Tema 2
Fundamentos de Complejidad Algorítmica

Pablo Sánchez

Dpto. Matemáticas, Estadística y Computación
Universidad de Cantabria
Santander (Cantabria, España)
p.sanchez@unican.es

Bibliografía Básica

Brasard, G. and Bratley, P. (2000).
Fundamentos de Algoritmia.
Prentice Hall.

Ricardo Peña (2005).
Diseño de Programas: Formalismo y Abstracción.
Pearson Educacion, 3 edition.

Objetivos

Objetivos

1. Conocer los conceptos y técnicas básicas de cálculo de complejidad algorítmica.
2. Saber estimar la complejidad de algoritmos iterativos básicos.
3. Conocer las técnicas de estimación de la complejidad de algoritmos recursivos.
4. Conocer los conceptos básicos de la complejidad computacional y la computabilidad.

Objetivo de la Complejidad Algorítmica

Complejidad Algorítmica

Estudiar de forma genérica (e independiente a la máquina) los recursos (tiempo y cantidad de memoria) requeridos por un algoritmo para resolver un problema.

Problemas de las pruebas empíricas:

1. Hay que implementar el algoritmo.
2. Hay que hacer muchas pruebas (y para casos largos).
3. Probar algoritmos *pesados* puede ser muy costoso.

Complejidad de un Algoritmo Simple

```
// Búsqueda de la posición de un entero dentro de un
// vector
FUNCION posicion(n: ENTERO, v : vENTERO) : ENTERO ES

    i : ENTERO;
    i := 0;

    MIENTRAS ((i<MAX_vENTERO) AND (v[i] != n)) HACER
      i := i + 1;
    FINMIENTRAS

    DEVOLVER pos;
FINFUNCION //posicion
```

$$t(n) = (2c + a)n + (a + d)$$

Preguntas Importantes a Responder

1. ¿Cómo se comporta el algoritmo para problemas grandes?
2. ¿Existe algún límite al tamaño de los datos de entrada?
3. ¿Cómo de eficiente es un algoritmo en comparación a otro algoritmo?
4. ¿Qué tipo de función describe el tiempo consumido por un algoritmo?

Medidas de Tiempo de un Algoritmo

- Para un tamaño de la entrada fijo, diversas variables sobre la forma de la entrada pueden afectar al tiempo de ejecución del algoritmo.
- Se analizan por tanto 3 casos:

  Caso peor: máximo tiempo de respuesta para un tamaño de entrada fijo.

  Caso mejor: mínimo tiempo de respuesta para un tamaño de entrada fijo.

  Caso promedio: tiempo medio de respuesta para un tamaño de entrada fijo.

Principios de Complejidad Algorítmica

Principio de Invarianza

Dado un algoritmo S, y dos implementaciones I_1 e I_2 de dicho algoritmo, cuyos tiempos de ejecución son $t_1(n)$ y $t_2(n)$, entonces existen constantes naturales k y n_0, tales que $\forall n \geq n_0, t_1(n) \leq k \cdot t_2$

Operación elemental

Una *operación elemental* es aquella cuyo tiempo de ejecución está acotado superiormente por un valor constante que depende sólo de la máquina donde se ejecuta y es por tanto independiente de los parámetros del problema que resuelve un algoritmo.

Notaciones asintóticas

Orden de una función $O(f(n))$

Sea $f : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$. El conjunto de funciones *del orden de $f(n)$*, denotado como $\mathcal{O}(f(n))$, se define como:

$$\mathcal{O}(f(n)) = \{g : \mathbb{N} \to \mathbb{R}^+ \cup 0 | \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}; \forall n \geq n_o; g(n) \leq c \cdot f(n)\}$$

Una función $g(n)$ es del orden de $f(n)$ cuando $g(n) \in \mathcal{O}(f(n))$.

Notaciones asintóticas

Cota inferior asintótica de una función $f(n)$

Sea $f : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$. El conjunto de funciones que son cotas inferiores asintóticas de $f(n)$, denotado $\Omega(f(n))$, se define como:

$$\Omega(f(n)) = \{g : \mathbb{N} \to \mathbb{R}^+ \cup 0 | \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}; \forall n \geq n_o; g(n) \geq c \cdot f(n)\}$$

Una función $g(n)$ es cota inferior asintótica de $f(n)$ cuando $g(n) \in \Omega(f(n))$.

Orden exacto de una función $f(n)$

Sea $f : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$. El conjunto de funciones del orden exacto de $f(n)$, denotado como $\Theta(f(n))$, se define como:

$$\Theta(f(n)) = \mathcal{O}(n) \cap \Omega(n)$$

Una función $g(n)$ es del orden exacto de $f(n)$ cuando $g(n) \in \Theta(f(n))$.

Operaciones entre Órdenes

Menor o Igual

Dados los órdenes de dos funciones $\mathcal{O}(f(n))$ y $\mathcal{O}(g(n))$, $\mathcal{O}(f(n)) \leq \mathcal{O}(g(n))$ si y sólo si $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$

Orden de una suma de funciones

$\mathcal{O}(f(n) + g(n)) = \mathcal{O}(max(f(n), g(n)))$

Regla del Límite

Regla del Límite

Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} =$

$$(1) = k \in \mathbb{R}^+ \Rightarrow f(n) \in \mathcal{O}(g(n)) \wedge g(n) \in \mathcal{O}(f(n))$$

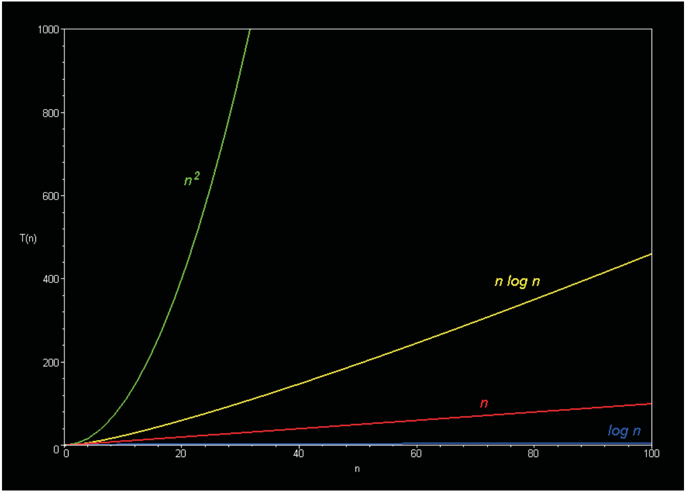
$$(2) = 0 \in \mathbb{R}^+ \Rightarrow f(n) \in \mathcal{O}(g(n)) \wedge g(n) \notin \mathcal{O}(f(n))$$

$$(3) = \infty \in \mathbb{R}^+ \Rightarrow f(n) \notin \mathcal{O}(g(n)) \wedge g(n) \in \mathcal{O}(f(n))$$

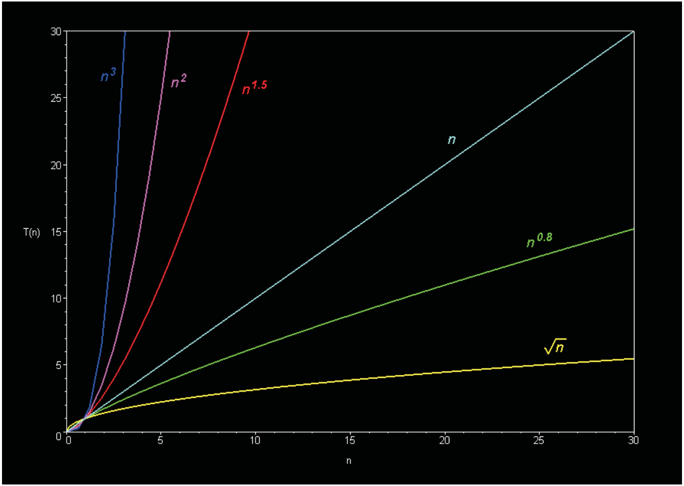
Por tanto,

$$\mathcal{O}(1) \leq \mathcal{O}(\log n) \leq \mathcal{O}(\sqrt(n)) \leq \mathcal{O}(n) \leq \mathcal{O}(n \log n)) \leq \mathcal{O}(n^2) \leq$$
$$\leq \mathcal{O}(n^2 \log n)) \leq \mathcal{O}(n^3) \leq \ldots \leq \mathcal{O}(n^k) \leq \mathcal{O}(2^n) \leq \mathcal{O}(n!)$$

Órdenes de Complejidad



Órdenes de Complejidad



Cálculo de Complejidad de Programas Iterativos

```
PROCEDIMIENTO OrdenarSeleccion(REF v: vEnteros) ES

    i, j, min, aux : ENTERO;

    PARA i DESDE 0 HASTA MAX_vENTERO-2 HACER
        min := v[i];
        PARA j DESDE i+1 HASTA MAX_vENTERO-1 HACER
            SI  (v[j] < v[min]) ENTONCES
                min := j;
            FINSI
        FINPARA

        aux := v[min];
        v[min] := v[i];
        v[i] := aux;
    FINPARA
FINPROCEDIMIENTO
```

Cálculo de Complejidad de Algoritmos Iterativos

Teorema de Anidación

Sean $f(n), g(n) : \mathbb{N} \to \mathbb{R}^+$ y $g(n) \in \mathcal{O}(h(n))$, entonces $f(n) \cdot g(n) \in \mathcal{O}(f(n) \cdot h(n))$

Reglas informales para el cálculo de complejidades algorítmicas

1. La complejidad de una secuencia de instrucciones $S_1; S_2;$ es $max(\mathcal{O}(S_1), \mathcal{O}(S_2))$.
2. La complejidad de una sentencia condicional `SI cond ENTONCES` S_1 `SINO` S_2 `FINSI` es $max(\mathcal{O}(cond), \mathcal{O}(S_1), \mathcal{O}(S_2))$.
3. La complejidad de una instrucción iterativa que ejecuta $f(n)$ veces una secuencia de instrucciones de complejidad $\mathcal{O}(g(n))$ es $\mathcal{O}(f(n) \cdot g(n))$

Problema de la Convergencia

Algoritmo Babilónico para el Cálculo de una Raíz Cuadrada

```
// Pre: x debe ser positivo
FUNCION raiz(x : Real) : Real  ES

  base, altura : Real;
  base   = x;
  altura = 1.0;

  MIENTRAS (base != altura) HACER
      base   := (altura + base) / 2;
      altura := (x / base);
  FINMIENTRAS

  DEVOLVER base;
FINFUNCION
```

¿Cuántas iteraciones realiza el bucle?

Cálculo de la Complejidad de Algoritmos Recursivos

```
FUNCION factorial(n : ENTERO) : ENTERO ES

    result : ENTERO;

    SI (n == 0) ENTONCES
        result := 1;
    SINO
        result := n*factorial(n-1);
    FINSI

    DEVOLVER result;
FINFUNCION
```

$$
\begin{aligned}
\mathrm{t}(0) &= 3 \\
\mathrm{t}(\mathrm{n}) &= \mathrm{t}(\mathrm{n}\text{-}1) + 3
\end{aligned}
$$

Ecuaciones de recurrencia

Ecuación de recurrencia

El valor una función para un cierto n se expresa en función de los valores de la función para n's más pequeños.

$$t(n) = \begin{cases} g(n) & si\ 0 \leq n < b \\ a \cdot t(n-b) + h(n) & si\ \ n \geq b \end{cases}$$

Recurrencias para Algoritmos Divide y Vencerás

Esquema Algoritmos Divide y Vencerás

```
FUNCION DyV(p : Problema) : Solucion ES
    result : Solucion;

    SI esCasoBase(p) ENTONCES
        result := resuelve(p)
    SINO
        sub1, sub2 : Problema;
        sol1, sol2 : Solucion;

        divide(p,sub1,sub2);
        sol1 := DyV(sub1); sol2 := DyV(sub2);

        result := combina(sol1,sol2);
    FINSI

    DEVOLVER result;
FINFUNCION
```

Recurrencias para Algoritmos Divide y Vencerás

Ecuación de Recurrencia para Algoritmos Divide y Vencerás

$$t(n) = \begin{cases} c \cdot n^k & si \;\; 1 \leq n < b \\ a \cdot t(\frac{n}{b}) + c \cdot n^k & si \;\; n \geq b \end{cases}$$

$$a, c \in \mathbb{R}^+; k \in \mathbb{R}^+ \cup \{0\}; n, b \in \mathbb{N}; b > 1$$

Solución de la Recurrencia para Algoritmos Divide y Vencerás

$$t(n) \in \begin{cases} \Theta(n^k) & si \;\; a < b^k \\ \Theta(n^k \log(n)) & si \;\; a = b^k \\ \Theta(n^{\log(a)}) & si \;\; a > b^k \end{cases}$$

$$a, c \in \mathbb{R}^+; k \in \mathbb{R}^+ \cup \{0\}; n, b \in \mathbb{N}; b > 1$$

Complejidad Computacional

Complejidad Computacional

Rama de la teoría de la computación que se dedica a la clasificación de *problemas computables* de acuerdo a la dificultad inherente para su resolución.

Ejemplo: Todo algoritmo de ordenación sobre un vector aleatorio de elementos tiene una complejidad mínima de $\mathcal{O}(n \cdot \log n)$ [Brasard and Bratley, 2000].

Problemas P

Problemas (de decisión) para los cuales existe al menos un algoritmo que los resuelve en tiempo polinómico.

Problemas NP

Problemas NP

Problemas para los que se puede verificar si un resultado es solución válida en tiempo polinómico.

Relaciones P y NP

1. Teorema: $P \subseteq NP$
2. Conjetura: $P \neq NP$

Problema polinómicamente Turing reducible

Dados dos problemas A y B, decimos que A es *polinómicamente Turing reducible*, denotado $A \leq_T^p B$ si existe un algoritmo que resuelve A en tiempo polinomial asumiendo que existe un algoritmo que resuelve B en tiempo constante o unitario.

Problemas NP

Problemas polinómicamente Turing equivalentes

Dados dos problemas A y B, decimos que A y B son *polinómicamente Turing equivalentes*, denotado $A \equiv_T^p$, si $A \leq_T^p B$ y $B \leq_T^p A$.

Teorema de reducibilidad

Dados dos problemas A y B. Si $A \leq_T^p B$ y B puede ser resuelto en tiempo polinomial, A puede ser resuelto en tiempo polinomial.

Problemas polinómicamente reducibles muchos a uno

Sean dos problemas de decisión X e Y definidos sobre dominios de elementos I y J, respectivamente. X es *polinómicamente reducible muchos a uno* a Y, denotado $X \leq_m^p Y$, si existe una función $f(x) : I \to J$ computable en tiempo polinómico tal que $x \in X \Leftrightarrow \forall x \in I; f(x) \in J$. $f(x)$ se denomina *función de reducción*.

Problemas NP

Problemas polinómicamente equivalentes muchos a uno

Dados dos problemas de decisión X e Y se dice que son *polinómicamente equivalentes muchos a uno*, denotado $X \equiv_m^p Y$ si y sólo si $X \leq_m^p Y$ y $Y \leq_m^p X$.

Teorema de equivalencia entre reducciones

Sean X e Y dos problemas de decisión tales $X \leq_m^p Y$, entonces $X \leq_T^p Y$.

Problemas NP Completos

Un problema de decisión X es NP-completo si:

1. $X \in NP$
2. $\forall Y \in NP; Y \leq_T^p X$

Problemas NP completos

Si se encuentra un algoritmo en tiempo polinómico para un problema NP-completo, todos los problemas de NP serían resolubles en tiempo polinómico.

Teorema

Sea X un problema de decisión NP-completo, y sea Z un problemas de decisión $Z \in NP$ tal que $X \leq_T^p Z$, entonces Z es también NP-completo.

Problemas NP-difíciles

Un problema X es NP-difícil si existe un problema Y NP-completo tal que $Y \leq_T^p X$

Computabilidad

Computabilidad

Rama de la Teoría de la Computación que estudia qué clase de problemas son computables, es decir, pueden ser resueltos de forma efectiva por algún algoritmo.

Clases de problemas según su computabilidad

Decidibles Hay un algoritmo que encuentra la solución y termina en todos los casos.

Semidecidibles Hay un algoritmo que si hay solución, la encuentra y termina, sino hay solución puede no terminar.

No decidibles No hay algoritmo que resuelva dicho problema (e.g., problema de la parada).

¿Qué tengo que saber de todo esto?

1. Conocer y entender los conceptos de $\mathcal{O}(n), \Omega(n), \Theta(n)$.
2. Ser capaz de realizar operaciones básicas sobre órdenes de complejidad.
3. Ser capaz de estimar la complejidad de algoritmos iterativos sencillos.
4. Conocer las técnicas de estimación de la complejidad de algoritmos recursivos.
5. Ser capaz de especificar la ecuación de recurrencia asociada a la complejidad de un algoritmo recursivo.
6. Ser capaz de estimar la complejidad de algoritmos recursivos *divide y vencerás*.
7. Conocer y entender los conceptos básicos de complejidad computacional y computabilidad.