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.

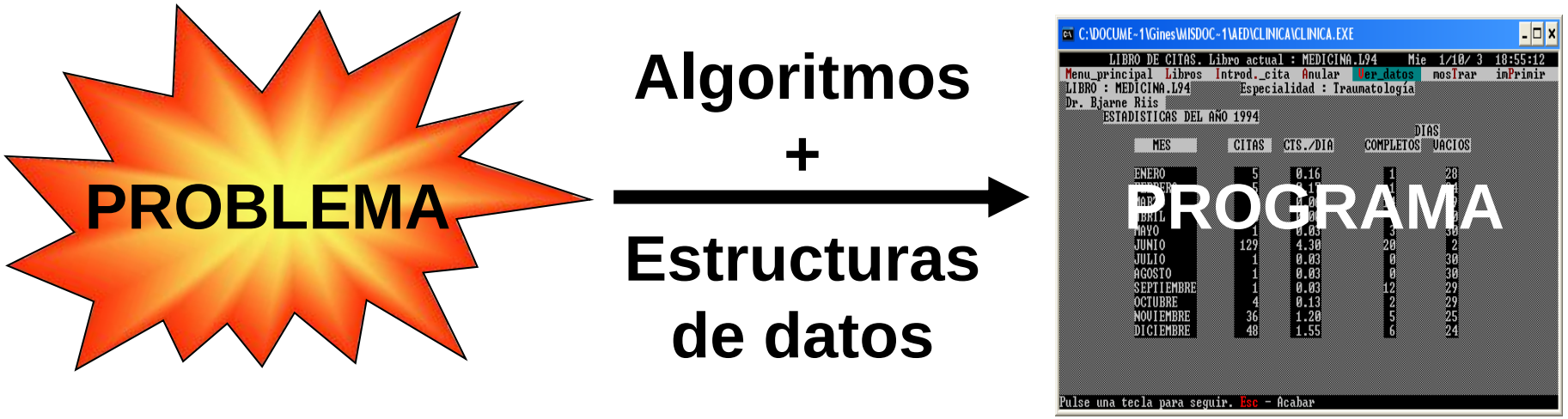
0.1. Definición y propiedades.

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?

0.1. Definición y propiedades.

ARQUITECTO	INFORMÁTICO
1. Estudio de viabilidad, análisis del terreno, requisitos pedidos, etc.	1. **Análisis** del problema
2. Diseñar los planos del puente y asignar los materiales.	2. **Diseño** del programa (alg. y estr.)
3. Poner los ladrillos de acuerdo con los planos.	3. **Implementación** (programación)
4. Supervisión técnica del puente.	4. **Verificación** y pruebas

0.1. Definición y propiedades.

MÉTODO CIENTÍFICO		**INFORMÁTICO**
1.Observación.	⟷	1. **Análisis** del problema
2.Hipótesis.	⟷	2. **Diseño** del programa (alg. y estr.)
3.Experimentación.	⟷	3. **Implementación** (programación)
4.Verificación.	⟷	4. **Verificación** y pruebas

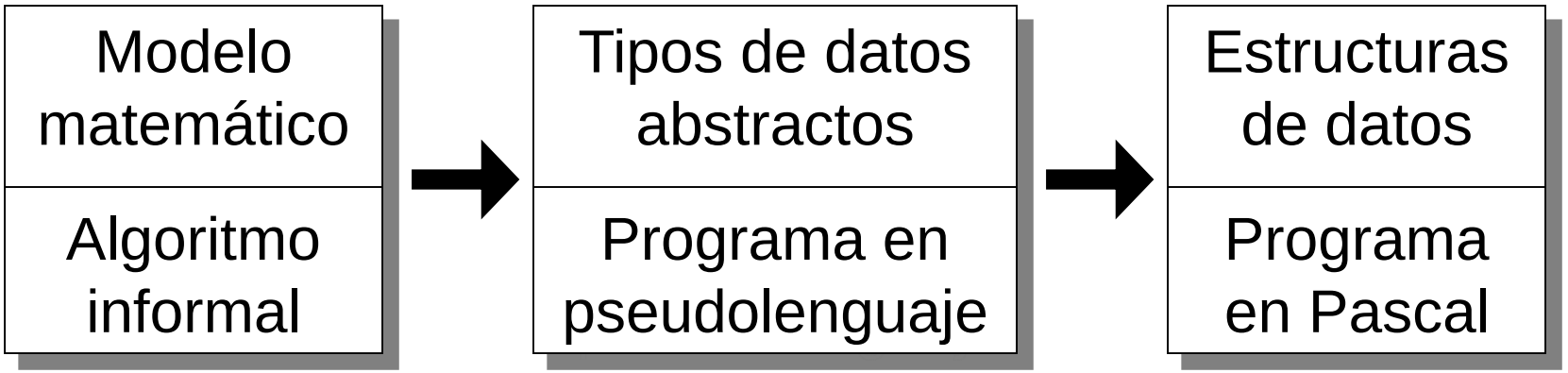
0.1. Definición y propiedades.

Otras ideas...

- **Refinamiento por pasos sucesivos.**
  - Escribir la estructura de la solución en pseudocódigo, de manera muy genérica.
  - Especificar los pasos de forma cada vez más detallada, y precisa.
  - Repetimos el refinamiento hasta llegar a una implementación.

0.1. Definición y propiedades.

- **Proceso de resolución propuesto por Aho.**



- Más en las asignaturas de Ingeniería del Software...

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.

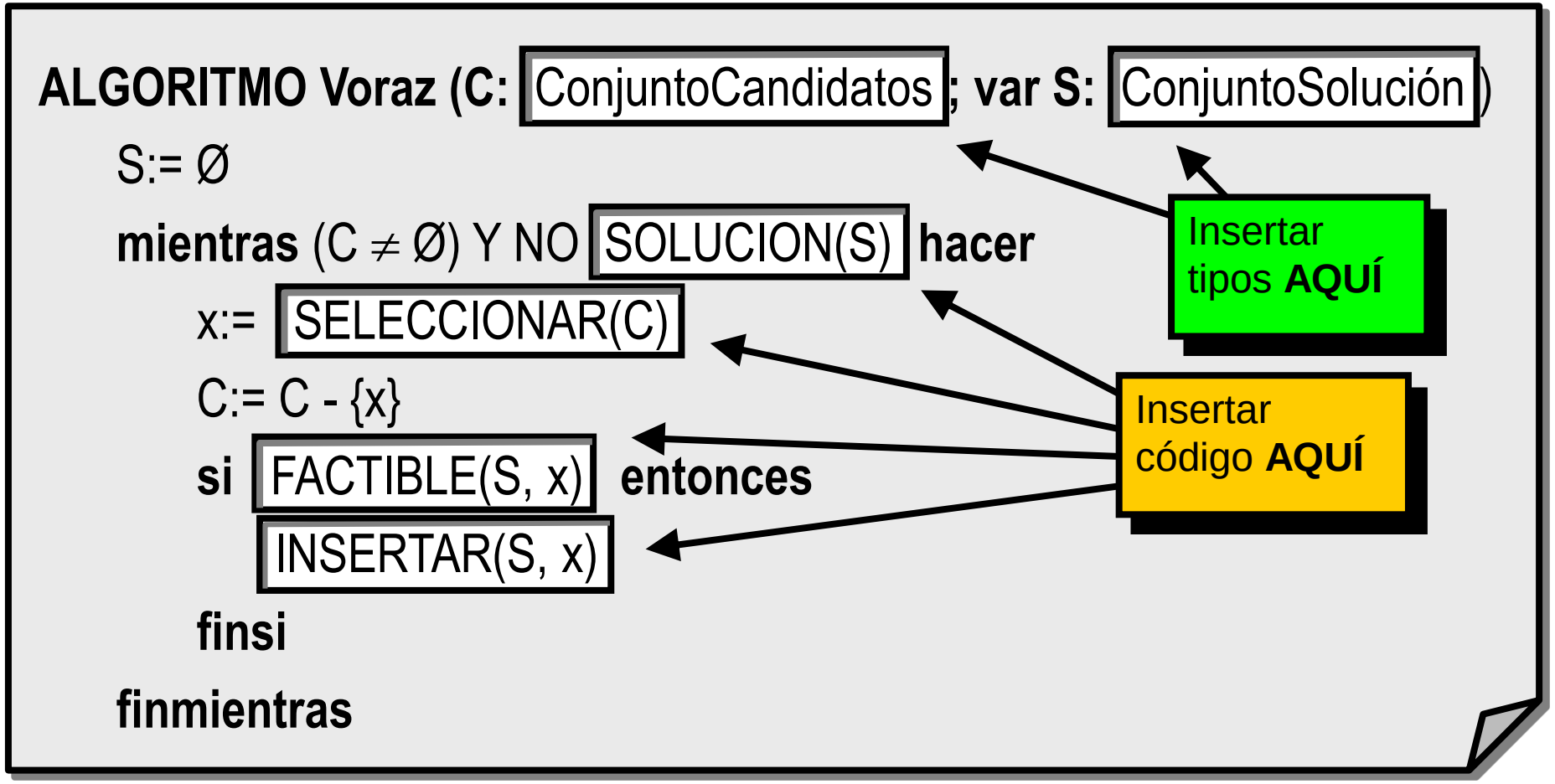
- **Ejemplo.** Analizar el tiempo de ejecución y el orden de complejidad del siguiente algoritmo.

```
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.

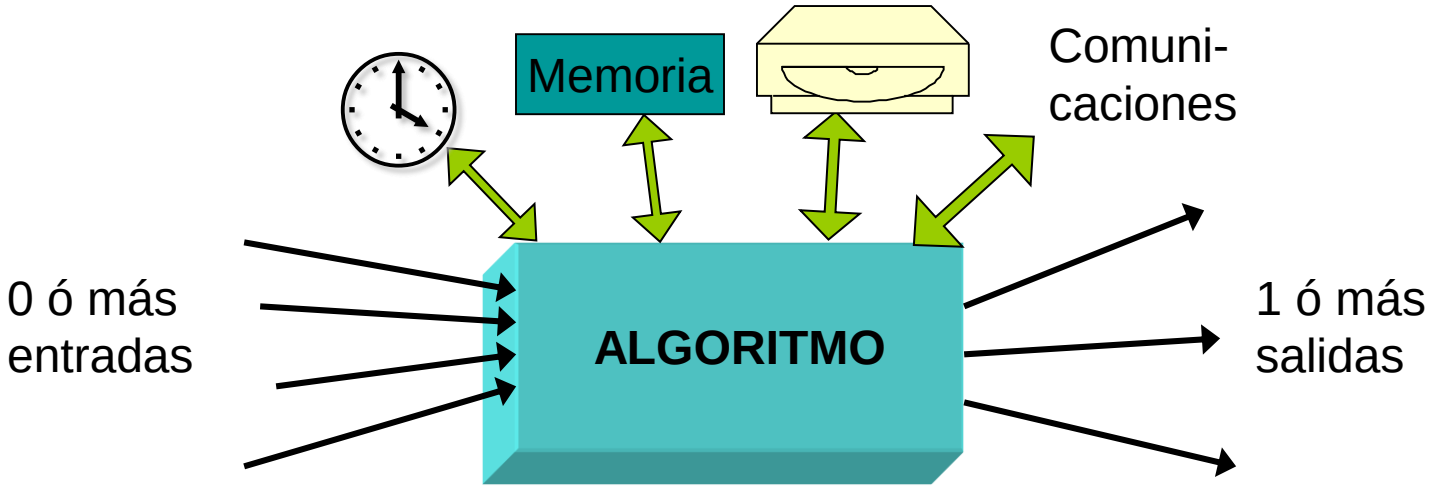
0.2. Análisis y diseño de algoritmos.

- **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).

```
for i:= 1 to N
   for j:= 1 to N
      suma:= 0
      for k:= 1 to N
         suma:=suma+a[i,k]*a[k,j]
      end
      c[i, j]:= suma
   end
end
```

```
Funcion Fibonacci (N: int): int;
if N<0 then
   error('No válido')
case N of
   0, 1: return N
else
   fnm2:= 0
   fnm1:= 1
   for i:= 2 to N
      fn:= fnm1 + fnm2
      fnm2:= fnm1
      fnm1:= fn
   end
   return fn
end
```

1.1. Introducción.

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

```
A[0, (n-1) div 2]:= 1
key:= 2
i:= 0
j:= (n-1) div 2
cuadrado:= n*n
while key<=cuadrado do
   k:= (i-1) mod n
   l:= (j-1) mod n
   if A[k, l] ≠ 0 then
      i:= (i + 1) mod n
   else
      i:= k
      j:= l
   end
   A[i, j]:= key
   key:= key+1
end
```

```
for i:= 1 to N do
   if Impar(i) then
      for j:= i to n do
         x:= x + 1
    else
      for j:= 1 to i do
         y:= y + 1
      end
   end
end
```

1.1. Introducción.

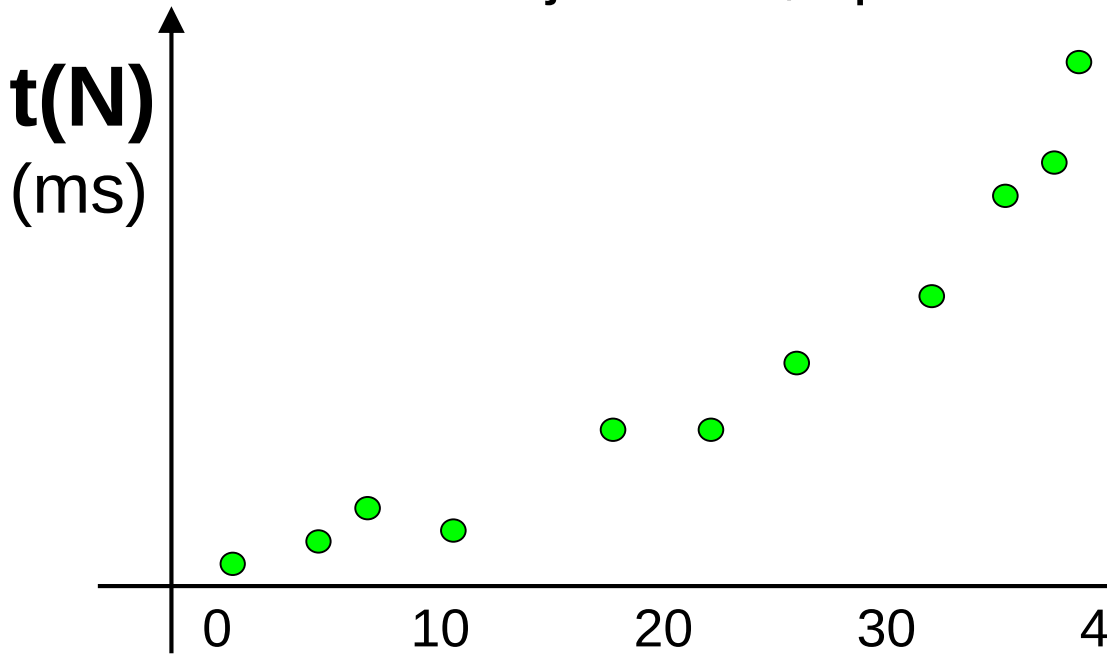
- **Ejemplos**. Estudiar t(n) en el caso promedio, para las instrucciones de asignación. Usar probabilidades.

```
i:= 1
mientras i ≤ n hacer
    si a[i] ≥ a[n] entonces
        a[n]:=a[i]
    finsi
    i:= i *2
finmientras
```

```
cont:=0
para i:= 1,...,n hacer
    para j:= 1,...,i-1 hacer
        si a[i] < a[j] entonces
            cont:= cont + 1
        finsi
    finpara
finpara
```

1.1. Introducción.

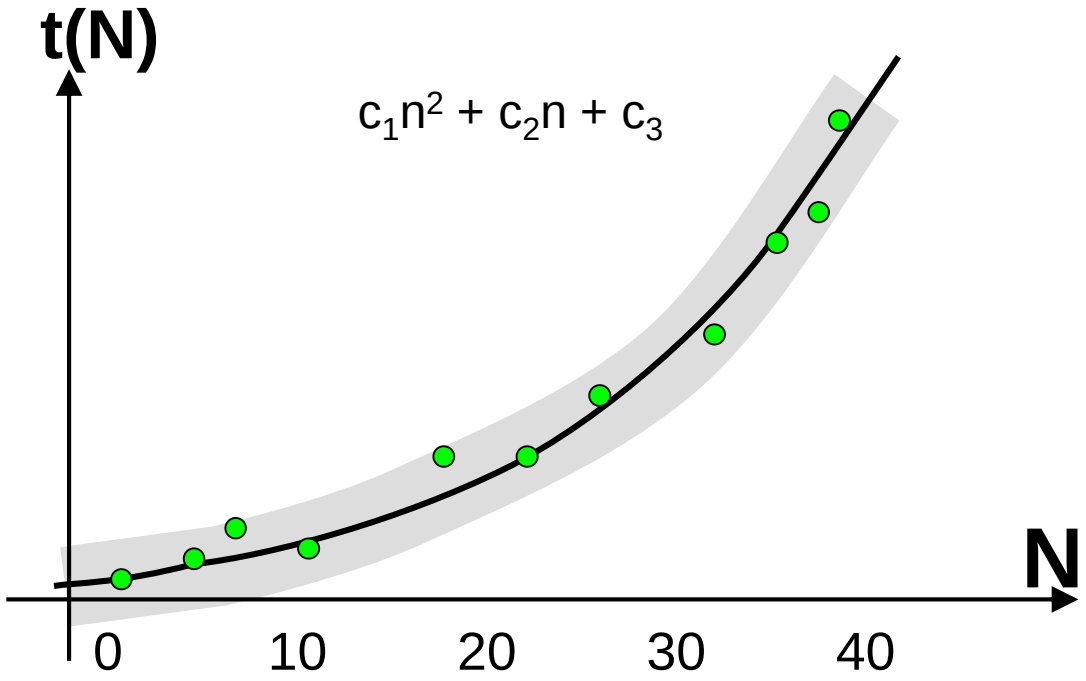
- El análisis de algoritmos también puede ser **a posteriori:** implementar el algoritmo y contar lo que tarda para distintas entradas.
- En este caso, cobran especial importancia las **herramientas** de la **estadística**: representaciones gráficas, técnicas de muestreo, regresiones, tests de hipótesis, 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

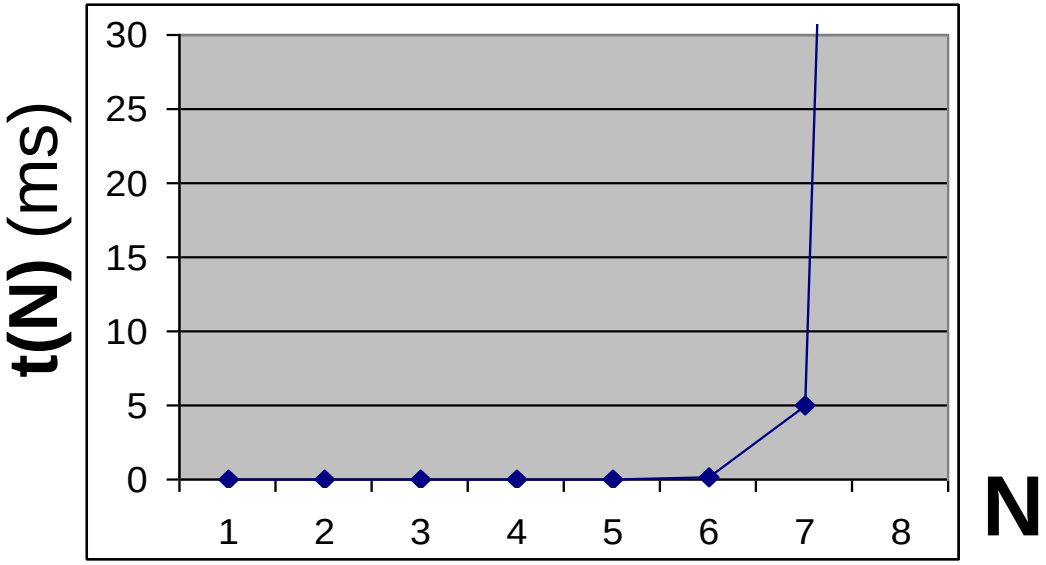
1.1. Introducción.

- 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 re-gresión. → ¿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.

1.2. Notaciones asintóticas.

- El tiempo de ejecución **t(n)** está dado en base a unas constantes que dependen de factores externos.
- Nos interesa un análisis que sea independiente de esos factores.
- **Notaciones asintóticas:** Indican como crece **t**, para valores suficientemente grandes (asintóticamente) sin considerar constantes.
- O(t): Orden de complejidad de t.
- Ω(t): Orden inferior de t, u omega de t.
- Θ(t): Orden exacto de t.

1.2.1. Definiciones.

Orden de complejidad de f(n): O(f)

- Dada una función **f: N $\rightarrow$ R$^+$**, llamamos **orden de f** al conjunto de todas las funciones de **N** en **R$^+$** acotadas superiormente por un múltiplo real positivo de **f**, para valores de **n** suficientemente grandes.

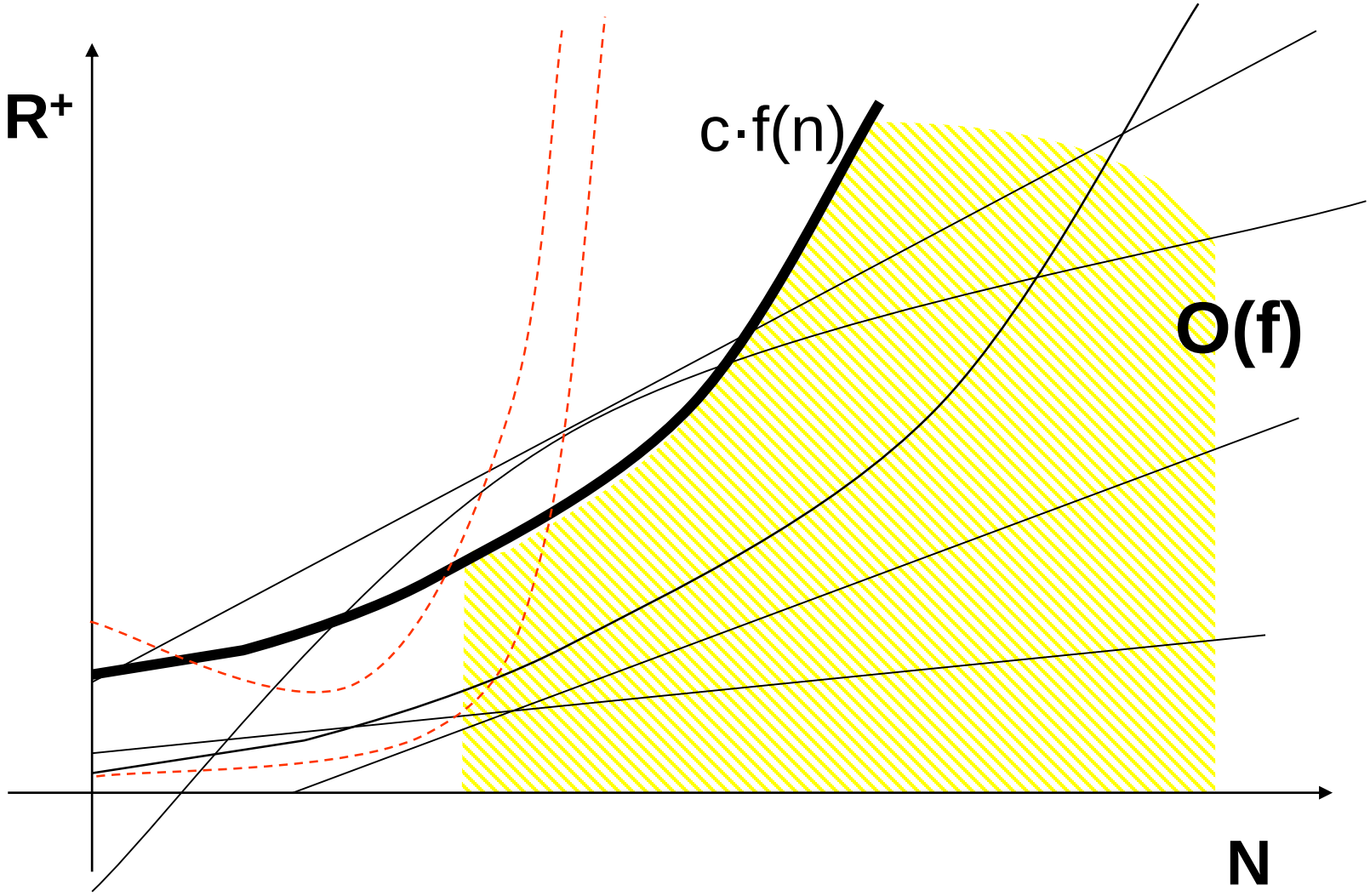
$$O(f)= \{ t: N \rightarrow R^+ / \exists\, c \in R^+, \exists\, n_0 \in N, \forall\, n \geq n_0; \; t(n) \leq c \cdot f(n) \}$$

1.2.1. Definiciones.

Observaciones:

- **O(f)** es un **conjunto de funciones**, no una función.
- “Valores de n suficientemente grandes...”: no nos importa lo que pase para valores pequeños.
- “Funciones acotadas superiormente por un múltiplo de f...”: nos quitamos las constantes multiplicativas.
- La definición es aplicable a cualquier función de N en R, no sólo tiempos de ejecución.

1.2.1. Definiciones.

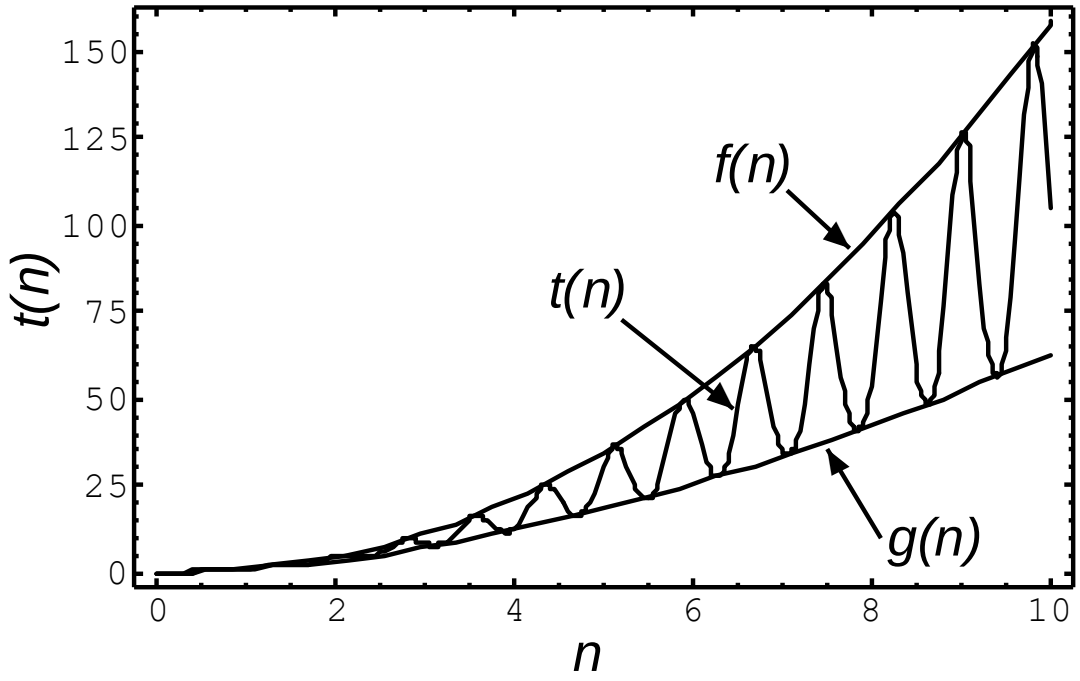


1.2.1. Definiciones.

Uso de los órdenes de complejidad

- 1) Dado un tiempo **t(n)**, encontrar la función **f** más simple tal que **t** $\in$ **O(f)**, y que más se aproxime asintóticamente.
- **Ejemplo**. $t(n) = 2n^2/5 + 6n + 3\pi \cdot \log_2 n + 2 \Rightarrow t(n) \in O(n^2)$

- 2) Acotar una función difícil de calcular con precisión.
- **Ejemplo.** $t(n) \in O(f(n))$

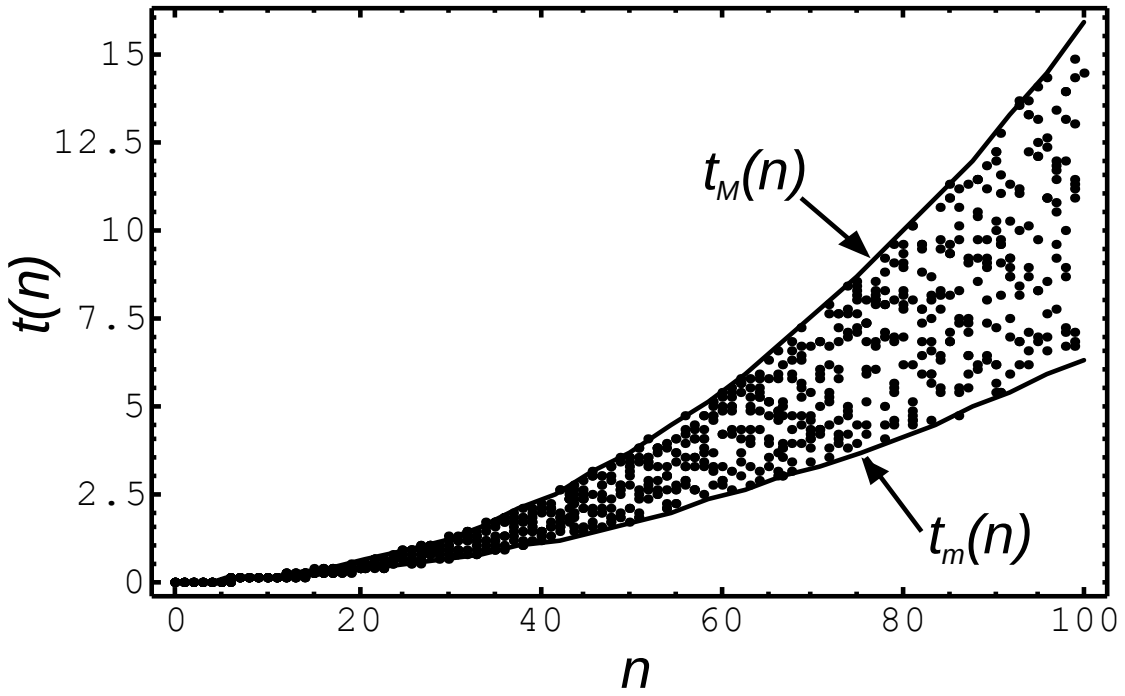


1.2.1. Definiciones.

Uso de los órdenes de complejidad

- 3) Acotar una función que no tarda lo mismo para el mismo tamaño de entrada (distintos casos, mejor y peor).

- **Ejemplo.**
  $t(n) \in O(t_M(n))$



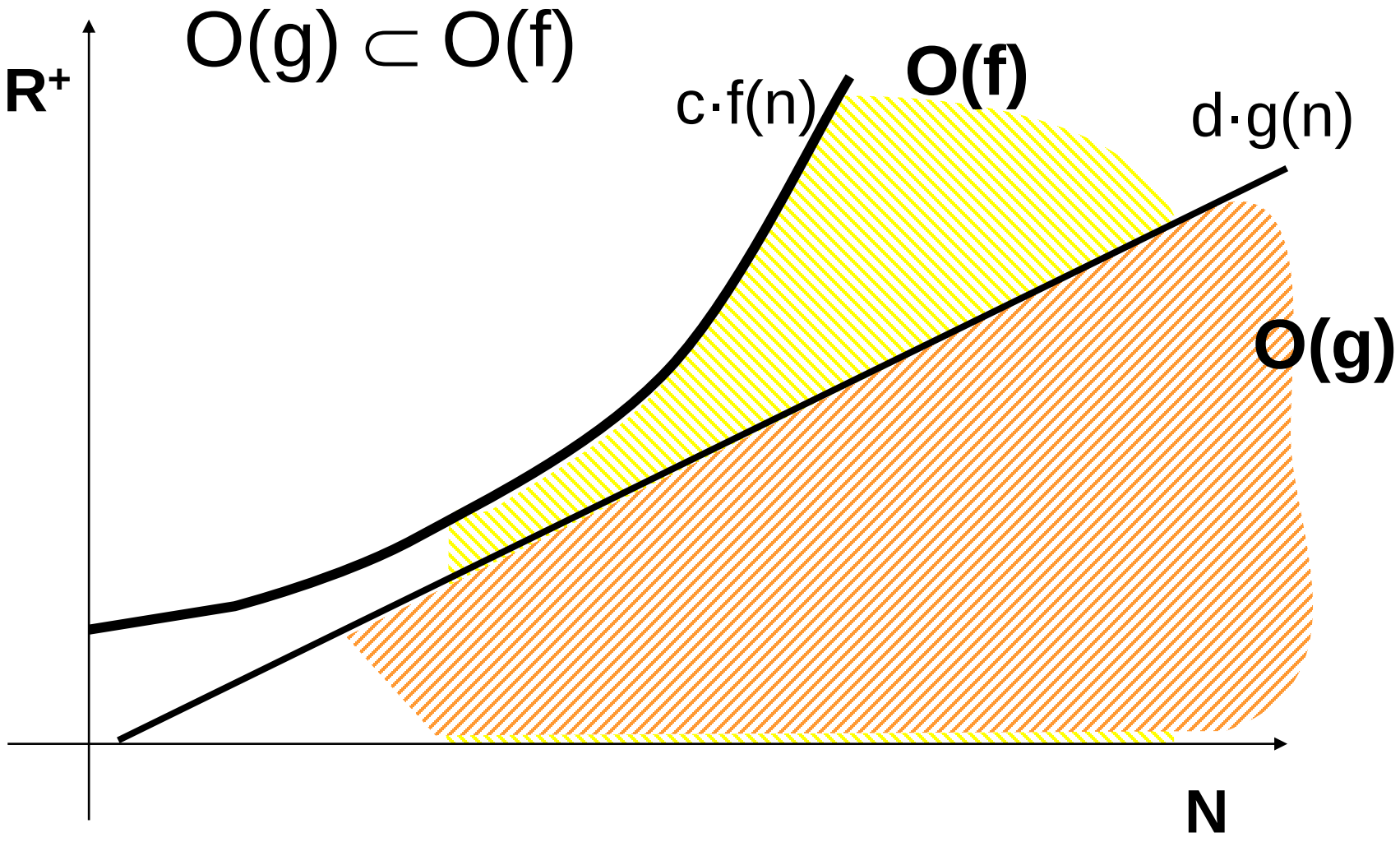
- Igual que con la cota superior, podríamos hacer con la cota inferior...

1.2.1. Definiciones.

- **Relación de orden entre O(..) = Relación de inclusión entre conjuntos.**
  - $O(f) \leq O(g) \Leftrightarrow O(f) \subseteq O(g) \Leftrightarrow$ Para toda $t \in O(f)$, $t \in O(g)$

- Se cumple que:
  - $O(c) = O(d)$, siendo **c** y **d** constantes positivas.
  - $O(c) \subset O(n)$
  - $O(cn + b) = O(dn + e)$
  - $O(p) = O(q)$, si **p** y **q** son polinomios del mismo grado.
  - $O(p) \subset O(q)$, si **p** es un polinomio de menor grado que **q**.

1.2.1. Definiciones.



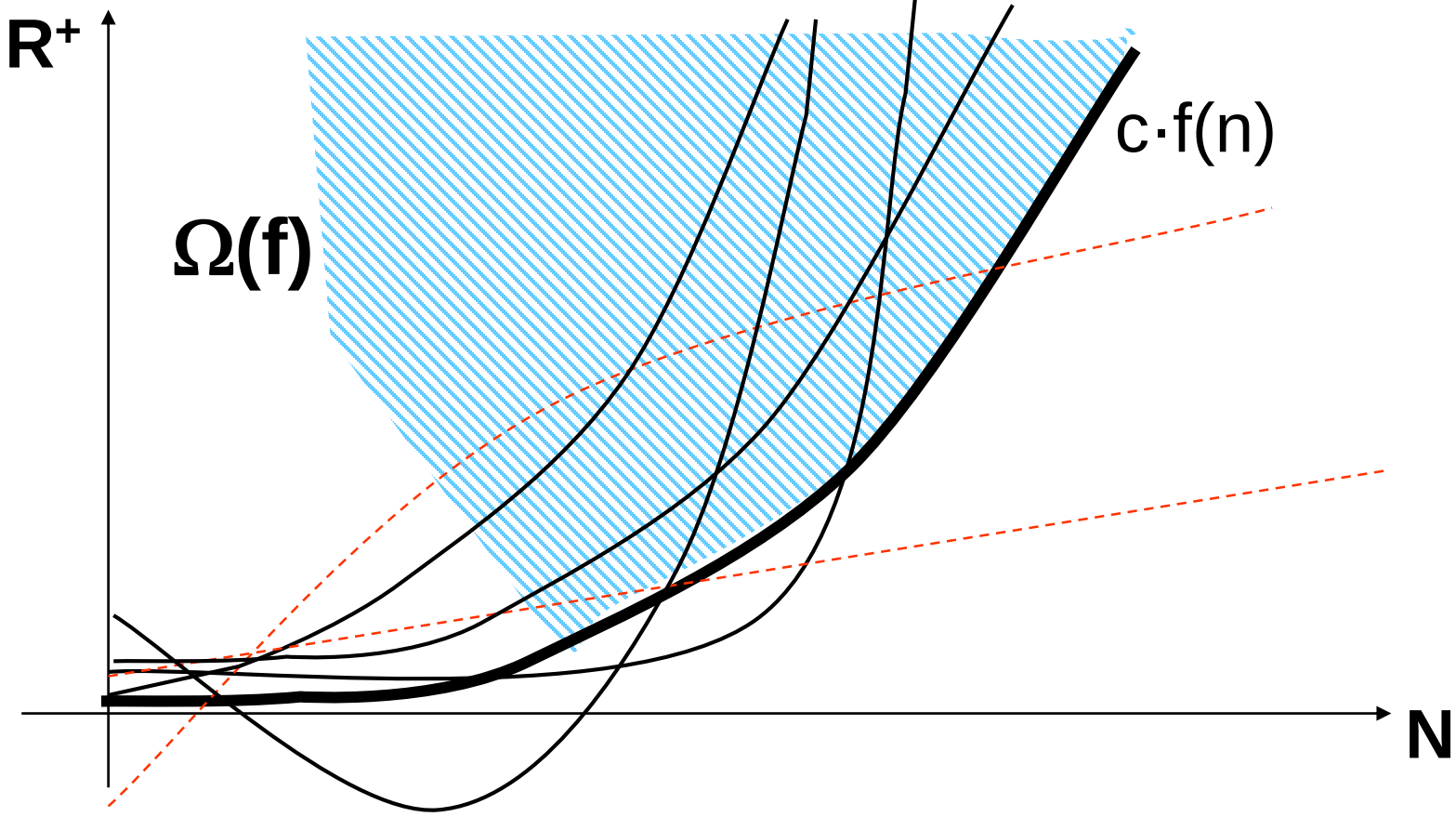
1.2.1. Definiciones.

Orden inferior u omega de f(n): $\Omega(f)$

- Dada una función **f: N $\rightarrow$ R^+**, llamamos **omega de f** al conjunto de todas las funciones de **N** en **R^+** acotadas **inferiormente** por un múltiplo real positivo de **f**, para valores de **n** suficientemente grandes.

$$\Omega(f) = \{ t: N \rightarrow R^+ / \exists\, c \in R^+, \exists\, n_0 \in N, \forall\, n \geq n_0; \; t(n) \geq c \cdot f(n) \}$$

1.2.1. Definiciones.



- La notación omega se usa para establecer cotas inferiores del tiempo de ejecución.
- **Relación de orden:** igual que antes, basada en la inclusión.

1.2.1. Definiciones.

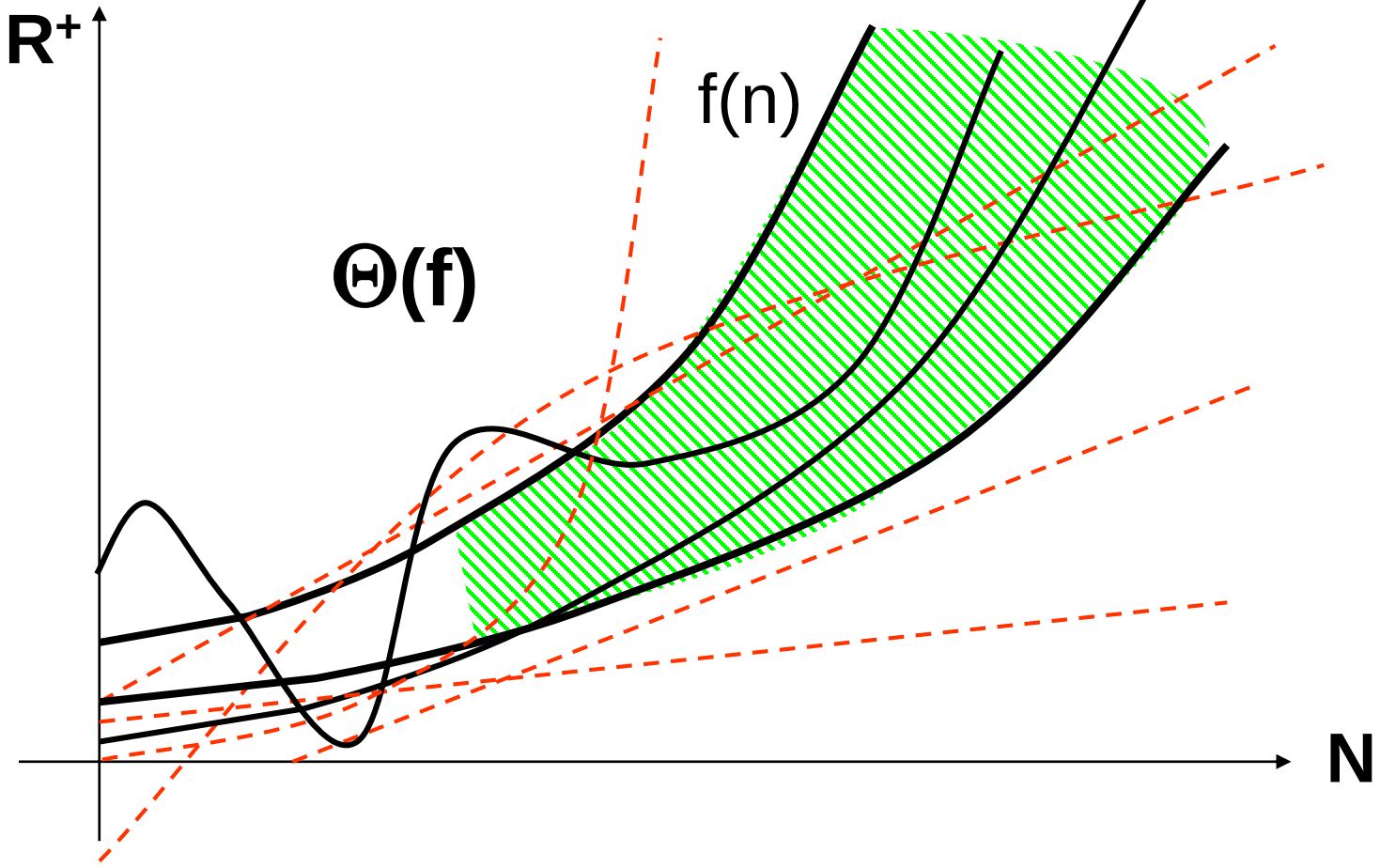
Orden exacto de f(n): Θ(f)

- Dada una función **f: N → R⁺**, llamamos orden **exacto de f** al conjunto de todas las funciones de **N** en **R⁺** que crecen igual que f, asintóticamente y salvo constantes.

$$\Theta(f) = O(f) \cap \Omega(f) =$$

$$= \{ t: N \to R^+ / \exists\, c, d \in R^+, \exists\, n_0 \in N, \forall\, n \geq n_0;$$
$$c \cdot f(n) \geq t(n) \geq d \cdot f(n) \}$$

1.2.1. Definiciones.



- Si un algoritmo tiene un **t** tal que t ∈ O(f) y t ∈ Ω(f), entonces **t ∈ Θ(f)**.

1.2.1. Definiciones.

- **Ejemplos. ¿Cuáles son ciertas y cuáles no?**

$3n^2 \in O(n^2)$	$n^2 \in O(n^3)$	$n^3 \in O(n^2)$
$3n^2 \in \Omega(n^2)$	$n^2 \in \Omega(n^3)$	$n^3 \in \Omega(n^2)$
$3n^2 \in \Theta(n^2)$	$n^2 \in \Theta(n^3)$	$n^3 \in \Theta(n^2)$
$2^{n+1} \in O(2^n)$	$(2+1)^n \in O(2^n)$	$(2+1)^n \in \Omega(2^n)$
$O(n) \in O(n^2)$	$(n+1)! \in O(n!)$	$n^2 \in O(n!!)$

1.2.1. Definiciones.

Notación o pequeña de f(n): o(f)

- Dada una función **f: N $\rightarrow$ R$^+$**, llamamos **o pequeña de f** al conjunto de todas las funciones de **N** en **R$^+$** que crecen igual que f asintóticamente:

$$\mathbf{o}(f) = \{ t: N \rightarrow R^+ / \lim_{n \rightarrow \infty} t(n)/f(n) = 1 \}$$

- Esta notación conserva las constantes multiplicativas para el término de mayor orden.

1.2.1. Definiciones.

Notación o pequeña de f(n): o(f)

- **Ejemplo**. $t(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$

  $t(n) \in o(a_m n^m) \neq o(n^m)$

- $t(n) = 3{,}2n^2 + 8n - 9 \in o(¿?)$
- $t(n) = 82\ n^4 + 3 \cdot 2^n + 91 \log_2 n \in o(¿?)$
- $t(n) = 4n^3 + 3n^3 \log_2 n - 7n^2 + 8 \in o(¿?)$
- $¿o(t) \subseteq O(t)?$

1.2.2. Propiedades de las notaciones asintóticas.

- **P1. Transitividad.**
  Si $f \in O(g)$ y $g \in O(h)$ entonces $f \in O(h)$.

  – Si $f \in \Omega(g)$ y $g \in \Omega(h)$ entonces $f \in \Omega(h)$
  – Ej. $2n+1 \in O(n)$, $n \in O(n^2) \Rightarrow 2n+1 \in O(n^2)$

- **P2.** Si $f \in O(g)$ entonces $O(f) \subseteq O(g)$.

  – ¿Cómo es la relación para los Ω?

1.2.2. Propiedades de las notaciones asintóticas.

- **P3. Relación pertenencia/contenido.**
  Dadas f y g de N en R^+, se cumple:
  - i) $O(f) = O(g) \Leftrightarrow f \in O(g)$ y $g \in O(f)$
  - ii) $O(f) \subseteq O(g) \Leftrightarrow f \in O(g)$

- ¿La relación de orden entre O(..) es completa? Dadas f y g, ¿se cumple $O(f) \subseteq O(g)$ ó $O(g) \subseteq O(f)$?

1.2.2. Propiedades de las notaciones asintóticas.

- **P4. Propiedad del máximo.**
  Dadas f y g, de N en R^+, $O(f+g) = O(\max(f, g))$.

  - Con omegas: $\Omega(f+g) = \Omega(\max(f, g))$
  - ¿Y para los $\Theta(f+g)$?
  - ¿Es cierto que $O(f - g) = O(\max(f, -g))$?
  - Ejemplo: $O(2^n + n^6 + n!) = ...$

- ¿Qué relación hay entre $O(\log_2 n)$ y $O(\log_{10} n)$?

1.2.2. Propiedades de las notaciones asintóticas.

- **P5. Equivalencia entre notaciones.**
  Dadas f y g de N en R^+, $O(f)=O(g) \Leftrightarrow \Theta(f)=\Theta(g) \Leftrightarrow f \in \Theta(g) \Leftrightarrow \Omega(f)=\Omega(g)$

- **P6. Relación límites/órdenes.**
  Dadas f y g de N en R^+, se cumple:
  - i) $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in R^+ \Rightarrow O(f) = O(g)$
  - ii) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow O(f) \subset O(g)$
  - iii) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow O(f) \supset O(g)$

1.2.3. Notaciones con varios parámetros.

- En general, el tiempo y la memoria consumidos pueden depender de muchos parámetros.
- **$f: N^m \rightarrow R^+$** ($f: Nx...^m..xN \rightarrow R^+$)

Orden de complejidad de $f(n_1, n_2, ..., n_m)$: $O(f)$

- Dada una función **$f: N^m \rightarrow R^+$**, llamamos **orden de f** al conjunto de todas las funciones de N^m en R^+ acotadas superiormente por un múltiplo real positivo de **f**, para valores de $(n_1, ..., n_m)$ suficientemente grandes.

$$O(f)= \{ t: N^m \rightarrow R^+ / \exists\, c \in R^+, \exists\, n_1, n_2, .., n_m \in N, \forall\, k_1 \geq n_1 ,$$
$$\forall\, k_2 \geq n_2 ,..,\forall\, k_m \geq n_m; t(k_1, k_2, ..., k_m) \leq c \cdot f(k_1, k_2, ..., k_m) \}$$

1.2.3. Notaciones con varios parámetros.

- **Ejemplo.** Tiempo de ejecución de la BPP con listas de adyacencia: **O(n+a)**.
  Memoria usada en una tabla hash: depende del número de cubetas, elementos, tamaño de celda...

- Podemos extender los conceptos de $\Omega(f)$ y $\Theta(f)$, para funciones con varios parámetros.
- Las propiedades se siguen cumpliendo $\rightarrow$ Demostrarlo.
- ¿Qué relación hay entre los siguientes órdenes?

  $O(n+m)$, $O(n^m)$ $O(n^2)$, $O(n+2^m)$

1.2.4. Notaciones condicionales.

- En algunos casos interesa estudiar el tiempo sólo para ciertos tamaños de entrada.
- **Ejemplo**. Algoritmo de búsqueda binaria: Si N es potencia de 2 el estudio se simplifica.

Orden condicionado de f(n): O(f | P)

- Dada una función **f: N $\rightarrow$ R$^+$**, y **P: N $\rightarrow$ B**, llamamos **orden de f según P** (o condicionado a P) al conjunto:

$$O(f \mid P) = \{ t: N \rightarrow R^+ / \exists\, c \in R^+, \exists\, n_0 \in N, \forall\, n \geq n_0; \; P(n) \Rightarrow t(n) \leq c \cdot f(n) \}$$

1.2.4. Notaciones condicionales.

- De igual forma, tenemos $\Omega(f \mid P)$ y $\Theta(f \mid P)$.

- **Ejemplo.**
  - Si estudiamos el tiempo para tamaños de entrada que sean potencia de 2:
    $t(n) \in O(f \mid n = 2^k)$
  - Para tamaños que sean múltiplos de 2:

    $t(n) \in O(f \mid n = 2k)$

- $O(f) = O(f \mid true)$.
- Para cualquier f y g, $f \in O(g \mid false)$.
- ¿$O(f) \leftrightarrow O(f \mid P)$?

1.2.5. Cotas de complejidad frecuentes.

- **Algunas relaciones entre órdenes frecuentes.**

  $O(1) \subset O(\log n) \subset O(n) \subset O(n \cdot \log n) \subset$
  $O(n \cdot (\log n)^2) \subset O(n^{1.001\ldots}) \subset O(n^2) \subset O(n^3) \subset \ldots$
  $\subset O(2^n) \subset O(n!) \subset O(n^n)$

- ¿Dónde va $O(3^n)$? ¿Y $O(n^3\, 2^n)$?
- ¿Qué pasa con las omegas? ¿Y con los órdenes exactos?

1.2.5. Cotas de complejidad frecuentes.

- El orden de un polinomio $a_n x^n + \ldots + a_1 x + a_0$ es $O(x^n)$.
- $\sum_{i=1}^{n} 1 \in O(n)$; $\sum_{i=1}^{n} i \in O(n^2)$ $\sum_{i=1}^{n} i^m \in O(n^{m+1})$
- Si hacemos una operación para n, otra para n/2, n/4, ..., aparecerá un orden logarítmico $O(\log_2 n)$.
- Los **logaritmos** son del mismo orden, independientemente de la base. Por eso, se omite normalmente.
- **Sumatorios:** se pueden aproximar con integrales, una acotando superior y otra inferiormente.
- Casos **promedios**: usar probabilidades.

1.3. Ecuaciones de recurrencia.

- Es normal que un algoritmo se base en procedimientos auxiliares, haga llamadas recursivas para tamaños menores o reduzca el tamaño del problema progresivamente.
- En el análisis, el tiempo t(n) se expresa en función del tiempo para t(n-1), t(n-2)...→ **Ecuaciones de recurrencia.**

- **Ejemplo**. ¿Cuántas operaciones **mover** se ejecutan?

```
Hanoi (n, i, j, k)
        if n>0 then
                Hanoi (n-1, i, k, j)
                mover (i, j)
                Hanoi (n-1, k, j, i)
        else
                mover (i, j)
```

1.3. Ecuaciones de recurrencia.

- En general, las ecuaciones de recurrencia tienen la forma:

$t(n) = b$	Para $0 \leq n \leq n_0$	**Casos base**
$t(n) = f(t(n), t(n-1), ..., t(n-k), n)$		En otro caso

- **Tipos de ecuaciones de recurrencia:**
  - Lineales y homegéneas:

    $a_0t(n) + a_1t(n-1) + ... + a_kt(n-k) = 0$
  - Lineales y no homegéneas:

    $a_0t(n) + a_1t(n-1) + ... + a_kt(n-k) = p(n) + ...$
  - No lineales:

    Ejemplo: $a_0t^2(n) + t(n-1)*t(n-k) + sqrt(t(n-2) + 1) = p(n)$

1.3.1. Ecuaciones lineales homogéneas.

- La ecuación de recurrencia es de la forma:

  $a_0 t(n) + a_1 t(n-1) + ... + a_k t(n-k) = 0$; a_i constante

- **Caso sencillo:**

$$t(n) = \begin{cases} 1 & \text{Si } n = 0 \\ x \cdot t(n-1) & \text{Si } n > 0 \end{cases}$$

- **Solución:** $t(n) = x^n$

1.3.1. Ecuaciones lineales homogéneas.

- Suponiendo que las soluciones son de la forma **$t(n) = x^n$**, la ecuación de recurrencia homogénea:

$$a_0 t(n) + a_1 t(n-1) + \ldots + a_k t(n-k) = 0$$

- Se transforma en:

$$a_0 x^n + a_1 x^{n-1} + \ldots + a_k x^{n-k} = 0 \Rightarrow /x^{n-k} \Rightarrow$$

$$\mathbf{a_0 x^k + a_1 x^{k-1} + \ldots + a_k = 0}$$

Ecuación característica de la ecuación recurrente lineal homogénea

1.3.1. Ecuaciones lineales homogéneas.

$$\mathbf{a_0x^k + a_1x^{k-1} + ... + a_k = 0}$$

Ecuación característica de la ecuación recurrente lineal homogénea

- **k**: conocida. $\mathbf{a_i}$: conocidas. **x**: desconocida.
- Resolver el sistema para la incógnita **x**. El resultado es:

  $t(n) = x^n$

- Pero... Un polinomio de grado **k** tendrá **k** soluciones...

1.3.1. Ecuaciones lineales homogéneas.

- Sean las soluciones x= (s_1, s_2, ..., s_k), todas distintas.
- La solución será:

$$\mathbf{t(n) = c_1 \cdot s_1^n + c_2 \cdot s_2^n + \ldots + c_k \cdot s_k^n = \sum_{i=1}^{k} c_i \cdot s_i^n}$$

- Siendo $\mathbf{c_i}$ constantes, cuyos valores dependen de los casos base (condiciones iniciales).

- Son constantes que añadimos nosotros. Debemos resolverlas, usando los casos base de la ecuación recurrente.

1.3.1. Ecuaciones lineales homogéneas.

- **Ejemplo**. El tiempo de ejecución de un algoritmo es:

$$t(n) = \begin{cases} 0 & \text{Si } n = 0 \\ 1 & \text{Si } n = 1 \\ 3 \cdot t(n-1) + 4 \cdot t(n-2) & \text{Si } n > 1 \end{cases}$$

- Encontrar una fórmula explícita para t(n), y calcular el orden de complejidad del algoritmo.

- ¿Qué pasa si no todas las soluciones son distintas?

1.3.1. Ecuaciones lineales homogéneas.

- Si no todas las soluciones $x= (s_1, s_2, ..., s_k)$ son distintas, entonces el polinomio característico será:

  $a_0x^n + a_1x^{n-1} + ... + a_kx^{n-k} = (x - s_1)^m \cdot (x - s_2) \cdot ... (x - s_p) \cdot x^{n-k}$

- ¿Cuál es la solución para **t(n)**?

- Las derivadas valen 0 en s_1, hasta la m-1-ésima.

  $a_0n \cdot x^{n-1} + a_1(n-1) \cdot x^{n-2} + ... + a_k(n-k) \cdot x^{n-k-1} = 0 \Rightarrow \cdot x \Rightarrow$

  $a_0n \cdot x^n + a_1(n-1) \cdot x^{n-1} + ... + a_K(n-k)x^{n-k} = 0$

1.3.1. Ecuaciones lineales homogéneas.

- Las derivadas valen 0 en s_1, hasta la m-1-ésima.
- **Conclusión: $t(n) = n \cdot s_1^n$** también será solución de la ecuación característica.
- Para la segunda derivada: **$t(n) = n^2 s_1^n$** será solución...
- Si **s_i** tiene multiplicidad **m**, entonces tendremos:

  s_i^n $n \cdot s_i^n$ $n^2 \cdot s_i^n$ ... $n^{m-1} \cdot s_i^n$

1.3.1. Ecuaciones lineales homogéneas.

- Dadas las soluciones x= (s_1, s_2, ..., s_k) siendo $\mathbf{s_k}$ de multiplicidad **m**, la solución será:

  $$\mathbf{t(n)} = c_1 \cdot s_1^n + c_2 \cdot s_2^n + \ldots + c_k \cdot s_k^n + c_{k+1} \cdot n \cdot s_k^n +$$
  $$+ c_{k+2} \cdot n^2 \cdot s_k^n + \ldots + c_{k+1+m} \cdot n^{m-1} \cdot s_k^n$$

- **Ejemplo.** Calcular t(n) y el orden de complejidad para:

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

  $t(0) = 0,\ t(1) = 3,\ t(2) = 10$

1.3.2. Recurrencias no homogéneas.

- ¿Qué pasa si tenemos algo como $t(n) = 2 \cdot t(n-1) + 1$?

- Términos que no tienen $t(x) \rightarrow$ **Recurrencia no homogénea.**

- **Ejemplo.** Calcular $t(n)$ para: $t(n) = 2t(n-1) + 3^n(n+1)$
  - $t(n) - 2t(n-1) = 3^n(n+1) \Rightarrow$
  - $t(n+1) - 5t(n) + 6t(n-1) = 3^{n+1} \Rightarrow$
  - $t(n+2) - 8t(n+1) + 21t(n) - 18t(n-1) = 0 \Rightarrow$

Ecuación característica: $(x-2)(x-3)^2 = 0$

1.3.2. Recurrencias no homogéneas.

- **Conclusión:** Si en la ecuación de recurrencia aparece un término de la forma $\mathbf{b^n \cdot p(n)}$ (p(n) polinomio de n), entonces en la ecuación característica habrá un factor:

  $(x-b)^{\mathbf{Grado(p(n))+1}} \rightarrow$ Sol. **b** con multiplicidad Grado(p(n))+1

- **Ejemplo:** $t(n) - t(n-3) = 2 + n^3 + n^2 \cdot 3^n + 2^{(n+1)} + 8n^2$
- ¿Cuál es la ecuación característica?

1.3.2. Recurrencias no homogéneas.

- En general, tendremos recurrencias de la forma:

  $a_0t(n) + a_1t(n-1) + ... + a_kt(n-k) = b_1^n p_1(n) + b_2^n p_2(n) + ...$

- Y la ecuación característica será:

  $(a_0x^k + a_1x^{k-1} + ... + a_k)(x-b_1)^{G(p_1(n))+1}(x-b_2)^{G(p_2(n))+1}... = 0$

- **Ejemplo.** Calcular t(n) y O(t(n)).

  $t(n) = 1 + n$ $\qquad$ $n = 0, 1$

  $t(n) = 4t(n-2) + (n+5)3^n + n^2$ $\qquad$ Si $n>1$

1.3.3. Cambio de variable.

$$t(n) = a \cdot t(n/4) + b \cdot t(n/8) + ...$$

Cambio de variable:

- Convertir las ecuaciones anteriores en algo de la forma $t'(k) = a \cdot t'(k-c_1) + b \cdot t'(k-c_2) + ...$
- Resolver el sistema en **k**.
- Deshacer el cambio, y obtener el resultado en **n**

Cambios típicos:

- $n = 2^k$; $k = \log_2 n$
- $n = 3^k$, $k = \log_3 k$
- $n = 5k$; $k = n/5$

1.3.3. Cambio de variable.

- **Ejemplo 1.** Resolver:

$t(n) = a$ Si $n=1$

$t(n) = 2\, t(\lfloor n/2 \rfloor) + b \cdot n$ Si $n>1$, con $b>0$

- **Ejemplo 2.** Resolver:

$t(n) = n$ Si $n<b$

$t(n) = 3 \cdot t(n-b) + n^2 + 1$ En otro caso

1.3.3. Cambio de variable.

- Los órdenes que obtenemos son condicionados a que se cumplan las condiciones del cambio:

$$t(n) \in O(f \mid P(n))$$

- ¿Cómo quitar la condición?

- **Teorema.** Sea **b** un entero ≥ 2, **f: N $\rightarrow$ R$^+$** una función no decreciente a partir de un n_0 (f es **eventualmente no decreciente**) y $f(bn) \in O(f(n))$ (f es **b-armónica**) y **t: N $\rightarrow$ R$^+$** eventualmente no decreciente. Entonces, si $t(n) \in \Theta(f(n) \mid n=b^k)$ se cumple que $t(n) \in \Theta(f(n))$.

1.3.4. Otras técnicas.
Transformación de la imagen

- Se utiliza en algunos casos, donde las ecuaciones recurrentes son no lineales. **Ejemplo.**

  $t(1) = 6;$ $\quad t(n) = n\, t^2(n/2)$

- Suponiendo **n** potencia de 2, hacemos el cambio $\mathbf{n=2^k}$:

  $t(2^0) = 6;$ $\quad t(2^k) = 2^k\, t^2(2^{k-1})$

- Tomando logaritmos (en base 2):

  $\log t(2^0) = \log 6;$ $\quad \log t(2^k) = k + 2 \cdot \log t(2^{k-1})$

- Se hace una transformación de la imagen:

  $v(x) = \log t(2^x) \Rightarrow$

  $v(0) = \log 6;$ $\quad v(k) = k + 2 \cdot v(k-1)$

1.3.4. Otras técnicas.
Transformación de la imagen

- Resolver la ecuación recurrente:

  $v(0) = \log 6; \qquad v(k) = k + 2 \cdot v(k-1)$

- Resultado:

  $v(k) = c_1 \cdot 2^k + c_2 + c_3 \cdot k \Rightarrow v(k) = (3+\log 3) \cdot 2^k - k - 2$

- Ahora deshacer el cambio $v(x) = \log t(2^x)$:

  $\log t(2^k) = \log t(n) = (3+\log 3) \cdot 2^k - k - 2$

- Y quitar los logaritmos, elevando a 2:

  $t(n) = 2^{(3+\log 3)n - \log n - 2} = 2^{3n} \cdot 2^{\log 3 \cdot n} \cdot 2^{-\log n} \cdot 2^{-2} =$

  $= (2^{3n-2} \cdot 3^n)/n = 24^n/4n$

- Quitar la condición de que n sea potencia de 2.
- ¿Cuánto vale O(t)?

1.3.4. Otras técnicas.

Expansión de recurrencias

- Aplicar varias veces la fórmula recurrente hasta encontrar alguna “regularidad”.
- **Ejemplo.** Calcular el número de **mover**, para el problema de las torres de Hanoi.

  $t(0) = 1$

  $t(n) = 2\ t(n-1) + 1.$

- Expansión de la ecuación recurrente:

  $t(n) = 2\ t(n-1) + 1 = 2^2\ t(n-2) + 2 + 1 = 2^3\ t(n-3)+4+2+1=$

  $$= \ldots\ldots^{n}\ \ldots\ldots = 2^n\ t(n-n) + \sum_{i=0}^{n-1} 2^i = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

1.3.4. Otras técnicas.

Expansión de recurrencias

- Puede ser adecuada cuando sólo hay un término recurrente o cuando la ecuación es no lineal.

- **Ejemplo.**

  $t(0) = 1$

  $t(n) = n\, t(n-1) + 1$

- No aplicar si aparecen varios términos recurrentes:

  $t(n) = 5\, t(n-1) - 8\, t(n-2) + 4n - 3$

  $t(1) = 3,\ t(2) = 10$

1.3.4. Otras técnicas.
Inducción constructiva

- Se usa cuando las ecuaciones son no lineales y no se puede aplicar ninguna de las técnicas anteriores.

- **Inducción**: Dado t(n), suponer que pertenece a algún orden O(f(n)) y demostrarlo por inducción.
  - **Caso base.** Para algún valor pequeño, $t(n) \leq c_1 \cdot f(n)$
  - **Caso general.** Suponiendo que $t(n-1) \leq c_1 \cdot f(n-1)$, entonces se demuestra que $t(n) \leq c_1 \cdot f(n)$

1.3.4. Otras técnicas.
Inducción constructiva

- **Ejemplo**. Dada la siguiente ecuación recurrente, demostrar que $t(n) \in \Theta(n!)$:

  $t(1) = a$

  $t(n) = b \cdot n^2 + n \cdot t(n - 1)$

  – Demostrar por inducción que $t(n) \in \Omega(n!)$.
  – Demostrar por inducción que $t(n) \in O(n!)$.

1.3.5. Condiciones iniciales.

- ¿Cuál es el significado de las condiciones iniciales?
- **Condición inicial:** caso base de una ecuación recurrente.
- ¿Cuántas aplicar?
  - Tantas como constantes indeterminadas.
  - **n** incógnitas, **n** ecuaciones: sistema determinado. Aplicamos el método de Cramer.
- ¿Cuáles aplicar?
  - Las condiciones aplicadas se deben poder alcanzar desde el caso general.
  - Si se ha aplicado un cambio de variable, deben cumplir las condiciones del cambio.

1.3.5. Condiciones iniciales.

- **Ejemplo.**

  $t(n) = n$ Si $n \leq 10$

  $t(n) = 5 \cdot t(n-1) - 8 \cdot t(n-2) + 4 \cdot t(n-3)$ Si $n > 10$

- Resultado: $t(n) = c_1 + c_2 2^n + c_3 n \cdot 2^n$

- Aplicar las condiciones iniciales para despejar c_1, c_2, c_3.
- ¿Cuántas aplicar? ¿Cuáles?

1.3.5. Condiciones iniciales.

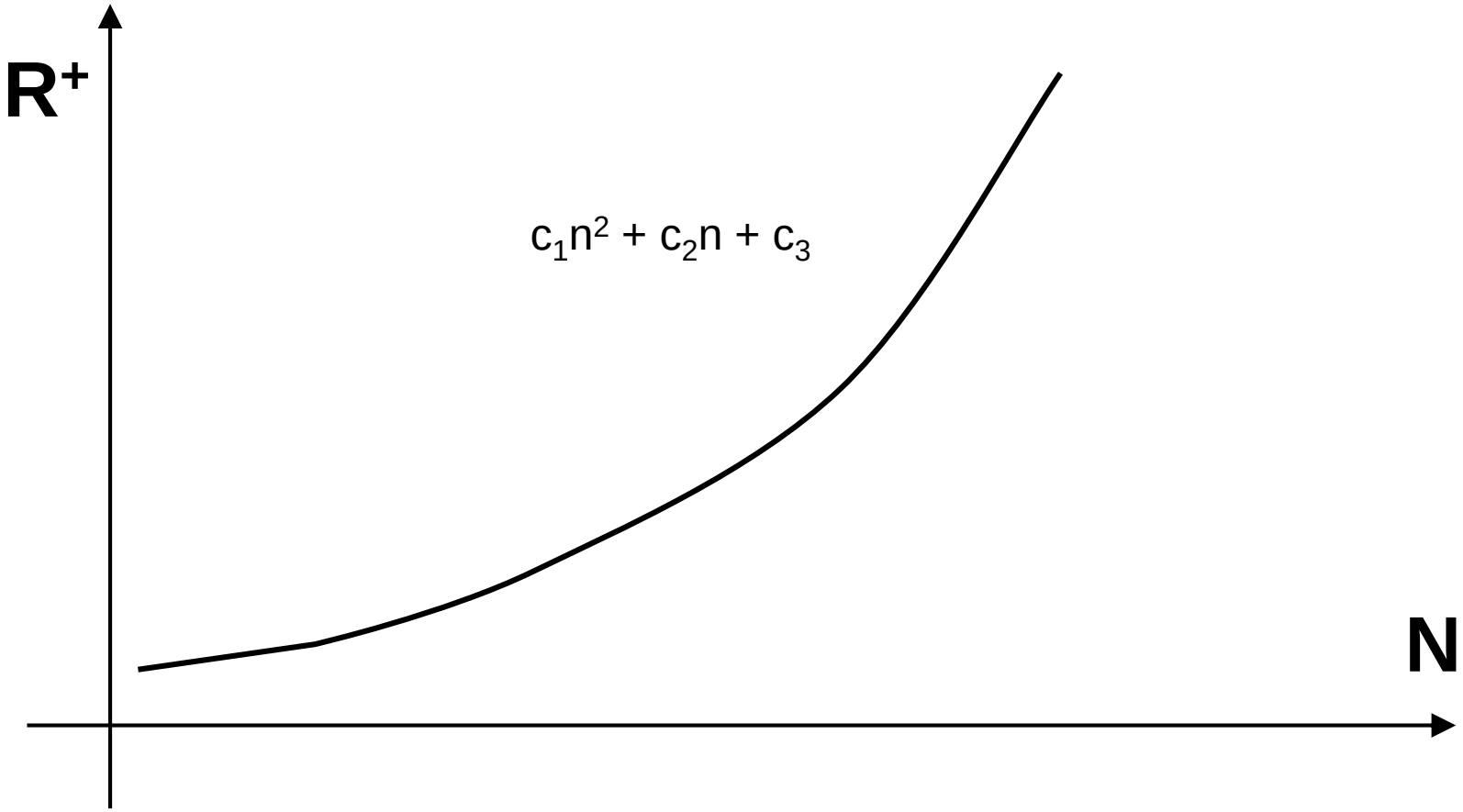
- El cálculo de constantes también se puede aplicar en el estudio experimental de algoritmos.

- **Proceso**

1. Hacer una estimación teórica del tiempo de ejecución.
2. Expresar el tiempo en función de constantes indefinidas.
3. Tomar medidas del tiempo de ejecución para distintos tamaños de entrada.
4. Resolver las constantes.

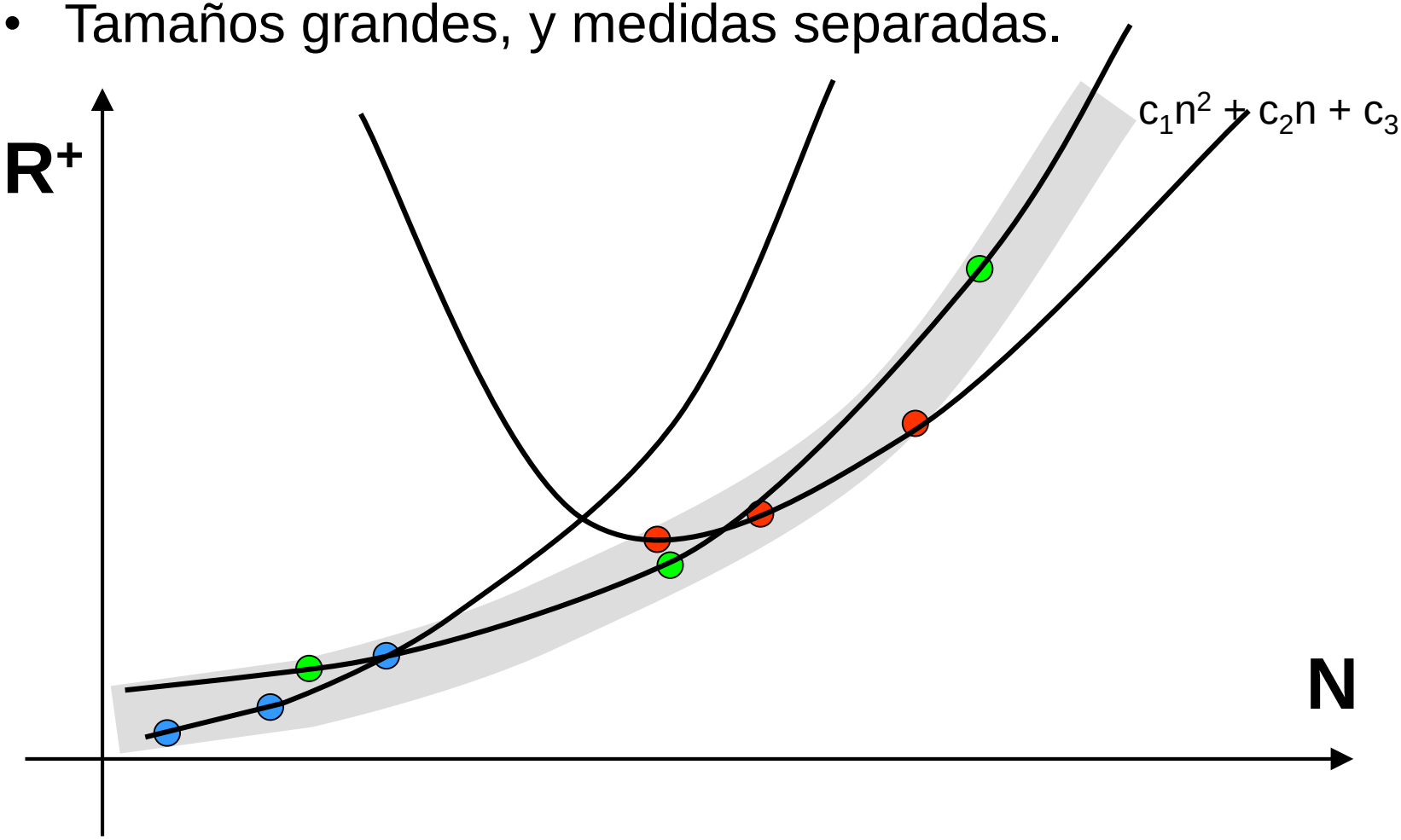
1.3.5. Condiciones iniciales.

- **Ejemplo:** $t(n) = a(n+1)^2 + (b+c)n + d$
- Simplificamos constantes: $t(n) = c_1n^2 + c_2n + c_3$



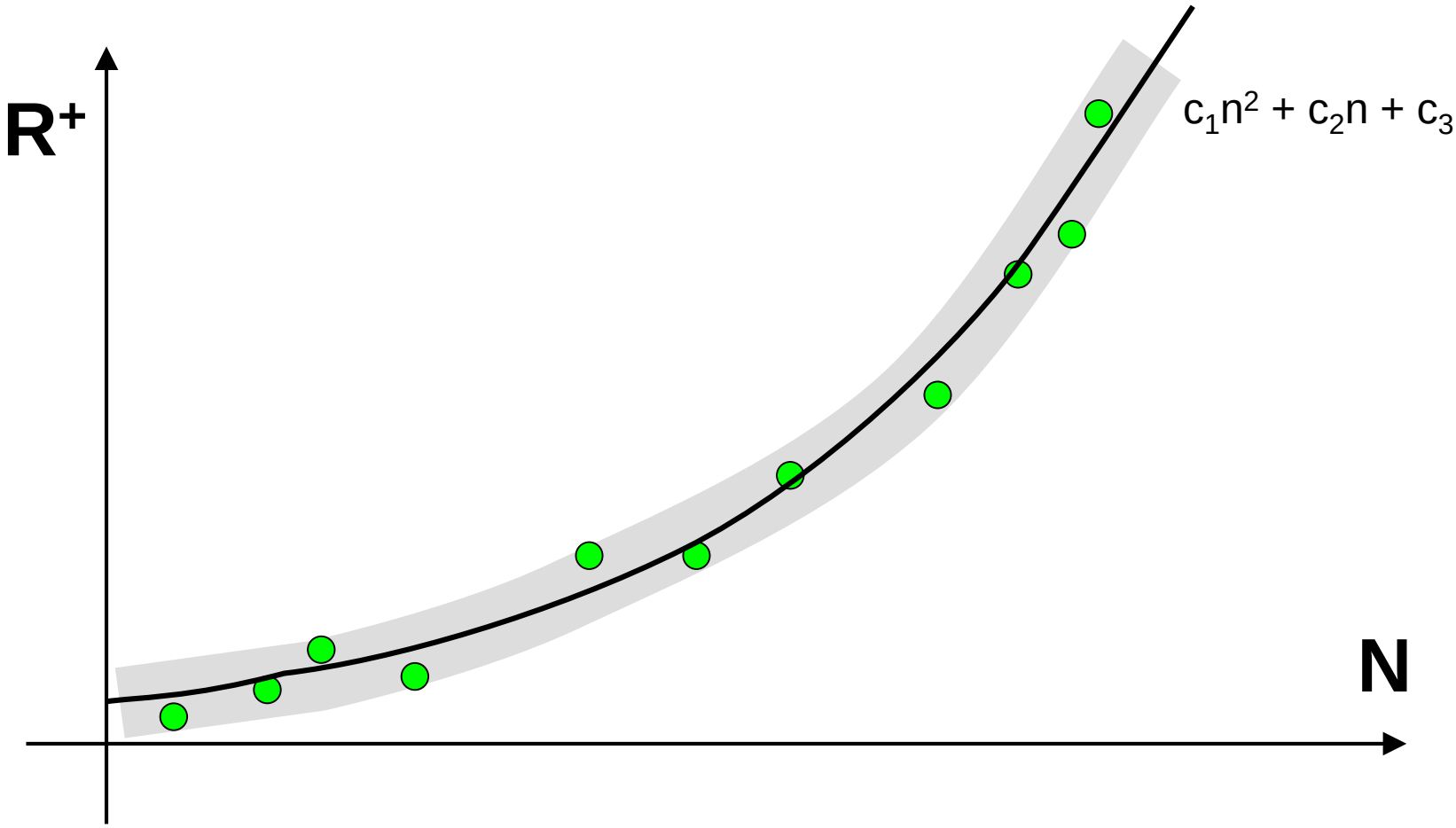
1.3.5. Condiciones iniciales.

- **Ajuste sencillo:** Tomar 3 medidas de tiempo.
  3 incógnitas y 3 ecuaciones: resolvemos c_1, c_2, c_3.
- Tamaños grandes, y medidas separadas.



1.3.5. Condiciones iniciales.

- **Ajuste preciso:** Tomar muchas medidas de tiempo.
- Hacer un ajuste de regresión.



1.4. Ejemplos.

- **Ejemplo 1.** Dada la siguiente ecuación de recurrencia, con a, b, c y $d \in R^+$ y e, $n_0 \in N^+$:

$$f(n) = \begin{cases} d & \text{Si } n \leq n_0 \\ a \cdot f(n-e) + bn + c & \text{Si } n > n_0 \end{cases}$$

Demostrar que:

$a < 1 \Rightarrow f \in O(n)$

$a = 1 \Rightarrow f \in O(n^2)$

$a > 1 \Rightarrow f \in O(a^{n/e})$

1.4. Ejemplos.

- **Ejemplo 2.** Dada la siguiente ecuación de recurrencia, con a, b, c y $p \in R^+$ y d, $n_0 \in N^+$:

$$f(n) = \begin{cases} c & \text{Si } n \leq n_0 \\ a \cdot f(n/d) + bn^p & \text{Si } n > n_0 \end{cases}$$

Demostrar que:

$a < d^p \Rightarrow f \in O(n^p)$

$a = d^p \Rightarrow f \in O(n^p \cdot \log n)$

$a > d^p \Rightarrow f \in O(n^{\log_d a})$

1.4. Ejemplos.

- **Ejemplo 3.** Calcular el número de instrucciones de asignación del siguiente algoritmo.

```
procedure Otro (n: integer): integer;
        for i:= 1 to n do
                M[i]:= M[i] + 1
        if i>0 then
                Otro(n-4)
```

1.4. Ejemplos.

- **Ejemplo 4.** El tiempo de ejecución de determinado programa se puede expresar con la siguiente ecuación de recurrencia:

$$t(n) = \begin{cases} 2n & \text{Si } n \leq 10 \\ 2t(\lfloor n/2 \rfloor) + 3t(\lfloor n/4 \rfloor) + 2n + 1 & \text{En otro caso} \end{cases}$$

- Calcula el tiempo de ejecución para los valores de **n** que sean potencia de 2. Exprésalo con las notaciones O, Ω ó Θ.
- Muestra las condiciones iniciales que se deberían aplicar.
- Eliminar la condición de que **n** sea potencia de 2.
- La afirmación **t(n)** $\in \Omega(\log n)$ ¿es correcta en este caso?, ¿es una buena cota para el orden de complejidad del programa?

1. Análisis de algoritmos.

Conclusiones:

- **Eficiencia:** consumo de recursos en función de los resultados obtenidos.
- **Recursos consumidos** por un algoritmo: fundamentalmente tiempo de ejecución y memoria.
- La estimación del tiempo, **t(n)**, es aproximada, parametrizada según el tamaño y el caso (t_m, t_M, t_p).
- **Conteo de instrucciones:** obtenemos como resultado la función **t(n)**
- Para simplificar se usan las notaciones asintóticas: $O(t)$, $\Omega(t)$, $\Theta(t)$, $o(t)$.

1. Análisis de algoritmos.

Conclusiones:

- **Ecuaciones recurrentes:** surgen normalmente del conteo (de tiempo o memoria) de algoritmos recursivos.
- Tipos de ecuaciones recurrentes:
  - Lineales, homogéneas o no homogéneas.
  - No lineales (menos comunes en el análisis de algoritmos).
- **Resolución de ecuaciones recurrentes**:
  - Método de expansión de recurrencias (el más sencillo).
  - Método de la ecuación característica (lineales).
  - Cambio de variable (previo a la ec. característica) o transformación de la imagen.
  - Inducción constructiva (general pero difícil de aplicar).