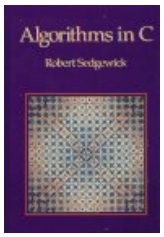
Análisis de algoritmos

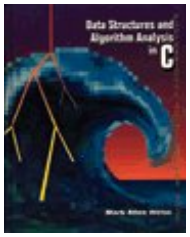
- **Eficiencia de un algoritmo**
- **Técnicas de diseño de algoritmos**

Bibliografía

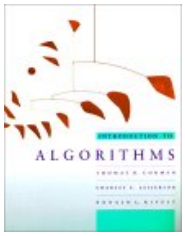


Robert Sedgewick:
"Algorithms in C"
Addison-Wesley, 1990
ISBN 0201514257

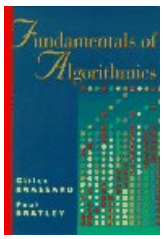
Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman:
"Data Structures and Algorithms"
Addison-Wesley, 1983
ISBN: 0201000237



Mark Allen Weiss
"Data Structures and Algorithm Analysis in C"
Addison-Wesley, Second Edition, 1996
ISBN 0201498405



Thomas H. Cormen, Charles E. Leiserson & Ronald L. Rivest
"Introduction to Algorithms"
MIT Press, 1990
ISBN 0262031418



Giles Brassard & Paul Bratley
"Fundamentals of Algorithmics"
Prentice Hall, 1996
ISBN 0133350681

"Fundamentos de Algoritmia"
Prentice Hall, 1997
ISBN 848966000X

Eficiencia de un algoritmo

El análisis de algoritmos estudia, desde el punto de vista teórico, los recursos computacionales que necesita la ejecución de un programa de ordenador: su eficiencia.

p.ej. Tiempo de CPU
Uso de memoria
Ancho de banda
…

NOTA:	En el desarrollo real de software, existen otros factores que, a menudo, son más importantes que la eficiencia: funcionalidad, corrección, robustez, usabilidad, modularidad, mantenibilidad, fiabilidad, simplicidad… y el tiempo del programador.

¿Por qué estudiar la eficiencia de los algoritmos?

Porque nos sirve para establecer la frontera entre lo factible y lo imposible.

Ejemplo: Algoritmos de ordenación

Observación
El tiempo de ejecución depende del tamaño del conjunto de datos.

Objetivo
Parametrizar el tiempo de ejecución en función del tamaño del conjunto de datos, intentando buscar una cota superior que nos sirva de **garantía**

Tipos de análisis

Peor caso (usualmente)
T(n) = Tiempo máximo necesario para un problema de tamaño n

Caso medio (a veces)
T(n) = Tiempo esperado para un problema cualquiera de tamaño n

- Requiere establecer una distribución estadística

Mejor caso (engañoso)

Análisis del peor caso

¿Cuál es el tiempo que necesitaría un algoritmo concreto?

- Varía en función del ordenador que utilicemos.
- Varía en función del compilador que seleccionemos.
- Puede variar en función de nuestra habilidad como programadores.

IDEA: Ignorar las constantes dependientes del contexto

SOLUCIÓN: Fijarse en el crecimiento de **T(n)** cuando **n** ® ¥

Notación O

$O(g(n)) = \{ f(n) \mid \exists c, n_0$ constantes positivas tales que $f(n) = c\, g(n) \ \forall\ n = n_0 \}$

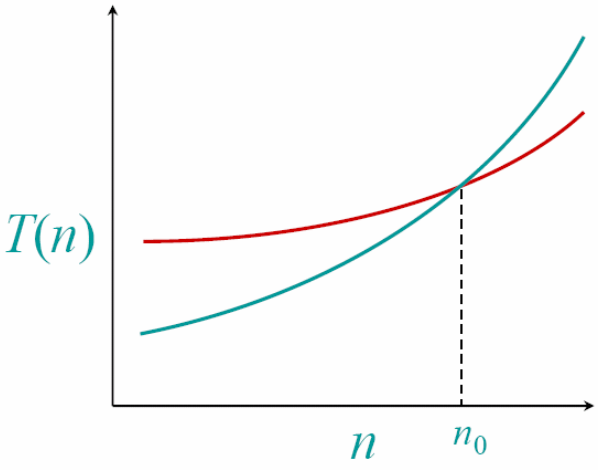
En la práctica, se ignoran las constantes y los términos de menor peso:

$$3n^3 + 90n^2 - 5n + 6046 = O(n^3)$$

Eficiencia asintótica

Cuando n es lo suficientemente grande…

un algoritmo **O(1)**
es más eficiente que
un algoritmo **O(log n)**
es más eficiente que
un algoritmo **O(n)**
es más eficiente que
un algoritmo **O(n log n)**
es más eficiente que
un algoritmo **O(n^2)**
es más eficiente que
un algoritmo **O(n^3)**
es más eficiente que
un algoritmo **O(2^n)**



NOTA: En ocasiones, un algoritmo más ineficiente puede resultar más adecuado para resolver un problema real ya que, en la práctica, hay que tener en cuenta otros aspectos además de la eficiencia.

Propiedades de la notación O

$$c\ O(f(n)) = O(f(n))$$

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

$$O(f(n)+g(n)) = O(f(n)+g(n))$$

$$O(f(n))O(g(n)) = O(f(n)g(n))$$

$$O(O(f(n))) = O(f(n))$$

Asignaciones y expresiones simples

```
x = x+1;
```

$T(n) = O(1)$

Secuencias de instrucciones

```
I1;
I2;
```

$T(n) = T_1(n) + T_2(n) = \max \{ O(T_1(n)), O(T_2(n)) \}$

Estructuras de control condicionales (if-then-else)

$$T(n) = O\ (T_{condición}(n)) + \max \{ O(T_{then}(n)), O(T_{else}(n)) \}$$

Estructuras de control iterativas

Producto del número de iteraciones por la complejidad de cada iteración.
Si la complejidad varía en función de la iteración, obtenemos una sumatoria.

> Si no conocemos con exactitud el número de iteraciones (bucles while y do-while), se estima este número para el peor caso.

Algoritmos recursivos

Se obtienen recurrencias que hay que resolver…

Ejemplo: Algoritmo de ordenación por inserción

```
void OrdenarInsercion (double v[], int N)
{
  int i, j;
  double tmp;

  for (i=1; i<N; i++) {

      tmp = v[i];

      for (j=i; (j>0) && (tmp<v[j-1]); j--)
          v[j] = v[j-1];

      v[j] = tmp;
  }
}
```

Peor caso: Vector inicialmente ordenado al revés

$$T(n) = \sum_{i=1}^{n-1} O(i) = O(n^2)$$

Caso promedio (con todas las permutaciones igualmente probables):

$$T(n) = \sum_{i=1}^{n-1} \Theta(i/2) = \Theta(n^2)$$

¿Es eficiente el algoritmo de ordenación por inserción?

✓ Moderadamente sí cuando n es relativamente pequeño.

✗ Absolutamente no cuando n es grande.

Ejemplo: Algoritmo de ordenación por mezcla (merge-sort)

IDEA

- Dividir el vector en dos y ordenar cada mitad por separado.
- Una vez que tengamos las mitades ordenadas, las podemos ir mezclando para obtener fácilmente el vector ordenado

IMPLEMENTACIÓN EN C

// Mezcla dos subvectores ordenados

```
double aux[maxN];

void merge (double v[], int l, int m, int r)
{
  int i, j, k;

  for (i=m+1; i>l; i--)           // Vector auxiliar
      aux[i-1] = v[i-1];          // O(n)
  for (j=m; j<r; j++)
      aux[r+m-j] = v[j+1];

  for (k=l; k<=r; k++)            // Mezcla
      if (aux[i]<aux[j])          // O(n)
         a[k] = aux[i++];
      else
         a[k] = aux[j--];
}
```

// Algoritmo de ordenación

```
void mergesort (double v[], int l, int r)   // T(n)
{
  int m = (r+l)/2;                          // O(1)

  if (r > l) {                              // O(1)
     mergesort (v, l, m);                   // T(n/2)
     mergesort (v, m+1, r);                 // T(n/2)
     merge (a, l, m, r);                    // O(n)
  }
}
```

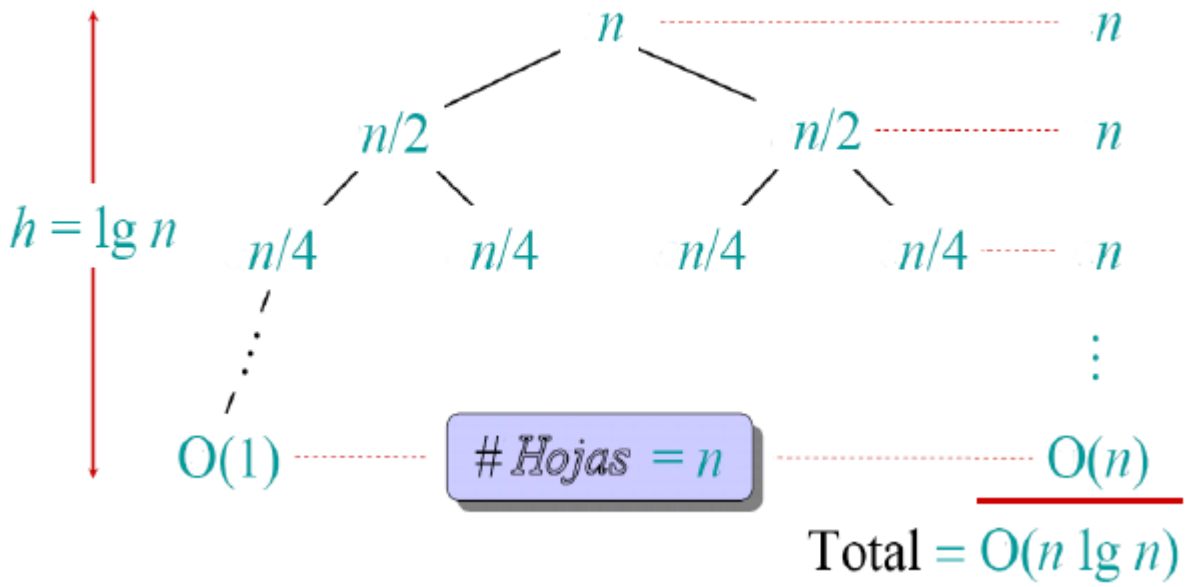
ANÁLISIS

1. Mezcla de dos subvectores ordenados (`merge`): $T_{merge}(n) = O(n^2)$

2. Algoritmo de ordenación

$$T(n) = \begin{cases} O(1) & si\ n = 1 \\ 2T(n/2) + O(n) & si\ n > 1 \end{cases}$$

Solución de la ecuación $T(n) = 2T(n/2) + n$



O(n log n) crece más despacio que $O(n^2)$

Por tanto,
la ordenación por mezcla es más eficiente que la ordenación por inserción.

Ejercicio
Comprobar experimentalmente a partir de qué valor de n el algoritmo de ordenación por mezcla es más rápido que el de ordenación por inserción.

Resolución de recurrencias

Se pueden aplicar distintas técnicas y trucos:

Método de sustitución

1. Adivinar la forma de la solución.
2. Demostrar por inducción.
3. Resolver las constantes.

$$T(n) = \begin{cases} O(1) & si\ n = 1 \\ 4T(n/2) + n & si\ n > 1 \end{cases}$$

¿T(n) es $O(n^3)$?

Suposición $T(k) = ck^3$ para k<n

Demostramos por inducción $T(n) = cn^3$

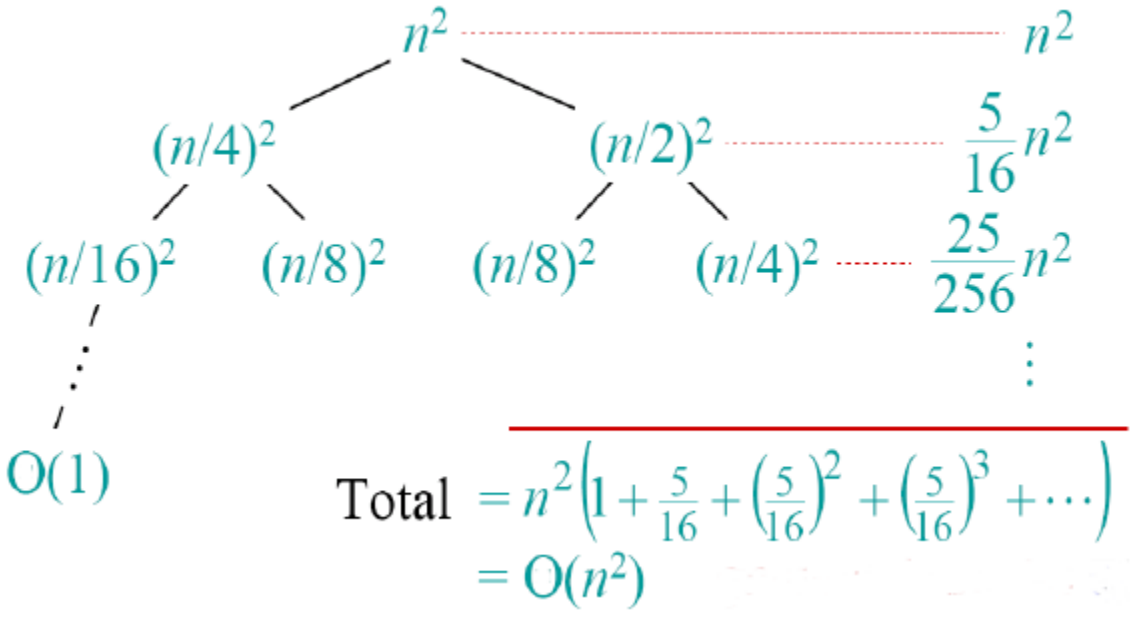
$T(n) = 4T(n/2) + n$
$= 4c(n/2)^3 + n = (c/2)n^3 + n = cn^3 - ((c/2)n^3 - n)$
$= cn^3$ siempre que $((c/2)n^3 - n)>0$ (p.ej. c=2 y n=1)

PROBLEMA: ¿Podríamos encontrar una cota superior más ajustada?
Sugerencia: Probar con $T(n) = cn^2$ y $T(n) = c_1n^2-c_2n$

Árbol de recursión

Se basa en construir una representación gráfica intuitiva…

$$T(n) = T(n/4) + T(n/2) + n^2$$



Expansión de recurrencias

Equivalente algebraica al árbol de recurrencias

Ejemplo: Cálculo del factorial

```
int factorial (int n)
{
  return (n==0)? 1: n*factorial(n-1);
}
```

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

T(n) = T(n-1) + 1
= (T(n-2)+1) + 1 = T(n-2) + 2
= (T(n-3)+1) + 2 = T(n-3) + 3
…
= T(n-k) + k

Cuando k=n:

T(n) = T(0) + n = 1 + n = O(n) Algoritmo de orden lineal

Ejemplo: Sucesión de Fibonacci

```
int fibonacci (int n)
{
  if ((n == 0) || (n == 1))
     return 1;
  else
     return fibonacci(n-1) + fibonacci(n-2);
}
```

$$T(n) = \begin{cases} O(1) & si\ n \le 1 \\ T(n-1)+T(n-2)+1 & si\ n > 1 \end{cases}$$

T(n) = T(n-1) + T(n-2) + 1
= (T(n-2)+T(n-3)+1) + (T(n-3)+T(n-4)+1) + 1
= T(n-2) + 2T(n-3) + T(n-4) + (2+1)
= T(n-3) + 3T(n-4) + 3T(n-5) + T(n-6) + (4+2+1)
…

$$T(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1+\sqrt{5}}{2}\right)^n\right]$$

Método de la ecuación característica

Recurrencias homogéneas lineales con coeficientes constantes

$$T(n) = c_1T(n-1) + c_2T(n-2) + \ldots + c_kT(n-k)$$

$$\mathbf{a_0t_n + a_1t_{n-1} + \ldots + a_kt_{n-k} = 0}$$

Sustituimos $t_n=x^n$:

$$a_0x^n + a_1x^{n-1} + \ldots + a_kx^{n-k} = 0$$

Ecuación característica (eliminando la solución trivial x=0):

$$\mathbf{a_0x^k + a_1x^{k-1} + \ldots + a_k = 0}$$

Obtenemos la solución a partir de las raíces del polinomio característico:

Caso 1: Raíces distintas r_i

$$t_n = \sum_{i=1}^{k} c_i r_i^n$$

Caso 2: Raíces múltiples r_i de multiplicidad m_i

$$t_n = \sum_{i=1}^{l} \sum_{j=1}^{m_i-1} c_{ij} n^j r_i^n$$

Finalmente, las constantes se determinan a partir de las k condiciones iniciales.

DEMOSTRACIÓN:

Aplicando el Teorema Fundamental del Álgebra,
factorizamos el polinomio de grado k como un producto de k monomios.

Si $p(r_i)=0$, r_i es una raíz del polinomio característico
$x=r_i$ es una solución de la ecuación característica
r_i^n es una solución de la recurrencia.

Cuando tenemos una raíz múltiple *r* de multiplicidad *m*, podemos llegar a la conclusión de que r^n, nr^n, n^2r^n … $n^{m-1}r^n$ son otras soluciones de la recurrencia.

Recurrencias no homogéneas lineales

A partir de

$$\mathbf{a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n p(n)}$$

donde b es una constante y p(n) un polinomio de grado d

podemos derivar un polinomio característico

$$\mathbf{(a_0 x^k + a_1 x^{k-1} + \ldots + a_k) (x - b)^{d+1}}$$

que resolveremos igual que el caso homogéneo.

Ejemplo: Las Torres de Hanoi

```
void hanoi (int n, int inic, int tmp, int final)
{
  if (n > 0) {
     hanoi (n-1, inic, final, tmp);
     printf ("Del poste %d al %d.\n", inic, final);
     hanoi (n-1, tmp, inic, final);
  }
}
```

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

Ecuación recurrente no homogénea: $T(n) - 2T(n-1) = 1$

Ecuación homogénea	$x-2 = 0$
Constante b	$b = 1$
Polinomio p(n)	$p(n) = 1$
Polinomio característico	$(x-2)(x-1)$
Solución:	$T(n) = c_1 1^n + c_2 2^n$

Condiciones iniciales: $T(0) = 0$ → $c_1+c_2 = 0$ → $c_1 = -1$

$T(1) = 2T(0)+1 = 1$ → $c_1+2c_2 = 1$ → $c_2 = 1$

$T(n) = 2^n - 1$ ➔ Algoritmo de orden exponencial $O(2^n)$

Técnicas de diseño de algoritmos

Divide y vencerás

1. Dividir el problema en subproblemas independientes.
2. Resolver los subproblemas de forma independiente.
3. Combinar las soluciones de los subproblemas.

Ejemplos: Búsqueda binaria
Ordenación por mezcla (merge-sort)
Ordenación rápida (quicksort)
Búsqueda de la mediana (≈quicksort)
Multiplicación de matrices (algoritmo de Strassen)
…

Problema: Calcular a^n

Algoritmo ingenuo: Eficiencia O(n)

$a^n = a \cdot a^{n-1}$

Estrategia divide y vencerás: $T(n) = T(n/2) + O(1) = O(\log_2 n)$

$$a^n = \begin{cases} a^{n/2}a^{n/2} & si\ n\ es\ par \\ a^{(n-1)/2}a^{(n-1)/2}a & si\ n\ es\ impar \end{cases}$$

Problema: Sucesión de Fibonacci

Algoritmo recursivo básico: Tiempo exponencial $O(\phi^n)$

Estrategia divide y vencerás: Eficiencia $O(\log_2 n)$

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

Algoritmos voraces (greedy)

En cada momento se toma una decisión local irrevocable

- Técnica especialmente adecuada cuando podemos garantizar que la mejor solución local nos lleva a la solución global del problema.
- Incluso, puede resultar adecuada cuando no tenemos esa garantía para obtener rápidamente una aproximación a la solución del problema: **heurísticas**.

Aplicaciones: Problema del viajante de comercio (problema NP: O(n!)).

Estrategia greedy
Solución rápida pero no siempre óptima

Ejemplo: Algoritmo de Dijkstra **$O(n^2)$**

Dado un grafo, obtener el camino más corto para llegar desde un vértice hasta otro:

`V`	Conjunto de vértices del grafo (p.ej. ciudades)
`S`	Conjunto de vértices cuya distancia más corta desde el origen ya se conoce
`D[i]`	Distancia más corta entre el origen y el vértice i.
`C[i,j]`	Coste de llegar desde el nodo i hasta el nodo j (p.ej. kilómetros)
`P[i]`	Vértice anterior a i en el camino más corto desde el origen hasta i (para poder reconstruir el camino más corto)

```
S = { origen }

Para todo i
    D[i] = C[origen,i]
    P[i] = origen

Mientras S ≠ V
    Elegir w ∈ V – S tal que D[w] sea mínimo
    S = S ∪ {w}
    Para cada vértice v ∈ V – S
        Si D[v] > D[w] + C[w,v] entonces
            D[v] = D[w] + C[w,v]
            P[v] = w
```

Programación dinámica

Cuando la solución óptima de un problema se puede construir a partir de las soluciones óptimas de subproblemas del mismo tipo.

p.ej. Cuando la implementación recursiva de un algoritmo repite muchas veces las mismas operaciones.

Estrategia de programación dinámica
- Trabaja etapa por etapa.
- Compara resultados con los ya obtenidos (uso masivo de memoria).

Uso de memoria O(f(n)) ➔ Tiempo de ejecución O(nf(n))

Ejemplos

Sucesión de Fibonacci: Solución iterativa

$$fib(n) = fib(n-1) + fib(n-2)$$

Números combinatorios

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Problema del camino mínimo (algoritmo de Floyd)

$$D_k(i,j) = \min\{D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j)\}$$

Variante: "Memoization"

IDEA: No calcular dos veces lo mismo

Cada vez que se obtiene la solución de un subproblema,
ésta se almacena en una tabla para no tener que volver a calcularla.

Aplicaciones
Reconocimiento de voz (DTW: Dynamic Time Warping).