Programación Avanzada y Estructuras de Datos

3: La eficiencia de los algoritmos

Víctor M. Sánchez Cartagena

Grado en Ingeniería en Inteligencia Artificial
Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

2 de octubre de 2024

Índice

Índice

Problema introductorio #1

- ¿Cuál de estas dos funciones para buscar un elemento en un vector es más rápida? Asume que el vector tiene cientos de elementos. Sólo una opción es verdadera
  a. `buscar1` siempre es más rápida
  b. `buscar2` siempre es más rápida
  c. Una u otra, depende del contenido del vector
  d. `buscar2` nunca es más rápida que `buscar1`
  e. `buscar1` nunca es más rápida que `buscar2`

```
int buscar1( int[] vec, int n,
     int z ){
  for( int i = 0; i < n i++ )
    if( vec[i] == z )
      return i;
  return -1;
}
```

```
int buscar2( int[] vec, int n,
     int z ){
  int posicion=-1;
  for( int i = 0; i < n; i++ )
    if( vec[i] == z )
      posicion=i;
  return posicion;
}
```

Problema introductorio #2

- ¿Cuál de estas dos funciones es más rápida para vectores muy grandes? Sólo una opción es verdadera
  a. `maximo1` es más rápida
  b. `maximo2` es más rápida
  c. Una u otra, depende del contenido del vector
  d. `maximo2` nunca es más rápida que `maximo1`
  e. `maximo1` nunca es más rápida que `maximo2`

```
int maximo1 (int[] v, int n){
  int maximo=v[0];
  for(int k=0; k<500000; k++)
     for(int i=1; i< n; i++)
        if(v[i]>maximo)
            maximo=v[i];
  return maximo;
}
```

```
int maximo2 (int[] v, int n){
  int maximo=v[0];
  for(int k=0; k<n; k++)
     for(int i=1; i< n; i++)
        if(v[i]>maximo)
            maximo=v[i];
  return maximo;
}
```

Índice

Noción de complejidad

Definición: complejidad de un algoritmo

Es una medida de los recursos que necesita un algoritmo para resolver un problema

Los recursos más usuales son:

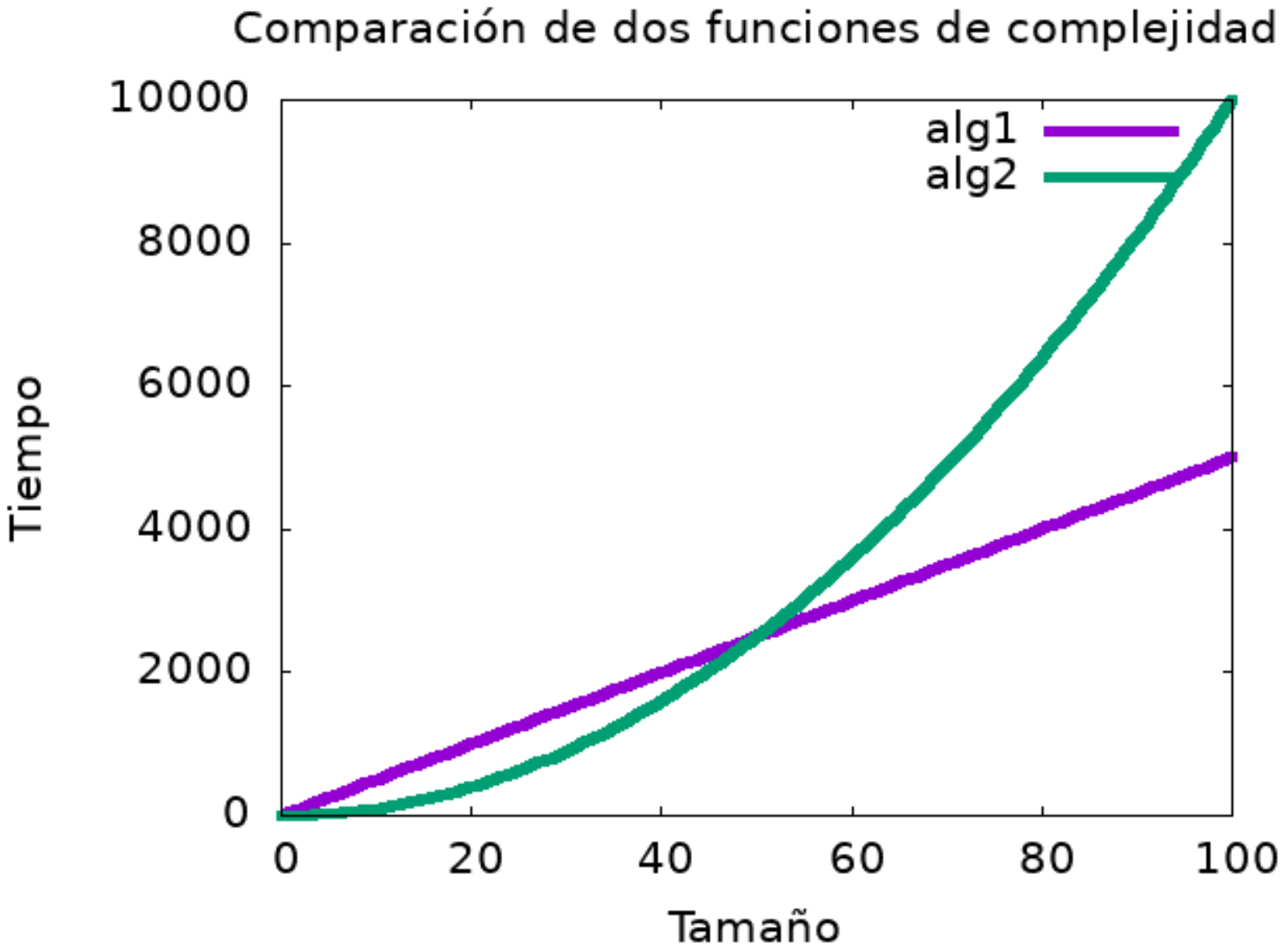
- Tiempo: Complejidad temporal
- Memoria: Complejidad espacial

Se suele expresar en función de la dificultad *a priori* del problema:

- Tamaño del problema: lo que ocupa su representación
- Parámetro representativo: el valor de n si queremos calcular $n!$

- La complejidad nos permite comparar y valorar algoritmos
- Nos centraremos, sobre todo, en la *complejidad temporal*

Noción de complejidad



Tiempo de ejecución de un algoritmo

El tiempo de ejecución de un algoritmo depende de:

Factores externos

- La máquina en la que se va a ejecutar
- El compilador
- La experiencia del programador

Factores internos

- El número de instrucciones que ejecuta el algoritmo y su duración

Principio de invarianza

Principio de invarianza

El tiempo de ejecución de dos implementaciones distintas de un algoritmo dado no va a diferir más que en una constante multiplicativa

Tiempo de ejecución: $c \cdot T(n)$

- c: constante multiplicativa que depende de factores externos
- $T(n)$: contribución de factores internos

Tiempo de ejecución de un algoritmo

Análisis empírico o *a posteriori*

Ejecutar el algoritmo para distintos valores de entrada y cronometrar el tiempo de ejecución

- ▲ Es una medida del comportamiento del algoritmo en su entorno
- ▼ El resultado depende de los factores externos e internos

Análisis teórico o *a priori*

Obtener una función que represente el tiempo de ejecución (en operaciones elementales) del algoritmo para cualquier valor de entrada

- ▲ El resultado depende sólo de los factores internos
- ▲ No es necesario implementar y ejecutar los algoritmos
- ▼ No obtiene una medida real del comportamiento del algoritmo en el entorno de aplicación

Tiempo de ejecución de un algoritmo

Operaciones elementales

Son aquellas operaciones que realiza el ordenador en un tiempo acotado por una constante

Operaciones elementales

- Operaciones aritméticas básicas
- Asignaciones a variables de tipo predefinido por el compilador
- Los saltos (llamadas a funciones, retorno desde ellos . . .)
- Comparaciones lógicas
- Acceso a estructuras indexadas básicas (vectores o matrices)
- **Cualquier combinación de las anteriores acotada por una constante**

Tiempo de ejecución de un algoritmo

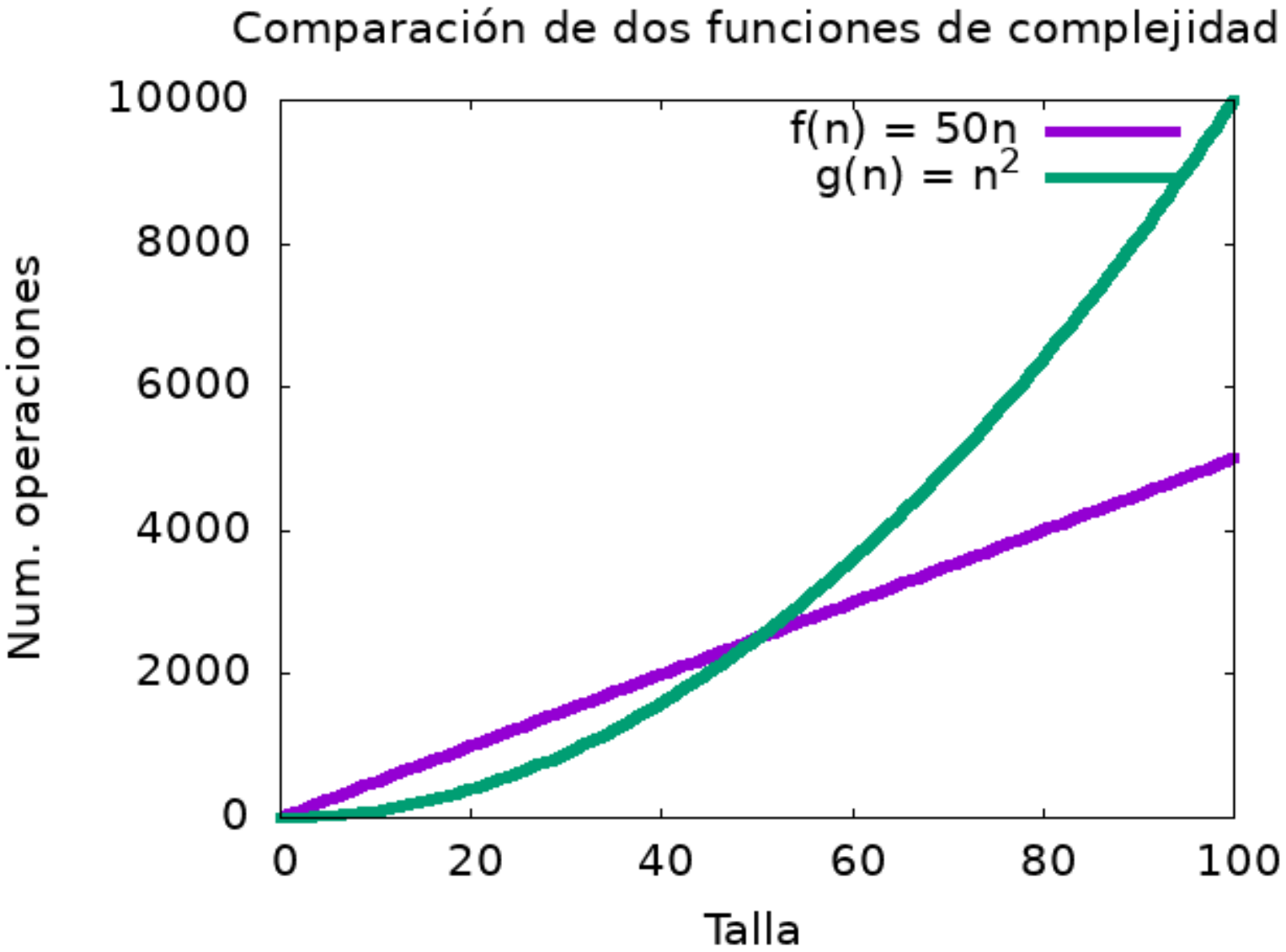
- Se suele considerar que el coste temporal de las operaciones elementales es unitario
- En el contexto de Programación Avanzada y Estructuras de Datos:

Tiempo de ejecución de un algoritmo

Función ($T(n)$) que mide el número de operaciones elementales que realiza el algoritmo cualquier talla de problema n

- *Talla*: tamaño del problema o parámetro representativo, según convenga

Tiempo de ejecución de un algoritmo



Índice

Ejemplos

- Talla: valor de n

Programa 1:

```
int ejemplo1 (int n){
  n+=n;
  return n;
}
```

Programa 2:

```
int ejemplo2 (int n){
  for(int i=0; i<=2000; i++)
    n+=n;
  return n;
}
```

Ejemplos

- Talla: valor de n

Programa 1:

```
int ejemplo1 (int n){
  n+=n;
  return n;
}
```

Programa 2:

```
int ejemplo2 (int n){
  for(int i=0; i<=2000; i++)
    n+=n;
  return n;
}
```

- El programa 1 calcula el resultado en 1 operación: $T(n) = 1$

Ejemplos

- Talla: valor de n

Programa 1:

```
int ejemplo1 (int n){
  n+=n;
  return n;
}
```

Programa 2:

```
int ejemplo2 (int n){
  for(int i=0; i<=2000; i++)
    n+=n;
  return n;
}
```

- El programa 1 calcula el resultado en 1 operación: $T(n) = 1$
- El programa 2 calcula el resultado en 1 operación: $T(n) = 1$

Ejemplos

- Talla: valor de n

Programa 3:

```
int ejemplo3 (int n){
  int i, j;
  j = 2;

  for (i=0; i<=2000; i++)
    j=j*j;
  for (i=0; i<= n; i++){
    j = j + j;
    j = j - 2;
  }
  return j;
}
```

Ejemplos

- Talla: valor de n

Programa 3:

```
int ejemplo3 (int n){
  int i, j;
  j = 2;

  for (i=0; i<=2000; i++)
    j=j*j;
  for (i=0; i<= n; i++){
    j = j + j;
    j = j - 2;
  }
  return j;
}
```

$$T(n) = 1 + (n+1) \cdot 1 = n + 2$$

Ejemplos

- Talla: valor de n

Programa 4:

```
int ejemplo4 (int n){
  int i, j ,k;
  k=1
  for (i=0; i<= n; i++){
    for(j=1; j<=n; j++)
      k=k+k;
  }
  return k;
}
```

Ejemplos

- Talla: valor de n

Programa 4:

```
int ejemplo4 (int n){
  int i, j ,k;
  k=1
  for (i=0; i<= n; i++){
    for(j=1; j<=n; j++)
      k=k+k;
  }
  return k;
}
```

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

Ejemplos

- Talla: valor de n

Programa 5:

```
int ejemplo5 (int n){
  int i, j ,k;
  k=1
  for (i=0; i<= n; i++){
    for(j=i; j<=n; j++)
      k=k+k;
  }
  return k;
}
```

¿ $T(n) = 1 + (n+1) \cdot n \cdot 1$?

Ejemplos

- Talla: valor de n

Programa 5:

```
int ejemplo5 (int n){
  int i, j ,k;
  k=1
  for (i=0; i<= n; i++){
    for(j=i; j<=n; j++)
      k=k+k;
  }
  return k;
}
```

¿ $T(n) = 1 + (n+1) \cdot n \cdot 1$?

ERROR

Resolución de sumatorios

- Sumatorio: expresión matemática para "escribir" una suma de manera compacta

$$\sum_{i=1}^{n}(\text{exp})$$

- ... equivalente a $\text{exp} + \text{exp} + \text{exp} + \ldots$ n veces (desde que i vale 1 hasta que vale n)

Ejemplos:

- $\sum_{i=1}^{4}(2) = 2 + 2 + 2 + 2 = 8$
- $\sum_{i=1}^{4}(2i) = 2 + 4 + 6 + 8 = 20 \rightarrow$ progresión aritmética
- $\sum_{i=1}^{4}(2^i) = 2 + 4 + 8 + 16 = 30 \rightarrow$ progresión geométrica

Resolución de sumatorios

- Sumatorio de una constante:

$$\sum_{i=a}^{b}(C) = (b - a + 1) \cdot C$$

- Suma de una progresión aritmética

$$\sum_{i=a}^{b}(p_i) = \frac{(p_a + p_b)}{2} \cdot (b - a + 1)$$

- No trataremos con progresiones geométricas

Ejemplos

- Talla: valor de n

Programa 5:

```
int ejemplo5 (int n){
  int i, j ,k;
  k=1
  for (i=0; i<= n; i++){
    for(j=i; j<=n; j++)
      k=k+k;
  }
  return k;
}
```

Ejemplo

Programa 5:

```
int ejemplo5 (int n){
  int i, j ,k;
  k=1
  for (i=0; i<= n; i++){
    for(j=i; j<=n; j++)
      k=k+k;
  }
  return k;
}
```

$$T(n) = 1 + \sum_{i=0}^{n}\sum_{j=i}^{n} 1 =$$

$$1 + \sum_{i=0}^{n}(n-i+1)\cdot 1 =$$

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

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

$$1 + \frac{n^2}{2} + \frac{n}{2} + n + 1 =$$

$$\frac{n^2}{2} + \frac{3n}{2} + 2$$

Ejercicios

Calcula la función tiempo de ejecución, para los siguientes fragmentos de código (talla $=$ valor de `n`):

Ejercicio 1:

```
for(i = sum = 0; i < n; i++) sum += a[i];
```

Ejercicio 2:

```
for(i = 0; i < n; i++) {
   for(j = 1, sum = a[0]; j <= i; j++) sum += a[j];
   cout << "La suma del subarray " << i << " es " << sum << endl; }
```

Ejercicio 3:

```
for(i = 4; i < n; i++) {
   for(j = i-3, sum = a[i-4]; j <= i; j++) sum += a[j];
   cout << "La suma del subarray " << i-4 << " es " << sum << endl; }
```

Índice

¿Por qué necesitamos cotas?

- Dado un vector de enteros `v` y el entero `z`
  - Devuelve el primer índice `i` tal que `v[i] == z`
  - Devuelve `-1` en caso de no encontrarlo

Búsqueda de un elemento

```
int buscar( int[] v, int n, int z ){
  for( int i = 0; i < n; i++ )
    if( v[i] == z )
      return i;
  return -1;
}
```

v	z	Operaciones
$(1,0,2,4)$	1	1
$(1,0,2,4)$	0	2
$(1,0,2,4)$	2	3
$(1,0,2,4)$	4	4
$(1,0,2,4)$	5	5

¿Por qué necesitamos cotas?

- No podemos contar el número de operaciones porque para diferentes entradas de un mismo tamaño de problema se obtienen diferentes complejidades
- ¿Qué podemos hacer?

¿Por qué necesitamos cotas?

- No podemos contar el número de operaciones porque para diferentes entradas de un mismo tamaño de problema se obtienen diferentes complejidades
- ¿Qué podemos hacer?

Solución

Acotar el coste mediante dos funciones que expresen respectivamente, el coste máximo y el coste mínimo del algoritmo (cotas de complejidad)

Cotas de complejidad

- Cuando aparecen diferentes casos para una misma talla n, se introducen las siguientes medidas de la complejidad
  - Caso peor: cota superior del algoritmo $\rightarrow C_s(n)$
  - Caso mejor: cota inferior del algoritmo $\rightarrow C_i(n)$
  - Caso promedio: coste promedio $\rightarrow C_m(n)$
- Todas son funciones de la talla del problema
- El coste promedio es difícil de evaluar a priori
  - Es necesario conocer la distribución de probabilidad de la entrada
  - ¡No es la media de la cota inferior y de la cota superior!

Cotas superior e inferior

Buscar elemento

```
int buscar( int[] v, int n, int z ){
  for( int i = 0; i < n; i++ )
    if( v[i] == z )
      return i;
  return -1;
}
```

- La talla es `n`

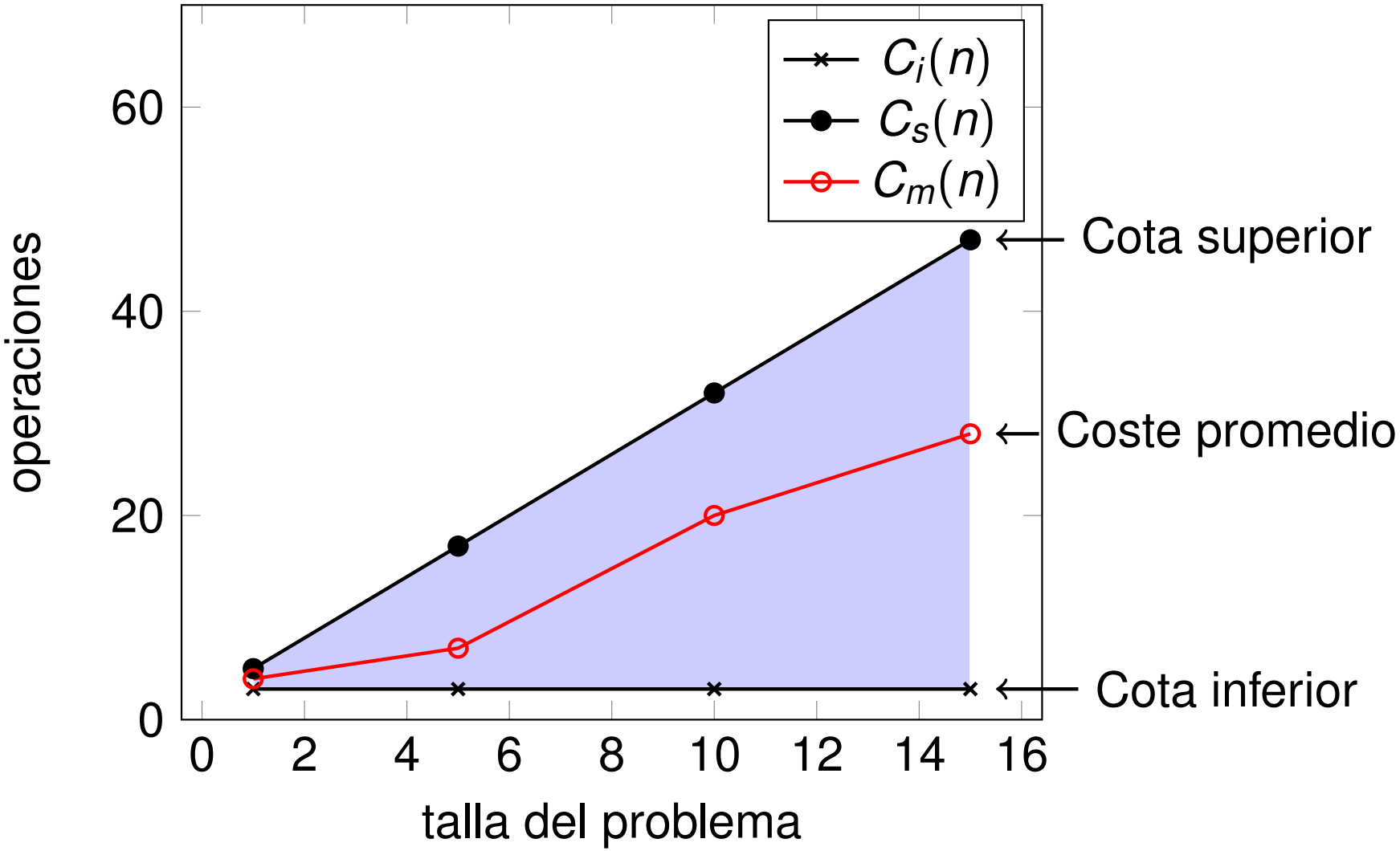
- **Mejor caso**: el elemento buscado está el primero
- **Peor caso**: el elemento buscado no está

Cotas:

$$C_s(n) = 1 + n$$
$$C_i(n) = 1$$

Cotas superior e inferior

- Coste de la función `buscar`



Problema introductorio #1

- ¿Cuál de estas dos funciones para buscar un elemento en un vector es más rápida? Asume que el vector tiene cientos de elementos. Sólo una opción es verdadera
  a. `buscar1` siempre es más rápida
  b. `buscar2` siempre es más rápida
  c. Una u otra, depende del contenido del vector
  d. `buscar2` nunca es más rápida que `buscar1`
  e. `buscar1` nunca es más rápida que `buscar2`

```
int buscar1( int[] vec, int n,
     int z ){
  for( int i = 0; i < n i++ )
    if( vec[i] == z )
      return i;
  return -1;
}
```

```
int buscar2( int[] vec, int n,
     int z ){
  int posicion=-1;
  for( int i = 0; i < n; i++ )
    if( vec[i] == z )
      posicion=i;
  return posicion;
}
```

Problema introductorio #1

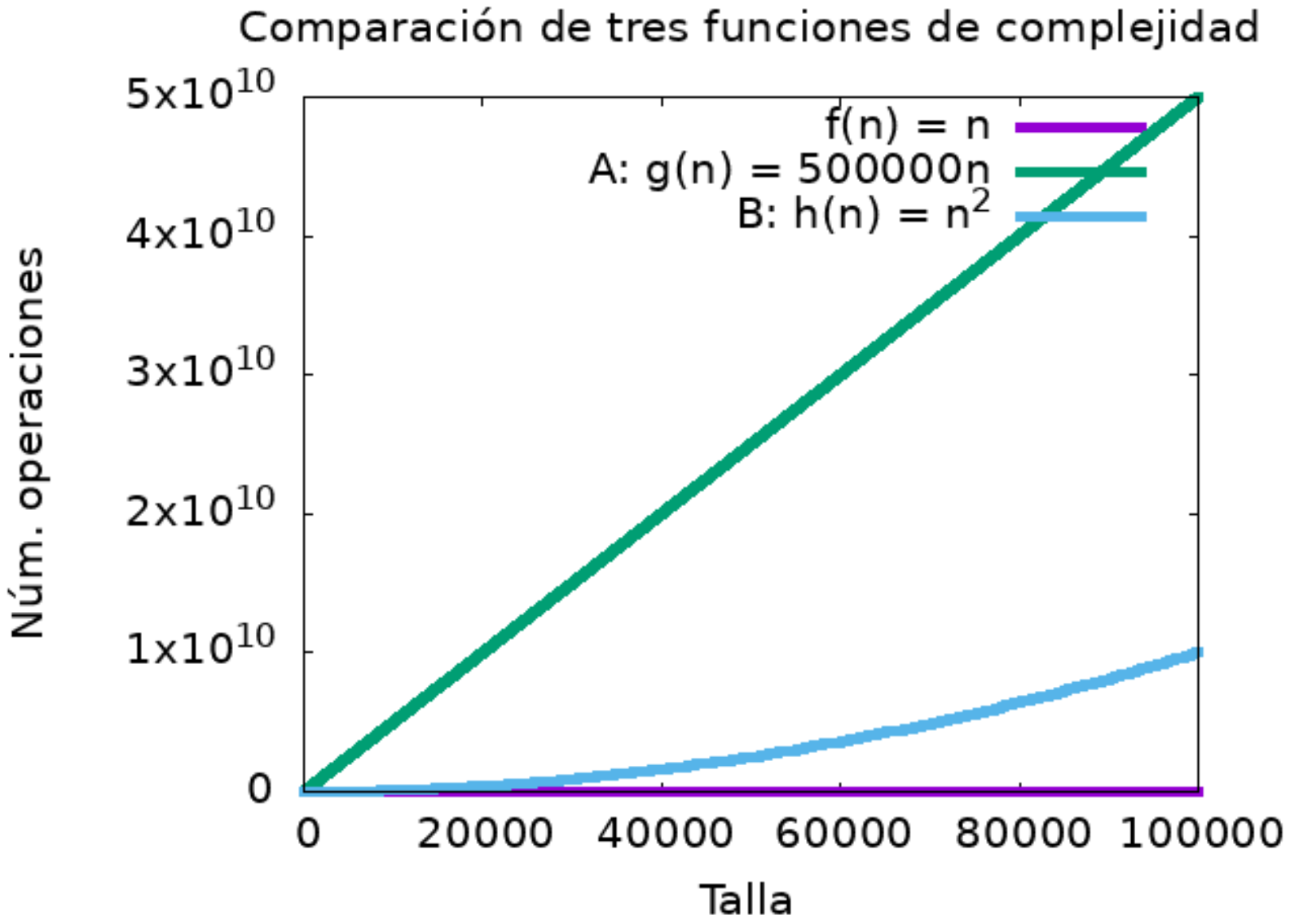
- ¿Cuál de estas dos funciones para buscar un elemento en un vector es más rápida? Asume que el vector tiene cientos de elementos. Sólo una opción es verdadera
  a. `buscar1` siempre es más rápida
  b. `buscar2` siempre es más rápida
  c. Una u otra, depende del contenido del vector
  d. **`buscar2` nunca es más rápida que `buscar1`**
  e. `buscar1` nunca es más rápida que `buscar2`

```
int buscar1( int[] vec, int n,
     int z ){
  for( int i = 0; i < n i++ )
    if( vec[i] == z )
      return i;
  return -1;
}
```

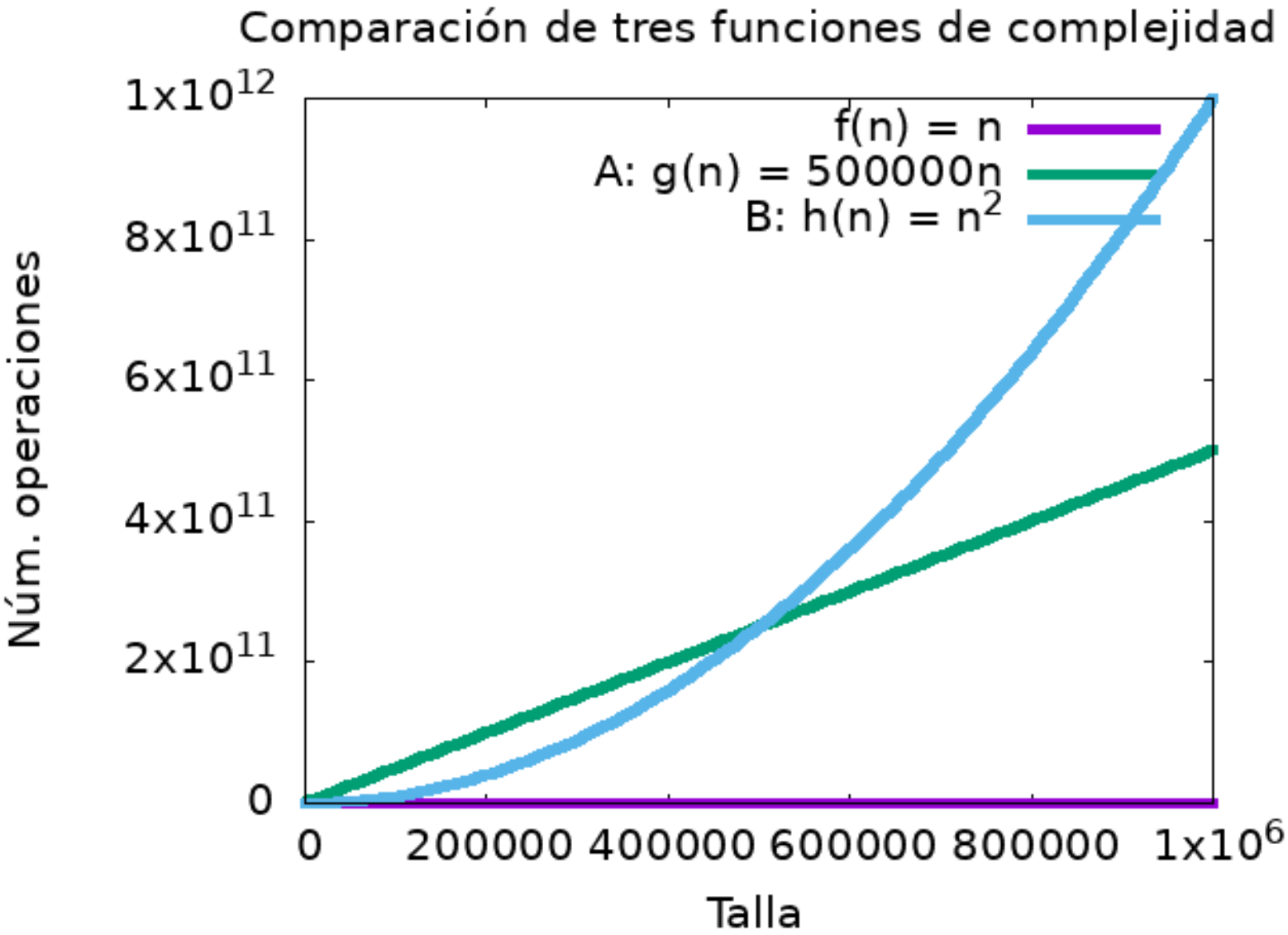
```
int buscar2( int[] vec, int n,
     int z ){
  int posicion=-1;
  for( int i = 0; i < n; i++ )
    if( vec[i] == z )
      posicion=i;
  return posicion;
}
```

Índice

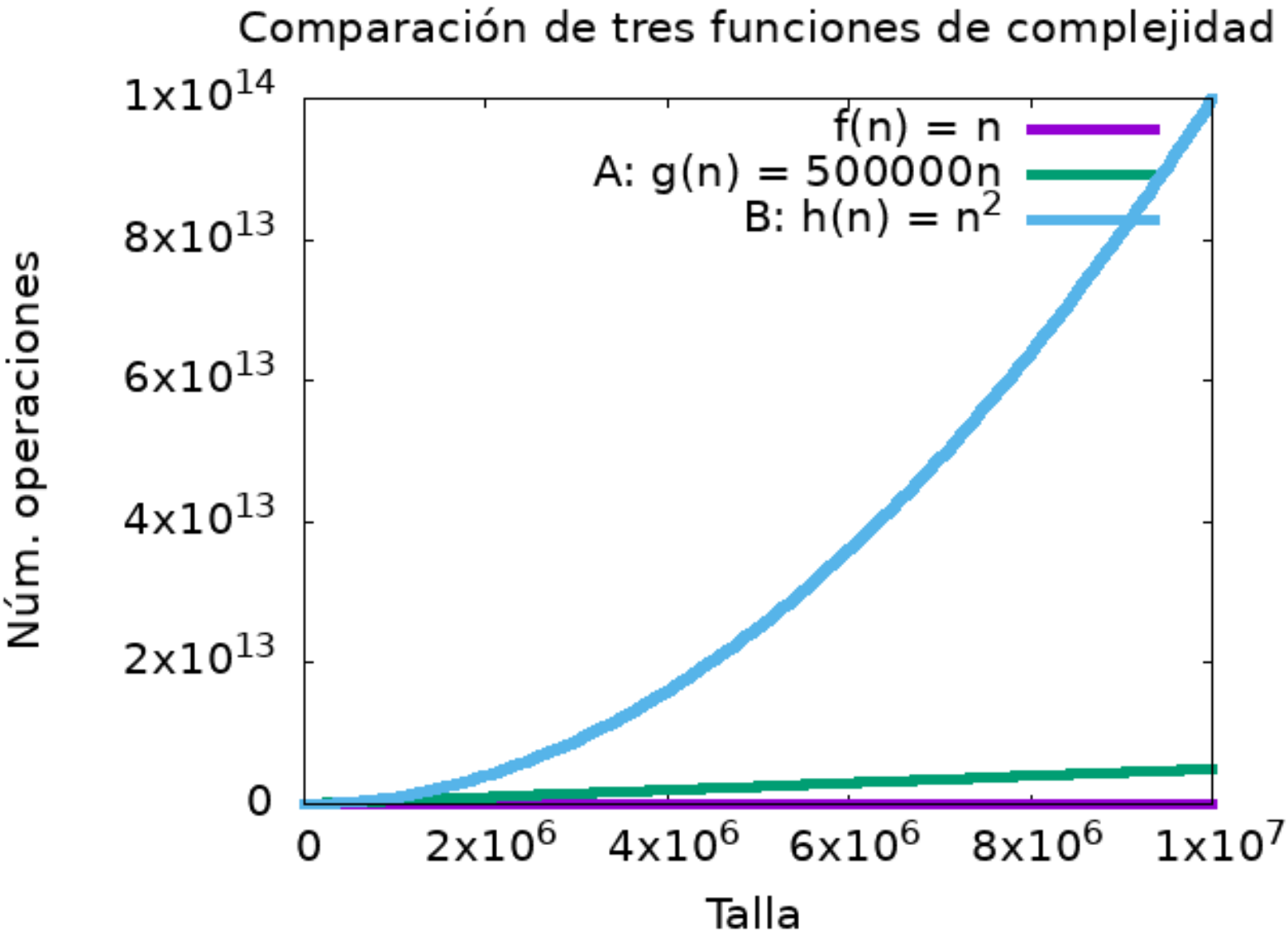
Análisis asintótico



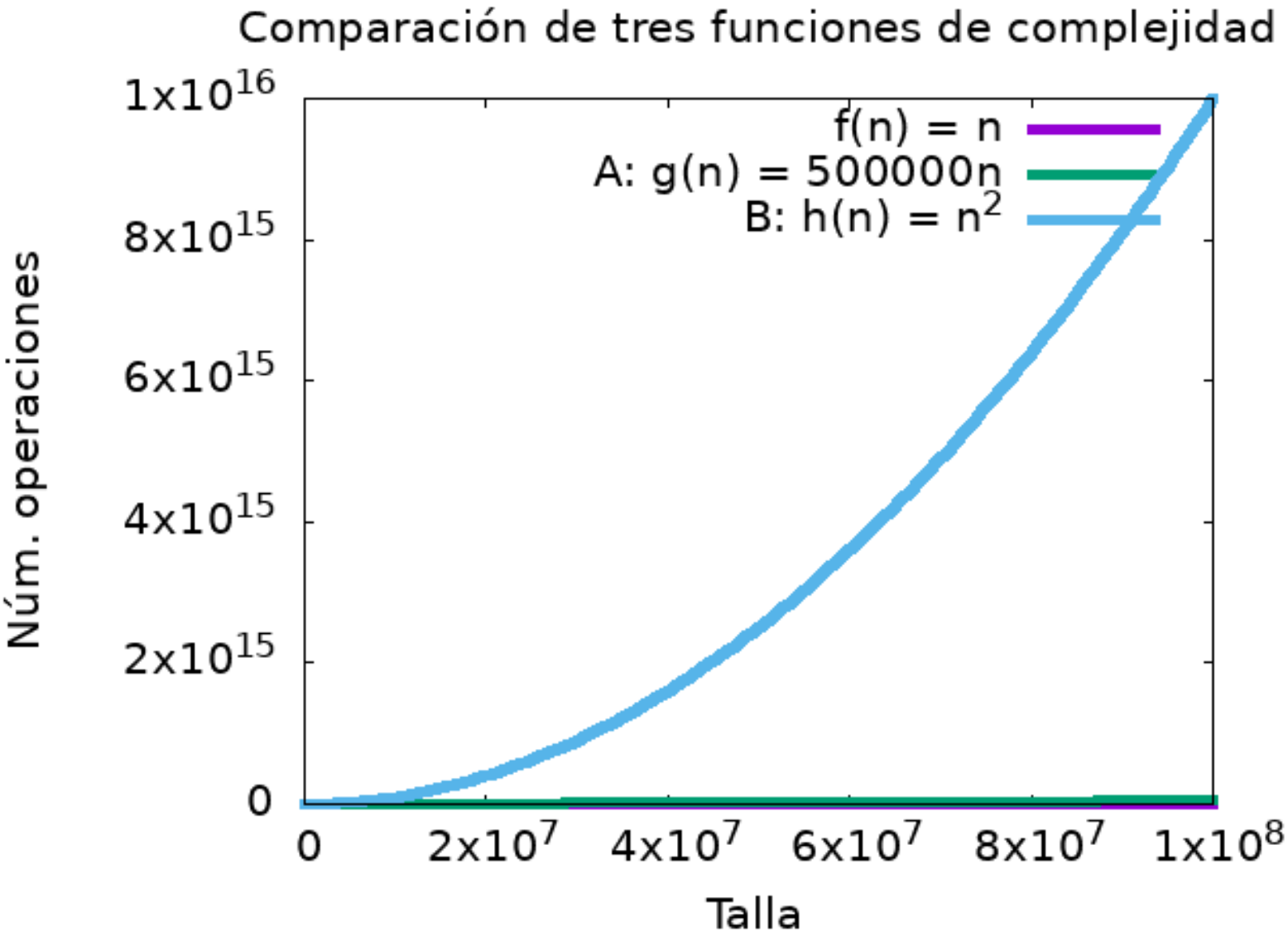
Análisis asintótico



Análisis asintótico



Análisis asintótico



Análisis asintótico

- El estudio de la complejidad resulta interesante para tamaños grandes de problema:
  - Diferencias en tiempo de ejecución de algoritmos con diferente coste para tamaños pequeños del problema son muy pequeñas
  - Es lógico invertir tiempo en el desarrollo de un buen algoritmo sólo si se prevé que éste realizará un gran volumen de operaciones
- Análisis asintótico: estudio de la complejidad para tamaños grandes de problema
  - Permite clasificar las funciones de complejidad de forma que podamos compararlas entre si fácilmente
  - Se definen clases de equivalencia que engloban a las funciones que “crecen de la misma forma”.
- Se emplea la notación asintótica

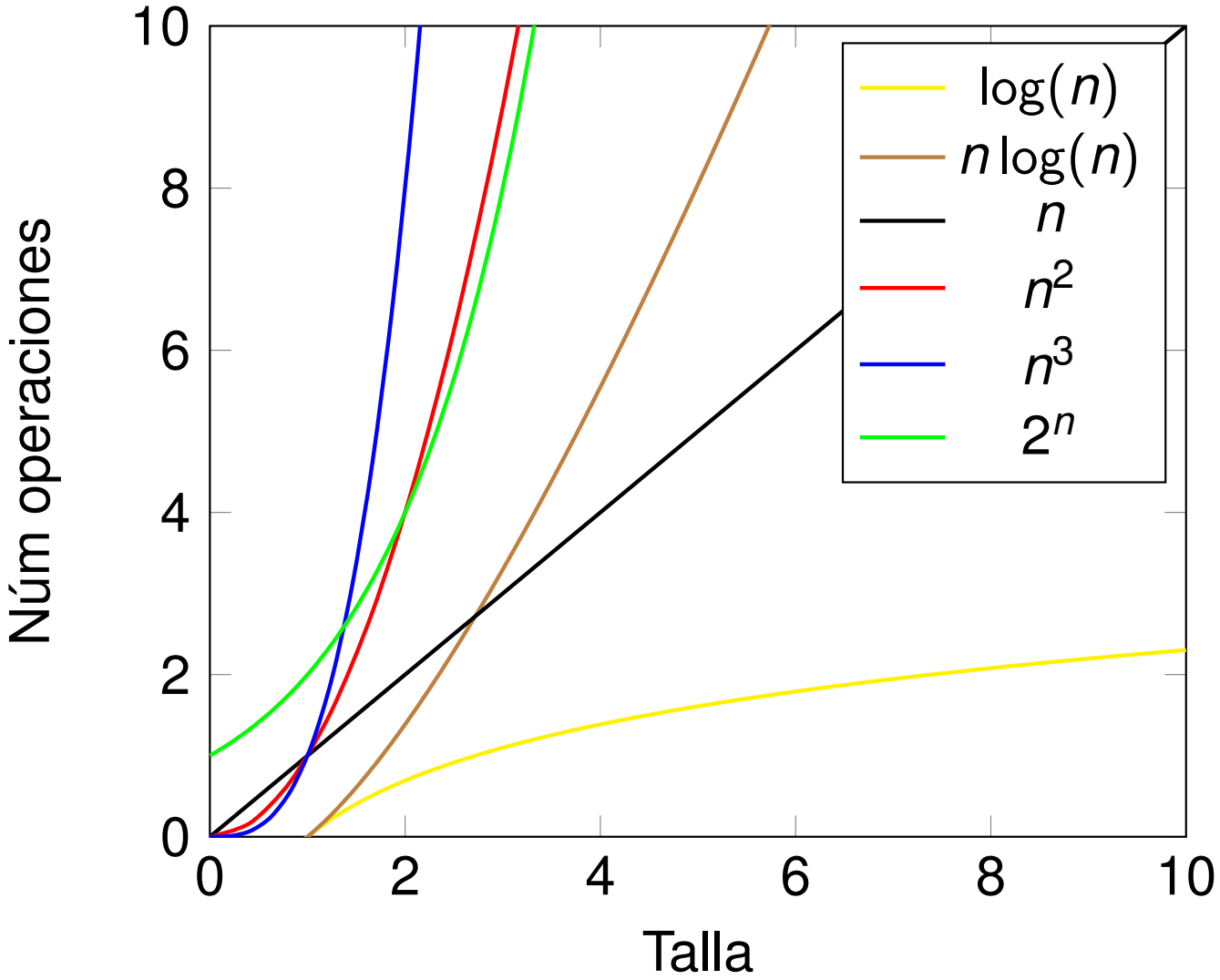
Análisis asintótico

Notación asintótica:

- Notación matemática utilizada para representar la complejidad cuando el tamaño de problema (n) crece ($n \to \infty$)
- Se definen tres tipos de notación:
  - Notación O (ómicron mayúscula o *big omicron*) $\Rightarrow$ cota superior
  - Notación Ω (omega mayúscula o *big omega*) $\Rightarrow$ cota inferior
  - Notación Θ (zeta mayúscula o *big theta*)$\Rightarrow$ coste exacto

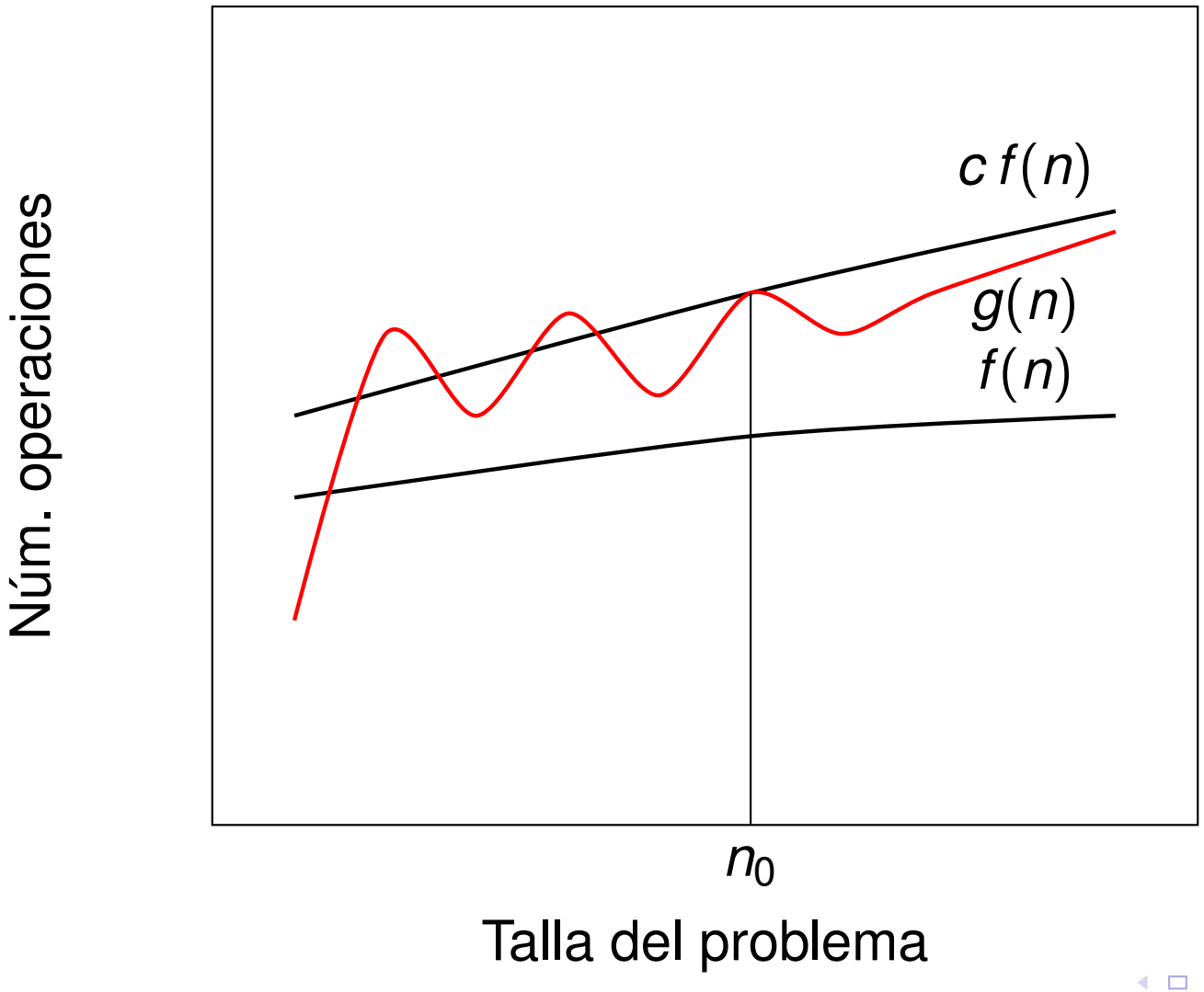
Análisis asintótico

- Permite agrupar en clases funciones con el mismo crecimiento



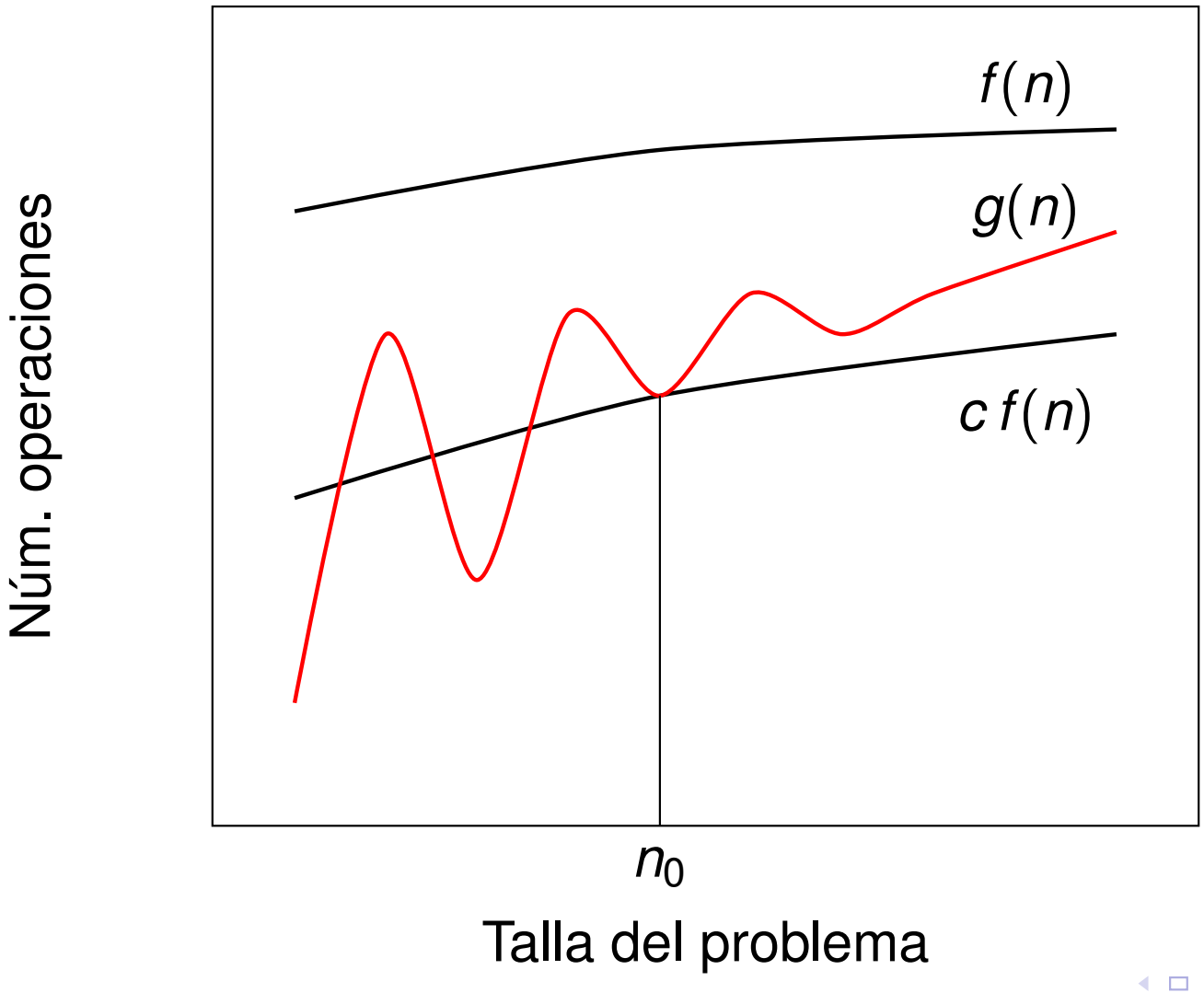
Cota superior. Notación O

- Sea $f : \mathbb{N} \to \mathbb{R}^+$; se define el conjunto $O(f)$ como el conjunto de funciones acotadas superiormente por un múltiplo de f



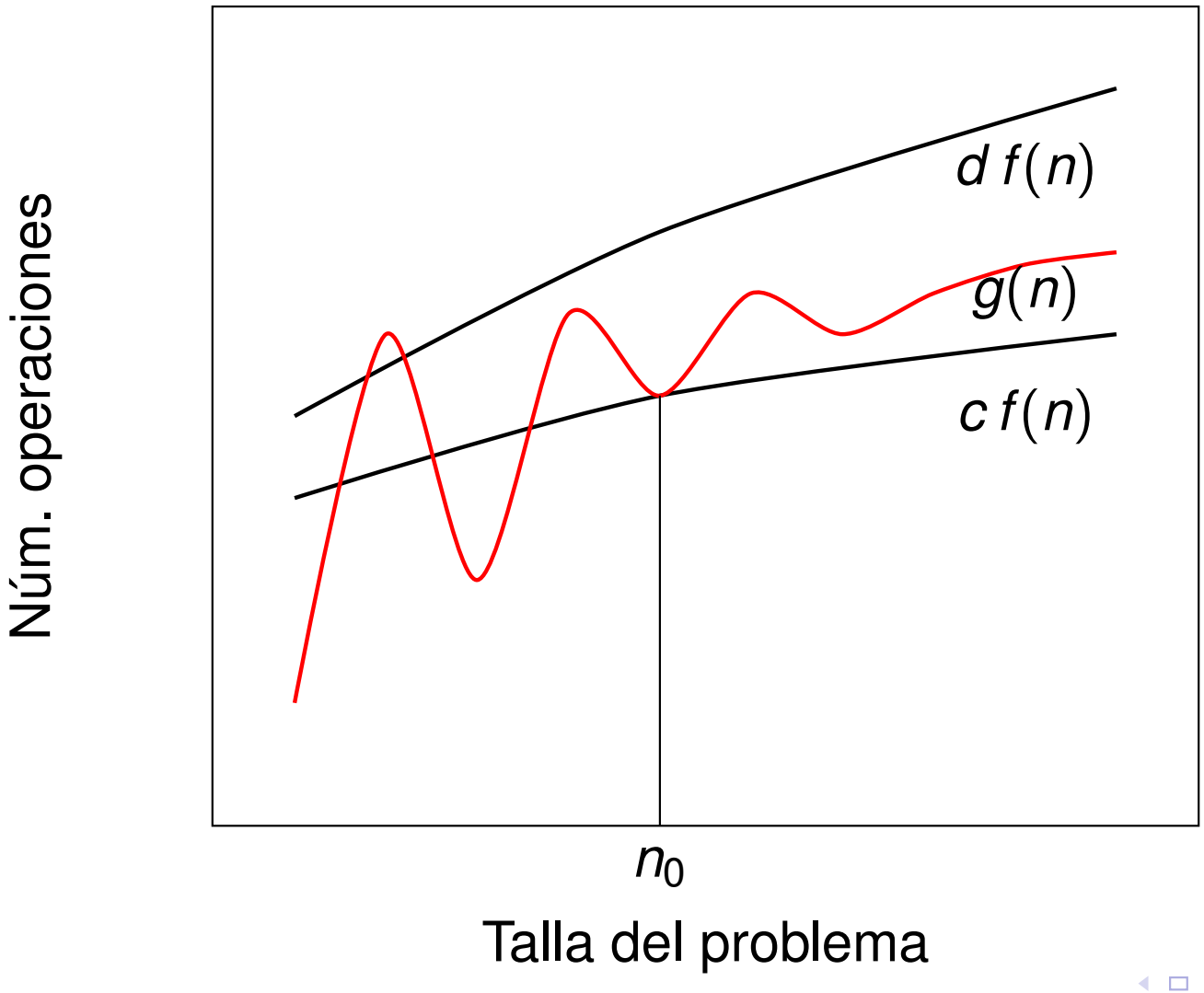
Cota inferior. Notación Ω

- Sea $f : \mathbb{N} \to \mathbb{R}^+$; se define el conjunto $\Omega(f)$ como el conjunto de funciones acotadas inferiormente por un múltiplo de f



Coste exacto. Notación Θ

- Sea $f : \mathbb{N} \to \mathbb{R}^+$; se define el conjunto $\Theta(f)$ como el conjunto de funciones acotadas superior e inferiormente por un múltiplo de f



Propiedades

Propiedades (también válidas para Ω y Θ):

- $f_1 \in O(g_1) \Rightarrow k \cdot f_1 \in O(g_1)$
- $f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(\text{máx}\{g_1, g_2\})$

Ejemplos

- $1000n + 1 \in O(n)$
- $n^2 + \log n \in O(n^2)$
- $n^3 + 2^n + n \log n \in O(2^n)$

Jerarquías

- Las clases más utilizadas son:

$$\underbrace{O(1)}_{\text{constantes}} \subset \underbrace{O(\log\log n)}_{\text{sublogarítmicas}} \subset \underbrace{O(\log n) \subset O(log^{a(>1)} n)}_{\text{logarítmicas}}$$

$$\subset \underbrace{O(\sqrt{n})}_{\text{sublineales}} \subset \underbrace{O(n)}_{\text{lineales}} \subset \underbrace{O(n\log n)}_{\text{lineal-logarítmicas}}$$

$$\subset \underbrace{O(n^2) \subset O(n^{a(>2)})}_{\text{polinómicas}} \subset \underbrace{O(2^n)}_{\text{exponenciales}} \subset \underbrace{O(n!) \subset O(n^n)}_{\text{superexponenciales}}$$

Índice

Cálculo de complejidades

1. Determinar la talla: variable de la función de complejidad que se pretende encontrar. Puede ser:
   - Tamaño del problema
   - Parámetro representativo
2. Determinar los casos mejor y peor (instancias para las que el algoritmo tarda más o menos)
   - Para algunos algoritmos, el caso mejor y el caso peor son el mismo ya que se comportan igualmente para cualquier instancia del mismo tamaño
3. Obtención de las cotas para cada caso, expresadas como clases de notación asintótica

Ejemplos

Detección de palíndromo

```
int es_palindromo(const string & s){
    for(int i=0, i < s.length()/2; i++)
        if ( s[i] != s[s.length()-1-i] )
            return false;
    return true;
}
```

1. Talla: longitud de `s = s.length()`

Ejemplos

Detección de palíndromo

```
int es_palindromo(const string & s){
    for(int i=0, i < s.length()/2; i++)
        if ( s[i] != s[s.length()-1-i] )
            return false;
    return true;
}
```

1. Talla: longitud de `s` = `s.length()`
2. Casos mejor y peor:

Ejemplos

Detección de palíndromo

```
int es_palindromo(const string & s){
    for(int i=0, i < s.length()/2; i++)
        if ( s[i] != s[s.length()-1-i] )
            return false;
    return true;
}
```

1. Talla: longitud de `s` = `s.length()`
2. Casos mejor y peor:
   - Mejor caso: primer y último carácter son distintos

Ejemplos

Detección de palíndromo

```
int es_palindromo(const string & s){
    for(int i=0, i < s.length()/2; i++)
        if ( s[i] != s[s.length()-1-i] )
            return false;
    return true;
}
```

1. Talla: longitud de $s = s.length()$
2. Casos mejor y peor:
   - Mejor caso: primer y último carácter son distintos
   - Peor caso: es palíndromo
3. Cotas:
   - Mejor caso:

Ejemplos

Detección de palíndromo

```
int es_palindromo(const string & s){
    for(int i=0, i < s.length()/2; i++)
        if ( s[i] != s[s.length()-1-i] )
            return false;
    return true;
}
```

1. Talla: longitud de $s = s.length()$
2. Casos mejor y peor:
   - Mejor caso: primer y último carácter son distintos
   - Peor caso: es palíndromo
3. Cotas:
   - Mejor caso: $c_i(n) = 1 \rightarrow c_i(n) \in \Omega(1)$
   - Peor caso:

Ejemplos

Detección de palíndromo

```
int es_palindromo(const string & s){
    for(int i=0, i < s.length()/2; i++)
        if ( s[i] != s[s.length()-1-i] )
            return false;
    return true;
}
```

1. Talla: longitud de `s` = `s.length()`
2. Casos mejor y peor:
   - Mejor caso: primer y último carácter son distintos
   - Peor caso: es palíndromo
3. Cotas:
   - Mejor caso: $c_i(n) = 1 \rightarrow c_i(n) \in \Omega(1)$
   - Peor caso: $c_s(n) = n/2 \rightarrow c_s(n) \in O(n)$

Problema introductorio #2

- ¿Cuál de estas dos funciones es más rápida para vectores muy grandes? Sólo una opción es verdadera
  a. `maximo1` es más rápida
  b. `maximo2` es más rápida
  c. Una u otra, depende del contenido del vector
  d. `maximo2` nunca es más rápida que `maximo1`
  e. `maximo1` nunca es más rápida que `maximo2`

```
int maximo1 (int[] v, int n){
  int maximo=v[0];
  for(int k=0; k<500000; k++)
     for(int i=1; i< n; i++)
        if(v[i]>maximo)
            maximo=v[i];
  return maximo;
}
```

```
int maximo2 (int[] v, int n){
  int maximo=v[0];
  for(int k=0; k<n; k++)
     for(int i=1; i< n; i++)
        if(v[i]>maximo)
            maximo=v[i];
  return maximo;
}
```

Problema introductorio #2

- ¿Cuál de estas dos funciones es más rápida para vectores muy grandes? Sólo una opción es verdadera
  a. `maximo1` es más rápida
  b. **`maximo2` es más rápida**
  c. Una u otra, depende del contenido del vector
  d. `maximo2` nunca es más rápida que `maximo1`
  e. `maximo1` nunca es más rápida que `maximo2`

```
int maximo1 (int[] v, int n){
  int maximo=v[0];
  for(int k=0; k<500000; k++)
     for(int i=1; i< n; i++)
        if(v[i]>maximo)
            maximo=v[i];
  return maximo;
}
```

```
int maximo2 (int[] v, int n){
  int maximo=v[0];
  for(int k=0; k<n; k++)
     for(int i=1; i< n; i++)
        if(v[i]>maximo)
            maximo=v[i];
  return maximo;
}
```

Ejemplos

- La búsqueda binaria permite buscar rápidamente un elemento en un array si sabemos que está ordenado

Búsqueda binaria

```
int busca(int [] v, int n, int z){
    int m;
    int pri=0; int ult=n-1;
    do{
        m=(pri+ult)/2;
        if(v[m] > z)
            ult=m-1;
        else
            pri=m+1;
    } while(pri <= ult && v[m] != z )
    if( v[m] == z)
        return m;
    else
        return -1;
}
```

Ejemplos

Cálculo de complejidad para búsqueda binaria

1. Talla: tamaño del array $= \mathtt{n}$
2. Casos mejor y peor:

Ejemplos

Cálculo de complejidad para búsqueda binaria

1. Talla: tamaño del array $=$ `n`
2. Casos mejor y peor:
   - Mejor caso: `z` está en la posición `(n-1)/2`

Ejemplos

Cálculo de complejidad para búsqueda binaria

1. Talla: tamaño del array $=$ `n`
2. Casos mejor y peor:
   - Mejor caso: `z` está en la posición `(n-1)/2`
   - Peor caso:

Ejemplos

Cálculo de complejidad para búsqueda binaria

1. Talla: tamaño del array $=$ `n`
2. Casos mejor y peor:
   - Mejor caso: `z` está en la posición `(n-1)/2`
   - Peor caso: `z` no está
3. Cotas:
   - Mejor caso:

Ejemplos

Cálculo de complejidad para búsqueda binaria

1. Talla: tamaño del array $=$ `n`
2. Casos mejor y peor:
   - Mejor caso: `z` está en la posición `(n-1)/2`
   - Peor caso: `z` no está
3. Cotas:
   - Mejor caso: $c_i(n) = 1 \rightarrow c_i(n) \in \Omega(1)$
   - Peor caso:

Ejemplos

Cálculo de complejidad para búsqueda binaria

1. Talla: tamaño del array $=$ `n`
2. Casos mejor y peor:
   - Mejor caso: `z` está en la posición `(n-1)/2`
   - Peor caso: `z` no está
3. Cotas:
   - Mejor caso: $c_i(n) = 1 \rightarrow c_i(n) \in \Omega(1)$
   - Peor caso: $c_s(n) = 1 + m \cdot 1$; $m =$ número iteraciones del bucle

Ejemplos

Cálculo de complejidad para búsqueda binaria

iteración	tam. array (`ult-pri+1`)
1	n
2	$\frac{n}{2}$
3	$\frac{n}{4}$
...	..
i	$\frac{n}{2^{i-1}}$
...	...
m	1

En la última iteración, el tamaño del array es 1, luego

$$\frac{n}{2^{m-1}} = 1$$

$$n = 2^{m-1}$$

$$\log_2 n = m - 1$$

$$m = log_2 n + 1$$

Ejemplos

Cálculo de complejidad para búsqueda binaria

1. Talla: tamaño del array $=$ `n`
2. Casos mejor y peor:
   - Mejor caso: `z` está en la posición `(n-1)/2`
   - Peor caso: `z` no está
3. Cotas:
   - Mejor caso: $c_i(n) = 1 \rightarrow c_i(n) \in \Omega(1)$
   - Peor caso: $c_s(n) = 1 + m \cdot 1 = 1 + log_2 n + 1 \rightarrow c_s(n) \in O(\log n)$

Ejercicios

Calcula la complejidad del siguiente fragmento de código (talla = valor de `n`):

Ejercicio 4:

```
for(i = 0, length = 1; i < n-1; i++) {
   for(i1 = i2 = k = i; k < n-1 && a[k] < a[k+1]; k++, i2++);
   if(length < i2 - i1 + 1) length = i2 - i1 + 1; }
```

Ejercicios

Calcula la complejidad del algoritmo de ordenación por inserción directa: *Insertion sort* en `https://visualgo.net/en/sorting`

Ejercicio 5:

```
int ordenacion_insercion(int [] v, int n){
    int x,j;
    for(int i =1; i < n; i++){
        x=v[i]; j=i-1;
        while(j >= 0 && v[j] > x){
            v[j+1]=v[j];
            j-=1
        }
        v[j+1]=x
    }
}
```

Ejercicios

Cálculo de complejidad para inserción directa

1. Talla: tamaño del array $= \mathtt{n}$
2. Casos mejor y peor:
   - Mejor caso: array ordenado ascendentemente
   - Peor caso: array ordenado descendentemente
3. Cotas:
   - Mejor caso: $c_i(n) = n - 1 \rightarrow c_i(n) \in \Omega(n)$
   - Peor caso: $c_s(n) = \sum_{i=1}^{n} \sum_{j=0}^{i-1} 1 \rightarrow c_s(n) \in O(n^2)$

Ejercicios

Calcula la complejidad del algoritmo de ordenación de la burbuja: *Bubble sort* en `https://visualgo.net/en/sorting`

Ejercicio 6:

```
int ordenacion_burbuja(int [] v, int n){
    for(int fin=n; fin >=2 ; fin--)
        for(int i=1; i < fin ; i++)
            if(v[i] < v[i-1]){
                swap(v[i],v[i-1]);
            }
    }
}
```

Ejercicios

Cálculo de complejidad para el algoritmo de la burbuja

1. Talla: tamaño del array $= \mathtt{n}$
2. Casos mejor y peor: no hay
3. Complejidad: $T(n) = \sum_{f=2}^{n} \sum_{i=1}^{f-1} 1 \rightarrow T(n) \in \Theta(n^2)$