Algoritmos y Estructuras de Datos:

Introducción a la Complejidad

Guillermo Román Díez (guillermo.roman@upm.es)
Lars-Åke Fredlund (larsake.fredlund@upm.es)

Universidad Politécnica de Madrid

Curso 2021/2022

Definición de Complejidad

¿qué entendemos por complejidad hablando de **software**?

Definición de Complejidad

¿qué entendemos por complejidad hablando de **software**?

Complejidad

"La complejidad de un programa es una medida (abstracta) de su uso de recursos (tiempo de ejecución, memoria, ...) "

Definición de Complejidad

¿qué entendemos por complejidad hablando de **software**?

Complejidad

"La complejidad de un programa es una medida (abstracta) de su uso de recursos (tiempo de ejecución, memoria, ...) "

- Habitualmente se habla de 2 tipos de complejidad en función de lo que miden:
  - Complejidad **temporal**: tiempo de ejecución
  - Complejidad **espacial**: espacio de memoria utilizado
- La complejidad en tiempo y en espacio suelen estar reñidas
  - Se ahorra tiempo a costa de usar más espacio
  - Se ahorra espacio a costa de usar más tiempo

Definición de Complejidad

¿qué entendemos por complejidad hablando de **software**?

Complejidad

"La complejidad de un programa es una medida (abstracta) de su uso de recursos (tiempo de ejecución, memoria, ...) "

- Habitualmente se habla de 2 tipos de complejidad en función de lo que miden:
  - Complejidad **temporal**: tiempo de ejecución
  - Complejidad **espacial**: espacio de memoria utilizado
- La complejidad en tiempo y en espacio suelen estar reñidas
  - Se ahorra tiempo a costa de usar más espacio
  - Se ahorra espacio a costa de usar más tiempo
- Ejemplo: una caché – algo que usa memoria para guardar resultados anteriores para no repetir cálculos
  - Ejemplo: podemos tener un caché que recuerda si un número natural es un número primo

Complejidad

- La complejidad de un programa se calcula en función de los datos de entrada (tamaño de la entrada o *input size*)
  - Se presentan como una expresión (función) de coste en términos de los datos de entrada que afectan a su complejidad
  - Al crecer el tamaño de la entrada, la complejidad del programa puede crecer o permanecer constante

En la vida real – Let's party!

- Quiero organizar una fiesta en mi casa
- Un parámetro a tener en cuenta es el número de invitados

En la vida real – Let's party!

- Primero tengo que ordenar mi casa – ¿el tiempo de limpiar la casa antes de la fiesta depende del número de invitados?

En la vida real – Let's party!

- Primero tengo que ordenar mi casa – ¿el tiempo de limpiar la casa antes de la fiesta depende del número de invitados?
- **No.** El trabajo es *constante*. Podemos expresar el tiempo de limpieza como una función $f(n)$, donde n es el número de invitados y c es un constante

$$f(n) = c$$

En la vida real – Let's party!

- Primero tengo que ordenar mi casa – ¿el tiempo de limpiar la casa antes de la fiesta depende del número de invitados?
- **No.** El trabajo es *constante*. Podemos expresar el tiempo de limpieza como una función $f(n)$, donde n es el número de invitados y c es un constante

$$f(n) = c$$

- Quizá el tiempo requerido de limpiar la casa **después** de la fiesta sí depende del número de invitados...

En la vida real – Let's party!

- Ahora, quiero mandar una invitación personal a cada persona invitada – ¿depende el tiempo de hacerlo del número de invitados?

En la vida real – Let's party!

- Ahora, quiero mandar una invitación personal a cada persona invitada – ¿depende el tiempo de hacerlo del número de invitados?
- Si:
  - si el tiempo para mandar 1 invitación es $i + c$ segundos
    (i – tiempo para escribir la carta, c – tiempo para ir a un correos)

En la vida real – Let's party!

- Ahora, quiero mandar una invitación personal a cada persona invitada – ¿depende el tiempo de hacerlo del número de invitados?
- Si:
  - si el tiempo para mandar 1 invitación es $i + c$ segundos
    (i – tiempo para escribir la carta, c – tiempo para ir a un correos)
  - entonces el tiempo para mandar 2 invitaciones es $i * 2 + c$ segundos

En la vida real – Let's party!

- Ahora, quiero mandar una invitación personal a cada persona invitada – ¿depende el tiempo de hacerlo del número de invitados?
- Si:
  - si el tiempo para mandar 1 invitación es $i + c$ segundos (i – tiempo para escribir la carta, c – tiempo para ir a un correos)
  - entonces el tiempo para mandar 2 invitaciones es $i * 2 + c$ segundos
  - ...

En la vida real – Let's party!

- Ahora, quiero mandar una invitación personal a cada persona invitada – ¿depende el tiempo de hacerlo del número de invitados?
- Si:
  - si el tiempo para mandar 1 invitación es $i + c$ segundos (i – tiempo para escribir la carta, c – tiempo para ir a un correos)
  - entonces el tiempo para mandar 2 invitaciones es $i * 2 + c$ segundos
  - ...
  - y el tiempo de mandar n invitaciones es $i * n + c$ segundos
- El tiempo gastado $f(n)$ es una función *lineal* con respecto al número de invitados:

$$f(n) = i * n * + c$$

En la vida real – Let's party!

- Un vez en la fiesta hay que saludar: cuando una persona invitada llega a la fiesta, yo la saludo, y también lo hacen las otras personas invitadas que hayan llegado antes
- Si un saludo tarda s segundos, ¿cuánto tiempo tardan todos los saludos?

En la vida real – Let's party!

- Un vez en la fiesta hay que saludar: cuando una persona invitada llega a la fiesta, yo la saludo, y también lo hacen las otras personas invitadas que hayan llegado antes
- Si un saludo tarda s segundos, ¿cuánto tiempo tardan todos los saludos?
- En total, ¿cuántos saludos hay?
  - 1 invitado $\Rightarrow$ 1 saludo (yo y el invitado)
  - 2 invitados $\Rightarrow 2 + 1 = 3$ saludos
  - 3 invitados $\Rightarrow 3 + 2 + 1 = 6$
  - 4 invitados $\Rightarrow 4 + 3 + 2 + 1 = 10$
  - ...
  - n invitados $\Rightarrow n + (n-1) + \ldots + 2 + 1 =$?

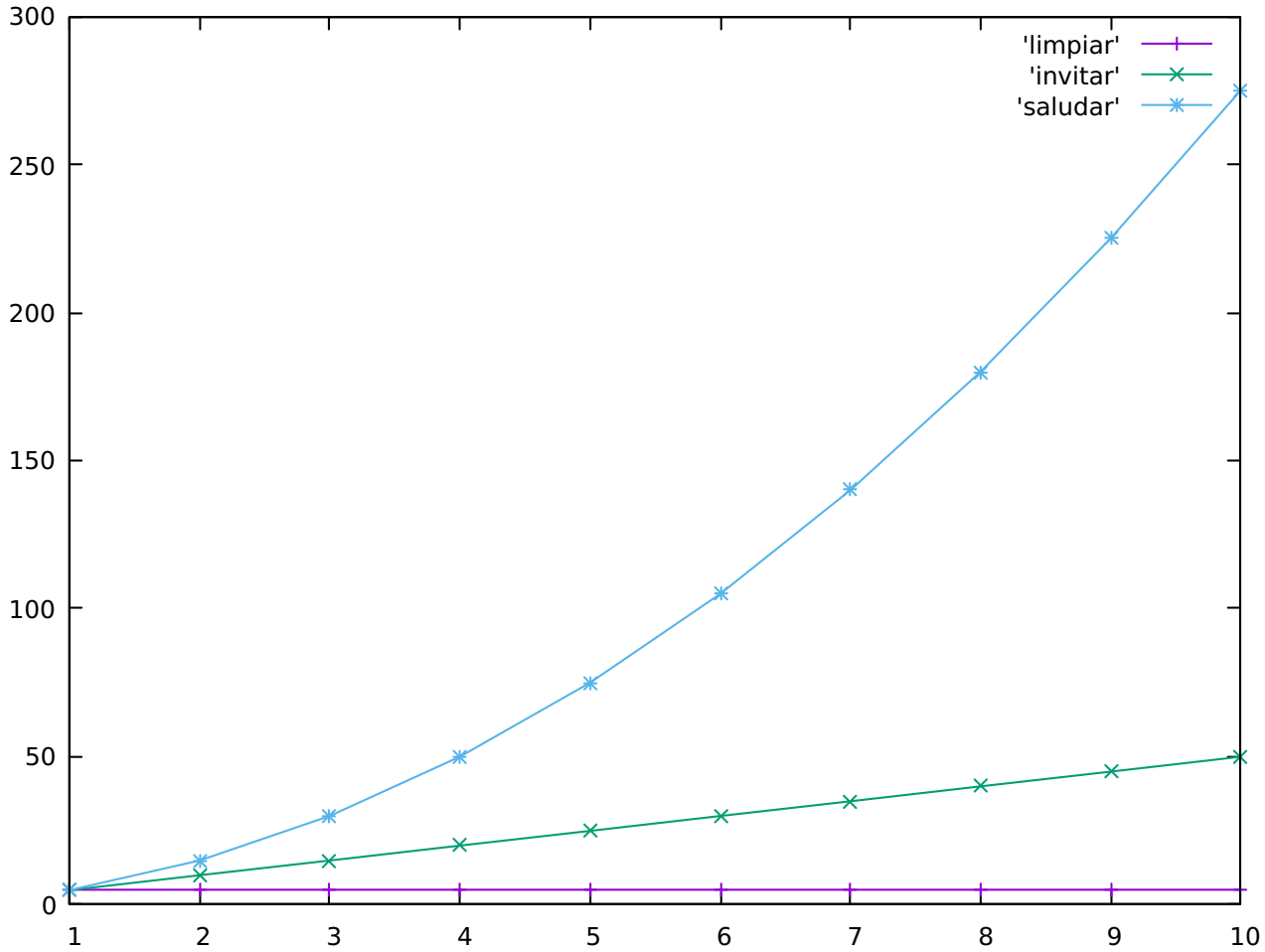
En la vida real – Let's party!

- Un vez en la fiesta hay que saludar: cuando una persona invitada llega a la fiesta, yo la saludo, y también lo hacen las otras personas invitadas que hayan llegado antes
- Si un saludo tarda s segundos, ¿cuánto tiempo tardan todos los saludos?
- En total, ¿cuántos saludos hay?
  - 1 invitado $\Rightarrow$ 1 saludo (yo y el invitado)
  - 2 invitados $\Rightarrow 2 + 1 = 3$ saludos
  - 3 invitados $\Rightarrow 3 + 2 + 1 = 6$
  - 4 invitados $\Rightarrow 4 + 3 + 2 + 1 = 10$
  - ...
  - n invitados $\Rightarrow n + (n-1) + \ldots + 2 + 1 =$?
- n invitados: $(n+1) * n/2$

En la vida real – Let's party!

- Un vez en la fiesta hay que saludar: cuando una persona invitada llega a la fiesta, yo la saludo, y también lo hacen las otras personas invitadas que hayan llegado antes
- Si un saludo tarda s segundos, ¿cuánto tiempo tardan todos los saludos?
- En total, ¿cuántos saludos hay?
  - 1 invitado $\Rightarrow$ 1 saludo (yo y el invitado)
  - 2 invitados $\Rightarrow 2+1=3$ saludos
  - 3 invitados $\Rightarrow 3+2+1=6$
  - 4 invitados $\Rightarrow 4+3+2+1=10$
  - ...
  - n invitados $\Rightarrow n+(n-1)+\ldots+2+1=?$
- n invitados: $(n+1)*n/2$
- El tiempo: $f(n)=s*(n+1)*n/2 \equiv s*(n^2+n)/2$
- El tiempo de todos los saludos es *cuadrático* con respecto al valor de n

Constante, lineal y cuadrático



- El gráfico asume que el tiempo para realizar cualquier tarea es 5
- Estos cálculos exactos para calcular el tiempo son complicados – ¿podemos simplificar?

Calculo de la Complejidad en la Informática

- Análisis experimental
  - Se hace un estudio estadístico de un programa concreto, en un entorno de ejecución concreto, con un compilador concreto, en un sistema operativa concreto...

Calculo de la Complejidad en la Informática

- Análisis experimental
  - Se hace un estudio estadístico de un programa concreto, en un entorno de ejecución concreto, con un compilador concreto, en un sistema operativa concreto...
- Análisis teórico
  - Estudia los programas independientemente del entorno de ejecución
  - Se pueden analizar algoritmos en pseudo-código o programas concretos
  - El objetivo es obtener los ordenes de magnitud de la complejidad
  - Las constantes no son relevantes a la hora de obtener la complejidad asintótica
  - Esas constantes pueden ser útiles a la hora de extrapolar los resultados teórico a entornos concretos (p.e. WCET)

En la Informática - una aproximación

- Cada operación (+, *, -, =, if, method call, etc) consume una unidad de tiempo (t)
- Un acceso a la memoria consume una unidad de tiempo

Ejemplo

```
int max(int i, int j) {
  if (i > j) return i;
  else return j;
}
```

- Cuantos unidades de tiempo "cuesta" una llamada `max(1,3)`?

Ejemplo

```
int max(int i, int j) {
  if (i > j) return i;
  else return j;
}
```

- Cuantos unidades de tiempo "cuesta" una llamada `max(1,3)`?
- ≈ 7:
  - 1 (llamada) + 2 (accesos a `i` y `j`) + 1 (comparación `i > j`) + 1 (if) + 2 (acceso a `j`, `return`)

Ejemplo

```
int max(int i, int j) {
  if (i > j) return i;
  else return j;
}
```

- Cuantos unidades de tiempo "cuesta" una llamada `max(1,3)`?
- ≈ 7:
  - 1 (llamada) + 2 (accesos a `i` y `j`) + 1 (comparación `i > j`) + 1 (if) + 2 (acceso a `j`, `return`)
- $f(n) = 7$
- Pero, ¿qué es n?

Ejemplo

Ejemplo

Un método que busca si un elemento está en un array

```
static <E> boolean member(E e, E[] arr) {
  boolean found = false;
  for (int i=0; i<arr.length && !found;; i++) {
    found = e.equals(arr[i]);
  }
  return found;
}
```

- El "tamaño de la entrada" es el tamaño del array en el que se hace la búsqueda
- El tamaño del elemento a buscar *no suele* ser relevante

Ejemplo

```
static <E> boolean member(E e, E[] arr) {
  boolean found = false;
  for (int i=0; i<arr.length && !found;; i++) {
    found = e.equals(arr[i]);
  }
  return found;
}
```

- ¿Cuántas unidades de tiempo "cuesta" una llamada `member(5,[1,2,3])`?

Ejemplo

```
static <E> boolean member(E e, E[] arr) {
  boolean found = false;
  for (int i=0; i<arr.length && !found;; i++) {
    found = e.equals(arr[i]);
  }
  return found;
}
```

- ¿Cuántas unidades de tiempo "cuesta" una llamada `member(5,[1,2,3])`?
  - El coste de una "ronda" r sin salida en el bucle for: $r \approx 11$:

$$4\ (i < arr.length) + 1\ (\texttt{if}) + 4\ (e.equals(arr[i])) + 2\ (\texttt{i++})$$

  - El coste total:

$$1\ (call) + 1\ (iniciar\ for) + 3 * r + 1\ (return) \equiv 3 + r * 3 \equiv 36$$

Ejemplo

```
static <E> boolean member(E e, E[] arr) {
  boolean found = false;
  for (int i=0; i<arr.length && !found;; i++) {
    found = e.equals(arr[i]);
  }
  return found;
}
```

- ¿Cuántas unidades de tiempo "cuesta" una llamada `member(5,[1,2,3])`?
  - El coste de una "ronda" r sin salida en el bucle for: $r \approx 11$:

$$4\ (i < arr.length) + 1\ (\texttt{if}) + 4\ (e.equals(arr[i])) + 2\ (\texttt{i++})$$

  - El coste total:

$$1\ (call) + 1\ (iniciar\ for) + 3 * r + 1\ (return) \equiv 3 + r * 3 \equiv 36$$

- En general: $f(n) = 3 + 11 * n$ donde n es el tamaño del array, si el elemento `i` **no esta presente** en `arr`

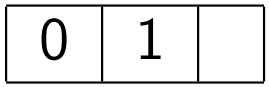
Complejidad – diferentes casos

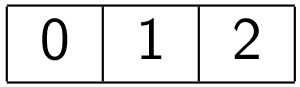
- No sólo el tamaño de los datos es relevante, también la **distribución de los datos** puede ayudar (o perjudicar)
- Podemos encontrarnos diferentes escenarios para `member(E e, E[] arr)`:
  - El elemento a buscar es siempre el primero en ser accedido
  - El elemento a buscar puede estar en cualquier sitio (y todos los casos son equiprobables)
  - El elemento a buscar está siempre el último o no está
  - El elemento a buscar estás más veces el primero que en otra posición
  - ...
- Podemos estudiar la complejidad:
  - En el caso mejor (lower-bound)
  - En el caso (pro-)medio
  - En el caso peor (upper-bound)
  - En el caso amortizado

Complejidad

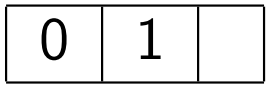
- La variabilidad de los casos experimentales o estadísticos es grande
- El estudio del caso peor (upper-bound) nos da resultados más precisos y fiables
  - El análisis del peor de los casos nos permite "razonar" que el código podrá ejecutar en un cierto hardware o en un cierto "tiempo"
  - Conocer el peor de los casos nos permite movernos en un escenario "seguro"
- La **complejidad asintótica** implica calcular la complejidad cuando el tamaño de los datos de entrada tiende a infinito

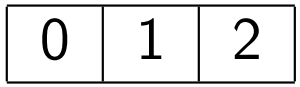
El caso amortizado

- El caso amortizado no razona sobre el coste de llamada aisladas c, sino sobre el coste de medio de una llamada c_i en una secuencia de llamadas $c_1, \ldots, c_n$
- Ejemplo motivador: añadir un elemento x a la ultima posición no ocupada en un array a – a.add(x):
- Ejecutar a.add(2) cuando a es 

es facil 

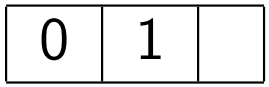
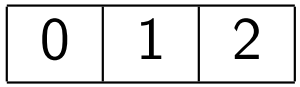
El caso amortizado

- El caso amortizado no razona sobre el coste de llamada aisladas c, sino sobre el coste de medio de una llamada c_i en una secuencia de llamadas $c_1, \ldots, c_n$
- Ejemplo motivador: añadir un elemento x a la ultima posición no ocupada en un array a – a.add(x):
- Ejecutar a.add(2) cuando a es 

es facil 

- ¿Cómo se ejecuta después a.add(3)?

El caso amortizado

- El caso amortizado no razona sobre el coste de llamada aisladas c, sino sobre el coste de medio de una llamada c_i en una secuencia de llamadas $c_1, \ldots, c_n$
- Ejemplo motivador: añadir un elemento x a la ultima posición no ocupada en un array a – a.add(x):
- Ejecutar a.add(2) cuando a es  es facil 
- ¿Cómo se ejecuta después a.add(3)?
- Normalmente creamos un array nuevo con el doble de tamaño y copiamos los elementos al array nuevo, y insertamos 3:

- El coste es *lineal* en el peor caso – mover todos los elementos de un array a otro

El caso amortizado

- Pero, ¿cuándo ocurre el caso peor?
- **Sólo ocurre cuando el array esta lleno**
- ¿Qué sabemos sobre las secuencias de llamadas $c_1, \ldots, c_n$ que contienen el caso peor?

El caso amortizado

- Pero, ¿cuándo ocurre el caso peor?
- **Sólo ocurre cuando el array esta lleno**
- ¿Qué sabemos sobre las secuencias de llamadas $c_1, \ldots, c_n$ que contienen el caso peor?
- Por ejemplo, después que ha tocado el caso peor no puede ocurrir otro caso peor hasta que se ha añadido n elementos más
- Por ejemplo, no hay ninguna secuencia de llamadas $\ldots; c_i; c_j; \ldots$ con dos "casos peores" consecutivos
- En el análisis amortizado pretende identificar cuáles son las secuencias de llamadas posibles y su coste

El caso amortizado para add: análisis

- Asumimos que el tamaño del array es n, estudiamos una secuencia de $n+1$ llamadas a add:

  `a.add(x1); a.add(x2); ...; a.add(xn); a.add(xn+1);`

- La llamada `a.add(xn+1)` tiene complejidad $c1 * n$ (lineal)
- Las demás llamadas tiene complejidad constante $c2$
- Entonces el tiempo de ejecución $f(n)$ de media de una llamada dentro la secuencia es:

$$f(n) = (c1 * n + c2 + ... + c2)/n + 1 = n * (c1 + c2)/n + 1$$

(complejidad constante)

La Complejidad en Java

- Los interfaces no implican ninguna medida de complejidad
- La complejidad va asociada a la implementación de los métodos de la interfaz en una clase
- En la especificación del interfaz se puede *exigir* que ciertos métodos se implementen con una complejidad determinada

Funciones de Complejidad

- Función **constante**

  $f(n) = c$

- Función **polinomial**

  $f(n) = c_1 * n^{e_1} + ... + c_m * n^{e_m}$

  - El grado del polinomio lo marca el exponente de mayor valor
  - Entre las funciones polinomiales encontramos la lineal (grado 1), cuadrática (grado 2) y cúbica (grado 3)

- Función **logarítmica**

  $f(n) = log_2(n)$

- Función **exponencial**

  $f(n) = c^n$

Notación $O()$

- El objetivo intuitivo es establecer la complejidad de una función en términos de n con una función proporcional que la acota asintóticamente
  - Ignorando los factores constantes y de orden menor
- Decimos que un método tiene un **orden de complejidad** $O(n)$ cuando O(n) acota asintóticamente la función de complejidad del método
- Una escala de complejidad de menor a mayor:

Constante	$O(1)$
Logarítmica	$O(log(n))$
Lineal	$O(n)$
N-Log-N	$O(n * log(n))$
Cuadrática	$O(n^2)$
Cúbica	$O(n^3)$
Polinomial de orden k	$O(n^k)$
Exponencial	$O(2^n)..O(m^n)$

Formal definition of big-Oh

- Usamos la notación $f(x) = O(g(x))$ cuando $x \to \infty$ si el valor de $f(x) \leq c * g(x)$, donde c es una constante, para todos los valores x suficientement grandes
- Estamos interesados en el caso cuando x es muy grande (complejidad asintótica). Podemos buscar $g(x)$ simplificando $f(x)$:
  1. Si f(x) es una suma de factores $t_1(x) + \ldots + t_n(x)$ podemos borrar todos los factores excepto el que crece mas rápido
  2. Si f(x) es un producto de factores $t_1(x) \times \ldots \times t_n(x)$ podemos borrar todos los factores que son constantes
- Por ejemplo: podemos simplificar $f(x) = 3x^4 + 2x^3 + x * log_2(x) + 4$ $\Rightarrow 3x^4$ (regla 1) $\Rightarrow x^4$ (regla 2) y entonces obtener $O(x^4)$

Algunas cifras

Función	n = 32	n = 64	n = 128
$log_2(n)$	5	6	7
n	32	64	128
$n * log_2(n)$	160	384	896
n^2	1.024	4.096	16.384
n^3	32.768	262.144	2.097.152
2^n	4.294.967.296	18.446.744.073.709.551.616	(no cabe)

Algunas cifras

Función	n = 32	n = 64	n = 128
$log_2(n)$	5	6	7
n	32	64	128
$n * log_2(n)$	160	384	896
n^2	1.024	4.096	16.384
n^3	32.768	262.144	2.097.152
2^n	4.294.967.296	18.446.744.073.709.551.616	(no cabe)

Pregunta

¿tan grande es 2^{128}?

Algunas cifras

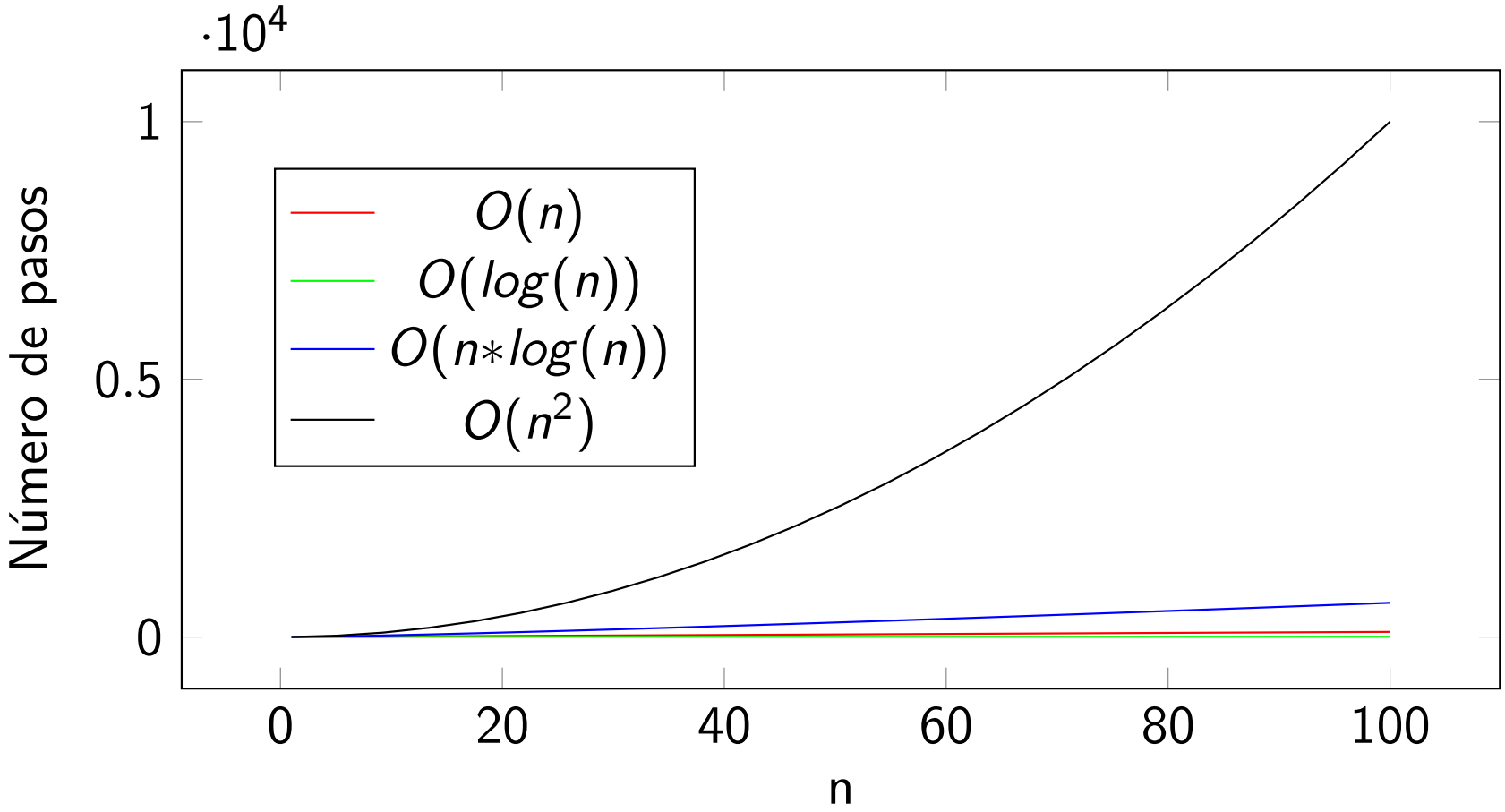
Función	n = 32	n = 64	n = 128
$log_2(n)$	5	6	7
n	32	64	128
$n * log_2(n)$	160	384	896
n^2	1.024	4.096	16.384
n^3	32.768	262.144	2.097.152
2^n	4.294.967.296	18.446.744.073.709.551.616	(no cabe)

Pregunta

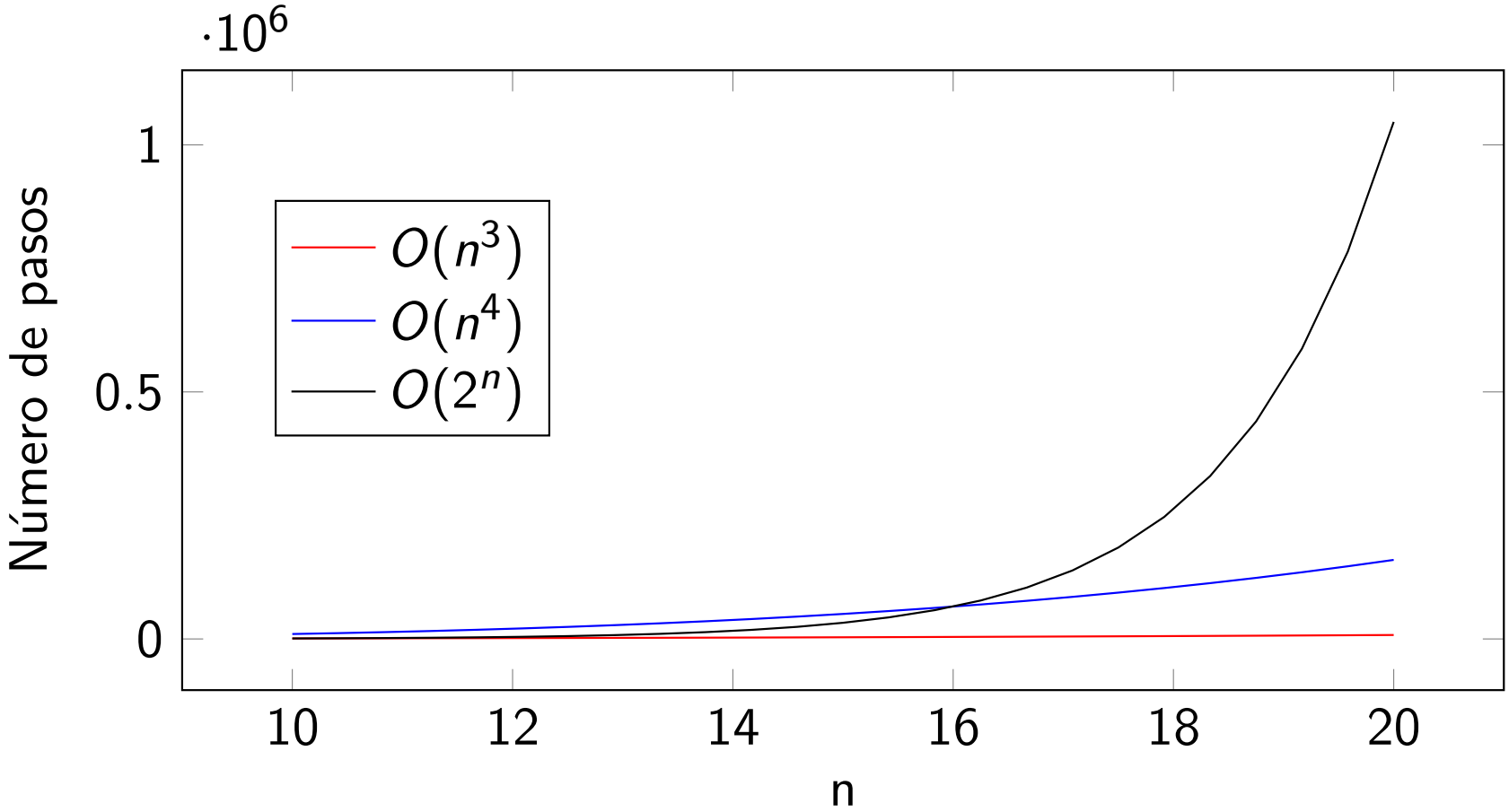
¿tan grande es 2^{128}?

2^{128} : 340.282.366.920.938.463.463.374.607.431.768.211.456

Gráficamente



Gráficamente



Algunos ejercicios de examen

Indicar la complejidad del método:

```
<E> boolean m1(IndexedList<E> l, E e) {
  boolean res = false;
  if (!l.isEmpty())
    res = l.get(0).equals(e);
  }
  return res;
}
```

Algunos ejercicios de examen

Indicar la complejidad del método:

```
<E> boolean m1(IndexedList<E> l, E e) {
  boolean res = false;
  if (!l.isEmpty())
    res = l.get(0).equals(e);
  }
  return res;
}
```

$O(1)$ – constante

Exam problems

Indicar la complejidad del método:

```
void m2(IndexedList<E> l) {
  int i = 0;
  while (i < l.size()) {
    int j = 0;
    while (j < l.size()) {
      ++j;
    }
    ++i;
  }
}
```

Exam problems

Indicar la complejidad del método:

```
void m2(IndexedList<E> l) {
  int i = 0;
  while (i < l.size()) {
    int j = 0;
    while (j < l.size()) {
      ++j;
    }
    ++i;
  }
}
```

$O(n^2)$ – cuadrático

Exam problems

Indicar la complejidad del método:

```
<E> int method (IndexedList<E> l) {
  int i = l.size();
  int counter = 0;
  while (i > 0) {
    counter ++;
    i = i / 2;
  }
  return counter;
}
```

Exam problems

Indicar la complejidad del método:

```
<E> int method (IndexedList<E> l) {
   int i = l.size();
   int counter = 0;
   while (i > 0) {
      counter ++;
      i = i / 2;
   }
   return counter;
}
```

$O(log\ N)$ – logarítmico

Exam problems

Indicar la complejidad del método:

```
<E> int method (IndexedList<E> l) {
  int counter = 0;
  for (int i = 0; i < l.size(); i++) {
    int j = l.size();
    while (j > 0) {
      counter ++;
      j = j / 2;
    }
  }
  return counter;
}
```

Exam problems

Indicar la complejidad del método:

```
<E> int method (IndexedList<E> l) {
  int counter = 0;
  for (int i = 0; i < l.size(); i++) {
    int j = l.size();
    while (j > 0) {
      counter ++;
      j = j / 2;
    }
  }
  return counter;
}
```

$O(N * log\ N)$