Algoritmos y Estructuras de Datos
Tema 2: Diseño de Algoritmos

Contenidos

Complejidad de Algoritmos

- Un mismo problema se puede resolver con muy diferentes algoritmos
- Para hacer una implementación debemos seleccionar uno. Criterios para seleccionar:
  - Seleccionar el que emplea **menor número de recursos** (por ejemplo, memoria, **cómputo**, ancho banda)
  - Seleccionar el algoritmo **menos complejo** (por ejemplo número de líneas de código)
  - Seleccionar aquel del que tenemos implementación **probada**

Complejidad de Algoritmos

- Los algoritmos se **analizan** para
  - Fijar una **clasificación** basada en los criterios de los análisis
  - **Predecir** el comportamiento **antes de implementar**
    - Predecir **eficiencia en el uso de recursos**
    - Predecir **escalabilidad**. Simples pruebas no bastan para

Complejidad de Algoritmos

- Criterio mas empleado: **tiempo de ejecución**
  - Cuantificable (hay varias formas) y fácil de comparar
  - Tiempos de ejecución suelen ser el origen de **cuellos de botella**
- El tiempo de ejecución de los algoritmos depende de **varios factores**:
  - **Plataforma hardware**. Por ejemplo, cache de multicores puede generar resultados muy diferentes para algoritmos muy similares
  - **Lenguaje de programación** y su ejecución
  - Estilos de *eficiencia de los programadores*
- Notación ***Big O***: análisis teórico que asume algunas simplificaciones
  - Estimar medida de tiempo de ejecución en función de la cantidad de datos tratados

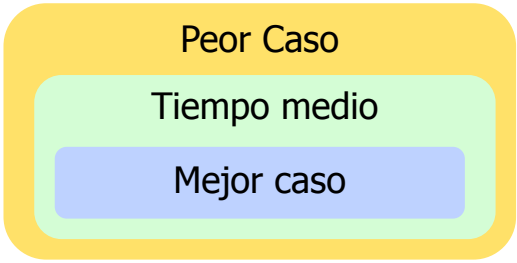
Medir cantidad de datos

- Necesitamos **medir la cantidad de los datos tratados** para poder estimar tiempos de ejecución, y después encontrar como se relacionan
- Algunos ejemplos de medidas:

Algoritmo	***Tamaño de datos***
Buscar en una lista	Número de elementos en la lista

Tiempos de ejecución

- No buscamos valores numéricos; buscamos **estimaciones de coste** en función de la cantidad de datos tratados
- Formas de medir tiempos de ejecución
  - **Tiempo medio:**
    - Valor ideal, pero difícil de predecir de forma teórica
  - **Mejor caso:**
    - Poco útil
  - **Peor caso:**
    - Es una estimación pesimista
    - Los análisis *de complejidad* asumen que lo importante es la escalabilidad y que el tamaño de los datos de entrada puede ser muy grande



Porque algoritmos eficientes?

- Supongamos que trabajamos sobre $n=10^6$ valores
  - Por ejemplo: operaciones registradas en un banco en un año
- Supongamos que un PC tarda 1 sec en *leer/procesar n* datos
- Pero si un algoritmo es de orden *n*n*. **Tarda un**

Función de complejidad (1)

- El **resultado del análisis** de complejidad será una **función *f(n)***, donde *n* es la cantidad de datos con la que tratamos
  - *f(n)* representa el trabajo que el algoritmo debe hacer para obtener el resultado
- *f(n)* es solo **una estimación**. En la práctica, el esfuerzo depende de los valores concretos que tengan los valores de entrada
  - En la práctica, **dos conjuntos** de datos de entrada del **mismo tamaño**, tienen **tiempos** de ejecución **distintos**

Función de complejidad (2)

- Lo que representamos en la función es **como** de rápido **va creciendo el coste de ejecución** según crece el tamaño de los datos tratados
  - Si *f(n)* es una función polinomial de orden *r* la complejidad será $O(n^r)$
- Órdenes mas utilizados

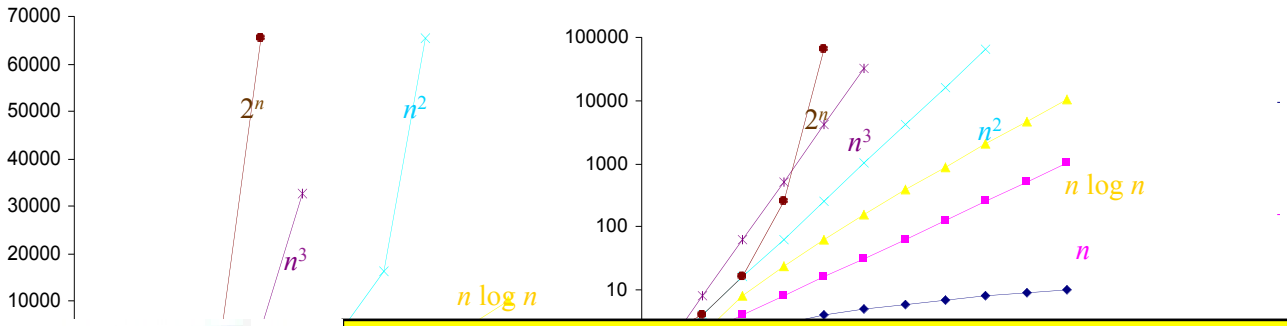
$O(\log n)$	de ejecución crece en una función logarítmicamente proporcional a la del de los datos tratados

Valor de las funciones

- Supongamos que el ordenador ejecuta 10^9 operaciones/sec. Los tiempos totales son del orden de:

	$f(n)$						
n	n	$n \log n$	n^2	n^3	n^4	n^{10}	2^n
10	.01μs	.03μs	.1μs	1μs	10μs	10s	1μs
20	.02μs	.09μs	.4μs	8μs	160μs	2.84h	1ms
30	.03μs	.15μs	.9μs	27μs	810μs	6.83d	1s
40	.04μs	.21μs	1.6μs	64μs	2.56ms	121d	18m
50	.05μs	.28μs	2.5μs	125μs	6.25ms	3.1y	13d
100	.1μs	.66μs	10μs	1ms	100ms	3171y	4×10^{13}y
10^3	1μs	9.96μs	1ms	1s	16.67m	3.17×10^{13}y	32×10^{283}y
10^4	10μs	130μs	100ms	16.67m	115.7d	3.17×10^{23}y	
10^5	100μs	1.66ms	10s	11.57d	3171y	3.17×10^{33}y	
10^6	1ms	19.92ms	16.67m	31.71y	3.17×10^{7}y	3.17×10^{43}y	

Valor de las funciones (2)



Algunos Ejemplos

- Suma de array de reales

```
for (int i=0; i<n; i++)
    Z[i] = A[i] + B[i]; // esto tarda en ejecutarse c
```

  - $f(n)=c\ n \rightarrow O(n)$
- Multiplicación de todos los elementos de un array

```
for (int i=0; i<n; i++)
    z = z * A[i]; // esto tarda en ejecutarse c2
```

  - $f(n)=c_2\ n \rightarrow O(n)$
- Multiplicación de 2 arrays

```
for (int i=0; i<n; i++)
   for (int j=0; j<n; j++)
      Z[i,j] = A[i] * B[j];
```

  - $f(n)=c_3\ n^2,\ O(n^2)$
- Un programa que hace las tres cosas
  - $f(n)=c\ n + c_2\ n + c_3\ n^2 \rightarrow O(n^2)$

Simplificación de la función

- Funciones como

$$c\ n + c_2\ n + c_3\ n^2$$

son incomodas de usar y comparar

- Nos interesan los **valores dominantes.** Nos quedamos el termino que hace crecer mas rápidamente la función $O(n^2)$

Análisis de algoritmos

- **Sentencias simples** con operaciones simples
  - Ejemplo: *a=b+c;*
  - *O(1)*
- Sentencias simples con **llamadas a método**
  - Ejemplo: *m(x)*
  - La función y complejidad del algoritmo que implementa el método
- **Secuencia** de sentencias
  - Ejemplo: *m(x); a=b+c;*
  - La complejidad la determina aquella que es dominante
- **Bucles** un número fijo de veces
  - Ejemplo: *for (int i=0; i < 20; i++) m(x);*
  - La función de complejidad es la operación ejecutada multiplicada por el número de iteraciones

Análisis de algoritmos

- Sentencias **condicionales**

if (cond) then	*O(1)*
body$_1$	$f_1(n)$
else	
body$_2$	$f_2(n)$
endif	

  - La complejidad es la dominante de las dos alternativas. Si solo hay

Análisis de algoritmos

- Bucles anidados o bucles con cuerpos no simples:
  - Ejemplo:

```
for (i=1; i<n; i++)
   for (j=n; j>= i+1;  j--)
      if (A[j-1] > A[j]) {
            temp = A[j-1];
            A[j-1] = A[j];
            A[j] = tmp;
```

  - Complejidad $O(n^2)$
- Recursividades
  - Las **recursividades simples** se pueden tratar como **bucles**
  - En otros casos hay que emplear **aproximaciones recurrentes** que dependen del algoritmo, y se hacen suponiendo niveles de recursiones de peor caso

Ejemplo práctico: serie Fibonacci

- La serie de Fibonacci es:

  1,1,2,3,5,8,13, …
- La función de la serie devuelve 1 para los argumentos 1 y 2 y los siguientes son la suma de los dos anteriores
- Dos soluciones, *cual debemos utilizar?*

Algoritmo 1

Algoritmo 2

}

Ejemplo práctico: serie Fibonacci

Algoritmo 1

```
int f(int i) {
   if (n <= 2)
      return 1;
   return f(i-2)+f(i-1);
}
```

Algoritmo 2

```
int f(int i) {
   int a=1; int b=1; int c=1;
   while (i-- > 2) {
      c=a+b; a=b; b=c;
   }
   return c;
}
```

- El *algoritmo 2* es *O(n)*, donde **n** es el valor para el que queremos calcular la función
- El *algoritmo 1* es $O(2^n)$. Para calcular *f(x)*-> f sigue los pasos:
  - *Paso 1*: 2 llamadas a f
  - *Paso 2*: 2*2 llamadas a f
  - *Paso 3*: 2*2*2 llamadas a f
  - *Paso k*: $2*2^{k-1}$ llamadas a f