



Instituto Politécnico Nacional

Escuela Superior de Cómputo



Análisis de algoritmos

Tema 01: El rol de los algoritmos en la Computación

M. en C. Edgardo Adrián Franco Martínez
<http://www.eafranco.com>
edfrancom@ipn.mx
[@edfrancom](https://twitter.com/edfrancom) [@edgaroadrianfrancocom](https://facebook.com/edgaroadrianfrancocom)



Contenido

- Introducción
- Historia
- Algoritmo
- Características de los algoritmos
- Importancia de los algoritmos en computación
- Introducción a la complejidad de los algoritmos
- Análisis A Priori y Prueba A Posteriori

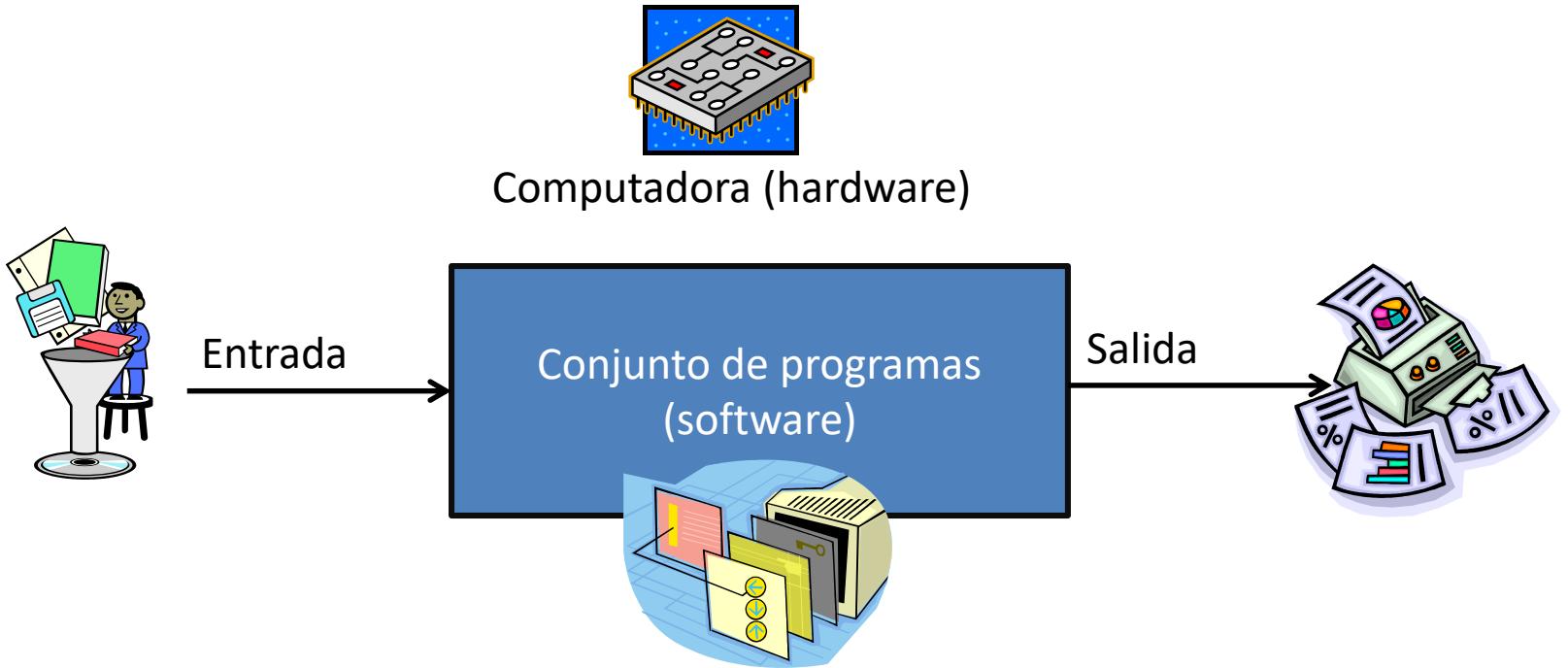


Introducción

- Una computadora es una **máquina capaz de procesar información digital a gran velocidad.**
- Una computadora está compuesta por un conjunto de componentes electrónicos, mecánicos e interfaces para interactuar con el exterior (usuarios u otros dispositivos) y por un conjunto de programas que determinan qué operaciones llevar a cabo.
- Los datos ordenados (**información**) que constituyen una entrada (*input*) a la computadora se procesan mediante una lógica (programa) para producir una salida (*output*).

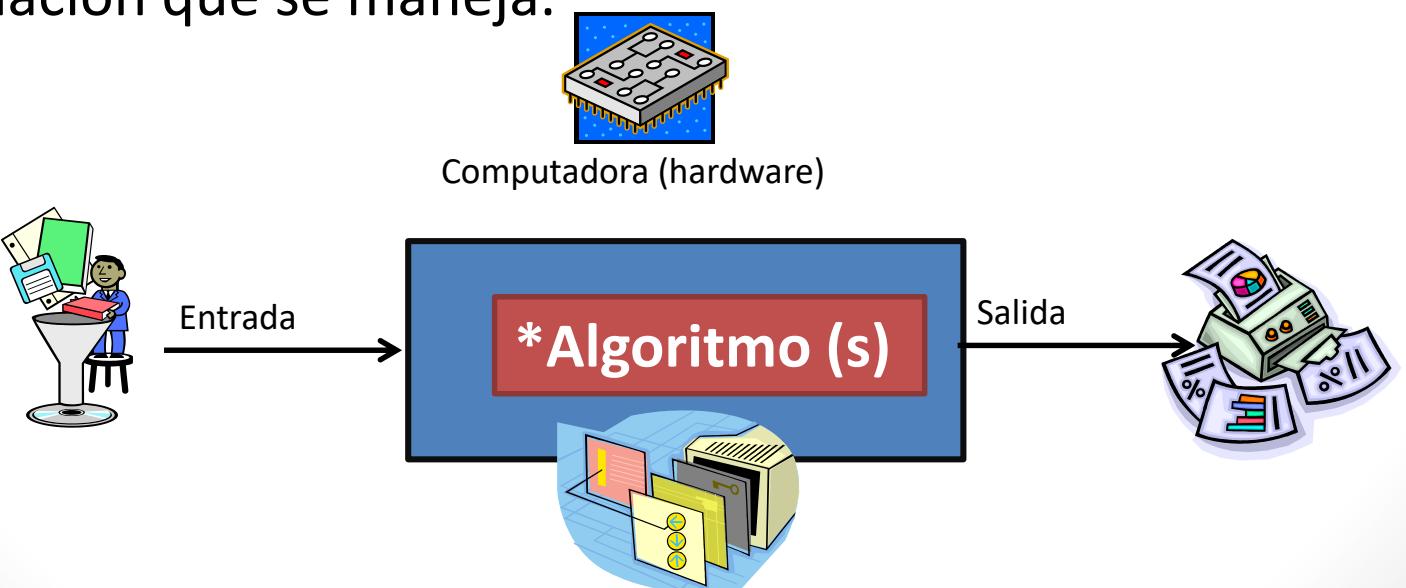


- Una computadora es una **máquina capaz de procesar información digital a gran velocidad.**



Una computadora esta formada por un parte física y otra lógica (hardware & software), la primera de estas esta conformada por los elementos físicos que la conforman (dispositivos electrónicos y mecánicos), la parte lógica es aquella que determina que procesos se van a realizar con la información de entrada.

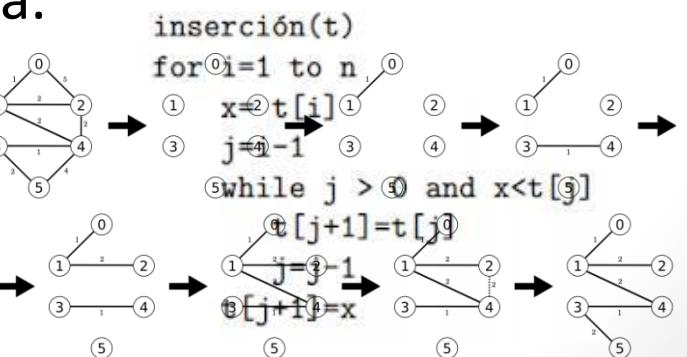
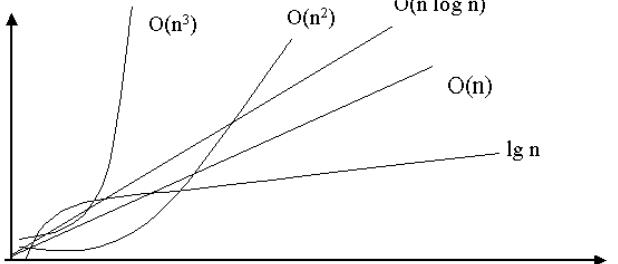
- La persona responsable de indicar a la computadora la **lógica** de procesamiento recae en el que lleva a cabo la construcción del software (**programador**).
- La razón de ser de una computadora es poder **resolver problemas** capaces de ser modelados y representados en datos coherentes y ordenados (información), apoyándose de su gran velocidad y **capacidad de seguir una serie de pasos programados con anterioridad*** y dependientes de la información que se maneja.



- **Algoritmo**, es un conjunto ordenado y finito de operaciones que permiten encontrar la solución a un problema.
- P.g. una receta de cocina, las instrucciones para armar una bicicleta, un mueble, etc.
- Los primeros algoritmos registrados datan de Babilonia, originados en las matemáticas como un método para resolver un problema usando una secuencia de cálculos más simples. Esta palabra tiene su origen en el nombre de un famoso matemático y erudito árabe del siglo IX, **Al-Khorezmi**.
- En el contexto de la computación **algoritmo** se usa para denominar a la **secuencia de pasos a seguir para resolver un problema usando una computadora**. Por esta razón, la algoritmia o ciencia de los algoritmos, es uno de los **pilares de la computación**.



- El **análisis de algoritmos** es una parte importante de la **Teoría de complejidad computacional**, que provee estimaciones teóricas para los recursos que necesita cualquier algoritmo que resuelva un problema computacional dado. Estas estimaciones resultan ser bastante útiles en la búsqueda de algoritmos eficientes.
- Los temas de mayor interés son el **análisis teórico de algoritmos** lo que permite, calcular su **complejidad en un sentido asintótico**, así como el **análisis de problemas comunes** que requieren una cantidad de procesamiento alta de los datos para poder ser resueltos con exactitud o aproximación a la respuesta óptima.





Historia

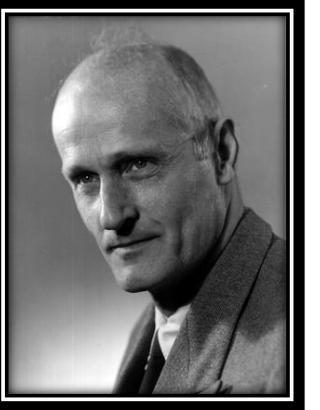
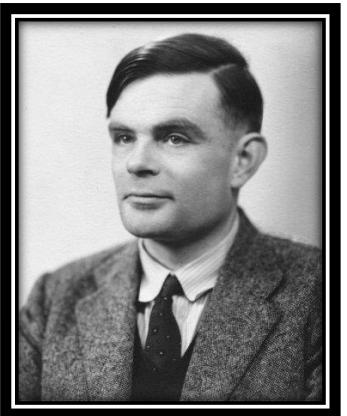
- El término **“algoritmo”** proviene del matemático árabe **Al'Khwarizmi**, que escribió un **tratado sobre los números**. El trabajo de Al'Khwarizmi permitió preservar y difundir el conocimiento de los griegos e indios, pilares de nuestra civilización.
- Rescató de los griegos la rigurosidad y de los indios la simplicidad (*en vez de una larga demostración, usar un diagrama junto a la palabra Mira*). Sus libros son intuitivos y prácticos y su principal contribución fue simplificar las matemáticas a un nivel entendible por no expertos.
- En el tratado se muestran las ventajas de usar el sistema decimal indio, un atrevimiento para su época, dado lo tradicional de la cultura árabe. La exposición clara de cómo **calcular de una manera sistemática a través de algoritmos diseñados** para ser usados con algún tipo de dispositivo mecánico similar a un ábaco, más que con lápiz y papel, muestra la intuición y el poder de **abstracción** de Al'Khwarizmi, que se preocupaba también de reducir el número de operaciones necesarias en cada cálculo.



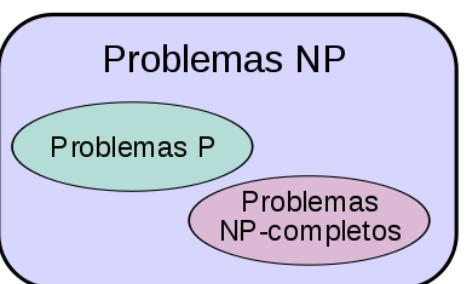
- Al-Khwarizmi vivió en Bagdad bajo los califatos de al-Ma'mum y al-Mu'tasim, en la edad de oro de la ciencia islámica. Su obra *Kitab al-jabr wa al-muqabalah* fue traducida al latín en el siglo XII dando origen al término "**álgebra**". En ella se compilan una **serie de reglas para obtener las soluciones aritméticas de las ecuaciones lineales y de las cuadráticas**; su método de resolución de tales ecuaciones no difiere en esencia del empleado en nuestros días.
- Como ya se vio es un error creer que los algoritmos son exclusivos de la computación, el algoritmo más famoso de la historia es **algoritmo de Euclides** para calcular el máximo común divisor.



- La investigación en **modelos formales de computación** se inició en los **30's y 40's** por **Alan Turing, Emil Leon Post, Stephen Kleene, Alonzo Church** y otros.
- En los **50's y 60's** los **lenguajes de programación, compiladores y sistemas operativos** estaban en desarrollo, por lo tanto, se convirtieron tanto en el sujeto como la base para la mayoría del trabajo teórico.



- El poder de las computadoras en este período estaba limitado por procesadores lentos y por pequeñas cantidades de memoria. Así, se desarrollaron **teorías (modelos, algoritmos y análisis)** para hacer un **uso eficiente** de las computadoras. Esto dio origen al desarrollo del área que ahora se conoce como "*Algoritmos y Estructuras de Datos*".
- Al mismo tiempo se hicieron estudios para comprender la **complejidad inherente en la solución de algunos problemas**. Esto dio origen a lo que se conoce como la "**Jerarquía de problemas computacionales**" y al área de "**Complejidad Computacional**".



Algoritmo

“Un algoritmo es una serie finita de pasos para resolver un problema”.

- Para que un algoritmo exista:
 1. El número de pasos debe ser finito. De esta manera el algoritmo debe terminar en un tiempo finito con la solución del problema.
 2. El algoritmo debe ser capaz de determinar la solución del problema.
- De este modo, podemos definir **algoritmo computacional** como un "conjunto de reglas operacionales inherentes a un cómputo".
 - Se trata de un **método sistemático, susceptible de ser realizado mecánicamente, para resolver un problema dado en un tiempo finito.**



Características de un algoritmo

1. **Entrada:** definir lo que necesita el algoritmo
 2. **Salida:** definir lo que produce.
 3. **No ambiguo:** explícito, siempre sabe qué comando ejecutar.
 4. **Finito:** El algoritmo termina en un número finito de pasos.
 5. **Correcto:** La solución debe ser correcta
 6. **Efectividad:** Cada instrucción se completa en tiempo finito.
 7. **General:** Debe contemplar todos los casos de entrada.
- Una definición formal para el algoritmo es:

“Un algoritmo es un procedimiento para resolver un problema cuyos pasos son concretos y no ambiguos. El algoritmo debe ser correcto, de longitud finita y debe terminar para todas las entradas”



- Un algoritmo puede ser visto como una función que depende de una entrada y asocia una salida, matemáticamente se expresa de la siguiente forma:

$$y = f(x)$$

donde:

“ y ” es la salida

“ x ” los datos de entrada

“ $f(x)$ ” el algoritmo que depende de la entrada



Importancia de los algoritmos en computación

- **Evitan reinventar soluciones**



- Para algunos problemas a resolver mediante una computadora ya existen buenos algoritmos para solucionarlos, ya que existe un análisis de sus propiedades (**confianza, eficiencia, velocidad, etc.**). Por lo que a no ser que se este investigando aún más la solución de ese problema es conveniente utilizar un algoritmo conocido.

- **Ayudar a generar nuevas soluciones con base en otras ya conocidas**



- No siempre existe un algoritmo desarrollado para resolver un problema.
- No existe regla general de creación de algoritmos. Muchos de los principios de proyecto de algoritmos ilustrados por algunos de los algoritmos que estudiaremos son importantes en todos los problemas de programación. El conocimiento de los algoritmos bien definidos provee una fuente de ideas que pueden ser aplicadas a nuevos algoritmos.



- **Generación de herramientas y software particular que emplea algoritmos específicos sin importar la tecnología de implementación**

- En muchos casos una solución específica requiere la implementación de otro tipo algoritmos específicos y no se pueden cambiar. P.g. Compresión de datos, seguridad, almacenamiento de datos, comunicaciones, etc.



Introducción a la complejidad de los algoritmos

- En Ciencias de la Computación cuando **se dice que un problema tiene solución**, significa que existe un **algoritmo susceptible de implantarse en una computadora**, capaz de producir la respuesta correcta para cualquier instancia (caso) del problema en cuestión.
- Para ciertos problemas es posible encontrar **más de un algoritmo** capaz de resolverlos.
- La tarea de decidir cuál de ellos es el mejor debe basarse en criterios acordes a nuestros intereses. En la mayoría de los casos la elección de un buen algoritmo está orientada hacia la **disminución del costo** que implica la **solución del problema**.



- El costo que implica resolver un problema genera criterios para la selección de los pasos a seguir (**programar**):
 1. Criterios orientados a minimizar el **costo de desarrollo**: claridad, sencillez y facilidad de implantación, depuración y mantenimiento.
 2. Criterios orientados a disminuir el **costo de ejecución**: tiempo de procesador y cantidad de memoria utilizados.
- Si el programa que implanta un algoritmo se va a ejecutar **unas cuantas veces**, el **costo de desarrollo es dominante**, en este caso se debe dar más peso a los primeros criterios.
- Por el contrario, si el programa se va a **utilizar frecuentemente**, **dominará el costo de ejecución**, y por ende, se debe elegir un algoritmo que haga uso eficiente de los recursos de la computadora.

- Si se implementa un sistema con **algoritmos** muy eficientes pero **confusos**, puede suceder que después no sea posible darle el mantenimiento adecuado.
- 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.
- Siempre que se trata de resolver un problema, puede interesar considerar distintos algoritmos, con el fin de utilizar el más eficiente **¿Cómo realizar esta selección?**
 - La **estrategia empírica** consiste en programar los algoritmos y ejecutarlos en un computador sobre algunos ejemplares de prueba.
 - La **estrategia teórica** consiste en determinar matemáticamente la cantidad de recursos (tiempo, espacio, etc.) que necesitará el algoritmo en función del tamaño del ejemplar considerado.



- Un **algoritmo es eficaz** cuando alcanza el objetivo primordial, el análisis de resolución del problema se realiza prioritariamente.

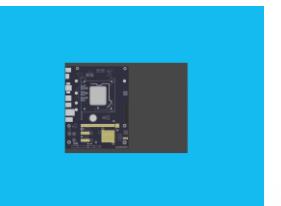


- Un **algoritmo es eficiente** cuando logra llegar a sus objetivos planteados **utilizando la menor cantidad de recursos posibles**, es decir, minimizando el uso memoria, de pasos y de esfuerzo.



- ¿Puede darse el caso de que exista un algoritmo eficaz pero no eficiente?
 - Si, por lo que se deben manejar estos dos conceptos conjuntamente

- La eficiencia de un programa tiene dos ingredientes fundamentales: **espacio** y **tiempo**.
 - La **eficiencia en espacio** es una medida de la cantidad de memoria requerida por un programa.
 - La **eficiencia en tiempo** se mide en términos de la cantidad de tiempo de ejecución del programa.
- Ambas dependen del tipo de **equipo de computo, compilador, sistema operativo**, etc. por lo que no se estudiará aquí la eficiencia de los programas, sino la **eficiencia de los algoritmos**.

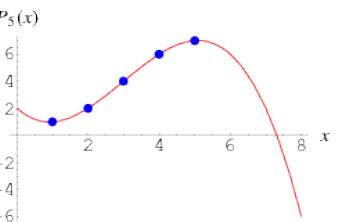
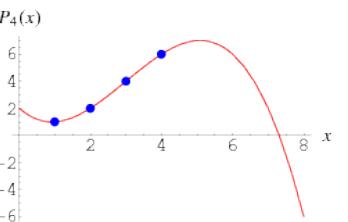
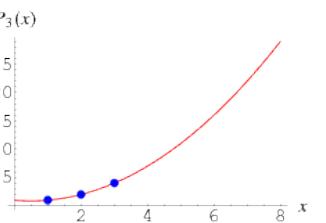
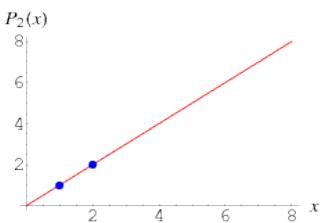
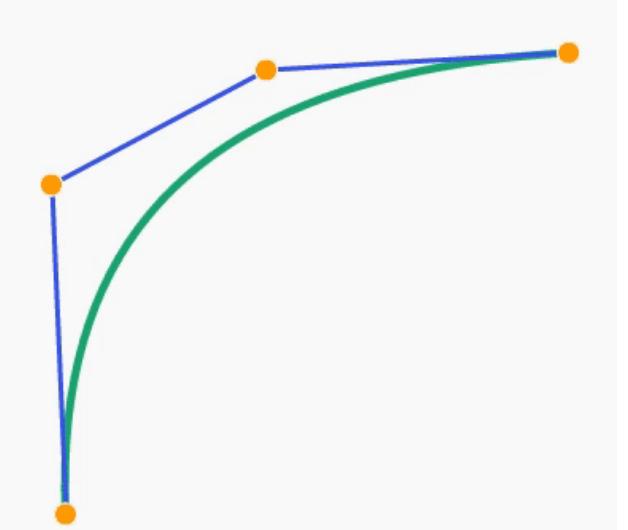


Análisis A Priori y Prueba A Posteriori

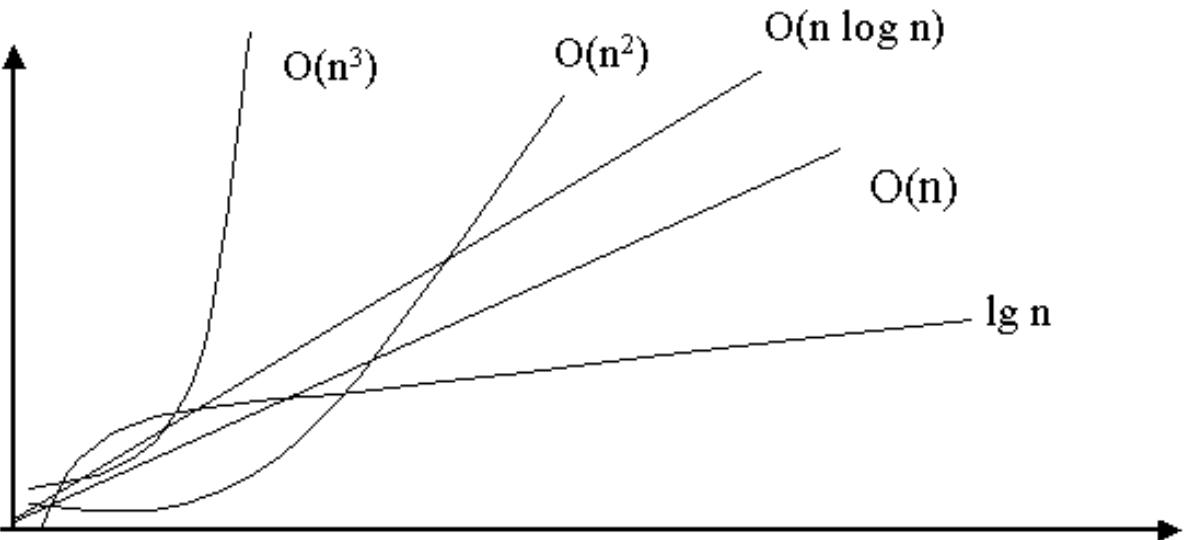
- El **análisis de la eficiencia de los algoritmos** (memoria y tiempo de ejecución) consta de dos fases: **Análisis A Priori y Prueba A Posteriori**.
- En la **Prueba A Posteriori (experimental o empírica)** se recogen estadísticas de tiempo y espacio consumidas por el algoritmo mientras se ejecuta.
 - La estrategia empírica consiste en programar los algoritmos y ejecutarlos en un computador sobre algunos ejemplares de prueba, haciendo medidas para:
 - Una máquina concreta,
 - Un lenguaje concreto,
 - Un compilador concreto
 - Datos concretos
- El **Análisis A Priori (o teórico)** entrega una función que limita el tiempo de cálculo de un algoritmo.
 - Consiste en obtener una expresión que indique el comportamiento del algoritmo en función de los parámetros que influyan. Esto es interesante porque: Permite la predicción del costo del algoritmo. Es aplicable en la etapa de diseño de los algoritmos.



- La **estrategia experimental (Análisis A Posteriori)** consiste en implementar y con base en varias instancias del problema aproximar el comportamiento experimental con una función que modela el experimento.



- La **estrategia teórica (Análisis A Priori)** tiene como ventajas:
 - No depende del computador ni del lenguaje de programación, ni siquiera de la habilidad del programador.
 - Permite evitar el esfuerzo inútil de programar algoritmos ineficientes y de desperdiciar tiempo de máquina para ejecutarlos.
 - Permite conocer la eficiencia de un algoritmo cualquiera que sea el tamaño del ejemplar al que se aplique.





Instituto Politécnico Nacional

Escuela Superior de Cómputo



Análisis de algoritmos

Tema 02: Complejidad de los algoritmos

M. en C. Edgardo Adrián Franco Martínez
<http://www.eafranco.com>
edfrancom@ipn.mx
[@edfrancom](https://twitter.com/edfrancom) [@edgardoадrianfranco](https://facebook.com/edgardo.adrian.franco)



Contenido

- Algoritmo
- Algoritmo vs. Proceso Computacional
- Tamaño de problema
- Función complejidad
- Análisis Temporal
- Análisis Espacial
- Medición del tiempo de ejecución
- Medición de memoria requerida
- Ejemplos



Algoritmo

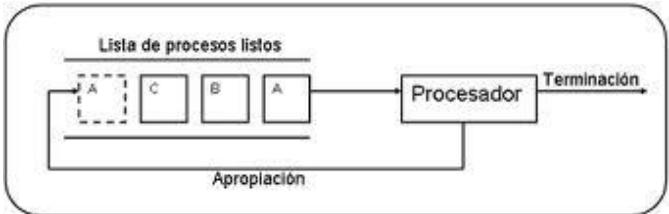
“Un algoritmo es un procedimiento para resolver un problema cuyos pasos son concretos y no ambiguos. El algoritmo debe ser correcto, de longitud finita y debe terminar para todas las entradas”

- Un **paso** es **NO ambiguo** cuando la acción a ejecutar está perfectamente definida:
 - $x \leftarrow \log(0)$ *Ambigua*
 - $x \leftarrow \log(10) + 5$ *NO Ambigua*
- Una **instrucción es concreta o efectiva** cuando se puede ejecutar en un intervalo finito de tiempo
 - $x \leftarrow 2 + 8$ *Efectiva*
 - $mensaje \leftarrow Concatena('Hola', 'Mundo')$ *Efectiva*
 - $x \leftarrow cardinalidad(números naturales)$ *NO Efectiva*



Algoritmo vs. Proceso Computacional

- Si un conjunto de instrucciones a computar tiene todas las **características de un algoritmo, excepto ser finito en tiempo** se le denomina **proceso computacional**.
- Los sistemas operativos son el mejor ejemplo de proceso computacional, pues están diseñados para ejecutar tareas mientras las haya pendientes, y cuando éstas se terminan, el sistema operativo entra en un estado de espera, hasta que llegan más, pero nunca termina.

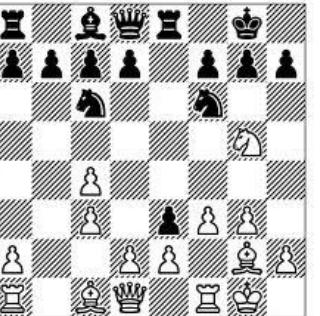


- En computación **se considera que un problema tiene solución algorítmica** si además de que el **algoritmo existe**, su **tiempo de ejecución es razonablemente corto**.



Tamaño de problema

- **Ejemplo 01:** Es posible diseñar un algoritmo para jugar ajedrez que **triunfe siempre**: *el algoritmo elige la siguiente tirada examinando todas las posibles secuencias de movimientos desde el tablero actual hasta uno donde sea claro el resultado y elige la tirada que le asegure el triunfo*; el pequeño inconveniente de esta algoritmo es que dicho espacio de búsqueda se ha estimado en 1000^{40} tableros por lo que puede tardarse años en tomar una decisión.



- Se considera que si un problema tiene una solución que **toma años en computar, dicha solución no existe.**

- **Ejemplo 02:** ordenar un conjunto de valores.

49	12	56	90	2	5	11	32	22	7	99	02	35	1
----	----	----	----	---	---	----	----	----	---	----	----	----	---

- Si el conjunto tiene 2 elementos es más fácil resolverlo que si tiene 20, análogamente un algoritmo que resuelva el problema tardará más tiempo mientras más grande sea el conjunto y requerirá una cantidad de memoria mayor para almacenar los elementos del conjunto.
- *“En general la cantidad de recursos que consume un algoritmo para resolver un problema se incrementa conforme crece el tamaño del problema”.*
- Dependiendo del problema en particular, uno o varios de sus parámetros serán elegidos como **tamaño del problema**.



- Determinar el tamaño del problema es relativamente fácil realizando un análisis del problema si el problema ya ha sido comprendido.

PROBLEMA	TAMAÑO DEL PROBLEMA
Búsqueda de un elemento en un conjunto	Número de elementos en el conjunto
Multiplicar dos matrices	Dimensión de las matrices
Recorrer un árbol binario de búsqueda	Número de nodos en el árbol
Resolver un sistema de ecuaciones lineales	Número de ecuaciones y/o incógnitas
Ordenar un conjunto de valores	Número de elementos en el conjunto
Cálculo de la sumatoria $\sum_{i=m}^n a_i$	Tamaño del intervalo (m,n)
Encontrar un elemento en una Tabla Hash Abierta	Número de elementos en la Tabla



Función 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 alguna computadora.
- La cantidad de recursos que consume un **algoritmo crece conforme el tamaño del problema** se incrementa, la **función complejidad** es **monótona creciente** $f(n) \geq f(m)$ si $n > m$ 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 se distinguen dos clases de función complejidad:

1. **Función complejidad espacial.** Mide la cantidad de memoria que necesitará un algoritmo para resolver un problema de tamaño n: $f_e(n)$.
2. **Función complejidad temporal.** 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 instrucciones de CPU que requiere el algoritmo para resolver un problema de tamaño n: $f_t(n)$.



- La **cantidad de memoria** que utiliza un algoritmo depende de la implementación, no obstante, es posible obtener una **medida del espacio** necesario con la sola inspección del algoritmo.
- Para obtener esta cantidad es necesario sumar todas las celdas de memoria que utiliza. En general se requerirán dos tipos de celdas de memoria:
 1. **Celdas estáticas.** Son las que se utilizan en todo el tiempo que dura la ejecución del programa, p.g., las variables globales.
 2. **Celdas dinámicas.** Se emplean sólo durante un momento de la ejecución, y por tanto pueden ser asignadas y devueltas conforme se ejecuta el algoritmo, p.g., el espacio de la pila utilizado por las llamadas recursivas.



- **El tiempo** que emplea un algoritmo en ejecutarse refleja la cantidad de trabajo realizado, así, la **complejidad temporal** da una medida de la cantidad de tiempo que requerirá la implementación de un algoritmo para resolver el problema, por lo que **se le puede determinar en forma experimental**.
- 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 ; se ejecuta en la máquina M y se alimenta con un conjunto de casos S . Se deberá de medir el tiempo que emplea para resolver los casos (**análisis a posteriori**).



Análisis Temporal

- Medir la complejidad temporal de manera experimental presenta, entre otros, el inconveniente de que los resultados obtenidos dependen de:
 - Las entradas proporcionadas,
 - La calidad del código generado por el compilador utilizado
 - La máquina en que se hagan las pruebas
- Cada **operación** requiere cierta **cantidad constante de tiempo** para ser ejecutada, por esta razón si se cuenta el número de operaciones realizadas por el algoritmo se obtiene una **estimación del tiempo** que le tomará resolver el problema.
- Dado un algoritmo, se puede determinar que tipos de operaciones utiliza y cuantas veces las ejecuta para una entrada específica (**análisis a priori**).

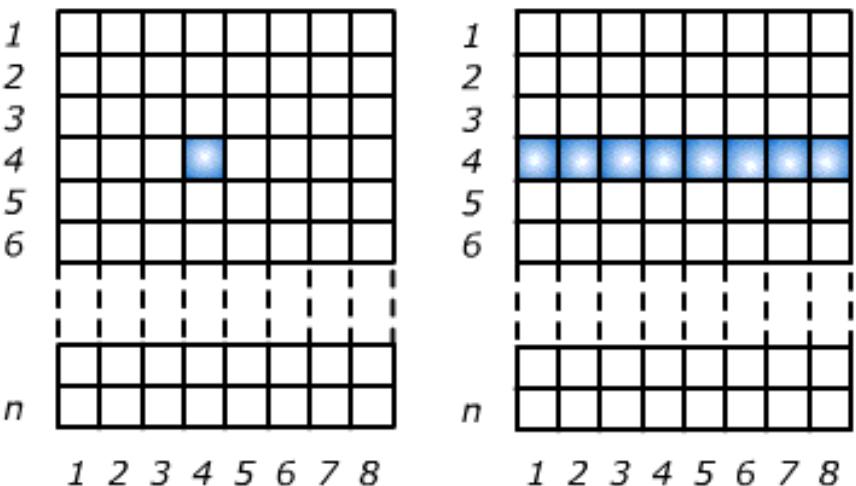


- Para evitar que **factores prácticos** se reflejen en el **cálculo de la función complejidad**, el **análisis temporal y el espacial a priori** se realiza únicamente **con base al algoritmo escrito en pseudocódigo**.
- Como el pseudocódigo no se puede ejecutar para medir la cantidad de tiempo que consume, la complejidad temporal no se expresará en unidades de tiempo, sino en términos de la **cantidad de operaciones que realiza**.

```
inserción(t)
for i=1 to n
    x= t[i]
    j=i-1
    while j > 0 and x<t[j]
        t[j+1]=t[j]
        j=j-1
    t[j+1]=x
```

Análisis Espacial

- Los casos en la función complejidad espacial, se pueden definir análogamente, considerando ahora el conjunto $C(n)$; como el conjunto formado por el número de celdas de memoria utilizadas por el algoritmo para resolver cada instancia del problema.



Medición del tiempo de ejecución

- **Medir**
 - **Cantidad de instrucciones básicas** (o elementales) que se ejecutan.
 - Ejemplos de instrucciones básicas:
 1. Asignación de variables
 2. Lectura o escritura de variables
 3. Saltos (goto's) implícitos o explícitos.
 4. Operaciones aritméticas
 5. Evaluación de condiciones
 6. Llamada a sentencias simples
 7. Llamadas y retornos de función*

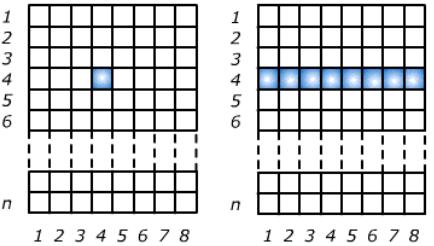


*Las funciones tienen un costo de tiempo que hay que considerar a parte.



Medición de la memoria requerida

- **Medir**
 - **Cantidad de celdas de memoria** (o elementales) que se requieren.
 - Ejemplos de celdas de memoria:
 1. Variables del algoritmo
 2. Numero de objetos instanciados requeridos
 3. Tamaño de las estructuras de datos empleadas
 4. Memoria de Entrada/Salida requerida
 5. Tamaño de arreglo, matrices u otro tipo de memoria continua estática o dinámica empleada por el algoritmo.



Ejemplo 1: Cantidad de instrucciones

cont = 1; → 1 asignación

do {

$x = x + a$ [cont]; → n asignaciones

$x = x + b$ [cont]; → n asignaciones

cont = cont + 1; → n asignaciones

}

while (*cont* <= *n*) → *n* + 1 (comparaciones) + *n* (goto implícito)

→1 (goto en falso)

TOTAL: $5n + 3$ instrucciones

Función de complejidad temporal

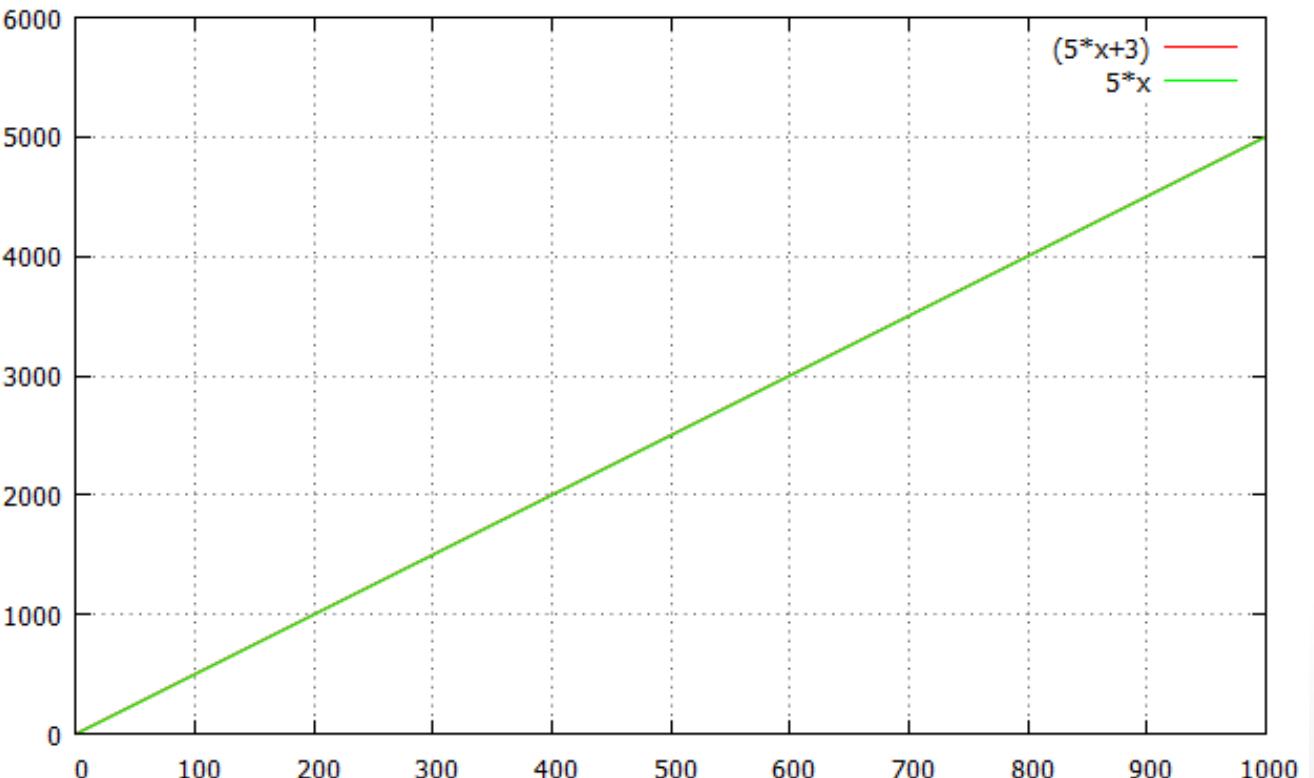
$$f(n)=5n+3$$



Función de complejidad temporal

$$f(n) = 5n + 3$$

n	Instrucciones
0	3
1	8
10	53
100	503
500	2503
1000	5003



```

cont = 1;
do {
    x = x + a[cont];
    x = x + b[cont];
    cont = cont + 1;
}
while (cont <= n)

```



Ejemplo 1: Cantidad de celdas de memoria

cont = 1; → 1 variable “cont”

do {

x = x + a[cont]; → 1 variable “x”

x = x + b[cont]; → n variables del arreglo “a”

cont = cont + 1; → n variables del arreglo “b”

}

while (cont <= n)

TOTAL: $2n + 2$

Función de complejidad espacial

$$f(n) = 2n + 2$$



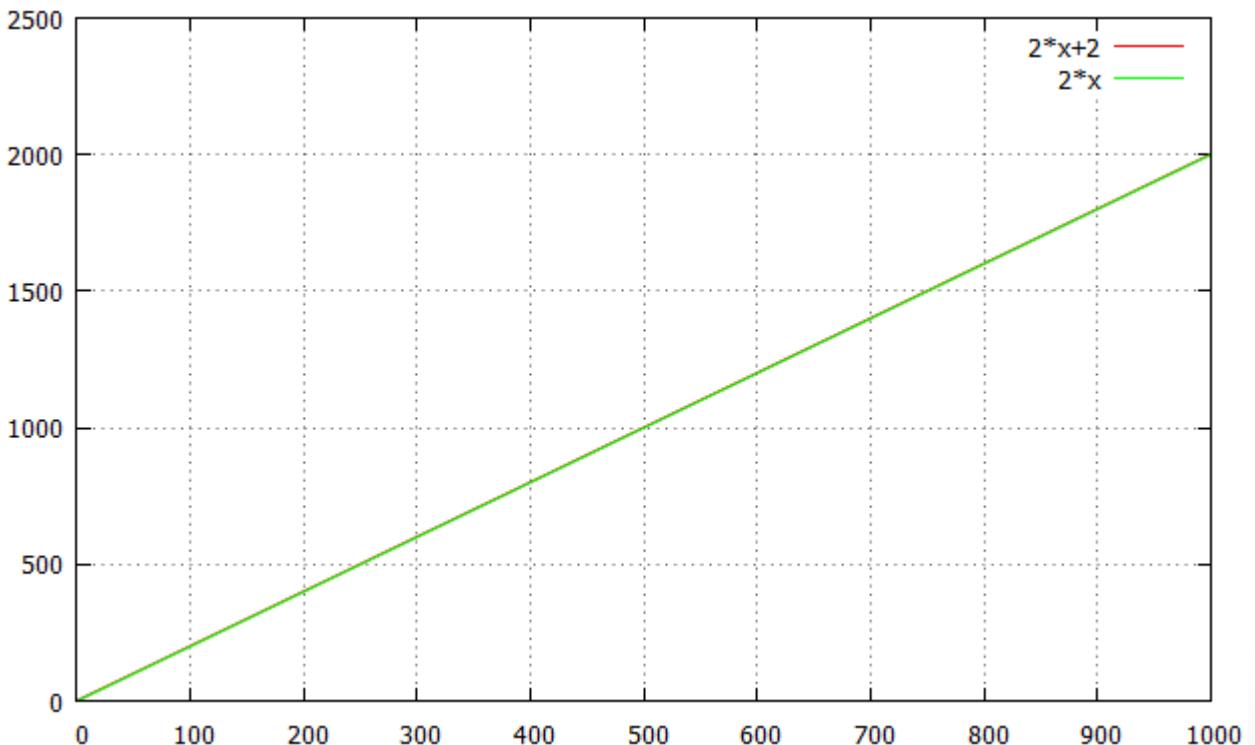
Función de complejidad espacial

$$f(n) = 2n + 2$$

n	Celdas de memoria
0	2
1	4
10	22
100	202
500	1002
1000	2002

```

cont = 1;
do {
    x = x + a[cont];
    x = x + b[cont];
    cont = cont + 1;
}
while (cont <= n)
  
```



Ejemplo 2: Cantidad de instrucciones

```

 $z = 0;$ 
for (int  $x=1$ ;  $x \leq n$ ;  $x++$ )
  for (int  $y=1$ ;  $y \leq n$ ;  $y++$ )
     $z = z + a[x,y];$ 
  
```

$\rightarrow 1$
 $\rightarrow 1 + (n+1) \text{ (asignación + comparaciones)} = 2+n$
 $\rightarrow n+(n+1)*n = n^2 + 2n$
 $\rightarrow n*n = n^2$
 $\rightarrow n^2+n^2 \text{ (incremento + goto implícito)} = 2n^2$
 $\rightarrow n \text{ (goto en falso for y)} = n$
 $\rightarrow n+n \text{ (incremento + goto implícito)} = 2n$
 $\rightarrow 1 \text{ (goto en falso for x)} = 1$

$$\rightarrow = 1 + 2 + n + n^2 + 2n + n^2 + 2n^2 + n + 2n + 1$$

TOTAL: $4n^2 + 6n + 4$

Función de complejidad temporal

$$f(n) = 4n^2 + 6n + 4$$



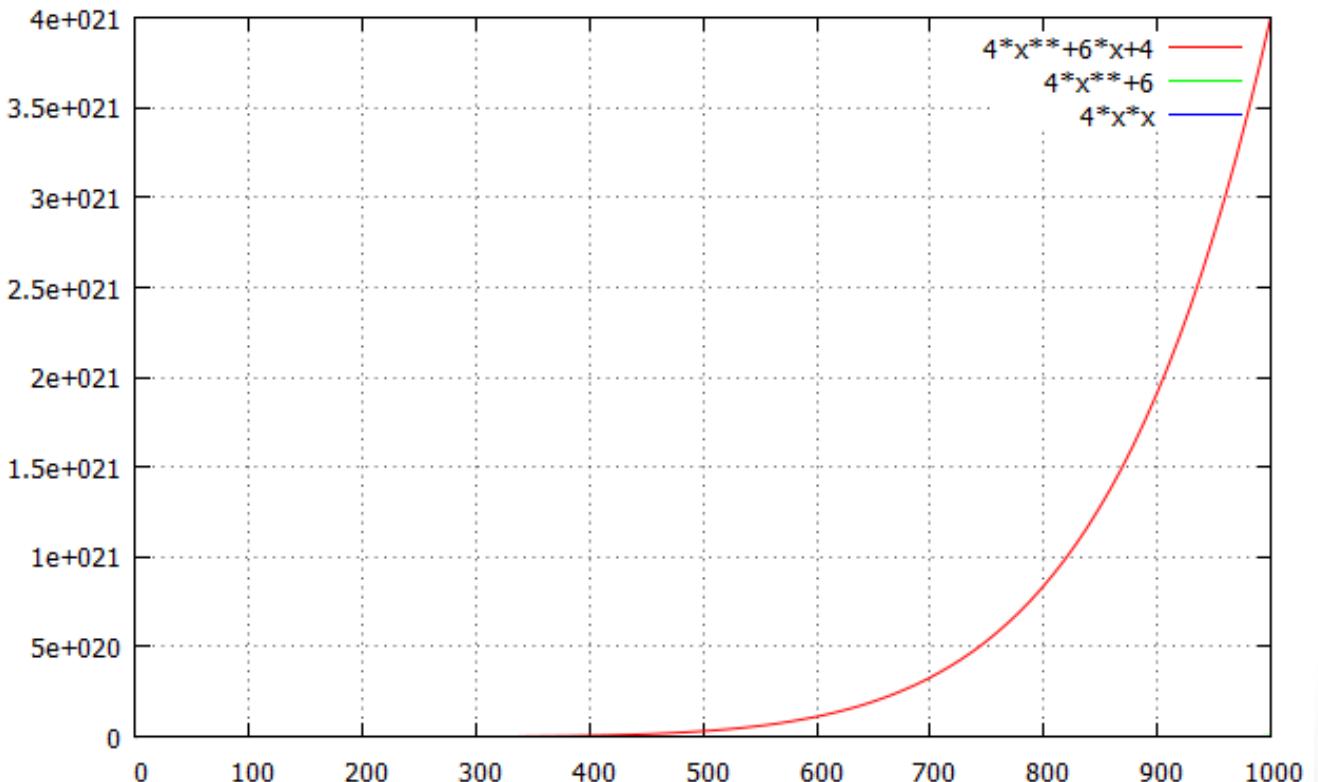
Función de complejidad temporal

$$f(n) = 4n^2 + 6n + 4$$

```

 $z = 0;$ 
 $for (int x=1; x<=n; x++)$ 
 $for (int y=1; y<=n; y++)$ 
 $z = z + a[x,y];$ 
  
```

n	Instrucciones
0	4
1	14
10	464
100	40,604
500	1,003,004
1000	4,006,004



Ejemplo 2: Cantidad de celdas de memoria

$z = 0;$

for (*int* $x=1$; $x \leq n$; $x++$)

for (*int* $y=1$; $y \leq n$; $y++$)

$z = z + a[x,y];$

→ 1 (variable “z”)

→ 1 (variable “x”)

→ 1 (variable “y”)

→ n^2 (variables de la matriz “a”)

TOTAL: $n^2 + 3$

Función de complejidad temporal

$$f(n) = n^2 + 3$$



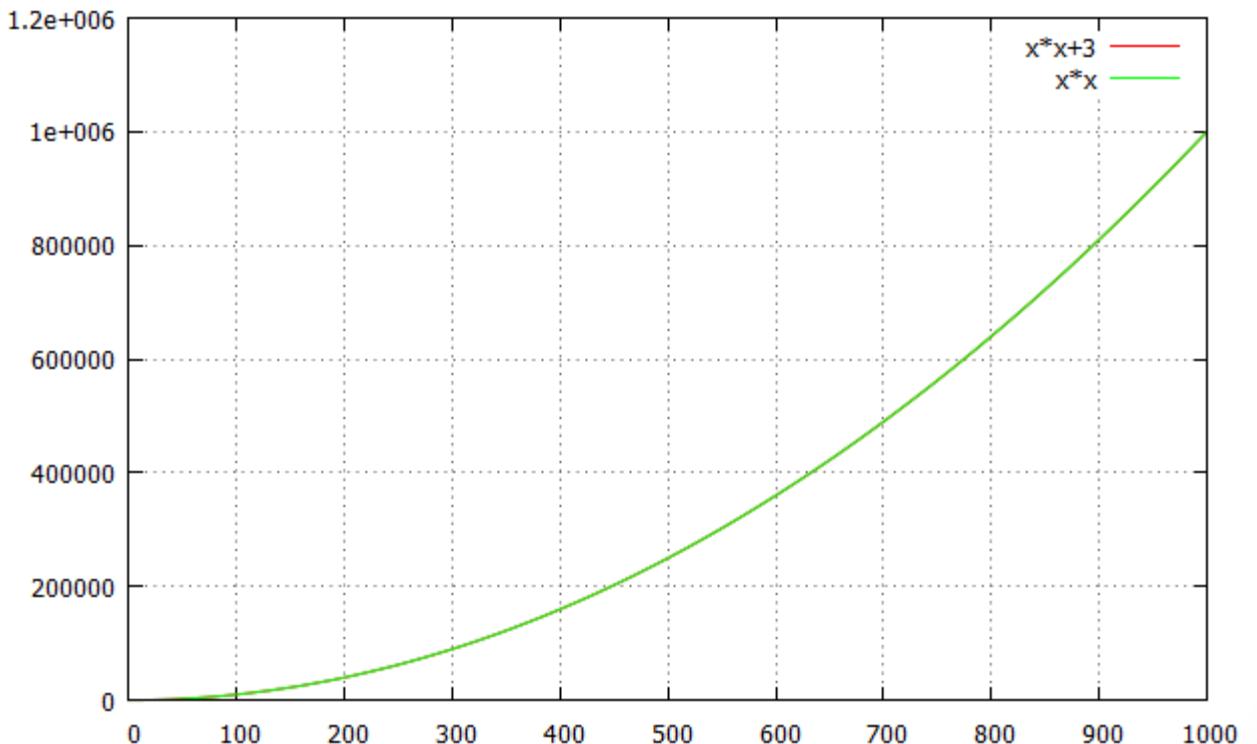
Función de complejidad espacial

$$f(n) = n^2 + 3$$

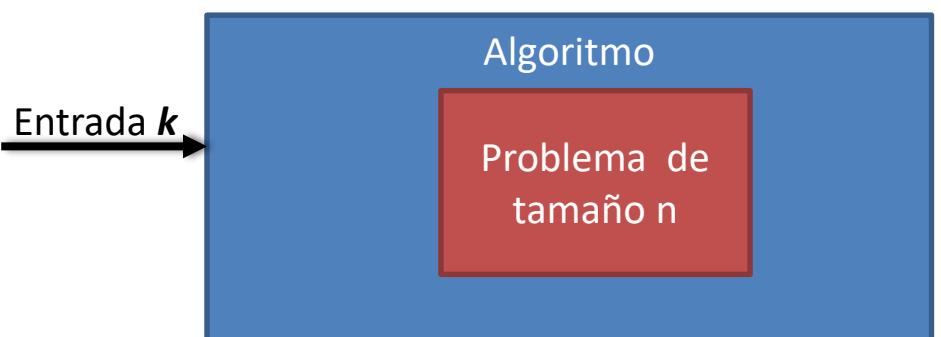
n	Celdas de memoria
0	3
1	4
10	103
100	10,003
500	250,003
1000	1,000,003

```

 $z = 0;$ 
for (int x=1; x<=n; x++)
    for (int y=1; y<=n; y++)
        z = z + a[x,y];
    
```



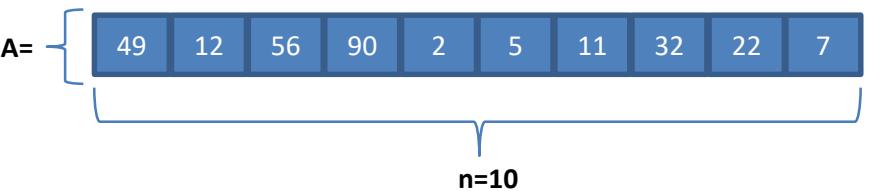
- Los algoritmos de los ejemplos 1 y 2 tienen un tamaño de problema directamente asociados a ***n i.e. solo existe un caso (instancia) del problema.***
- En la mayoría de los algoritmos también se deberá de considerar que el número de operaciones y celdas memoria dependerá de los **casos de entrada** por lo que debe de realizarse el análisis considerando cada caso ***K (instancia del problema)*** con el tamaño de problema ***n***.
 - i.e. en un algoritmo, se debe determinar que tipos de operaciones utiliza, cuantas veces las ejecuta y cuanta memoria requiere para cada entrada específica ***k***.



Ejemplo 3: Cantidad de instrucciones

```

func BusquedaLineal(Valor,A,n)
{
    i=1;
    while(i<=n&&A[i]!=Valor)
    {
        i=i+1;
    }
    return i;
}
    
```



Caso 0: Valor=11, A={49,12,56,90,2,5,11,32,22,7}, n=10
Si el Valor es 11 \Rightarrow se entra al ciclo 6 veces $\Rightarrow k=6$

Si el ciclo se ejecutará k veces (k puede tomar valor de 1 hasta n)

→ 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.



Función de complejidad temporal

$$\rightarrow f(k) = 6k + 6$$

$\rightarrow k$ =Número de veces que se entra al ciclo

k	Instrucciones
0	6
1	12
10	66
100	606
500	3006
1000	6006

func BusquedaLineal(Valor,A,n)

{

i=1;

while(i<=n&&A[i]!=Valor)

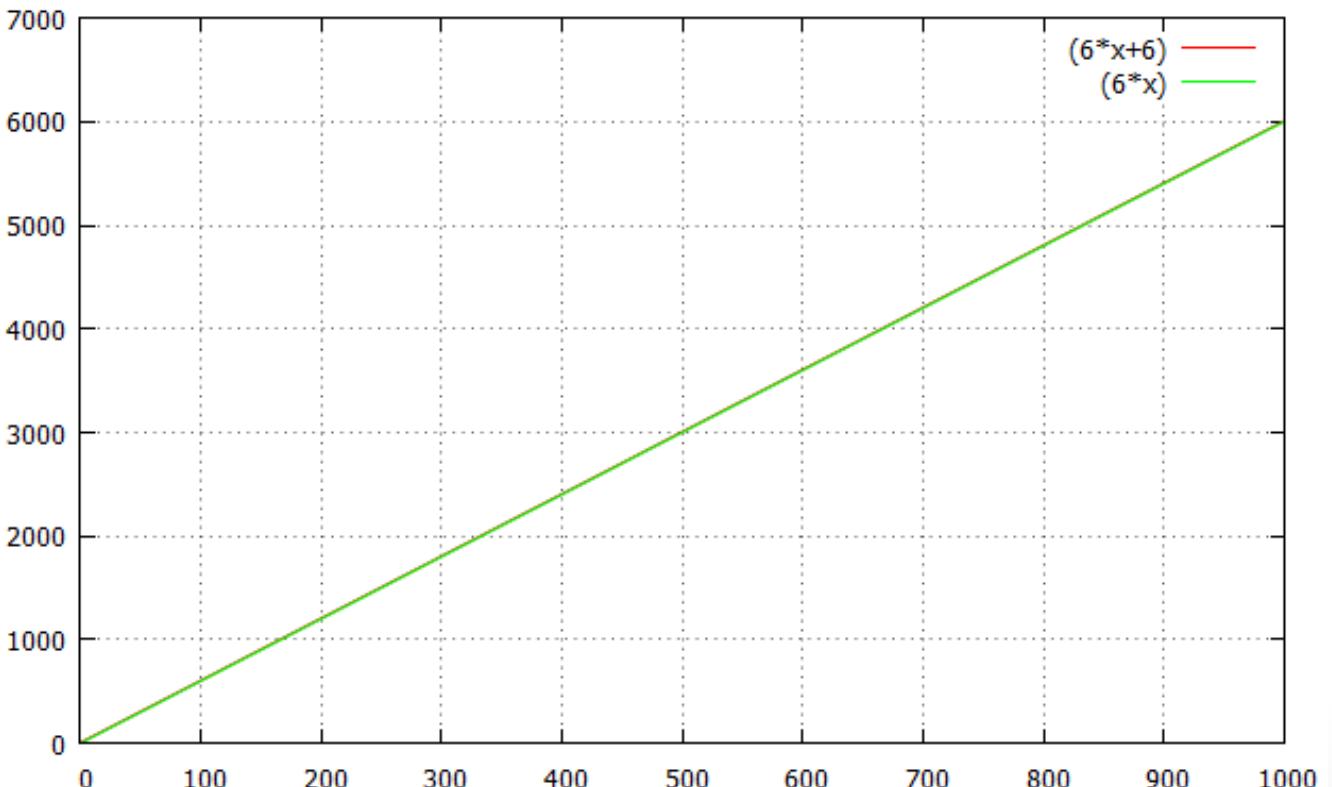
{

i=i+1;

}

return i;

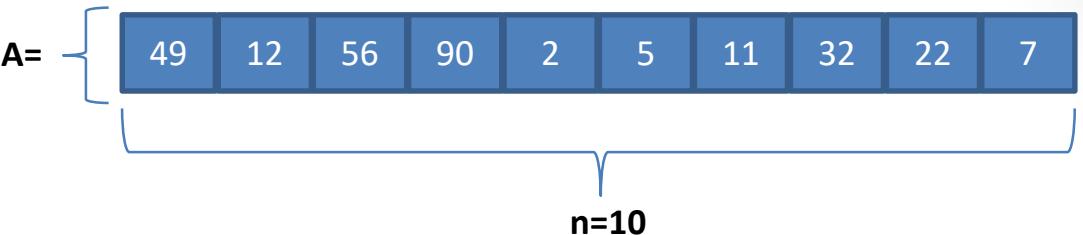
}



Ejemplo 3: Cantidad de celdas de memoria

```
func BusquedaLineal(Valor,A,n)
{
```

```
    i=1;
    while(i<=n&&A[i]!=Valor)
    {
        i=i+1;
    }
    return i;
}
```



Caso 0: Valor=11, A={49,12,56,90,2,5,11,32,22,7}, n=10

Si el Valor=11 \Rightarrow se requieren 10 + 3 variables

Caso 1: Valor=12, A={49,12,56,90,2,5,11,32,22,7}, n=10

Si el Valor=12 \Rightarrow se requieren 10 + 3 variables

Si se analizaran todos los casos

→ Para este algoritmo la complejidad espacial no varia

→ **n + 3 celdas de memoria en total.**

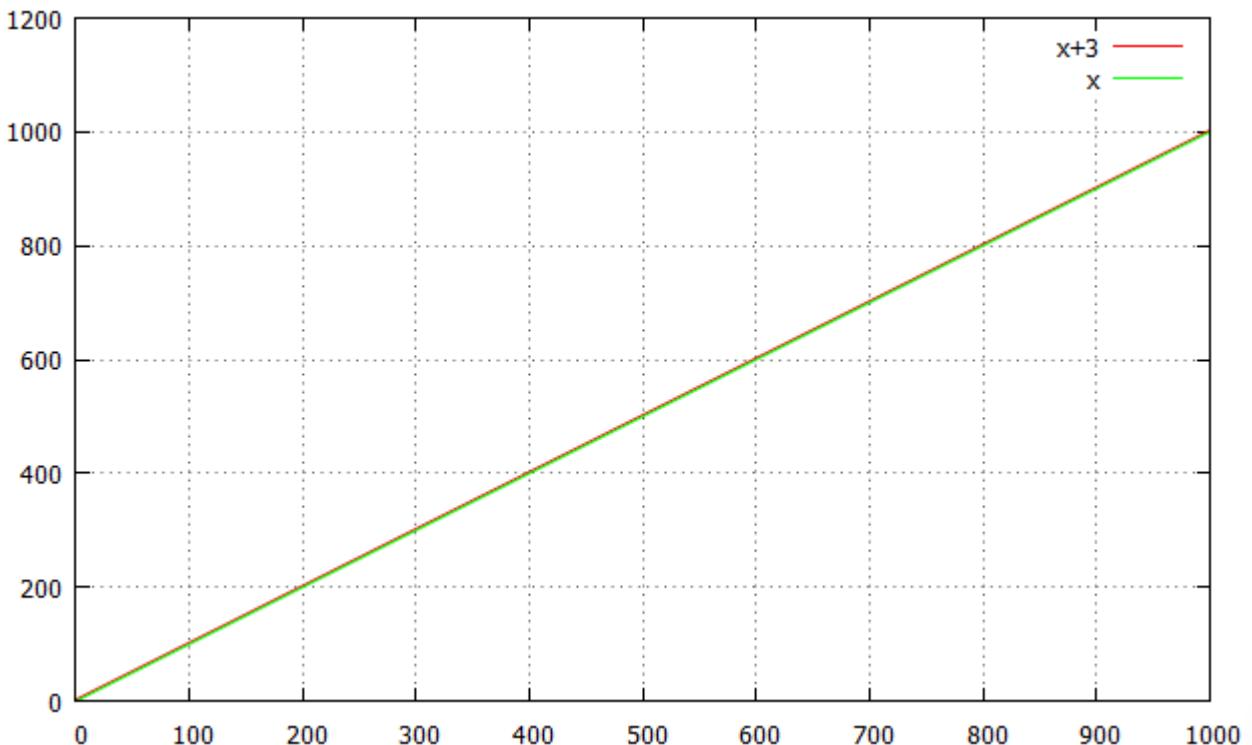


Función de complejidad espacial constante para todos los casos

$$\rightarrow f(n) = n + 3$$

k	Celdas de memoria
0	3
1	4
10	13
100	103
500	503
1000	1003

```
func BusquedaLineal(Valor,A,n)
{
    i=1;
    while(i<=n && A[i]!=Valor)
    {
        i=i+1;
    }
    return i;
}
```





Instituto Politécnico Nacional

Escuela Superior de Cómputo



Análisis de algoritmos

Tema 03: Análisis temporal

M. en C. Edgardo Adrián Franco Martínez
<http://www.eafranco.com>
edfrancom@ipn.mx
[@edfrancom](https://twitter.com/edfrancom) [@edgardoадrianfranco](https://facebook.com/edgardo.adrian.franco)



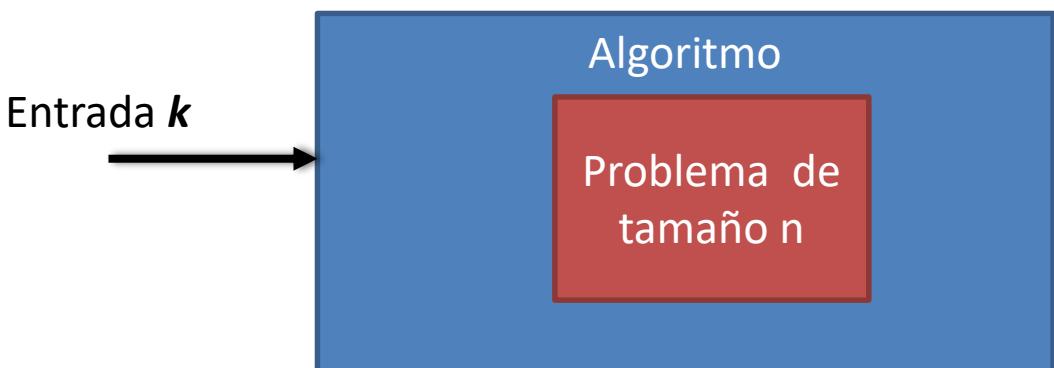
Contenido

- Caso de entrada
 - Ejemplo 1 (Búsqueda lineal)
- Operación básica
 - Ejemplo 2 (Producto de 2 mayores)
- Elección de la operación básica
- Concepto de Instancia
- Análisis Peor Caso, Mejor Caso y Caso Promedio
- Análisis Temporal (mejor, peor y caso medio)
 - Retomando el Ejemplo 1 (Búsqueda lineal)
 - Retomando el Ejemplo 2 (Producto de 2 mayores)



Caso de entrada

- Un **caso de entrada** para un algoritmo **es una instancia** del problema inicial.
- Un algoritmo para resolver un problema de tamaño n , puede recibir una o más entradas, las cuales pueden definir el número de operaciones que hará.
 - Se puede determinar que tipos de operaciones utiliza y cuantas veces las ejecuta para una entrada específica k .
 - Por ejemplo realizar una búsqueda lineal del 40 en un arreglo de 100,000 elementos ($n=100,000$).



Ejemplo 1 (Búsqueda lineal)

```
func BusquedaLineal(Valor,A,n)
```

```
{
```

```
i=1;
```

```
while(i<=n&&A[i]!=Valor)
```

```
{
```

```
i=i+1;
```

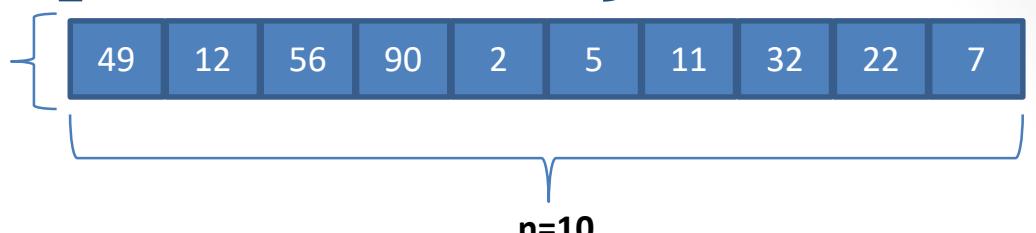
```
}
```

```
return i;
```

Caso 0: Valor=11, A={49,12,56,90,2,5,11,32,22,7}, n=10

Si el Valor=11 \Rightarrow se entra al ciclo 6 veces $\Rightarrow k=6$

```
}
```



Si el ciclo se ejecutará k veces

→ 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.



Operación básica

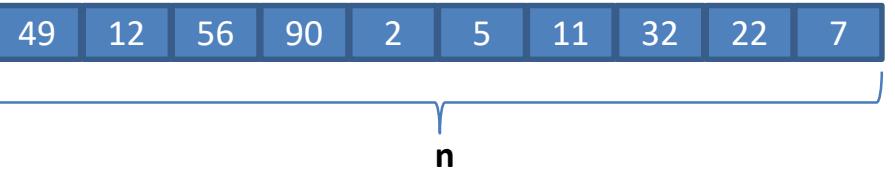
- Del ejemplo notamos que el número de veces que se ejecutan algunas operaciones, para valores sucesivamente mayores que k y/o n ; se presenta un modelo de crecimiento similar al que tiene el número total de operaciones que ejecuta.
- Para hacer una estimación de la cantidad de tiempo que tarda un algoritmo 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.



- 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).
- Pg. la operación básica en el Algoritmo 3 (búsqueda lineal) es la **comparación entre los elementos del arreglo y el valor buscado**.

```

func BusquedaLineal(Valor,A,n)
{
    i=1;
    while(i<=n&&A[i]!=Valor)
    {
        i=i+1;
    }
    return i;
  
```



→**Operación básica “Comparación A[i]!=Valor”**



Ejemplo 2 (Producto de 2 mayores)

- El algoritmo siguiente obtiene el producto de los dos valores más grandes contenidos en un arreglo A de n enteros

```

func Producto2Mayores(A,n)
  if(A[1] > A[2])
    mayor1 = A[1];
    mayor2 = A[2];
  else
    mayor1 = A[2];
    mayor2 = A[1];
  i = 3;

  while(i<=n)
    if(A[i] > mayor1)
      mayor2 = mayor1;
      mayor1 = A[i];
    else if (A[i] > mayor2)
      mayor2 = A[i];
    i = i + 1;

  return = mayor1 * mayor2;

```



- En este algoritmo se realizan las siguientes operaciones:
 - a) Comparación con los elementos del arreglo
 - b) Asignaciones a mayor1 y mayor2
 - c) Asignación al índice i
 - d) Asignación a la función
 - e) Producto de los mayores
 - f) Incremento al índice i
 - g) Comparación entre el índice i y la longitud del arreglo n
- Las operaciones (c), (f) y (g) no se consideran por realizarse entre índices, las operaciones (d) y (e) se ejecutan una sola vez y no son proporcionales al número total de operaciones.
- Entonces, se tiene que las **operaciones que se pueden considerar para hacer el análisis** son: (a) Comparación con los elementos del arreglo y (b) las asignaciones a los elementos mayores.



- (c), (f), (g), (d) y (e) pueden omitirse para el análisis
- (a) y (b) operaciones básicas del algoritmo

```

func Producto2Mayores(A,n)
    if(A[1] > A[2])                                →(d)
        mayor1 = A[1];
        mayor2 = A[2];
    else
        mayor1 = A[2];
        mayor2 = A[1];
    i = 3;
    while(i<=n)                                     →(g)
        if(A[i] > mayor1)                           →(a)
            mayor2 = mayor1;
            mayor1 = A[i];
        else if(A[i] > mayor2)                     →(a)
            mayor2 = A[i];
        i = i + 1;                                  →(f)
    return = mayor1 * mayor2;                         →(e)

```



Elección de la operación básica

- El análisis de un algoritmo se puede hacer considerando sólo aquella operación que cumpla los siguientes **criterios**:
 - a) Debe estar **relacionada** con el tipo de **problema** que se resuelve.
 - b) Debe **ejecutarse** un número de veces cuyo **modelo de crecimiento sea similar al del número total de operaciones** que efectúa el algoritmo.
- Si ninguna de las operaciones encontradas cumple con ambos criterios, es posible declinar por el primero. Si aun así no es posible encontrar una operación representativa, se debe hacer un análisis global, contando todas las operaciones.



- Elección de la operación básica para algunos problemas:

- Búsqueda de un elemento en un conjunto
 - Comparación entre el valor y los elementos del conjunto
- Multiplicar dos matrices
 - Producto de los elementos de las matrices
- Recorrer un árbol
 - Visitar un nodo
- Resolver un sistema de ecuaciones lineales
 - Suma y resta de las ecuaciones
- Ordenar un conjunto de valores
 - Comparación entre valores



Concepto de Instancia

- Un **problema computacional** consiste en una caracterización de un conjunto de datos de entrada, junto con la especificación de la salida deseada con base a cada entrada.
- Un problema computacional tiene una o más **instancias**, que son **valores particulares para los datos de entrada**, sobre los cuales se puede ejecutar el algoritmo para resolver el problema; i.e. un caso específico de un problema.
- **Ejemplo:** el problema computacional de *multiplicar dos números enteros* tiene, las siguientes instancias: multiplicar 345 por 4653, multiplicar 2637 por 10000, multiplicar -32341 por 12, etc.



Análisis Peor Caso, Mejor Caso y Caso Promedio

- El comportamiento de un algoritmo puede variar notablemente para diferentes instancias.
- Suelen estudiarse tres casos para un mismo algoritmo: caso mejor, caso peor, caso medio.
- **Tipos de análisis:**
 - **Peor caso:** indica el mayor tiempo obtenido, teniendo en consideración todas las entradas posibles.
 - **Mejor caso:** indica el menor tiempo obtenido, teniendo en consideración todas las entradas posibles.
 - **Caso medio:** indica el tiempo medio obtenido, considerando todas las entradas posibles.



Retomando el ejemplo 01: Búsqueda lineal

49	12	56	90	2	5	11	32	22	7	99	02	35	1
----	----	----	----	---	---	----	----	----	---	----	----	----	---

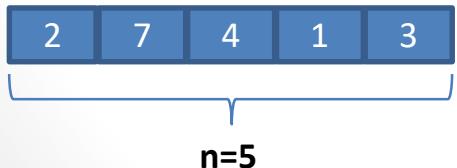
- **Problema:** Encontrar la posición de un determinado número en un arreglo desordenado.
- ¿Cuales serían el peor caso, mejor caso y caso promedio?

**Conclusión: Si se conociera la distribución de los datos, podemos sacar provecho de esto, para un mejor análisis y diseño del algoritmo. Por otra parte, sino conocemos la distribución, entonces lo mejor es considerar el peor de los casos.*



Análisis Temporal (mejor, peor y caso medio)

- Con los conceptos anteriores es posible llevar a cabo el análisis temporal de un algoritmo i.e. calcular la **función complejidad temporal** $f_t(n)$.
- Considérese el Algoritmo del ejemplo 3 (Búsqueda lineal), para hacer el análisis de su comportamiento:
 - **Operación básica:** Comparaciones con elementos del arreglo
 - **Caso muestra:** $A = [2, 7, 4, 1, 3]$ y $n = 5$:
 - Si **Valor = 2**, se hace una comparación, $f_t(5) = 1$
 - Si **Valor = 4**, se hacen tres comparaciones, $f_t(5) = 3$
 - Si **Valor = 8**, se hacen cinco comparaciones, y $f_t(5) = 5$

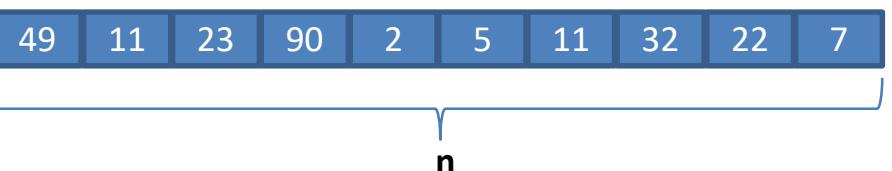


```
func BusquedaLineal(Valor,A,n)
{
    i=1;
    while(i<=n && A[i]!=Valor)
    {
        i=i+1;
    }
    return i;
}
```

- Del análisis anterior es posible descubrir que la **función complejidad temporal** no es tal, en realidad es una relación, ya que **para un mismo tamaño de problema se obtienen distintos valores de la función complejidad.**
- Para la mayoría de los algoritmos el número de operaciones depende, no sólo del tamaño del problema, sino también de la **instancia específica que se presente (caso de entrada)**.

```

func BusquedaLineal(Valor,A,n)
{
  i=1;
  while(i<=n&&A[i]!=Valor)
  {
    i=i+1;
  }
  return i;
}
  
```



- Caso a: 1 comparación (Valor = 49)
- Caso b: 2 comparaciones (Valor = 11)
- Caso c: 3 comparaciones (Valor = 23)
- ...
- Caso x: n+1 comparaciones. (Valor = 8)



Sea:

- $I(n)=\{I_1, I_2, I_3, \dots, I_k\}$ el conjunto de instancias del problema de tamaño n .
- $O(n)=\{O_1, O_2, O_3, \dots, O_k\}$ el conjunto formado por el número de operaciones que un algoritmo realiza para resolver cada instancia.
- Entonces, O_j es el número de operaciones ejecutadas para resolver la instancia I_j , para $1 \leq j \leq k$.
- Se distinguen **tres casos en el valor de la función complejidad temporal**

Peor caso $f_t(n) = \max (\{O_1, O_2, O_3, \dots, O_k\})$

Mejor caso $f_t(n) = \min (\{O_1, O_2, O_3, \dots, O_k\})$

Caso medio $f_t(n) = \sum_{i=1}^k O_i P(i)$

$P(i)$ es la probabilidad de que ocurra la instancia I_i



- **El mejor caso** se presenta cuando para un caso de entrada I_1 a un problema de tamaño n ; el algoritmo ejecuta el **mínimo número posible de operaciones**.
- **El peor caso** se presenta cuando para un caso de entrada I_2 a un problema de tamaño n ; el algoritmo ejecuta el **máximo número de operaciones**.
- **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 I_3 .

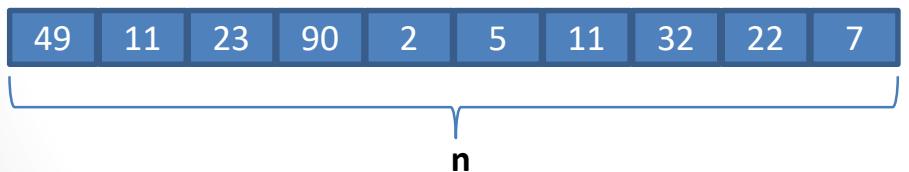


Retomando el Ejemplo 1 (Búsqueda lineal)

- **Algoritmo:** Búsqueda lineal
- **Problema:** Búsqueda lineal de un valor en un arreglo de tamaño n :
- **Tamaño del Problema:** n = número de elementos en el arreglo.
- **Operación básica:** Comparación del valor con los elementos del arreglo $A[i] \neq Valor$.

→ Mejor caso 1 comparación (Ejemplo Buscar: Valor = 49)

→ Peor caso $n+1$ comparaciones. (Ejemplo Buscar: Valor = 8)



```
func BusquedaLineal(Valor,A,n)
{
    i=1;
    while(i<=n&&A[i]!=Valor)
    {
        i=i+1;
    }
    return i;
}
```



• Análisis Temporal

- **Mejor caso:** ocurre cuando el valor es el primer elemento del arreglo.

$$f_t(n) = 1 \text{ (Una comparación } A[i] \neq \text{Valor})$$

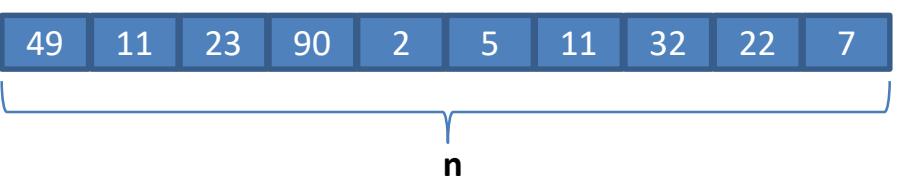
- **Peor caso:** sucede cuando el valor es el ultimo en el arreglo o no se encuentra en el arreglo.

$$f_t(n) = n \text{ (n comparaciones } A[i] \neq \text{Valor})$$

- **Caso medio:**

$$f_t(n) = 1P(1) + 2P(2) + 3P(3) + 4P(4) + \dots + nP(n) + nP(n + 1)$$

- Donde $P(i)$ es la probabilidad de que el valor se encuentre en la localidad i ; ($1 \leq i \leq n$) y $P(n + 1)$ es la probabilidad de que no esté en el arreglo y el número que lo acompaña es la cantidad de operaciones básicas para cada uno de ellos.



- Si se supone que todos los casos son igualmente probables:

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

Observación

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

- **Caso medio:**

Número de operaciones básicas por caso

$$f_t(n) = 1P(1) + 2P(2) + 3P(3) + 4P(4) + \dots + nP(n) + (n)P(n+1) .$$

$$f_t(n) = \frac{1}{n+1} \left(\sum_{i=1}^n i + n \right)$$

$$f_t(n) = \frac{1}{n+1} \left(\frac{n(n+1)}{2} + n \right) = \frac{n}{2} + \frac{n}{n+1} = \frac{n^2+n+2n}{2(n+1)} = \frac{n^2+3n}{2(n+1)} = \frac{n(3+n)}{2(n+1)}$$

$$f_t(n) = \frac{n(3+n)}{2(n+1)}$$

Comparaciones si la probabilidad de todos los casos es la misma



Retomando el Ejemplo 2 (Productos Mayores)

- **Algoritmo:** Producto mayores.
- **Problema:** Dado un arreglo de valores, encontrar el producto de los dos números mayores.
- **Tamaño del Problema:** n =número de elementos en el arreglo.
- **Operación básica:** Comparación con los elementos del arreglo y las asignaciones a los elementos mayores.

```

func Producto2Mayores(A,n)
    if(A[1] > A[2])
        mayor1 = A[1];
        mayor2 = A[2];
    else
        mayor1 = A[2];
        mayor2 = A[1];
    i = 3;

    while(i<=n)
        if(A[i] > mayor1)
            mayor2 = mayor1;
            mayor1 = A[i];
        else if (A[i] > mayor2)
            mayor2 = A[i];
        i = i + 1;

    return = mayor1 * mayor2;

```



func Producto2Mayores(A,n)

```

if(A[1] > A[2])
    mayor1 = A[1];
    mayor2 = A[2];
else
    mayor1 = A[2];
    mayor2 = A[1];
i = 3;
  
```

→ Primer comparación

→ Asignacion

→ Asignacion

→ Asignacion

→ Asignacion

while(*i*<=n)

```

if(A[i] > mayor1)
    mayor2 = mayor1;
    mayor1 = A[i];
else if (A[i] > mayor2)
    mayor2 = A[i];
i = i + 1;
  
```

→ n-2 Comparaciones

→ Asignacion

→ Asignacion

→ *Si A[i]≤mayor2

→ Asignacion

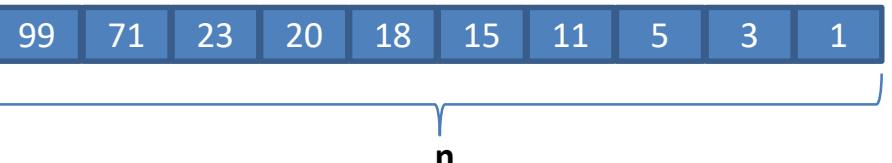
return = mayor1 * mayor2;



• Análisis Temporal

- **Mejor caso:** ocurre cuando el arreglo está ordenado descendente (*se realiza la primer comparación y dos asignaciones, posteriormente solo se compara n-2 veces el if y n-2 veces el else if*).

Comparaciones	Asignaciones
$1+(n-2)+(n-2)$	2



$$f_t(n) = 1 + (n-2) + (n-2) + 2 = 3 + 2(n-2) = 2n - 1$$

```

func Producto2Mayores(A,n)
    if(A[1] > A[2])
        mayor1 = A[1];
        mayor2 = A[2];
    else
        mayor1 = A[2];
        mayor2 = A[1];
    i = 3;

    while(i<=n)
        if(A[i] > mayor1)
            mayor2 = mayor1;
            mayor1 = A[i];
        else if (A[i] > mayor2)
            mayor2 = A[i];
        i = i + 1;

    return mayor1 * mayor2;
  
```

$$f_t(2) = 2(2) - 1 = 3$$

$$f_t(3) = 2(3) - 1 = 5$$

$$f_t(5) = 2(5) - 1 = 9$$

$$f_t(10) = 2(10) - 1 = 19$$

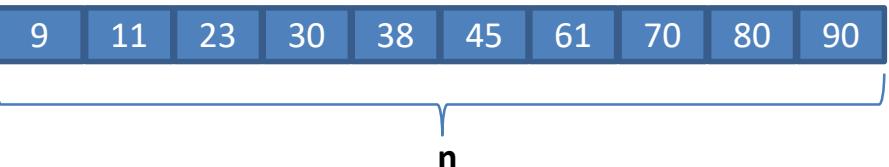
$$f_t(20) = 2(20) - 1 = 39$$



- **Peor caso:** el arreglo está ordenado de manera ascendente (*se realiza la primer comparación y dos asignaciones, posteriormente solo se compara $n-2$ veces el if y siempre se cumplirá por lo que hará $2(n-2)$ asignaciones.*)

$$f_t(n) = 1 + (n-2) + 2(n-2) + 2 = 3 + 3(n-2) = 3n - 3$$

Comparaciones	Asignaciones
$1+(n-2)$	$2+2(n-2)$



```

func Producto2Mayores(A,n)
    if(A[1] > A[2])
        mayor1 = A[1];
        mayor2 = A[2];
    else
        mayor1 = A[2];
        mayor2 = A[1];
    i = 3;

    while(i <= n)
        if(A[i] > mayor1)
            mayor2 = mayor1;
            mayor1 = A[i];
        else if(A[i] > mayor2)
            mayor2 = A[i];
        i = i + 1;

    return = mayor1 * mayor2;
  
```

$$f_t(2) = 3(2) - 3 = 3$$

$$f_t(3) = 3(3) - 3 = 6$$

$$f_t(5) = 3(5) - 3 = 12$$

$$f_t(10) = 3(10) - 3 = 27$$

$$f_t(20) = 3(20) - 3 = 57$$

- **Caso medio:** en este problema se tienen $\binom{|U|}{n} n!$ casos, donde U es el conjunto del que se extraen los elementos del arreglo.

- $\binom{|U|}{n}$ Determina el número de posibles conjuntos de n elementos del conjunto U .
 - $n!$ Determina el número de maneras de acomodar los n elementos
 - Para hacer el cálculo se deben de contar las operaciones que se harían en cada caso. (Laborioso y complicado)
-
- El algoritmo hace siempre **una comparación al inicio y dos asignaciones** y en el interior del ciclo puede ser que se realice **una comparación con dos asignaciones, dos comparaciones con una asignación o dos comparaciones y ninguna asignación**; obsérvese que para cada $A[i]$ puede ser cierta una de tres aseveraciones:
 - $A[i] > mayor1$: Se hace una comparación y dos asignaciones
 - $A[i] \leq mayor1 \quad \&\& \quad A[i] > mayor2$: Se hacen dos comparaciones y una asignación
 - $A[i] \leq mayor1 \quad \&\& \quad A[i] \leq mayor2$: Se hacen dos comparaciones



- **Caso medio** Si cada caso tiene la misma probabilidad de ocurrencia en promedio se harán:

- **Caso: $A[i] > \text{mayor1}$**

$$\bullet \frac{1}{3} (1 + 2 + (n - 2) + 2(n - 2)) = \frac{1}{3} (3 + 3(n - 2)) = \frac{1}{3} (3n - 3)$$

- **Caso: $A[i] \leq \text{mayor1} \ \&\& A[i] > \text{mayor2}$**

$$\bullet \frac{1}{3} (1 + 2 + 2(n - 2) + (n - 2)) = \frac{1}{3} (3 + 3(n - 2)) = \frac{1}{3} (3n - 3)$$

- **Caso: $A[i] \leq \text{mayor1} \ \&\& A[i] \leq \text{mayor2}$**

$$\bullet \frac{1}{3} (1 + 2 + (n - 2) + (n - 2)) = \frac{1}{3} (3 + 2(n - 2)) = \frac{1}{3} (2n - 1)$$

- **Caso medio:**

$$f_t(n) = \frac{1}{3} (2(3n - 3) + (2n - 1)) = \frac{1}{3} (8n - 7)$$

$$f_t(n) = \frac{8n - 7}{3} \text{ operaciones básicas}$$

$f_t(2)=3$
$f_t(3)=5.6$
$f_t(5)=11$
$f_t(10)=24.3$
$f_t(20)=51$



- Es importante mencionar que no todos los algoritmos presentan casos que hacen variar la complejidad temporal para un tamaño de problema específico, y resulta interesante tener una herramienta de análisis para detectar cuándo se particionará el análisis en casos; por el momento la única ayuda con la que se cuenta es la intuición y preguntarse: **¿Se puede resolver el problema de manera trivial para alguna instancia específica?**. Si la respuesta es afirmativa el algoritmo tendrá casos, por lo que el problema de detección se reduce a contestar esta “simple” pregunta.





Instituto Politécnico Nacional

Escuela Superior de Cómputo



Análisis de algoritmos

Tema 04: Notación asintótica

M. en C. Edgardo Adrián Franco Martínez
<http://www.eafranco.com>
edfrancom@ipn.mx
[@edfrancom](https://twitter.com/edfrancom) [@edgardoадrianfranco](https://facebook.com/edgardo.adrian.franco)



Contenido

- Introducción
- Asíntota
- Dominio asintótico
 - Ejemplo 1
 - Ejemplo 2
- Dominio asintótico a la función complejidad
- Notación de orden (Cotas asintóticas)
 - Cota Superior: Notación O mayúscula
 - Cota Superior no ajustada: Notación o minúscula
 - Diferencia entre O y o
 - Cota Inferior: Notación Ω
 - Cota ajustada asintótica: Notación Θ
- Observaciones sobre las cotas asintóticas
- Ordenes de complejidad (Cota superior)



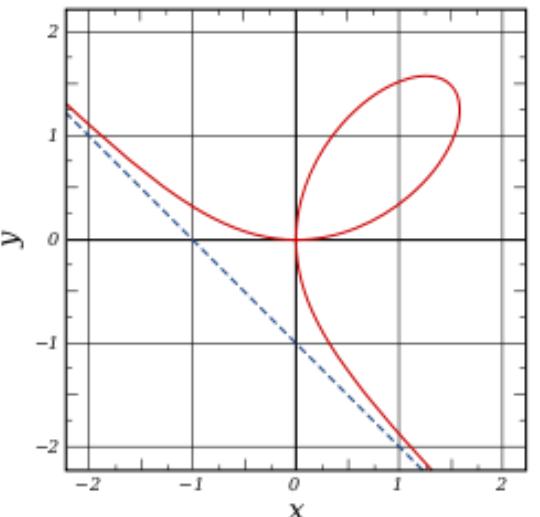
Introducción

- El **costo para obtener una solución** de un problema concreto **aumenta con el tamaño n** del problema.
- Para valores suficientemente pequeños de n , el coste de ejecución de cualquier algoritmo es pequeño, incluyendo los algoritmos ineficientes; i.e. la elección del algoritmo no es un problema crítico para problemas de pequeño tamaño.
- **El análisis de algoritmos se realiza para valores grandes de n .** Para lo cual se considera el comportamiento de sus funciones de complejidad para valores grandes de n , es decir se estudia el **comportamiento asintótico** de $f(n)$, lo cuál permite conocer el comportamiento en el límite del coste cuando n crece.



Asíntota

- Se le llama **asíntota** a una línea recta que se aproxima continuamente a otra función o curva; es decir que la distancia entre las dos tiende a cero, a medida que se extienden indefinidamente.
- También se puede decir que es la curva la que se aproxima continuamente a la recta; o que ambas presentan un **comportamiento asintótico**.



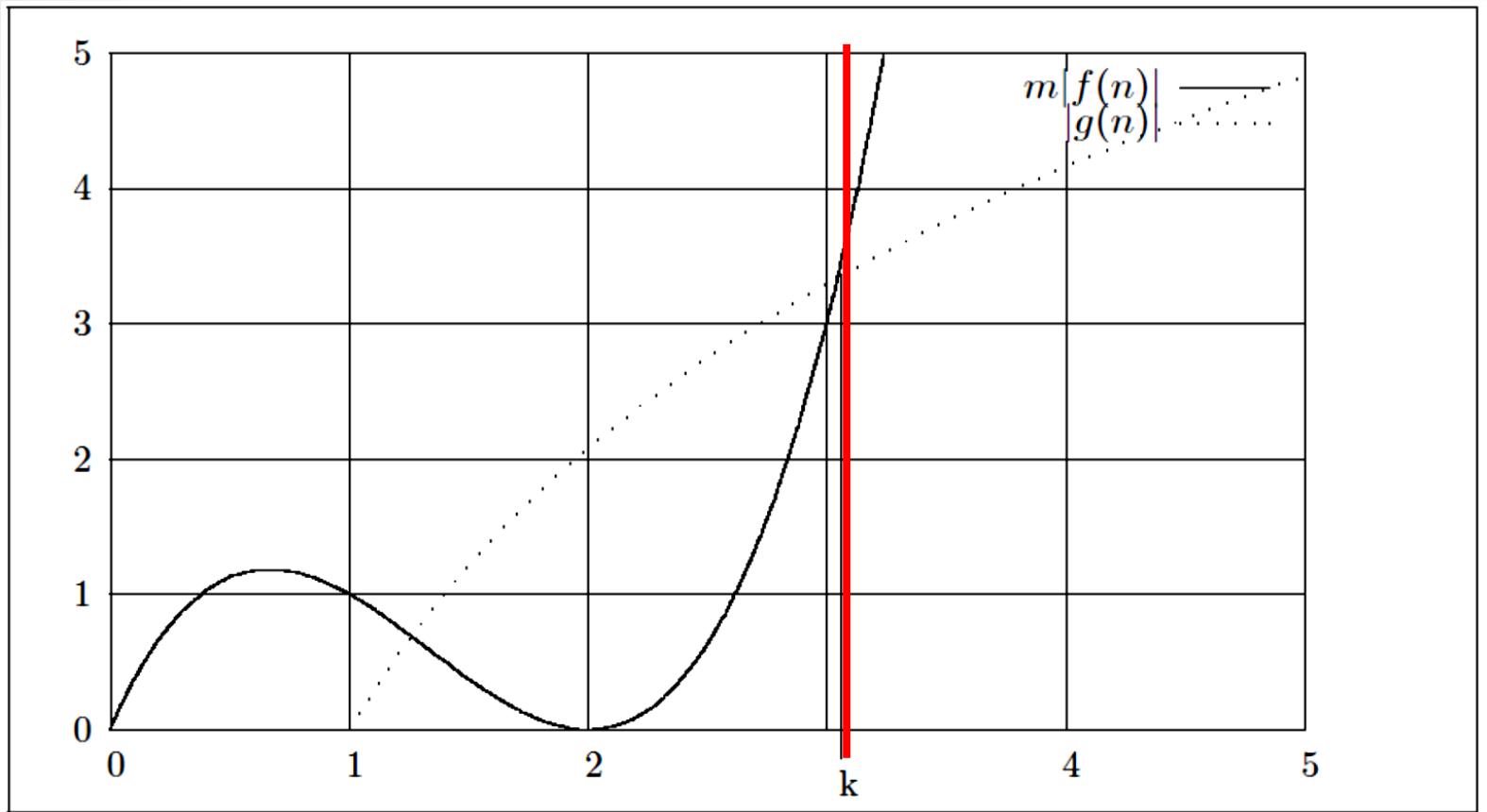
Dominio asintótico

- Sean f y g funciones de \mathbb{N} a \mathbb{R} . Se dice que f domina asintóticamente a g o que g es dominada asintóticamente por f ; si $\exists k \geq 0$ y $m \geq 0$ tales que:

$$|g(n)| \leq m|f(n)|, \forall n \geq k$$

- En otros términos, podemos decir que si una función domina a otra, su velocidad de crecimiento es mayor o igual.
- Puesto que las **funciones complejidad** son funciones con dominio \mathbb{N} (números naturales), y contradominio (\mathbb{R})números reales; los conceptos y las propiedades de **dominio asintótico** proporcionan una manera conveniente de expresarlas y manipularlas.





- Gráficas de las funciones $m|f(n)|$ y $|g(n)|$, donde k es el valor a partir del cual $m|f(n)|$ es mayor que $|g(n)|$ y esta relación de magnitud se conserva conforme n crece.



Dominio asintótico (Ejemplo 1)

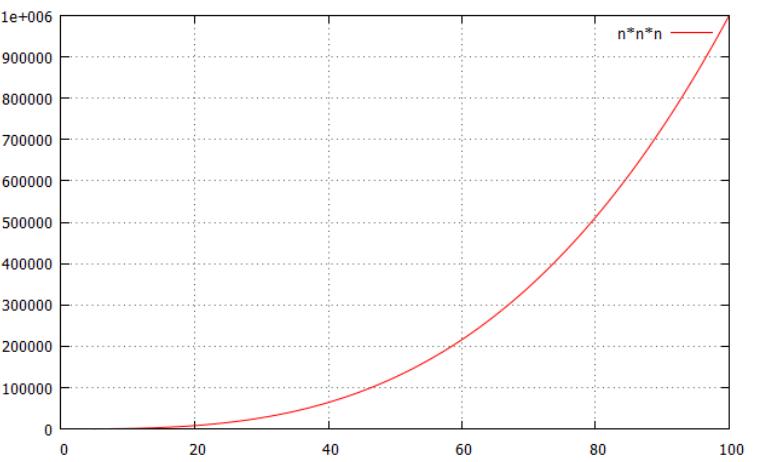
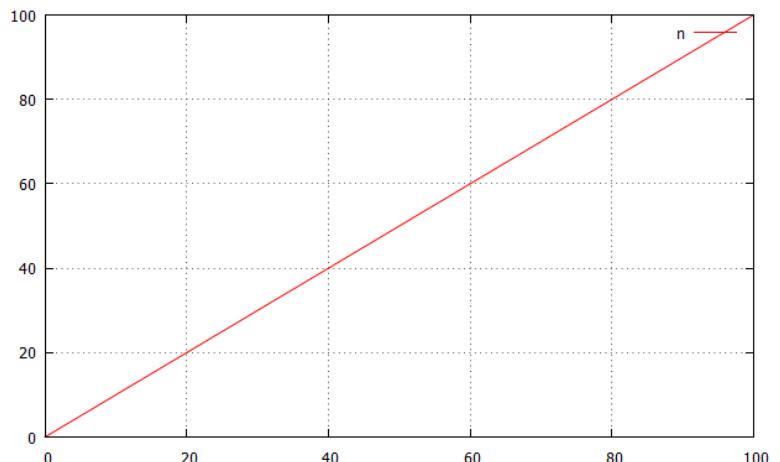
- **Ejemplo 1:** Sean $f(n) = n$ y $g(n) = n^3$ funciones de \mathbb{N} a \mathbb{R} .

i. Demostrar que g domina asintóticamente a f

$$\exists m \geq 0, k \geq 0 \text{ tales que } |f(n)| \leq m|g(n)|, \forall n \geq k$$

ii. Demostrar que f no domina asintóticamente a g

$$\neg(\exists m \geq 0, k \geq 0 \text{ tales que } |g(n)| \leq m|f(n)|, \forall n \geq k)$$



- **Ejemplo 1:** Sean $f(n) = n$ y $g(n) = n^3$ funciones de \mathbb{N} a \mathbb{R} .

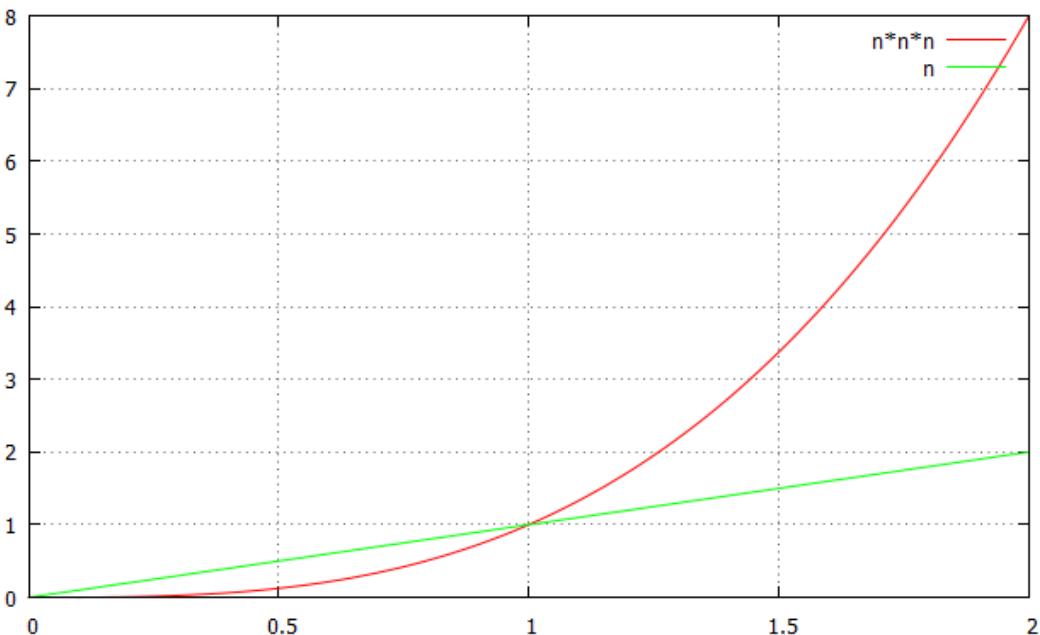
- Demostrar que g domina asintóticamente a f

$\exists m \geq 0, k \geq 0$ tales que $|f(n)| \leq m|g(n)|, \forall n \geq k$

- Substituyendo $f(n)$ y $g(n)$

$$|n| \leq m|n^3|, \forall n \geq k$$

- Si se toman $m = 1$ y $k = 1$, las desigualdades anteriores se cumplen, por lo tanto, m y k existen, y en consecuencia g domina asintóticamente a f .



- **Ejemplo 1:** Sean $f(n) = n$ y $g(n) = n^3$ funciones de \mathbb{N} a \mathbb{R} .

- Demostrar que f no domina asintóticamente a g
 $\neg(\exists m \geq 0, k \geq 0 \text{ tales que } |g(n)| \leq m|f(n)|, \forall n \geq k)$

- Aplicando la negación se tiene

$$\forall m \geq 0, k \geq 0, \exists n \geq k \text{ tal que } |g(n)| > m|f(n)|$$

- Sustituyendo g y f en cada lado de la desigualdad

$$|n^3| > m|n|$$

- Simplificando

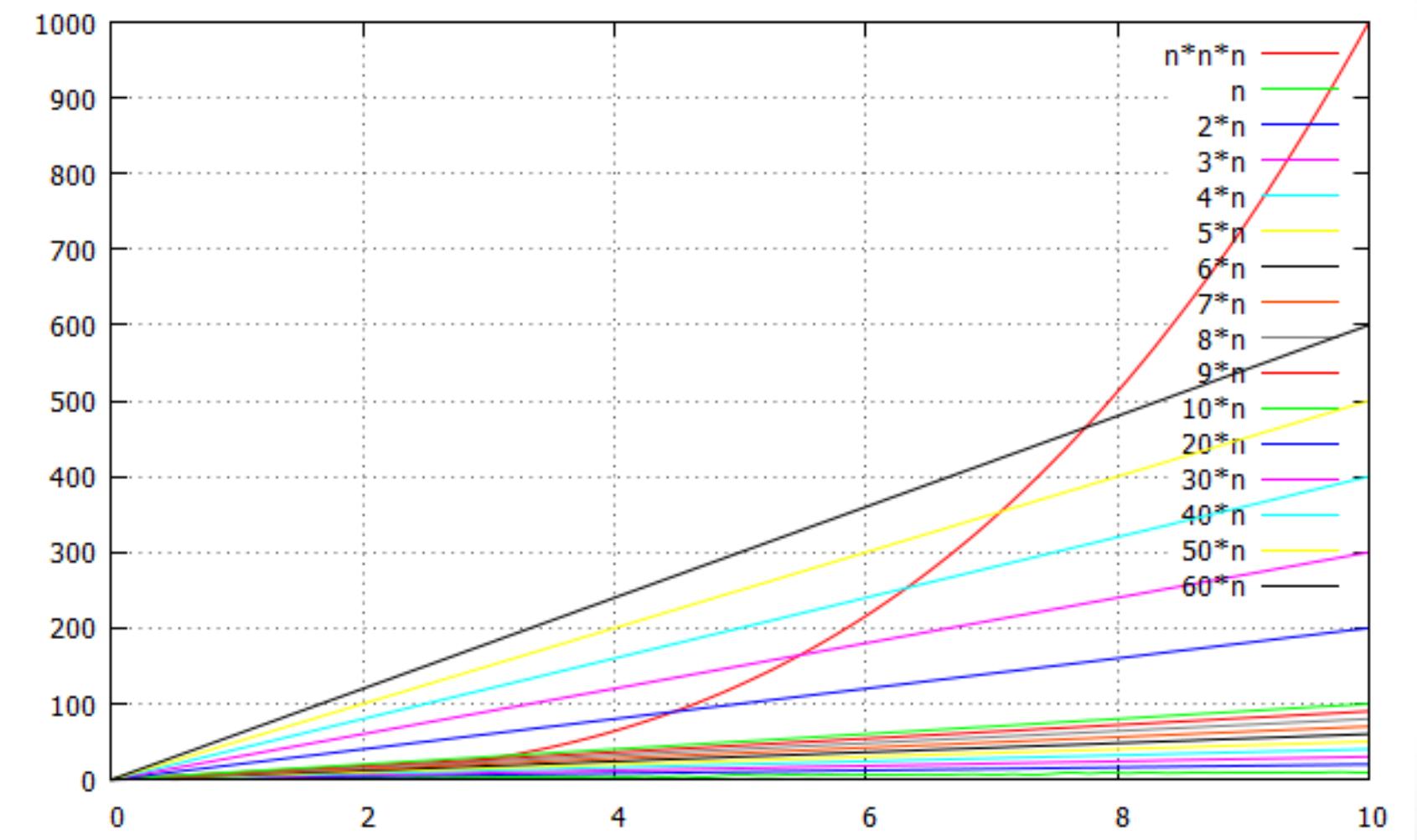
$$\begin{aligned} |n^2| &> m \\ n^2 &> m \end{aligned}$$

$$n > \sqrt{m} \quad \text{y} \quad n \geq k$$

Si se toma $n > \max(\sqrt{m}, k)$ ambas desigualdades se cumplen para toda $m \geq 0$ y $k \geq 0$, $\therefore f$ no domina asintóticamente a g



- Ejemplo 1: Sean $f(n) = n$ y $g(n) = n^3$ funciones de \mathbb{N} a \mathbb{R} .



Dominio asintótico (Ejemplo 2)

- **Ejemplo 2:** Sea $g(n)$ una función de \mathbb{N} a \mathbb{R} y $f(n) = cg(n)$ con $c > 0$ y $c \in \mathbb{R}$.

i. Demostrar que f d. a. g

$$\exists m \geq 0, k \geq 0 \text{ tales que } |g(n)| \leq m|cg(n)|, \forall n \geq k$$

ii. Demostrar que g d. a. f

$$\exists m \geq 0, k \geq 0 \text{ tales que } |cg(n)| \leq m|g(n)|, \forall n \geq k$$



- **Ejemplo 2:** Sea g una función de \mathbb{N} a \mathbb{R} y $f(n) = cg(n)$ con $c > 0$ y $c \in \mathbb{R}$.

i. Demostrar que f d. a. g

$$\exists m \geq 0, k \geq 0 \text{ tales que } |g(n)| \leq m|cg(n)|, \forall n \geq k$$

- Pero $|ab| = |a||b|$ por lo tanto

$$|g(n)| \leq |m||c||g(n)|, \forall n \geq k$$

- Como $c > 0$; entonces

$$|g(n)| \leq mc|g(n)|, \forall n \geq k$$

- Tomando $m = \frac{1}{c}$ y $k = 0$ se tiene

$$|g(n)| \leq |g(n)|, \forall n \geq 0 \therefore f \text{ d. a. } g$$



- **Ejemplo 2:** Sea g una función de \mathbb{N} a \mathbb{R} y $f(n) = cg(n)$ con $c > 0$ y $c \in \mathbb{R}$.

i. Demostrar que g d.a. f

$$\exists m \geq 0, k \geq 0 \text{ tales que } |cg(n)| \leq m|g(n)|, \forall n \geq k$$

- Esto es

$$c|g(n)| \leq m|g(n)|, \forall n \geq k$$

- Tomando $m = c$ y $k = 0$ se tiene

$$c|g(n)| \leq c|g(n)|, \forall n \geq 0 \therefore g \text{ d.a. } f$$



Dominio asintótico a la función complejidad

- Cuando se hace el **análisis teórico** para obtener la **función complejidad** $f(n)$ que caracterice a un algoritmo, se está obteniendo un **modelo de comportamiento** para la demanda de recursos en función del parámetro n ; de tal forma que si $t(n)$ es la cantidad real del recurso que se consume para una implantación específica del algoritmo se tiene que:

$$\begin{aligned} t(n) &\alpha f(n) \\ t(n) &= cf(n) \\ |t(n)| &\leq c|f(n)| \end{aligned}$$

- i.e. **$f(n)$ domina asintóticamente a cualquier $t(n)$** ; dicho de otra manera la **demandas de recursos** se va a regir por el **modelo de crecimiento** que observe $f(n)$.



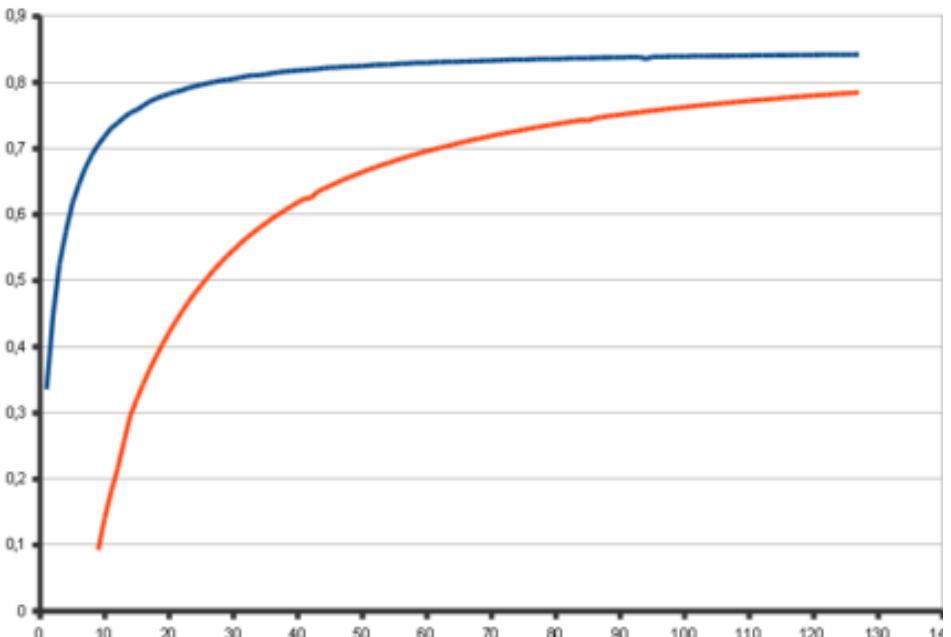
Notación asintótica

- El interés principal del análisis de algoritmos radica en saber cómo crece la demanda de recursos, cuando el tamaño del problema crece. Esto es la **eficiencia asintótica** del algoritmo. Se denomina “asintótica” porque analiza el comportamiento de las funciones en el *límite*, es decir, su **tasa de crecimiento**.
- La **notación asintótica** captura el **comportamiento** de la función para **valores grandes de n** . Se consideran las **funciones asintóticamente no negativas**.
- Las notaciones no son dependientes de los tres casos anteriormente vistos, es por eso que **una notación que determine el peor caso** puede estar presente en una o en todas las situaciones.



Notación de orden

- Cuando describimos cómo es que el número de operaciones $f(n)$ depende del tamaño n ; lo que nos interesa es encontrar el patrón de crecimiento o cota para la función complejidad y así caracterizar al algoritmo; una vez hecha esta caracterización podremos agrupar las funciones de acuerdo al número de operaciones que realizan.



Cota Superior: Notación O mayúscula



- La notación O (Omicron mayúscula) se utiliza para comparar funciones. Dada una función f , se desea estudiar funciones g que a lo sumo crezcan tan deprisa como f .

- **Definición:** Sean f y g funciones de \mathbb{N} a \mathbb{R} . Si existen constantes c y x_0 tales que:

$$\forall x > x_0, |f(x)| \leq c |g(x)|$$

- i.e. que para $x > x_0$, f es menor o igual a un múltiplo c de g , decimos que:

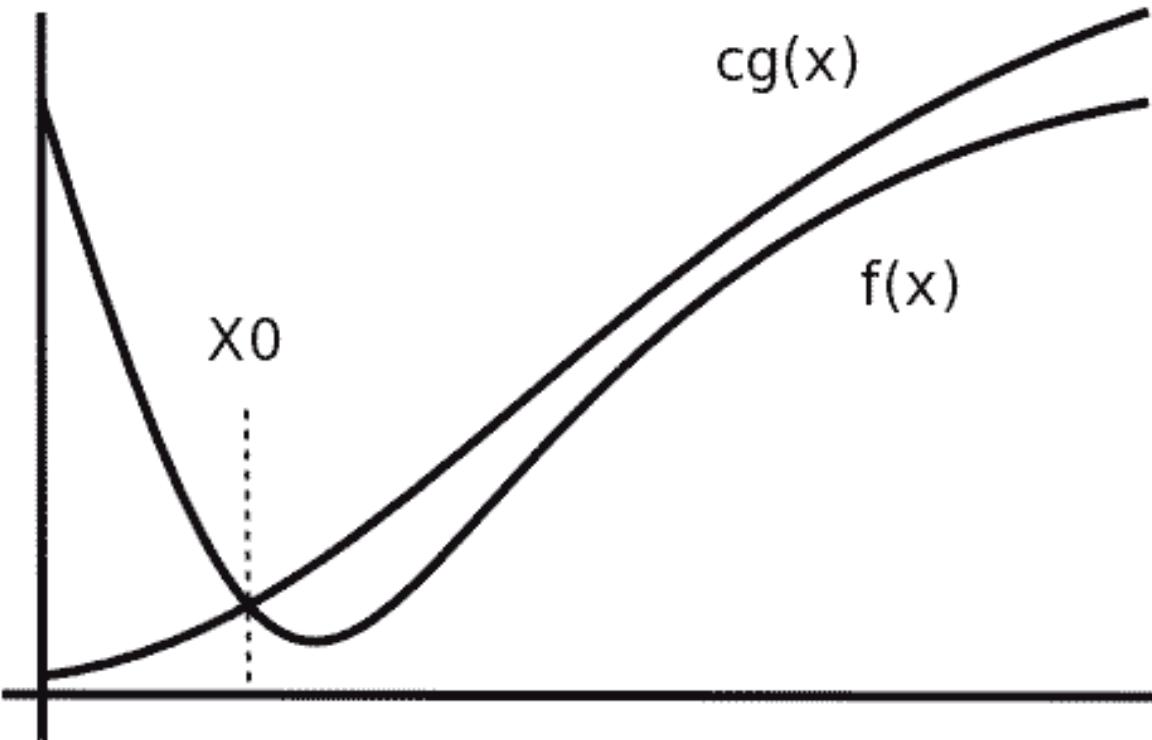
$$f(x) = O(g(x))$$

- La definición formal es:

$$f(x) = O(g(x)) \Leftrightarrow \exists c > 0, x_0 > 0 \mid \forall x > x_0, |f(x)| \leq c |g(x)|$$



- $f(x) = O(g(x)) \Leftrightarrow \exists c > 0, x_0 > 0 \mid \forall x > x_0, |f(x)| \leq c|g(x)|$



$$O(g(x)) = \left\{ f(x) : \text{existen } c, x_0 > 0 \text{ tales que} \right. \\ \left. \forall x \geq x_0 : 0 \leq |f(x)| \leq c|g(x)| \right\}$$



- **Ejemplo 3:** Muestre que $f_t(n) = 3n^3 + 5n^2 + 9 = O(n^3)$

- Partiendo de:

$$f(x) = O(g(x)) \Leftrightarrow \exists c > 0, x_0 > 0 | \forall x > x_0, |f(x)| \leq c|g(x)|$$

- Sustituyendo

$$|3n^3 + 5n^2 + 9| \leq c|n^3|$$

- Como ambas funciones van de \mathbb{N} a x y $x_0 > 0$, y desde $x_0 = 0$ no negativa

$$3n^3 + 5n^2 + 9 \leq cn^3$$

- Agrupando

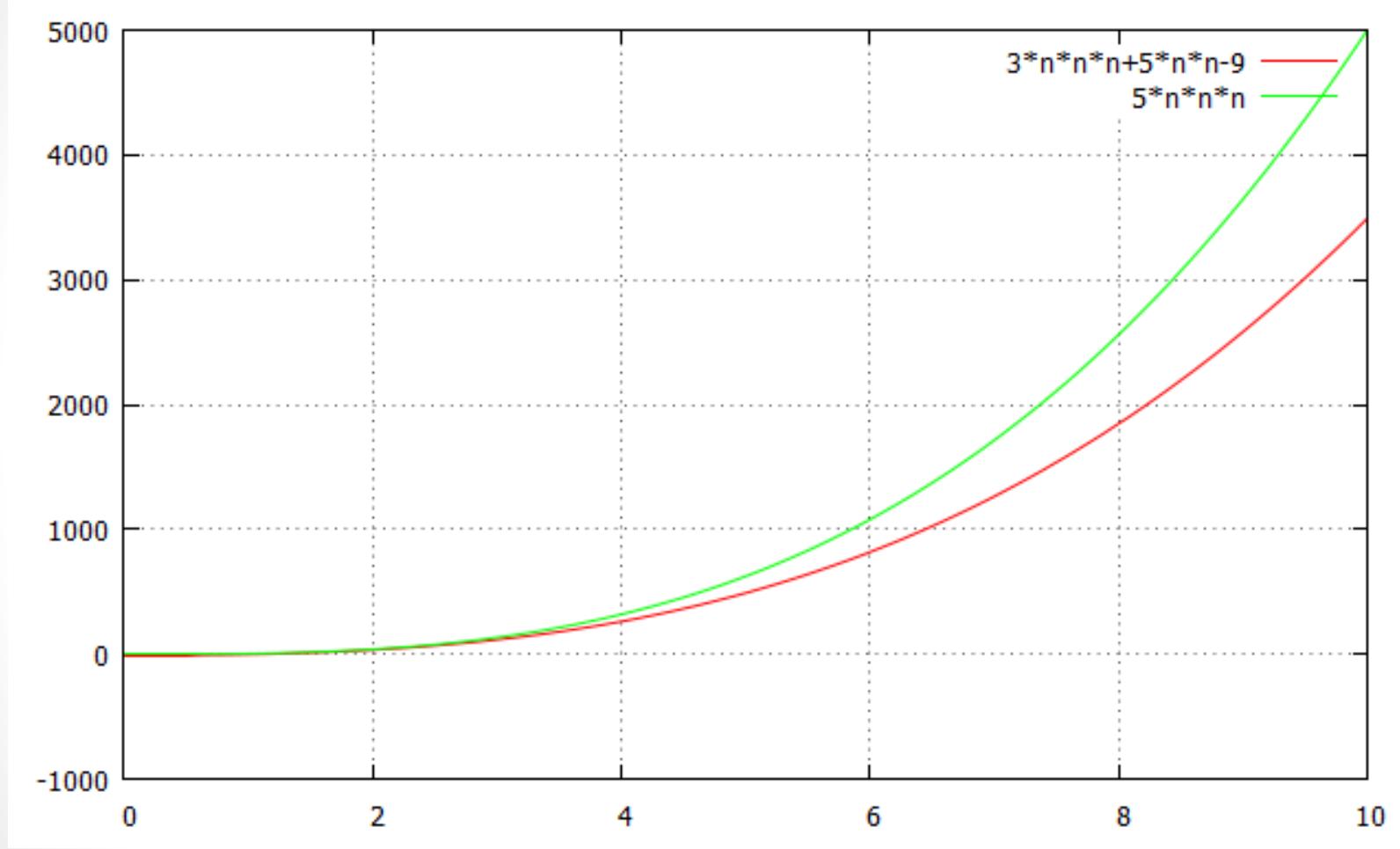
$$5n^2 \leq (c - 3)n^3 - 9$$

- Si $c = 5$ y $x_0 = 0$, $\forall n > x_0$ *ERROR: Revisar

$$5n^2 \leq 2n^3 - 9 \therefore 3n^3 + 5n^2 + 9 = O(n^3)$$



- Ejemplo 3: $5n^2 \leq 2n^3 - 9 \therefore 3n^3 + 5n^2 + 9 = O(n^3)$



Cota Superior no ajustada: Notación o minúscula

- La cota superior asintótica dada por la notación O puede o no ser ajustada asintóticamente. La cota $2n^2 = O(n^2)$ es ajustada asintóticamente, pero la cota $2n = o(n^2)$ no lo es. Se emplea la notación o para denotar una cota superior que no es ajustada asintóticamente.
- **Definición:** Sean f y g funciones de \mathbb{N} a \mathbb{R} . Si para toda constante $c > 0$ y una constante x_0 se cumple que:

$$\forall x > x_0, c > 0, |f(x)| \leq c |g(x)|$$

- i.e. que para $x > x_0$, f es menor o igual a todos los múltiplos $c > 0$ de g , decimos que:

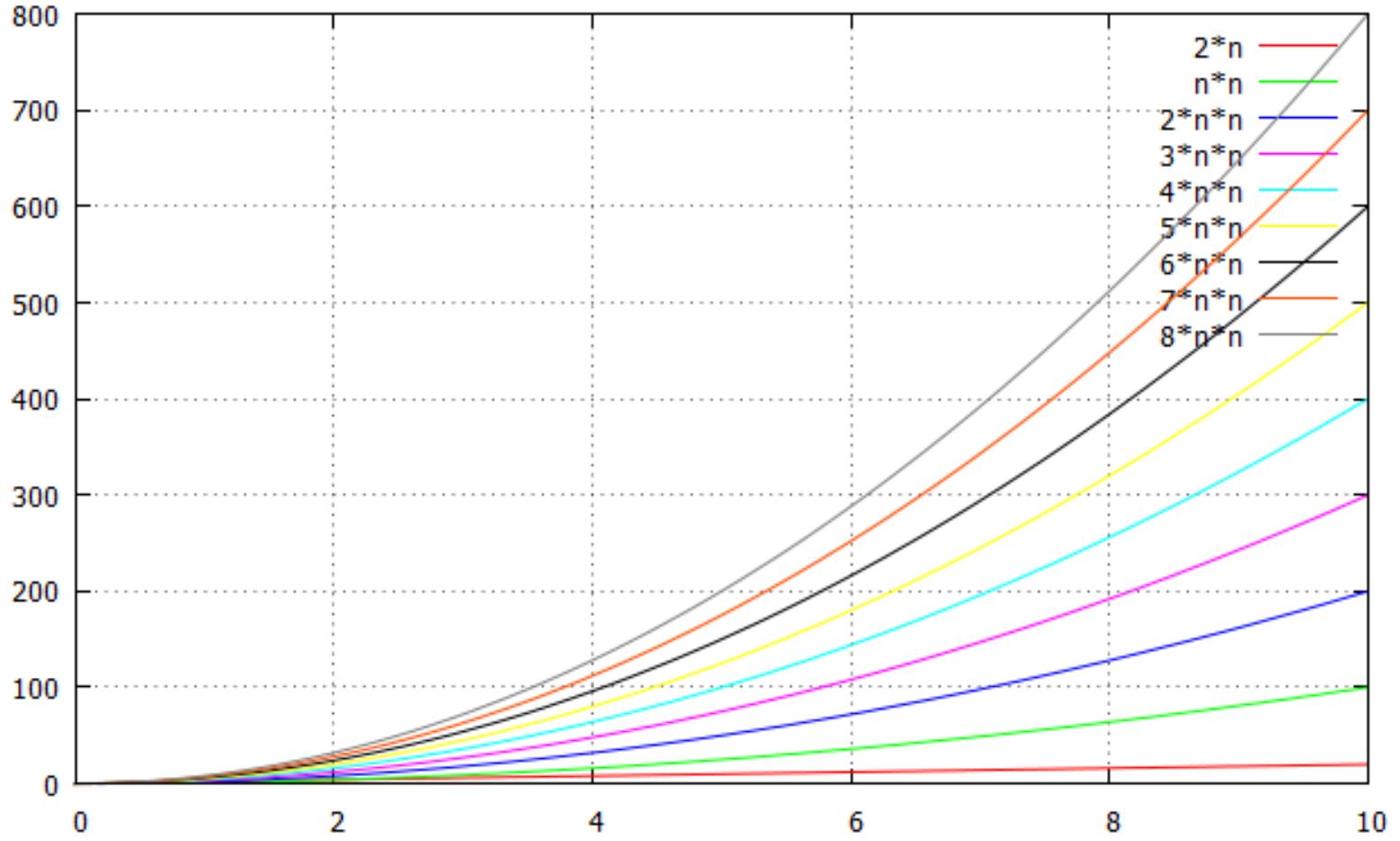
$$f(x) = o(g(x))$$

- La definición formal es:

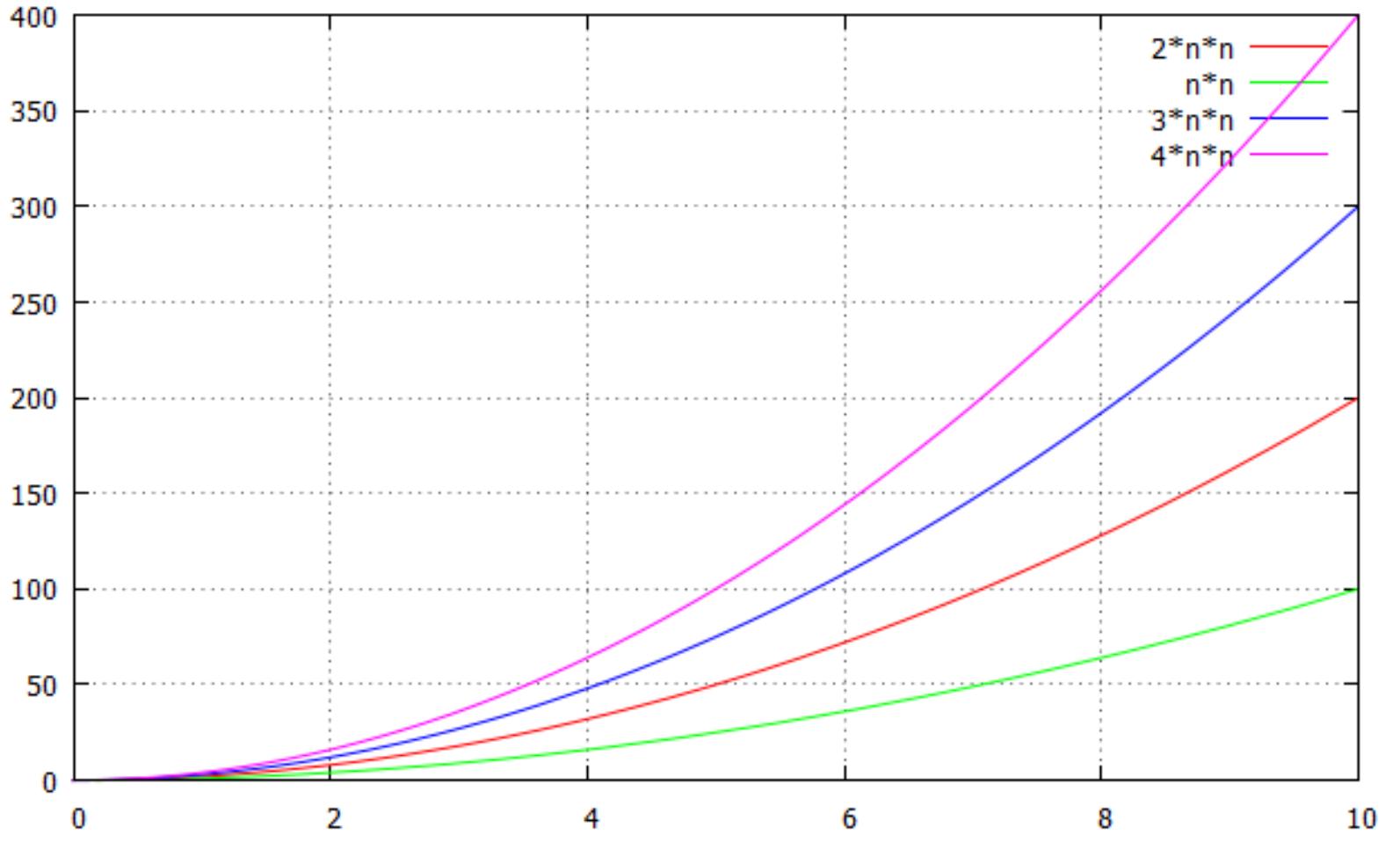
$$f(x) = o(g(x)) \Leftrightarrow \exists x_0 > 0 \mid \forall x > x_0, c > 0, |f(x)| \leq c |g(x)|$$



$$2n = o(n^2)$$



$$2n^2 \neq o(n^2)$$



Diferencia entre O y o

- Las notaciones de O y o son similares. La diferencia principal es, que en $f(n) = O(g(n))$, la cota $0 \leq f(n) \leq cg(n)$ se cumple para *alguna* constante $c > 0$, pero en $f(n) = o(g(n))$, la cota $0 \leq f(n) \leq cg(n)$ se cumple para ***todas*** las constantes $c > 0$.
- Intuitivamente en la notación o , la función $f(n)$ se vuelve insignificante con respecto a $g(n)$ a medida que n se acerca a infinito

$$\lim_{n \rightarrow \infty} \left(\frac{f(n)}{g(n)} \right) = 0$$

- Para o la desigualdad se mantiene para todas las constantes positivas, mientras que para O la desigualdad se mantiene sólo para algunas constantes positivas.

$$f(x) = O(g(x)) \Leftrightarrow \exists c > 0, x_0 > 0 \mid \forall x > x_0, |f(x)| \leq c|g(x)|$$

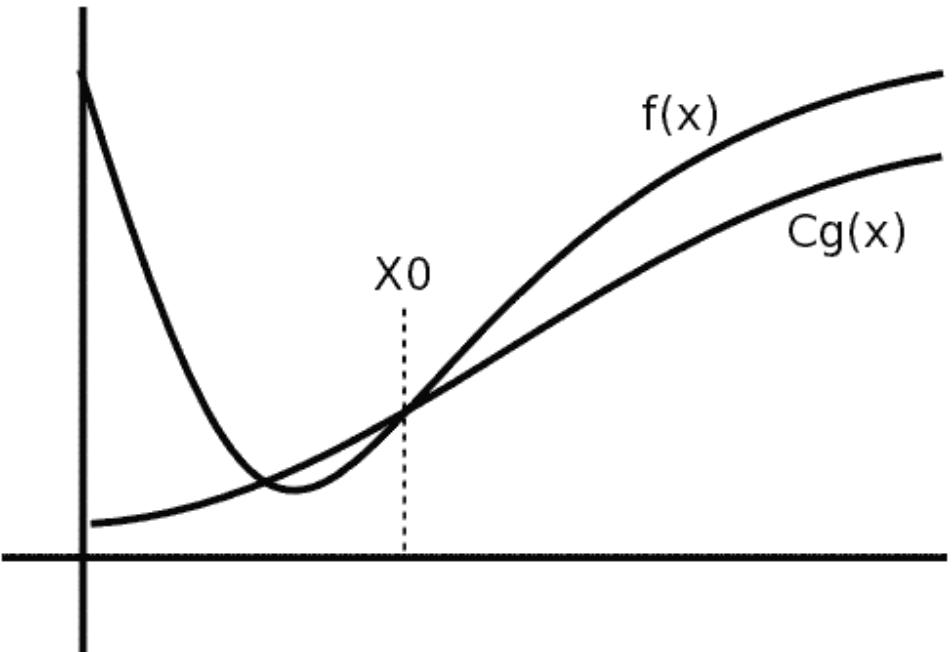
$$f(x) = o(g(x)) \Leftrightarrow \exists x_0 > 0 \mid \forall x > x_0, c > 0, |f(x)| \leq c|g(x)|$$



Cota Inferior: Notación Ω

- La notación Ω Es el reverso de O , i.e es una función que sirve de **cota inferior** de otra función cuando el argumento tiende a infinito

$$f(x) = \Omega(g(x)) \Leftrightarrow \exists c > 0, x_0 > 0 \mid \forall x > x_0, c|g(x)| \leq |f(x)|$$



$$\Omega(g(x)) = \left\{ f(x) : \text{existen } c, x_0 \text{ constantes positivas tales que} \right. \\ \left. \forall x : x_0 \leq x : 0 \leq cg(x) \leq f(x) \right\}$$

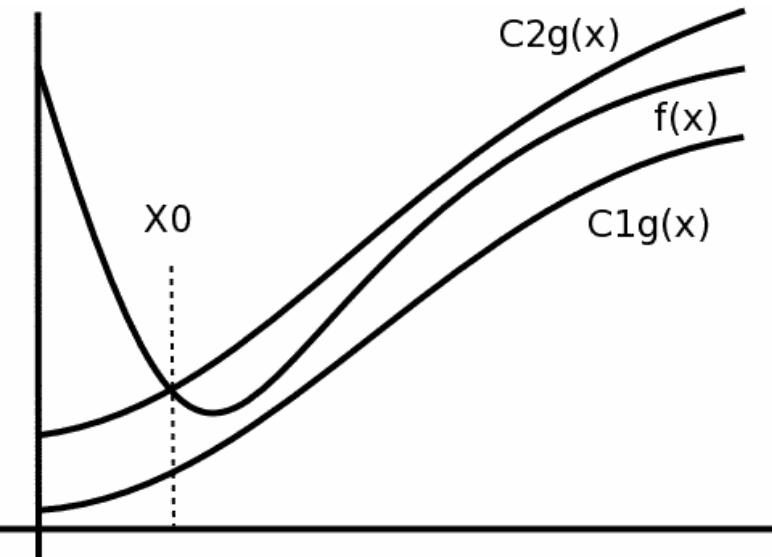


Cota ajustada asintótica: Notación Θ

- La **cota ajustada asintótica o de orden exacto** es una función que sirve de cota tanto superior como inferior de otra función cuando el argumento tiende a infinito.

$$f(x) = \Theta(g(x)) \Leftrightarrow \exists c_1 > 0, c_2 > 0, x_0 > 0 \mid \forall x > x_0, c_1|g(x)| \leq |f(x)| \leq c_2|g(x)|$$

Θ Grande dice que ambas funciones se dominan mutuamente, en otras palabras, son asintóticamente equivalentes.



$$f(x) = \Theta(g(x)) \text{ si y solo si } f(x) = O(g(x)) \text{ y } f(x) = \Omega(g(x))$$

$$\Theta(g(x)) = \left\{ f(x) : \begin{array}{l} \text{existen } c_1, c_2, x_0 \text{ constantes positivas tales que} \\ \forall x : x_0 \leq x : 0 \leq c_1 g(x) \leq f(x) \leq c_2 g(x) \end{array} \right\}$$

Observaciones sobre las cotas asintóticas

1. La utilización de las cotas asintóticas para comparar funciones de tiempo de ejecución se basa en la hipótesis de que son suficientes para decidir el mejor algoritmo, prescindiendo de las constantes de proporcionalidad. Sin embargo, esta hipótesis puede no ser cierta cuando el tamaño de la entrada es pequeño.

2. Para un algoritmo dado se pueden obtener tres funciones que miden su tiempo de ejecución, que corresponden a sus casos mejor, medio y peor, y que denominaremos respectivamente $T_m(n)$, $T_{1/2}(n)$ y $T_p(n)$, para cada una de ellas podemos dar hasta 4 cotas asintóticas (O , o , Ω , Θ) de crecimiento, por lo que se obtiene un total de 12 cotas para el algoritmo.



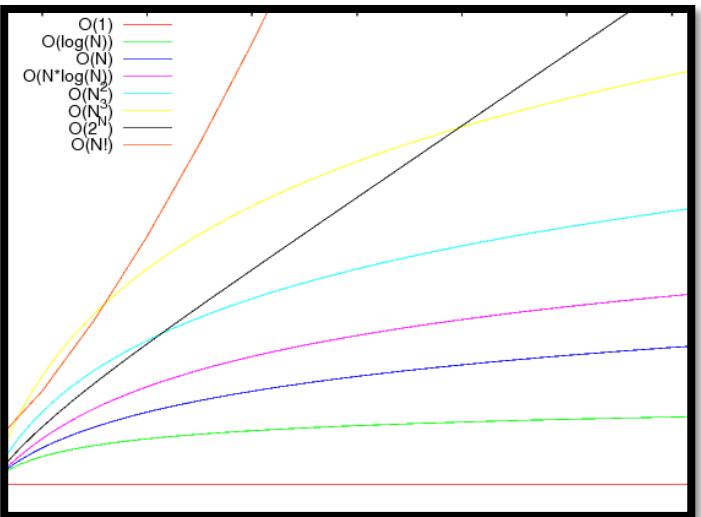
3. Para simplificar, dado un algoritmo diremos que su orden de complejidad es $O(f)$ si su tiempo de ejecución para el peor caso es de orden O de f , es decir, $T_p(n)$ es de orden $O(f)$. De forma análoga diremos que su orden de complejidad para el mejor caso es $\Omega(g)$ si su tiempo de ejecución para el mejor caso es de orden Ω de g , es decir, $T_m(n)$, es de orden $\Omega(g)$.
4. Por último, diremos que un algoritmo es de orden exacto $\Theta(f)$ si su tiempo de ejecución en el caso medio $T_{1/2}(n)$ es de este orden.



Ordenes de complejidad (Cota superior)

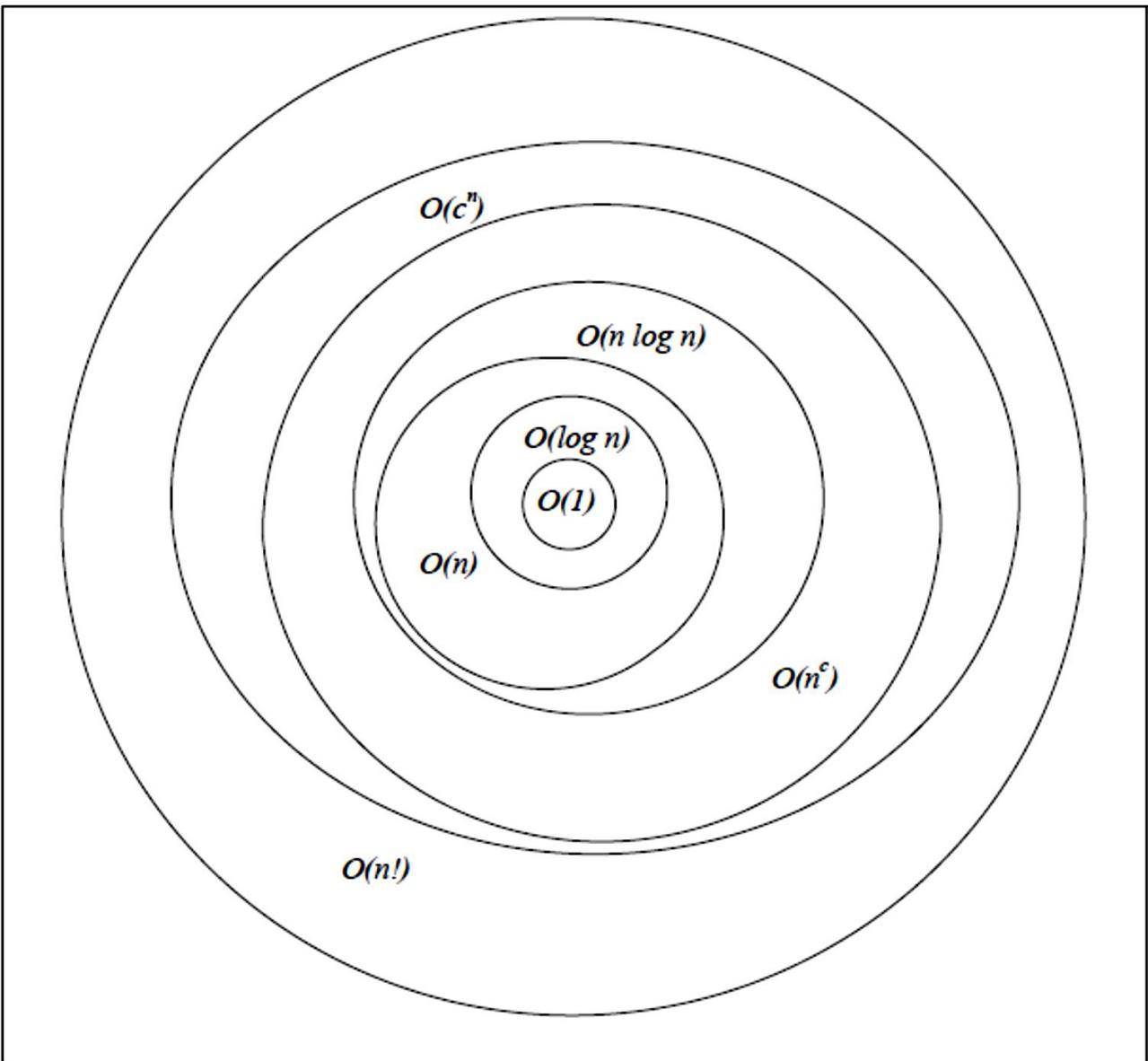
- Dado que las funciones complejidad están en el conjunto de funciones que van de \mathbb{N} a \mathbb{R} ; es posible clasificar los algoritmos según el orden de su función complejidad. Gran parte de los algoritmos tienen complejidad que cae en uno de los siguientes casos:

- $O(1)$ Complejidad constante**
- $O(\log n)$ Complejidad logarítmica**
- $O(n)$ Complejidad lineal**
- $O(n \log n)$ Complejidad “ $n \log n$ ”**
- $O(n^2)$ Complejidad cuadrática**
- $O(n^3)$ Complejidad cubica**
- $O(c^n); c > 1$ Complejidad exponencial**
- $O(n!)$ Complejidad factorial**



$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(c^n) \subset O(n!)$$

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(c^n) \subset O(n!)$$



$f(n) = O(1)$ Complejidad constante

- Los algoritmos de complejidad constante ejecutan siempre el mismo número de pasos sin importar cuan grande es n.

$f(n) = O(\log n)$ Complejidad logarítmica

- Los algoritmos de complejidad logarítmica, habitualmente son algoritmos que resuelven un problema transformándolo en problemas menores.

$f(n) = O(n)$ Complejidad lineal

- Los algoritmos de complejidad lineal generalmente tratan de manera constante cada n del problema por lo que si n dobla su tamaño el algoritmo también dobla el número de pasos.

$f(n) = O(n \log n)$ Complejidad “n log n”

- Los algoritmos de complejidad “n log n” generalmente dividen un problema en problemas más sencillos de resolver para finalmente combinar las soluciones obtenidas.



$f(n) = O(n^2)$ Complejidad cuadrática

- Los algoritmos de complejidad cuadrática aparecen cuando los datos se procesan por parejas, en la mayoría de los casos en bucles anidados.

$f(n) = O(n^3)$ Complejidad cubica

- Los algoritmos de complejidad cubica son útiles para resolver problemas pequeños p.g. si $n=100$ el número de operaciones es de 1,000,000.

$O(c^n); c > 1$ Complejidad exponencial

- Los algoritmos de complejidad exponencial no son útiles desde el punto de vista practico, aparecen cuando un problema se soluciona empleando fuerza bruta.



$O(n!)$ Complejidad factorial

- Un algoritmo de complejidad factorial generalmente aparece cuando el problema también es resuelto por fuerza bruta y es un problema complejo por definición; o cuando se maneje de mala manera un algoritmo recursivo.

Función de coste	Tamaño n					
	10	20	30	40	50	60
n	0,00001 seg.	0,00002 seg.	0,00003 seg.	0,00004 seg.	0,00005 seg.	0,00006 seg.
n^2	0,0001 seg.	0,0004 seg.	0,0009 seg.	0,0016 seg.	0,0035 seg.	0,0036 seg.
n^3	0,001 seg.	0,008 seg.	0,027 seg.	0,064 seg.	0,125 seg.	0,316 seg.
n^5	0,1 seg.	3,2 seg.	24,3 seg.	1,7 min.	5,2 min.	13 min.
2^n	0,001 seg.	1 seg.	17,9 min.	12,7 días	35,7 años	366 siglos
3^n	0,059 seg.	58 min.	6,5 años	3855 siglos	10^8 siglos	10^{13} siglos





Instituto Politécnico Nacional

Escuela Superior de Cómputo



Análisis de algoritmos

Tema 05: Análisis de algoritmos no recursivos

M. en C. Edgardo Adrián Franco Martínez
<http://www.eafranco.com>
edfrancom@ipn.mx
[@edfrancom](https://twitter.com/edfrancom) [f edgaroadrianfrancocom](https://facebook.com/edgaroadrianfrancocom)



Contenido

- Análisis de algoritmos no recursivos
- La notación de Landau O
 - La notación O
 - Principio de invarianza del análisis asintótico
- Reglas prácticas del análisis de algoritmos
- Ordenes más comunes de los algoritmos
- Comportamiento de las funciones
- Análisis por bloques de algoritmos iterativos
 - Regla 1: Secuencia de instrucciones
 - Regla 2: Decisiones
 - Regla 3: Ciclos
 - Consideraciones especiales
- Logaritmos
 - Propiedades de los logaritmos
- Ejemplos
 - Ejemplo 01: Ordenamiento por intercambio
 - Ejemplo 02: Multiplicación de matrices de $n \times n$
 - Ejemplo 03: Algoritmo de Horner
 - Ejemplo 04: Fibonacci (Iterativo)
 - Ejemplo 05: Búsqueda binaria



Análisis de algoritmos no recursivos

- El análisis de complejidad de los algoritmos no recursivos (**iterativos**) se realiza bajo los principios del **peor caso** y generalmente devolverá la cota superior ajustada del orden de este (O).
- En principio se considera válido **cuando solo se desea obtener una cota para valores grandes de n** , basándose en que se cumple el **principio de invarianza del análisis asintótico**.
- Para los algoritmos iterativos es únicamente necesario conocer los órdenes de complejidad O de las tres estructuras de control que todo algoritmo iterativo puede emplear.



La notación de Landau (O)

- Se dice que la función $f(n)$ “es de orden $O(g(n))$ ” [$O(g(n))$], si existen constantes positivas c y n_0 tales que $|f(n)| \leq c |g(n)|$ cuando $n \geq n_0$
- Ejemplos:
 - $n+5$ es $O(n)$ pues $n+5 \leq 2n$ para toda $n \geq 5$
 - $(n+1)^2$ es $O(n^2)$ pues $(n+1)^2 \leq 4n^2$ para $n \geq 1$
 - $(n+1)^2$ NO es $O(n)$ pues para cualquier $c > 1$ no se cumple que $(n+1)^2 \leq c*n$



La notación O

- La notación O proporciona una cota superior para la tasa de crecimiento de una función.
- La siguiente tabla muestra la relación asintótica de la notación de orden O .

	$f(n)$ es $O(g(n))$	$g(n)$ es $O(f(n))$
$g(n)$ crece más	Sí	No
$f(n)$ crece más	No	Sí
Igual crecimiento	Sí	Sí



Principio de invarianza del análisis asintótico

- Cambios en el entorno HW o SW afectan a factores constantes pero no al orden de complejidad **$O(f(n))$**
- El análisis de la eficiencia es **asintótico** → sólo es válido para tamaños de problema suficientemente grandes lo que hace valido el principio de invarianza.

“Dos implementaciones de un mismo algoritmo no diferirán más que en una constante multiplicativa”

Si $f_1(n)$ y $f_2(n)$ son los tiempos consumidos por dos implementaciones de un mismo algoritmo, se verifica que:

$$\begin{aligned} & \exists c, d \in \mathbb{R}, \\ & f_1(n) \leq c \cdot f_2(n) \\ & f_2(n) \leq d \cdot f_1(n) \end{aligned}$$



Reglas prácticas del análisis de algoritmos

- **Operaciones primitivas:** tienen complejidad constante **O(1)**
- **Secuencia de instrucciones:** máximo de la complejidad de cada instrucción (**regla de la suma**).
- **Condiciones simples:** operaciones necesarias para evaluar la condición más las requeridas para ejecutar la consecuencia (**peor caso**).
- **Condiciones alternativas:** operaciones necesarias para evaluar la condición más las operaciones requeridas para ejecutar el mayor número de operaciones de las consecuencias (**peor caso**).



- **Bucle con iteraciones fijas:** multiplicar el número de iteraciones por la complejidad del cuerpo (**regla del producto**).
- **Bucle con iteraciones variables:** igual pero poniéndose en el peor caso (**ejecutar el mayor número de iteraciones posible**).
- **Llamadas a subprogramas**, funciones o métodos:
Operaciones de asignación de cada parámetro más las operaciones de ejecución del cuerpo, más el número de operaciones del retorno.

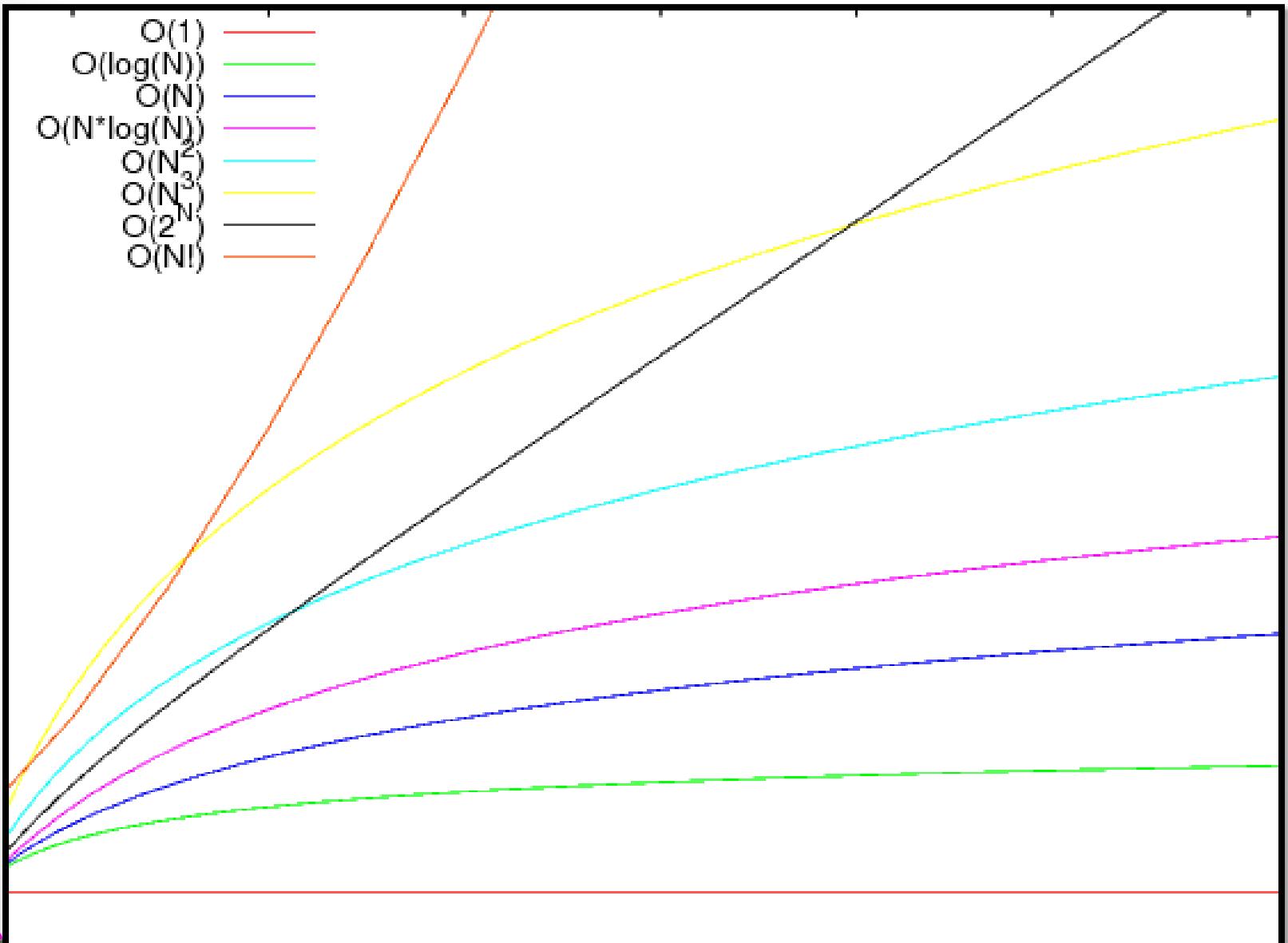


Ordenes más comunes de los algoritmos

- $O(1)$ Constante
- $O(n)$ Lineal
- $O(n^2)$ Cuadrático
- $O(n^3)$ Cúbico
- $O(n^m)$ Polinomial
- $O(\log(n))$ Logarítmico
- $O(n\log(n))$ $n\log(n)$
- $O(m^n)$ exponencial
- $O(n!)$ factorial

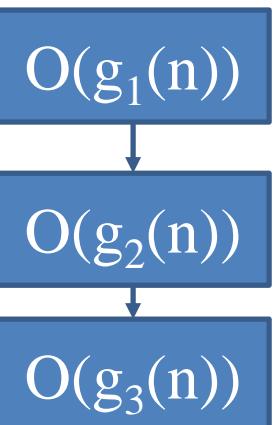


Comportamiento de las funciones



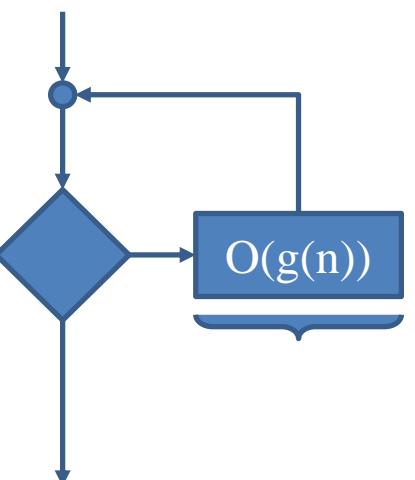
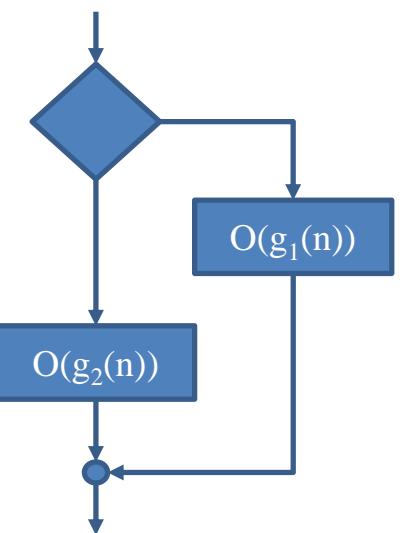
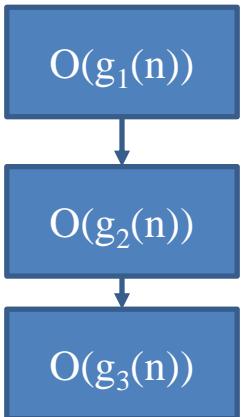
Análisis por bloques de algoritmos iterativos

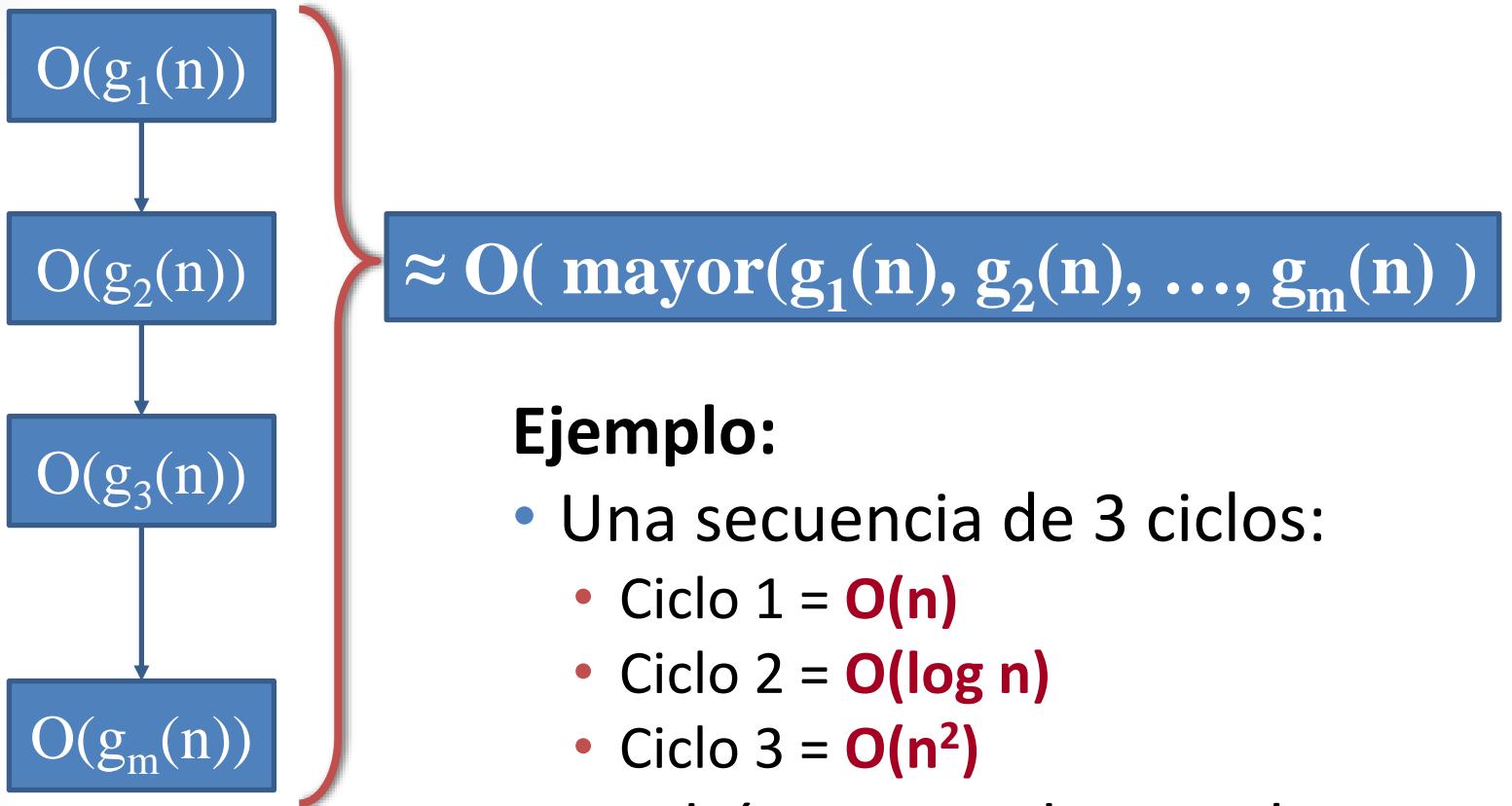
- El análisis de complejidad por bloques proporciona un importante medio para hacer un análisis más dinámico, que se basa principalmente en el conocimiento del orden de los bloques de instrucciones.
- Un bloque de instrucciones puede ser un algoritmo del cual conocemos su complejidad computacional. Para determinar el orden de un bloque, es necesario tener en mente el orden de las estructuras de control más usuales.



- El análisis por bloques implica el análisis de:

1. Secuencias de instrucciones
2. Condicionales
3. Ciclos



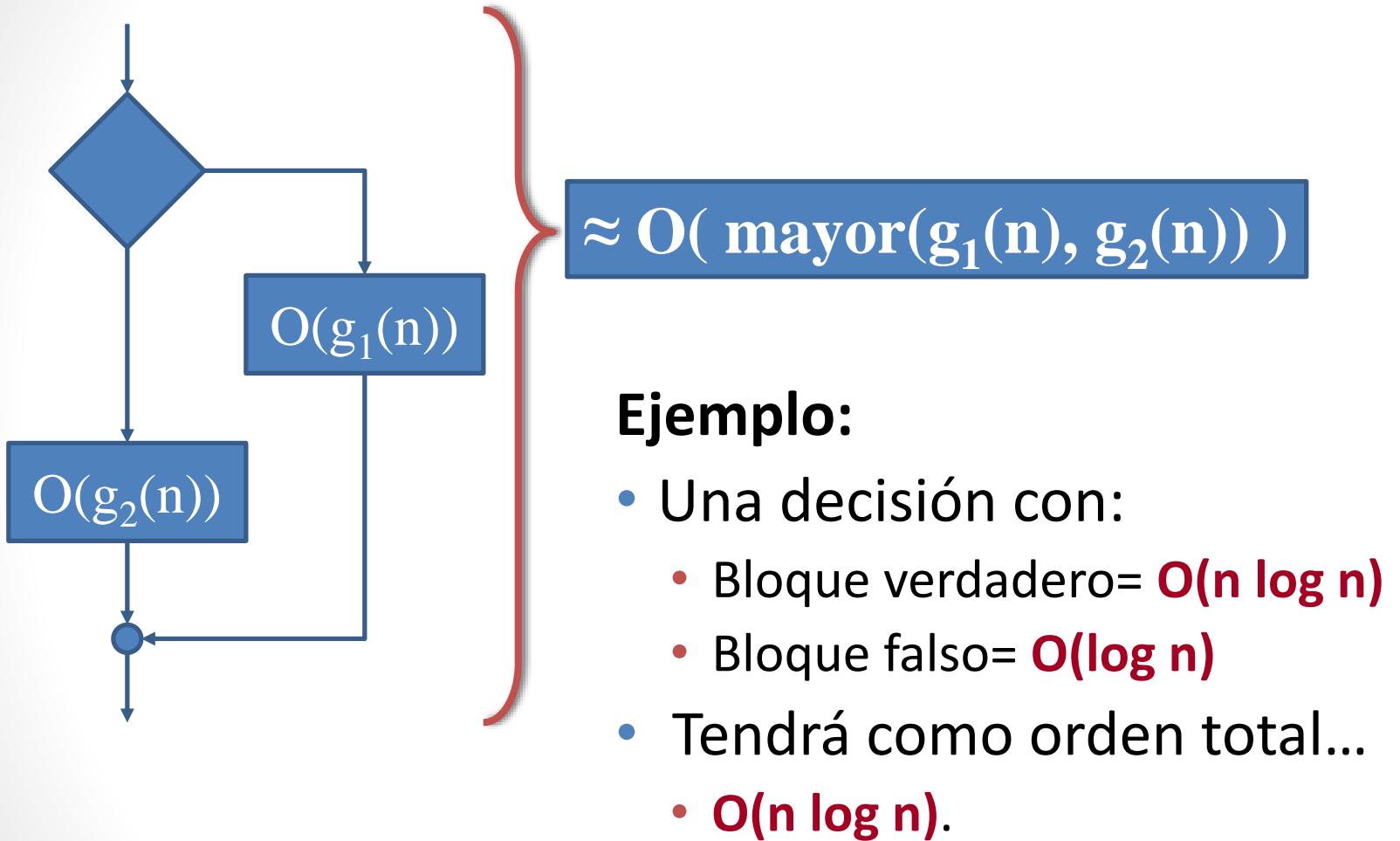


Ejemplo:

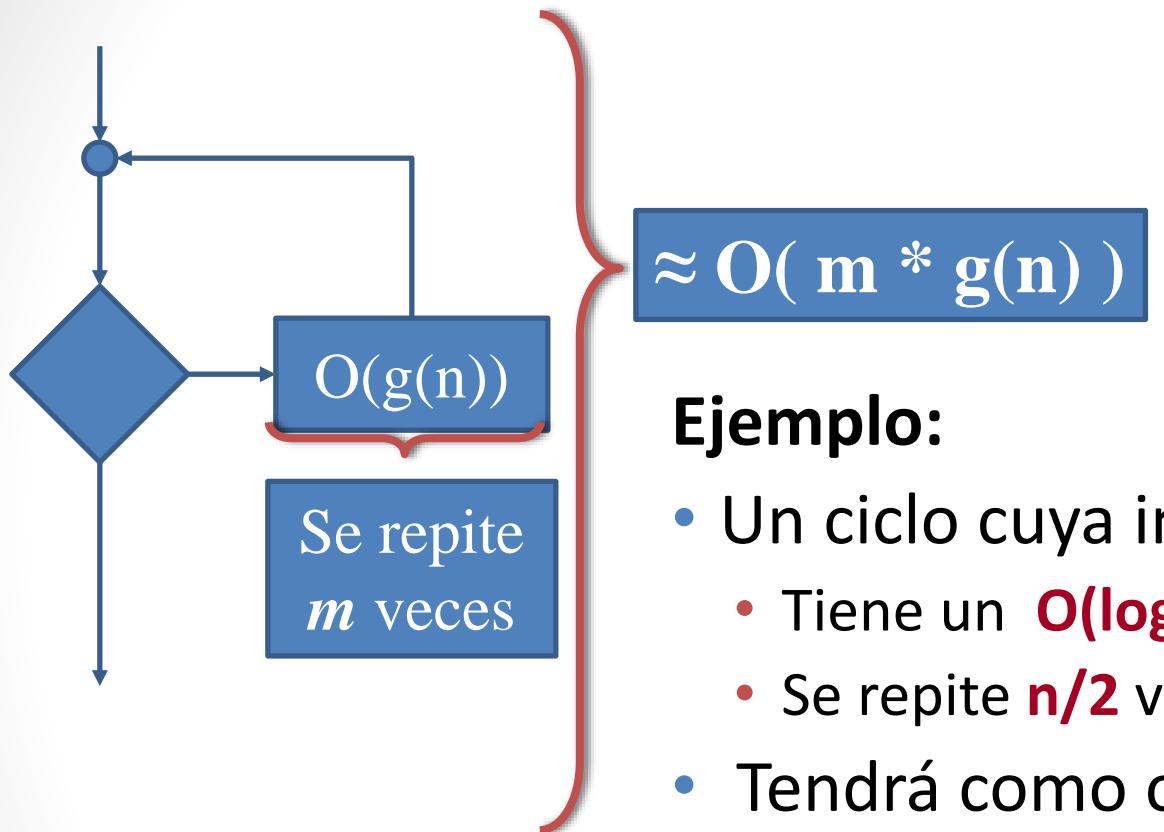
- Una secuencia de 3 ciclos:
 - Ciclo 1 = $O(n)$
 - Ciclo 2 = $O(\log n)$
 - Ciclo 3 = $O(n^2)$
- Tendrá como orden total...
 - $O(n^2)$.



Regla 2: Decisiones



Regla 3: Ciclos



Ejemplo:

- Un ciclo cuya instrucción:
 - Tiene un **O(log n)**
 - Se repite **n/2** veces
 - Tendrá como orden total...
 - **O(½ n log n) = O(n log n).**



Consideraciones especiales

- En decisiones y ciclos anidados:
 - Analizar el código desde la instrucción más interna hacia el más externa.
- Tips para los ciclos:
 - ¿“Normalmente” cuál es el orden de la instrucción interna?
 - Si la variable de control se incrementa o decrementa con un valor constante: **Orden LINEAL**.
 - Si la variable de control se multiplica o divide por un valor constante: **Orden LOGARÍTIMICO**.



Ejemplo 01: Ordenamiento por intercambio

```
for (int i=1; i<n; i++)  
    for (int j=i+1; j<=n;j++)  
        if(a[ j ] < a[ i ])  
            intercambia(a[ i ], a[ j ]); → O( 1 )
```

Regla 2: Decisiones = mayor de las 2 ramas



```

for (int i=1; i<n; i++)
    for (int j=i+1; j<=n;j++)
        if (a[ j ] < a[ i ])
            intercambia(a[ i ], a[ j ]);
```

Peor caso: se repite n-1 veces

$O(1)$

$O(n)$

*Regla 3: Ciclos = # veces * orden de la instrucción interna*



Se repite $n-1$ veces

```

for (int i=1; i<n; i++)
    for (int j=i+1; j<=n;j++)
        if(a[ j ] < a[ i ])
            intercambia(a[ i ], a[ j ]);
```

$\rightarrow O(n) \quad \rightarrow O(n^2)$

*Regla 3: Ciclos = # veces * orden de la instrucción interna*



Análisis espacial

- $fe(n) = \text{espacio de}(a)$
- $fe(n) = n$ es $O(n)$

49	12	56	90	2	5	11	32	22	7	99	02	35	1
----	----	----	----	---	---	----	----	----	---	----	----	----	---



Ejemplo 02: Multiplicación de matrices de n x n

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \dots & \dots & \dots & \dots \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{pmatrix}$$

$$c_{11} = a_{11} * b_{11} + a_{12} * b_{21} + \dots + a_{1n} * b_{n1}$$

$$c_{12} = a_{11} * b_{12} + a_{12} * b_{22} + \dots + a_{1n} * b_{n2}$$

...

$$c_{21} = a_{21} * b_{11} + a_{22} * b_{21} + \dots + a_{2n} * b_{n1}$$

...

$$c_{nn} = a_{n1} * b_{1n} + a_{n2} * b_{2n} + \dots + a_{nn} * b_{nn}$$

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$



O(n³) ← *for i = 1 to n do*

O(n²) ← *for j = 1 to n do*

~~O(1)~~ ← *C[i,j] = 0;*

O(n) ← *for k = 1 to n do*

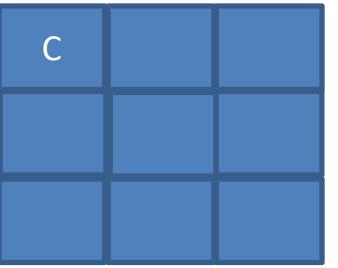
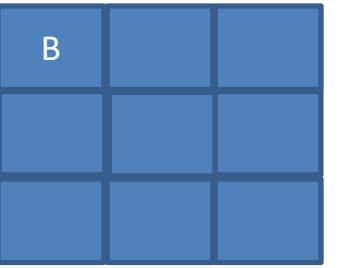
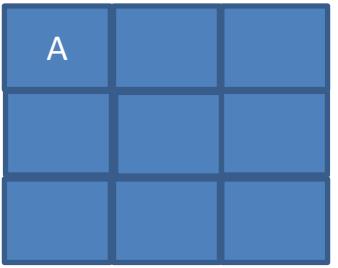
O(1) ← *C[i,j] = C[i,j] + A[i,k]*B[k,j];*



Análisis espacial

- $fe(n) = \text{espacio de}(n) + \text{espacio de } (A, B, C)$
- $fe(n) = 1 + 3n^2$ es $O(n^2)$

3



Ejemplo 03: Algoritmo de Horner

- Sea A un vector de coeficientes y sea $P_n(z) = \sum_{i=0}^n A[i] z^i$, un polinomio de grado n ; evaluado para un argumento real z :
- Encontrar la función complejidad, temporal y espacial, para el algoritmo que evalúa $P_n(z)$.

$$p(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_nx^n,$$



- **Algoritmo:** Método de Horner

- **Tamaño del Problema:** n = grado del polinomio
- **Operación básica:** La multiplicación * (Se realiza un número de veces del mismo orden al tamaño del problema)
- **Caso:** El algoritmo hace el mismo número de operaciones en todos los casos.

```

proc Horner(n,z,A)
{
    polinomio=0;
    for(i=0;i<=n;i++)
    {
        polinomio=polinomio*z + A[n-i];*
    }
}
    
```



polinomio=0;

for($i=0$; $i \leq n$; $i++$)

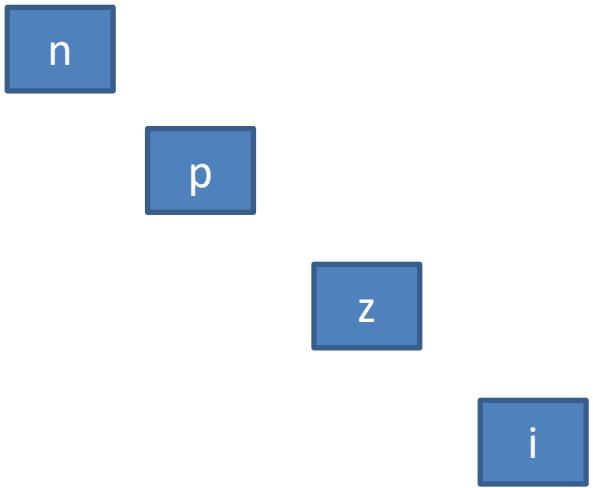
$\leftarrow O(n)$

*polinomio=polinomio*z + A[n-i];* $\leftarrow O(1)$



Análisis espacial

- $fe(n) = \text{espacio de } (n) + \text{espacio de (polinomio)} + \text{espacio de } (z) + \text{espacio de } (i) + \text{espacio de } (A)$
- $fe(n) = 1 + 1 + 1 + 1 + n = 4 + n \text{ es } O(n)$



Ejemplo 04: Fibonacci (Iterativo)

```

anterior = 1;           --> 1 Asignación
actual = 1;            --> 1 Asignación
while (n>2){          --> n-2 + 1 Condicionales
    aux = anterior + actual; --> n-2 Asignaciones
    anterior = actual;      --> n-2 Asignaciones
    actual = aux;          --> n-2 Asignaciones
    n = n - 1;            --> n-2 Asignaciones
}
return (actual);         --> n-2+1 Saltos implicitos
                           --> 1 Asignación

```

$$T(n) = 6n-7=O(n)$$



Ejemplo 05: Búsqueda binaria

BusquedaBinaria(A,n,dato)

inferior=0; ← O(1)

superior=n-1;

while(inferior<=superior)

centro=(superior+inferior)/2;

if(A[centro]==dato) ← O(log₂n)

return centro;

else if(dato < A[centro])

superior=centro-1;

else

inferior=centro+1;

return -1;





Instituto Politécnico Nacional

Escuela Superior de Cómputo



Análisis de algoritmos

Tema 06: Análisis de algoritmos recursivos

M. en C. Edgardo Adrián Franco Martínez

<http://www.eafranco.com>

edfrancom@ipn.mx

[@edfrancom](https://twitter.com/edfrancom) [edgaroadrianfrancocom](https://facebook.com/edgaroadrianfrancocom)



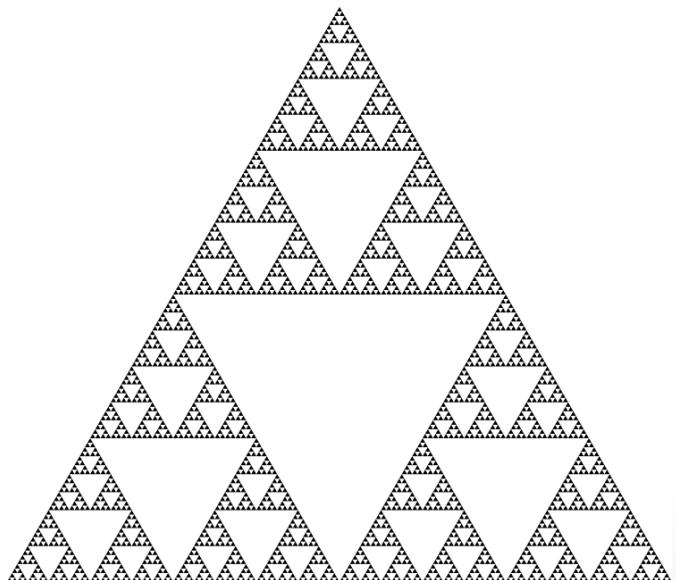
Contenido

- Recursividad
- Ecuaciones en recurrencia
 - Ejemplo 01: Factorial recursivo
 - Ejemplo 02: Fibonacci recursivo
 - Ejemplo 03: Torres de Hanói recursivo
 - Ejemplo 04: Búsqueda binaria recursiva
- Recurrencias homogéneas
 - Ejemplo
- Recurrencias no homogéneas
 - Ejemplo
- Recurrencias no lineales
 - Teorema maestro
 - Ejemplo 01
 - Ejemplo 02



Recursividad

- La **recursividad** es un concepto fundamental en matemáticas y en computación. Es una **alternativa diferente para implementar** estructuras de repetición (**iteración**).
- Se puede usar en toda situación en la cual la solución pueda ser expresada como una secuencia de movimientos, pasos o transformaciones gobernadas por un conjunto de **reglas no ambiguas**.



- La **recursividad** es un recurso muy poderoso que permite expresar **soluciones simples y naturales** a ciertos tipos de problemas. Es importante considerar que no todos los problemas son naturalmente recursivos.
- Un **objeto recursivo** es aquel que **aparece en la definición de si mismo**, así como el que *se llama a sí mismo*.



- La recursividad es un fenómeno que se presenta en muchos problemas de forma natural, delegando la **solución de un problema en la solución de otro más pequeño.**
- El **análisis temporal de un algoritmo recursivo** vendrá en función del **tiempo requerido por la(s) llamada(s) recursiva(s)** que aparezcan en él.
- El **análisis temporal de un algoritmo iterativo** es simple con base en la **operación básica** de este, para los **algoritmos recursivos** nos vamos a encontrar con una dificultad añadida, pues **la función que establece su tiempo de ejecución** viene dada por **una ecuación en recurrencia**, es decir, $T(n) = E(n)$, en donde en la expresión E aparece la propia función T .



Ecuaciones en recurrencia

- Cuando se quiere calcular la demanda de recursos de un algoritmo definido recursivamente, la función complejidad que resulta no está definida sólo en términos del tamaño del problema y algunas constantes, sino en términos de la función complejidad misma.
- Además no es una sola ecuación, dado que existen otras (al menos una) que determinan la cantidad de recursos para los **casos base** de los algoritmos recursivos. Dada esta situación, **para poder obtener el comportamiento del algoritmo, es necesario resolver el sistema recurrente obtenido.**



Ejemplo 01: Factorial recursivo

- Considerando el **producto de num*Factorial (num-1)** como operación básica, y el **costo del caso base como 1 ($T(0) = 1$)**, podemos construir la ecuación recurrente para calcular la complejidad del algoritmo como sigue:

```
int Factorial( int num )
{
    if ( num == 0 )
        return 1;
    else
        return num * Factorial( num - 1 );
}
```

Costo de la multiplicación

Costo de la llamada a Factorial (n-1)

$$T(n) = 1 + T(n - 1)$$

$$T(0) = 1$$

Costo del caso base



Ejemplo 02: Fibonacci recursivo

- Considerando la **suma** Fibonacci (**num-1**) + Fibonacci (**num-2**) **como operación básica**, y el **costo 1** de los **2 casos base** podemos construir la ecuación recurrente para calcular la complejidad del algoritmo.

```
int Fibonacci(int num)
{
    if (num == 0)
        return 0;
    else if (num == 1)
        return 1;
    else
        return Fibonacci(num-1) + Fibonacci (num-2);
}
```

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

$$T(0) = 1$$

$$T(1) = 1$$

Costo de la llamada a Fibonacci(n-1) Costo de la suma Costo de la llamada a Fibonacci(n-2)

Costo de n=0 y n=1



Ejemplo 03: Torres de Hanói recursivo

- Considerando la operación `Mover_de(Src, Dst)` como operación básica, y tomado un **coste de 0 cuando N=0**, podemos construir la ecuación recurrente para calcular la complejidad del algoritmo.

```
Hanoi(N, Src, Aux, Dst)
{
    if(n>0)
    {
        Hanoi(N - 1, Src, Dst, Aux);
        Mover_de(Src,Dst);
        Hanoi(N - 1, Aux, Src, Dst);
    }
    return;
}
```

$$T(n) = T(n - 1) + 1 + T(n - 1)$$

$$T(0) = 0$$



Ejemplo 04: Búsqueda binaria recursiva

- Considerando la **operación como operación básica las comparaciones**, y tomado un **coste de 0 cuando Tam(numeros[])=0**, podemos construir la ecuación recurrente para calcular la complejidad del algoritmo.

```
int BusquedaBinaria(int num_buscado, int numeros[], int inicio, int centro, int final)
{
    if (inicio>final)
        return -1;
    else if (num_buscado == numeros[centro])
        return centro;
    else if (num_buscado < numeros[centro])
        return BusquedaBinaria(num_buscado,numeros,inicio,(int)((inicio+centro-1)/2),centro-1);
    else
        return BusquedaBinaria(num_buscado,numeros,centro+1,(int)((final+centro+1)/2),final);
}
```

$$T(n) = 3 + T(n/2)$$

$$T(0) = 0$$



Ejemplo 05: Merge-sort

- Sea A un arreglo de n elementos y p, r índices del rango a ordenar.

```
Merge-Sort(a, p, r)
{
  if ( p < r )
  {
    q = parteEntera((p+r)/2);
    Merge-Sort(a, p, q);
    Merge-Sort(a, q+1,r);
    Merge(a, p, q, r);
  }
}
```

$$T(n) = 2T(n/2) + n$$

$$T(1) = 1$$



Recurrencias homogéneas

- Son de la forma:

$$a_0 T(n) + a_1 T(n - 1) + a_2 T(n - 2) + \dots + a_k T(n - k) = 0$$

- Donde los coeficientes a_i son números reales, y k es un número natural entre 1 y n .
- Para eliminar la recurrencia se buscan términos que sean combinaciones de funciones exponenciales de la forma:

$$T(n) = c_1 p_1(n) r_1^n + c_2 p_2(n) r_2^n + \dots + c_k p_k(n) r_k^n = \sum_{i=1}^k c_i p_{i1}(n) r_i^n$$

- Donde los valores c_1, c_2, \dots, c_n y r_1, r_2, \dots, r_n son números reales, y $p_1(n), \dots, p_k(n)$ son polinomios en n con coeficientes reales.



$$a_0 T(n) + a_1 T(n - 1) + a_2 T(n - 2) + \dots + a_k T(n - k) = 0$$

- Para resolverlas haremos el cambio $x^k = T(n)$, con lo cual obtenemos la **ecuación característica** asociada:

$$a_0 x^k + a_1 x^{k-1} + a_2 x^{k-2} + \dots + a_k = 0$$

- Llamemos r_1, r_2, \dots, r_k a sus raíces, ya sean reales o complejas. Dependiendo del orden de multiplicidad de tales raíces, pueden darse los siguientes casos:
 - *Caso 1: Raíces distintas*
 - *Caso 2: Raíces con multiplicidad mayor que 1*



Caso 1: Raíces distintas

- Si todas las raíces de la ecuación característica son distintas, esto es, $r_i \neq r_j$ si $i \neq j$, entonces la solución de la ecuación en recurrencia viene dada por la expresión:

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

- Donde los coeficientes c_i se determinan a partir de las condiciones iniciales.



Caso 2: Raíces con multiplicidad mayor que 1

Supongamos que alguna de las raíces (p.e. r_1) tiene multiplicidad $m > 1$. Entonces la ecuación característica puede ser escrita en la forma:

$$(x - r_1)^m(x - r_2) \dots (x - r_{k-m+1})$$

- En cuyo caso la solución de la ecuación en recurrencia viene dada por la expresión:

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

- Donde los coeficientes c_i se determinan a partir de las condiciones iniciales.



- Este caso puede ser generalizado, si r_1, r_2, \dots, r_k son las raíces de la ecuación característica de una ecuación en recurrencia homogénea, cada una de multiplicidad m_i , esto es, si la ecuación característica puede expresarse como:

$$(x - r_1)^{m_1}(x - r_2)^{m_2} \dots (x - r_k)^{m_k} = 0$$

- Entonces la solución a la ecuación en recurrencia viene dada por la expresión:

$$T(n) = \sum_{i=1}^{m_1} c_{1i} n^{i-1} r_1^n + \sum_{i=1}^{m_2} c_{2i} n^{i-1} r_2^n + \dots + \sum_{i=1}^{m_k} c_{ki} n^{i-1} r_k^n$$

- Donde los coeficientes c_i se determinan a partir de las k condiciones iniciales.



Recurrencias homogéneas (Ejemplo)

- Se tiene la siguiente ecuación de recurrencia

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

$$T(k) = k \text{ para } k = 0, 1, 2$$

- Reordenando términos

$$T(n) - 5T(n-1) + 8T(n-2) - 4T(n-3) = 0$$

- Haciendo el cambio $x^3 = T(n)$ obtenemos su ecuación característica $x^3 - 5x^2 + 8x - 4 = 0$, lo que es igual a $(x - 2)^2(x - 1) = 0$, por lo tanto: $r_1 = 2, r_2 = 2$ y $r_3 = 1$

$$T(n) = c_1 2^n + c_2 n 2^n + c_3 1^n$$

- De las condiciones iniciales obtenemos:

$$c_1 = 2, c_2 = -\frac{1}{2} \text{ y } c_3 = -2$$

$$T(n) = 2^{n+1} - n 2^{n-1} - 2 \in O(n 2^n)$$



Recurrencias no homogéneas

- Son de la forma:

$$a_0 T(n) + a_1 T(n - 1) + a_2 T(n - 2) + \dots + a_k T(n - k) = b^n p(n)$$

- Donde los coeficientes a_i y b son números reales, y $p(n)$ es un polinomio en n de grado d .

- Para resolver este tipo de ecuaciones generalmente se deben manipular para llegar a una ecuación homogénea. Una formula general para resolvlerla es mediante la ecuación característica:

$$(a_0 x^k + a_1 x^{k-1} + a_2 x^{k-2} + \dots + a_k)(x - b)^{d+1} = 0$$

- Lo que nos lleva a aplicar el método para las recurrencias homogéneas.



- Generalizando este proceso, si tenemos una ecuación de la forma:

$$a_0 T(n) + a_1 T(n - 1) + a_2 T(n - 2) + \dots + a_0 T(n - k) = b_1^n p_1(n) + b_2^n p_2(n) + \dots + b_s^n p_s(n)$$

- Donde los coeficientes a_i y b_j son números reales, y $p_j(n)$ son polinomios en n de grado d_j

- La ecuación característica es:

$$(a_0 x^k + a_1 x^{k-1} + a_2 x^{k-2} + \dots + a_k)(x - b_1)^{d_1+1}(x - b_2)^{d_2+1} \dots (x - b_s)^{d_s+1} = 0$$



Recurrencias no homogéneas (Ejemplo)

- Se tiene la siguiente ecuación de recurrencia

$$T(n) = 5T(n-1) - 8T(n-2) + 4T(n-3) + 2^n 3n, n \geq 2$$

$$T(k) = k \text{ para } k = 0, 1, 2$$

- Reordenando términos

$$T(n) - 5T(n-1) + 8T(n-2) - 4T(n-3) = 2^n 3n$$

- Haciendo el cambio $x^3 = T(n)$, $b = 2$ y $d = 1$ obtenemos su ecuación característica $(x^3 - 5x^2 + 8x - 4)(x - 2)^2 = 0$, lo que es igual a $(x - 2)^2(x - 1)(x - 2)^2 = 0$, por lo tanto: $r_1 = 2$, $r_2 = 2$, $r_3 = 1$, $r_4 = 2$ y $r_5 = 2$
- Si $c_4 \neq 0$

$$T(n) = c_1 2^n + c_2 n 2^n + c_3 n^2 2^n + c_4 n^3 2^n + c_5 1^n \in O(n^3 2^n)$$



```

int Factorial( int num )
{
    if ( num == 0 )
        return 1;
    else
        return num * Factorial( num - 1 );
}
    
```

$$n! = \begin{cases} 1 & \text{si, } n = 0 \\ (n-1)! \times n & \text{si, } n > 0 \end{cases}$$

$$\begin{aligned} T(n) &= 1 + T(n-1); n > 0 \\ T(0) &= 1 \end{aligned}$$

- Reordenando términos

$$T(n) - T(n-1) = 1$$

- Haciendo el cambio $x^{k=1} = T(n)$, $b = 1$ y $d = 0$ obtenemos su ecuación característica $(x - 1)(x - 1) = 0$, lo que es igual a $(x - 1)^2 = 0$, por lo tanto: $r_1 = 1$ y $r_2 = 1$.

$$T(n) = c_1 + c_2 n$$

- Si $T(0) = 1$ y $T(1) = 2$ $c_1 = 1$, $c_2 = 1$

(21)

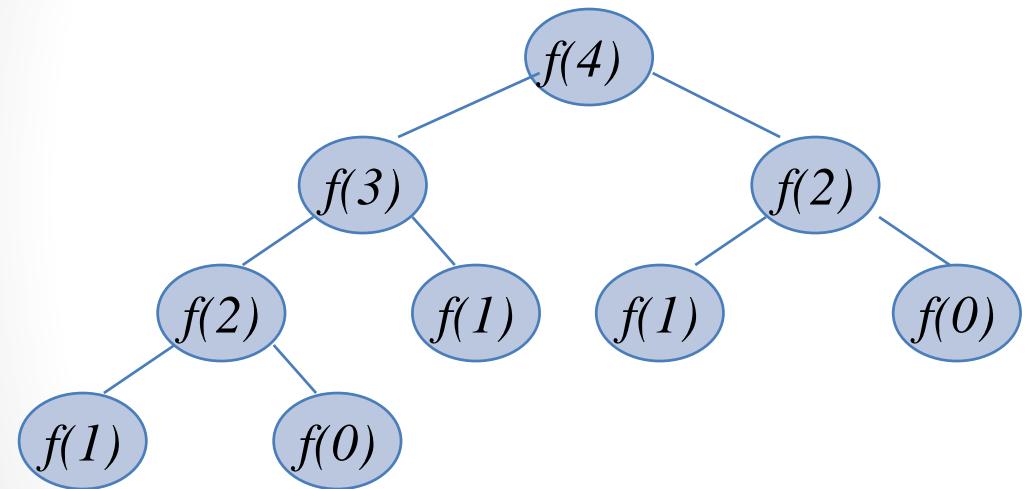
$$T(n) = 1 + n \in O(n)$$



Ejemplo 02: Fibonacci recursivo

```
Function fibonacci (n:int): int;
if (n=0) return 0;
else if (n=1) return 1;
else return fibonacci(n-1) + fibonacci(n-2);
```

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



¿Cuántas operaciones básicas se requieren para calcular:

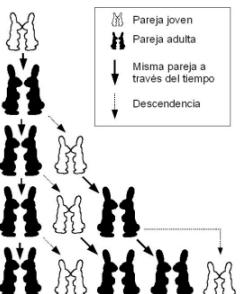
$f(4)$? $\rightarrow 9$

$f(3)$? $\rightarrow 5$

$f(2)$? $\rightarrow 3$

$f(1)$? $\rightarrow 1$

$f(0)$? $\rightarrow 1$



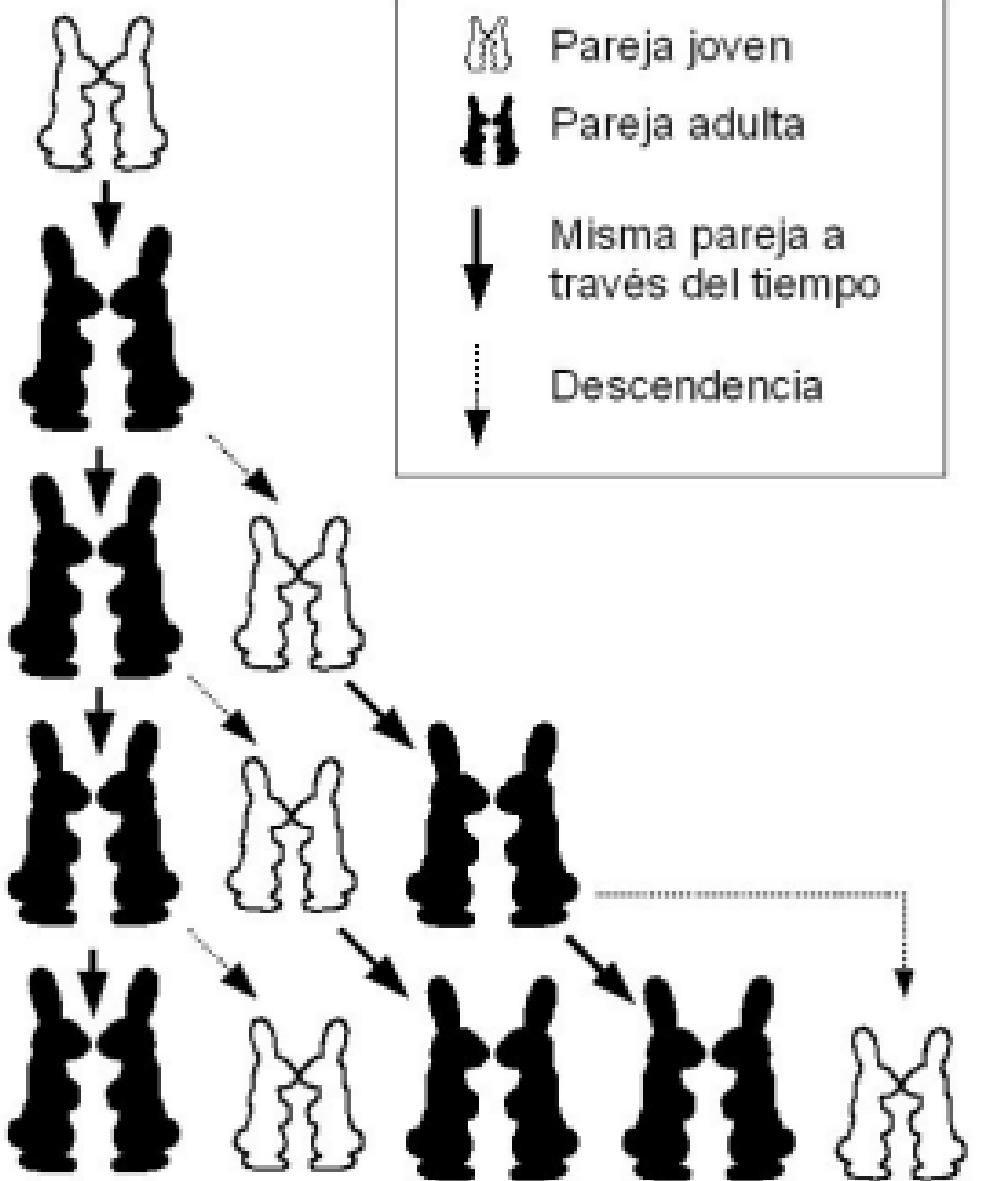
Relación:

El término $T(n)$ requiere $T(n-1)+1+T(n-2)$ términos para calcularse.

$$T(0)=1$$

$$T(1)=1$$





- La ecuación en recurrencia definida para la sucesión de Fibonacci con las condiciones iniciales:

$$T(0) = 1 \text{ y } T(1) = 1.$$

$$T(n) = T(n-1) + T(n-2) + 1, n \geq 2$$

- Reordenando términos

$$T(n) - T(n-1) - T(n-2) = 1$$

- Haciendo el cambio $x^{k=1} = T(n)$, $b = 1$ y $d = 0$ obtenemos su ecuación característica

- $(x^2 - x - 1)(x - 1) = 0$, cuyas raíces son:

$$r_1 = \frac{1+\sqrt{5}}{2}, \quad r_2 = \frac{1-\sqrt{5}}{2}, \quad r_3 = 1$$

- Y por lo tanto $T(n) = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^n + c_3$



- Para calcular las constantes c_1 , c_2 y c_3 necesitamos utilizar las condiciones iniciales de la ecuación original, obteniendo:

$$T(0) = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^0 + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^0 + c_3 = c_1 + c_2 + c_3 = 1$$

$$T(1) = c_1 \left(\frac{1 + \sqrt{5}}{2}\right)^1 + c_2 \left(\frac{1 - \sqrt{5}}{2}\right)^1 + c_3 = 1$$

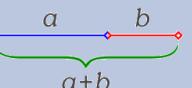
$$T(2) = c_1 \left(\frac{1 + \sqrt{5}}{2}\right)^2 + c_2 \left(\frac{1 - \sqrt{5}}{2}\right)^2 + c_3 = 3$$

$$T(n) = c_1 \left(\frac{1 + \sqrt{5}}{2}\right)^n + c_2 \left(\frac{1 - \sqrt{5}}{2}\right)^n + c_3$$

- Si c_1 o c_2 no valen 0 se tendrá $\in O(c^n)$

El número áureo surge de la división en dos de un segmento guardando las siguientes proporciones: La longitud total $a+b$ es al segmento más largo a , como a es al segmento más corto b .

$$\varphi = \left(\frac{1+\sqrt{5}}{2}\right) \approx 1.6180339887498948948482045868343656... \text{ (número áureo)}$$

$$\frac{a+b}{a} = \frac{a}{b}$$


A diagram showing a horizontal blue line segment divided into two parts, a (the longer part) and b (the shorter part). The point where they meet is marked with a red dot. Below the line, a green bracket underlines the entire length a+b. Above the line, a blue bracket underlines the longer part a. A red bracket underlines the shorter part b.



Ejemplo 03: Torres de Hanói

Solve(N, Src, Aux, Dst)

if N is 0

exit

else

Solve(N - 1, Src, Dst, Aux) T(n-1)

Move from Src to Dst 1

Solve(N - 1, Aux, Src, Dst) T(n-1)

- $T(n) = 2T(n-1) + 1 \text{ si } n > 0$

- Acomodando los términos se tiene:

$$T(n) - 2T(n-1) = 1$$

$$(a_0x^k + x^{k-1} + a_2x^{k-2} + \dots + a_k)(x - b)^{d+1} = 0$$

- No homogénea con $b = 1, p(n) = 0$ y $d = 0$;

**Operación básica
movimiento de cero o
un disco**

$$T(0) = 0$$

$$T(1) = 1$$



- $(a_0x^k + x^{k-1} + a_2x^{k-2} + \dots + a_k)(x - b)^{d+1} = 0$
- $T(n) - 2T(n-1) = 1; b = 1, p(n) = 0$ y $d = 0$
- Su Ecuación característica es:

$$(x - 2)(x - 1) = 0$$

$$x_1 = 2$$

$$x_2 = 1$$

Sustituir como raíces distintas

- Por lo tanto

$$T(n) = c_1 2^n + c_2 1^n \in \Theta(2^n)$$

Con los casos iniciales

$$T(0) = 0$$

$$T(1) = 1$$

Encontramos

$$c_1 = 1$$

$$c_2 = -1$$

$$T(n) = 2^n - 1 \in \Theta(2^n)$$



Recurrencias no lineales

- El **teorema maestro** es un método matemático que se usa para resolver ciertos casos particulares de ecuaciones de recurrencia como la siguiente:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- Donde los coeficientes $a \geq 1$ y $b > 1$ y $\left(\frac{n}{b}\right)$ puede ser tomada como $\left[\frac{n}{b}\right]$ o $\left\lceil \frac{n}{b} \right\rceil$ entonces $T(n)$ tiene los siguientes límites asintóticos :
 1. Si $f(n) = O(n^{\log_b a - \varepsilon})$ para alguna $\varepsilon > 0$, entonces
 - $T(n) = \Theta(n^{\log_b a})$
 2. Si $f(n) = \Theta(n^{\log_b a})$, entonces
 - $T(n) = \Theta(n^{\log_b a} \lg(n))$
 3. Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ para alguna $\varepsilon > 0$ y si $af(n/b) \leq cf(n)$, para alguna constante $c < 1$ y suficientemente grandes n entonces
 - $T(n) = \Theta(f(n))$



Logaritmos

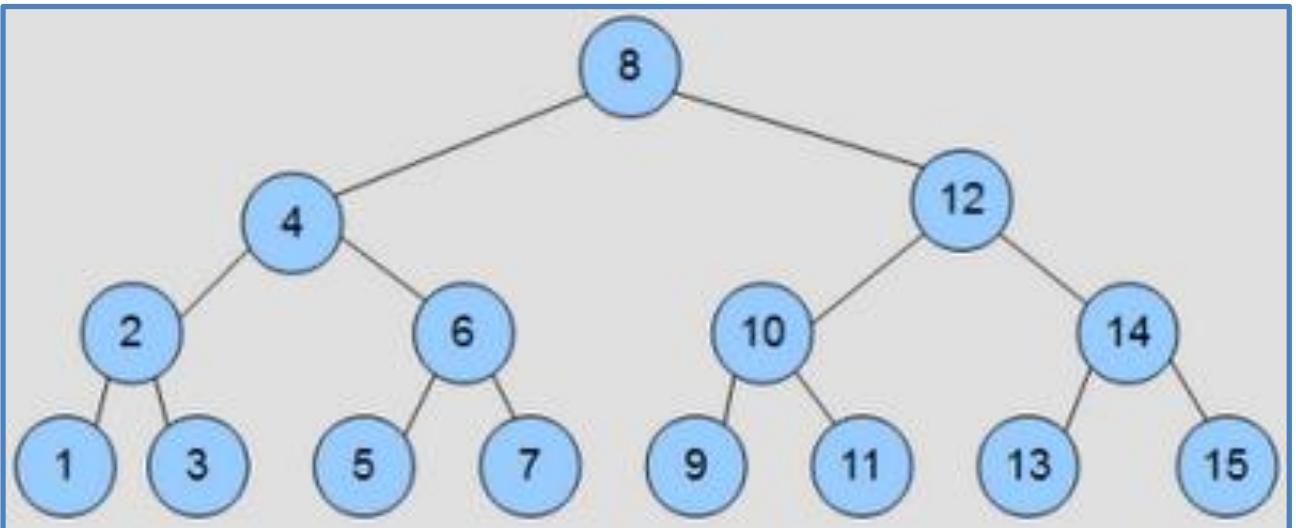
- Un logaritmo es un exponente o potencia, a la que un número fijo (llamado base), se ha de elevar para dar un cierto número.
- Entonces, el logaritmo es la función inversa de la función exponente.

$$\log_a c = b$$

$$a^b = c$$



- $\log_2(n)$ = el exponente a la que 2 se ha de elevar para obtener n.
 - P.g: $\log_2(8) = 3$ (porque $2^3 = 8$)
 - P.g: $\log_2(1024) = 10$ (porque $2^{10} = 1024$)



Propiedades de los logaritmos

1. El logaritmo de la base siempre es igual a uno, es decir:

$$\log_a a = 1$$

2. El logaritmo de 1 en cualquier base es siempre igual a cero:

$$\log_a 1 = 0$$

3. El logaritmo de un producto es igual a la suma de los logaritmos de sus factores:

$$\log_a (b \cdot c) = \log_a b + \log_a c$$



4. El logaritmo de una fracción es igual a la resta del logaritmo del numerador menos el logaritmo del denominador.

$$\log_a (\textcolor{red}{b}/\textcolor{blue}{c}) = \log_a \textcolor{green}{b} - \log_a \textcolor{red}{c}$$

5. El logaritmo de una potencia es igual a la potencia multiplicando al logaritmo de la base de la potencia:

$$\log_a \textcolor{red}{b}^{\textcolor{blue}{c}} = \textcolor{blue}{c} \log_a \textcolor{green}{b}$$

6. El logaritmo de la base elevado a una potencia es igual a la potencia.

$$\log_a \textcolor{red}{a}^{\textcolor{green}{b}} = \textcolor{blue}{b}$$



7. **Cambio de base de logaritmo:** El logaritmo en base a un número es igual a la fracción entre el logaritmo del primer número con base en un tercer número y el logaritmo del segundo número con base en un tercer número.

$$\log_a b = \log_c b / \log_c a$$

8. Un número elevado al logaritmo con base en el mismo número, es igual al número del logaritmo.

$$a^{\log_a b} = b$$



Ejemplo 01 Uso del teorema maestro

- Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ para algún $\varepsilon > 0$, y
 si $a f(n/b) \leq c f(n)$ para
 alguna constante $c < 1$ y suficientemente grandes n ,
 $\Rightarrow T(n) = \Theta(f(n))$.

$$T(n) = 9T(n/3) + n$$

$$\begin{aligned} T(n) &= 9T(n/3) + n \\ a = 9, \quad b = 3, \quad f(n) = n &\Rightarrow n^{\log_b a} = n^{\log_3 9} = \Theta(n^2) \\ \therefore f(n) &= O(n^{\log_3 9 - \varepsilon}), \varepsilon = 6 \end{aligned}$$

$$\therefore T(n) = \Theta(n^2)$$



Ejemplo 02 Uso del teorema maestro

1. Si $f(n) = O(n^{\log_b a - \varepsilon})$ para algún $\varepsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. Si $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg n)$
3. Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ para algún $\varepsilon > 0$, y
si $a f(n/b) \leq c f(n)$ para
alguna constante $c < 1$ y suficientemente grandes n ,
 $\Rightarrow T(n) = \Theta(f(n))$.

$$T(n) = T(2n/3) + 1$$

$$T(n) = T(2n/3) + 1 \Rightarrow \text{Caso 2}$$

$$a = 1, \quad b = 3/2, \quad f(n) = 1 \quad \Rightarrow \quad n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$$

$$\therefore T(n) = \Theta(\lg n)$$



Ejemplo 03 Uso del teorema maestro

1. Si $f(n) = O(n^{\log_b a - \varepsilon})$ para algún $\varepsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. Si $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg n)$
3. Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ para algún $\varepsilon > 0$, y
 si $a f(n/b) \leq c f(n)$ para
 alguna constante $c < 1$ y suficientemente grandes n ,
 $\Rightarrow T(n) = \Theta(f(n))$.

$$T(n) = 2T(n/2) + n$$

$$T(n) = 2T(n/2) + n$$

$a = 2, \quad b = 2, \quad f(n) = n \quad \Rightarrow \quad n^{\log_b a} = n^{\log_2 2} = n^1 = n$
 $f(n) = \Theta(n^{\log_b a}) \quad \Rightarrow \quad \text{Caso 2}$

$$T(n) = \Theta(n \lg(n))$$



Ejemplo 04 Uso del teorema maestro

1. Si $f(n) = O(n^{\log_b a - \varepsilon})$ para algún $\varepsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. Si $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg n)$
3. Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ para algún $\varepsilon > 0$, y
 si $a f(n/b) \leq c f(n)$ para
 alguna constante $c < 1$ y suficientemente grandes n ,
 $\Rightarrow T(n) = \Theta(f(n))$.

$$T(n) = 2T(n/2) + n^2$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

$$a = 2, \quad b = 2, \quad f(n) = n^2 = \Theta(n^2). \quad \Rightarrow \quad f(n) = \Omega(n^{\log_b a + \varepsilon})$$

$$n^{\log_b a + 2} = n^{\log_2 4} = n^2$$

\Rightarrow Caso 3 \Rightarrow si $f(n/b) \leq c f(n) \Rightarrow (n/2)^2 \leq c n^2$ para alguna $c < 1 \Rightarrow c=1/2$
 $T(n) = \Theta(f(n))$.

$$T(n) = \Theta(n^2)$$



Otra forma de ver el teorema maestro

Teorema 1.1 Sean $a, b, c, d \in \mathbb{R}$, $a \geq 1$, $b > 1$, $d > 0$, y

$$T(n) = \begin{cases} d & n = 1 \\ aT(n/b) + n^c & n > 1 \end{cases}$$

Donde n/b puede interpretarse como $\lfloor n/b \rfloor$ o $\lceil n/b \rceil$. Entonces,

$$T(n) = \begin{cases} O(n^c) & a < b^c \\ \Theta(n^c \lg n) & a = b^c \\ \Theta(n^{\log_b a}) & a > b^c \end{cases}$$

