Algoritmia Básica: Algoritmos, Complexidade e Correcção

1ª edição: Março/2002

RICARDO J. MACHADO

Email: rmac@dsi.uminho.pt
URL: http://www.dsi.uminho.pt/~rmac

Universidade do Minho

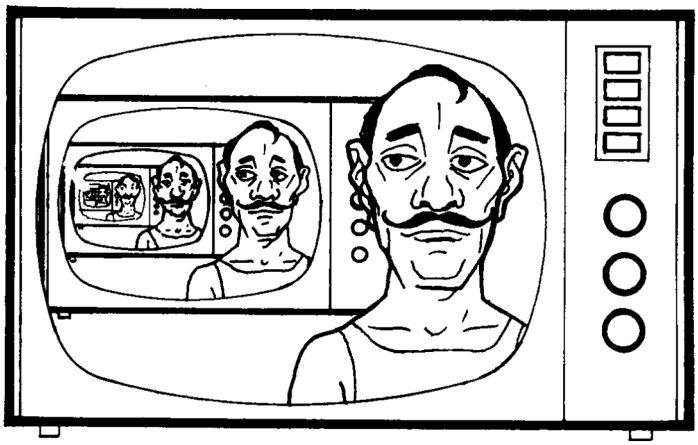
Departamento de Sistemas de Informação

Sumário

1. Algoritmos Recursivos (1/14)

- contexto -

- **Recursividade**



3

1. Algoritmos Recursivos (2/14)

- contexto -

- **Definição recursiva**
  - **uma definição na qual a entidade que está ser definida surge como parte da definição é chamada *definição indutiva* ou *definição recursiva***
  - **uma definição recursiva é composta, tipicamente, por duas partes**
    - **uma base, onde alguns casos simples da entidade que está a ser definida são fornecidos explicitamente, podendo ser utilizados como condições de paragem nos algoritmos recursivos**
    - **um passo indutivo ou recursivo, onde outros casos da entidade que está a ser definida são fornecidos em termos dos casos anteriores**

4

1. Algoritmos Recursivos (3/14)

- contexto -

- **Algoritmo recursivo**
  - **um *algoritmo recursivo* consiste numa procedimentalização directa de uma definição recursiva segundo uma implementação funcional ao nível das interfaces**
  - **quando um problema algorítmico sugere naturalmente uma definição recursiva, a elaboração de uma solução recursiva (algoritmo recursivo) é, tipicamente, uma tarefa trivial**
  - **quando um problema algorítmico não sugere naturalmente uma definição recursiva, a elaboração de uma solução recursiva (algoritmo recursivo) é, tipicamente, uma tarefa muito mais difícil**
  - **chama-se *profundidade* ao número de vezes que um algoritmo recursivo se chama a si próprio até obter o resultado pretendido, ou seja, a profundidade designa os número de *níveis de recursividade***

1. Algoritmos Recursivos (4/14)

- contexto -

- **Exemplo #1**
  - **definição recursiva de uma sequência**

    s [1] = 2
    s [n] = 2 × s [n - 1], para n ≥ 2

  - **algoritmo recursivo de *S [n]***

    S (n: Integer): Integer

    1. se n = 1,
    então devolver S ← 2 e terminar,
    senão devolver S ← 2 × S (n - 1) e terminar

1. Algoritmos Recursivos (5/14)

- contexto -

- **Exemplo #1 (continuação)**
  - **algoritmo iterativo (não recursivo) de *S [n]***

    S (n: Integer): Integer

    1. se n = 1, então devolver S ← 2 e terminar

    2. I ← 2 e value ← 2

    3. se I ≤ n, então value ← 2 × value
    I ← I + 1
    ir para 3

    4. devolver S ← value e terminar

1. Algoritmos Recursivos (6/14)

- contexto -

- **Exemplo #2**
  - **definição recursiva para a multiplicação *m× n***

    m [1] = m
    m [n] = m [n - 1] + m, para n ≥ 2

  - **algoritmo recursivo de *m× n***

    Prod (m, n: Integer): Integer

    1. se n = 1,
    então devolver Prod ← m e terminar,
    senão devolver Prod ← Prod (m, n - 1) + m e terminar

1. Algoritmos Recursivos (7/14)

- cálculo do factorial -

Seja $n \in \mathbb{N}^+$

$$n! = \prod_{i=1}^{n} i$$

factorial (n: Integer): Integer

1. se n = 0,
então devolver factorial ← 1 e terminar,
senão devolver factorial ← n × factorial (n - 1) e terminar

***nota*: *factorial (3)* possui uma profundidade de 3, ou seja, 3 níveis de recursividade**

1. Algoritmos Recursivos (8/14)

- cálculo do MDC (algoritmo de Euclides) -

mdc (m: Integer, n: Integer): Integer

1. se (m mod n) = 0,
então devolver mdc ← n e terminar,
senão devolver mdc ← mdc (n, m mod n) e terminar

1. Algoritmos Recursivos (9/14)

- cálculo dos números de Fibonacci -

- **sequência de Fibonacci**

  0, 1, 1, 2, 3, 5, 8, 13, 21 ...

  fib [1] = 0
  fib [2] = 1
  fib [n] = fib [n - 1] + fib [n - 2], para n > 2

- **algoritmo recursivo de *fib [n]***

  fib (n: Integer): Integer

  1. se n = 1,
  então devolver fib ← 0 e terminar,
  senão se n = 2,
  então devolver fib ← 1 e terminar,
  senão devolver fib ← fib (n - 1) + fib (n - 2) e terminar

1. Algoritmos Recursivos (10/14)

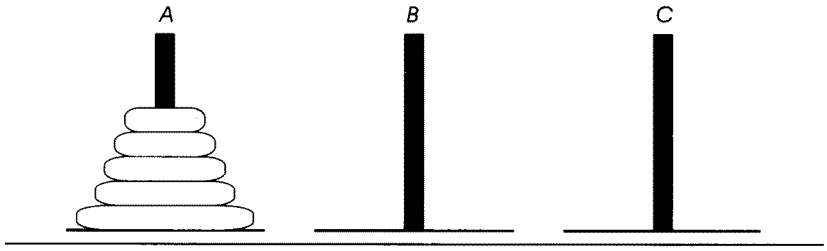
- torres de Hanoi -

- **O problema**
  - **existem três estacas A, B e C**
  - **cinco discos de diferentes diâmetros são encaixados na estaca A, de forma a que um determinado disco nunca se encontra posicionado por cima de um outro de diâmetro menor**
  - **pretende-se mover os cinco discos da estaca A para a estaca C, usando a estaca B como auxiliar, sem violar as seguintes regras**
    - **só é possível movimentar um disco de cada vez**
    - **o disco a deslocar é sempre o que se encontra por cima**
    - **um disco de maior diâmetro não pode ficar posicionado por cima de um de menor diâmetro**

1. Algoritmos Recursivos (11/14)

- torres de Hanoi -

- **Configuração inicial**



1. Algoritmos Recursivos (12/14)

- torres de Hanoi -

- **Estratégia da solução**
  - **considerar o caso geral, não de cinco discos, mas sim de *n* discos**
  - **supor que existe uma solução para *n-1* discos**
  - **desenvolver a solução para *n* discos em função da (suposta) solução para *n-1* discos**

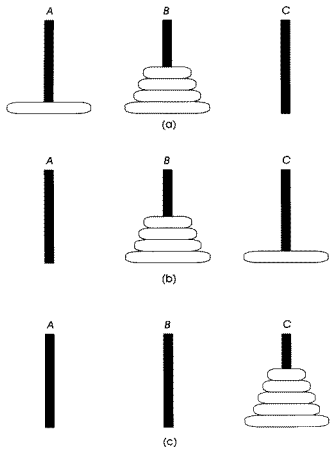
  ***Notas*:**

  - **no caso de *n = 1*, a solução para o problema de mover o disco da estaca A para a C é trivial**
  - **reduzir o problema de *n* discos à solução de *n-1* discos irá decair no problema do caso *n = 1*, pelo que uma abordagem recursiva se apresenta vantajosa para a solução do problema das torres de Hanoi**

1. Algoritmos Recursivos (13/14)

- torres de Hanoi -

- **Estratégia recursiva (versão 5 discos)**



1. Algoritmos Recursivos (14/14)

- torres de Hanoi -

- **Estratégia recursiva (algoritmo não refinado)**

1. se n = 1, então deslocar o único disco de A para C e terminar

2. deslocar os n–1 primeiros discos de A para B, usando C como auxiliar

3. deslocar o último disco de A para C

4. deslocar os n–1 discos de B para C, usando A como auxiliar e terminar

2. Algoritmos de Procura (1/7)

- contexto -

- **Problema de procura**
  - **dada uma colecção K de objectos (elementos), pretende-se saber se um determinado elemento X se encontra presente na colecção**
- **Algoritmos**
  - **procura sequencial (*linear search*)**
  - **procura binária (*binary search*)**
  - **procura binária recursiva (*recursive binary search*)**

2. Algoritmos de Procura (2/7)

- procura sequencial (*linear search*) -

- **Algoritmo *linear search***
  - **a colecção K é um vector (*array*), não necessariamente ordenado, com N + 1 elementos (N ≥ 1)**
  - **o algoritmo percorre sequencialmente o vector à procura de um elemento em particular cujo valor seja igual a X**
  - **o elemento K [N + 1] do vector é utilizado como uma *sentinela* que recebe o valor de X antes de ser iniciada a procura**
  - **se a procura for bem sucedida, o algoritmo devolve o valor do índice do elemento do vector igual a X**
  - **se a procura for mal sucedida, o algoritmo devolve o valor 0**

***Nota*: este é um algoritmo simples e eficaz para colecções pequenas, mas completamente ineficaz para colecções de grande dimensão**

- **mau exemplo: procurar "José Silva" numa lista telefónica regional**

2. Algoritmos de Procura (3/7)

- procura sequencial (*linear search*) -

1. [iniciar procura]

1.1. I ← 1

1.2. K [N + 1] ← X

2. [percorrer o vector]

2.1. se K [I] ≠ X,
então I ← I + 1 e ir para 2,
senão continuar (ir para 3)

3. [decidir sobre o sucesso da procura]

3.1. se I = N + 1,
então indicar "PROCURA SEM SUCESSO", devolver 0 e terminar,
senão indicar "PROCURA COM SUCESSO", devolver I e terminar

2. Algoritmos de Procura (4/7)

- procura binária (*binary search*) -

- **Algoritmo *binary search***
  - **a colecção K é um vector ordenado (alfabética ou numericamente crescente) com N elementos**
  - **o algoritmo percorre o vector à procura de um elemento em particular cujo valor seja igual a X, pesquisando em intervalos cada vez menores através da divisão em duas partes do espaço de procura em cada momento**
  - **as variáveis LOW, MIDDLE e HIGH denotam, respectivamente, os limites (índices do vector) inferior, médio e superior do intervalo de procura**
  - **se a procura for bem sucedida, o algoritmo devolve o valor do índice do elemento do vector igual a X**
  - **se a procura for mal sucedida, o algoritmo devolve o valor 0**

***Nota*: este é um algoritmo que reduz para metade (relativamente à procura sequencial) a complexidade de procura em vectores**

2. Algoritmos de Procura (5/7)

- procura binária (*binary search*) -

1. [iniciar procura]
1.1. LOW ← 1
1.2. HIGH ← N

2. [repetir procura]
2.1. se LOW ≤ HIGH, então continuar (ir para 3), senão ir para 5

3. [obter o índice do ponto médio do intervalo de procura]
3.1. MIDDLE ← (LOW + HIGH) / 2

4. [comparação]
4.1. se X < K [MIDDLE], então HIGH ← MIDDLE – 1,
senão se X > K [MIDDLE], então LOW ← MIDDLE + 1,
senão indicar "PROCURA COM SUCESSO",
devolver MIDDLE e terminar
4.2. ir para 2

5. indicar "PROCURA SEM SUCESSO", devolver 0 e terminar

2. Algoritmos de Procura (6/7)

- procura binária (*binary search*) -

- **Exemplo**
  - **vector K ordenado composto pelos seguintes elementos:**

    75, 151, 203, 275, 318, 489, 524, 591, 647, 727

  - **2 procuras: X = 275 e X = 727**

Search for 275				*Search for* 727			
Iteration	L	H	M	*Iteration*	L	H	M
1	1	10	5	1	1	10	5
2	1	4	2	2	6	10	8
3	3	4	3	3	9	10	9
4	4	4	4	4	10	10	10

2. Algoritmos de Procura (7/7)

- procura binária recursiva (*recursive binary search*) -

- **Algoritmo *recursive binary search***

RBS (LOW: Integer, HIGH: Integer, K: Sequence, X: T): Integer

1. se LOW > HIGH,
então LOC ← 0 e ir para 3,
senão MIDDLE ← (LOW + HIGH) / 2

2. se X < K [MIDDLE]
então LOC ← RBS (LOW, MIDDLE – 1, K, X)
senão se X > K [MIDDLE]
então LOC ← RBS (MIDDLE + 1, HIGH, K, X)
senão LOC ← MIDDLE

3. devolver LOC e terminar

exercícios

1. Considere os 6 algoritmos recursivos apresentados nas aulas teóricas (sequência S[n], multiplicação de dois números inteiros positivos, factorial, máximo divisor comum, sequência de Fibonacci, torres de Hanoi).

a) escreva, para cada um deles, um programa em C que implemente a versão recursiva dos respectivos algoritmos

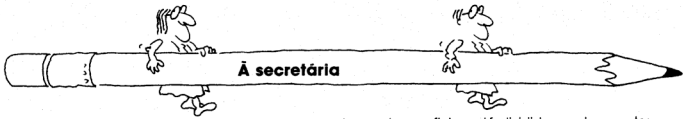
b) determine a profundidade de
b.1) S [34]
b.2) mdc (63, 44)
b.3) hanoi (27, A, C, B)

c) escreva, para cada um deles, um programa em C que implemente a versão iterativa dos respectivos algoritmos e compare com as versões recursivas obtidas na alínea a)

2. Considere os 3 algoritmos de procura apresentados nas aulas teóricas (procura sequencial, procura binária, procura binária recursiva).

a) escreva, para cada um deles, um programa em C que implemente directamente as versões apresentadas daqueles 3 algoritmos

b) determine, para os 3 algoritmos, o número de comparações efectuadas no pior caso



1. a) → Algoritmo refinado para as torres de Hanoi

tower: enum {A, B, C}

hanoi (n: Integer, fromTower, toTower, auxTower: tower)

1. se n = 1,
então
indicar "mover disco 1 da estaca" fromTower "para a estaca" toTower
terminar

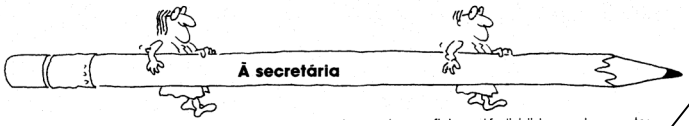
2. hanoi (n–1, fromTower, auxTower, toTower)

3. indicar "mover disco" n "da estaca" fromTower "para a estaca" toTower

4. hanoi (n–1, auxTower, toTower, fromTower)

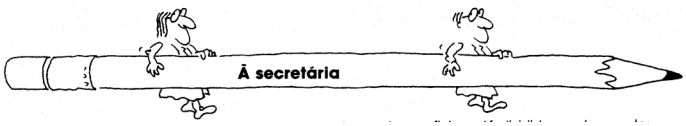
5. terminar

25



1. a) → Execução de hanoi (4, A, C, B)

26



1. a) → Resultado da execução de hanoi (4, A, C, B)

```
mover disco 1 da estaca A para a estaca B
mover disco 2 da estaca A para a estaca C
mover disco 1 da estaca B para a estaca C
mover disco 3 da estaca A para a estaca B
mover disco 1 da estaca C para a estaca A
mover disco 2 da estaca C para a estaca B
mover disco 1 da estaca A para a estaca B
mover disco 4 da estaca A para a estaca C
mover disco 1 da estaca B para a estaca C
mover disco 2 da estaca B para a estaca A
mover disco 1 da estaca C para a estaca A
mover disco 3 da estaca B para a estaca C
mover disco 1 da estaca A para a estaca B
mover disco 2 da estaca A para a estaca C
mover disco 1 da estaca B para a estaca C
```


3. Algoritmos de Ordenação (1/22)

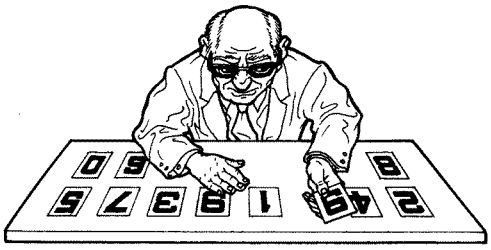
- contexto -

- **Problema de ordenação**
  - **dada uma colecção K de objectos (elementos), pretende-se rearranjar os objectos por uma ordem bem definida (utilizando um ou mais critérios)**
  - **o propósito da ordenação consiste na facilitação de operações de pesquisa (execução de algoritmos de procura sobre conjuntos já ordenados)**
  - **a escolha do algoritmo de ordenação a utilizar em cada caso deve, de entre outros critérios, depender da estrutura dos dados a manipular, originando, segundo este critério, duas categorias de algoritmos de ordenação:**
    - ***ordenação interna*, quando o algoritmo processa dados armazenados em memória principal (pequena, rápida e de acesso aleatório)**
    - ***ordenação externa*, quando o algoritmo processa dados armazenados em memória secundária (grande, lenta e de acesso sequencial)**

3. Algoritmos de Ordenação (2/22)

- contexto -

- **Ordenação interna**
  - **a ordenação interna, de vectores (por exemplo), corresponde a ordenar um baralho de cartas colocado em cima de uma mesa, onde cada uma das cartas é, individualmente, visível e acessível**



3. Algoritmos de Ordenação (3/22)

- contexto -

- **Ordenação externa**
  - **a ordenação externa, de ficheiros (por exemplo), corresponde a ordenar vários baralhos de cartas colocados em várias pilhas em cima de uma mesa, onde de cada uma das pilhas só a carta de cima é, individualmente, visível e acessível**

3. Algoritmos de Ordenação (4/22)

- contexto -

- **Algoritmos**
  - **ordenação por selecção (*selection sort*)**
  - **ordenação por troca (*bubble sort*)**
  - **ordenação por inserção (*insertion sort*)**
  - **ordenação por fusão (*simple merge*)**
  - **ordenação por partição (*quick sort*)**

3. Algoritmos de Ordenação (5/22)

- ordenação por selecção *(selection sort)* -

- **Algoritmo *selection sort***
  - **a colecção K é um vector (*array*) com N elementos**
  - **o algoritmo realiza inspecções sucessivas sequenciais de partes do vector, seleccionando o menor elemento dessa parte, que é, então, colocado na posição correcta, por troca com o elemento que se encontrava na posição a ordenar**
  - **a variável PASS denota a posição do vector (índice do vector) a ordenar em cada passagem**
  - **de facto, PASS consiste no índice do limite inferior da parte do vector a inspeccionar em cada passagem, o limite superior é sempre o índice N**
  - **a variável MIN_INDEX denota a posição do menor elemento por passagem**

3. Algoritmos de Ordenação (6/22)

- ordenação por selecção *(selection sort)* -

1. [passagens pelo vector]

1.1. repetir os passos 2 a 4 com PASS ← 1, 2, ..., N – 1

2. [iniciar posição do menor elemento por passagem]

2.1. MIN_INDEX ← PASS

3. [encontrar menor elemento dentro da passagem actual]

3.1. repetir com I ← PASS + 1, PASS + 2, ..., N
se K [I] < K [MIN_INDEX], então MIN_INDEX ← I

4. [trocar elementos]

4.1. se MIN_INDEX ≠ PASS, então K [PASS] ↔ K [MIN_INDEX]

5. [terminar]

3. Algoritmos de Ordenação (7/22)

- ordenação por selecção *(selection sort)* -

- **Exemplo**

Unsorted		*Pass Number* (i)								*Sorted*
j	K_j	1	2	3	4	5	6	7	8	9
1	42	11	11	11	11	11	11	11	11	11
2	23	23	23	23	23	23	23	23	23	23
3	74	74	74	36	36	36	36	36	36	36
4	11	42	42	42	42	42	42	42	42	42
5	65	65	65	65	65	58	58	58	58	58
6	58	58	58	58	58	65	65	65	65	65
7	94	94	94	94	94	94	94	74	74	74
8	36	36	36	74	74	74	74	94	87	87
9	99	99	99	99	99	99	99	99	99	94
10	87	87	87	87	87	87	87	87	94	99

3. Algoritmos de Ordenação (8/22)

- ordenação por troca *(bubble sort)* -

- **Algoritmo *bubble sort***
  - **a colecção K é um vector (*array*) com N elementos**
  - **o algoritmo realiza comparações directas entre cada dois elementos adjacentes do vector e troca-os da sua ordem original, se tal contribuir para que os elementos de menor valor (segundo os critério de ordenação adoptados) sejam conduzidos para posições de índices de menor valor do vector**
  - **a variável PASS denota o contador de passagens**
  - **a variável LAST denota a posição do último elemento não ordenado**
  - **a variável EXCHS denota o número de trocas efectuadas em cada passagem**

3. Algoritmos de Ordenação (9/22)

- ordenação por troca *(bubble sort)* -

1. [iniciar posição do último elemento não ordenado]

1.1. LAST ← N

2. [passagens pelo vector]

2.1. repetir os passos 3 a 5 com PASS ← 1, 2, ..., N – 1

3. [iniciar contador de trocas na passagem actual]

3.1. EXCHS ← 0

4. [efectuar comparações entre pares adjacentes]

4.1. repetir com I ← 1, 2, ..., LAST – 1
se K [I] > K [I + 1], então K [I] ↔ K [I + 1] e EXCHS ← EXCHS + 1

5. [verificar se existiram trocas na passagem actual]

5.1. se EXCHS = 0, então terminar, senão LAST ← LAST – 1

6. [terminar]

3. Algoritmos de Ordenação (10/22)

- ordenação por troca *(bubble sort)* -

- **Exemplo**

	Unsorted	*Pass Number* (i)					*Sorted*
j	K_j	1	2	3	4	5	6
1	42	23	23	11	11	11	11
2	23	42	11	23	23	23	23
3	74	11	42	42	42	36	36
4	11	65	58	58	36	42	42
5	65	58	65	36	58	58	58
6	58	74	36	65	65	65	65
7	94	36	74	74	74	74	74
8	36	94	87	87	87	87	87
9	99	87	94	94	94	94	94
10	87	99	99	99	99	99	99

3. Algoritmos de Ordenação (11/22)

- ordenação por inserção *(insertion sort)* -

- **Algoritmo *insertion sort***
  - **a colecção K é um vector (*array*) com N elementos**
  - **o algoritmo coloca cada um dos elementos directamente na posição (do vector) até à qual o vector fica ordenado a partir do início**
  - **a variável PASS denota o contador de passagens**
  - **a variável ELEM denota o elemento a inserir na ordem em cada passagem**

***Nota*: este é o algoritmo vulgarmente utilizado pelos jogadores de cartas**

3. Algoritmos de Ordenação (12/22)

- ordenação por inserção *(insertion sort)* -

1. [passagens pelo vector]

1.1. repetir os passos 2 a 3 com PASS ← 2, 3, ..., N

2. [iniciar passagem]

2.1. ELEM ← K [PASS]
2.2. K [0] ← ELEM
2.3. I ← PASS – 1

3. [efectuar comparações sucessivas entre pares adjacentes]

3.1. enquanto ELEM < K [I] fazer
K [I + 1] ← K [I]
I ← I – 1
3.2. K [I + 1] ← ELEM

4. [terminar]

3. Algoritmos de Ordenação (13/22)

- ordenação por inserção *(insertion sort)* -

- **Exemplo**

Initial Keys	44	55	12	42	94	18	06	67
$i = 2$	44	55	12	42	94	18	06	67
$i = 3$	12	44	55	42	94	18	06	67
$i = 4$	12	42	44	55	94	18	06	67
$i = 5$	12	42	44	55	94	18	06	67
$i = 6$	12	18	42	44	55	94	06	67
$i = 7$	06	12	18	42	44	55	94	67
$i = 8$	06	12	18	42	44	55	67	94

3. Algoritmos de Ordenação (14/22)

- ordenação por fusão *(simple merge)* -

- **Algoritmo *simple merge***
  - **a colecção K é um vector (*array*) que armazena 2 sequências ordenadas concatenadas**
  - **o algoritmo recebe um vector com 2 sequências ordenadas e funde as duas, no vector original, de tal forma que a sequência resultante permanece ordenada**
  - **as variáveis FIRST e SECOND denotam, respectivamente, os índices de K em que se iniciam a primeira e a segunda sequências**
  - **a variável LAST denota o maior índice do vector K**
  - **a variável TEMP denota um vector temporário utilizado durante o processo de fusão (ordenada) das duas sequências**

***Nota*: este algoritmo pode ser utilizado para ordenar um vector K que não possua duas sequências ordenadas, desde que se comece por considerar os elementos individuais, depois os pares, ...**

3. Algoritmos de Ordenação (15/22)

- ordenação por fusão *(simple merge)* -

```
1. [iniciar variáveis]

  1.1. I ← FIRST
  1.2. J ← SECOND
  1.3. L ← 0

2. [comparar elementos correspondentes]

  2.1. enquanto I < SECOND e J ≤ LAST fazer
       se K [I] ≤ K [J],
       então L ← L + 1
            TEMP [L] ← K [I]
            I ← I + 1
       senão L ← L + 1
            TEMP [L] ← K [J]
            J ← J + 1
```

3. Algoritmos de Ordenação (16/22)

- ordenação por fusão *(simple merge)* -

3. [copiar os elementos restantes]

3.1. se I ≥ SECOND,
então enquanto J ≤ LAST fazer
L ← L + 1
TEMP [L] ← K [J]
J ← J + 1
senão enquanto I < SECOND fazer
L ← L + 1
TEMP [L] ← K [I]
I ← I + 1

4. [copiar vector temporário para vector K]

4.1. repetir com I ← 1, 2, ..., L
K [FIRST – 1 + I] ← TEMP [I]

5. [terminar]

3. Algoritmos de Ordenação (17/22)

- ordenação por fusão *(simple merge)* -

- **Exemplo:** **vector K [] = 11, 23, 42, 9, 25**

	Table 1	11	23	42		
	Table 2	25				
New	Table	9				
	Table 1	23	42			
	Table 2	25				
New	Table	9	11			
	Table 1	42				
	Table 2	25				
New	Table	9	11	23		
	Table 1	42				
	Table 2					
New	Table	9	11	23	25	
	Table 1					
	Table 2					
New	Table	9	11	23	25	42

3. Algoritmos de Ordenação (18/22)

- ordenação por partição *(quick sort)* -

- **Algoritmo *quick sort***
  - **a colecção K é um vector (*array*) com N elementos**
  - **o algoritmo particiona o vector em dois, garantindo que a partição da esquerda só possui elementos menores do que o elemento a ordenar e que a partição da direita só possui elementos maiores do que o elemento a ordenar, e procede a uma invocação recursiva para ordenar as duas partições criadas**
  - **as variáveis LB e UB denotam, respectivamente, os limites (índices de K) inferior e superior da partição (sub-vector) em ordenação**
  - **a variável ELEM denota a o elemento a inserir na ordem em cada partição**
  - **a variável FLAG é booleana e indica (quando for falsa) que foi atingido o particionamento pretendido**

45

3. Algoritmos de Ordenação (19/22)

- ordenação por partição *(quick sort)* -

```
QUICK_SORT (K: Sequence, LB, UB: Integer)

1. [iniciar variáveis]

  1.1. FLAG ← true

2. [executar ordenação]

  2.1. se LB < UB, então
        I ← LB
        J ← UB + 1
        ELEM ← K [LB]

        2.1.1. [particionar vector em dois]

        K [LB] ↔ K [J]
        QUICK_SORT (K, LB, J – 1)
        QUICK_SORT (K, J + 1, UB)

3. [terminar]
```


46

3. Algoritmos de Ordenação (20/22)

- ordenação por partição *(quick sort)* -

2.1.1. [particionar vector em dois]

enquanto FLAG = *true* fazer
I ← I + 1
enquanto K [I] < ELEM fazer I ← I + 1
J ← J – 1
enquanto K [J] > ELEM fazer J ← J – 1
se I < J, então K [I] ↔ K [J], senão FLAG ← *false*

***Nota*: este algoritmo é inicialmente invocado com QUICK_SORT (K, 1, N)**

3. Algoritmos de Ordenação (21/22)

- ordenação por partição *(quick sort)* -

- **Exemplo**

(42)	(23)	74	11	65	58	94	36	99	87	
(42)	23	(74)	11	65	58	94	36	99	87	
(42)	23	74	11	65	58	94	36	99	(87)	
(42)	23	74	11	65	58	94	36	(99)	87	
(42)	23	74	11	65	58	94	(36)	99	87	interchange 74 and 36
(42)	23	36	(11)	65	58	94	74	99	87	
(42)	23	36	11	(65)	58	94	74	99	87	
(42)	23	36	11	65	(58)	(94)	74	99	87	
(42)	23	36	11	65	(58)	94	74	99	87	
(42)	23	36	11	(65)	58	94	74	99	87	
(42)	23	36	(11)	65	58	94	74	99	87	interchange 42 and 11
11	23	36	42	65	58	94	74	99	87	i ⩾ j

3. Algoritmos de Ordenação (22/22)

- ordenação por partição *(quick sort)* -

- **Exemplo**

K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8	K_9	K_{10}
{42	23	74	11	65	58	94	36	99	87}
{11	23	36}	42	{65	58	94	74	99	87}
11	{23	36}	42	{65	58	94	74	99	87}
11	23	{36}	42	{65	58	94	74	99	87}
11	23	36	42	{58}	65	{94	74	99	87}
11	23	36	42	58	65	{94	74	99	87}
11	23	36	42	58	65	{87	74}	94	{99}
11	23	36	42	58	65	{74}	87	94	{99}
11	23	36	42	58	65	74	87	94	{99}
11	23	36	42	58	65	74	87	94	99

exercícios

1. **Considere os 5 algoritmos de ordenação apresentados nas aulas teóricas (*selection sort*, *bubble sort*, *insertion sort*, *simple merge*, *quick sort*).**

   a) construa, para cada um deles, o respectivo fluxograma

   b) escreva, para cada um deles, um programa em C que implemente directamente a versão do algoritmo apresentada

   c) enuncie e justifique, para cada um dos programas escritos na alínea b), as decisões de implementação subjacentes à codificação em C

4. Complexidade de Algoritmos (1/18)

- limitações computacionais -

- **Limitações da computação**
  - **uma função diz-se computável se puder ser avaliada para quaisquer dados válidos, num número finito de passos, numa máquina de Turing**
  - **é possível mostrar que uma vastíssima classe de funções numéricas são computáveis**
  - **a computacionalidade destas funções permite na prática a computação de qualquer problema algoritmizável, desde que previamente convertido num problema numérico através de codificação conveniente**
  - **as excepções conhecidas de funções não computáveis são de interesse puramente teórico e derivam de um importante resultado acerca do famoso *problema da paragem das máquinas de Turing***

4. Complexidade de Algoritmos (2/18)

- limitações computacionais -

- **Limitação de espaço de memória**
  - **um computador, ao contrário de uma máquina de Turing, possui uma memória finita (constitui uma variante de uma MT designada de *autómato linearmente limitado*)**
  - **contudo, dado o actual desenvolvimento tecnológico, as restrições físicas de espaço de memória são muito pouco importantes, embora continuem a motivar o desenvolvimento de métodos eficientes de gestão de memória**

4. Complexidade de Algoritmos (3/18)

- limitações computacionais -

- **Limitações de complexidade de implementação**
  - **esta limitação continua a ser importante e está intimamente ligada com a linguagem de programação**
  - **à medida que a complexidade dos problemas aumenta, aumenta também a necessidade de manipular entidades de maior nível de abstracção (cada vez mais afastadas da implementação física da máquina)**
  - **tal tem contribuído para o desenvolvimento das linguagens de programação e a pesquisa de novos tipos de linguagens**

4. Complexidade de Algoritmos (4/18)

- limitações computacionais -

- **Limitação de tempo de execução**
  - **a utilidade prática de uma computação depende obviamente do tempo que demora a realizar**
  - **existe, neste caso, um claro limite físico:**

    nenhum sistema computacional (mecânico ou biológico) pode processar mais do que $2x10^{47}$ bit/g.s [Bremermann, 1962]
  - **assim, por exemplo, um computador com a massa do planeta Terra (≅ $6x10^{27}$ g), durante o seu período de existência (≅ 10^{10} anos), só poderia computar um número de bits inferior a 10^{93}**
  - **ou seja, este número de bits é inferior ao número de estados possíveis de um computador com 128 Kbytes e também inferior ao número de sequências possíveis no jogo de xadrês (≅ 10^{120}) !**

4. Complexidade de Algoritmos (5/18)

- análise de algoritmos -

- **Função de custo**
  - **a busca de métodos de programação eficiente em tempo de execução é muito importante, mas não é suficiente**
  - **de facto, existe uma classe de problemas que pela sua índole exige tempos de computação elevados mesmo para dimensões reduzidas dos dados de entrada**
  - **a determinação das exigências de tempo de um algoritmo exige, frequentemente, a realização de uma análise matemática dos vários casos (por exemplo, melhor caso, pior caso e caso médio)**
  - **o resultado típico dessa análise é uma *função de custo* que permite determinar o tempo médio (ou número de operações) necessário para o algoritmo processar N entradas**

4. Complexidade de Algoritmos (6/18)

- análise de algoritmos -

- **Objectivos da análise de algoritmos**
  - **a análise de algoritmos é a área da computação que visa a determinação da complexidade (*custo*) de um algoritmo, tornando possível:**
    - ***comparar algoritmos* - genericamente, existem vários algoritmos que resolvem uma mesma classe de problemas, pelo que através da determinação da complexidade torna-se possível compará-los entre si**
    - ***determinar se um algoritmo é óptimo* - para determinados problemas sabe-se, por determinação matemática, existir um "limite inferior" de complexidade para os algoritmos que os solucionam, pelo que um algoritmo é classificado de *óptimo* quando a sua complexidade é igual à "complexidade inferior limite" do problema**

4. Complexidade de Algoritmos (7/18)

- análise de algoritmos -

- **Exemplo da corda (qualitativo)**
  - **considere-se o problema de dispor espacialmente uma corda de tamanho *T* de tal forma que ocupe o menor espaço possível**
  - **considerem-se três hipóteses alternativas**
    - ***enrolar a corda no braço* - segurar uma das extremidades com uma mão e, dando a volta sob o cotovelo, enrolar a corda várias vezes, mantendo sempre o mesmo comprimento do laço**
    - ***enrolar a corda sobre ela própria* - enrolar a corda à volta de um ponto fixo, aumentando gradualmente o tamanho do laço**
    - ***dobrar a corda sucessivamente* - dobrar a corda ao meio, unindo as duas extremidades, e repetir o processo até que não seja possível dobrá-la mais**
  - **após uma análise matemática das três hipóteses, chega-se à conclusão de que a 3ª hipótese é a mais eficiente, uma vez que resolve o problema em menos passos, i.e., o seu custo é menor**

4. Complexidade de Algoritmos (8/18)

- análise de algoritmos -

- **Exemplo genérico (quantitativo)**
  - **se, por exemplo, da análise efectuada resultar a seguinte fórmula $0{,}01.n^2 + 10.n$, o quadro seguinte ilustra o comportamento do algoritmo para vários valores de entradas**

n	$a = 0.01n2$	$b = 10n$	$a + b$	$\frac{(a+b)}{n^2}$
10	1	100	101	1,01
50	25	500	525	0,21
100	100	1.000	1.100	0,11
500	2.500	5.000	7.500	0,03
1.000	10.000	10.000	20.000	0,02
5.000	250.000	50.000	300.000	0,01
10.000	1.000.000	100.000	1.100.000	0,01
50.000	25.000.000	500.000	25.500.000	0,01
100.000	100.000.000	1.000.000	101.000.000	0,01
500.000	2.500.000.000	5.000.000	2.505.000.000	0,01

4. Complexidade de Algoritmos (9/18)

- complexidade -

- **Ordem de grandeza**
  - **à medida que *n* aumenta, a diferença entre 0,01.n^2 e 10.*n* também aumenta de tal forma que chega a uma altura (*n* = 1 000) a partir da qual em que o termo 0,01.n^2 domina o 10.*n*, pelo que a contribuição deste último torna-se cada vez mais insignificante**
  - **nestas situações, diz-se que a função 0,01.n^2+ 10.*n* é da *ordem de grandeza* da função n^2, ou 0,01.n^2+ 10.*n* é O(n^2), se, porque com o aumento de *n* torna-se cada vez mais proporcional a n^2**
  - **a ordem de grandeza mede a complexidade da função (algoritmo) quando o tamanho da entrada tende para infinito**
  - **a ordem de grandeza permite avaliar o algoritmo independentemente do tamanho real da entrada**

4. Complexidade de Algoritmos (10/18)

- complexidade -

- **Determinação da complexidade**
  - **a complexidade de um algoritmo não é medida em termos reais, ou seja, o algoritmo não é analisado, por exemplo (no que diz respeito ao critério tempo), em termos de escalas absolutas de tempo de execução, mas antes segundo a sua ordem de grandeza**
  - **esse tipo de avaliação não se apresenta adequado, uma vez que existem vários factores não constantes a influenciar o resultado no momento das medições, tais como a eficiência do hardware do computador e a carga do sistema operativo**

X	O(X)
2n + 5	n
n	n
127n * n + 457n	n * n
n * n * n + 5	n * n * n

4. Complexidade de Algoritmos (11/18)

- complexidade -

- **Determinação da complexidade**
  - **a complexidade de um algoritmo pode ser analisada segundo dois critérios ortogonais: tempo e espaço**
  - ***tempo***
    - **é a complexidade mais estudada em algoritmos**
    - **tempo (relativo) necessário à sua execução e cujo modelo associado é baseado no custo de cada operação realizada em função do tamanho da entrada (N)**
    - **em geral, não são consideradas todas as operações, levando-se em consideração apenas as de maior custo**
  - ***espaço***
    - **comportamento relativamente à sua necessidade de memória**
    - **em geral, quanto menor é a complexidade de tempo de um algoritmo, maior é a sua complexidade de espaço**

4. Complexidade de Algoritmos (12/18)

- complexidade -

- **Definição de ordem de grandeza**
  - **sejam $f(n)$ e $g(n)$ duas funções:**
    $f(n)$ é $O[g(n)]$, se
    $(\forall\ n \geq b)\ (\exists\ a \in \mathbb{N} \wedge \exists\ b \in \mathbb{N}$) tal que $f(n) \leq a.g(n)$
  - **exemplos:**
    - **$f(n) = n^2 + 100.n$ e $g(n) = n^2$**
      - **$f(n)$ é $O[g(n)]$, pois $f(n) \leq 2.g(n), \forall\ n \geq 100$**
      - **$f(n)$ é $O[n^3]$, pois $f(n) \leq 2.n^3, \forall\ n \geq 8$**
    - **quando $f(n) = c, \forall\ n$, então $f(n)$ é $O(1)$**
    - **quando $f(n) = c.n$, então $f(n)$ é $O(n^k)$**

***Nota*: a ordem de grandeza beneficia da propriedade transitiva**

4. Complexidade de Algoritmos (13/18)

- complexidade -

- **Função logaritmo**
  - **uma função importante para a análise de algoritmos é a função logaritmo**
  - **prova-se que $\log_m n$ é $O[\log_k n]$ e que $\log_k n$ é $O[\log_m n]$, $\forall$ *m, k***

n	$n \log_{10} n$	n^2
1×10^1	$1{,}0 \times 10^1$	$1{,}0 \times 10^2$
5×10^1	$8{,}5 \times 10^1$	$2{,}5 \times 10^3$
1×10^2	$2{,}0 \times 10^2$	$1{,}0 \times 10^4$
5×10^2	$1{,}3 \times 10^3$	$2{,}5 \times 10^5$
1×10^3	$3{,}0 \times 10^3$	$1{,}0 \times 10^6$
5×10^3	$1{,}8 \times 10^4$	$2{,}5 \times 10^7$
1×10^4	$4{,}0 \times 10^4$	$1{,}0 \times 10^8$
5×10^4	$2{,}3 \times 10^5$	$2{,}5 \times 10^9$
1×10^5	$5{,}0 \times 10^5$	$1{,}0 \times 10^{10}$
5×10^5	$2{,}8 \times 10^6$	$2{,}5 \times 10^{11}$
1×10^6	$6{,}0 \times 10^6$	$1{,}0 \times 10^{12}$
5×10^6	$3{,}3 \times 10^7$	$2{,}5 \times 10^{13}$
1×10^7	$7{,}0 \times 10^7$	$1{,}0 \times 10^{14}$

4. Complexidade de Algoritmos (14/18)

- complexidade -

- **Comportamento**
  - **a complexidade pode ser classificada quanto ao seu comportamento como:**
    - ***polinomial*, quando as funções são $O(n^k)$**
    - ***exponencial*, quando as funções são $O(d^n)$, $\forall$ $d > 1$**

Função	N=10	N=100	N=1000
N Log N	10	200	3000
N ** 2	100	10000	1000000
N ** 3	1000	1000000	1000000000
N ** N	1024	1.26x10**30	*
3 ** N	59049	5.15x10**47	*

(*) Maior que 10 ** 99!

4. Complexidade de Algoritmos (15/18)

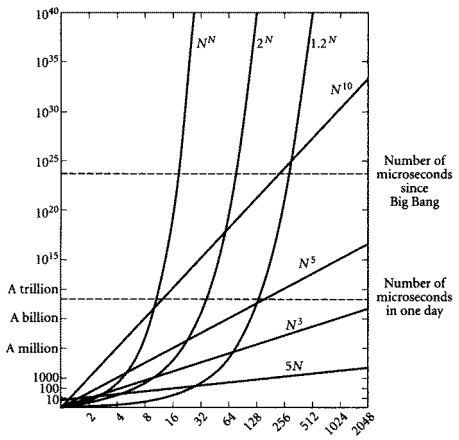
- complexidade -

- **Problemas P-complexos**
  - **os problemas considerados computacionalmente *tratáveis* são aqueles cujo tempo de execução cresce polinomialmente**
  - **a sua complexidade é O[p(*n*)], com p(*n*) um polinómio em *n***
  - **são os chamados problemas da classe P (P de polinomial), ou *P-complexos***
- **Problemas NP-complexos**
  - **os problemas considerados computacionalmente *intratáveis* são aqueles cujo tempo de execução cresce exponencialmente**
  - **são os chamados problemas da classe NP (NP de não polinomiais), ou *NP-complexos*, para os quais é necessário restringir a dimensão dos dados de entrada**

4. Complexidade de Algoritmos (16/18)

- complexidade -

- **Comportamento polinomial vs. exponencial**



4. Complexidade de Algoritmos (17/18)

- complexidade -

- **Comportamento polinomial vs. exponencial**
  - **4 algoritmos executados num computador de 1 MIPS (10^6/s)**

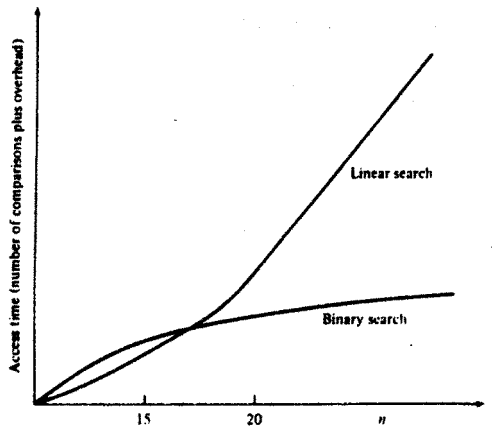
	Input length				
	10	**20**	**50**	**100**	**200**
N^2	1/10 000 second	1/2500 second	1/400 second	1/100 second	1/25 second
N^5	1/10 second	3.2 seconds	5.2 minutes	2.8 hours	3.7 days
2^N	1/1000 second	1 second	35.7 years	Over 400 trillion centuries	A 45-digit no. of centuries
N^N	2.8 hours	3.3 trillion years	A 70-digit no. of centuries	A 185-digit no. of centuries	A 445-digit no. of centuries

For comparison, the Big Bang was 12–15 billion years ago.

4. Complexidade de Algoritmos (18/18)

- complexidade -

- ***linear search* vs. *binary search***



5. Correcção de Algoritmos (1/-)

- verificação vs. validação -

- **Verificação e validação de algoritmos**
  - **a *verificação de algoritmos* tenta garantir que os mesmos estão *correctos***
  - **diz-se que um algoritmo está correcto se o seu comportamento está de acordo com a respectiva especificação**
  - **isto não quer, necessariamente, dizer que um algoritmo correcto resolve o problema para o qual foi projectado para resolver**
  - **as especificações podem não estar de acordo com os requisitos do utilizador, ou podem não prever todos os aspectos inerentes àqueles requisitos**
  - **é na *validação de algoritmos* que se tenta garantir que os mesmos cumprem os requisitos do utilizador**

5. Correcção de Algoritmos (2/-)

- teste vs. prova -

- **Formas de verificação**
  - **a verificação de algoritmos pode ser executada segundo duas abordagens distintas:**
    - ***teste de correcção* (ou rastreio), em que se pretende mostrar informalmente que, para determinados valores de entrada, o algoritmo gera um conjunto de valores de saída válidos**
    - ***prova de correcção*, em que se recorre a técnicas de lógica formal para provar que, dadas quaisquer variáveis de entrada que satisfaçam determinados predicados ou propriedades, o algoritmo gera um conjunto de variáveis de saída que satisfazem outras propriedades especificadas**
  - **a decisão entre realizar teste ou prova de algoritmos recai na avaliação, caso a caso, da dificuldade e necessidade estrita em realizar uma abordagem formal, relativamente aos resultados parciais e mais acessíveis decorrentes da execução de alguns testes ao algoritmo que se pretende verificar**

5. Correcção de Algoritmos (3/-)

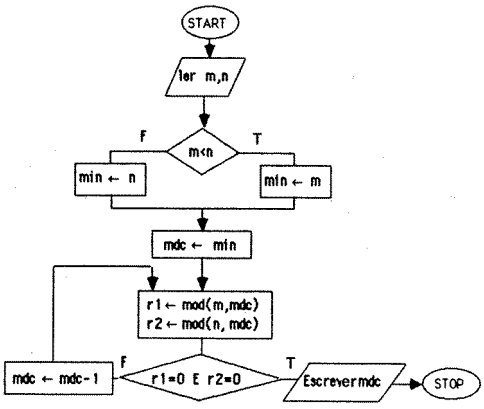
- verificação por teste -

- **Teste de correcção**
  - **o rastreio de algoritmos é efectuado construindo uma tabela descritiva dos sucessivos valores das variáveis manipuladas pelo algoritmo, para determinados valores de entrada e de acordo com o passo do algoritmo em análise**
  - **no rastreio, apenas devem ser assinalados os valores das variáveis quando existe uma alteração dos mesmos**
  - **é recomendável efectuar o rastreio dos algoritmos para duas classes de cenários de valores de entrada:**
    - ***normalidade*: correspondendo a valores típicos**
    - ***contingencionalidade*: correspondendo a valores extremos**
  - **ao contrário da prova, o rastreio pode detectar a existência de erros, mas nunca a sua ausência**

5. Correcção de Algoritmos (4/-)

- verificação por teste -

- **Exemplo: fluxograma do algoritmo do MDC**



5. Correcção de Algoritmos (5/-)

- verificação por teste -

- **Exemplo: tabela de rastreio para o algoritmo do MDC**

TABELA I

Passo	m	n	min	mdc	r1	r2	Comentário
1	12	30					Leitura de m,n
2			12				
3				12			
4					0	6	
5				11			→ 4
4					1	8	
5				10			→ 4
.							
.							
5				6			→ 4
4					0	0	
5							Escreve mdc
							STOP

5. Correcção de Algoritmos (6/-)

- verificação por teste -

- **Exemplo: rastreio do algoritmo MUL**

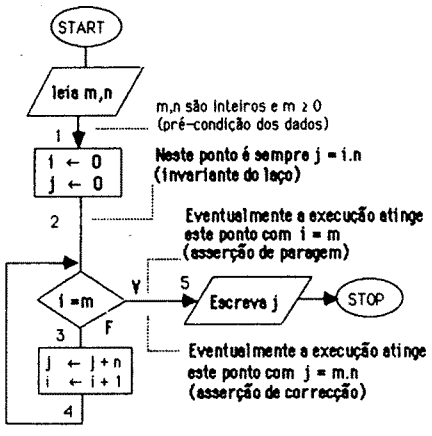


TABELA II

Iteração Ponto 2	i	j
1	0	0
2	1	n
3	2	2.n
4	3	3.n
.	.	.
.	.	.
.	.	.
.	.	.
m+1	m	m.n