Análisis de eficiencia de algoritmos

Análisis de eficiencia de algoritmos
Objetivos

- Comprender por qué es necesario analizar eficiencias.
- Comprender el concepto de eficiencia.
- Comprender las notaciones asintóticas.
- Comprender el concepto de operación elemental.
- Saber calcular la eficiencia de un algoritmo.
- Comprender el concepto de principio de invarianza.

Análisis de eficiencia de algoritmos
Comprender por qué es necesario analizar eficiencias

Si disponemos de más de un algoritmo para resolver un problema, **¿con cuál nos quedamos?**

Tenemos que escoger a partir de:

- **Tiempo de ejecución** (a menor, mejor).
- **Memoria consumida** (debe ser <= a la que disponemos).

Hoy día, la memoria es muy barata, de manera que el factor más importante a la hora de escoger un algoritmo es el **tiempo de ejecución** del mismo.

El análisis de eficiencia permite comprender cómo de rápido va a crecer el tiempo de ejecución de un algoritmo conforme aumenta el tamaño del problema.

Análisis de eficiencia de algoritmos
Comprender por qué es necesario analizar eficiencias

Ejemplo, contar número de letras “a” en una cadena:

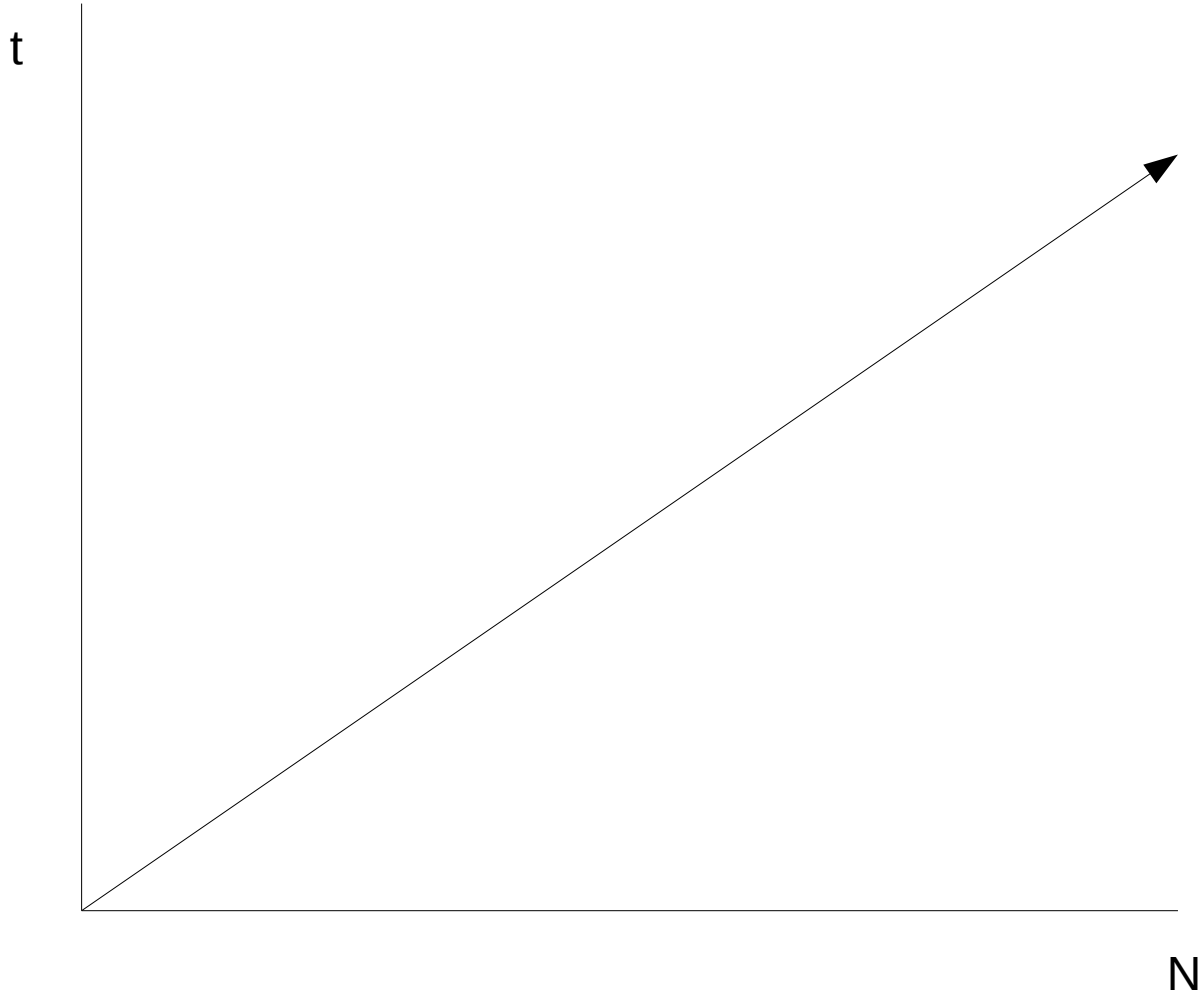
```
char cadena[20];
... //Lectura de la cadena
int i = 0;
int cuantas = 0;
while (cadena[i] != '\0') {
  if (cadena[i] == 'a') {
    cuantas++;
  }
  i++;
}
```

o	v	e	j	a	\0

m	u	r	c	i	e	l	a	g	o	\0

Análisis de eficiencia de algoritmos
Comprender por qué es necesario analizar eficiencias

Ejemplo, contar número de letras “a” en una cadena:



Análisis de eficiencia de algoritmos
Comprender el concepto de eficiencia

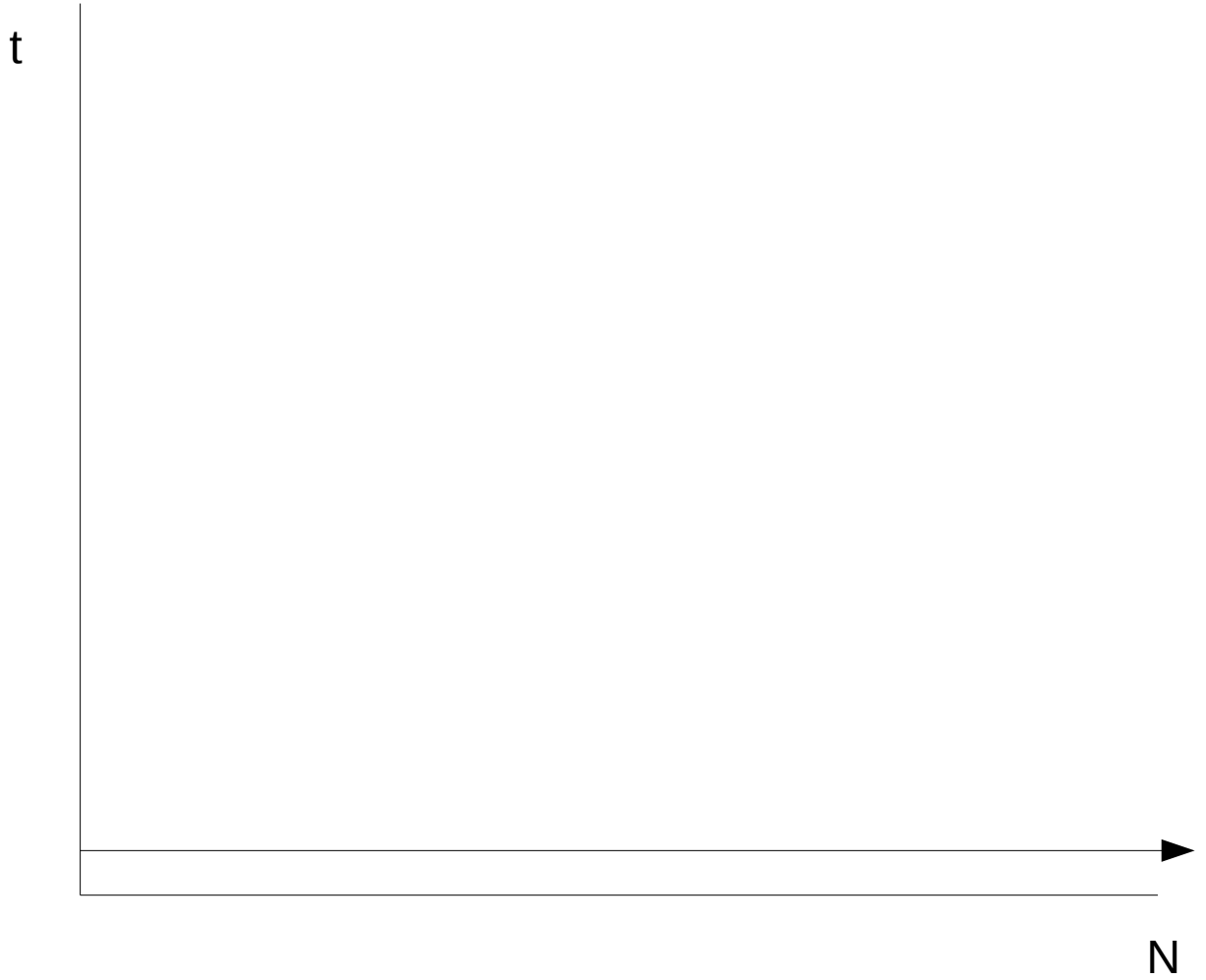
La eficiencia es **la relación** entre el tiempo de ejecución del algoritmo y el tamaño del problema.

La eficiencia **no nos permitirá conocer** el tiempo necesario para la ejecución de un algoritmo para un problema de un tamaño determinado**.**

La eficiencia **nos permitirá conocer** en cómo crecerá el tiempo de ejecución del algoritmo conforme crece el tamaño del problema.

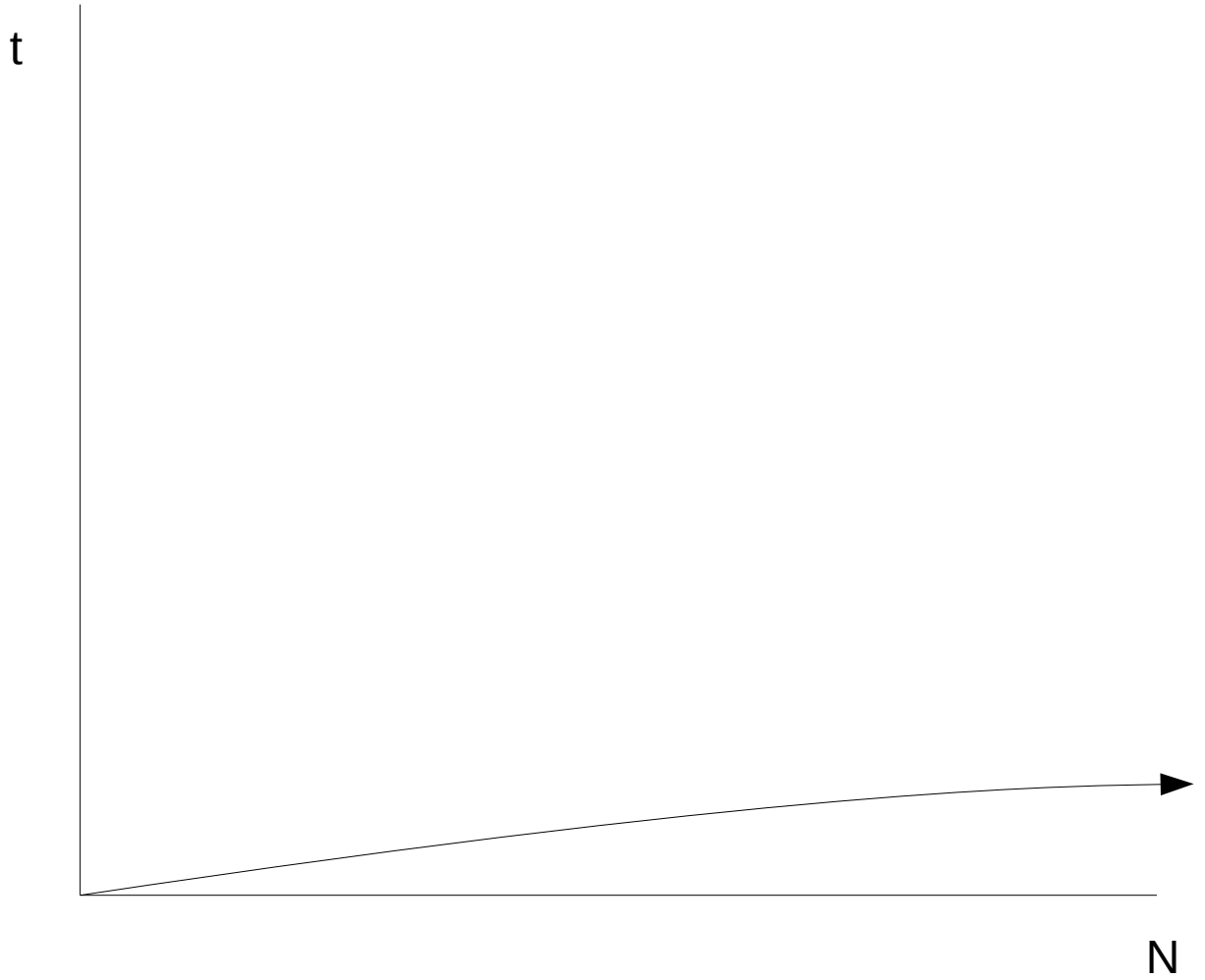
Análisis de eficiencia de algoritmos
Comprender el concepto de eficiencia

Orden constante O(1):



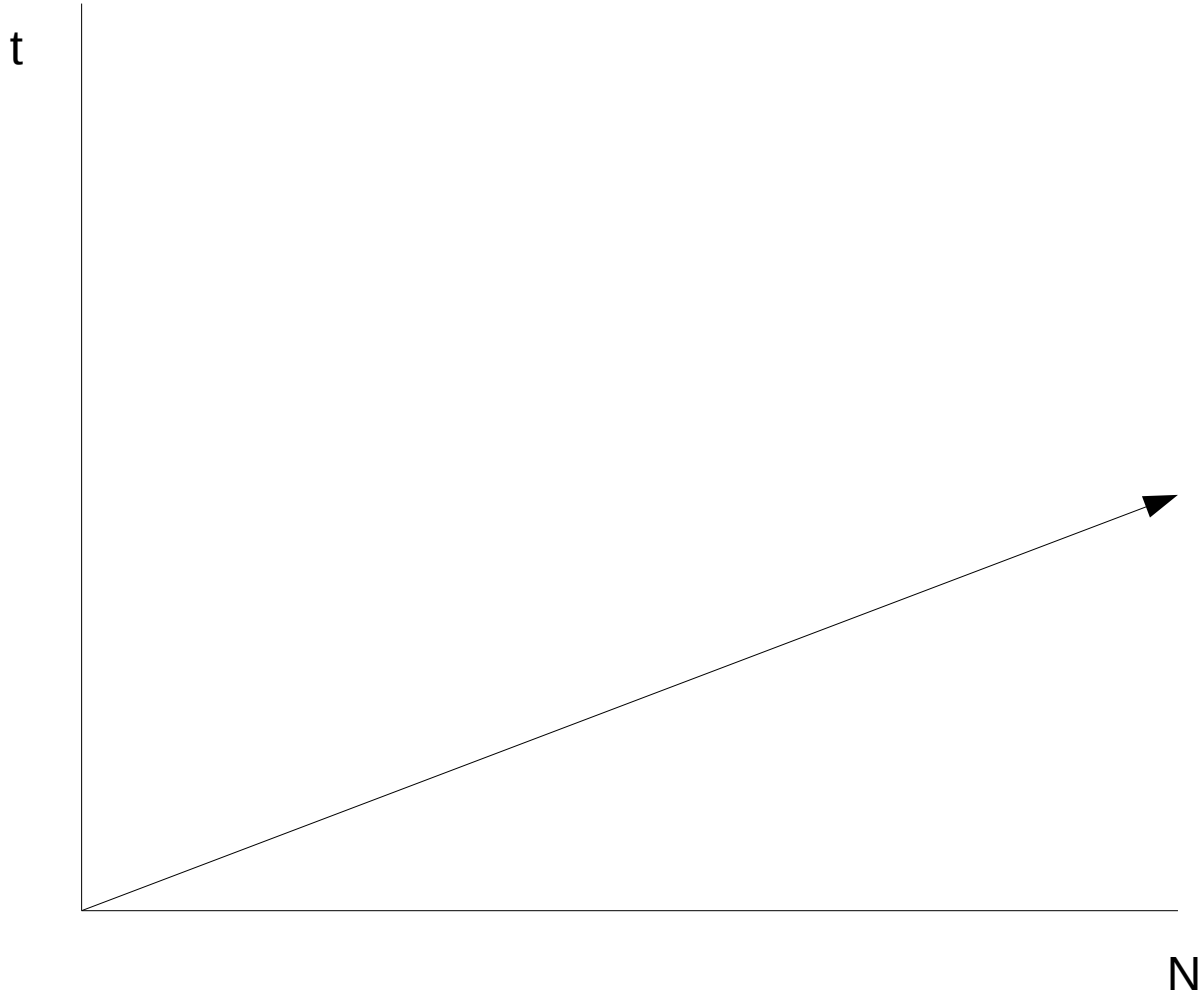
Análisis de eficiencia de algoritmos
Comprender el concepto de eficiencia

Orden logarítmico $O(\log_2 n)$:



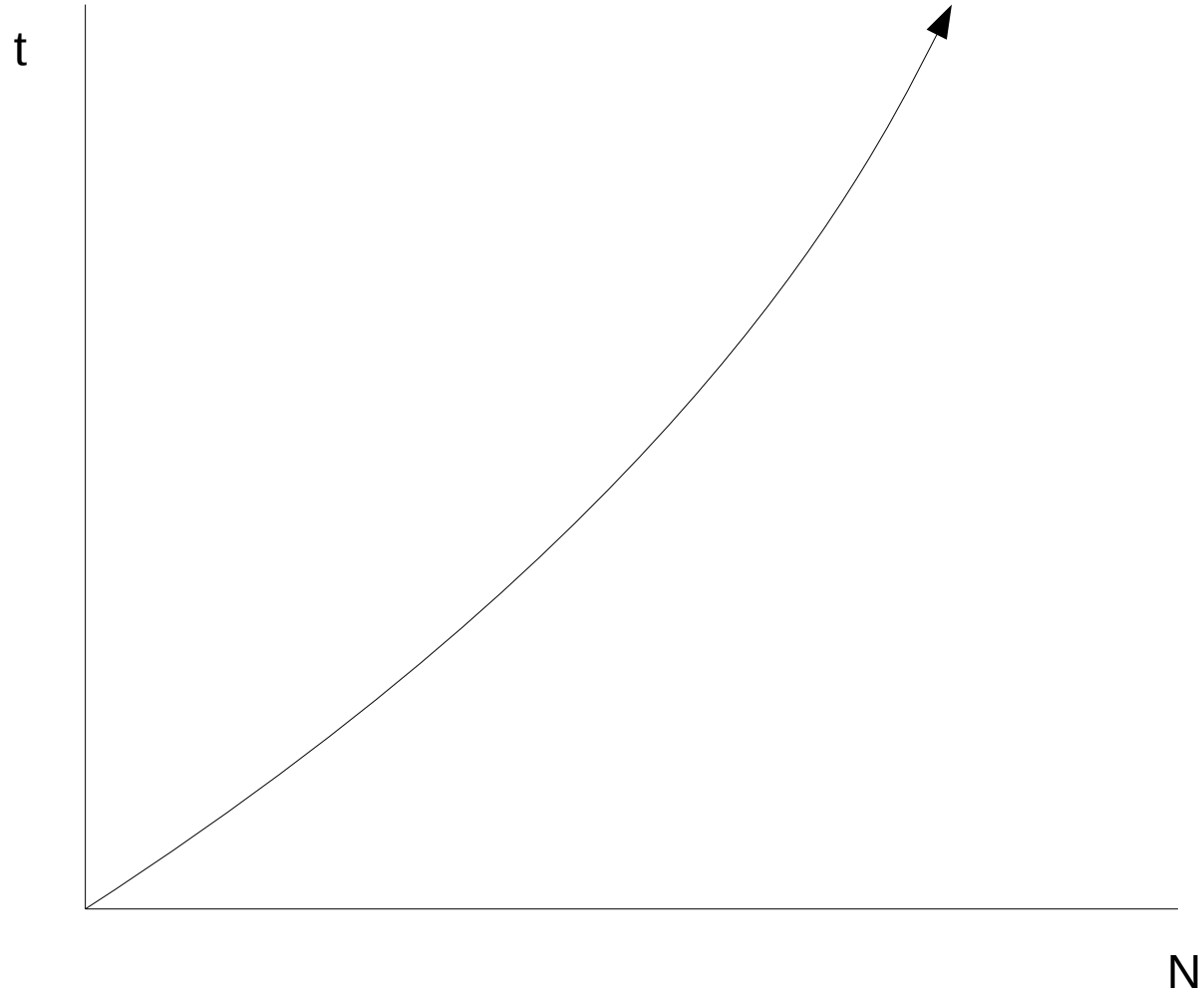
Análisis de eficiencia de algoritmos
Comprender el concepto de eficiencia

Orden lineal O(n):



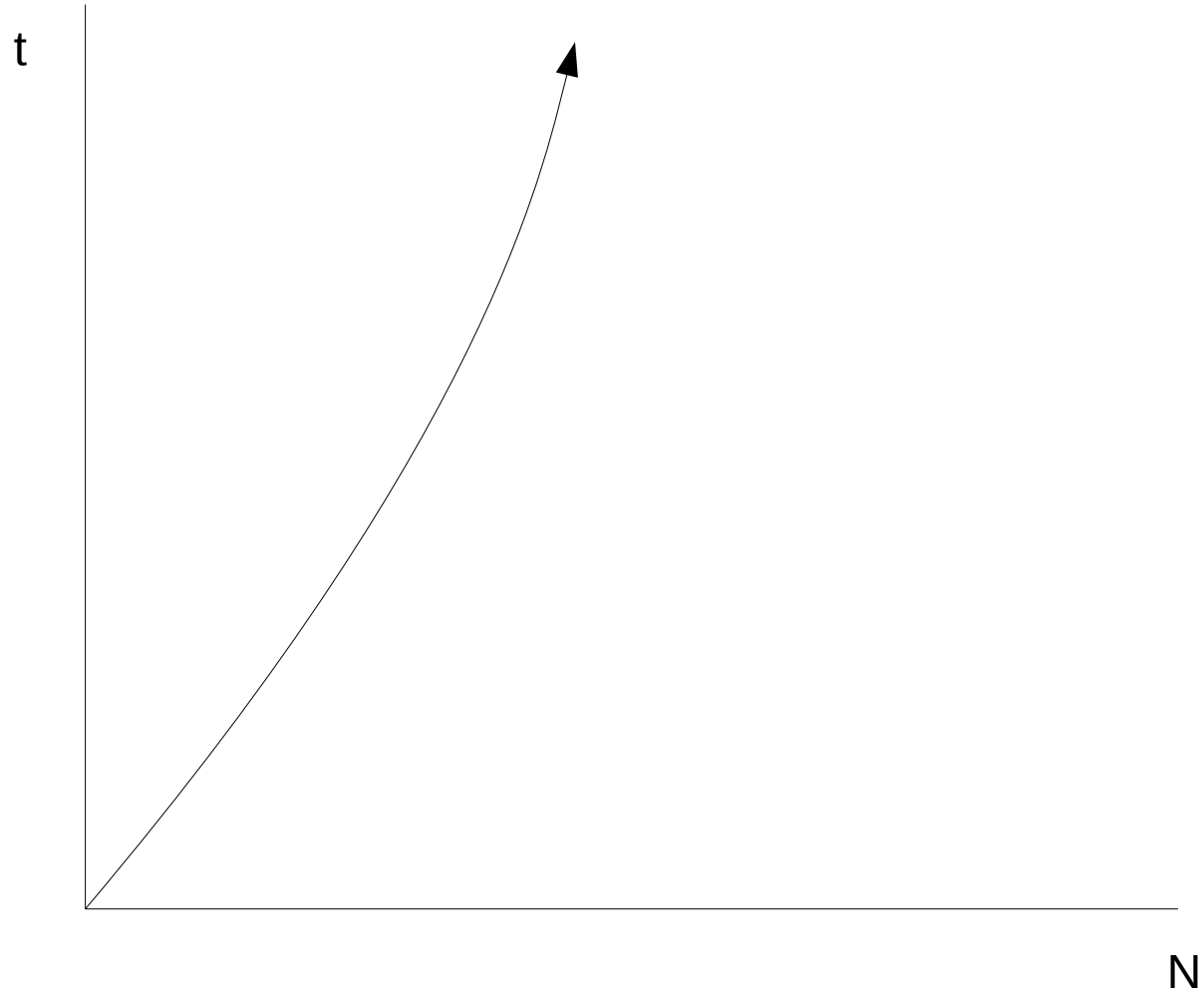
Análisis de eficiencia de algoritmos
Comprender el concepto de eficiencia

Orden lineal-logarítmico $O(n*\log_2 n)$:



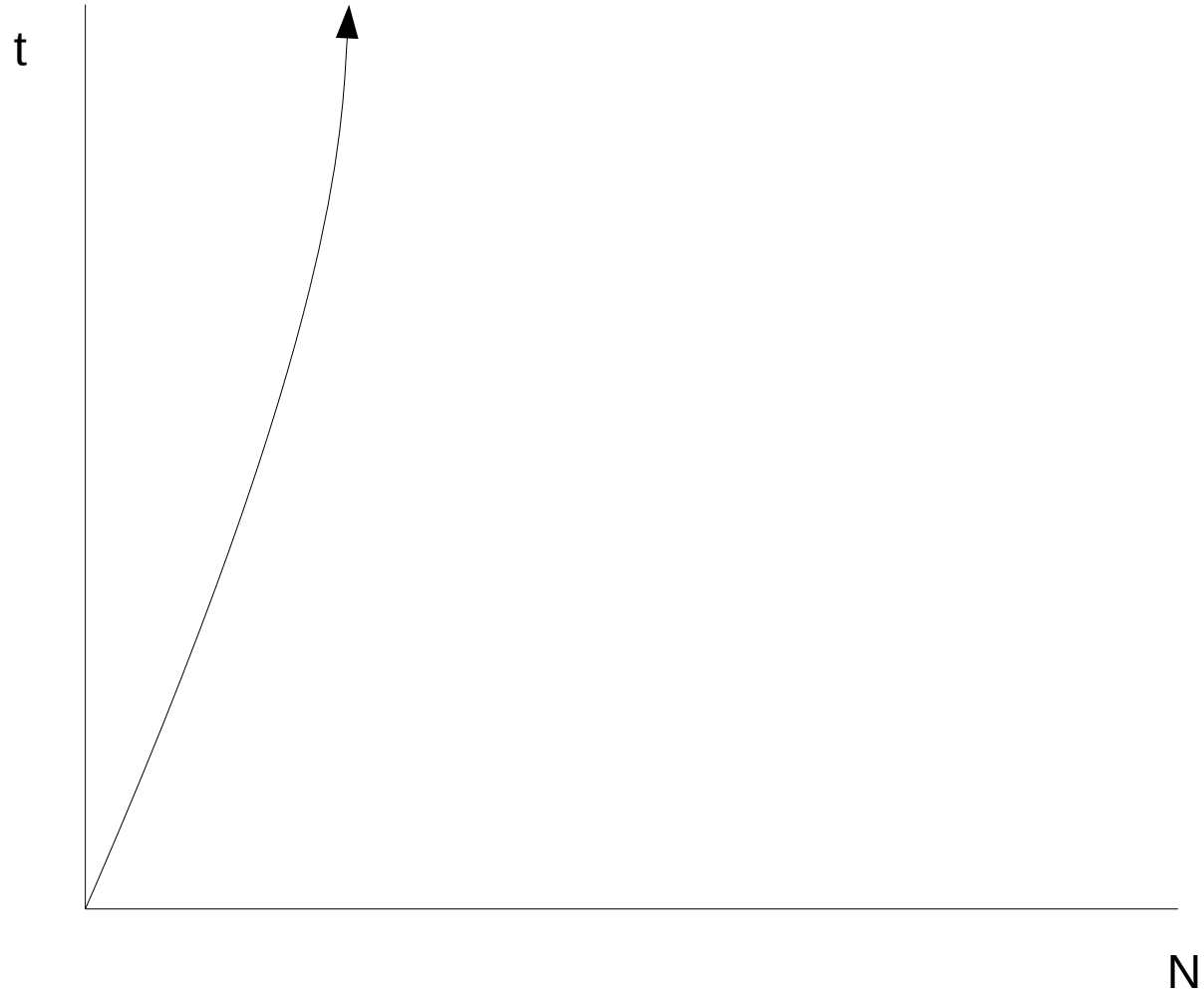
Análisis de eficiencia de algoritmos
Comprender el concepto de eficiencia

Orden cuadrático $O(n^2)$:



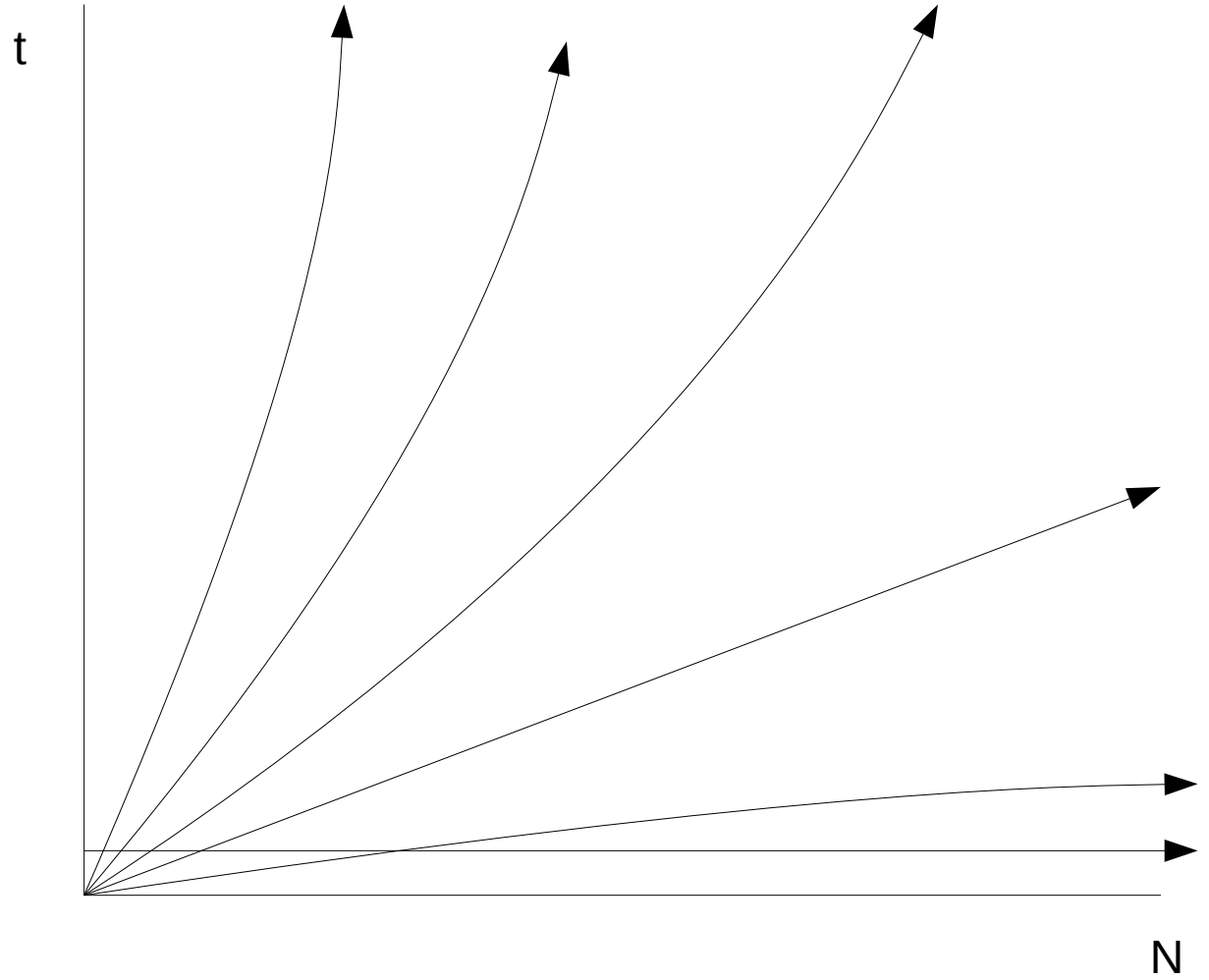
Análisis de eficiencia de algoritmos
Comprender el concepto de eficiencia

Orden exponencial $O(2^n)$:



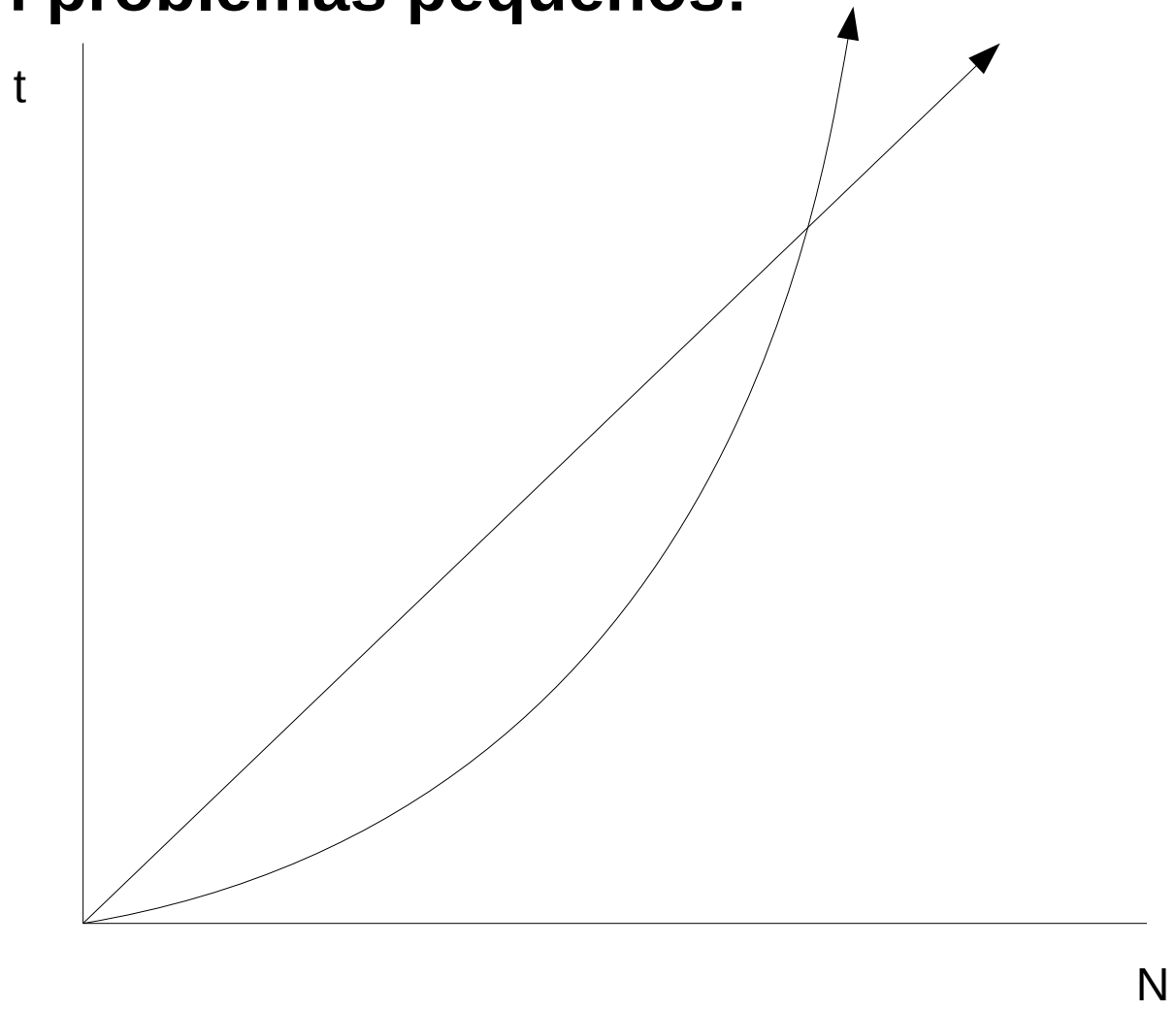
Análisis de eficiencia de algoritmos
Comprender el concepto de eficiencia

Todos juntos:



Análisis de eficiencia de algoritmos
Comprender el concepto de eficiencia

Ojo, **¡menor orden de complejidad no implica menor tiempo en problemas pequeños!**



Análisis de eficiencia de algoritmos
Comprender el concepto de eficiencia

Aunque podríamos calcular la eficiencia para el **mejor caso**, el **caso medio** y el **peor caso**, el que más limita nuestras posibilidades de elección de algoritmos es el peor caso, por lo que centraremos nuestro estudio en él.

Análisis de eficiencia de algoritmos
Comprender las notaciones asintóticas

Notación O Mayúscula:

Supone una **cota superior** (**infinito**) al tiempo de ejecución.

T(n) es O(f(n)) si existe una constante c, tal que para cualquier n, $T(n) < n*f(n)$.

Por ejemplo, si $T(n) = 2n^2+3n+5$, T(n) es $O(n^2)$, porque existe una constante, por ejemplo 3, tal que en cualquier caso:

$T(n) < 3*n^2$

Como podemos comprobar:

$2n^2+3n+5 < 3*n^2$

Nos interesa tomar el menor orden posible.

Análisis de eficiencia de algoritmos
Comprender las notaciones asintóticas

Notación O Mayúscula:

Supone una **cota superior** (**infinito**) al tiempo de ejecución.

Si $T(n) = (n+1)^2$	T(n) es $O(n^2)$
Si $T(n) = 3n^3+2n^2$	T(n) es $O(n^3)$
Si $T(n) = 2^n$	T(n) es $O(2^n)$
Si T(n) = n*log(n)+5	T(n) es O(n*log(n))
Si $T(n) = n*\log(n)+n^2$	T(n) es $O(n^2)$
Si T(n) = 15	T(n) es O(1) ← 15,1, es relativo

Podemos emplear O(f(n)) cuando en un número finito de valores de n, f(n) sea negativa o no esté definida.

$n/\log_2(n)$ no está definida para n=0 o n=1. $O(n/\log_2(n))$.

Análisis de eficiencia de algoritmos
Comprender las notaciones asintóticas

Notación Ω (Omega):

Supone una **cota inferior** (**infinito**) al tiempo de ejecución.

T(n) es Ω(f(n)) si existe una constante c, tal que para cualquier n, T(n) > n*f(n).

Por ejemplo, si $T(n) = 2n^2+3n+5$, T(n) es $\Omega(n^2)$, porque existe una constante, por ejemplo 3, tal que en cualquier caso:

$T(n) > 3*n^2$

Como podemos comprobar:

$2n^2+3n+5 > 3*n^2$

Nos interesa tomar el mayor orden posible.

Análisis de eficiencia de algoritmos
Comprender las notaciones asintóticas

Notación Ω (Omega):

Supone una **cota inferior** (**infinito**) al tiempo de ejecución.

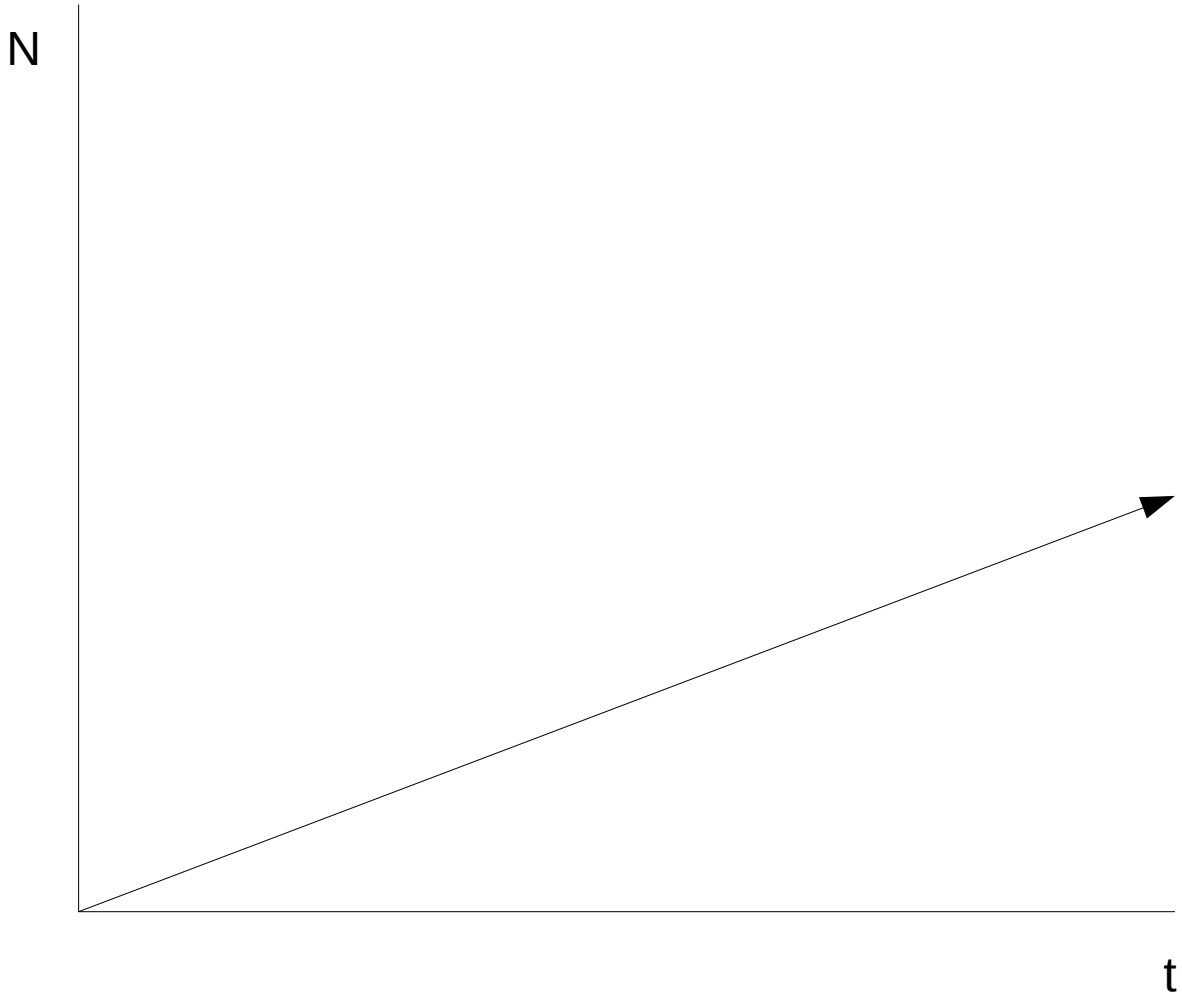
Si $T(n) = (n+1)^2$	$T(n)$ es $\Omega(n^2)$
Si $T(n) = 3n^3+2n^2$	$T(n)$ es $\Omega(n^3)$
Si $T(n) = 2^n$	$T(n)$ es $\Omega(2^n)$
Si $T(n) = n*\log(n)+5$	$T(n)$ es $\Omega(n*\log(n))$
Si $T(n) = n*\log(n)+n^2$	$T(n)$ es $\Omega(n^2)$
Si $T(n) = 15$	$T(n)$ es $\Omega(1)$ ← 15,1, es relativo

Podemos emplear Ω(f(n)) cuando en un número finito de valores de n, f(n) sea negativa o no esté definida.

$n/\log_2(n)$ no está definida para n=0 o n=1. $\Omega(n/\log_2(n))$.

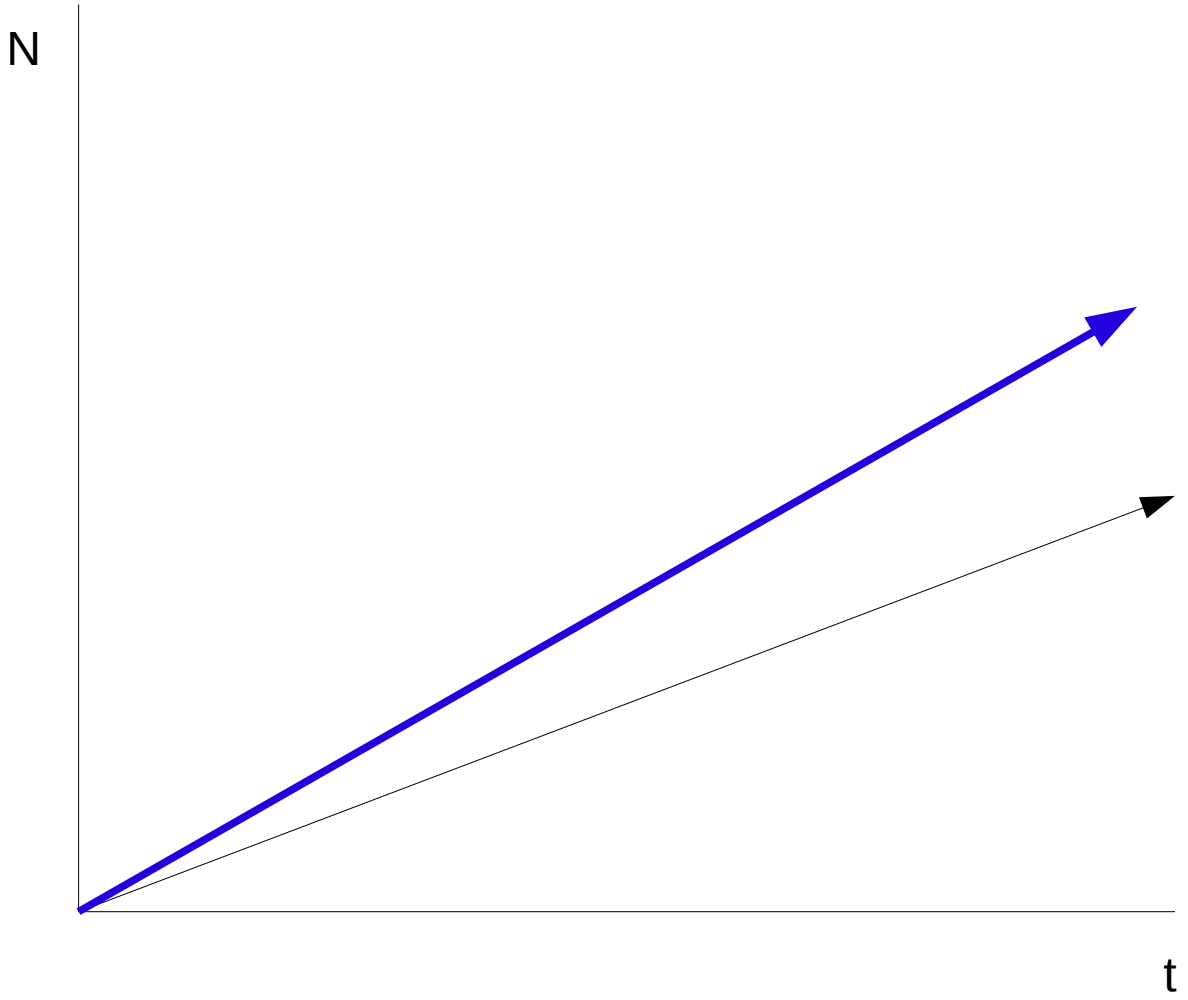
Análisis de eficiencia de algoritmos
Comprender las notaciones asintóticas

T(n) = n/2.



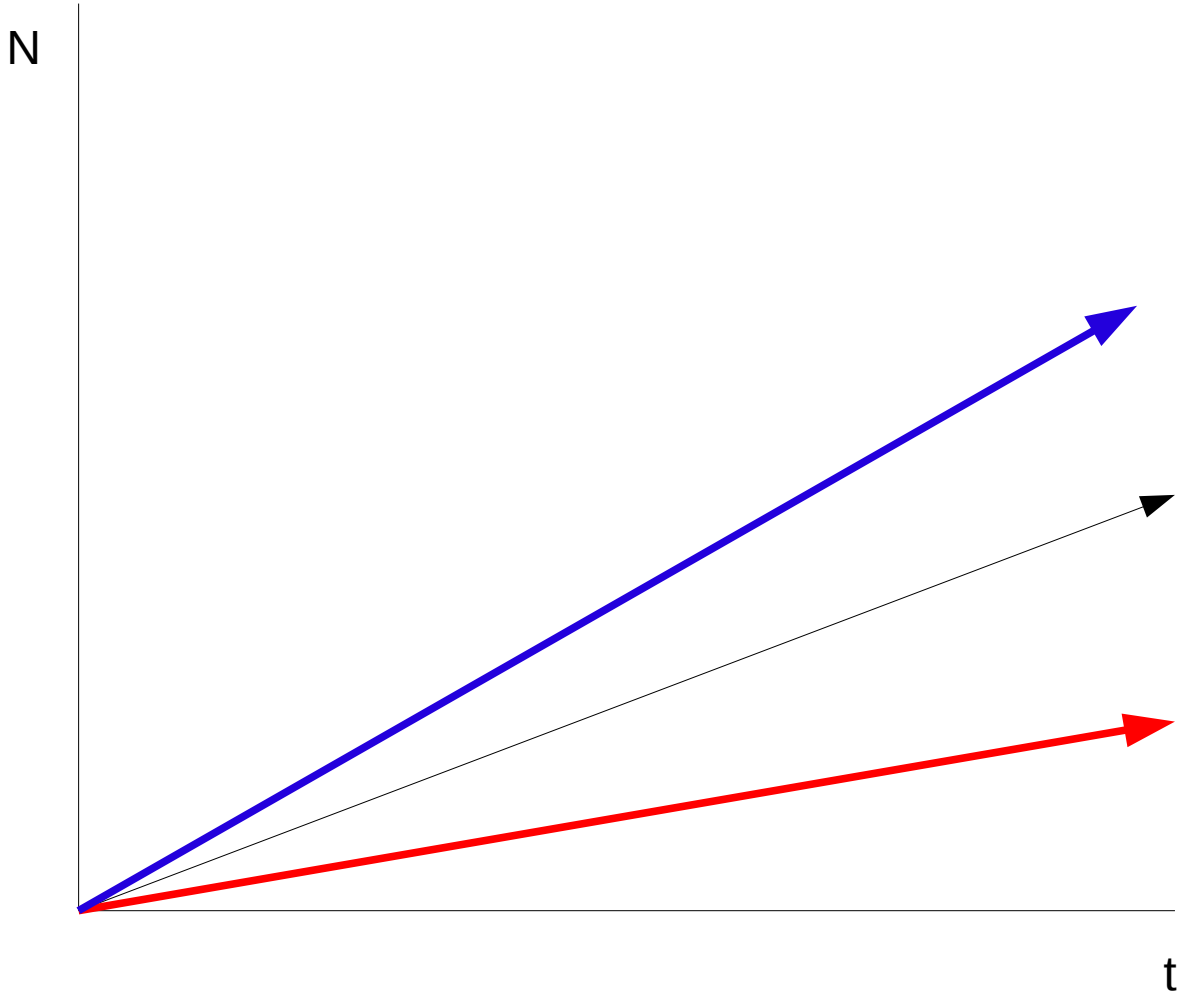
Análisis de eficiencia de algoritmos
Comprender las notaciones asintóticas

T(n) = n/2. T(n) es O(n).



Análisis de eficiencia de algoritmos
Comprender las notaciones asintóticas

T(n) = n/2. T(n) es O(n). T(n) es Ω(n).



Análisis de eficiencia de algoritmos
Comprender las notaciones asintóticas

Notación Θ (Theta):

Cuando se puede acotar el tiempo de un algoritmo tanto superior como inferiormente por la misma función, se usa la notación Theta.

Si T(n) es O(f(n))

Y T(n) es Ω(f(n))

Entonces se dice T(n) es orden exacto de f(n) y se escribe:

T(n) es Θ(f(n))

Por ejemplo, si $T(n) = 2n^2+3n+5$, T(n) es $\Theta(n^2)$.

Análisis de eficiencia de algoritmos
Comprender las notaciones asintóticas

Notación Θ (Theta):

Podemos observar un ejemplo en el que O(f(n)) no coincidirá con Ω(f(n)), y no podemos encontrar Θ(f(n)):

```
char cadena[30];
... //Lectura de la cadena, n almacena la longitud
int i = 0;
int cuantas = 0;
if (n > 5) {
  for (i = 0;i < n;i++)) {
    if (cadena[i] == 'a')
      cuantas++;
    i++;
  }
}
```

En este caso, O(T(n)) = n, porque si n>5, T(n) = n.

Además, Ω(T(n)) = 1, porque si n<=5, T(n) = cte.

Análisis de eficiencia de algoritmos
Comprender las notaciones asintóticas

Propiedades de estas notaciones:

Transitiva (O, Ω y Θ):

Si f(n) es O(g(n)) y g(n) es O(h(n)), f(n) es O(h(n)).

Reflexiva (O, Ω y Θ):

f(n) es O(f(n)).

Simétrica (Θ):

f(n) es Θ(g(n)) si y solo si g(n) es Θ(f(n)).

Análisis de eficiencia de algoritmos
Comprender las notaciones asintóticas

Propiedades de estas notaciones:

Suma (O, Ω y Θ):

Si $T_1(n)$ es O(f(n)) y $T_2(n)$ es O(g(n)).

$T_1(n) + T_2(n)$ es de orden O(max{f(n),g(n)}).

Producto (O, Ω y Θ):

Si $T_1(n)$ es O(f(n)) y $T_2(n)$ es O(g(n)).

$T_1(n) * T_2(n)$ es de orden O(f(n)*g(n)).

Análisis de eficiencia de algoritmos
Comprender el concepto de operación elemental

Una operación elemental es aquella cuyo **tiempo de ejecución** se puede acotar superiormente por una **constante**. **Independiente** del tamaño del problema. O(1).

Por ejemplo, son operaciones elementales:

```
i = 0;
i++;
if (i > 5) ... // ¡Sólo la comparación!
```

No lo serían:

```
for (i = 0;i < N;i++) {
  //¡INDEPENDIENTEMENTE de lo que haya dentro!
}
```

Pueden serlo o no serlo las llamadas a funciones.

```
double b = log(N)/log(2);
ordena(vector);
```

Análisis de eficiencia de algoritmos
Comprender el concepto de operación elemental

En nuestro análisis, sólo consideraremos el **número de operaciones elementales ejecutadas**, no el tiempo necesario para cada una de ellas.

El número de líneas de código **no tiene relación alguna** con el número de operaciones elementales.

Algunas operaciones matemáticas **no son elementales** (por ejemplo el tiempo necesario para resolver sumas y multiplicaciones depende de la longitud de los operandos), pero en la práctica **las consideramos elementales** siempre que los datos que se usen tengan un tamaño razonable.

Consideramos elementales: suma, resta, multiplicación, división, módulo, operaciones booleanas, comparativas y asignaciones.

Análisis de eficiencia de algoritmos
Saber calcular la eficiencia de un algoritmo

Sentencias simples (operaciones elementales):

```
1:    A[i] = A[indice];
2:    indice = i;
3:    A[i] = A[i]*2+1;
```

Cada operación elemental o secuencia simple es O(1).

Podemos considerarlas a nivel de:

- Sentencias simples (cada una vale 1).
- Operaciones elementales (A[i]*2+1 vale 3).
- Incluyendo acceso a vectores: (A[i]*2+1 valdría 4).

Da igual, en cualquier caso el tiempo es **O(1)**.

Por comodidad recomiendo considerarlas sentencias simples.

Análisis de eficiencia de algoritmos
Saber calcular la eficiencia de un algoritmo

Bloques de sentencias:

```
1:    A[i] = A[indice];
2:    indice = i;
3:    A[i] = A[i]*2+1;
```

Se suman:

- T(n) = 1+1+1 = 3.

Si se consideran operaciones elementales o acceso a vectores da igual, en cualquier caso el tiempo es de orden **O(1)**.

- Como sentencias simples: T(n) = 1+1+1 = 3.
- Como operaciones elementales: T(n) = 1+1+3 = 5.
- Considerando el acceso a vectores: T(n) = 3+1+5 = 9.

Análisis de eficiencia de algoritmos
Saber calcular la eficiencia de un algoritmo

Bucles:

```
1:  for (i = 0; i < n-1; i++) {
2:    A[i] = 'a';
3:  }
```

El tiempo necesario es el número de iteraciones del bucle multiplicado por el tiempo necesario en cada iteración.

¡Si el número de iteraciones depende del tamaño del problema, el orden crece! T(n) = n*1 = n, que es O(n).

```
1:  for (i = 0; i < n-1; i++) {
2:    for (j = 0; j < n; j++) {
3:      A[i][j] = 'a';
4:    }
5:  }
```

T(n) = n*n*1 = n^2, que es O(n^2).

Análisis de eficiencia de algoritmos
Saber calcular la eficiencia de un algoritmo

Bucles:

Por cierto, ¿Cómo podemos saber cuántas veces se ejecuta la instrucción interna a estos dos bucles?

```
1:  for (i = 0; i < n; i++) {
2:    for (j = i; j < n; j++) {
3:      A[i][j] = 'a';
4:    }
5:  }
```

En la primera iteración del bucle externo se ejecuta n veces.

En la segunda iteración se ejecuta n-1 veces...

…

En la penúltima se ejecuta 2 veces.

En la última se ejecuta 1 vez...

Análisis de eficiencia de algoritmos
Saber calcular la eficiencia de un algoritmo

Bucles:

Por cierto, ¿Cómo podemos saber cuántas veces se ejecuta la instrucción interna a estos dos bucles?

```
1:  for (i = 0; i < n; i++) {
2:    for (j = i; j < n; j++) {
3:      A[i][j] = 'a';
4:    }
5:  }
```

It 1: n veces

It 2: n-1 veces

It 3: n-2 veces

It n: 1 vez

It n-2: 2 veces

It n-3: 3 veces

Si emparejamos la última con la primera, la penúltima con la segunda...

Análisis de eficiencia de algoritmos
Saber calcular la eficiencia de un algoritmo

Bucles:

Por cierto, ¿Cómo podemos saber cuántas veces se ejecuta la instrucción interna a estos dos bucles?

```
1:  for (i = 0; i < n; i++) {
2:    for (j = i; j < n; j++) {
3:      A[i][j] = 'a';
4:    }
5:  }
```

It 1: n veces ⟶ n+1 en dos It ⟵ It n: 1 vez

It 2: n-1 veces ⟶ n+1 en dos It ⟵ It n-2: 2 veces

It 3: n-2 veces ⟶ n+1 en dos It ⟵ It n-3: 3 veces

Obtenemos que por cada dos It, se ejecuta n+1 veces.

Por tanto, T(n) = (n+1)*n/2, que sigue siendo O(n^2).

Análisis de eficiencia de algoritmos
Saber calcular la eficiencia de un algoritmo

Sentencias condicionales:

```
1:    if (a == 0)
2:      a = -1;
```

En las sentencias condicionales siempre se considera el **peor caso**. En este caso supondremos que la condición del if se cumple, porque requiere más tiempo de computación.

```
1:    if (a == 0)
2:      a = -1;
3:    else {
4:      for (i = 0;i < n;i++)
5:        vector[i] = 0;
6:    }
```

En este caso, supondremos que la condición del if no se cumple, porque el else requiere más tiempo de computación. **Siempre tomaremos el máximo**. max{O(1),O(n)} = O(n).

Análisis de eficiencia de algoritmos
Saber calcular la eficiencia de un algoritmo

Llamadas a funciones:

```
1: void vacia(int v[],int n) {
2:   int i;
3:   for (i = 0;i < n;i++)
4:     v[i] = 0;
5: }
6:
...
7:   a = 0;
8:   b = a*2;
9:   vacia(v,n);
```

Si una función es de orden f(n), su valor en todo código que la llame será f(n) porque como se va a ejecutar ahí, el tiempo que tarda se suma al tiempo de ejecución de ese código.

vacia(v,n) es O(n).

Análisis de eficiencia de algoritmos
Saber calcular la eficiencia de un algoritmo

Llamadas a funciones:

```
1: void vacia(int v[],int n) {
2:   int i;
3:   for (i = 0;i < n;i++)
4:     v[i] = 0;
5: }
6:
...
7:   a = 0;
8:   b = a*2;
9:   vacia(v,5);
```

OJO: En este caso la llamada vacia(v,5) no se puede considerar O(n), porque supone la ejecución de un número de instrucciones constante, que no depende del tamaño del problema.

Análisis de eficiencia de algoritmos
Comprender el concepto de principio de invarianza

Principio de invarianza:

El tiempo de ejecución de dos implementaciones de un mismo algoritmo no diferirán más que en una constante multiplicativa.

Si $T_1(n)$ y $T_2(n)$ son los tiempos de dos implementaciones de un mismo algoritmo, existe una constante c tal que:

$T_1(n) = c*T_2(n)$.

Por tanto, el orden de complejidad de dos implementaciones del mismo algoritmo va a ser siempre **igual**.

Análisis de eficiencia de algoritmos
¿Qué hemos aprendido?

- **Concepto de eficiencia.**
- **Notación O Mayúscula.**
- **Notación Ω (Omega).**
- **Notación Θ (Theta).**
- **Concepto de operación elemental.**
- **Cálculo de eficiencia.**
- **Principio de invarianza.**

Análisis de eficiencia de algoritmos
Ejemplos

```
1:  void seleccion(int A[], int n) {
2:    int i, j, min, t;
3:    for (i = 0; i < n - 1; i++) {
4:      min = i;
5:      for (j = i + 1; j < n; j++)
6:        if (A[j] < A[min])
7:          min = j;
8:      t = A[min];
9:      A[min] = A[i];
10:     A[i] = t;
11:   }
12: }
```

Análisis de eficiencia de algoritmos
Ejemplos

```
1:  void insercion(int A[], int n) {
2:    int  i, j, valor;
3:    for  (i = 1; i < n; i++) {
4:      valor = A[i];
5:      j = i;
6:      while ((j>0) && (A[j-1]>valor)) {
7:        A[j] = A[j-1];
8:        j--;
9:      }
10:     A[j] = valor;
11:   }
12: }
```

Análisis de eficiencia de algoritmos
Ejemplos

```
1:  void ejemplo(int n) {
2:    int x, contador;
3:    contador = 0;
4:    x = 2;
5:    while (x <= n) {
6:      x++;
7:      contador++;
8:    }
9:    cout << contador << endl;
10: }
```

Análisis de eficiencia de algoritmos
Ejemplos

```
1:  void ejemplo(int n) {
2:    int x, contador;
3:    contador = 0;
4:    x = 2;
5:    while (x <= n) {
6:      x = 2 * x;
7:      contador++;
8:    }
9:    cout << contador << endl;
10: }
```

En este caso, x aumenta multiplicándose por 2. Vamos a estudiar este ejemplo a fondo.

Análisis de eficiencia de algoritmos
Ejemplos

```
1:  void ejemplo(int n) {
2:    int x, contador;
3:    contador = 0;
4:    x = 2;
5:    while (x <= n) {
6:      x = 2 * x;
7:      contador++;
8:    }
9:    cout << contador << endl;
10: }
```

Ite	x
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256

Por ejemplo, en la iteración 6, podremos resolver un problema de tamaño 64. Pero lo que nos interesa saber es cuántas iteraciones harán falta para resolver un problema genérico de tamaño n. Para ello...

Análisis de eficiencia de algoritmos
Ejemplos

```
1:  void ejemplo(int n) {
2:    int x, contador;
3:    contador = 0;
4:    x = 2;
5:    while (x <= n) {
6:      x = 2 * x;
7:      contador++;
8:    }
9:    cout << contador << endl;
10: }
```

Ite	x
1	2^1 = 2^ite
2	2^2 = 2^ite
3	2^3 = 2^ite
4	2^4 = 2^ite
5	2^5 = 2^ite
6	2^6 = 2^ite
7	2^7 = 2^ite
8	2^8 = 2^ite

Para ello obtenemos la relación entre ite y el tamaño del problema. En este caso, en la iteración ite habremos procesado un problema de tamaño 2^ite.

Análisis de eficiencia de algoritmos
Ejemplos

```
1:  void ejemplo(int n) {
2:    int x, contador;
3:    contador = 0;
4:    x = 2;
5:    while (x <= n) {
6:      x = 2 * x;
7:      contador++;
8:    }
9:    cout << contador << endl;
10: }
```

Ite	x
1	2^1 = 2^ite
2	2^2 = 2^ite
3	2^3 = 2^ite
4	2^4 = 2^ite
5	2^5 = 2^ite
6	2^6 = 2^ite
7	2^7 = 2^ite
8	2^8 = 2^ite

Despejamos el número de iteraciones:

n = 2^ite

$\log_2(n) = \log_2(2^{ite})$

$\log_2(n) = ite$

Análisis de eficiencia de algoritmos
Ejemplos

```
1:  void ejemplo(int n) {
2:    int x, contador;
3:    contador = 0;
4:    x = 2;
5:    while (x <= n) {
6:      x = 2 * x;
7:      contador++;
8:    }
9:    cout << contador << endl;
10: }
```

Ite	x
1	2^1 = 2^ite
2	2^2 = 2^ite
3	2^3 = 2^ite
4	2^4 = 2^ite
5	2^5 = 2^ite
6	2^6 = 2^ite
7	2^7 = 2^ite
8	2^8 = 2^ite

Por tanto, el bucle se ejecuta $\log_2(n)$ veces, si su contenido es T(n) = 2, el bucle en total es $T(n) = 2*\log_2(n)$.