

Algorítmica (1): Análisis de complejidad

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Septiembre 2013



Outline

- 1 Complejidad y notación asintótica
- 2 Analisis de complejidad



Outline

1 Complejidad y notación asintótica

2 Analisis de complejidad



Complejidad

Para poder caracterizar implementaciones: **aislar la o las “variables” que permiten caracterizar la complejidad del problema que resolver**, para expresar la complejidad de los operaciones. Típicamente, un numero de elementos dentro del contenedor. . .

Ejemplos:

- Pila.
- Cadena de caracteres.
- Grafo.



Complejidad en espacio

La primera manera de caracterizar una implementación es en términos de la **memoria** que requiere (espacio): **una función $M(n)$, característica del algoritmo, en función del tamaño de los datos (n), que permite evaluar la cantidad de memoria requerida para usar el algoritmo.**



Complejidad en tiempo

Es una función $T(n)$ que describe el comportamiento del algoritmo en cuanto a su tiempo de ejecución, en términos de n , el tamaño de los datos de input al algoritmo...

Se expresa esta función $T(n)$ de manera exacta si posible, en términos de las operaciones elementales (operaciones aritméticas, operaciones de asignación, de dereferenciación), y no en términos de tiempo absoluto... Lo que nos interesa es la tasa de crecimiento, la tendencia cuando n es grande.



Complejidad en tiempo

Aunque una operación elemental puede ser más costosa en tiempo que otra, por ejemplo, no considerar la constante multiplicativa que hay entre los dos. En cambio, se considera la complejidad en terminos de número total de **operaciones elementales**.

```
void Ex1(int n) {  
    int i, j;  
    int sum=0, summ=0, limit=n*n;  
    for (i=1; i<limit; i++)  
        ++sum  
    for (j=1; j<sum; j++)  
        ++summ;  
}
```



Complejidad en tiempo

En el ejemplo:

$$\begin{aligned} T(n) &= 4 + 1 + n^2 + 2(n^2 - 1) + 1 + n^2 - 1 + 2(n^2 - 2) \\ &= 6n^2 - 1 \end{aligned}$$

Aquí lo que nos importa realmente es sobre todo que se comporta como n^2 para grandes valores de n .



Complejidad en tiempo

A veces no se puede establecer una forma exacta o aun aproximada de $T(n)$, sino **expresarla estadísticamente**:

- en **promedio**
- en el **peor de los casos**

Ejemplo: búsqueda en un *quadtree*.

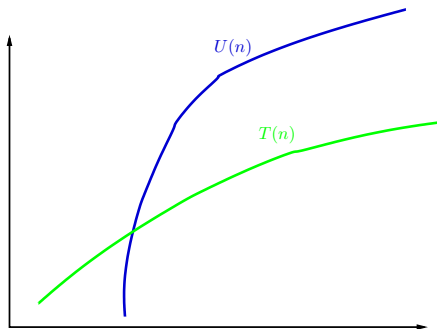


Gran O

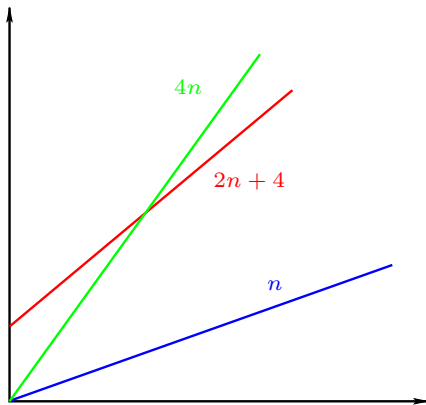
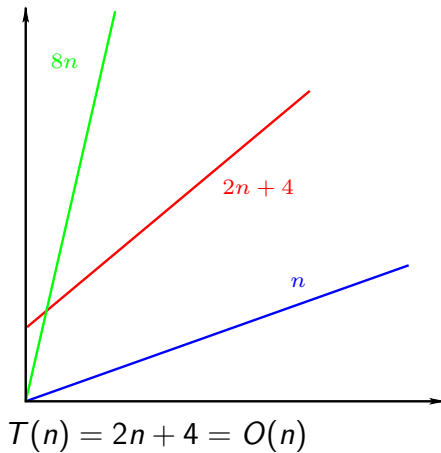
Como lo que nos interesa es el **comportamiento asintótico de las complejidades**, se usa herramientas para comparar esos comportamientos cuando $n \rightarrow \infty$.

Dadas dos funciones T y U de \mathbb{N} en \mathbb{R} , se dice que

$T(n) = O(U(n))$ ssi existe un real $c > 0$ y un entero n_0 tal que $\forall n \geq n_0, T(n) \leq cU(n)$



Gran O



Gran O

- La notación gran O **define una cuota superior a una función** y una **familia** de tendencia de crecimiento.
- $T(n) = O(U(n))$ significa que **el crecimiento de T es inferior o igual al de U para grandes valores de n .**
- Se puede **comparar funciones**:

$$1 \leq \log n \leq n \leq n \log n \leq n^2 \leq 2^n \leq n^n.$$

- Las constantes multiplicativas no cuentan.



Gran O

- Si $T(n)$ es un polinomio de grado g , $T(n) = O(n^g)$: se puede “olvidar” los términos de grado inferior y las constantes.
- Usar **la clase de funciones más chica**: $2n$ es $O(n)$ (no se dice, aunque es cierto, que es $O(n^2)$).
- Usar la **expresión mas simple**: no se dice que $2n$ es $\frac{1}{2}n$.



Gran O

Propiedad: suma y producto.

Si $T_1(n) = O(U_1(n))$ y $T_2(n) = O(U_2(n))$, entonces

$$T(n) = T_1(n) + T_2(n) = O(\max(U_1(n), U_2(n))).$$

Si $T_1(n) = O(U_1(n))$ y $T_2(n) = O(U_2(n))$, entonces

$$T(n) = T_1(n) T_2(n) = O(U_1(n) U_2(n)).$$



Gran O

Propiedad: suma y producto.

Si $T_1(n) = O(U_1(n))$ y $T_2(n) = O(U_2(n))$, entonces

$$T(n) = T_1(n) + T_2(n) = O(\max(U_1(n), U_2(n))).$$

Si $T_1(n) = O(U_1(n))$ y $T_2(n) = O(U_2(n))$, entonces

$$T(n) = T_1(n) T_2(n) = O(U_1(n) U_2(n)).$$

Prueba:

$$\exists (c_1 > 0, n_0^1) / \forall n \geq n_0^1, T_1(n) \leq c_1 U_1(n)$$

$$\exists (c_2 > 0, n_0^2) / \forall n \geq n_0^2, T_2(n) \leq c_2 U_2(n)$$

entonces :

$$\forall n \geq \max(n_0^1, n_0^2), T(n) \leq (c_1 + c_2) \max(U_1(n), U_2(n)) \quad \square$$

$$\forall n \geq \max(n_0^1, n_0^2), T(n) \leq c_1 c_2 U_1(n) U_2(n) \quad \square$$



Gran O

Propiedad: transitividad.

Si $T(n) = O(U(n))$, y $U(n) = O(W(n))$, entonces
 $T(n) = O(W(n))$.



Gran O

Propiedad: transitividad.

Si $T(n) = O(U(n))$, y $U(n) = O(W(n))$, entonces
 $T(n) = O(W(n))$.

Prueba:

$$\exists (c_1 > 0, n_0^1) / \forall n \geq n_0^1, T(n) \leq c_1 U(n),$$

$$\exists (c_2 > 0, n_0^2) / \forall n \geq n_0^2, U(n) \leq c_2 V(n),$$

$$\text{entonces } \forall n \geq \max(n_0^1, n_0^2), T(n) \leq c_1 c_2 V(n) \quad \square$$



Gran O

Propiedad: base de logaritmo y O.

$\log_a n = O(\log_b n)$ para cualesquiera $a, b > 1$. Eso significa que asintóticamente, **no importa la base del logaritmo**, de la misma manera que no importa un factor multiplicativo en una potencia constante de n . Se podrá usar base 2, e o 10.



Gran O

Propiedad: base de logaritmo y O.

$\log_a n = O(\log_b n)$ para cualesquiera $a, b > 1$. Eso significa que asintóticamente, **no importa la base del logaritmo**, de la misma manera que no importa un factor multiplicativo en una potencia constante de n . Se podrá usar base 2, e o 10.

Prueba:

$$\log_a n = \frac{1}{\log_b a} \log_b n.$$



Gran omega

Se dice que $T(n) = \Omega(U(n))$ ssi existe un real $c > 0$ y un entero n_0 tal que $\forall n \geq n_0, T(n) \geq cU(n)$.

Establece una cuota **inferior** a $T(n)$. Es equivalente a $U(n) = O(T(n))$: si existe n_0 y $c > 0$ tal que $T(n) \geq cU(n)$, $U(n) \leq \frac{1}{c} T(n)$.

Las **propiedades de O valen para Ω** (transitividad, polinomios, producto), y el max se hace min en la propiedad relativa a la suma.



Gran theta

Se dice que $T(n) = \Theta(U(n))$ ssi $T(n) = O(U(n))$ y $T(n) = \Omega(U(n))$.

Establece la equivalencia de comportamiento de dos funciones para grandes n . Lo que nos interesará particularmente es la cuota superior.



Mas propiedades

- Se puede escribir que $T(n) = O(U(n))$ si $\lim_{n \rightarrow \infty} \frac{T(n)}{U(n)}$ es una constante positiva.

Prueba: si el limite es c , existe n_0 tal que $|\frac{T(n)}{U(n)} - c| < \frac{1}{2}$, lo que implica $\frac{T(n)}{U(n)} < c + \frac{1}{2}$ para $n \geq n_0$,

$$T(n) < (\frac{1}{2} + c)U(n) \quad \square$$

- Se puede escribir que $T(n) = \Theta(U(n))$ si $\lim_{n \rightarrow \infty} \frac{T(n)}{U(n)}$ es constante positiva no nula.

Prueba: igual, pero implica $\frac{T(n)}{U(n)} \leq \frac{3c}{2}$, y $\frac{T(n)}{U(n)} \geq \frac{c}{2} \quad \square$



Mas propiedades

Dominaciones:

- $n^k = O((1 + \epsilon)^n)$ para todo $k > 0$ y todo $\epsilon > 0$: un polinomio es dominado por una exponencial, incluso con una potencia arbitrariamente pequeña.
- $(\log n)^k = O(n^\epsilon)$ para todo $k > 0$ y todo $\epsilon > 0$: un logaritmo es dominado por una potencia, incluso de factores arbitrariamente pequeños.



Outline

1 Complejidad y notación asintótica

2 Análisis de complejidad



Análisis: el por qué

Antes de optimizar cualquiera cosa en su programa **técnicamente**, es importante determinar cual es la complejidad del algoritmo subyacente!

- elecciones algorítmicas **cambian el comportamiento asintótico** de la función de complejidad: n , $n \log n$, $n^2 \dots$
- optimizaciones en el código **“sólo” cambian constantes o factores multiplicativos** a este comportamiento asintótico: $\frac{n}{2}$, $4n \dots$



Limites de la análisis

A veces se tiene que **relativizar las relaciones de dominación a través del O**: si tengo dos algoritmos para una funcionalidad dada y que

$$\begin{aligned}T(n) &= 1000000n \\ U(n) &= n^{\frac{12}{10}},\end{aligned}$$

aunque $T(n)$ está dominado por $U(n)$, el comportamiento de U puede ser preferible en la gran mayoría de los casos!



Ejemplo 1: ordenamiento por selección

Ordenar un arreglo de enteros, por selección: **se cambia el valor examinado por el mínimo de los que siguen**

index 0 1 2 3 4 5 6 accion

```

      | 4 0 3 6 1 8 5
it=0 | 0 4 3 6 1 8 5 swap 0,4
it=1 | 0 1 3 6 4 8 5 swap 1,4
it=2 | 0 1 3 6 4 8 5 -
it=3 | 0 1 3 4 6 8 5 swap 4,6
it=4 | 0 1 3 4 5 8 6 swap 5,6
it=5 | 0 1 3 4 5 6 8 swap 6,8

```

Complejidad?



Ejemplo 1: ordenamiento por selección

```
// Search for min between start and end
int minIndex (int *tab, int start, int end) {
    int imin = start;
    for (int i=start+1; i<end; i++)
        if (tab[i] < tab[imin])
            imin = i;

    return imin;
}

// Swap i and j
void swap (int *tab, int i, int j) {
    int t    = tab[i];
    tab[i]   = tab[j];
    tab[j]   = t;
}
```



Ejemplo 1: ordenamiento por selección

La función principal:

```
void selectSort(int *tab, int nels) {  
    for (int i=0; i<nels-1; i++)  
        swap(tab, i, minIndex (tab, i, nels));  
}
```

En cada elemento del recorrido, se cambia el valor por el valor mínimo encontrado al recorrer el final del arreglo.



Ejemplo 1: ordenamiento por selección

Complejidad $T(n)$:

$$T(n) = \sum_{i=0}^{n-2} (T_{\text{swap}} + T_{\text{minIndex}(tab,i,n)}),$$

- swap se hace en **tiempo constante**, con 4 accesos al arreglo, 3 asignaciones (7).
- minIndex **depende del rango de los índices**.



Ejemplo 1: ordenamiento por selección

El método *minIndex*:

- términos constantes: 3 (2 asignaciones y una adición),
- términos hechos ($end - start$) veces: la comparación,
- términos hechos ($end - start - 1$) veces: una incrementación, dos accesos a arreglo y una comparación,
- queda el término hecho en caso de que la condición es true, 1 asignación. En el caso **peor**, estara true en todos casos.

En el peor de los casos:

$$\begin{aligned}T_{minIndex(tab,i,n)} &= 3 + (n - i) + 5(n - i - 1) \\ &= 6n - 6i - 2\end{aligned}$$



Ejemplo 1: ordenamiento por seleccion

En total:

$$\begin{aligned}
 T(n) &= \sum_{i=0}^{n-2} (T_{\text{swap}} + T_{\text{minIndex}(tab,i,n)}) \\
 &= 7(n-2) + \sum_{i=0}^{n-2} (6n - 6i - 2) \\
 &= 7(n-2) + (n-1)6n - 2(n-1) - 6 \sum_{i=0}^{n-2} i \\
 &= 6n^2 - n - 12 - 3(n-2)(n-1) \\
 &= 3n^2 - 10n - 18
 \end{aligned}$$

Entonces el **algoritmo es cuadrático en el peor caso**:

$T(n) = O(n^2)$, no es muy eficiente para grandes valores de n . ¿En el mejor?



Ejemplo 2: mergeSort

Otro algoritmo famoso para ordenamiento: el mergeSort.

Adopta el *divide-and-conquer* para el ordenamiento en sitio:

```
void mergeSort(int *tab,int start,int end,int pivot){  
    if(start == end) return;  
    else {  
        // Primera parte  
        mergeSort(tab, start, pivot, (start+pivot)/2);  
        // Segunda  
        mergeSort(tab, pivot+1, end, (pivot+end+1)/2);  
    }  
    // Combinar los arreglos  
    merge(tab, start, end, pivot);  
}
```



Ejemplo 2: mergeSort

```
void merge(int *tab, int start, int end, int pivot) {  
    int tmptab[end-start+1];  
    int i=start, j=pivot+1, k=0;  
    for (; j<end && i<pivot+1;) {  
        if (tab[j]<=tab[i])  
            tmptab[k]=tab[j++];  
        else  
            tmptab[k]=tab[i++];  
        k++;  
    }  
    for (; j<end; j++) tmptab[k++]=tab[j++];  
    for (; i<pivot+1; j++) tmptab[k++]=tab[i++];  
    for (i=start, k=0; i<end;) tab[i++]=tmptab[k++];  
}
```



Ejemplo 2: mergeSort

- En el primer ciclo, una vez cada de los elementos de tab , excepto los que vienen en el ciclo final.
- En total, $T_{merge}(n) = O(n)$ donde $n = end - start + 1$.
- Luego, observar que el algoritmo es recursivo, entonces, **en cada nivel, tenemos dos instancias del problema original, pero con tamaño de dato dividido por dos**
- En el caso que $n = 1$, salimos directamente después de la comparación: $T(1) = 1$



Ejemplo 2: mergeSort

En consecuencia, se puede escribir:

$$\begin{cases} T(1) &= 1 \\ T(n) &= 2T(\frac{n}{2}) + O(n) \\ &\leq 2T(\frac{n}{2}) + cn \end{cases}$$

expresando la complejidad de manera recursiva.

Solución?



Ejemplo 2: mergeSort

Poner $n = 2^i$ y $U(i) = T(n)$, la relación se hace:

$$\begin{aligned}
 U(0) &= 1 \\
 U(i) &= 2U(i-1) + c2^i \\
 &= 4U(i-2) + c(2^i + 2 \cdot 2^{i-1}) \\
 &= 8U(i-3) + c(2^i + 2 \cdot 2^{i-1} + 2^2 2^{i-2}) \\
 &\dots \\
 &= 2^i U(0) + ci2^i.
 \end{aligned}$$

Entonces

$$T(n) = n(1 + c \log n) = O(n \log n).$$

El algoritmo de **mergeSort** es en $n \log n$.



Ejemplo 3: búsqueda por dicotomía

```
bool search(int *tab, int val, int start, int end, int &ind) {  
    if (end==start) {  
        if (tab[start]==val) {  
            ind=start;  
            return true;  
        } else return false;  
    }  
    int pivot=(start+end)/2;  
    if (val<tab[pivot])  
        return search(tab, val, start, pivot, ind);  
    else if (val>tab[pivot])  
        return search(tab, val, pivot+1, end, ind);  
    else {  
        ind=pivot;  
        return true;  
    }  
}
```



Ejemplo 3: búsqueda por dicotomía

En un arreglo ordenado, la complejidad de una búsqueda es de orden $O(\log n)$: todos los datos pueden estar vistos como un **árbol binario** conteniendo todos los enteros. El algoritmo solo busca un camino de la raíz hasta una hoja (y uno solo).

$$T(n) = O(\log n).$$



Complejidades usuales

- $O(1)$: complejidad **constante**. Occure cuando instrucciones son ejecutadas una y una vez solamente, independientemente del tamaño del problema
- $O(\log n)$: complejidad **logarítmica**, que crece ligeramente con n . Problemas de dicotomía, típicamente, donde el problema es compartido en varias instancias y solo una esta procesada.
- $O(n)$: complejidad **lineal**. Problemas con ciclos sobre los datos con procesamiento de duración constante. . .
- $O(n \log(n))$: complejidad **n-logarítmica**. Problemas donde en cada iteración se divide en sub-problemas, donde el procesamiento es lineal (mergeSort, quickSort)



Complejidades usuales

- $O(n^2)$: complejidad **cuadrática**. Caso de todos los algoritmos que implica dos ciclos imbricados, donde el ciclo interno se hace sobre todos los datos o aun sobre un número de datos lineal en el índice del primer ciclo
- $O(n^3)$: complejidad **cúbica**, implica generalmente tres ciclos imbricados
- $O(2^n)$: complejidad **exponencial**, eso corresponde a algoritmos ingenuos, que a partir de ciertos tamaños no muy altos son completamente inutilizables



Complejidad del problema

Hasta ahora solo hemos analizado la complejidad de soluciones a unos problemas; para varios problemas (como el ordenamiento), hay varios tipos de soluciones de complejidad diferente, pero ninguno puede llegar abajo de cierto nivel: se habla de **complejidad intrínseca al problema** (la **complejidad del mejor algoritmo que puede solvarlo**).



Complejidad del problema

Unas nociones importantes:

- problemas: decisión / optimización/ ...,
- maquinas deterministas (Turing): hacen una instrucción a la vez, corresponden al modelo teórico clásico para las maquinas actuales,
- maquinas no-deterministas: maquinas teóricas, que tendrían la posibilidad, frente a una situación en que hay que explorar varias posibilidades, de **explorarlas “a la vez”**.



Complejidad del problema

Clases de problemas:

- L: problemas que pueden estar resueltos en tiempo logarítmico.
- P: problemas que pueden estar resueltos en tiempo polinomial (problemas “fáciles”).
- NL: problemas que pueden estar resueltos por un algoritmo en una maquina no determinista en tiempo logarítmico.
- NP: problemas que pueden estar resueltos por un algoritmo en una maquina no determinista en tiempo polinomial.



Complejidad del problema: problemas NP

- Problemas **combinatorios**.
- Implican una **enumeración de posibilidades que hay que examinar**.
- Son equivalente a problemas de decisión que, si se da una solución candidata, se puede contestar en tiempo polinomial que si o no.
- NP-difícil: los mas difíciles, aun, todo problema NP se puede reducir polinomialmente en uno de esos.



Complejidad del problema: problemas NP-difícil

