Análise da Complexidade de Algoritmos II

30/09/2024

Sumário

- Recap
- Procurar o maior elemento de um array não-ordenado
- Análise do Melhor Caso, do Pior Caso e do Caso Médio
- Linear Search – Procura sequencial de um elemento num array
- Ordens de complexidade
- Notações habituais
- Exercícios / Tarefas
- Sugestões de leitura

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

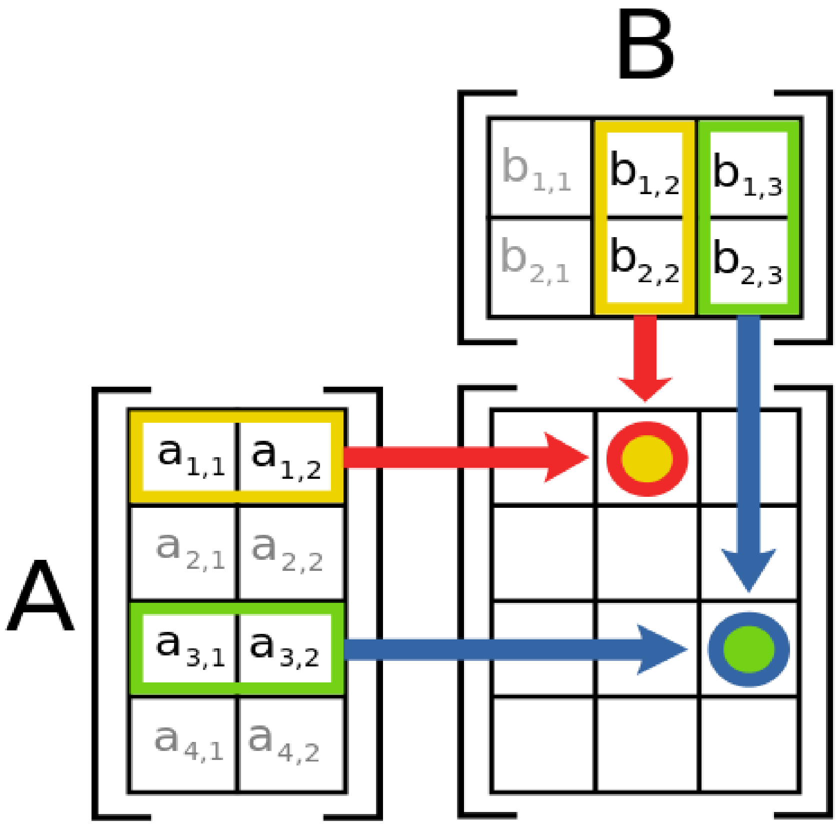
- Qual é o factor que determina o desempenho ?
- O esforço da fase de inicialização é importante ?

Multiplicação de matrizes – Caso geral

A(m x n) x B(n x p) = C(m x p)

- Quantas multiplicações são efetuadas ?
- Ordem de complexidade ?

- Modificaram o código da aula anterior ?



[Wikimedia.org]

Procura do Maior Elemento de um Array Não Ordenado

Procura da 1ª ocorrência 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;
}
```

Procura da 1ª ocorrência do maior elemento

- Quantas comparações ?
- **Ncomp(n) = n – 1**
- Número fixo de comparações
- Algoritmo linear no número de comparações efetuadas

Procura da 1ª ocorrência do maior elemento

- 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 **?**

$$\mathbf{(1 + 2 + 3 + \ldots + n) / n = (n + 1) / 2}$$

Melhor Caso, Pior Caso e Caso Médio

Best case, Worst case

- Dn = conjunto de instâncias de dimensão n
- I é uma instância de Dn

- t(I) = tempo de execução ou nº de operações para a instância I

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

Average case

- Dn = conjunto de instâncias de dimensão n
- I é uma instância de Dn

- 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 Sequential – Array Não Ordenado

Procura sequencial de um valor num array

- Dado um array não ordenado com n elementos
- Procurar um dado valor x
- Devolver o índice da sua 1ª ocorrência, se pertencer ao array

Procura sequencial de um valor 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 -> $p(x==a[i]) = 1/n$

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

Ordens de Complexidade

Ordem de Complexidade

- Classificar a eficiência de um algoritmo para instâncias 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}	?	?

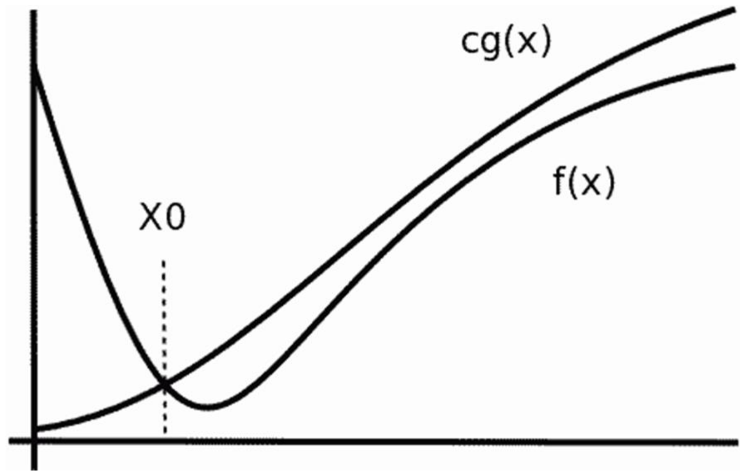
Notação habitual

- 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 : **$O(n)$, $\Omega(n)$, $\Theta(n)$**

Big-Oh : **t(n) ∈ O(g(n))**



[Wikipedia]

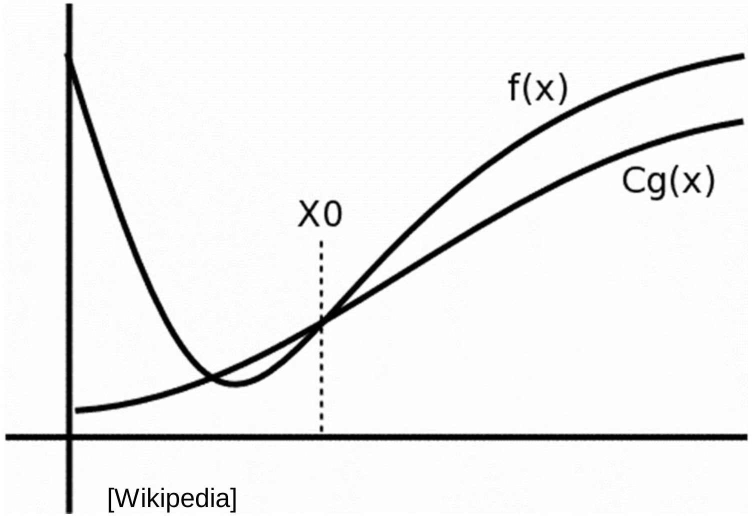
- Majorante / Limite superior

- **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) ≤ 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 : $t(n) \in \Omega(g(n))$

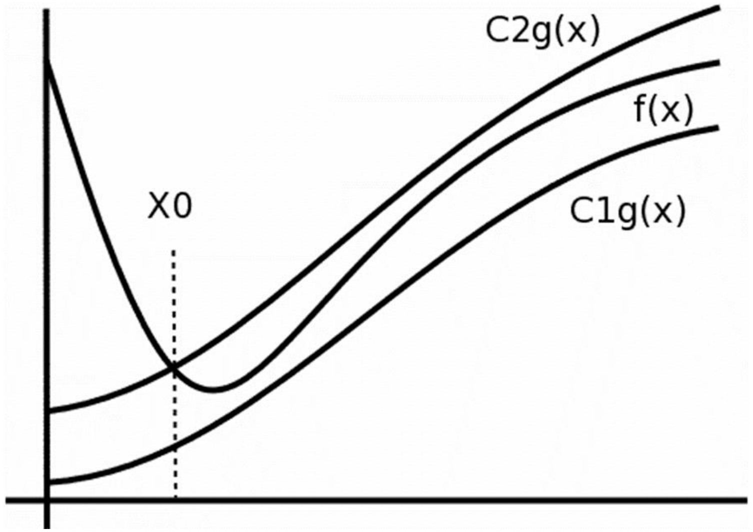


- Minorante / Limite inferior

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



[Wikipedia]

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

Notação – Exemplo

- A notação oculta detalhes que não são importantes quanto ao modo como uma função cresce
  - Esquecer constantes e termos de ordem inferior

- $T_1(n) = 2\ n^2 + 3000\ n + 5$
- $T_2(n) = 10\ n^2 + 100\ n - 23$
- 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%

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

[Sedgewick & Wayne]

Exercícios / Tarefas

Exercício 1 – Completar com $\in$ ou $\notin$

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

$T(n)$? $O(n^2)$	$T(n)$? $O(n^3)$	$T(n)$? $O(n)$
$T(n)$? $\Omega(n^2)$	$T(n)$? $\Omega(n^3)$	$T(n)$? $\Omega(n)$
$T(n)$? $\Theta(n^2)$	$T(n)$? $\Theta(n^3)$	$T(n)$? $\Theta(n)$

Exercício 2 – Verdadeiro ou Falso

- Se $\boldsymbol{t(n) \geq cg(n)}$, para todo o $n > n_0$, então $t(n)$ é da ordem de $\boldsymbol{O(g(n))}$, i.e., $t(n)$ pertence a $\boldsymbol{O(g(n))}$.

- Se $\boldsymbol{t(n) \leq cg(n)}$, para todo o $n > n_0$, então $t(n)$ é da ordem de $\boldsymbol{\Omega(g(n))}$, i.e., $t(n)$ pertence a $\boldsymbol{\Omega(g(n))}$.

- Se $\boldsymbol{t(n) > cg(n)}$, para todo o $n > n_0$, então $t(n)$ é da ordem de $\boldsymbol{\Omega(g(n))}$, i.e., $t(n)$ pertence a $\boldsymbol{\Omega(g(n))}$.

Exercício 3 – Verdadeiro ou Falso

- Seja $t(n) = 2\,n\log(n) + 3\,n$. Então $t(n)$ é da ordem de $O(n^2)$, i.e., $t(n)$ pertence a $O(n^2)$.

- Seja $t(n) = n\log(n) + 1000\,n$. Então $t(n)$ é da ordem de $O(n)$, i.e., $t(n)$ pertence a $O(n)$.

- Seja $t(n) = 1000\,n^2 + n^3$. Então $t(n)$ é da ordem de $O(n^2)$, i.e., $t(n)$ pertence a $O(n^2)$.

Tarefa – Procura Sequencial num Array

- Variações do algoritmo básico :

- 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 ?

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