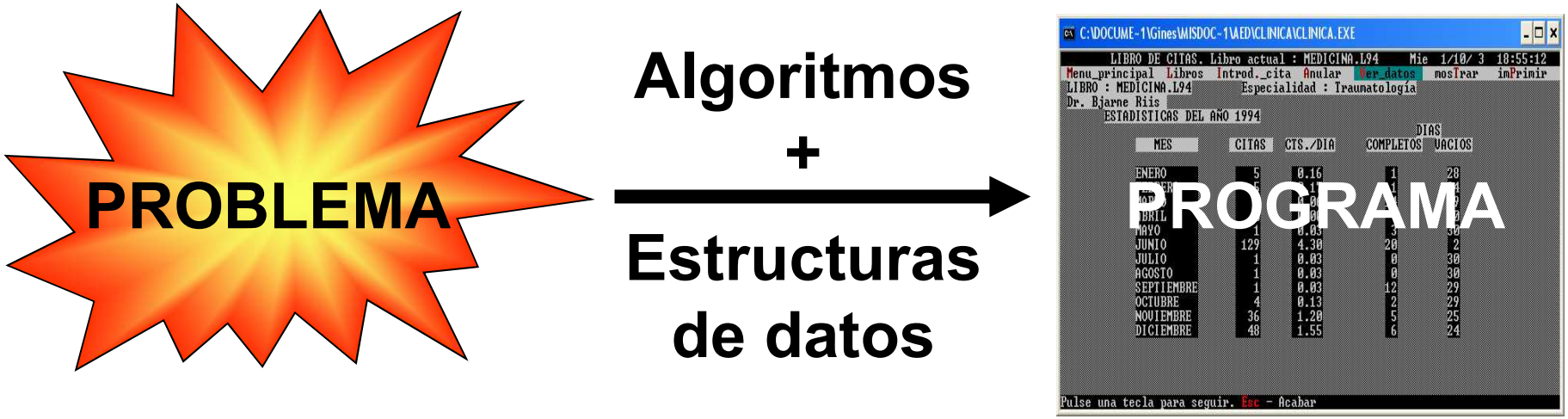
0.1. Definición y propiedades.

Recordamos:
Objetivo de la asignatura

Ser capaz de **analizar**, **comprender** y **resolver** una amplia variedad de **problemas** de programación, diseñando soluciones **eficientes** y de **calidad**.

Pero **ojo**, los algoritmos no son el único componente en la resolución de un problema de programación.

0.1. Definición y propiedades.



Algoritmos + Estructuras de Datos = Programas

- **Estructura de datos:** Parte estática, almacenada.
- **Algoritmo:** Parte dinámica, manipulador.

0.1. Definición y propiedades.

Resolver problemas

¿Cómo se resuelve un problema?

¿Cuándo se dice que la solución es eficiente y de calidad?

¿Qué clase de problemas?

0.2. Análisis y diseño de algoritmos.

ALGORITMIA = ANÁLISIS + DISEÑO

- **Análisis de algoritmos:** Estudio de los recursos que necesita la ejecución de un algoritmo.
- No confundir con análisis de un problema.

- **Diseño de algoritmos:** Técnicas generales para la construcción de algoritmos.
- Por ejemplo, divide y vencerás: dado un problema, divídelo, resuelve los subproblemas y luego junta las soluciones.

0.2. Análisis y diseño de algoritmos.

- **Análisis de algoritmos.** Normalmente estamos interesados en el estudio del tiempo de ejecución.

- Dado un algoritmo, usaremos las siguientes notaciones:
  - t(..): Tiempo de ejecución del algoritmo.
  - O(..): Orden de complejidad.
  - o(..): O pequeña del tiempo de ejecución.
  - Ω(..): Cota inferior de complejidad.
  - Θ(..): Orden exacto de complejidad.

0.2. Análisis y diseño de algoritmos.

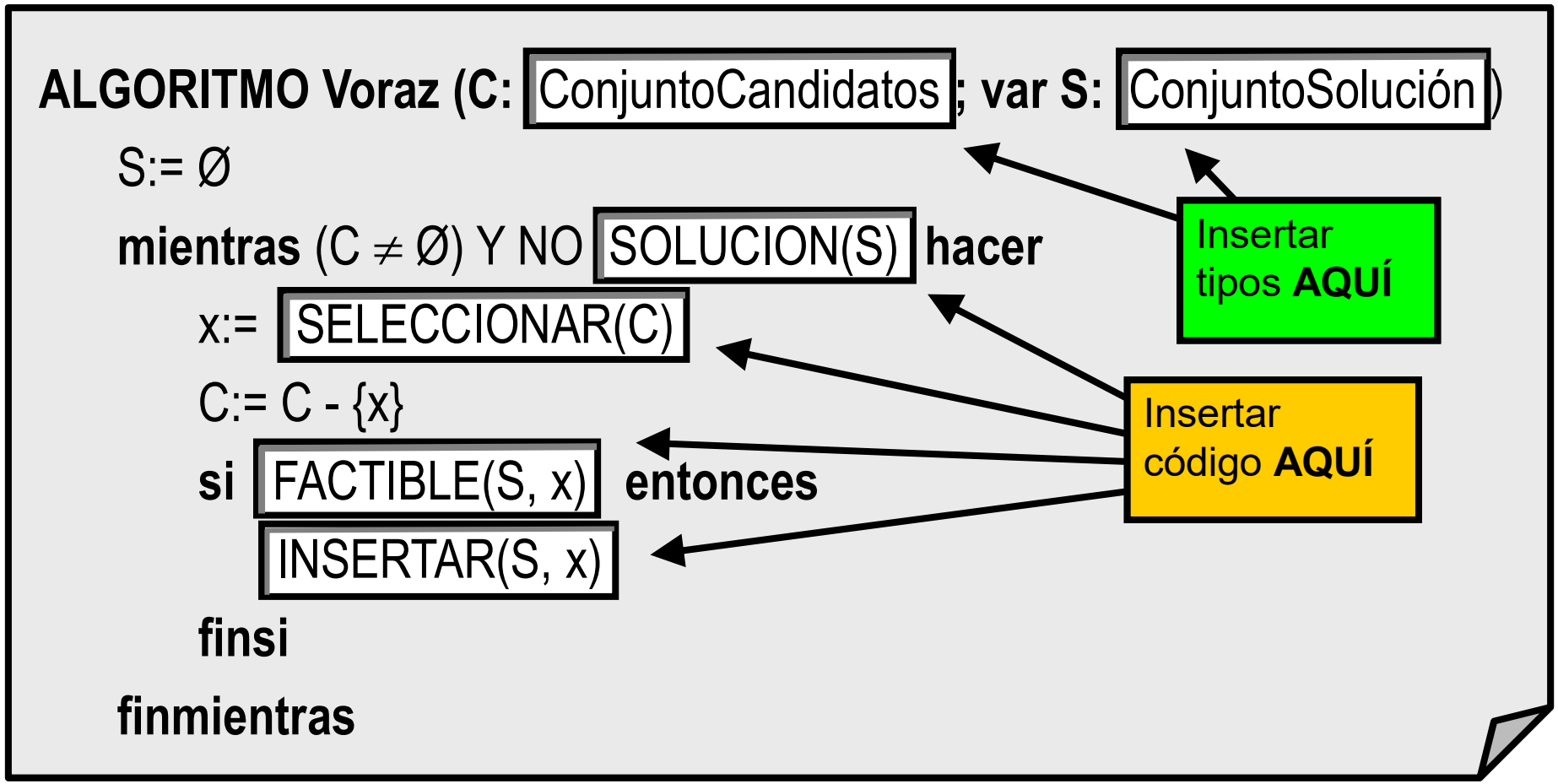
- **Ejemplo.** Analizar el tiempo de ejecución y el orden de complejidad del siguiente algoritmo.

```
Hanoi (N, A, B, C: integer)
    if  N=1 then
        Mover (A, C)
    else begin
        Hanoi (N-1, A, C, B)
        Mover (A, C)
        Hanoi (N-1, B, A, C)
    end
```

- Mecanismos:
  - Conteo de instrucciones.
  - Uso de ecuaciones de recurrencia.

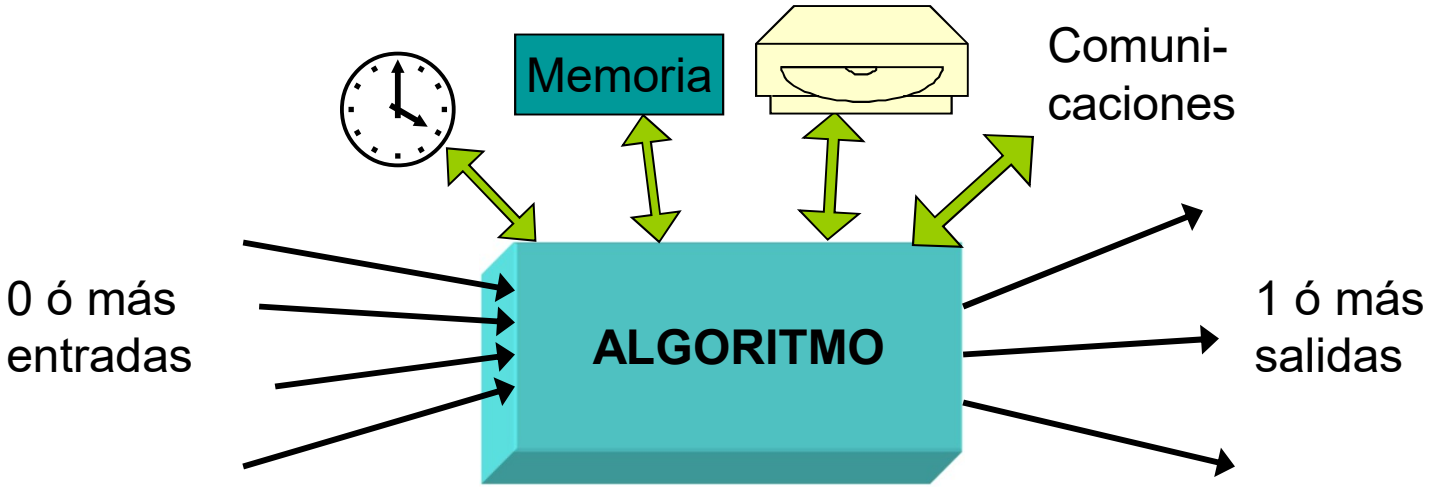
0.2. Análisis y diseño de algoritmos.

- **Diseño de Algoritmos.** Técnicas generales, aplicables a muchas situaciones.
- **Esquemas algorítmicos.** Ejemplo:



1.1. Introducción.

- **Algoritmo:** Conjunto de reglas para resolver un problema. Su ejecución requiere unos recursos.



- Un algoritmo es mejor cuantos menos recursos consuma. Pero....
- **Otros criterios:** facilidad de programarlo, corto, fácil de entender, robusto...

1.1. Introducción.

- **Criterio empresarial:** Maximizar la eficiencia.
- **Eficiencia:** Relación entre los recursos consumidos y los productos conseguidos.
- **Recursos consumidos:**
  - **Tiempo de ejecución.**
  - **Memoria principal.**
  - Entradas/salidas a disco.
  - Comunicaciones, procesadores,...
- **Lo que se consigue:**
  - Resolver un problema de forma exacta.
  - Resolverlo de forma aproximada.
  - Resolver algunos casos...

1.1. Introducción.

- **Recursos consumidos.**

  Ejemplo. ¿Cuántos recursos de tiempo y memoria consume el siguiente algoritmo sencillo?

```
i:= 0
a[n+1]:= x
repetir
        i:= i + 1
hasta a[i] == x
```

- **Respuesta:** Depende.
- ¿De qué depende?
- De lo que valga **n** y **x**, de lo que haya en **a**, de los tipos de datos, de la máquina...

1.1. Introducción.

- Factores que influyen en el consumo de recursos:
  - **Factores externos.**
    - El ordenador donde se ejecute.
    - El lenguaje de programación y el compilador usado.
    - La implementación que haga el programador del algoritmo. En particular, de las estructuras de datos utilizadas.
  - **Tamaño de los datos de entrada.**
    - Ejemplo. Procesar un fichero de log con N líneas.
  - **Contenido de los datos de entrada.**
    - **Mejor caso (t_m).** El contenido favorece una rápida ejecución.
    - **Peor caso (t_M).** La ejecución más lenta posible.
    - **Caso promedio (t_p).** Media de todos los posibles contenidos.

1.1. Introducción.

- Los factores externos no aportan información sobre el algoritmo.
- **Conclusión**: Estudiar la variación del tiempo y la memoria necesitada por un algoritmo respecto al tamaño de la entrada y a los posibles casos, de forma aproximada (y parametrizada).
- **Ejemplo**. Algoritmo de búsqueda secuencial.
  - Mejor caso. Se encuentra x en la 1ª posición:

    $t_m(N) = a$
  - Peor caso. No se encuentra x:

    $t_M(N) = b \cdot N + c$
- **Ojo**: El mejor caso no significa tamaño pequeño.

1.1. Introducción.

Normalmente usaremos la notación t(N)=..., pero **¿qué significa t(N)?**

- Tiempo de ejecución en segundos. t(N) = bN + c.
  - Suponiendo que b y c son constantes, con los segundos que tardan las operaciones básicas correspondientes.
- Instrucciones ejecutadas por el algoritmo. t(N) = 2N + 4.
  - ¿Tardarán todas lo mismo?
- Ejecuciones del bucle principal. t(N) = N+1.
  - ¿Cuánto tiempo, cuántas instrucciones,...?
  - Sabemos que cada ejecución lleva un tiempo constante, luego se diferencia en una constante con los anteriores.

1.1. Introducción.

- El proceso básico de análisis de la eficiencia algorítmica es el conocido como **conteo de instrucciones** (**o de memoria**).

- **Conteo de instrucciones:** Seguir la ejecución del algoritmo, sumando las instrucciones que se ejecutan.
- **Conteo de memoria:** Lo mismo. Normalmente interesa el máximo uso de memoria requerido.

- **Alternativa:** Si no se puede predecir el flujo de ejecución se puede intentar predecir el **trabajo total realizado**.
  - Ejemplo. Recorrido sobre grafos: se recorren todas las adyacencias, aplicando un tiempo cte. en cada una.

1.1. Introducción.

Conteo de instrucciones. Reglas básicas:

- Número de instrucciones t(n) → sumar 1 por cada instrucción o línea de código de ejecución constante.
- Tiempo de ejecución t(n) → sumar una constante (c_1, c_2, ...) por cada tipo de instrucción o grupo de instrucciones secuenciales.

- **Bucles FOR**: Se pueden expresar como un sumatorio, con los límites del FOR como límites del sumatorio.

$$\sum_{i=1}^{n} k = kn \qquad \sum_{i=a}^{b} k = k(b-a+1) \qquad \sum_{i=1}^{n} i = n(n+1)/2$$

$$\sum_{i=a}^{b} r^i = \frac{r^{b+1} - r^a}{r - 1} \qquad \sum_{i=1}^{n} i^2 \approx \int_{0}^{n} i^2 \, di = (i^3)/3 \Big]_{0}^{n} = (n^3)/3$$

1.1. Introducción.

Conteo de instrucciones. Reglas básicas:

- **Bucles WHILE y REPEAT**: Estudiar lo que puede ocurrir. ¿Existe una cota inferior y superior del número de ejecuciones? ¿Se puede convertir en un FOR?

- **Llamadas a procedimientos**: Calcular primero los procedimientos que no llaman a otros. $t_1(n)$, $t_2(n)$, ...

- **IF** y **CASE**: Estudiar lo que puede ocurrir. ¿Se puede predecir cuándo se cumplirán las condiciones?
  - Mejor caso y peor caso según la condición.
  - Caso promedio: suma del tiempo de cada caso, por probabilidad de ocurrencia de ese caso.

1.1. Introducción.

- **Ejemplos**. Estudiar t(n).

Instrucción	Costo	Veces que se repite
1 for j←2 to length[A]	c_1	
2 do key←A[j]	c_2	
3 i←j-1	c_3	
4 while i >0 and A[i] > key	c_4	
5 do A[i+1]←A[i]	c_5	
6 i←i-1	c_6	
7 A[i+1]←key	c_7	

1.1. Introducción.

- **Ejemplos**. Estudiar t(n).

for j←1 to 3 ⟶ for (int j=1; j<=3; j++)

j=1 √

j=2 √

j=3 √

j=4 ×

La cantidad de comparaciones en un for es:

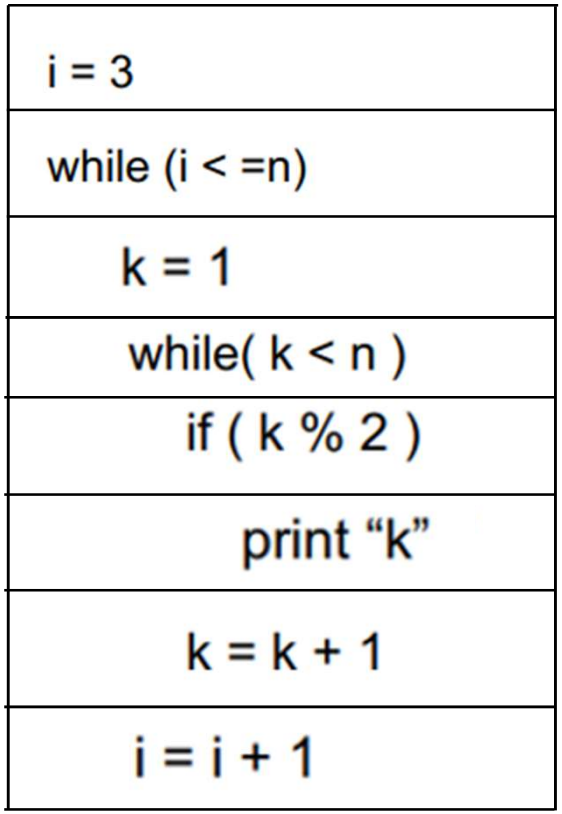
cantidad de números válidos + 1

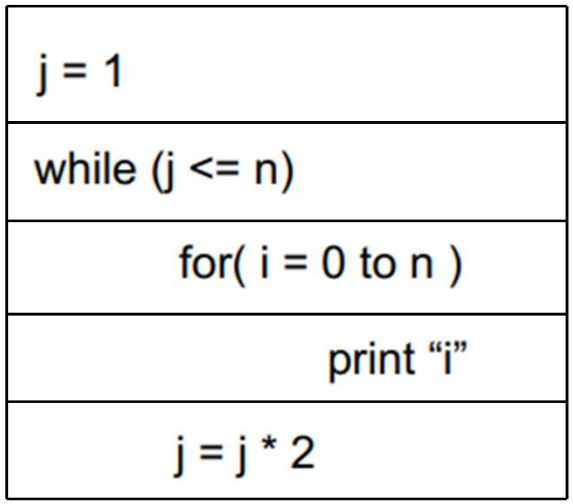
1.1. Introducción.

Instrucción	Costo
1 s ← 0	c_1
2 i ← 1	c_2
3 while i<=n	c_3
4 t ← 0	c_4
5 j ← 1	c_5
6 while j<=i	c_6
7 t ← t+1	c_7
8 j ← j+1	c_8
9 s ← s+t	c_9
10 i ← i+1	c_{10}

```
1  i ← 1
2  while i<=n
3      k ← i
4      while k<=n
5          k ← k+1
6      k ← 1
7      while k<=i
8          k ← k+1
9      i ← i+1
```

```
1  i ← 1
2 while i<=n
3      k ← i
4      while k<=n
5          k ← k+2
6      k ← 1
7      while k<=i
8          k ← k+1
9      i ← i+2
```

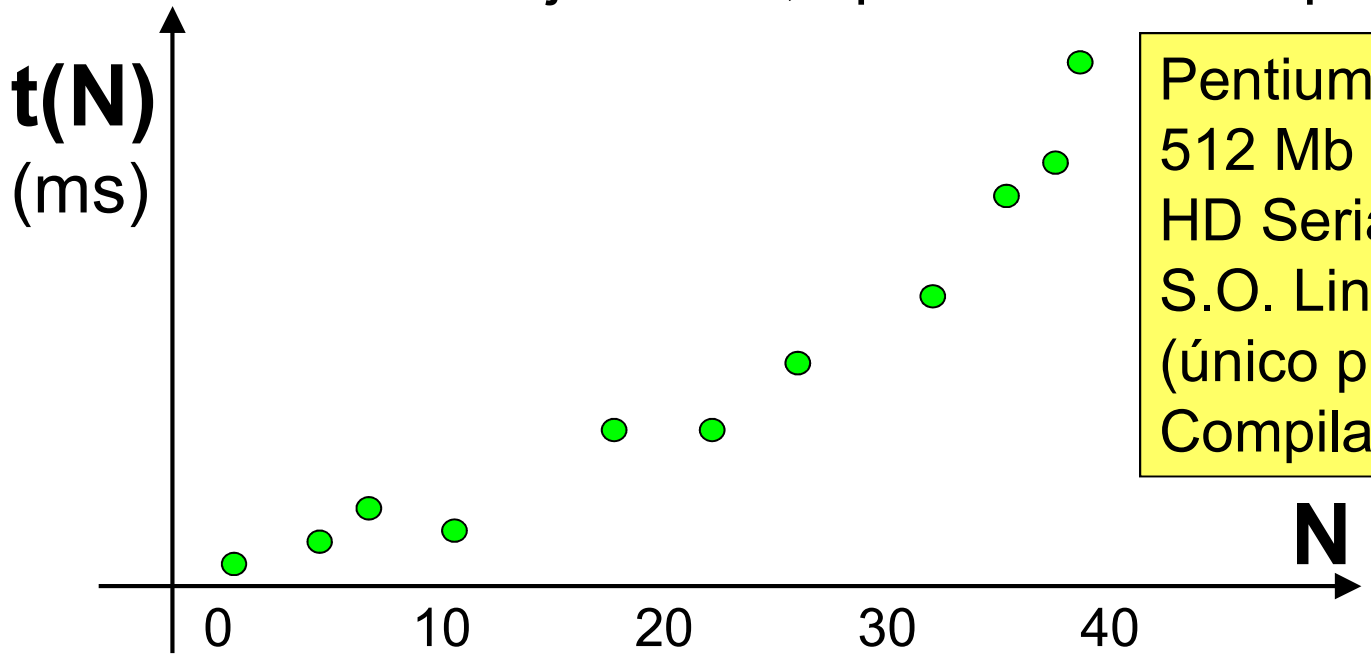

i = 3
while (i < =n)
k = 1
while(k < n)
if (k % 2)
print “k”
k = k + 1
i = i + 1


j = 1
while (j <= n)
for(i = 0 to n)
print “i”
j = j * 2

```
i = 3
j = 4
while (i < n)
        j = i
        while (j <= 2n)
                j = j * 2
        i = i + 1
```

1.1. Introducción.

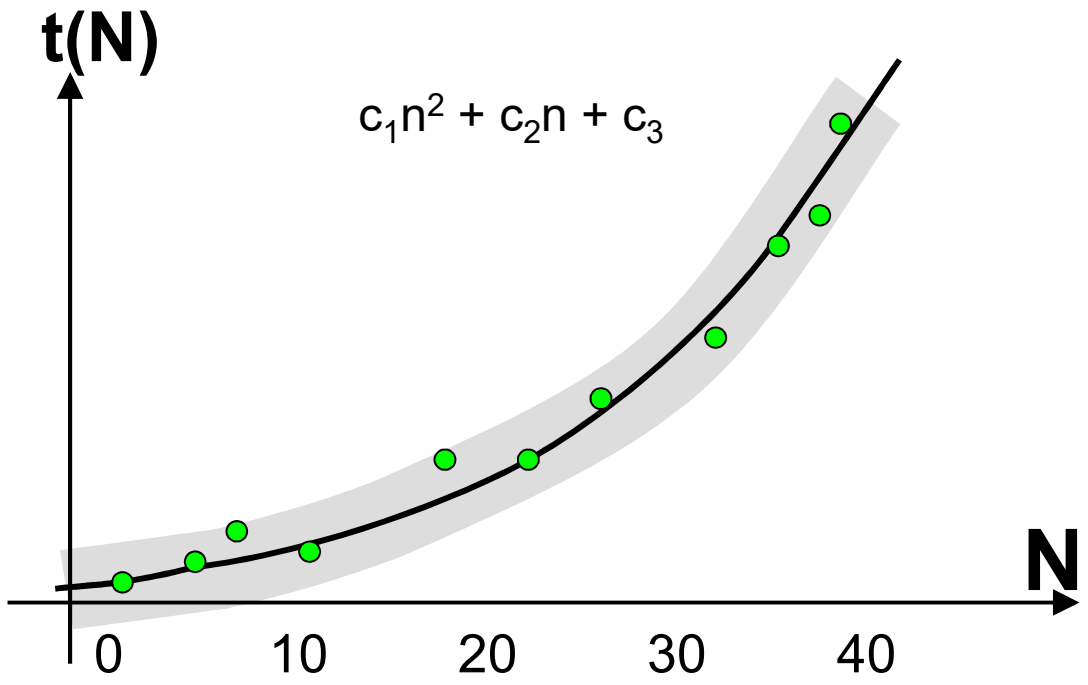
- El análisis de algoritmos también puede ser **a posteriori:** implementar el algoritmo y contar lo que tarda para distintas entradas.
- En este caso, cobran especial importancia las **herramientas** de la **estadística**: representaciones gráficas, técnicas de muestreo, regresiones, tests de hipótesis, etc.
- Hay que ser muy **específicos**, indicar: ordenador, S.O., condiciones de ejecución, opciones de compilación, etc.



Pentium IV a 2,66Mhz
512 Mb de RAM, DDR
HD Serial ATA, 60 Gb.
S.O. Linux RedHat 8
(único proceso en ejec.)
Compilado con -o3

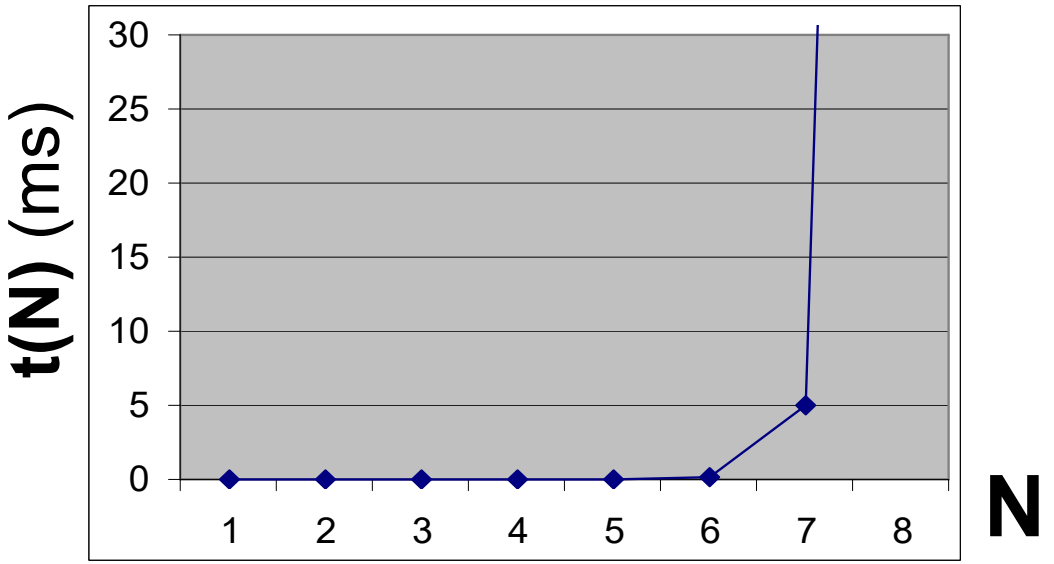
1.1. Introducción.

- Indicamos los **factores externos**, porque influyen en los tiempos (multiplicativamente), y son útiles para comparar tiempos tomados bajo condiciones distintas.
- La medición de los tiempos es un **estudio experimental**.
- El análisis a posteriori suele complementarse con un **estudio teórico** y un **contraste teórico/experimental**.
- **Ejemplo**. Haciendo el estudio teórico del anterior programa, deducimos que su tiempo es de la forma: $c_1n^2 + c_2\,n + c_3$
- Podemos hacer una regresión. → ¿Se ajusta bien? ¿Es correcto el estudio teórico?



1.1. Introducción.

- El contraste teórico/experimental **permite**: detectar posibles errores de implementación, hacer previsiones para tamaños inalcanzables, comparar implementaciones.
- Sin el estudio teórico, extraer conclusiones relevantes del tiempo de ejecución puede ser complejo.
- **Ejemplo.** Programa "cifras.exe":
  - N= 4, T(4)= 0.1 ms
  - N= 5, T(5)= 5 ms
  - N= 6, T(6)= 0.2 s
  - N= 7, T(7)= 10 s
  - N= 8, T(8)= 3.5 min



- ¿Qué conclusiones podemos extraer?
- El **análisis a priori** es siempre un estudio teórico previo a la implementación. Puede servir para evitar la implementación, si el algoritmo es poco eficiente.