Eficiencia de algoritmos

Javier Campos

Eficiencia de algoritmos

- *Problema de cálculo*:

  especificación de una relación existente entre unos valores de *entrada* (datos del problema) y otros de *salida* (resultados)

  Ejemplo: *problema de ordenación*

  entrada: una secuencia de números $(a_1, a_2, \ldots, a_n)$

  salida: una permutación $(a_1', a_2', \ldots, a_n')$ de la secuencia de entrada tal que $a_1' \leq a_2' \leq \ldots \leq a_n'$

- *Algoritmo*:

  herramienta para resolver un problema de cálculo, es decir, método de cálculo bien definido que a partir de un valor o valores de entrada produce un valor o conjunto de valores de salida

Eficiencia de algoritmos

- *Ejemplar* de un problema:

  valores concretos necesarios para construir la entrada para un problema (satisfaciendo las restricciones impuestas por el problema)

  Ejemplo:

  $(4,2,6,5,1) \rightarrow$ algoritmo de ordenación $\rightarrow (1,2,4,5,6)$

Eficiencia de algoritmos

- Algoritmo correcto:

  para todo ejemplar de problema (todos los valores posibles de los datos de entrada) produce una salida correcta (i.e., los valores de salida verifican la relación especificada en el problema con los de entrada)

  Ejemplo: algoritmo correcto de ordenación

```
algoritmo ordenaciónPorInserción(A)
principio
  para j:=2 hasta long(A) hacer
    dato:=A[j]
    i:=j-1;
    mq i>0 and A[i]>dato hacer
      A[i+1]:=A[i];
      i:=i-1
    fmq;
    A[i+1]:=dato
  fpara
fin
```

Eficiencia de algoritmos

- *Eficiencia* de un algoritmo:
  - cantidad de recursos que usa un algoritmo para su ejecución
  - normalmente se considera el tiempo de ejecución del algoritmo, la utilización de espacio de memoria o el número de procesadores (en programación paralela)
  - en este curso nos concentraremos en el tiempo de ejecución en función del tamaño de la entrada, se denomina también *coste*, *rendimiento* o *complejidad* del algoritmo

Eficiencia de algoritmos

- Técnicas de cálculo de la eficiencia:
  - Pruebas (en inglés *benchmarking*):
    - elaborar una muestra significativa o típica de los posibles datos de entrada y tomar medidas de los tiempos de ejecución para cada elemento de la muestra
    - a partir de los datos anteriores aplicar técnicas estadísticas para inferir el rendimiento del programa para datos de entrada no presentes en la muestra (por tanto, las conclusiones pueden no ser muy fiables)
  - Análisis
    - procedimientos matemáticos para determinar el coste (conclusiones fiables, pero realización difícil o imposible a efectos prácticos)

Eficiencia de algoritmos

- Tamaño de los datos:
  - El coste de un algoritmo es función del tamaño de los datos de entrada
  - Sugerencias para definir el tamaño de los datos:
    - datos naturales o enteros:
      - su propio valor
      - el tamaño de su descripción binaria
      - ¡Ojo! ¡Hay que aclararlo, se diferencian en una exponencial!

Eficiencia de algoritmos

- Sugerencias para definir el tamaño de los datos (continuación):
  - pares o ternas de enteros:
    - en función de todos ellos
      - » es lo más deseable pero muchas veces el análisis es difícil de realizar
    - en función de uno de ellos (suponiendo los demás constantes)
      - » es una decisión irreal (utilidad discutible) pero en la práctica mucho más sencilla de realizar
      - » a efectos de comparar distintos algoritmos para resolver un mismo problema puede ser suficiente

Eficiencia de algoritmos

- Sugerencias para definir el tamaño de los datos (continuación):
  - estructuras homogéneas (vectores, ficheros,…):
    - número de componentes (longitud del vector)
      - » normalmente es lo más adecuado
      - » puede ser inadecuado ocasionalmente si los elementos son a su vez estructuras complejas, o enteros de magnitudes muy elevadas para los que no basta una palabra de máquina
  - otros casos:
    - como se pueda, *ad hoc*, con sentido común,...

Eficiencia de algoritmos

- Tiempo de ejecución
  - función *T(n)* que representa el número de unidades de tiempo (segundos, milisegundos, …) que un algoritmo tarda en ejecutarse al suministrarle unos datos de entrada de tamaño *n*
  - problema: variaciones sustanciales según la marca y modelo de computador utilizado
  - solución: es preferible que *T(n)* sea el *número de instrucciones simples* (asignaciones, comparaciones, operaciones aritméticas, etc.) que se ejecutan

    sería como el tiempo de ejecución en un computador idealizado, donde cada asignación, comparación, lectura, etc., consume 1 unidad de tiempo

    si *T(n)* es el tiempo de ejecución en ese computador idealizado, el tiempo de ejecución en un computador real *X* será $c_x T(n)$, donde c_x es una constante que depende del computador (y que puede determinarse usando técnicas de *benchmarking* y estadísticas convencionales)

Eficiencia de algoritmos

- Caso peor, caso mejor, caso promedio:

  Con frecuencia, el coste de un algoritmo depende de los datos de entrada particulares y no sólo del tamaño de los mismos

  – *Coste en el caso peor*:
    - coste máx. del algoritmo para datos de entrada de tamaño *n*
    - es el más comúnmente usado, lo denotaremos *T(n)*
    - es excesivamente pesimista
  – *Coste en el caso mejor*:
    - coste mín. del algoritmo para datos de entrada de tamaño *n*
    - demasiado optimista, demasiado irreal para ser de utilidad práctica

Eficiencia de algoritmos

- Caso peor, caso mejor, caso promedio (cont.)

  – *Coste promedio*:
    - cálculo de una media del uso de recursos para todos los datos de entrada posibles de tamaño *n*
    - lo denotaremos $\overline{T(n)}$
    - la media se hace ponderando por la frecuencia de aparición de cada caso, i.e., requiere el cálculo de una *esperanza matemática* (la media) a partir de una distribución de probabilidad
    - da más información
    - precisa más datos (la distribución de probabilidad de los datos de entrada de tamaño *n*), que son muy difíciles de calcular
    - normalmente se basa en la hipótesis de equiprobabilidad de las posibles entradas (hipótesis que a menudo no es cierta)

Eficiencia de algoritmos

- El ejemplo:

```
algoritmo ordenaciónPorInserción(A)
principio
  para j:=2 hasta long(A) hacer
    dato:=A[j]
    i:=j-1;
    mq i>0 and A[i]>dato hacer

      A[i+1]:=A[i];

      i:=i-1
    fmq;
    A[i+1]:=dato
  fpara
fin
```

coste	veces
c_1	n
c_2	n-1
c_3	n-1
c_4	$\sum_{j=2}^{n} t_j$
c_5	$\sum_{j=2}^{n} (t_j - 1)$
c_6	$\sum_{j=2}^{n} (t_j - 1)$
c_7	n-1

t_j es el número de veces que se ejecuta el `mq` para cada valor de j

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Eficiencia de algoritmos

- Un ejemplo (cont.):

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Si el vector ya estaba ordenado (**caso mejor**):

`A[i]` $\leq$ `dato` para `i=j-1` $\Rightarrow t_j = 1$ para $j = 2, 3, \ldots, n$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \\ &= an + b \end{aligned}$$

función lineal en *n*

Eficiencia de algoritmos

- Un ejemplo (cont.):

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Si el vector estaba ordenado en sentido decreciente (**caso peor**):

$t_j = j$ para $j = 2, 3, \ldots, n$

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \qquad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

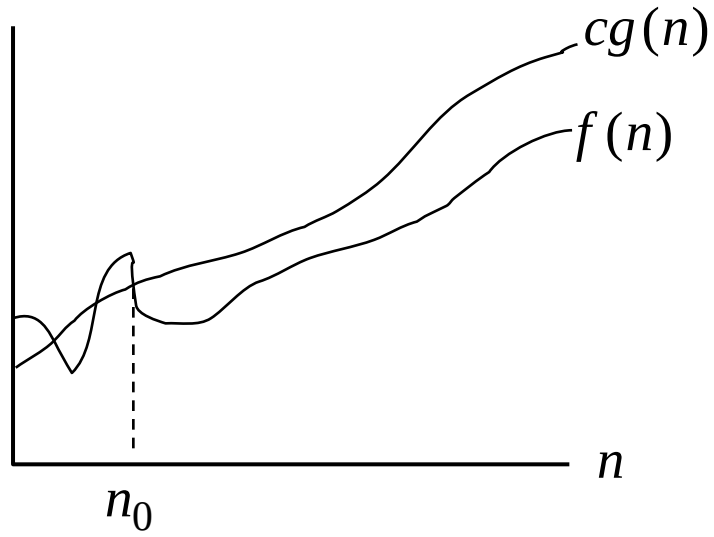
$$= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right)n - (c_2 + c_3 + c_4 + c_7)$$

$$= an^2 + bn + c$$

→ **función cuadrática en *n***

Eficiencia de algoritmos

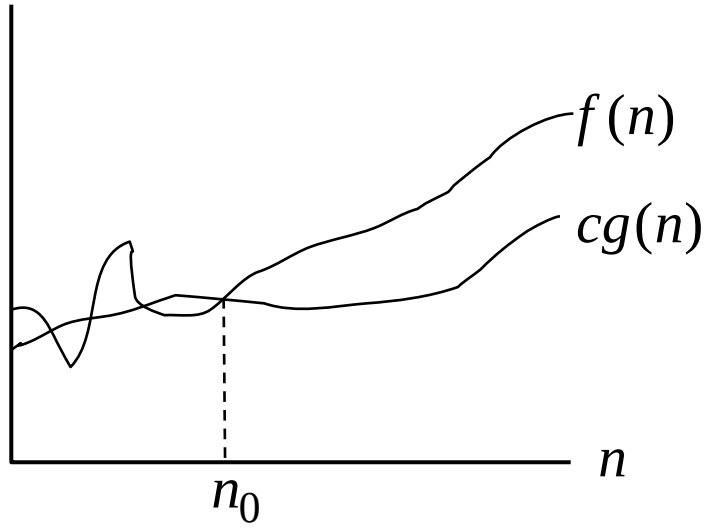
- Órdenes de crecimiento



g(*n*) es cota superior asintótica de *f*(*n*) (i.e., *f*(*n*) es del mismo orden o de orden inferior a *g*(*n*))

$f(n)$ es $O(g(n))$

($c > 0$)



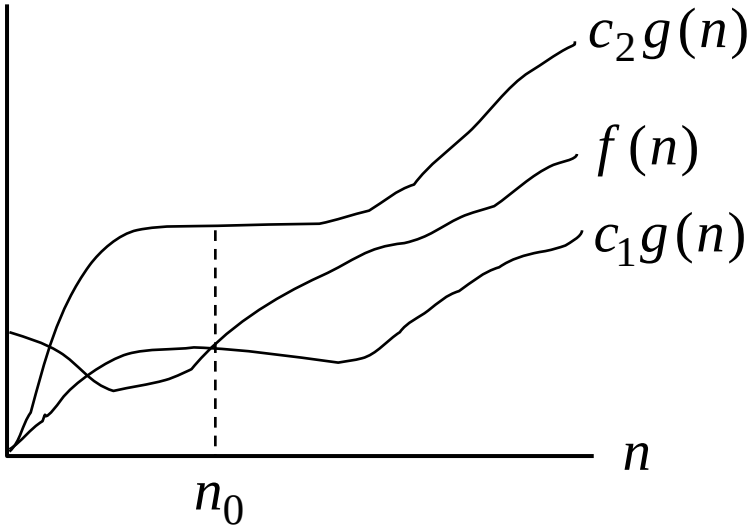
g(*n*) es cota inferior asintótica de *f*(*n*) (i.e., *f*(*n*) es del mismo orden o de orden superior a *g*(*n*))

$f(n)$ es $\Omega(g(n))$

($c > 0$)

Eficiencia de algoritmos

- Órdenes de crecimiento



f(*n*) es del orden de magnitud de *g*(*n*)

$f(n)$ es $\Theta(g(n))$

(c_1, $c_2 > 0$)

Eficiencia de algoritmos

- Propiedades del orden de crecimiento O:
  - Propiedad 1.

    $\forall\ 0 \leq a \leq b$: n^a es $O(n^b)$.

    Además n^a es de orden inferior a n^b si $a < b$ y son del mismo orden si $a = b$.
  - Propiedad 2.

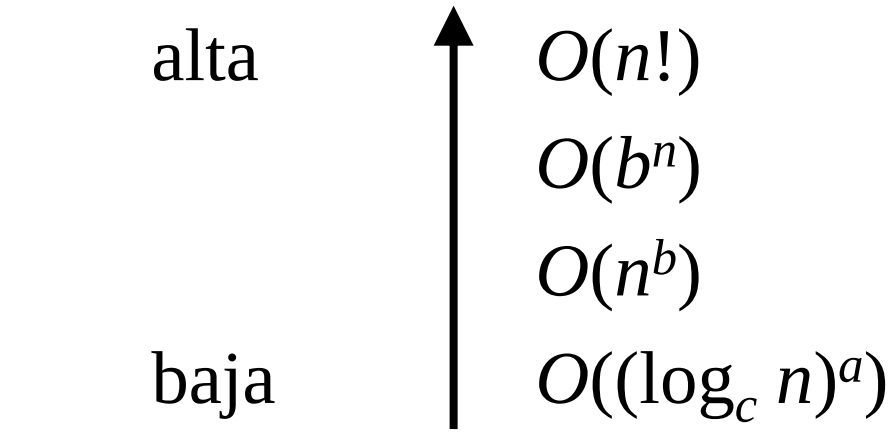
    $\forall\ a \geq 0, b > 0, c > 1$: $(\log_c n)^a$ es $O(n^b)$ y es de orden inferior.
  - Propiedad 3.

    $\forall\ a \geq 0, b > 1$: n^a es $O(b^n)$ y es de orden inferior.
  - Propiedad 4.

    $\forall\ b > 1$: b^n es $O(n!)$ y es de orden inferior.

Eficiencia de algoritmos

- Recapitulando… principios generales de *O*:



Eficiencia de algoritmos

- Recapitulando… principios generales de *O*:
  - Los factores constantes no importan: si $T(n)$ es $O(f(n))$ entonces $d_1 T(n)$ es $O(d_2 f(n))$, para todas $d_1, d_2 > 0$.
  - Los términos de orden inferior no importan. Por ejemplo, si $T(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$, con $a_k > 0$, entonces $T(n)$ es $O(n^k)$. De hecho, si $T(n)$ es una suma de funciones $f_i(n)$ positivas entonces $T(n)$ es $O(f_k(n))$, donde $f_k(n)$ es la función de mayor orden entre las distintas $f_i(n)$.
  - La base de logaritmos no importa, porque $\log_b n = \log_c n \cdot \log_b c$, para cualesquiera $b, c > 1$.
  - Las expresiones usando la notación *O* son transitivas. Si $f(n)$ es $O(g(n))$ y $g(n)$ es $O(h(n))$ entonces $f(n)$ es $O(h(n))$.

Eficiencia de algoritmos

- Coste de algoritmos y notación *O*:
  - Si el coste de un algoritmo o de una de sus partes es independiente del tamaño de la entrada, se dice que el coste es *O*(1).
  - Cualquier asignación, lectura de variable, escritura, operación aritmética o comparación tiene coste *O*(1).
  - Si en alguna expresión interviene una función, se añade el coste de la función.

Eficiencia de algoritmos

- Coste de algoritmos y notación *O* (cont.):
  - Composición secuencial: regla de las sumas

    Si un algoritmo consta de *k* partes en secuencia, cada una con coste $T_i(n)$ de $O(f_i(n))$, entonces

    $T(n) = T_1(n) + \cdots + T_k(n) = O(\max\{f_1(n), \ldots, f_k(n)\})$
  - Composición condicional: también se aplica la regla de las sumas (caso peor…)
  - Composición iterativa: regla de los productos

    Si el coste del cuerpo de una iteración es $O(f(n))$ y el número de iteraciones es $O(g(n))$ entonces el coste total de la iteración es $O(f(n) \cdot g(n))$

Eficiencia de algoritmos

- Coste de algoritmos recursivos:

  El coste $T(n)$ de una función para datos de tamaño n se define en función del coste $T(m)$ de las llamadas recursivas para otros tamaños m (menores que n)

Teorema (*Master Theorem*):

$$T_1(n) = \begin{cases} f(n) & \text{si } 0 \le n < c \\ a \cdot T_1(n-c) + b \cdot n^k & \text{si } c \le n \end{cases} \Rightarrow T_1(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n/c}) & \text{si } a > 1 \end{cases}$$

$$T_2(n) = \begin{cases} g(n) & \text{si } 0 \le n < c \\ a \cdot T_2(n/c) + b \cdot n^k & \text{si } c \le n \end{cases} \Rightarrow T_2(n) \in \begin{cases} \Theta(n^k) & \text{si } a < c^k \\ \Theta(n^k \cdot \log n) & \text{si } a = c^k \\ \Theta(n^{\log_c a}) & \text{si } a > c^k \end{cases}$$