Tema 1: Análisis de la eficiencia de los algoritmos

Slides adaptadas a partir del original de Enrique Martín
Estructuras de Datos y Algoritmos (EDA)
Grado en Desarrollo de Videojuegos
Fac. Informática

Contenidos

Introducción

Introducción

- Análisis de eficiencia
  1. **Análisis de eficiencia de los algoritmos**
- Algoritmos
  2. Divide y vencerás (DV), o *Divide-and-Conquer*
  3. Vuelta atrás (VA), o *backtracking*
- Estructuras de datos
  4. Especificación e implementación de TADs
  5. Tipos de datos lineales
  6. Tipos de datos arborescentes
  7. Diccionarios
  8. Aplicación de TADs

Introducción

- Aproximadamente cada año y medio se duplica el número de instr/seg que son capaces de ejecutar los computadores.
- ¿Basta esperar unos años para que problemas que hoy necesitan horas de cálculo puedan resolverse en segundos?
- No. Para un algoritmo ineficiente ningún avance en la velocidad de las máquinas podría conseguir tiempos aceptables.
- Habitualmente, el factor que determina lo que es soluble en un tiempo razonable es precisamente el **algoritmo** elegido.

Introducción

- En este capítulo estudiamos:
  - Cómo medir la **eficiencia** de los algoritmos, y,
  - Cómo comparar la eficiencia de distintos algoritmos para un mismo problema.
- Después de la **corrección**, conseguir eficiencia debe ser el principal objetivo del programador.
- Mediremos la eficiencia en **tiempo de ejecución**.
  - Los mismos conceptos son aplicables a la medición de otros recursos $\Rightarrow$ memoria, energía, etc.

Ejemplo del cálculo de eficiencia

Consideremos la siguiente función *genérica* para buscar si un valor `x` aparece en un vector `v`.

```
template <class T>
bool search(vector<T> const& v, T const& x) {
  int i = 0;
  while (i < v.size() && v[i] != x) {
    ++i;
  }
  return i < v.size();
}
```

- Si el valor aparece en la primera posición del vector, consumirá muy poco tiempo → *caso mejor*.
- Si el valor no aparece en el vector deberá recorrer todo el vector y consumirá mucho tiempo → *caso peor*.

Ejemplo del cálculo de eficiencia

- Una manera de medir los recursos utilizados por un algoritmo es contar **cuántas instrucciones de cada tipo** se ejecutan y multiplicar por el **consumo de recursos que necesita cada instrucción**:
  - t_a: tiempo de una asignación
  - t_c: tiempo de una comparación
  - t_i: tiempo de un incremento/decremento
  - t_v: tiempo de un acceso a vector
- El coste del algoritmo `search` será una función que, dado el tamaño n del vector `v`, devuelve la cantidad de recursos consumidos durante la ejecución de `search`.
- El **tamaño** n del vector será el **número de elementos que contiene**.

Ejemplo del cálculo de eficiencia

Podemos calcular la función de coste para el algoritmo anterior en dos contextos (*ignoramos el coste de* $v.size()$):

- **Caso mejor**: *el elemento está en la primera posición*
  $T_{min}(n) = \underbrace{t_a}_{L3} + \underbrace{3t_c + t_v}_{L4} + \underbrace{t_c}_{L7} = t_a + 4t_c + t_v \in \mathcal{O}(1)$
- **Caso peor**: *el elemento no aparece en el vector*

$$\begin{aligned} T_{max}(n) &= \underbrace{t_a}_{L3} + \underbrace{n(3t_c + t_v)}_{L4} + \underbrace{t_c}_{L4} + \underbrace{nt_i}_{L5} + \underbrace{t_c}_{L7} \\ &= n(3t_c + t_v + t_i) + 2t_c + t_a \in \mathcal{O}(n) \end{aligned}$$

- Se observan claramente los tres factores de los que en general depende el tiempo de ejecución de un algoritmo:
    1. El **tamaño** de los datos de entrada, simbolizado aquí por la longitud n del vector.
    2. El **contenido** de los datos de entrada, que en el ejemplo hace que el tiempo para diferentes vectores del mismo tamaño esté comprendido entre los valores T_{min} y T_{max}.
    3. El código generado por el **compilador** y el **computador** concreto utilizados, que afectan a los tiempos elementales (t_a, t_c, t_i y t_v).

- Para poder comparar algoritmos independientemente del valor de los datos de entrada, el segundo factor debemos eliminarlo:
  - O bien midiendo solo el **caso peor**, es decir la ejecución que tarde más tiempo de todos los ejemplares de tamaño n.
  - O bien midiendo todos los casos de tamaño n y calculando el tiempo del **caso promedio** (exige conocer la probabilidad de cada caso).
- Nos concentraremos en el caso peor por dos razones:
  1. El caso peor establece una cota superior fiable para **todos** los casos del mismo tamaño.
  2. El caso peor es más fácil de calcular.

- El caso promedio es más difícil de calcular pero puede ser más informativo.
    - Exige conocer la probabilidad de cada caso.
- Raramente puede ser útil conocer el **caso mejor** de un algoritmo para un tamaño n dado $\Rightarrow$ *cota inferior*.
- El tercer factor impide comparar algoritmos escritos en diferentes lenguajes, traducidos por diferentes compiladores, o ejecutados en diferentes máquinas $\Rightarrow$ **Lo ignoramos**.
- Mediremos la eficiencia de un algoritmo en función del **tamaño** de los datos de entrada $\Rightarrow$ **medida asintótica**.

Medidas asintóticas de la eficiencia

- El **criterio asintótico** para medir la eficiencia de los algoritmos tiene como objetivo comparar algoritmos **independientemente** de los lenguajes en que están escritos, de las máquinas en que se ejecutan y del valor concreto de los datos que reciben como entrada.
- Tan solo considera importante el **tamaño** de dichos datos.
- Para cada problema habrá que definir qué se entiende por tamaño del mismo.

Medidas asintóticas de la eficiencia

- Se basa en tres principios:
  1. El coste o eficiencia es una función que solo depende del tamaño de la entrada, e.g. $f(n) = n^2$.
  2. Las constantes multiplicativas o aditivas no se tienen en cuenta, e.g. $f(n) = n^2$ y $g(n) = 3n^2 + 27$ se consideran costes equivalentes.
  3. La comparación entre funciones de coste se hará para valores de n **suficientemente grandes**, es decir los costes para tamaños pequeños se consideran irrelevantes.

Medidas asintóticas de la eficiencia. El orden $\mathcal{O}$

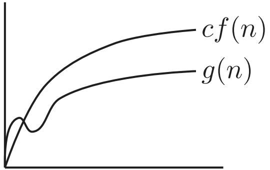
El orden $\mathcal{O}$: Conjuntos de funciones acotadas superiormente

Sea una función $f : \mathbb{N} \to \mathbb{R}^+$. El conjunto de las funciones *del orden de* $f(n)$, llamado $\mathcal{O}(f(n))$, se define como:

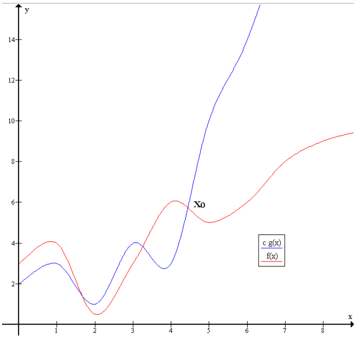
$$\mathcal{O}(f(n)) = \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}.\ \forall n \geq n_0.\ g(n) \leq cf(n)\}$$

En otras palabras, es el conjunto de todas las funciones $g(n)$ tales que a partir de un *n suficientemente grande* son inferiores a $f(n)$.

Si $g \in \mathcal{O}(f(n))$ diremos que g **es del orden de** $f(n)$.



Medidas asintóticas de la eficiencia



[1]

$f(n) \in \mathcal{O}(g(n))$ ya que existe $c \in \mathbb{R}^+$ y $n_0 = 5 \in \mathbb{N}$ tales que

$$f(n) \leq cg(n) \text{ para todo } n \geq n_0$$

[1] https://en.wikipedia.org/wiki/Big_O_notation#/media/File:Big-O-notation.png

Medidas asintóticas de la eficiencia. Ejemplos

$\mathbf{f(n) = 5}$

- $f(n) \in \mathcal{O}(1)$ $[\forall n \in \mathbb{N}, c = 5]$
- $f(n) \in \mathcal{O}(n)$ $[n_0 \geq 5, c = 1]$
- $f(n) \in \mathcal{O}(n^2)$ $[n_0 \geq 3, c = 1]$
- $f(n) \in \mathcal{O}(5^n)$ $[n_0 \geq 1, c = 1]$

$\mathbf{g(n) = 10n + 5}$

- $g(n) \notin \mathcal{O}(1)$
- $g(n) \in \mathcal{O}(n)$ $[n_0 \geq 1, c = 20]$
- $g(n) \in \mathcal{O}(n^2)$ $[n_0 \geq 11, c = 1]$
- $g(n) \in \mathcal{O}(5^n)$ $[n_0 \geq 2, c = 1]$

Como se puede ver, una función tiene varias cotas superiores. Estamos interesados en aquella **cota superior más ajustada**: $f(n) \in \mathcal{O}(1)$, $g(n) \in \mathcal{O}(n)$.

Medidas asintóticas de la eficiencia

- Si el tiempo de ejecución $g(n)$ de una implementación concreta de un algoritmo es del orden de $f(n)$, entonces el tiempo $g'(n)$ de cualquier otra implementación del mismo que difiera de la anterior en el lenguaje, el compilador, o/y la máquina empleada, también será del orden de $f(n)$.
- Por tanto, el coste $\mathcal{O}(f(n))$ expresa la eficiencia del algoritmo *per se*, no el de una implementación concreta del mismo.

Órdenes de complejidad

A cada familia $\mathcal{O}(f(n))$ se le denomina **clase de complejidad** u **orden de complejidad**. Dado que por definición las constantes multiplicativas o aditivas no afectan, se elige como representante del orden $\mathcal{O}(f(n))$ a la función más sencilla posible:

- $\mathcal{O}(1)$: constante
- $\mathcal{O}(log\ n)$: logarítmico
- $\mathcal{O}(n)$: lineal
- $\mathcal{O}(n.log\ n)$: cuasi-lineal
- $\mathcal{O}(n^2)$: cuadrático
- $\mathcal{O}(n^k)$: polinomial
- $\mathcal{O}(2^n)$: exponencial
- $\mathcal{O}(n!)$: factorial

Jerarquía de órdenes de complejidad

Los distintos órdenes de complejidad se pueden ordenar por inclusión, es decir, «de mejor a peor»:

$$\underbrace{\underbrace{\mathcal{O}(1) \subset \mathcal{O}(log\ n) \subset \mathcal{O}(n) \subset \mathcal{O}(n.log\ n) \subset \mathcal{O}(n^2)}_{\text{razonables en la práctica}} \subset \mathcal{O}(n^k) \ldots \subset}_{\text{tratables}} \underbrace{\mathcal{O}(2^n) \subset \mathcal{O}(3^n) \ldots \subset \mathcal{O}(n!)}_{\text{intratables}}$$

Importancia del orden de complejidad

- Es importante entender las implicaciones prácticas de que el coste de un algoritmo pertenezca a una u otra clase de complejidad.
- La siguiente figura muestra el crecimiento de algunas de estas funciones, suponiendo que expresan un tiempo en milisegundos ($ms =$ milisegundos, $s =$ segundos, $m =$ minutos, $h =$ horas, etc.).

n	$\log_{10} n$	n	$n \log_{10} n$	n^2	n^3	2^n
10	1 ms	10 ms	10 ms	0,1 s	1 s	1,02 s
10^2	2 ms	0,1 s	0,2 s	10 s	16,67 m	$4{,}02 * 10^{20}$ sig
10^3	3 ms	1 s	3 s	16,67 m	11,57 d	$3{,}4 * 10^{291}$ sig
10^4	4 ms	10 s	40 s	1,16 d	31,71 a	$6{,}3 * 10^{3000}$ sig
10^5	5 ms	1,67 m	8,33 m	115,74 d	317,1 sig	$3{,}16 * 10^{30093}$ sig
10^6	6 ms	16,67 m	1,67 h	31,71 a	317 097,9 sig	$3{,}1 * 10^{301020}$ sig

Figura: Crecimiento de distintas funciones de complejidad

Importancia del orden de complejidad

- Obsérvese la **extraordinaria eficiencia** de los algoritmos de coste $\mathcal{O}(\log n)$: pasar de un tamaño de $n = 10$ a $n = 1\,000\,000$ solo hace que el tiempo crezca de 1 ms a 6.
  - La búsqueda binaria en un vector ordenado, y la búsqueda en ciertas estructuras de datos de este curso, tienen este coste en el caso peor.
- Por otro lado, los algoritmos de coste $\mathcal{O}(2^n)$ son **prácticamente inútiles**: mientras que un problema de tamaño $n = 10$ se resuelve en aprox. un segundo, la edad del universo conocido ($1,4 \times 10^8$ siglos) sería totalmente insuficiente para resolver uno de tamaño $n = 100$.
  - Algunos algoritmos de *vuelta atrás* que veremos en este curso tienen ese coste en el caso peor.

Importancia del orden de complejidad

- Esta tabla confirma la afirmación hecha al comienzo de este tema de que para ciertos algoritmos es inútil esperar a que los computadores sean más rápidos.
- Es más productivo invertir esfuerzo en diseñar **mejores algoritmos** para ese problema.

Importancia del orden de complejidad

- Hagamos el siguiente experimento: supongamos seis algoritmos con los costes anteriores, tales que tardan todos ellos 1 hora en resolver un problema de tamaño $n = 100$.
  - ¿Qué ocurre si duplicamos la velocidad del computador? O lo que es lo mismo, ¿qué ocurre si duplicamos el tiempo disponible?

$\mathcal{O}$	$t = 1h.$	$t = 2h.$
$\mathcal{O}(\log n)$	$n = 100$	$n = 10\,000$
$\mathcal{O}(n)$	$n = 100$	$n = 200$
$\mathcal{O}(n \log n)$	$n = 100$	$n = 178$
$\mathcal{O}(n^2)$	$n = 100$	$n = 141$
$\mathcal{O}(n^3)$	$n = 100$	$n = 126$
$\mathcal{O}(2^n)$	$n = 100$	$n = 101$

Importancia del orden de complejidad

- El de coste logarítmico es capaz de resolver problemas 100 veces más grandes
- El de coste exponencial resuelve un tamaño prácticamente igual al anterior!
- Los de coste $\mathcal{O}(n)$ y $\mathcal{O}(n \log n)$ se comportan de acuerdo a la intuición de un usuario no informático: al duplicar la velocidad del computador (o el tiempo disponible), se duplica aprox. el tamaño del problema resuelto.
- En los de coste $\mathcal{O}(n^k)$, al duplicar la velocidad, el tamaño se multiplica por un factor $\sqrt[k]{2}$.

Medidas asintóticas de la eficiencia

- El orden $\mathcal{O}$ se puede aplicar tanto a un análisis en el caso peor, como a un análisis en el caso promedio.
  - Hay algoritmos cuyo coste está en $\mathcal{O}(n^2)$ en el caso peor y en $\mathcal{O}(n \log n)$ en el caso promedio.
- Las unidades en que se mide el coste en tiempo (segundos, milisegundos, etc.), o en memoria (bytes, palabras, etc.) **no son relevantes** en la complejidad asintótica:
  - Dos unidades distintas se diferencian en una constante multiplicativa (e.g. 120 n^2 segundos son 2 n^2 minutos, ambos en $\mathcal{O}(n^2)$).

Orden de complejidad de un algoritmo

En los siguientes apartados veremos cómo calcular directamente el orden de complejidad de la función de coste de algoritmos iterativos y recursivos.

- Como regla general nos centraremos en el análisis para el **caso peor**. En ocasiones puede interesar el análisis para el caso promedio.
- Diremos que un algoritmo **está en** $\mathcal{O}(f(n))$ si su función de coste $T(n) \in \mathcal{O}(f(n))$. Es decir, para valores suficientemente grandes de su parámetro de entrada n el tiempo consumido por el algoritmo está acotado superiormente por $f(n)$.
- De manera menos formal usaremos el nombre del orden de complejidad: algoritmo constante, algoritmo lineal, algoritmo cuadrático, ...

Otras medidas asintóticas

Además de $\mathcal{O}(f(n))$, en el análisis de eficiencia se pueden utilizar otras medidas asintóticas:

- $\Omega(f(n))$ es el conjunto de funciones para las cuales $f(n)$ es una cota inferior para valores suficientemente grandes de n
- Si una función $f(n) \in \mathcal{O}(g(n))$ y además $f(n) \in \Omega(g(n))$, diremos que $f(n)$ está en el **orden exacto** de $g(n)$: $f(n) \in \Theta(g(n))$

Siempre que sea posible su obtención, el orden exacto del coste de un algoritmo es preferible puesto que es más informativo que únicamente una cota inferior o superior.

Consideremos la siguiente función:

```
int fact_par(int n) {
  int ret = 1;
  if (n % 2 == 0) { // 'n' es par
    for (int i = 2; i <= n; i++) {
      ret = ret * i;
    }
  }
  return ret;
}
```

- Si n es par, el coste es lineal
- Si n es impar, el coste es constante
- Por lo tanto: $T_{\text{fact\_par}}(n) \in \mathcal{O}(n),\ T_{\text{fact\_par}}(n) \notin \mathcal{O}(1)$
  $T_{\text{fact\_par}}(n) \in \Omega(1),\ T_{\text{fact\_par}}(n) \notin \Omega(n)$
  $T_{\text{fact\_par}}(n) \notin \Theta(1),\ T_{\text{fact\_par}}(n) \notin \Theta(n)$

¡Atención!

Es común equiparar el análisis en el caso mejor con el orden $\mathcal{O}(f(n))$, y el análisis en el caso mejor con $\Omega(f(n))$, pero son cosas distintas:

- El análisis en el caso mejor/peor considera que **dado un tamaño** n del argumento de entrada, el **contenido de dicho argumento es lo más beneficioso/perjudicial** para el coste.
- Una vez tomada esa suposición sobre el contenido del argumento, podemos encontrar tanto cotas superiores $\mathcal{O}(f(n))$ a su crecimiento como cotas inferiores $\Omega(f(n))$.

Resumen

- **Análisis en el caso peor** $\neq \mathcal{O}(f(n))$
- **Análisis en el caso mejor** $\neq \Omega(f(n))$

Análisis de eficiencia en algoritmos iterativos

Análisis de eficiencia en algoritmos iterativos

- En el primer ejemplo realizamos un cálculo exacto del tiempo por cada instrucción (t_a, t_c, t_i ...)
- Si estamos interesados únicamente en el **orden** $\mathcal{O}(f(n))$ del coste en el caso peor, podemos simplificar el proceso y centrarnos únicamente en lo que nos importará, ignorando diferencias tecnológicas.
- Para obtener el orden $\mathcal{O}(f(n))$ del coste de un algoritmo iterativo utilizaremos una serie de **reglas** que aplicaremos **instrucción a instrucción**.

Reglas para el cálculo de la eficiencia

1. Las **instrucciones básicas** (asignación, entrada/salida, acceso a vectores, operaciones aritméticas, etc.) tendrán un coste en $\mathcal{O}(1)$.
   (*) Si la instrucción requiere la evaluación de expresiones complejas (p.ej. una llamada a función) el coste estará en el máximo orden de dichas evaluaciones.
   Ejemplos:
   - x = 4; $\in \mathcal{O}(1)$
   - x > 4; $\in \mathcal{O}(1)$
   - x+1 > 4; $\in \mathcal{O}(1)$
   - a = fact_par(x); $\in \mathcal{O}(x)$
2. Las **llamadas a función** tienen un coste del orden de la función aplicado a los argumentos concretos. **Ejemplos**:
   - `fact_par(x);` $\in \mathcal{O}(x)$
   - `fact_par(6);` $\in \mathcal{O}(1)$
   - `search(v, 6);` $\in \mathcal{O}(n)$, siendo n la longitud del vector v

Reglas para el cálculo de la eficiencia

3. Una **secuencia de instrucciones** $S_1; S_2$ tiene coste en $\mathcal{O}(\max(f_1(n), f_2(n)))$, donde el coste de S_1 está en $\mathcal{O}(f_1(n))$ y el coste de S_2 está en $\mathcal{O}(f_2(n))$
   Ejemplos:
   - x = 4; x > 4; $\in \mathcal{O}(1)$
   - x > 4; a = fact_par(x); $\in \mathcal{O}(x)$
   - fact_par(n); fact_par(m); $\in \mathcal{O}(max(n, m)) = \mathcal{O}(n + m)$

Apunte

$\mathcal{O}(max(f_1(n), f_2(n))) = \mathcal{O}(f_1(n) + f_2(n))$, así que podéis usar *max* o $+$ para indicar los costes que no se puedan simplificar

4. Una **instrucción condicional** **if** (B) $\{S_1\}$ **else** $\{S_2\}$ tiene un coste en $\mathcal{O}(\max(f_B(n), f_1(n), f_2(n)))$, donde el coste de B está en $\mathcal{O}(f_B(n))$, el de S_1 en $\mathcal{O}(f_1(n))$ y el de S_2 en $\mathcal{O}(f_2(n))$.
   Ejemplos:
   - **if** (x > 4) {a = 5;} **else** {a = 0;} $\in \mathcal{O}(1)$
   - **if** (x > 4) {a = fact_par(x);} **else** {a = 0;} $\in \mathcal{O}(x)$
   - **if** (fact_par(x) > 56) {b = **true**;} $\in \mathcal{O}(x)$

5. Consideremos un **bucle while** $(B)\ \{S\}$ donde el coste de $B; S$ está en $\mathcal{O}(f_{B;S}(n))$ y el número de iteraciones del bucle es $iter(n)$. El coste del bucle será el **sumatorio** del coste de cada iteración para cada valor desde 1 hasta $iter(n)$.

   Ejemplo:

```
int i = 0;
while (i < n) {
  a = a + fact_par(i);
  i++;
}
```

   - Una iteración (i < n; a = a + fact_par(i); i++;) $\in \mathcal{O}(i)$
   - Número de iteraciones: n, desde $i = 0$ hasta $i = n - 1$
   - Coste de todas las iteraciones:

$$\sum_{i=0}^{n-1} i = \frac{(n-1)n}{2} \in \mathcal{O}(n^2)$$

Reglas para el cálculo de la eficiencia

- El caso de bucles **for** es igual considerando que

  for (<init>; <cond>; <incr>) { S } ≡ <init>; **while** (<cond>) {S; <incr>}

- Si en un bucle el coste de cada iteración es constante, resolver el sumatorio se simplifica. **Ejemplo**:

```
int i = 0;
while (i < n) {
  a = a + fact_par(n);
  i++;
}
```

  - Una iteración: (i < n; a = a + fact_par(n); i++;) $\in \mathcal{O}(n)$
  - Número de iteraciones: n, desde $i = 0$ hasta $i = n - 1$
  - Coste de todas las iteraciones:

$$\sum_{i=0}^{n-1} n = n \cdot \sum_{i=0}^{n-1} 1 = n \cdot n \in \mathcal{O}(n^2)$$

Recordatorio de sumatorios[2]

- **«Constantes fuera»:**

$$\sum_{i=a}^{n} k \cdot s_i = k \cdot \sum_{i=a}^{n} s_i$$

- **Serie aritmética:**

$$\sum_{i=a}^{n} i = \frac{\overbrace{(a+n)}^{suma\_extremos} \qquad \overbrace{(n-a+1)}^{num\_elementos}}{2}$$

- **Serie geométrica:**

$$\sum_{i=0}^{n} k^i = \frac{1-k^{n+1}}{1-k}$$

[2] Si durante el cálculo de eficiencia de un algoritmo os encontráis con otros sumatorios que no sabéis manejar, buscad más propiedades sobres los sumatorios. **¡Hay muchas!**

Ejemplo completo de eficiencia asintótica

```
1 template <class T>
2 bool search (vector<T> const& v, T const& x) {
3   int i = 0;
4   while (i < v.size() && v[i] != x) {
5     ++i;
6   }
7   return i < v.size();
8 }
```

Sabiendo que el coste de `v.size()` $\in \mathcal{O}(1)$ tendríamos que:

- Instrucción en L3 $\in \mathcal{O}(1)$
- Bucle en L4–L6:
  - Comparación en L4 $\in \mathcal{O}(1)$
  - Instrucción en L5 $\in \mathcal{O}(1)$
  - Secuencia L4;L5 $\in \mathcal{O}(max(1,1)) = \mathcal{O}(1)$
  - **Total bucle**: $\sum_{i=0}^{n-1} 1 \in \mathcal{O}(n)$ *[n es la longitud de v]*
- Instrucción en L7 $\in \mathcal{O}(1)$

Coste de **toda la función** L3–L7 $\in \mathcal{O}(max(1,n,1)) = \mathcal{O}(n)$

Análisis de eficiencia en algoritmos recursivos

Análisis de eficiencia en algoritmos recursivos

- Al calcular el coste de algoritmos recursivos nos encontraremos con **funciones de coste recursivas** y con distinción de casos, p.ej.

$$T(n) = \begin{cases} 5 & \text{si } n = 0 \\ T(n-1) + 8 & \text{si } n > 0 \end{cases}$$

- Estas ecuaciones se denominan *relaciones recurrentes* o **recurrencias**.
- Lo que queremos es encontrar el orden de complejidad de la recurrencia $T(n)$. Posibilidades:
  - Obtener una expresión no recursiva de $T(n)$ mediante la expansión de la recurrencia.
  - Encajar $T(n)$ con una **plantilla** y obtener directamente el orden de complejidad.

Obtención de recurrencias

El primer paso será analizar el código de la función y crear la recurrencia. Nos centramos en el número de **instrucciones elementales ejecutadas** e ignoramos el coste de **return**.

```
int fact(const int n) {
  int s;
  if (n == 0) {
    s = 1;
  } else {
    s = n * fact(n-1);
  }
  return s;
}
```

Este es el código clásico para el cálculo del factorial de manera recursiva. Vemos que tenemos dos casos: cuando $n = 0$ y cuando $n > 0$.

Obtención de recurrencias

```
int fact(const int n) {
  int s;
  if (n == 0) {
    s = 1;
  } else {
    s = n * fact(n-1);
  }
  return s;
}
```

- $n = 0$) Ejecutamos 2 instrucciones: 1 comparación y 1 asignación.
- $n > 0$) Ejecutamos 4 instrucciones (1 comparación, 1 multiplicación, 1 resta y 1 asignación) **más el coste de invocar a** `fact(n-1)`

$$T(n) = \begin{cases} 2 & \text{si } n = 0 \\ T(n-1) + 4 & \text{si } n > 0 \end{cases}$$

Resolución de recurrencias

Para obtener una expresión no recursiva a partir de una recurrencia usaremos la técnica de **expansión**.

1. Vamos expandiendo la recurrencia para detectar patrones en la expresión.
2. Generamos la expresión expandida k veces.
3. Detectamos cuánto debe de valer k para llegar al caso base.
4. Usamos ese valor de k para *evaluar* la expresión.
5. Finalmente, demostraremos por inducción que la expresión obtenida es correcta para la recurrencia. **Este paso es imprescindible, porque la generalización realizada en los pasos 1-2 la hemos hecho «a ojo»**

Expansión

Consideremos la recurrencia de la función fact

$$T(n) = \begin{cases} 2 & \text{si } n = 0 \\ T(n-1) + 4 & \text{si } n > 0 \end{cases}$$

Para expandirla iríamos aplicando el caso recursivo repetidas veces:

$$\begin{array}{rcll} T(n) & = & T(n-1) + 4 & k = 1 \\ & = & T(n-2) + 4 + 4 & k = 2 \\ & = & T(n-3) + 4 + 4 + 4 & k = 3 \\ & = & T(n-4) + 4 + 4 + 4 + 4 & k = 4 \\ & & \vdots & \\ & = & \mathbf{T(n-k) + 4k} & \end{array}$$

Evaluación

Ahora tenemos la expresión *general* de la recurrencia expandida k veces, que hemos inferido detectando patrones:

$$T(n-k)+4k$$

Sabemos que el caso base ocurre cuando el argumento es 0, ¿cuántas veces deberíamos expandir la recurrencia (k) para llegar a dicho caso base?

$$n-k=0 \iff k=n$$

Si sustituimos ese valor de k en la expresión general tendremos que

$$T(n-n)+4\cdot n = T(0)+4n = 2+4n \in \mathcal{O}(n)$$

Demostración

Ahora deberíamos demostrar por inducción que $\forall n \geq 0. T(n) = T_{exp}$ donde el coste obtenido mediante expansiones es $T_{exp}(n) = 4n + 2$ y

$$T(n) = \begin{cases} 2 & \text{si } n = 0 \\ T(n-1) + 4 & \text{si } n > 0 \end{cases}$$

- **Caso base,** $n = 0$)

$$T(0) \stackrel{def}{=} 2 = 4 \cdot 0 + 2 \stackrel{def}{=} T_{exp}(0)$$

- **Paso Inductivo,** $n > 0$)

$$\begin{array}{rcl} T(n) & \stackrel{def}{=} & T(n-1) + 4 \\ & \stackrel{HI}{=} & 4(n-1) + 2 + 4 \\ & = & 4n + 2 \stackrel{def}{=} T_{exp}(n) \end{array}$$

Expansión

Lamentablemente las recurrencias no suelen expandirse y evaluarse tan fácilmente... Por ejemplo:

$$T(n) = \begin{cases} 8 & \text{si } n = 0 \\ 3T(n-1)+5 & \text{si } n > 0 \end{cases}$$

Se expandiría de la siguiente manera:

$$\begin{array}{rcll}
T(n) & = & 3T(n-1)+5 & k=1 \\
 & = & 3[3T(n-2)+5]+5 & k=2 \\
 & = & 3\cdot 3T(n-2)+3\cdot 5+5 & \\
 & = & 3\cdot 3[3T(n-3)+5]+3\cdot 5+5 & k=3 \\
 & = & 3\cdot 3\cdot 3T(n-3)+3\cdot 3\cdot 5+3\cdot 5+5 & \\
 & & \vdots & \\
 & = & \mathbf{3^k T(n-k)+\sum_{i=0}^{k-1} 5\cdot 3^i} &
\end{array}$$

Evaluación

Sabemos que el caso base ocurre cuando el argumento es 0, luego

$$n - k = 0 \iff k = n$$

Si sustituimos en la expresión general tendremos que:

$$\begin{array}{rl} & 3^n T(n-n) + \sum_{i=0}^{n-1} 5 \cdot 3^i \\ = & 3^n T(0) + 5 \sum_{i=0}^{n-1} 3^i \\ = & 3^n \cdot 8 + 5 \frac{1-3^n}{-2} \\ = & 3^n \cdot 8 + \frac{5}{2} 3^n - \frac{5}{2} \\ = & (8 + \frac{5}{2}) 3^n - \frac{5}{2} \\ = & \frac{21}{2} \mathbf{3^n} - \frac{5}{2} \in \mathcal{O}(3^n) \end{array}$$

Demostración

Sabiendo $T_{exp}(n) = \frac{21}{2}3^n - \frac{5}{2}$ demostramos por inducción que $\forall n \geq 0.\ T(n) = T_{exp}(n)$, donde $T(n)$ estaba definida como

$$T(n) = \begin{cases} 8 & \text{si } n = 0 \\ 3T(n-1) + 5 & \text{si } n > 0 \end{cases}$$

- **Caso base,** $n = 0)$

$$T(0) \stackrel{def}{=} 8 = \frac{16}{2} = \frac{21}{2} - \frac{5}{2} = \frac{21}{2}3^0 - \frac{5}{2} \stackrel{def}{=} T_{exp}(0)$$

- **Paso Inductivo,** $n > 0)$

$$\begin{array}{lll} T(n) & \stackrel{def}{=} & 3T(n-1) + 5 \\ & \stackrel{HI}{=} & 3[\frac{21}{2}3^{n-1} - \frac{5}{2}] + 5 \\ & = & \frac{21}{2}3^n - \frac{15}{2} + 5 \\ & = & \frac{21}{2}3^n - \frac{15}{2} + \frac{10}{2} = \frac{21}{2}3^n - \frac{5}{2} \stackrel{def}{=} T_{exp}(n) \end{array}$$

Expansión

Otro ejemplo más:

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \vee n = 1 \\ 2T(\frac{n}{2}) + n^2 & \text{si } n > 1 \end{cases}$$

Se expandiría de la siguiente manera:

$$\begin{array}{rcll}
T(n) & = & 2T(\frac{n}{2}) + n^2 & k = 1 \\
 & = & 2[2T(\frac{n}{4}) + (\frac{n}{2})^2] + n^2 & k = 2 \\
 & = & 4T(\frac{n}{4}) + 2\frac{n^2}{2^2} + n^2 & \\
 & = & 4T(\frac{n}{4}) + \frac{1}{2}n^2 + n^2 & \\
 & = & 4[2T(\frac{n}{8}) + (\frac{n}{4})^2] + \frac{1}{2}n^2 + n^2 & k = 3 \\
 & = & 8T(\frac{n}{8}) + 4(\frac{n^2}{4^2}) + \frac{1}{2}n^2 + n^2 & \\
 & = & 8T(\frac{n}{8}) + \frac{1}{4}n^2 + \frac{1}{2}n^2 + n^2 & \\
 & & \vdots & \\
 & = & \mathbf{2^k T(\frac{n}{2^k}) + \sum_{i=0}^{k-1} \frac{1}{2^i} n^2} &
\end{array}$$

Evaluación

Para evitar *sutilezas* con la división entera, supondremos un n que es potencia de 2. El caso base ocurrirá cuando $k = 1$, por lo tanto:

$$\frac{n}{2^k} = 1 \iff n = 2^k \iff k = log_2\ n$$

Si sustituimos la k en la expresión general tendremos que:

$$\begin{array}{rl}
 & 2^{(log_2\ n)} T(\frac{n}{2^{log_2\ n}}) + \sum_{i=0}^{(log_2\ n)-1} \frac{1}{2^i} n^2 \\
= & nT(1) + n^2 \sum_{i=0}^{(log_2\ n)-1} (\frac{1}{2})^i \\
= & n + n^2 \sum_{i=0}^{(log_2\ n)-1} (\frac{1}{2})^i \\
= & n + n^2 (\frac{1 - \frac{1}{2}^{(log_2\ n)}}{1 - \frac{1}{2}}) \\
= & n + n^2 (\frac{1 - \frac{1}{n}}{1 - \frac{1}{2}}) = \ldots = n + n^2 (\frac{2n-2}{n}) \\
= & n - 2n + 2n^2 = 2\mathbf{n^2} - n \in \mathcal{O}(n^2)
\end{array}$$

Demostración

La expresión no funciona para $n = 0$, pero no es problema ya que queremos que la igualdad se cumpla para *valores de n suficientemente grandes*, concretamente $n \geq 1$. Por lo tanto demostramos por inducción que $\forall n \geq 1$, n potencia de 2. $T(n) = T_{exp}(n)$, donde $T_{exp} = 2n^2 - n$ y $T(n)$ se definía como

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \vee n = 1 \\ 2T(\frac{n}{2}) + n^2 & \text{si } n > 1 \end{cases}$$

- **Caso base,** $n = 1$)

$$T(1) \stackrel{def}{=} 1 = 2 \cdot 1^2 - 1 \stackrel{def}{=} T_{exp}(1)$$

- **Paso Inductivo,** $n > 1$)

$$\begin{array}{lll} T(n) & \stackrel{def}{=} & 2T(\frac{n}{2}) + n^2 \stackrel{HI}{=} 2[2(\frac{n}{2})^2 - \frac{n}{2}] + n^2 \\ & = & 4\frac{n^2}{2^2} - n + n^2 = n^2 - n + n^2 \\ & = & 2n^2 - n \stackrel{def}{=} T_{exp}(n) \end{array}$$

Plantillas para obtener el coste

- Una vez hemos calculado la recurrencia, podemos utilizar *plantillas* de coste. Si la recurrencia encaja con alguna de las dos plantillas, únicamente tendremos que detectar en qué caso concreto estamos y directamente obtendremos el coste.
- El método de las plantillas también se llama *master method* en [Corment *et al.*, 2009]
- Las plantillas se obtienen aplicando el método del expansión-evaluación de manera *simbólica* y luego realizando distinción de casos sobre la expresión resultante. En [Peña, 03] podéis ver el desarrollo completo.

Plantillas con sustracción

Reducción del problema mediante sustracción

Consideremos una recurrencia con el siguiente aspecto:

$$T(n) = \begin{cases} c_1 & \text{si } 0 \leq n < b \\ \mathbf{a}T(n - \mathbf{b}) + c_2 n^{\mathbf{k}} & \text{si } n \geq b \end{cases}$$

El orden de esta recurrencia estará en:

$$\begin{array}{lcl} \text{si } a = 1 & \longrightarrow & T(n) \in \Theta(n^{k+1}) \\ \text{si } a > 1 & \longrightarrow & T(n) \in \Theta(a^{n \ div \ b}) \end{array}$$

Plantillas con división

Reducción del problema mediante división

Consideremos una recurrencia con el siguiente aspecto:

$$T(n) = \begin{cases} c_1 & \text{si } 0 \leq n < b \\ \mathbf{a}T(n/\mathbf{b}) + c_2 n^{\mathbf{k}} & \text{si } n \geq b \end{cases}$$

El orden de esta recurrencia estará en:

$$\begin{array}{lcl} \text{si } a < b^k & \longrightarrow & T(n) \in \Theta(n^k) \\ \text{si } a = b^k & \longrightarrow & T(n) \in \Theta(n^k log\ n) \\ \text{si } a > b^k & \longrightarrow & T(n) \in \Theta(n^{log_b\ a}) \end{array}$$

Conclusiones

Comparación de algoritmos

- El análisis asintótico de coste y los órdenes de complejidad nos proporciona una manera sencilla de comparar la eficiencia de dos algoritmos. Sin embargo hay que tener en cuenta 2 aspectos:
  - Se centran en tamaños *suficientemente grandes*.
  - Ignoran las constantes multiplicativas y aditivas.
- Así que a la hora de comparar algoritmos utilizando sus costes asintóticos hay que tener cuidado y entender bien la información que proporcionan.

Comparación de algoritmos

- Tengo dos algoritmos A y B con $T_A(n) \in \mathcal{O}(n)$ y $T_B(n) \in \mathcal{O}(n^2)$. **¿Cuál es mejor?**
- Para valores suficientemente grandes de n el mejor es A sin ningún tipo de duda.
- ¿Y para valores pequeños? Es muy posible que B sea más rápido para datos pequeños. (*Piensa que quizá A tiene una «constante oculta» más elevada que B*).
  - Si estos algoritmos únicamente se aplicarán a datos pequeños, puede ser *mejor* usar B. **¿Y qué significa pequeño?**
  - Puedes detectar a partir de qué valor n_0 empieza a ser mejor A, y llamar a un algoritmo u otro.
  - La determinación de este valor n_0 se puede realizar de manera empírica o realizando un análisis de coste preciso y luego comparando ambas expresiones de coste.

Bibliografía

Bibliografía

- Ricardo Peña Marí. *Diseño de Programas: Formalismo y Abstracción (Tercera edición)*. Pearson/Prentice Hall, 2005. **Capítulo 1**.
  `http://cisne.sim.ucm.es/record=b2175612~S6*spi`
- Mario Rodriguez Artalejo, Pedro Antonio González Calero, Marco Antonio Gómez Martín. *Estructuras de datos: un enfoque moderno*. Editorial Complutense, 2011. **Capítulos 1.4 y 2.4**.
  `http://cisne.sim.ucm.es/record=b3643210~S6*spi`
- Thomas H. Cormen, Charles E. Leiserson, Ronarld L. Rivest, Clifford Stein. *Introduction to Algorithms (Third Edition)*. MIT Press, 2009. **Capítulos 3 y 4**.
  `https://ucm.on.worldcat.org/oclc/1025499183`