Definiciones de Algoritmo

- Un algoritmo es la expresión de una secuencia precisa de operaciones que conduce a la resolución de un problema.

- Sistema de reglas que permiten obtener una salida específica a partir de una entrada específica. Cada paso debe estar definido exactamente, de forma que pueda traducirse a un lenguaje de programación.

Propiedades de un algoritmo

- Debe ser finito,
- Toda regla debe definir perfectamente la acción a desarrollar,
- Todos sus pasos deben ser simples y tener un orden definido,
- Un algoritmo no debe resolver un solo problema particular sino una clase de problemas,
- Un algoritmo debe ser **eficiente** y rápido.

Objetivo

En el presente curso vamos a centrar gran parte de nuestra atención en dos aspectos muy importantes de los algoritmos, como son su ´***diseño***´ y el estudio de su ´***eficiencia***´.

Objetivo

El **Diseño** se refiere a la búsqueda de métodos o procedimientos, secuencias finitas de instrucciones adecuadas al dispositivo que disponemos, que permitan resolver el problema.

La **Eficiencia** nos permite medir de alguna forma el costo (en tiempo y recursos) que consume un algoritmo para encontrar la solución y nos ofrece la posibilidad de comparar distintos algoritmos que resuelven un mismo problema.

Complejidad (eficiencia) de Algoritmos

- Tenemos un arreglo de naturales positivos de m posiciones, n de ellas ocupadas.
- Queremos implementar la función Alternar (nat x) que si encuentra a x lo elimina y si no lo agrega.
- Veamos dos opciones:

  1. Recorro el arreglo de la posición 1 a la n, si el elemento no esta, lo agrego en la posición n + 1. Si esta, comprimo el arreglo moviendo todos los elementos una posición "hacia atrás".
  2. Recorro el arreglo desde la posición 1 hasta haber pasado n elementos no anulados. Si no esta, lo agrego en una de las posiciones libres. Si esta, marco la posición como anulada pero no comprimo.
- Cual de los dos es mejor algoritmo? Con mas precisión aun: cual tarda menos?

Comparando algoritmos

- Hay varias formar de categorizar los algoritmos para tratar de

responder a la pregunta de cual es mejor.

  - Claridad.
  - Facilidad de programación.
  - Tiempo de ejecución.
  - Necesidad de almacenamiento (memoria).
- Probablemente para ciertos criterios algunos sean mejores y otros peores.
- En esta materia nos vamos a preocupar por los dos últimos.

Cómo calculamos esos parámetros?

Empírico vs. teórico

Análisis teórico de algoritmos

- El interés de comparar algoritmos surge para problemas grandes. Para problemas suficientemente chicos no suele ser tan importante que algoritmo se utiliza.
- Que significa grande o chico? Debemos dar alguna medida del tamaño del problema.
- A los efectos del análisis de algoritmos tomaremos como medida del problema al tamaño de la entrada, concepto que precisaremos mas adelante.
- Dado que para distintos problemas el concepto de grande varía, realizaremos un análisis asintótico: si n es el tamaño de la entrada, nos preguntaremos como se comporta el algoritmo cuando n -> infinito.

Análisis teórico de algoritmos

- Medida de tiempo: numero de pasos o instrucciones que ejecuta la maquina de referencia.
- Medida de espacio: cantidad de posiciones de memoria en la maquina de referencia.

Contando operaciones

- Ahora bien, dado un algoritmo en particular, como sabemos cuanto tarda?
- Contaremos la cantidad de operaciones elementales (OE).
- Que es una OE?
  - Aquellas que el procesador realiza en una cantidad de tiempo acotada por una constante (que no depende del tamaño de la entrada).
  - Operaciones aritméticas básicas, comparaciones lógicas, transferencia de control, asignaciones de variables de tipos básicos, etc.
- Llamaremos t(a; d) al numero de OE del algoritmo a para el conjunto de datos de entrada d. Abusaremos la notación omitiendo alguno de los parámetros, según convenga.

Cómo se calcula?

- Vamos a definir que t(OE) = 1.
- Además, t(o1; o2) = t(o1) + t(o2) (el tiempo de la ejecución secuencial es la suma de los tiempos).

Contando operaciones

Tamaño de la entrada:

número de componentes sobre los que se va a ejecutar el algoritmo.

Por ejemplo:

- la dimensión de un vector a ordenar
- el tamaño de dos matrices a multiplicar

Contando operaciones

La unidad de tiempo a la que debe hacer referencia estas medidas de eficiencia no se puede expresar en segundos o en otra unidad de tiempo concreta, pues no existe una computadora estándar al que puedan hacer referencia todas las medidas.

Contando operaciones

Denotaremos por T(n) el tiempo de ejecución de un algoritmo para una entrada de tamaño n.

Teóricamente T(n) debe indicar el número de instrucciones ejecutadas por una computadora ideal.

Contando operaciones

Debemos buscar por tanto medidas simples y abstractas.

Para ello es necesario acotar de alguna forma la diferencia que se puede producir entre distintas implementaciones de un mismo algoritmo, ya sea de:

- mismo código ejecutado por dos máquinas de distinta velocidad,
- dos códigos que implementen el mismo método.

Principio de la invarianza

Esta diferencia es la que acota el siguiente principio:

Principio de Invarianza

Dado un algoritmo y dos implementaciones suyas I_1 e I_2, que tardan $T_1(n)$ y $T_2(n)$ respectivamente, el *Principio de Invarianza* afirma que:

Existe:

- una constante real $c > 0$, y
- un número natural n_0

tales que para todo $n >= n_0$ se verifica:

$$T_1(n) <= cT_2(n)$$

Es decir, el tiempo de ejecución de dos implementaciones distintas de un algoritmo dado no va a diferir más que en una constante multiplicativa.

Orden de T(n)

Con esto podemos definir sin problemas que un algoritmo tarda un tiempo ***del orden de*** T(n) si:

- existe una constante real $c > 0$ y
- una implementación *I* del algoritmo que tarda menos que cT(n), para todo *n (*tamaño de la entrada).

Orden de T(n)

Sea T(n) una función

t: N → N+, y tamaño de la entrada n.

Consideremos dos algoritmos distintos para un mismo problema:

- Algoritmo 1: $t(n) = 10^{-4} \times 2^n$

n	10	20	30	38
t(n)	0,1s	2m	>1 día	1 año

- Algoritmo 2: $t(n) = 10^{-2} \times n^3$

n	200	1000
t(n)	1 día	1 año

T(n)

- Vamos a presentar tres cotas para comportamiento asintótico.

- Esto es fundamental. El objetivo del estudio de la complejidad algorítmica es poder establecer estas cotas.

- Tenemos:
  - T(n) es O(g) (O grande), cota superior.
  - T(n) es (g) (omega), cota inferior.
  - T(n) es (g) (theta), orden exacto.

- Idea intuitiva: si para T(n) puedo presentar
  - T(n) es O(g), “se” cuanto va a tardar como máximo.
  - T(n) es (g), “se” cuanto va a tardar como mínimo.
  - T(n) es (g), “se” ambas cosas

T(n) Definición formal

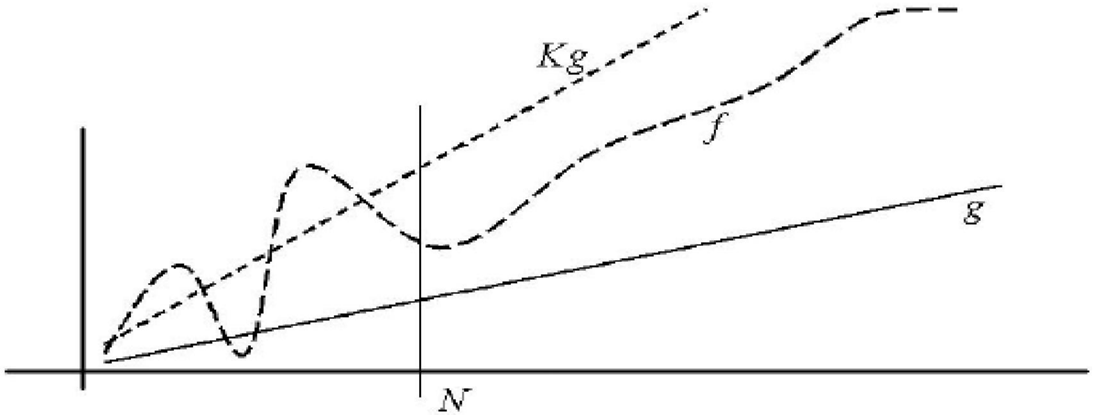
Supongamos una función $\boldsymbol{f}$: N^+ à N^+ que caracterice la eficiencia temporal o espacial de un algoritmo en función de los datos utilizados.

Una notación asintótica define el conjunto de funciones que acotan el crecimiento de $\boldsymbol{f}$.

<u>Definición </u>: Una *asíntota* es una recta o curva que se acerca indefinidamente sin llegar a ser tangente.

T(n) Definición formal

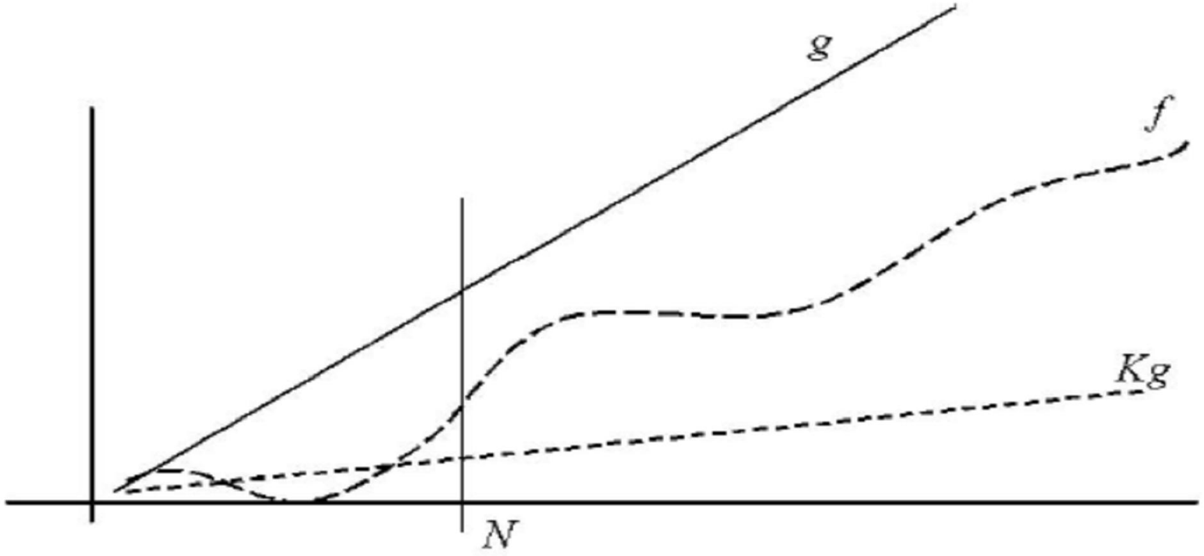
- Definición formal $O(g) =$
  $\{f : \mathbb{N} \rightarrow \mathbb{R}_+ | \exists n_0 > 0, c > 0 \text{ tales que } (\forall n \geq n_0)\ f(n) \leq cg(n)\}$.



- Idea: $O(g)$ define el conjunto de funciones que, a partir de cierto valor n_0, tienen a g multiplicado por un constante c como cota superior.
- Vamos a decir que $T(n) \in O(g(n))$. Abusando la notación, también diremos que $T(n) = O(g(n))$.
- Por ejemplo, "$T(n)$ está en $O(n^2)$" o "el algoritmo tiene $O(log(n))$".

T(n) Definición formal

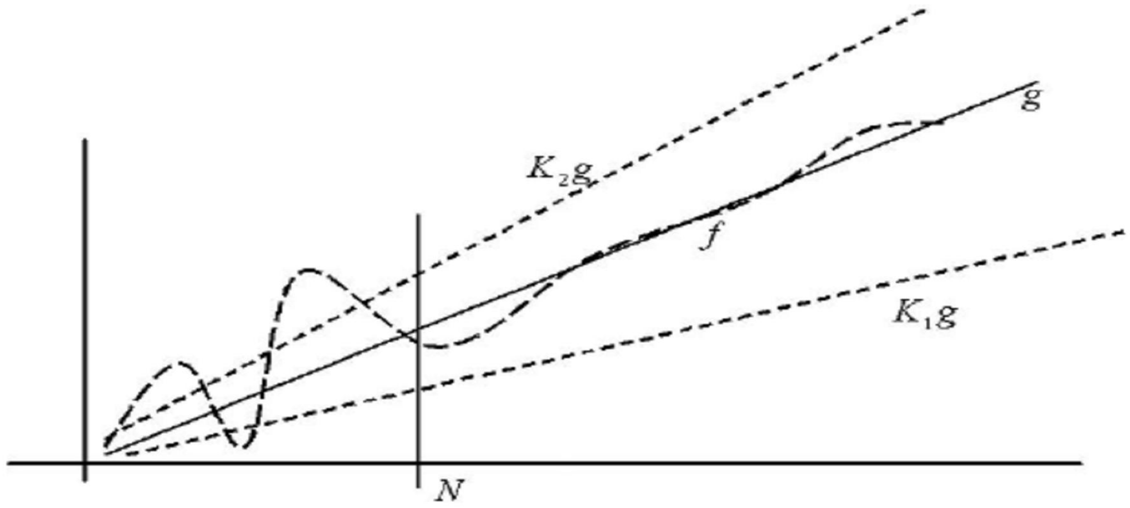
- Definición formal $\Omega(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}_+ | \exists n_0 > 0, c > 0$ tales que $(\forall n \geq n_0)\ f(n) \geq cg(n)\}$.



- Idea: $\Omega(g)$ define el conjunto de funciones que, a partir de cierto valor n_0, tienen a g multiplicado por un constante c como cota inferior.

T(n) Definición formal

- Definición formal $\Theta(g) = \{f : \mathbb{N} \to \mathbb{R}_+ | \exists n_0 > 0, c_1 > 0, c_2 > 0 \text{ tales que } (\forall n \geq n_0)\ c_1 g(n) \leq f(n) \leq c_2 g(n)\}$.



- Idea: $\Theta(g)$ define el conjunto de funciones que, a partir de cierto valor n_0, quedan "ensandwichadas" por g multiplicado por dos constantes c_1 y c_2.
- Además, se cumple que $\Theta(f) = O(f) \cap \Omega(f)$.

T(n) Definición formal

- Notemos que las tres definen conjuntos de funciones matemáticas, y podrán utilizarse independientemente de la complejidad algorítmica.
- Interpretación:
  - f ∈ O(g) signica que f crece, a lo sumo, tanto como g.
  - f ∈ (g) signica que f crece por lo menos como g.
  - f ∈ (g) signica que f crece a la misma velocidad que g.
- En todos los casos, a partir de cierto momento (n0).

- Veamos ejemplos:
  - 100n2 + 300n + 10e20 ∈ O(n2) ⊂ O(n3)
  - 100n2 + 300n + 10e20 ∈ Ω(n2) ⊂ Ω(n)
  - 100n2 + 300n + 10e20 ∈ Θ(n2)

T(n)

- Como se relaciona todo esto con la complejidad de un algoritmo?
- Primero, recordemos lo que habamos dicho.

Tamaño de la entrada

- Cual es la complejidad de multiplicar dos enteros?
- Depende de cual sea la medida del tamaño de la entrada.
- Podrá considerarse que todos los enteros tienen tamaño O(1), pero eso no será útil para comparar este tipo de algoritmos.
- En este caso, conviene pensar que la medida es el logaritmo del numero.
- Si por el contrario estuviésemos analizando algoritmos que ordenan arreglos de enteros, lo que importa no son los enteros en sí, sino cuantos tengamos.
- Entonces, para ese problema, la medida va a decir que todos los enteros miden lo mismo.

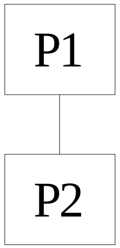
Cómo acotar el tiempo de ejecución?

Tenemos que analizar las estructuras fundamentales en un algoritmo:

- Concatenación
- Selección
- Iteración

Cómo acotar el tiempo de ejecución?

- Concatenación:



$T_1(n) \in O(\ f(n)\ )$

$T_2(n) \in O(\ g(n)\ )$

$T_1(n) + T_2(n)$ es $O(\ \max(\ f(n),g(n)\ )\ )$

<u>Dem</u>:

$T_1(n) <= c_1 f(n) \quad \forall\, n \geq n_1$

$T_2(n) <= c_2 g(n) \quad \forall\, n \geq n_2$

Entonces,

$T_1(n) + T_2(n) <= (c_1 + c_2)\ \max(\ f(n),g(n)\ )\ )$

Donde $c = c_1 + c_2$

y $n_0 = \max(n_1,n_2)$

Cómo acotar el tiempo de ejecución?

- Selección

<div></div>

if<condicion>

<parte_then> → $O(f(n))$

else

<parte_else> → $O(g(n))$

Si no tiene llamadas a funciones (por ej. que no sean de biblioteca), la consideramos O(1).

$T(n) \in O(\max(f(n), g(n)))$

Cómo acotar el tiempo de ejecución?

- Iteración

while<condicion>

<cuerpo_while> grupo de sentencias S1

$$T(n) = t_{cond} + (t_{cond} + t_{s1}) * \#iteraciones$$

Algunos ejemplos sencillos:

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

$T(n)$ es $O(n^2)$??

$(n+1)^2 < = c.n^2$

$n^2+2n+1 <= cn^2$

$C >= 1+ 2/n + 1/n^2$ para $n0=1$ y $c=4$

Algunos ejemplos sencillos:

- $T(n) = n^3 + 2\ n + 5$ es $O(n^3)$???

$n^3 + 2n + 5 <= c\ n^3$

$1 + 2/n^2 + 5/n^3 <= c$ para $c=8$ y $n_0=1$

- $T(n) = 2^n + n^3$ es $O(2^n)$???

$2^n + n^3 <= c.2^n$; $n^3 <= 2^n$

para $c=2$ y $n_0=10$