Análisis de la eficiencia de Algoritmos. Complejidad temporal y espacial.

Tema 1

Algorítmica y Modelos de Computación

1. Introducción.

2. Eficiencia y complejidad.

   2.1. Coste temporal y espacial.
   2.2. Análisis por conteo de operaciones elementales.
   2.3. Elección de una operación básica (instrucción crítica).
   2.4. Caso mejor, peor y medio.

3. Cotas de complejidad. Medidas asintóticas.

4. Análisis de algoritmos iterativos.

5. Análisis de algoritmos recursivos. Resolución de recurrencias.

1. Introducción.

- En Ciencias de la Computación un problema tiene solución si existe un **algoritmo** susceptible de implantarse en un ordenador, capaz de producir la respuesta correcta para cualquier instancia del problema en cuestión.
- Para ciertos problemas es posible encontrar más de un algoritmo capaz de resolverlos, surge el problema de **escoger** alguno de ellos.
- La elección de un buen algoritmo, **decidir** cual de ellos es el mejor, está orientada hacia la disminución del costo que implica la solución del problema. Clases de criterios:
  - Criterios orientados a **minimizar el costo de desarrollo**: claridad, sencillez y facilidad de implantación, depuración y mantenimiento.
  - Criterios orientados a **disminuir el costo de ejecución**: **tiempo** de procesador y cantidad de **memoria** utilizados.
- Los recursos que consume un algoritmo pueden estimarse mediante herramientas teóricas y constituyen, por lo tanto, una base confiable para la elección de un algoritmo.
- La **eficiencia** de un algoritmo es la medida del uso de los recursos en función del tamaño de las entradas
- En Ciencias de la Computación, la actividad dedicada a determinar la **cantidad de recursos que consumen** los algoritmos se le denomina **análisis de algoritmos**.

2. Eficiencia y Complejidad.

- La **función complejidad**, **f(n);** donde ***n*** es el tamaño del problema, da una medida de la cantidad de recursos que un algoritmo necesitará al implantarse y ejecutarse en algún ordenador. Puesto que la cantidad de recursos que consume un algoritmo crece conforme el tamaño del problema se incrementa, la función complejidad es monótona creciente ($f(n) > f(m)$, $n > m$) con respecto al tamaño del problema.
- La memoria y el tiempo de procesador son los recursos sobre los cuales se concentra todo el interés en el análisis de un algoritmo, así pues distinguiremos dos clases de función complejidad:
  - **Función complejidad espacial ($f_e(n)$ o $E(n)$)**. Mide la cantidad de memoria que necesitará un algoritmo para resolver un problema de tamaño **n**.
  - **Función complejidad temporal ($f_t(n)$ o $T(n)$)**. Indica la cantidad de tiempo que requiere un algoritmo para resolver un problema de tamaño **n**; viene a ser una medida de la cantidad de CPU que requiere el algoritmo.

2. Eficiencia y Complejidad. 2.1. Coste espacial.

- La cantidad de memoria que utiliza un algoritmo depende de la implantación, no obstante, es posible obtener una medida del espacio necesario con la sola inspección del algoritmo sumando todas las celdas de memoria que utiliza. En general hay dos tipos :
  - Celdas estáticas. Son las que se utilizan en todo el tiempo que dura la ejecución del programa, por ejemplo, las variables globales.
  - Celdas dinámicas. Se emplean sólo durante un momento de la ejecución, y por tanto pueden ser asignadas y devueltas conforme se ejecuta el algoritmo, por ejemplo, el espacio de la pila utilizado por las llamadas recursivas.
- Una celda de memoria no tiene una medida física en correspondencia, especificará sólo el espacio empleado por una variable simple, sin importar el tipo al que pertenezca.
  - Las variables simples ocuparán una celda de memoria.
  - Variables de tipos compuestos, se les asignan tantas celdas como requieran los elementos que las componen.

2. Eficiencia y Complejidad. 2.1. Coste temporal.

- El **tiempo** que emplea un algoritmo en ejecutarse refleja la cantidad de trabajo realizado.
- La complejidad temporal da una medida de la cantidad de tiempo que requerirá la implantación de un algoritmo para resolver el problema, por lo que se le puede determinar en forma experimental:
  - Por ejemplo, para encontrar el valor de la función complejidad de un algoritmo *A* que se codifica un lenguaje de programación *L;* se compila utilizando el compilador *C* y se ejecuta en la máquina *M;* damos algunas entradas al programa y medimos el tiempo de procesador empleado para resolver los casos del problema.
  - **Inconvenientes**: Los resultados obtenidos dependen de:
    - las entradas proporcionadas,
    - la calidad del código generado por el compilador utilizado, y
    - de la máquina en la que se ejecutan.
- **Solución**: el análisis temporal (y el espacial) se hará únicamente con base al algoritmo escrito en **pseudocódigo**, como no se puede ejecutar, la complejidad temporal no se expresará en unidades de tiempo, sino en términos de la **cantidad de operaciones que realiza**.

- El **tiempo de ejecución de un algoritmo** va a ser una **función** que mide el **número de operaciones** elementales (las que el ordenador realiza en cierta cantidad constante de tiempo) que realiza el algoritmo para un tamaño de entrada dado.
- Consideraremos **operaciones elementales** (contabilizándolas como 1 operación elemental) las siguientes:
  - Operaciones aritméticas básicas
  - Asignaciones a variables de tipo predefinido.
  - Saltos:
    - Llamadas a funciones o procedimientos
    - Retorno desde funciones o procedimientos
  - Comparaciones lógicas
  - Accesos a estructuras indexadas básicas (vectores, matrices…)
- Dado un algoritmo, se puede determinar que tipos de operaciones utiliza y cuantas veces las ejecuta para una entrada específica. Observemos este procedimiento en el siguiente ejemplo.

- **Ejemplo 1.** El Algoritmo 1 realiza **búsqueda lineal** sobre un arreglo *A* con **n** elementos, y devuelve la posición en la que se encuentra el elemento **Valor**; si Valor no se encuentra devuelve **n+1***:* El algoritmo realiza sumas (+), asignaciones (←), comparaciones (≤ y ≠) y operaciones lógicas (AND).

```
funcion BúsquedaLineal (Valor; A; n);
 comienza
      i ← 1;                                          /*1*/
      mientras (i ≤ n) AND (A[i] ≠ Valor) hacer       /*2*/
            i ← i + 1;                                /*3*/
      fmientras;
      BúsquedaLineal ← i;       /* return i*/         /*4*/
 termina.
```

Algoritmo 1: Búsqueda lineal

- El número de operaciones elementales (OE) que se realizan en cada línea son:
  1. Se ejecuta 1 OE (1 asignación)
  2. La condición del bucle, con un total de 4 OE (2 comparaciones y acceso al vector y 1 operación lógica.)
  3. 2 OE(un incremento y 1 asignación)
  4. Se ejecuta 1 OE (1 asignación)

- **Ejemplo 1.**

Algoritmo 1: Búsqueda lineal

```
funcion BúsquedaLineal (Valor; A; n);
 comienza
      i ← 1;
      mientras (i ≤ n) AND (A[i] ≠ Valor)
      hacer
            i ← i + 1;
      fmientras;
      BúsquedaLineal ← i;
termina.
```

- Si el ciclo mientras se realiza ***k*** veces, el algoritmo ejecutará las operaciones siguientes:
  - **k** sumas (una por cada iteración).
  - **k + 2** asignaciones (las del ciclo y las realizadas fuera del ciclo).
  - ***k* + 1** operaciones lógicas (la condición se debe probar *k* + 1 veces, la última es para saber que el ciclo no debe volver a ejecutarse).
  - **k + 1** comparaciones con el índice.
  - **k + 1** comparaciones con elementos de *A.*
  - **k + 1** accesos a elementos de *A.*
- **6k + 6** operaciones en total.

Reglas generales para el cálculo de operaciones elementales que definen el número de operaciones elementales de cada estructura básica del lenguaje. El número total de un algoritmo se calcula por inducción sobre ellas. Considerando que el tiempo de una **OE** es de **orden 1**, las reglas para calcular el tiempo de ejecución son:

- Para una secuencia consecutiva de instrucciones se calcula sumando los tiempos de ejecución de cada una de las instrucciones.
- Para la sentencia "**caso** E **sea** v1:S1 | v2:S2 |...| vn:Sn | otro So **fcaso** " es

  T = T(E) + max{T(S1),T(S2),...,T(Sn)}

  (T(E) incluye el tiempo de comparación con v1, v2 ,..., vn.)
- Para la sentencia "**Si** C **entonces** S1 **otro** S2 **fsi**" es

  T = T(C) + max{T(S1),T(S2)}
- Para el bucle "**mientras** C **hacer** S **fmientras**" es

  T = T(C) + (nº iteraciones)*(T(S) + T(C))

  (T(C) y T(S) pueden variar en cada iteración y habrá que tenerlo en cuenta para su cálculo)
- Para calcular el tiempo de ejecución del resto de sentencias iterativas (PARA REPETIR) basta expresarlas como un bucle MIENTRAS.
- Para una llamada a un procedimiento o función **F(P1, P2,..., Pn)** es 1 (por la llamada), más el tiempo de evaluación de los parámetros *P*1, *P*2,..., *Pn*, más el tiempo que tarde en ejecutarse *F*. **T = 1 + T(P1) + T(P2) + ... + T(Pn) + T(F)**
- El tiempo de ejecución de las llamadas a procedimientos recursivos va a dar lugar a ecuaciones en recurrencia.

- Del ejemplo 1 observamos que el número de veces que se ejecutan algunas operaciones, para valores sucesivamente mayores que *k* presenta un **modelo de crecimiento** similar al que tiene el número total de operaciones que ejecuta el algoritmo (ambas cantidades son del mismo orden).

- Para hacer una estimación de la cantidad de tiempo que tarda un algoritmo en ejecutarse, no es necesario contar el número total de operaciones que realiza. Se puede elegir alguna, a la que se identificará como **operación básica** que observe un comportamiento parecido al del número total de operaciones realizadas y que, por lo tanto, será proporcional al tiempo total de ejecución. Para el algoritmo anterior se puede contar cualquiera de las operaciones mencionadas.

- En general, debe procurarse que la operación básica, en la cual se basa el análisis, de alguna forma esté relacionada con el tipo de problema que se intenta resolver, ignorando
  - las asignaciones de valores iniciales y
  - las operaciones sobre variables para control de ciclos (índices).

- La operación básica en el Algoritmo 1 (búsqueda lineal) es la comparación entre los elementos del arreglo y el valor buscado, pues cumple con las consideraciones anteriores.

2. Eficiencia y Complejidad. 2.4. Caso mejor, peor y medio.

- **Ejemplo 2.** Supongamos el Algoritmo 1, para hacer el análisis de su comportamiento tomamos como operación básica las comparaciones con elementos del arreglo y como caso muestra: A = [2, 7, 4, 1, 3] y n = 5:
  - Si Valor = 2, se hace una comparación, $T(5) = 1$
  - Si Valor = 4, se hacen tres comparaciones, $T(5) = 3$
  - Si Valor = 8, se hacen seis comparaciones, y $T(5) = 6$
- Por tanto, la función complejidad no es tal, en realidad es una relación, ya que para un mismo tamaño de problema se obtienen distintos valores de la función complejidad.
- Hay muchos algoritmos en los que el número de operaciones depende, no sólo del tamaño del problema, sino también de la instancia específica que se presente, entonces, en el valor de la función complejidad se distinguen tres casos: mejor, peor y medio.

```
funcion BúsquedaLineal (Valor; A; n);
        i ← 1;                                          /*1*/
        mientras (i ≤ n) AND (A[i] ≠ Valor) hacer       /*2*/
                i ← i + 1;                              /*3*/
        fmientras;
        BúsquedaLineal ← i;      /* return i*/         /*4*/
ffuncion.
```

2. Eficiencia y Complejidad. 2.5. Caso mejor, peor y medio.

- **Definiciones:**
  - $I(n) = \{ I_1, I_2, I_3, \ldots\ldots, I_K \}$ conjunto de instancias del problema de tamaño **n.**
  - $O(n) = \{ O_1, O_2, O_3, \ldots\ldots, O_K \}$ conjunto formado por el número de operaciones que un algoritmo realiza para resolver cada instancia. Entonces,
  - $\mathbf{O_j}$ es el nº operaciones ejecutadas para resolver la instancia $\mathbf{I_j}$ para $1 \leq j \leq k$.
- Se distinguen tres casos en el valor de la función complejidad temporal:
  - **Caso Peor:** $\mathbf{T(n) = máx (\{ O_1 , O_2 , O_3 , \ldots\ldots, O_K \})}$
  - **Caso Mejor:** $\mathbf{T(n) = mín (\{ O_1 , O_2 , O_3 , \ldots\ldots, O_K \})}$
  - **Caso medio**

$$T(n) = \sum_{i=1}^{k} O_i P(i)$$

donde *P(i)* es la probabilidad de que ocurra la instancia I_i.

- el **mejor caso** se presenta cuando para una entrada de tamaño ***n;*** el algoritmo ejecuta el mínimo número posible de operaciones.
- el **peor caso** cuando hace el máximo número posible de operaciones.
- en el **caso medio** se consideran todos los casos posibles para calcular el **promedio** de las operaciones que se hacen tomando en cuenta la **probabilidad** de que ocurra cada **instancia**.

2. Eficiencia y Complejidad. 2.5. Caso mejor, peor y medio.

```
funcion BúsquedaLineal (Valor; A; n);
        i ← 1;                                              /*1*/
        mientras (i ≤ n) AND (A[i] ≠ Valor) hacer           /*2*/
                i ← i + 1;                                  /*3*/
        fmientras;
        BúsquedaLineal ← i;          /* return i*/          /*4*/
ffuncion.
```

- **Ejemplo 3.** Algoritmo de búsqueda lineal. **Análisis Temporal por conteo de OE**.

- **Mejor caso**: ocurre cuando el valor es el primer elemento del vector. Se efectuará la linea 1 (1OE) , la linea 2(4OE) y la linea 4 (1OE)

  T(n) = 1+4+1=6

- **Peor caso**: sucede cuando el valor no se encuentra en el vector. El bucle se repite n veces hasta que no se cumple la primera condición y después se ejecuta la línea 4. Cada iteración del bucle está compuesta por las líneas 2 (4OE) y 3(2OE), junto con una ejecución adicional de la línea 2 que es la que ocasiona la salida del bucle.

$$T(n) = 1 + \left(\left(\sum_{i=1}^{n} (4+2) \right) + 4 \right) + 1 = 1 + ((n(6)) + 4) + 1 = 6n + 6$$

- **(cont) Ejemplo 3.** Algoritmo de búsqueda lineal. Análisis por conteo de OE.

- **Caso medio**: el bucle se ejecutará un nº de veces entre 1 y n, y suponemos que cada una de ellas tiene la misma probabilidad de suceder. Como existen n+1 posibilidades (puede que no esté) suponemos a priori que son equiprobables y por tanto cada una tendrá una probabilidad asociada de

$$P(i) = \frac{1}{n+1}$$

- Así, el nº medio de veces que se efectuará el bucle será:

$$nº\ veces\_bucle = \frac{1}{n+1}\sum_{i=1}^{n} i = \frac{n(n+1)}{2(n+1)} = \frac{n}{2}$$

$$\sum_{i=1}^{i=n} a_i = \frac{n(a_1 + a_n)}{2}$$

- Y el tiempo medio de ejecución:

$$T(n) = 1 + \left(\left(\sum_{i=1}^{\frac{n}{2}} (4+2)\right) + 4\right) + 1 = 1 + \left(\left(\frac{n}{2}(6)\right) + 4\right) + 1 = 3n + 6$$

2. Eficiencia y Complejidad. 2.5. Caso mejor, peor y medio.

- **Ejemplo 4.** Algoritmo de búsqueda lineal. Análisis temporal con Operación Básica: Comparación del valor con los elementos del vector.

- **Mejor caso**: el valor es el primer elemento del vector.

  T(n) = 1

- **Peor caso**: el valor no se encuentra en el vector.

  T(n) = n + 1

- **Caso medio**:

  T(n) = 1P(1) + 2P(2) + 3P(3) + 4P(4) + ::: + nP(n) + (n+1)P(n + 1)

  - donde P(i) es la probabilidad de que el valor se encuentre en la posición i; (1≤i ≤n) y P(n + 1) es la probabilidad de que no esté en el arreglo. Si se supone que todos los casos son igualmente probables, $P(i) = \frac{1}{n+1}$

$$T(n) = \frac{1}{n+1}\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2(n+1)} = \frac{n+2}{2}$$

$$\sum_{i=1}^{i=n} a_i = \frac{n(a_1 + a_n)}{2}$$

2. Eficiencia y Complejidad. 2.5. Caso mejor, peor y medio.

- **NO** todos los algoritmos presentan casos.
- Un algoritmo tendrá casos **SI** se puede resolver el problema de manera trivial para alguna instancia específica y se tendrá que realizar un **análisis en casos.**

3. Cotas de complejidad. Medidas asintóticas.

- Una vez vista la forma de calcular el tiempo de ejecución T(n) de un algoritmo, el propósito es clasificar dichas funciones de forma que se puedan comparar. Para ello, se definen clases de equivalencia, correspondientes a las funciones que "crecen de la misma forma".

- 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.

- Estudiaremos el comportamiento de un algoritmo al aumentar el tamaño de los datos; es decir, como aumenta su tiempo de ejecución. Esto se conoce como **eficiencia asintótica**, representada mediante **funciones de orden**. Esta notación facilita la comparación de eficiencia entre algoritmos diferentes

- **Notaciones asintóticas:** indican como crece **t**, para valores suficientemente grandes (asintóticamente) sin considerar constantes.

  - O(t): Orden de complejidad de t. Cota Superior.
  - Ω(t): Orden inferior de t, u omega de t. Cota Inferior.
  - Θ(t): Orden exacto de t. Cota Exacta.

- **Cota Superior. Notación O.**

  Dada una función f, se estudia aquellas funciones g que a lo sumo crecen tan deprisa como f. Al conjunto de tales funciones se le llama cota superior de f y lo denominamos O(f). Conociendo la cota superior de un algoritmo se puede asegurar que, en ningún caso, el tiempo empleado será de un orden superior al de la cota.

- **Definición 1. Orden de complejidad de f(n): O(f).**

  Dada una función **f: N → R⁺**, llamamos **orden de f** (orden O(Omicron) 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.

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

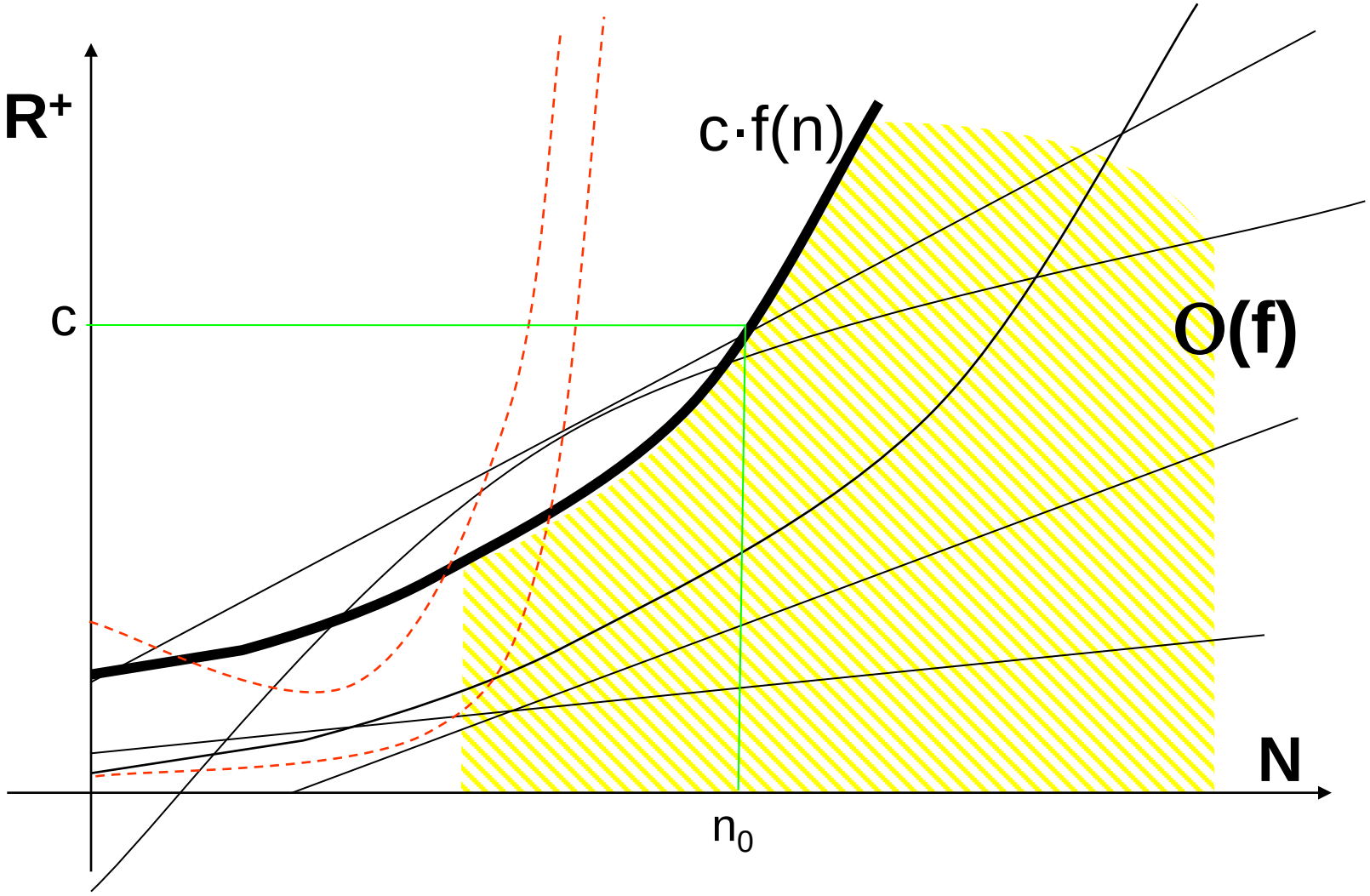
- Diremos que una función t: N → [0,∞) es de orden O de f si t(n) ∈ O(f(n)).
- Cuando t(*n*) ∈ O(*f*(*n*)) se dice que t(*n*) *está en el* **orden** *de f*(*n*). O(*f*(*n*)) representa una **cota superior** de la complejidad de una función t(*n*),
- Intuitivamente, t ∈ O(f) indica que t está acotada superiormente por algún múltiplo de f. Normalmente estaremos interesados en la **menor** función f tal que t pertenezca a O(f).

3. Cotas de complejidad. Orden de complejidad.

- **Ejemplo de notación O(f (n))**
  - T (n) = 9 + 7n
  - T (n) ∈ O(n) (c = 8 y n_0 = 9) (9+7*9≤ (8*9))
  - T (n) ∈ O(n^2) (c =1 y n_0 =8, o bien c =2 y n_0 =7)(9+7*8 ≤8*8)
  - T (n) ∉ O(lg n)

- **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.

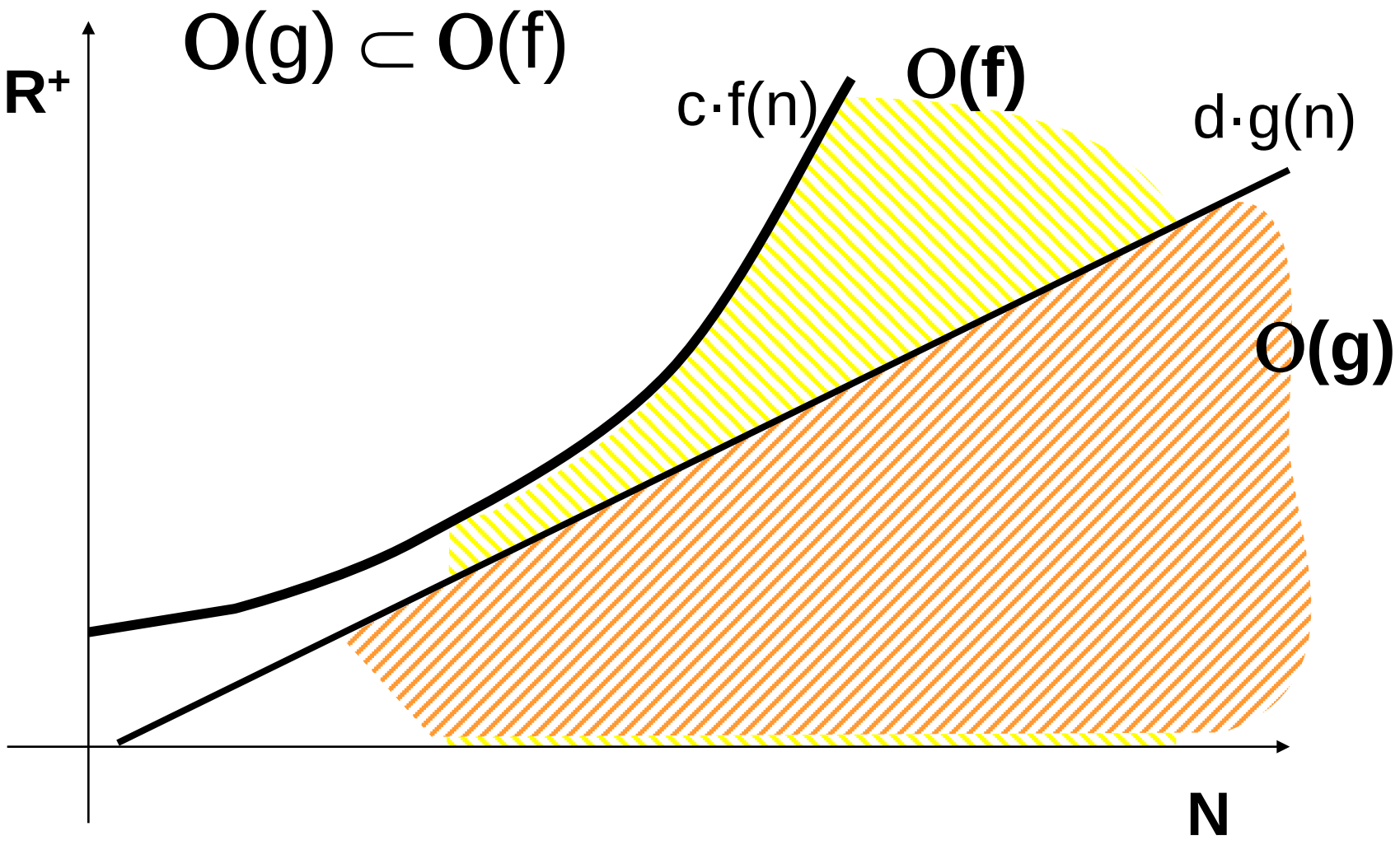
3. Cotas de complejidad. Orden de complejidad.



3. Cotas de complejidad. Relación de orden.

- Relación de orden entre O(..) = Relación de **inclusión** entre conjuntos.
  - $\mathbf{O}(f) \leq \mathbf{O}(g) \Leftrightarrow \mathbf{O}(f) \subseteq \mathbf{O}(g) \Leftrightarrow$ Para toda $t \in \mathbf{O}(f)$, $t \in \mathbf{O}(g)$
- Se cumple que:
  - $\mathbf{O}(c) = \mathbf{O}(d)$, siendo **c** y **d** constantes positivas.
  - $\mathbf{O}(c) \subset \mathbf{O}(n)$
  - $\mathbf{O}(cn + b) = \mathbf{O}(dn + e)$
  - $\mathbf{O}(p) = \mathbf{O}(q)$, si **p** y **q** son polinomios del mismo grado.
  - $\mathbf{O}(p) \subset \mathbf{O}(q)$, si **p** es un polinomio de menor grado que **q**.

3. Cotas de complejidad. Relación de orden.



3. Cotas de complejidad. Orden inferior u omega.

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

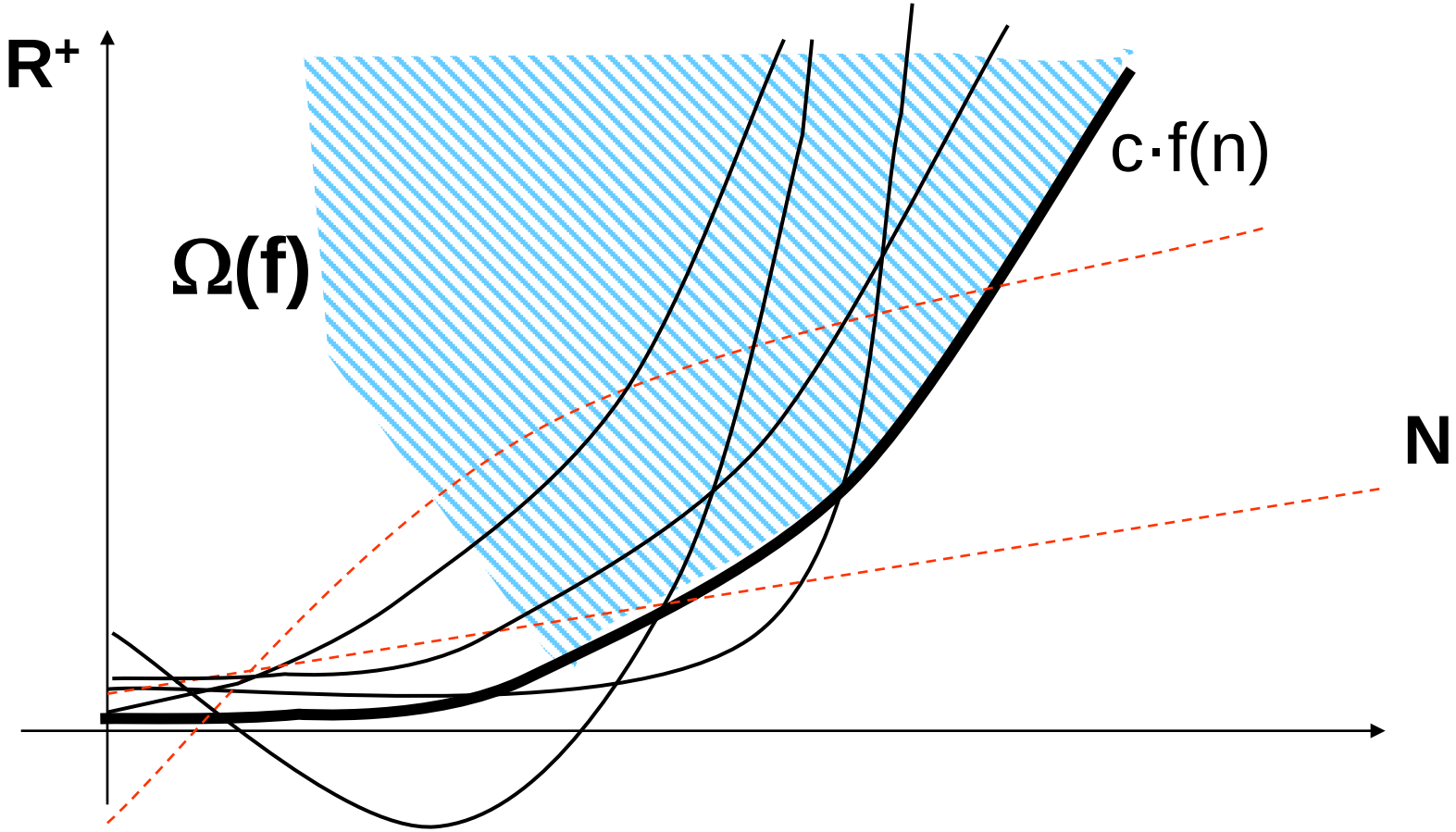
  Dada una función **f: N** $\rightarrow$ **R⁺**, llamamos **omega de f** al conjunto de todas las funciones de **N** en **R⁺** acotadas **inferiormente** por un múltiplo real positivo de **f**, para valores de **n** suficientemente grandes.

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

- **Regla de dualidad**: $f(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in O(f(n))$
- Si se hace un **análisis de caso peor** de un algoritmo con complejidad $T(n)$:
  - si se determina que $T(n) \in O(g(n))$, esto quiere decir que $g(n)$ es una cota superior para todos los casos de ejecución del algoritmo
  - si se determina que $T(n) \in \Omega(h(n))$, esto quiere decir que $h(n)$ es una cota inferior para el caso peor solamente; otros casos podrían estar por debajo de esta cota inferior
- **Ejemplo** de $\Omega(f)$. Número de OE en el caso peor: $T(n) = 5n^2 + 8n - 11$
  - $T(n) \in O(n^2)$ ($c = 6$, $n_0 = 7$) (49*5+8*7-11<6*49)
  - $T(n) \in \Omega(n^2)$ ($c = 1$, $n_0 = 1$) (5+8-11>1)

3. Cotas de complejidad. Orden inferior u omega.

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



3. Cotas de complejidad. Orden exacto.

- **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.

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

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

- Si una función t(n) pertenece a O(f (n)) y a Ω(f (n)), entonces pertenece a Θ(f (n)).
- Θ(f (n)) representa la complejidad exacta de una función. En este caso se dice que t(n) esta en el orden exacto de f (n).
- Θ(f (n)) indica que la función de complejidad esta acotada tanto inferiormente como superiormente por una misma función (normalmente con constantes multiplicativas distintas)
- Ejemplo de Θ(f). $T(n) = 5n^2 + 8n - 11$
  - $T(n) \in O(n^2)$ ($c = 6$, $n_0 = 7$) (49*5+8*7-11<6*49)
  - $T(n) \in \Omega(n^2)$ ($c = 1$, $n_0 = 1$) (5+8-11>1)
  - $T(n) \in \Theta(n^2)$

3. Cotas de complejidad. Orden exacto.

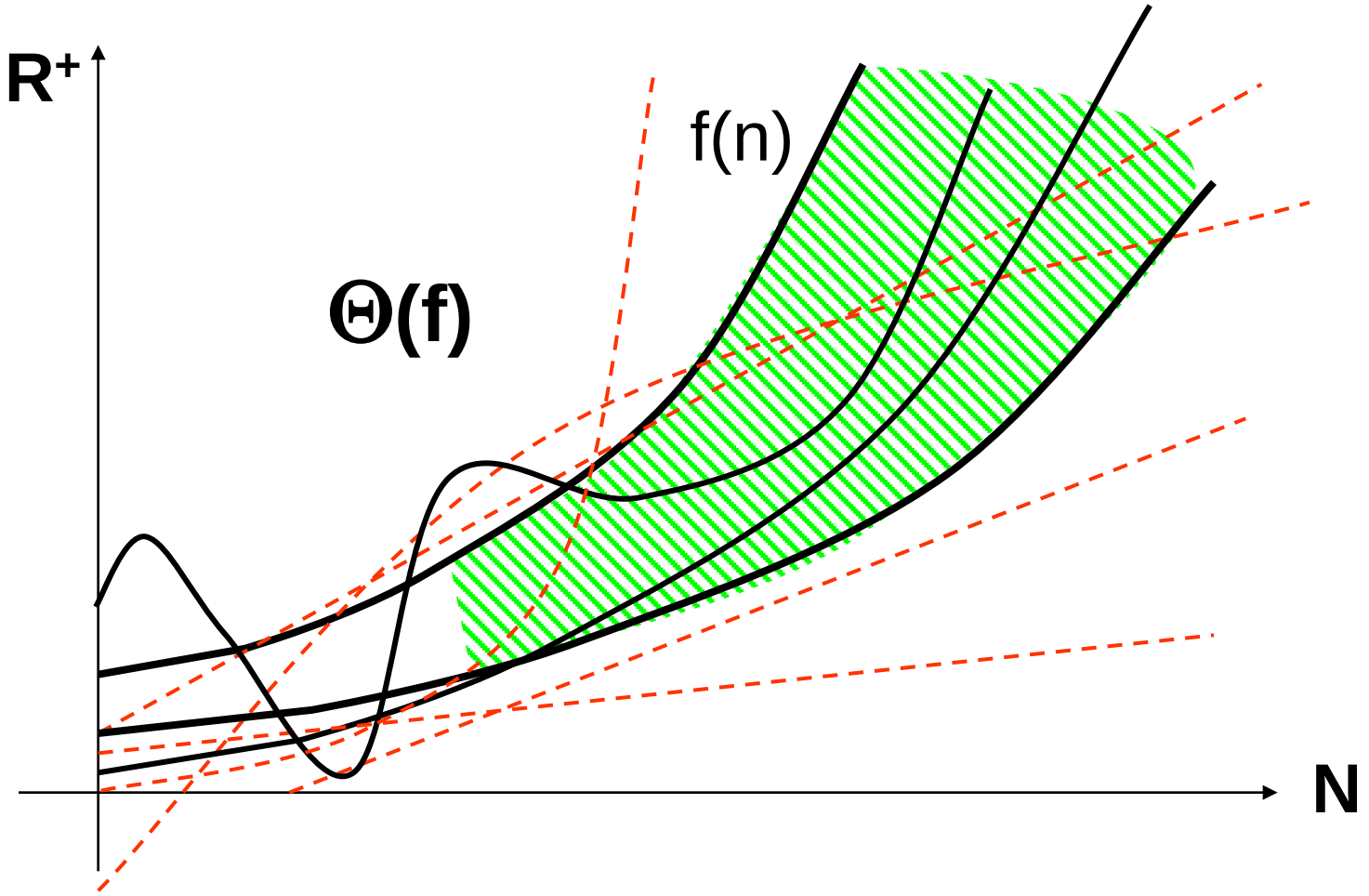
- Se cumple la siguiente propiedad:
  - Si $c \geq 0$, $d>0$, $g(n) \in \Theta(f(n))$ y $h(n) \in \Theta(f(n)) \Rightarrow$ $c.g(n)+d.h(n) \in \Theta(f(n))$
- Θ permite clasificar las funciones de complejidad en conjuntos disjuntos, categorías de complejidad.
- Las categorías de complejidad mas comunes se nombran mediante el componente mas sencillo:

$\Theta(\log n), \Theta(n), \Theta(n \cdot \log n), \Theta(n^2), \Theta(n^k), \Theta(a^n), \Theta(n!), \Theta(n^n)$

Donde $k>2$, $a>1$

3. Cotas de complejidad. Orden exacto.

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



3. Cotas de complejidad. Propiedades de las notaciones asintóticas

1. **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 $f \in \Omega(h)$
   - Si $f \in \Theta(g)$ y $g \in \Theta(h)$ entonces $f \in \Theta(h)$
     - Ej. $2n+1 \in O(n)$, $n \in O(n^2) \Rightarrow 2n+1 \in O(n^2)$
2. **P2.** Si $f \in O(g)$ entonces $O(f) \subseteq O(g)$.
   - Si $f \in \Omega(g)$ entonces $\Omega(f) \subseteq \Omega(g)$.
3. **P3. Relación pertenencia/contenido.** Dadas f y g de N en R^+, se cumple:
   $O(f) = O(g) \Leftrightarrow f \in O(g)$ y $g \in O(f)$
   - $\Omega(f) = \Omega(g) \Leftrightarrow f \in \Omega(g)$ y $g \in \Omega(f)$
4. **P4. Propiedad del mínimo.** Si $f \in O(g)$ y $f \in O(h)$ entonces $f \in O($**mínimo**(g,h).
5. **P5. Propiedad del producto.**
   Si $f_1 \in O(g)$ y $f_2 \in O(h)$ entonces $f_1 \cdot f_2 \in O(g \cdot h)$.
   - Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h)$ entonces $f_1 \cdot f_2 \in \Omega(g \cdot h)$.
6. **P6. Propiedad del máximo.** Dadas f y g, de N en R^+,
   - $O(f_1+f_2) \in O(\max(f_1, f_2))$
     Esta regla es extensible a $f_1+\ldots.+f_k$siempre que k no dependa de n
   - $\Omega(f+g) \in \Omega(\max(f, g))$
   - $\Theta(f+g) \in \Omega(\max(f, g))$
     - Ejemplo: $O(2^n + n^6 + n!) = ..$

7. P7. Relación límites/órdenes.

Dadas f y g de N en R^+, se cumple:

- Para O :
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in R^+ \Rightarrow O(f)= O(g)$ ($f(n) \in O(g(n))$ y $g(n) \in O(f(n))$)
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow O(f) \subset O(g)$, ($f(n) \in O(g(n))$, $g(n) \notin O(f(n))$)
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow O(f) \supset O(g)$, ($g(n) \in O(f(n))$, $f(n) \notin O(g(n))$,

7. P7. Relación límites/órdenes. (cont.)

- Para Ω :
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in R^+ \Rightarrow (f(n) \in \Omega(g(n)) \text{ y } g(n) \in \Omega(f(n))$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow (g(n) \in \Omega(f(n)), f(n) \notin \Omega(g(n)))$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow (f(n) \in \Omega(g(n)), g(n) \notin \Omega(f(n)),$
- Para Θ :
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in R^+ \Rightarrow (f(n) \in \Theta(g(n))$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow (f(n) \in O(g(n)), \text{ pero } f(n) \notin \Theta(g(n)))$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow (f(n) \in \Omega(g(n)) \text{ pero } f(n) \notin \Theta(g(n)),$

3. Cotas de complejidad. Jerarquías de Complejidades

- Algunas relaciones entre órdenes frecuentes:

$O(1) \subset O(\log n) \subset O(n) \subset O(n \cdot \log n) \subset O(n^2) \subset O(n^3) \subset O(n^j) \subset O(n^k) \subset O(a^n) \subset O(b^n) \subset O(n!) \subset O(n^n)$, donde k>j>2, b>a>1

- El orden de un polinomio $a_n x^n + ... + a_1 x + a_0$ es $O(x^n)$.

- $\sum_{i=1}^{n} 1 \in O(n)$; $\sum_{i=1}^{n} i \in O(n^2)$ $\sum_{i=1}^{n} i^m \in O(n^{m+1})$

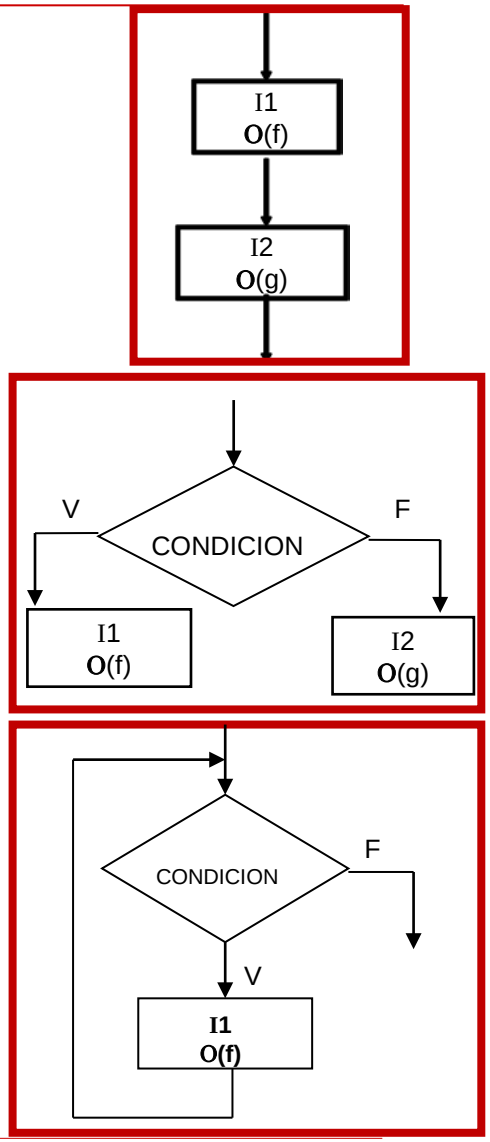
- Si hacemos una operación para n, otra para n/2, n/4, ..., aparecerá un orden logarítmico $O(\log_2 n)$.
- Los **logaritmos** son del mismo orden, independientemente de la base. Por eso, se omite normalmente.
- **Sumatorios:** se pueden aproximar con integrales, una acotando superior y otra inferiormente.
- Casos **promedios**: usar probabilidades.

4. Análisis de algoritmos iterativos.

- El análisis de un algoritmo precisa de un conteo de los recursos consumidos.
- En un algoritmo sin llamadas recursivas, una vez seleccionados el tamaño del problema y la operación básica, el análisis se realiza utilizando técnicas tradicionales de conteo (sucesiones, progresiones aritméticas, sumas, etc.) y el concepto de orden de una función.
- Análisis por bloques
  - El modelo matemático para comparar y evaluar el costo de los algoritmos que considera el comportamiento de las funciones complejidad cuando el tamaño del problema es grande, proporciona un importante medio para hacer un análisis más dinámico, que se basa principalmente en el conocimiento del orden de los bloques de instrucciones.
  - El orden de un bloque, se determina por el orden de las estructuras de control más usuales.

4. Análisis de algoritmos iterativos.

- **Secuencia.** Sea { I_1, I_2 } una secuencia de instrucciones, con complejidades O(*f*) y O(*g*) respectivamente, entonces el orden de la secuencia es, O(**máx**(*f,g*)).
- **Selección Condicional.** En el constructor selección condicional se evalúa la condición *C,* si es verdadera, se ejecuta el bloque I1 que tiene complejidad O(*f*) y si es falso se ejecuta *I*2 que es *O*(*g*); la **complejidad** de la selección es:
  - En el peor caso, esta determinada por el orden de la instrucción que implica más operaciones, O(**máx**(*f,g*))
  - Para el mejor caso, se considera la instrucción con menor nº de operaciones, la complejidad será O(**mín**(*f,g*))
  - Para el caso promedio, se toma en cuenta la probabilidad *P* de que la condición sea verdadera, la complejidad es O(***Pf***) + O(**(1-*P*)*g*).
- **Iteración.** Si el bloque de complejidad O(*f*) se realiza ***n*** veces, deducimos que la iteración es O(*nf*(*n*)).
  - En muchas ocasiones la complejidad del bloque depende de algún índice. Si la complejidad es O(*f*(*i*)) para 1≤i≤n la complejidad de la iteración es $\sum_{i=1}^{n} f(i)$



4. Análisis de algoritmos iterativos. Ejemplo.

- **Ejemplo**. Sean *A* y *B* dos matrices de *nxn* con entradas reales. Calcular la matriz producto *C* = *AB.* El algoritmo que se utilizará es implicado por la definición del producto de matrices, donde la matriz *C* tiene la forma :

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \qquad i, j \in [1, n]$$

```
Algoritmo Matriz_Producto (n, A, B)
        para_ i = 1  hasta n  hacer                          I1
              para_ j = 1 hasta n  hacer                     I2
                    C [i, j] ← 0;                            I3
                    para_k = 1 hasta n  hacer                I4
                      C [i, j] ← C [i, j] + A [i, k] * B [k, j]   I5
                    fpara
              fpara
         fpara
f Algoritmo
```

- **Tamaño del problema**: ***n;*** la dimensión de las matrices.
- **Operación básica**: la multiplicación entre las entradas de las matrices.
- **Caso** : El algoritmo hace el mismo número de operaciones en todos los casos.

Análisis Temporal

1. De manera informal se puede decir que para obtener una entrada de c_{ij} se hacen *n* multiplicaciones, como la matriz *C* tiene *n x n* entradas se tiene que:

$$\mathbf{T(n) = n(n^2) = n^3 \text{ es } O(n^3)}$$

2. Podemos obtener el resultado anterior, si asignamos un orden (según su comportamiento asintótico) a cada bloque de instrucciones según las estructuras de control:
   - **I5** tiene orden constante, entonces es **O(1)**
   - Como **I5** está dentro del ciclo **I4** entonces el orden es **O(*nx*1)= O(*n*).**
   - El ciclo **I2** contiene al conjunto de instrucciones I3 e I4*;* si I3 es *O*(1) se tiene que la secuencia tiene orden *O*(*n*)*;* pero ésta se realiza *n* veces, así el orden para este bloque es **O(*n x n*)= O(n^2)**.
   - Por último, el ciclo **I1** contiene a la instrucción **I2** que se realiza *n* veces de ésta forma el orden para **I1***;* y en consecuencia para el Algoritmo es:

$$\mathbf{O(n \times n^2) = O(n^3)}$$

5. Análisis de algoritmos recursivos. Resolución de recurrencias.

- Al analizar un algoritmo recursivo habitualmente se obtiene la función de complejidad en forma de recurrencia o **sistema recurrente.**
- **Ejemplo**: función factorial.

```
funcion factorial(n)
        si n=0 entonces
            factorial ← 1
        sino
            factorial ← n*factorial(n-1)
        fin si
ffuncion
```

- El número de operaciones elementales en este caso viene dado por:

$$T(n) = \begin{cases} 2 & \text{si } n = 0 \\ 5 + T(n-1) & \text{si } n > 0 \end{cases}$$

- Para clasificar el algoritmo en una categoría de complejidad, necesitamos resolver este tipo de recurrencias o sistema recurrente o el equivalente con la elección de operación básica.

5. Análisis de algoritmos recursivos. Resolución de recurrencias.

- Elección de operación básica.
  - Para encontrar la complejidad temporal debemos elegir una operación tal que el orden del número de veces que se realiza sea igual al orden del total de operaciones que hace el algoritmo, puesto que la función es recursiva, la parte más costosa es la recursión y tenemos que elegir una operación que se efectúe en todas las llamadas que hace el algoritmo o en un número de ellas que sea del mismo orden que el orden del número total de llamadas.
  - De las operaciones que realiza el algoritmo podemos contar multiplicaciones (operación que cumple con las condiciones estipuladas).
  - El algoritmo para resolver un problema de tamaño ***n,*** hace una multiplicación y resuelve un problema de tamaño ***n-1*** hace una multiplicación y resuelve un problema de tamaño ***n-2*** . . . y resuelve un problema de tamaño ***1*** hace una multiplicación y resuelve un problema de tamaño ***0***, para el que ya no ejecuta dicha operación.
- Puesto que la función tiene 2 partes, el caso base y la parte recursiva, la función complejidad se puede expresar como un sistema de 2 ecuaciones:

$$T(0) = 0 \quad \text{para el caso base}$$
$$T(n) = T(n-1) + 1 \quad \text{para el caso general}$$

  - Estas ecuaciones forman un sistema recurrente.

5. Análisis de algoritmos recursivos. Resolución de recurrencias.

- En un sistema recurrente se distinguen 2 partes:
  1. La ecuación recurrente, que da el valor de una función en un punto *n* en términos de su valor en otros puntos *m* con *n* > *m:* En nuestro caso, la ecuación T(n) = T(n - 1) + 1 da el valor de T(n) en términos del valor de T(n - 1)
  2. Un conjunto de valores de frontera o condiciones iniciales (los valores de una función en puntos específicos) que determinan, junto con la ecuación recurrente, completamente a una función. En nuestro ejemplo T(0) = 0
- Si se tiene una condición inicial, tal como T(0) = 0*;* entonces podemos calcular T(n); usando la ecuación recurrente, para otros valores de *n.* Así

  $T(1) = T(0) + 1 = 1$

  $T(2) = T(1) + 1 = 2$

  $T(3) = T(2) + 1 = 3$
- Una solución a un sistema recurrente es una función que satisface la ecuación recurrente y las condiciones iniciales.
- La solución a nuestro sistema es $\boldsymbol{T(n) = n}$
- **Comprobamos** que la solución a nuestro sistema es **T(n) = n:**
  - Si *n* = 0 entonces T(0) = 0 satisface la condición inicial, y
  - Si *n* > 0 entonces T(n) = T(n - 1) + 1 = (n -1) + 1= n satisface la ecuación recurrente, por lo tanto es una solución al sistema.

5. Análisis de algoritmos recursivos. Resolución de recurrencias.

- Para encontrar la solución a un sistema recurrente existe toda una teoría que expone diversos métodos. En el análisis de algoritmos recursivos, los sistemas recurrentes que resultan son de baja complejidad y en la mayoría de los casos basta la aplicación de una técnica muy simple para resolverlos.

- Los métodos para resolver las recurrencias mas comunes son:
  - Expansión de recurrencias
  - Método de la ecuación característica
  - Cambio de variable
  - Recurrencias No Lineales.
  - Recurrencias típicas
    - Reducción por sustracción
    - Reducción por división

5. Resolución de recurrencias. Expansión de recurrencias.

- Este método consiste en ir sustituyendo las llamadas recurrentes por su definición, con el objetivo de encontrar una regla general.

- Este método de solución, utiliza la recurrencia misma para sustituir $T(n)$ por su valor en términos de $T(m)$*;* donde $m < n$*;* iterativamente hasta encontrar una condición de frontera $T(r)$ que tendrá un valor constante.

  1. En la ecuación recurrente se ponen todos los términos con *T* en el lado izquierdo y

  2. se realiza la expansión.

  3. Al sumar todas las ecuaciones obtendremos una expresión para ***T(n)*** en términos de *n* y algunas constantes.

  A esta expresión final se le denomina forma cerrada para $T(n)$*:*

5. Resolución de recurrencias. Expansión de recurrencias.

- Ejemplo. Utilicemos el sistema obtenido para la función factorial:
  $T(n) = T(n-1) + 1$

1. ponemos los términos de T en el lado izquierdo
   $T(n) - T(n-1) = 1$
2. expandimos la recurrencia
   $T(n) - T(n-1) = 1$
   $T(n-1) - T(n-2) = 1$
   $T(n-2) - T(n-3) = 1$
   ...
   $T(2) - T(1) = 1$
   $T(1) - T(0) = 1$
   $T(0) = 0$
3. sumamos todas las ecuaciones
   $T(n) = \underbrace{1 + 1 + \ldots\ldots\ldots + 1}_{n \text{ veces}}$
   $T(n) = n$

5. Resolución de recurrencias. Método de la ecuación característica.

- Este método es útil para recurrencias lineales, en las que el valor *n* de la recurrencia depende de los *k* anteriores.
- Se pueden dar dos casos:

1. **Recurrencias homogéneas**, de la forma:

$$a_0T(n) + a_1T(n-1) + \ldots + a_kT(n-k) = 0$$

2. **Recurrencias no homogéneas**, de la forma:

$$a_0T(n) + a_1T(n-1) + \ldots + a_kT(n-k) = f(n)$$

- En ambos casos este método consiste en obtener una ecuación característica, cuyas raíces nos permitirán obtener la representación no recurrente de la función de complejidad

5. Resolución de recurrencias. Recurrencias homogéneas.

- Las recurrencias homogéneas son de la forma:

$$\mathbf{a_0T(n) + a_1T(n-1) + \ldots + a_kT(n-k) = 0} \qquad (1)$$

  - Haciendo el cambio $\mathbf{T(n) = x^n}$, obtenemos:

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

  - $r = 0$ es una raíz trivial sin interés de multiplicidad $n-k$ de la expresión anterior. Las demás raíces son las de la siguiente ecuación:

    $\mathbf{a_0\, x^k + a_1\, x^{k-1} + \ldots + a_k = 0}$ (**ecuación característica**)

  - Sean $r_1, \ldots, r_k$ las raíces de la ecuación característica. Según su multiplicidad se dan los siguentes casos:

1. Raices distintas (multiplicidad 1)

  - Deshaciendo el cambio, r_i^n son soluciones de (1)
  - Por tanto, la forma de la función de complejidad es:

$$T(n) = c_1 r_1^n + c_2 r_2^n + \ldots + c_k r_k^n = \sum_{i=1}^{k} c_i r_i^n$$

  - Los **coeficientes $\mathbf{c_i}$** se obtienen a partir de las condiciones iniciales

5. Resolución de recurrencias. Recurrencias homogéneas.

- Ejemplo: secuencia de Fibonacci:
  - T(0) = 0, T(1) = 1
  - T(n) = T(n-1) + T(n–2) n≥2
- Solución:

$$T(n) \in O(\varphi n)$$

$$\varphi = \left(\frac{1+\sqrt{5}}{2} \right) = número\_áureo$$

5. Resolución de recurrencias. Recurrencias homogéneas.

2. Raíces con multiplicidad mayor a 1, el método es una generalización del anterior.

- Sea la ecuación característica $a_0 x^k + a_1 x^{k-1} + \ldots + a_k = 0$ con raíces $r_1, \ldots, r_k$ de multiplicidad $m_1, \ldots, m_k$.
  - Supongamos que alguna de las raíces (por ej. r_1) tiene multiplicidad $m \geq 1$, entonces la ecuación característica se puede poner como $(x-r_1)^m (x-r_2)\ldots (x-r_{k-m+1})$
  - En cuyo caso la función de complejidad es:

$$T(n) = \sum_{i=1}^{m} c_i n^{i-1} r_1^n + \sum_{i=m+1}^{k} c_i r_{i-m+1}^n$$

- En general la **función complejidad** es:

$$T(n) = \sum_{h=0}^{m_1-1} c_{1h} n^h r_1^n + \ldots + \sum_{h=0}^{m_k-1} c_{kh} n^h r_k^n$$

$$T(n) = \sum_{i=1}^{k} \sum_{j=0}^{m_i-1} c_{ij} n^j r_i^n$$

  - Los coeficientes c_{ij} se obtienen a partir de las condiciones iniciales.

- **Ejemplo**: (raíces del pol. característico: $r_1 = 2$ con mult. 2 y $r_2 = 1$)
  - T(n) = 5T(n - 1) - 8T(n - 2) + 4T(n - 3),
  - T(k) = k, para k=0,1,2

5. Resolución de recurrencias. Recurrencias no homogéneas.

- Las recurrencias no homogéneas son de la forma:

$$a_0T(n) + a_1T(n-1) + \ldots + a_kT(n-k) = f(n)$$

donde f (n) es distinta de la función nula (f(n) ≠ 0 para algún n)

- No se conoce una solución general para cualquier f (n), pero sí para las de la forma:

$$a_0T(n) + a_1T(n-1) + \ldots + a_kT(n-k) = b^n p(n)$$

donde p(n) es un polinomio de grado ***d*** en n

  - La **ecuación característica** en este caso es:

$$(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x - b)^{d+1} = 0$$

  - **Se procede igual que en las recurrencias homogéneas.**

- Ejemplo: $T(n) = 2T(n-1) + 3^n$, $T(0) = 0$, $T(1) = 1$

5. Resolución de recurrencias. Recurrencias no homogéneas.

- Generalizando el proceso, el caso en el que f (n) tenga la forma $b_1^n p_1(n) + ... + b_s^n p_s(n)$ cada término contribuye a la ecuación característica.

- La ecuación característica es:

$$(a_0 x^k + a_1 x^{k-1} + ... + a_k)(x - b_1)^{d_1+1})(x - b_2)^{d_2+1} \dots (x - b_s)^{d_s+1} = 0$$

- Ejemplo: T(n) = 2T(n-1)+n+2^n, n≥1 con la condición inicial T(0)=1

5. Resolución de recurrencias. Cambio de variable.

- **Cambio de variable.** Consiste en hacer un cambio de variable para transformar la recurrencia original T(n) en otra recurrencia t_k para la que se pueden aplicar las técnicas anteriores.
- Esta técnica se aplica cuando *n* es potencia de un número real *a,* esto es, $\mathbf{n=a^k}$.
- **Ejemplo**: Sea por ejemplo, para el caso a=2, la ecuación

  T(n) = 4T(n/2) + n, donde n es una potencia de 2(n>3), T(1) = 1 y T(2)=6.
  - Si $n=2^k$ podemos escribir la ecuación como:

    $T(2^k) = 4T(2^{k-1}) + 2^k$
  - Haciendo el cambio de variable $t_k = T(2^k)$, tenemos

    $t_k = 4\ t_{k-1} + 2^k$ [$p(x)=(x-4)(x-2) \Rightarrow r_1=4 ; r_2=2$]
  - Esta es una recurrencia no homogénea, con solución

    $t_k = c_1(2^k)^2 + c_2 2^k$
  - Deshacemos el cambio de variable:

    $T(n) = c_1 n^2 + c_2 n$
  - Aplicando las condiciones iniciales, obtenemos $T(n) = 2n^2 - n$

5. Resolución de recurrencias. Recurrencias No Lineales.

- **Recurrencias No Lineales:** la ecuación que relaciona T(n) con el resto de los términos no es lineal. Para resolverla se intenta convertir en una ecuación lineal como las anteriores.
- **Ejemplo**: $T(n)=nT^2(n/2)$ para n potencia de 2(n>1), T(1) = 1/3.
- Llamando $t_k = T(2^k)$, la ecuación queda como:

  $t_k = T(2^k)= 2^kT^2(2^{k-1}) = 2^k t_{k-1}^2$, que no corresponde a ninguno de los tipos estudiados por lo que hacemos un nuevo cambio:
- Tomamos logaritmos y haciendo el cambio $u_k=\log t_k$ obtenemos: (logxy=logx+logy)

  $u_k-2u_{k-1}=k$ ec. en recurrencia no homogénea con ec. característica: $(x-2)(x-1)^2=0$. Por tanto:

  $u_k = c_1 2^k +c_2+c_3 k$
- Deshaciendo los cambios:

1. $u_k=\log t_k$ entonces $t_k = 2^{c_1 2^k +c_2+c_3 k}$
2. $t_k = T(2^k)$ entonces $T(n) = 2^{c_1 n+c_2+c_3 \log n}$
3. Calculamos las constantes con la condición inicial y utilizando la ecuación de recurrencia original para calcular las otras 2 que necesitamos (son 3 incógnitas).

   $T(2)= 2T^2(1)= 2/9$ y $T(4)= 4T^2(2)=16/81$ con lo cual

   $c_1=\log(4/3)=2-\log 3$; $c_2=-2$; $c_3=-1$ y por tanto: $T(n) = \frac{2^{2n}}{4n3^n}$

5. Resolución de recurrencias. Recurrencias típicas. Fórmulas maestras.

- **Reducción por sustracción** :

$$T(n) = \begin{cases} c & \text{Si } 0 \le n < b \\ a \cdot T(n-b) + p(n) & \text{Si } n \ge b \end{cases}$$

c es el coste en el caso directo,
a es el número de llamadas recursivas,
b es la disminución del tamaño de los datos, y
$p(n)=c * n^k$ es el coste de preparación de las llamadas y de combinación de los resultados.

donde $a, c \in R^+$, $p(n)$ es un polinomio de grado k ($p(n)=cn^k$), y $b \in N$

entonces se tiene que $\mathbf{T(n)} \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n/b}) & \text{si } a > 1 \end{cases}$

- **Ejemplo:**

$$T(n) = \begin{cases} c_1 & \text{Si } n=0 \\ T(n-1) + c_2 & \text{Si } n > 0 \end{cases}$$

$a=1, b=1, k=0$ entonces $T(n) \in \Theta(n^{k+1})= \Theta(n)$

- **Reducción por división** :

$$T(n) = \begin{cases} c & \text{Si } 1 \le n < b \\ a \cdot T(n/b) + f(n) & \text{Si } n \ge b \end{cases}$$

c es el coste en el caso directo,
a es el número de llamadas recursivas,
b factor de disminución del tamaño de los datos, y
f(n)= c * n^k es el coste de preparación de las llamadas y de combinación de los resultados.

donde $a, c \in R^+$, $f(n) \in \Theta(n^k)$, $b \in N, b > 1$

entonces se tiene que $T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log n) & \text{si } a = b^k \\ \Theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$

- **Ejemplo.** Búsqueda binaria

$$T(n) = \begin{cases} c_1 & \text{Si } n=0 \\ T(n/2) + c_2 & \text{Si } n > 0 \end{cases}$$

a=1, b=2, k=0 entonces **$a= b^k$** y la complejidad es: $T(n) \in \Theta(n^k \log n)= \Theta(n^0 \log n)= \Theta(\log n)$