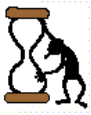
Análisis de algoritmos

Complejidad



Índice

- *¿ Por qué es necesario ?*
- *¿Cómo estimar el tiempo de ejecución ?*
  - De qué depende
  - Como calcularlo
    - Experimentalemente
    - Estimándolo matemáticamente
- Análisis de algoritmos
  - Operaciones primitivas
  - Caso medio y caso peor
  - Notación asintótica
- *Limitaciones del análisis asintótico*

Complejidad
¿Por qué es necesario su análisis?

- Una vez dado un algoritmo para resolver un problema y comprobado que es correcto
- Hay que determinar su **complejidad computacional** (cantidad de recursos que necesita para su ejecución)
- ***¿Cómo estimar el tiempo de ejecución ?***
  - Tiempo de computación
  - Espacio
    - En memoria
    - En disco

Complejidad
Cómo estimar el tiempo de ejecución ?

- ¿Qué medimos? Tiempo de ejecución
- ¿De qué depende?
  - Tamaño de la entrada
  - Otros factores (Hw y Sw)
    - Velocidad de la máquina
    - Calidad del compilador
    - Calidad del programa
- ¿Cómo medirlo?
  - Experimentalmente
  - Estimandolo matemáticamente

4

Complejidad
¿Cómo medimos?. Experimentalmente

- Midiendo el tiempo de ejecución en función del tamaño de la entrada
  - No podemos probar todas las entradas
  - Es necesario implementar el algoritmo
  - Depende del Software y Hardware
- Busquemos otra medida
  - Mas abstracta
  - Mas fácil de obtener

5

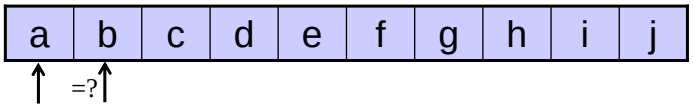
Complejidad
¿Cómo medimos?. Objetivo

- Considerar todas las entradas
- Independiente del entorno de Software y Hardware
- Puedan ejecutarse estudiando una descripción de alto nivel de algoritmo, sin implementarlo

6

Complejidad
¿Cómo medimos?

- Vamos a suponer que tenemos un algoritmo que verifica si los valores de la primera y segunda posición de un array son iguales



- ¿Cuantas comparaciones necesitamos si el array tiene 10 elementos?
- ¿Cuantas comparaciones necesitamos si el array tiene 100 elementos?

7

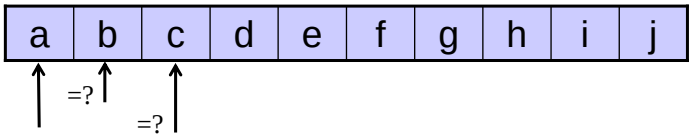
Complejidad
¿Cómo medimos?

- El número de comparaciones a realizar es constante
  (no hay que realizar más comparaciones aunque haya más elementos)
- El algoritmo es independiente del tamaño del array
- La complejidad del algoritmo es $O(1)$
  (la velocidad de crecimiento del algoritmo es constante)

8

Complejidad
¿Cómo medimos?

- Vamos a suponer que tenemos un algoritmo que verifica si el valor de la primera posición de un array es igual al de la segunda y tercera posición

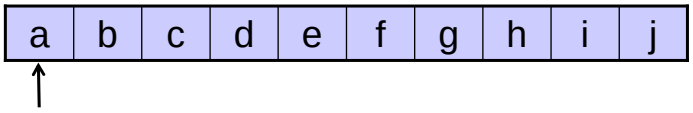


- El número de comparaciones a realizar es constante
(no hay que realizar más comparaciones aunque haya más elementos)

9

Complejidad
¿Cómo medimos?

- Vamos a suponer que tenemos un algoritmo que verifica si el valor de la primera posición de un array está repetido



- ¿En el peor de los casos cuantas comparaciones habrá que realizar?

n-1 → ¿Cúal es el termino dominante?
(cuando n es muy grande…)

10

Complejidad
¿Cómo medimos?. Estimación matemática

- Buscamos para cada algoritmo una función que dependa de la entrada: f(n)
- Emplearemos el caso peor para caracterizar el tiempo de ejecución
- Interesa la velocidad de crecimiento del tiempo de ejecución en función del tamaño de la entrada

11

Complejidad
¿Cómo medimos?. Estimación matemática

- Método de “contar pasos”:
  - Tiempo constante
    - asignación, llamada método, op. aritmética, index. array, escribir/leer un valor .....
  - Suma de tiempo de cada iteración
    - Ciclos (más eval. de la condición en cada paso)
  - Máximo de tiempos
    - Instrucción condicional
  - Coste adicional
    - Llamada a subprogramas

12

Complejidad: Ejemplo

¿Cómo medimos?. Estimación matemática

- Vamos a suponer que se quiere desarrollar el siguiente método:

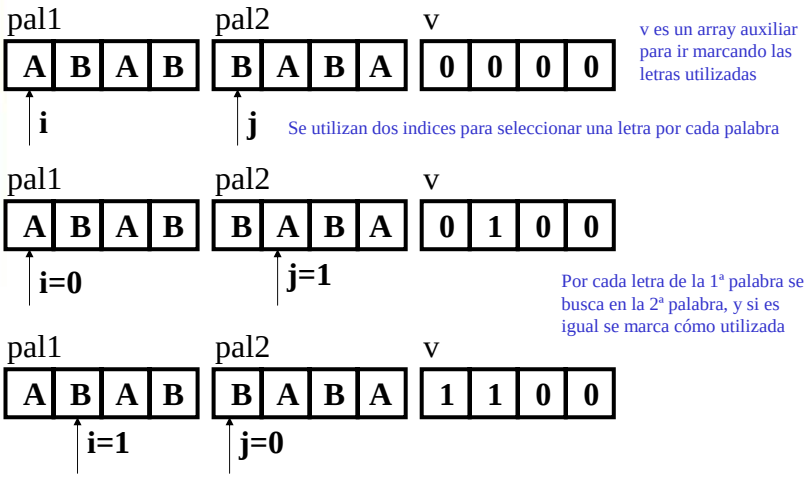
```
public boolean esAnagrama(Palabra pal1, Palabra pal2)
```

```
//Pre: pal1 y pal2 representan dos palabras, estando todas sus
//      letras en mayúsculas
//Post: Devuelve True si pal1 y pal2 tienen las mismas letras
```

Por ejemplo: pal1="VITORIA" y pal2="VORIATI" son anagrama

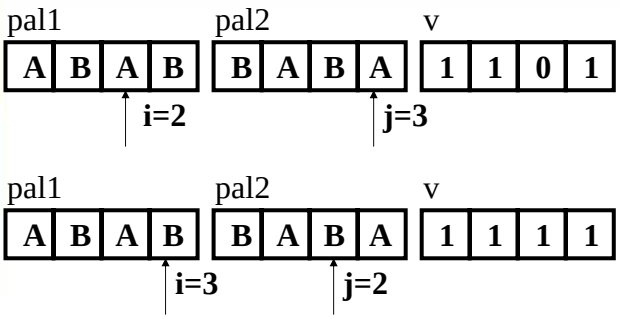
13

Ejemplo: esAnagrama? (I)



14

Ejemplo: esAnagrama? (I)



Coste:

4 marcas iniciales

Buscar cada letra de pal1 en pal2

4*4 comprobaciones (en el caso peor)

Ver si las 4 letras han sido marcadas (en el caso peor)

15

EsAnagrama(I)

Programa Java

```
public static boolean esAnagrama (Palabra p1, Palabra p2) {
  char[ ] pal1 = p1.getLetras();
  char[ ] pal2 = p2. getLetras();
  int[ ] v = new int[4]; int p,j;
  //Inicializar posiciones visitadas
  for (int i=0;i<4;i++)
      v[i]=0;
  for (int i=0;i<4;i++) {
      j=0;
     while ( (j<4) && !(pal1[i]==pal2[j] && v[j]==0) )
         j++;
     if ( (j<4) && pal1[i]==pal2[j] && v[j]==0 )
        v[j]=1;
  }
  p=0;
  while ((p<4) && (v[p]==1))
      p++;
  if (p==4) return true;
    else return false;
}
```

16

Ejemplo: esAnagrama? (II)

pal1 pal2

A	H	A	H

H	A	H	A

0	0	0	0	0	0	0	0	....	0	0	0	0
A	B	C	D	E	F	G	H		W	X	Y	Z

2	0	0	0	0	0	0	2	....	0	0	0	0
A	B	C	D	E	F	G	H		W	X	Y	Z

incrementar las letras del pal1 y decrementar las de pal2

0	0	0	0	0	0	0	0	....	0	0	0	0
A	B	C	D	E	F	G	H		W	X	Y	Z

Coste:

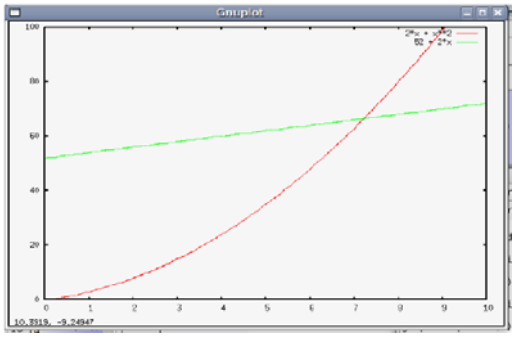
Asignar 26 ceros
Hacer 4 incrementos (pal1)
Hacer 4 decrementos (pal2)
Hacer 26 comprobaciones (en el caso peor)

17

Comparativa algoritmos

Tamaño	Algoritmo1	Algoritmo2
4	4+4*4+4=24	26+4+4+26=60
n	n+n*n+n=$2n+n^2$	26+n+n+26=52+2n

Cuando n es lo bastante grande el algoritmo2 es mas eficiente que el algoritmo1.



18

Complejidad

Cómo se expresa el coste en tiempo de un algortimo

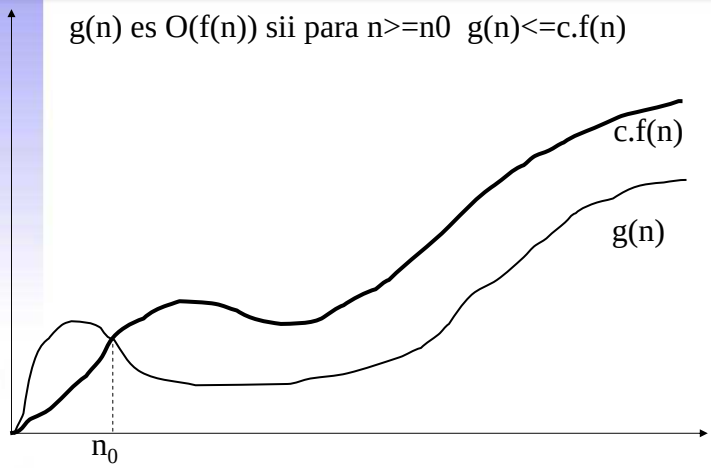
- Clasificación de las funciones para expresar cómo crece el tiempo en función del tamaño de los datos.
- *O* mayúscula de f: *O*(f)
  - Denota el **conjunto de funciones** en *O*(f) que crecen “a lo más tan rápido como f”

$$O(f) = \{g \mid \exists c \in R\ \exists n0 \in N\ \forall n > n0\ g(n) < c.f(n)\}$$

19

Notación O

g(n) es O(f(n)) sii para n>=n0 g(n)<=c.f(n)

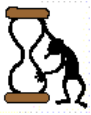


$$O(f) = \{g \mid \exists c \in R\ \exists n0 \in N\ \forall n > n0\ g(n) < c.f(n)\}$$

Ejemplos

- 7n-3 es $O(n)$
  - c=7, n0=1
  - 7n-3 <= 7n
- $20n^3+10n \log n+5$ es $O(n^3)$
- $3 \log n+\log(\log n)$ es $O(\log n)$
- 2^{100} es $O(1)$
- 5/n es $O(1/n)$

21



Notación O

- Formas de crecimiento más habituales
  - $O(\log n)$ logarítmica
  - $O(n)$ lineal
  - $O(n.\log n)$
  - $O(n^2)$ cuadrática
  - $O(n^3)$ cúbica
  - $O(n^m)$ polinómica
  - $O(2^n)$ exponencial
- $O(\log n)$ $O(n)$ $O(n.\log n)$ $O(n^2)$ $O(n^3)$ $O(n^m)$ $O(2^n)$
- Interesa afinar lo más posible en escala de $O(f)$

22

Crecimiento de funciones

log n	$\sqrt{n}$	n	n log n	n^2	n^3	2^n
1	1,4	2	2	4	8	4
2	2,0	4	8	16	64	16
3	2,8	8	24	64	512	256
4	4,0	16	64	256	4.096	65.536
5	5,7	32	160	1.024	32.768	4.294.967.296
6	8,0	64	384	4.096	262.144	$1,8 * 10^{19}$
7	11,0	128	896	16.536	2.097.152	$3,4 * 10^{38}$

23

Complejidad

Justificación de la notación asintótica

- Para un N suficientemente grande, el valor de la función esta completamente determinado por su término dominante
- El valor del coeficiente del termino dominante se conserva al cambiar de máquina
- Los valores pequeños de N, generalmente, no son importantes

24

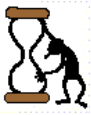
Ejemplo I

- Dado un array de enteros, hallar el máximo
  - Entrada: Un array A con n enteros
  - Salida: el elemento mayor de A
- Algoritmo: `arrayMax(A, n)`

```
current=A[0];
for(i=1;i<n;i++)
        if (A[i]>current)
    current=A[i];
return current
```

  - Caso mejor = 2+4*(n-1)+1 = 4n-1
  - Caso peor = 2+6*(n-1)+1 = 6n-3

25

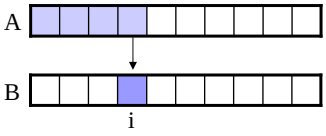


Ejemplo II

- Dado un array A de n números, calcular otro array B, tal que:

$$B[i] = \sum_{j=0}^{i} A[j]$$

A partir de una secuencia de números A[j] calculamos otra (B[i]) tal que cada uno de sus elementos sea la suma de todos los anteriores en la secuencia original



26

Ejemplo II
Solución 1

```
for (i=0;i<n;i++) {
      s=0;
      for(j=0;j<=i;j++)
            s=s+A[j];
      B[i]=s;
}
return B;
```

Partes:
Inicializar y devolver el array:O(n)
Bucle i: se ejecuta n veces: O(n)
Bucle j: se ejecuta 1+2+3+...+n=(1+n)n/2 : $O(n^2)$

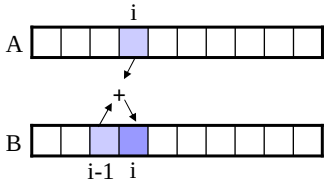
Total: O(n)+O(n)+$O(n^2)$ es $O(n^2)$

27

Ejemplo II
Solución 2

B[i]= A[0]+A[1]+ ....A[i-1] + A[i]
B[i]= B[i-1] + A[i]

Aprovechamos las operaciones realizadas en el término anterior para reducir la complejidad del algoritmo



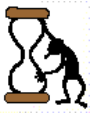
28

Ejemplo II
Solución 2

```
B[0]=A[0]
for(i=1;i<n;i++)
    B[i]=B[i-1]+A[i]
return B
```

Partes:
Inicializar y devolver el array A[]:O(n)
Bucle i: se ejecuta n-1 veces
Total: O(n)+O(1)+O(n) es O(n) (Orden lineal)

29



Complejidad
Limitaciones del análisis asintótico

- No es apropiado para **pequeñas cantidades de datos**
- A veces el análisis asintótico es una **sobreestimación**
- La cota de tiempo de ejecución en el caso promedio puede ser significativamente menor que la cota en el caso peor
- El caso peor en ocasiones es poco representativo por lo que puede ser ignorado

30