

Análise da Complexidade de Algoritmos II

25/09/2023

Sumário

- Recap
- Procurar o maior elemento de um array não-ordenado
- Análise do Melhor Caso, do Pior Caso e do Caso Médio
- Ordens de complexidade
- Notações habituais
- Exercícios adicionais
- Sugestões de leitura

Let's
RECAP

Recapitulação

Algoritmos

- Algoritmos deterministas **vs** Algoritmos não-deterministas
- Análise do desempenho / eficiência computacional
 - Complexidade Temporal **vs** Complexidade Espacial
- Análise experimental **vs** Análise formal
- Classes de complexidade – Quais ?
- Eficiência relativa

Análise da Complexidade – Para quê ?

- **Vários algoritmos** para resolver uma **instância** de um problema
 - Diferentes classes / ordens de complexidade
- Qual é o algoritmo mais eficiente / com melhor desempenho ?
- **Um algoritmo** para resolver **várias instâncias** de um problema
 - Dimensão sucessivamente maior
 - Configurações diferentes para a mesma dimensão
- Como estimar o desempenho / o tempo de execução ?

Exemplo

```
de i = 0 até 256:  
    contador[i] = 0;  
enquanto não fim de ficheiro:  
    ler próximo carater;  
    incrementar contador[próximo carater];
```

- Inicialização : 256 incrementos da **variável i**
- Inicialização : 256 atribuições ao array
- Leitura do ficheiro : $(n + 1)$ comparações para detetar o fim do ficheiro
- Leitura do ficheiro : n incrementos de elementos do array
- Qual é o factor que determina o **desempenho** ?
- O esforço da fase de **inicialização** é importante ?

Ciclos – Quantas iterações?

```
for(k=1; k<6; k++) {
```

```
    ...
```

```
}
```

$$\sum_{k=1}^5 1 = 5$$

```
for(i=0; i<m; i++) {
```

```
    for(j=0; j<n; j++) {
```

```
        ...
```

```
    }
```

```
}
```

$$\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} 1 = \sum_{i=0}^{m-1} n = m \times n$$

Multiplicação de matrizes quadradas

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

- Quantos elementos tem a matriz resultado ? $m \cdot p$
- Quantas multiplicações se efetuam para calcular cada elemento ?
- Qual é o número total de multiplicações ?

Multiplicação de matrizes quadradas

```
for(int i=0; i<n; i++) {  
    for(int j=0; j<n; j++) {  
        c[i][j] = 0;  
        for(int k=0; k<n; k++) {  
            c[i][j] += a[i][k] * b[k][j];  
        }  
    }  
}
```



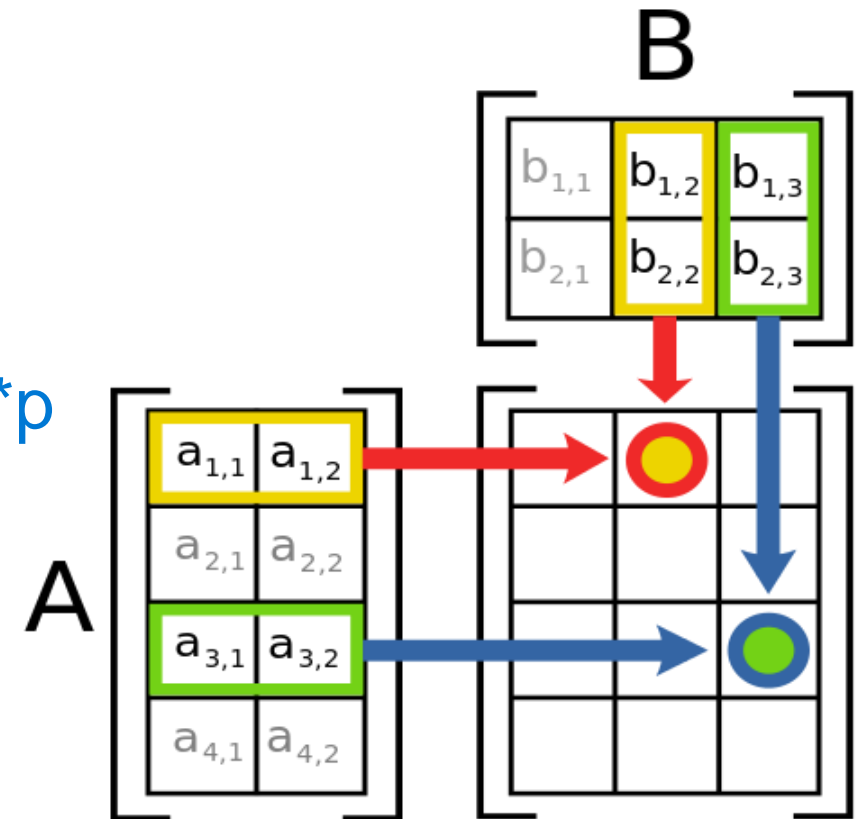
$$\sum_{i=0}^{n-1} \left[\sum_{j=0}^{n-1} \left(\sum_{k=0}^{n-1} 1 \right) \right]$$

Algoritmo **cúbico**

Multiplicação de matrizes – Caso geral

$$A(m \times n) \times B(n \times p) = C(m \times p)$$

- Quantas **multiplicações** são efetuadas ? $m \times n \times p$
- Ordem de complexidade ?
- Modificar o código anterior



[Wikimedia.org]

Fizeram ?

- Expressão para o nº de vezes que a **instrução mais interna** é executada
- Expressão para o **resultado** de cada uma das funções

```
int f3(int n) {  
    int i,j,r=0;  
    for(i = 1; i <= n; i++)  
        for(j = i; j <= n; j++)  
            r += 1;  
    return r;  
}
```

```
int f4(int n) {  
    int i,j,r=0;  
    for(i = 1; i <= n; i++)  
        for(j = 1; j <= i; j++)  
            r += j;  
    return r;  
}
```

```
int f1(int n) {  
    int i,r=0;  
    for(i = 1; i <= n; i++)  
        r += i;  
    return r;  
}
```

```
int f2(int n) {  
    int i,j,r=0;  
    for(i = 1; i <= n; i++)  
        for(j = 1; j <= n; j++)  
            r += 1;  
    return r;  
}
```

n^2

Alguns resultados úteis

- $\sum_{k=1}^n 1 = n$
- $\sum_{k=1}^n k = \frac{n(n+1)}{2}$
- $\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6}$

- $\sum_{k=1}^n k^3 = \left(\frac{n(n+1)}{2} \right)^2$
- $\sum_{k=1}^n \frac{1}{k} = \log n + \underbrace{\gamma}_{\substack{\text{Euler's constant} \\ \approx 0.577216}} + \frac{1}{2n} + O(n^{-2})$
- $\sum_{k=n}^m f(k) = \sum_{k=1}^m f(k) - \sum_{k=1}^{n-1} f(k)$

Procura do Maior Elemento de um Array Não Ordenado

Procura do maior elemento

```
int searchMax( int a[], int n ) {  
    int indexMax = 0;  
    for( int i=1; i<n; i++ ) {  
        if( a[i] > a[indexMax] ) {  
            indexMax = i;  
        }  
    }  
    return indexMax;  
}
```

Nº de Atribuições -> depende da org. do array. Varia entre 1 (melhor caso) e $(n-1)+1$ (pior caso)



Procura do maior elemento

- Quantas **comparações** ?
- $N_{comp}(n) = n - 1$
- Número **fixo** de comparações $n-1 \rightarrow$ **Função Linear**
- Algoritmo **linear** no número de comparações efetuadas

Procura do maior elemento

Nº de Atribuições -> depende da org. do array. Varia entre 1 (melhor caso) e $(n-1)+1$ (pior caso)

- Quantas atribuições à variável indexMax ?
- Número de atribuições depende da localização da 1ª ocorrência do maior elemento !!
- Melhor caso : 1 atribuição – Quando ?
- Pior caso : n atribuições – Quando ?
- Caso médio ? -> Simplificação : Equiprobabilidade – Verosímil ?
$$(1 + 2 + 3 + \dots + n) / n = (n + 1) / 2$$

Tarefa 1

- **Variações** do algoritmo anterior :
- Encontrar a **última ocorrência** do **maior** elemento
- Encontrar a **primeira ocorrência** do **menor** elemento
- Encontrar a **última ocorrência** do **menor** elemento
- Como modificar o código ?
- O que se mantém da análise anterior ?
- O que muda da análise anterior ?

Melhor Caso, Pior Caso e Caso Médio

Best case, Worst case

- D_n = conjunto de instâncias de dimensão n
- I é uma instância de D_n
- $t(I)$ = tempo de execução ou nº de operações para a instância I

$$B(n) = \min_{I \in D_n} t(I)$$

$$W(n) = \max_{I \in D_n} t(I)$$

Average case

- D_n = conjunto de instâncias de dimensão n
- I é uma instância de D_n
- $p(I)$ = **probabilidade** de ocorrência da instância I
- $t(I)$ = tempo de execução **ou** nº de operações para a instância I

$$A(n) = \sum_{I \in D_n} p(I) \times t(I)$$

Procura sequencial num array

- Dado um **array não ordenado** com n elementos
- Procurar um dado **valor x**
- Devolver o **índice da sua primeira ocorrência**, se pertencer ao array

Procura sequencial num array

```
int search( int a[], int n, int x ) {  
    for( int i=0; i<n; i++ ) {  
        if( a[i] == x ) {  
            return i;  
        }  
    }  
    return -1;  
}
```



Comparações ?

- $B(n) = 1$ - Quando ?
- $W(n) = n$ - Quando ?
- $A(n) = ?$
- **Simplificação** : o elemento procurado pertence ao array
- **Simplificação** : equiprobabilidade $\rightarrow p(x=a[i]) = 1/n$

$$A(n) = 1/n \times (1 + 2 + \dots + n) = (n + 1) / 2 \approx n / 2$$

Ordens de Complexidade

Ordem de Complexidade

- Classificar a **eficiência** de um algoritmo para dados de **grande dimensão**
- Qual é a rapidez com que **cresce** o **tempo de execução** (i.e., o nº de operações) , quando a dimensão dos dados se torna (muito) **maior** ?
- O que acontece se a dimensão dos dados é
 - **o dobro** ?
 - **dez vezes maior** ?
 - ...
- Como representar essa taxa / rapidez ?

Ordens de Complexidade

- Valores aproximados para algumas funções habituais

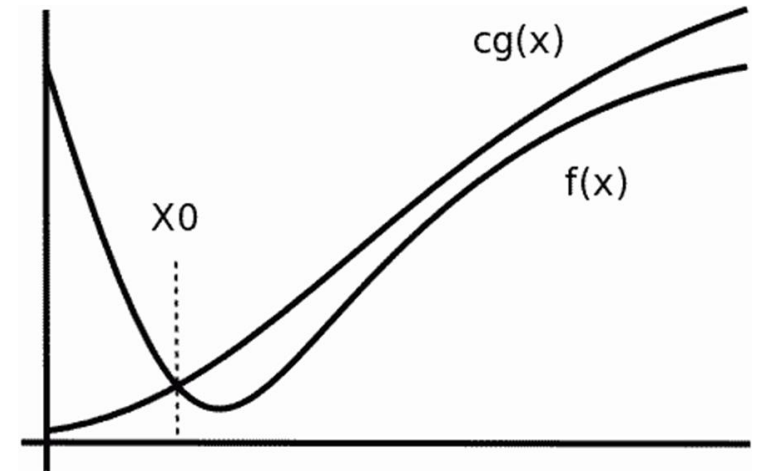
n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10	3.3×10^1	10^2	10^3	10^3	3.6×10^6
10^2	6.6	10^2	6.6×10^2	10^4	10^6	1.3×10^{30}	9.3×10^{157}
10^3	10	10^3	10^4	10^6	10^9	?	?
10^4	13	10^4	1.3×10^5	10^8	10^{12}	?	?
10^5	17	10^5	1.7×10^6	10^{10}	10^{15}	?	?
10^6	20	10^6	2.0×10^7	10^{12}	10^{18}	?	?

Notação habitual

- A rapidez com **que cresce o nº de operações** é um indicador da **eficiência** de um algoritmo
- Como comparar / **classificar** algoritmos para um mesmo problema ?
 - Comparando as suas **ordens de complexidade** !!
- Notação habitual : **$O(n)$, $\Omega(n)$, $\Theta(n)$**
Majorante, Minorante e Enquadramento

Big-Oh : $t(n) \in O(g(n))$

- Majorante / Limite superior

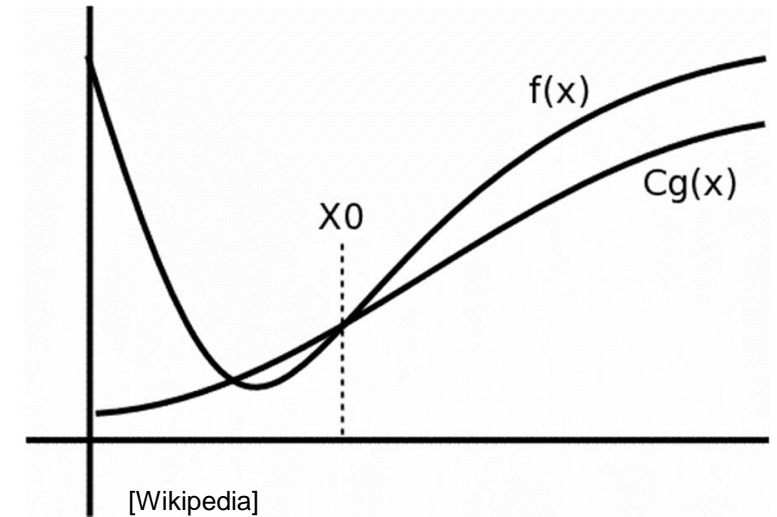


[Wikipedia]

- $O(g(n))$: conjunto de todas as funções com a mesma ordem de crescimento que $g(n)$ ou com uma ordem de crescimento inferior
- $t(n) \leq c g(n)$, para todo o $n \geq n_0$, c é uma constante positiva
- $t(n), g(n)$: funções não negativas sobre o conjunto dos números naturais


Big-Omega : $t(n) \in \Omega(g(n))$

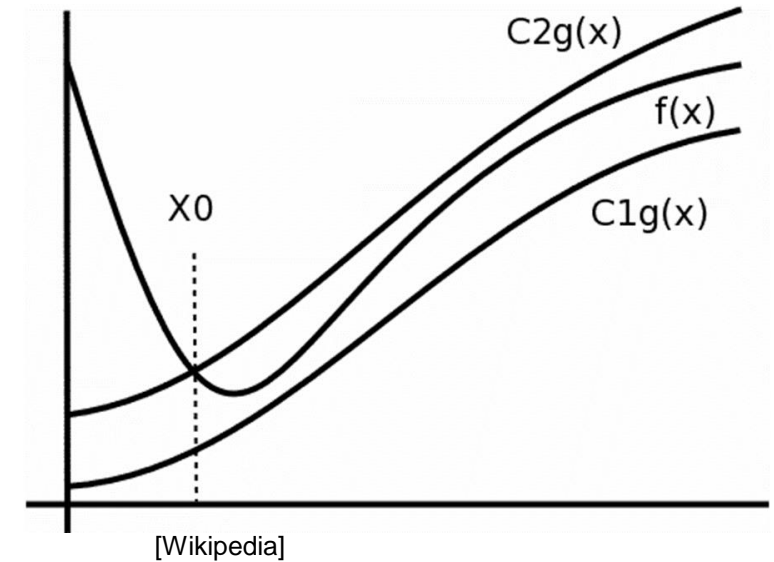
- Minorante / Limite inferior



- $\Omega(g(n))$: conjunto de todas as funções com **a mesma ordem de crescimento** que $g(n)$ **ou** com uma ordem de crescimento **superior**
- $t(n) \geq c g(n)$, para todo o $n \geq n_0$, c é uma **constante positiva**

Big-Theta : $t(n) \in \Theta(g(n))$

- Enquadramento
- $\Theta(g(n))$: conjunto de todas as funções com a **mesma ordem de crescimento** que $g(n)$
- $c_1 g(n) \leq t(n) \leq c_2 g(n)$, para todo o $n \geq n_0$, c_1, c_2 **constantes positivas**
- $t(n) \in O(g(n))$ e $t(n) \in \Omega(g(n))$ 



Notação – Exemplo

- Ocultar **detalhes que não são importantes** quanto ao modo como uma função cresce
 - Esquecer **constantes** e **termos de ordem inferior**
- $T_1(n) = 2n^2 + 3000n + 5$
- $T_2(n) = 10n^2 + 100n - 23$
- Para **valores elevados** de n , $T_2(n)$ cresce **mais depressa** do que $T_1(n)$
- MAS, ambas crescem de modo quadrático : $\Theta(n^2)$

Exemplo – Importância do termo de maior grau

- $f(n) = a n^2 + b n + c$, com $a = 0.0001724$, $b = 0.0004$ e $c = 0.1$

n	f(n)	$a n^2$	$a n^2 / f(n)$
125	2.8	2.7	94.7%
250	11.0	10.8	98.2%
500	43.4	43.1	99.3%
1000	172.9	172.4	99.7%

Tarefa 2 – Completar com \in ou \notin

- $T(n) = 10n^2 + 100n - 23$

$$T(n) \text{ ? } O(n^2)$$

$$T(n) \text{ ? } O(n^3)$$

$$T(n) \text{ ? } O(n)$$

$$T(n) \text{ ? } \Omega(n^2)$$

$$T(n) \text{ ? } \Omega(n^3)$$

$$T(n) \text{ ? } \Omega(n)$$

$$T(n) \text{ ? } \Theta(n^2)$$

$$T(n) \text{ ? } \Theta(n^3)$$

$$T(n) \text{ ? } \Theta(n)$$

Notação – Exemplos

notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ \vdots	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ \vdots	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$ \vdots	develop lower bounds

[Sedgewick & Wayne]

Ordens de Complexidade/Classes de Eficiência

- $O(1)$: constante
 - Que algoritmos ?
- $O(\log n)$: logarítmico
 - E.g., **diminuir-para-reinar**
- $O(n)$: linear
 - Processar todos os elementos de um array, uma lista, etc.
- $O(n \log n)$: n-log-n
 - E.g., **dividir-para-reinar**

Ordens de Complexidade/Classes de Eficiência

- $O(n^k)$: polinomial (quadrático, cúbico, etc.)
 - k ciclos encastelados
- $O(2^n)$: exponencial
 - Gerar **todos os subconjuntos** de um conjunto com n elementos
- $O(n!)$: fatorial
 - Gerar **todas as permutações** de um conjunto com n elementos

Ordens de Complexidade/Classes de Eficiência

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$
1	constant	<code>a = b + c;</code>	statement	add two numbers	1
$\log N$	logarithmic	<pre>while (N > 1) { N = N / 2; ... }</pre>	divide in half	binary search	~ 1
N	linear	<pre>for (int i = 0; i < N; i++) { ... }</pre>	loop	find the maximum	2
$N \log N$	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	quadratic	<pre>for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }</pre>	double loop	check all pairs	4
N^3	cubic	<pre>for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }</pre>	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$



[Sedgewick & Wayne]

Sugestões de leitura

Sugestões de leitura

- J. J. McConnell, Analysis of Algorithms, 1st Edition, 2001
 - Capítulo 1: secções 1.1, 1.2, 1.4
- A. Levitin, Introduction to the Design and Analysis of Algorithms, 3rd Edition, 2012
 - Capítulo 2: secções 2.1, 2.2, 2.3