Departamento de Matemática da Universidade de Coimbra

Métodos de Programação II – 2004/2005

Exame de 28 de Janeiro de 2005 Duração: **2h40m**

Nota: As respostas que envolvem calculos só serão consideradas para correcção se forem apresentados os cálculos respectivos.

1. Considere um vector, `d[0..n]`, de elementos inteiros e não–negativos.

   (a) Implemente um algoritmo que devolva a soma de todos os elementos de um qualquer vector `d`.

   (b) Elabore um procedimento para estabelecer a representação decimal de um número $k \in \mathbb{N}_0$ num vector do tipo `d[0..n]`, onde, obviamente, todos os elementos de `d` serão um dos possíveis dez algarismos: $\{0,1,2,3,4,5,6,7,8,9\}$.

   (c) É sabido que qualquer $n \in \mathbb{N}_0$ goza de uma conhecida propriedade que pode ser expressa como: $P(n) \equiv \{$ *O resto da divisão de n por nove é igual ao resto da divisão por nove da soma de todos os seus dígitos* $\}$.

   Construa um procedimento para, dado um número natural não–negativo, verificar esta afirmação.

2. Os *números de Euclides*, $2, 3, 7, 43, 1807, \ldots$, podem ser definidos usando a fórmula

$$E_n = \begin{cases} 2 & \text{se } n = 1 \\ E_{n-1}^2 - E_{n-1} + 1 & \text{se } n > 1 \end{cases}$$

   (a) Elabore uma função **recorrente** para calcular e devolver o valor de E_n, dado n.

   (b) Escreva agora uma solução **iterativa** para o mesmo efeito.

   (c) Calcule o número de *somas* efectuadas por cada uma das duas versões anteriores.

   (d) Que diferenças existem em termos de ocupação de memória entre as duas versões? Em função da sua resposta e da alínea anterior, conclua indicando qual é a mais eficiente.

3. Considere as seguintes declarações de tipos:

```
type
  Tiponome = packed array [1..20] of char;
  Seta     = ^Aluno;
  Aluno    = record
               nome: Tiponome;
               prox: Seta
             end;
```

(a) Leia os seguintes sub-programas e explique qual o efeito de cada um deles, supondo que ponta é uma estrutura, já construída, do tipo Seta.

```
procedure XX(var ponta: Seta; nom: Tiponome);
begin
 new(ponta); ponta^.nome := nom; ponta^.prox := nil;
end;

function XY(ponta: Seta; nom: Tiponome):integer;
 var
  p: Seta;
  c: integer;
  continua: boolean;
begin
  p := ponta;  c := 0; continua := true;
  while p <> nil and continua do
  begin
    c := c + 1;
    if p^.nome = nom then continua := false;
    p := p^.prox
  end;
  XY := c
end;

procedure YY(var ponta: Seta);
 var este: Seta;
begin
  while ponta <> nil do
  begin
    este := ponta; ponta := ponta^.prox;
    dispose(este);
  end;
end;
```

(b) Escreva um subprograma para, dados *k*, *estenome* e uma lista de nomes do tipo Seta, procurar e remover o elemento que está na *k*-ésima posição dessa lista *desde que esse elemento contenha estenome.*

4. Considere as seguintes declarações de tipos em Pascal:

```
const Limite = 100;
type
    NTipos = (NReal, NComplexo);
    Complexo = record
                  re, im: real;
               end;
    Numeros = record case tipo: NTipos of
                        NReal: (re: real);
                        NComplexos: (z: complexo);
              end;
   Vector = array[1..Limite] of Numeros;
```

(a) Que tipo de informação representam as declarações acima?

(b) Escreva um procedimento para, dados dois "números" do tipo **Numeros**, efectuar a sua soma, devolvendo o resultado num registo do mesmo tipo.

(c) Modifique, no que for necessário, as declarações apresentadas, de modo a alterar o tipo **Vector** de tabela de **Numeros** para lista ligada de **Numeros**.

(d) Indique quais as vantagens e quais as desvantagens decorrentes da alteração efectuada.

(e) Utilizando o módulo construído na alínea (b) e a **nova declaração de Vector**, elabore um novo módulo para efectuar a soma de todos os elementos de um dado **Vector** v, e devolver o valor obtido.

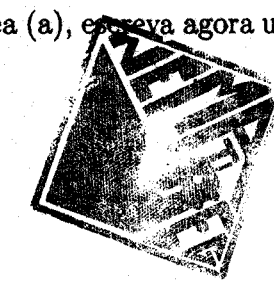
O grupo que se segue implica a desistência da nota dos trabalhos de casa.

5. No que se segue, considere tabelas (vectores) de *elementos inteiros todos distintos.*

(a) Elabore um procedimento *iterativo* para, dado um vector de dimensão n, procurar e devolver os *indíces dos dois menores elementos* desse vector.

(b) Caracterize o pior dos casos para o algoritmo da alínea anterior, calculando qual o número máximo de comparações (envolvendo apenas elementos do vector) e explicitando as circunstâncias em que esse pior dos casos pode ocorrer.

(c) Usando o procedimento implementado na alínea (a), escreva agora um procedimento para ordenar por ordem crescente um dado vector.



(d) Determine a ordem de complexidade, em termos de comparações e no pior dos casos, do algoritmo de ordenamento que propôs na alínea anterior.

(e) Em função das ordens de complexidade dos algoritmos estudados neste curso, como caracteriza o resultado obtido na alínea anterior e em que situações pensa ser adequado o uso deste algoritmo para o ordenamento de tabelas?

Proposta de Resolução para o Exame da Época Normal, 1o. semestre, 2004/2005

1. Neste grupo, e sem perda de generalidade, vamos considerar pré-declarados os seguintes tipos:

```
typedef
    IntNneg = 0..MAXINT-1;
    Vector = array [0..LIMITE] of IntNneg;
```

sendo LIMITE uma constante anteriormente definida com valor considerado adequado.

(a) Dado um qualquer vector, $d[0..n]$, de n elementos inteiros e não–negativos, onde consideramos que o valor n é, também, um dos dados do problema, vamos construir um algoritmo para devolver a soma de todos os n elementos de d. Assim, temos:

Dados: $d[0..n]$ do tipo *Vector* e $0 \leq n \leq LIMITE$;

Resultados: o valor da soma de todos os n elementos de d.

Para obter este resultado, basta utilizar um ciclo a variar de 1 a n para, a partir do primeiro elemento, somar o valor acumulado das adições anteriores com o valor do próximo elemento, ou seja,

Algoritmo:

soma $\leftarrow d[0]$;

para i a variar de 1 a n: soma $\leftarrow$ soma + $d[i]$;

devolver o valor de soma;

Portanto, a implementação, em linguagem Pascal, do algoritmo acima descrito poderá ser a seguinte:

```
function soma_vector(d: Vector; n: IntNneg): IntNneg;
var soma, i: IntNneg;
begin
   if (n >= 0) and ( n <= LIMITE)
   then begin
           soma := d[0];
           for i:=1 to n do soma := soma + d[i];
           soma_vector := soma;
        end
   else soma_vector := -MAXINT; { Erro }
end;
```

A única validação importante neste momento é a que garanta que não se ultrapassa nenhum dos limites previamente definidos para o vector. (O facto de algum dos elementos do vector dado não ser não-negativo não altera em nada o algoritmo e não deve ser testado dentro deste.) A devolução do valor negativo de *-MAXINT* significa que ocorreu um erro devido a ultrapassagem de algum dos limites do vector.

(b) Sabemos que a representação decimal de um dado número $k \in \mathbb{N}_0$ é dada pela notação posicional de base 10, ou seja, $k = \sum_{i=0}^{n} d_i 10^i$, sendo n a posição do dígito mais significativo e $d_i \in \{0, 1, 2, \ldots, 9\}$. Assim, para estabelecer esta representação num vector do tipo Vector, basta

obter cada dígito de k, o que se consegue através da divisão sucessiva de k por 10. O seguinte algoritmo pretende estabelecer, então, a representação pedida:

Dados: $k \in \{0, 1, 2, \ldots, MAXINT-1\}$

Resultados: um vector $d[0..n]$ do tipo *Vector* e n, indíce da última posição ocupada no vector d, tais que, $k = \sum_{i=0}^{n} d[i]10^i$

Note-se que, o algoritmo seguinte devolve um vector do tipo pedido onde, de facto, na primeira posição aparece o dígito das unidades e na última posição fica o dígito mais significativo.

Algoritmo:

$i \leftarrow -1$;
repetir até que k = 0:
    $i \leftarrow i + 1$;
    $d[i] \leftarrow$ resto da divisão de k por 10;
    $k \leftarrow$ quociente da divisão de k por 10;
devolver o vector d e o valor de i;

Uma implementação possível para este algoritmo será:

```
procedure vector_de_digitos(k: IntNneg; var d: Vector; var n: IntNneg);
begin
   n := -1;
   repeat
      n := n+1;
      d[n] := k mod 10;
      k := k div 10;
   until (k = 0) or (n = LIMITE);
   if  (k <> 0) and (n = LIMITE) then { mensagem de erro }
end;
```

Neste caso, devemos ter o cuidado de não ultrapassar a capacidade máxima do vector, enviando uma mensagem de erro caso o número seja maior do que a capacidade de armazenamento. (Uma alteração simples que assinalaria o erro seria a de implementar uma função booleana que devolveria *false* em caso de erro e *true* caso contrário.)

(c) Em função das alíneas anteriores, dado $k \in \mathbb{N}_0$, para testar a propriedade expressa no enunciado bastará, então, implementar o seguinte algoritmo.

Algoritmo:

- estabelecer a representação decimal de k num vector $d[0..n]$, conseguindo o valor de n;
- somar os elementos de $d[0..n]$;
- comparar o resto da divisão desta soma por 9 com o resto da divisão de k por 9;
- se iguais devolver *verdadeiro* senão devolver *falso*

Ou seja, supondo que está assegurada a ausência de erro,

Dados: $k \in \{0, 1, 2, \ldots, MAXINT-1\}$;

Resultados: *true* se $P(k)$ e *false* se $\neg P(k)$;

Implementação:

```
function P(n: IntNneg): boolean;
var
   v  : Vector;
   m  : IntNneg;
begin
   vector_de_digitos(k,v,m);
   if ( soma_vector(v,m) mod 9 ) = ( k mod 9 )
   then P := true
   else P := false
end;
```

2.

(a) No caso da versão recorrente para a implementação da função que define, recorrentemente, os *números de Euclides*, o algoritmo corresponde à tradução directa da definição matemática. Assim, temos,

Dados: $n \in \{1, 2, \ldots, MAXINT\text{-}1\}$;

Resultados: O valor do número de Euclides de n: E_n;

Algoritmo:

se $n = 1$ devolver 2;

senão:

calcular o valor do número de Euclides para $n - 1$, E_{n-1};

devolver $E_{n-1}^2 - E_{n-1} + 1$;

Note-se que, sendo uma avaliação recorrente pesada computacionalmente, o algoritmo calcula **uma única vez** o valor de E_{n-1}, e é esse valor que depois é usado no cálculo da fórmula seguinte. Ou seja, numa implementação deste algoritmo é feita **uma única avaliação recorrente**. Portanto, podemos implementar este algoritmo usando a seguinte função em Pascal, que faz a validação da entrada n:

Implementação:

```
function EucR(n: integer): integer;
var aux: integer;
begin
   if n = 1 then EucR := 2
   else if n > 1 then begin
                         aux := EucR(n-1);
                         EucR := aux * aux - aux + 1
                      end
        else EucR := -MAXINT; { ERRO: n, negativo ou nulo }
end;
```

(b) Mais uma vez, a solução iterativa é extremamente simples: basta inicializar uma variável auxiliar ao primeiro valor possível para um *número de Euclides* e, em seguida, usar um ciclo com contador crescente (de 2 a n) que vá acumulando os valores sucessivos do cálculo da fórmula recorrente. Temos, então,

Dados: $n \in \{1, 2, \ldots, MAXINT\}$;

Resultados: O valor do número de Euclides de n: E_n;

Implementação:

```
function EucI(n: integer): integer;
var aux, i: integer;
begin
   if n > 0
   then begin
           aux := 2;
           for i:= 2 to n do aux := aux * aux - aux + 1;
           EucI := aux;
        end
   else EucI := -MAXINT; { ERRO: n, negativo ou nulo }
end;
```

(c) Para o cálculo do número de somas da versão recorrente, $S_R(n)$, note-se que só existem somas no caso de $n > 1$, ou seja,

$$S_R(n) = \begin{cases} 0 & \text{se } n = 1 \\ 3 + S_R(n-1) & \text{se } n > 1 \end{cases}$$

Donde, dado $n > 1$, aplicando recorrentemente a fórmula acima obtemos,

$$S_R(n) = 3 + S_R(n-1) = 3 \times 2 + S_R(n-2) = 3 \times 3 + S_R(n-3) = \ldots = 3k + S_R(n-k)$$

Esta recorrência tem de parar quando $n - k = 1$, logo, quando $k = n - 1$. Assim, no pior dos casos temos,

$$S_R(n) = 3(n-1) + S_R(1) = 3n - 3\,,$$

e este é o número máximo de somas efectuadas pela versão recorrente no cálculo do número E_n.

A versão iterativa só efectua somas no ciclo *for*. Em cada volta do ciclo são feitas 3 somas e, como o ciclo efectua $n - 2 + 1 = n - 1$ voltas, temos, finalmente, que o número de somas, S_I, efectuado por esta última verão, dado $n > 1$, é:

$$S_I(n) = (n-1) \times 3 = 3n - 3\,.$$

(d) A ocupação de memória efectuada pela versão iterativa, *EucI*, é muito diminuta pois, para além da própria chamada que inclui a cópia da entrada n, apenas necessita de mais 2 variáveis do tipo *integer* para calcular o valor pedido. Portanto, EucR, quando é chamada, usa a memória para uma chamada a função mais 3 variáveis do tipo *integer*, que podemos fazer corresponder, de um modo simplista, a, aproximadamente, 4 "blocos" de memória.

Já a função recorrente, necessita, para cada entrada, de guardar uma chamada de função, a cópia da respectiva entrada e uma variável auxiliar do tipo *integer*. Dado n, vamos, em cascata, abrir

$n-1$ chamadas recorrentes, que só terminarão, também em cascata, após o cálculo da fórmula respectiva e saída da correspondente chamada. Logo, para calcular o mesmo valor da versão iterativa, *EucR* ocupa memória correspondente a: n chamadas de funcção $+ n$ cópias do respectivo parâmetro $+ (n-1)$ variáveis auxiliares.

Neste caso temos, então, uma ocupação de memória que é directamente proporcional ao número entrado n, e que será, aproximadamente, de $3n-1$ "blocos" de memória.

Em conclusão, apesar de ambas as versões terem o mesmo "trabalho" para o cálculo do número E_n, a verdade é que a versão iterativa ocupa substancialmente menos memória para esse cálculo, pelo que é a versão mais eficiente.

3.

(a) O procedimento *XX* reserva uma nova estrutura de registo do tipo *Aluno*, que fica apontada pela variável *ponta* do tipo *Seta*, devolvendo esta alteração ao módulo que chamou este procedimento.

A função *XY* recebe uma lista ligada de alunos através do argumento por valor *ponta* do tipo *Seta* e um argumento por valor identificado como *nom* e do tipo *Tiponome*. A função vai procurar, na lista ligada dada, a primeira ocorrência de um registo que contenha um campo *nome* com valor igual ao do parâmetro *nom* e devolve o indíce posicional de tal registo na lista dada.

Note-se que, caso não encontre uma tal identificação, é devolvido o índice de posição do último elemento da lista ou, o que é o mesmo, o número de elementos na lista dada.

O procedimento *YY* recebe uma lista ligada através da variável *ponta* do tipo *Seta* e destrói totalmente essa lista, libertando todos os registos para o Sistema Operativo. No final, a variável ponta é devolvida o valor *nil*.

(b) O algoritmo para implementar o subprograma pedido deve, em primeiro lugar, procurar o k-ésimo elemento da lista ligada dada. Em seguida, caso tenha encontrado e caso haja coincidência entre o valor do argumento *estenome* e o valor no campo *nome* do registo encontrado, deve remover esse registo, actualizando a lista dada. Em qualquer outra situação, nada há a fazer.

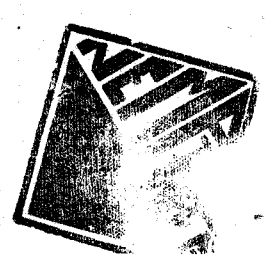
Uma implementação para o algoritmo anteriormente descrito poderá, então, ser a seguinte:

Dados: *ponta* do tipo *Seta*, k do tipo *integer*, *estenome* do tipo *Tiponome*;

Resultados: *ponta* com $k-1$ elementos sse a k-ésima posição dessa lista continha *estenome*;

Implementação:

```
procedure removeK(var ponta: Seta; k: integer; estenome: Tiponome);
var
   aux, p: Seta;
   c     : integer;
begin
   aux := nil; p := ponta; c := 1;
   while (c < k) and (p <> nil) do
   begin
      aux := p;
      p := p^.prox;
      c := c+1;
   end;
```



```
    if (c = k) and ( p <> nil)
    then if p^.nome = estenome
         then begin
                 if aux <> nil
                 then aux^.prox := p^.prox
                 else ponta := p^.prox;
                 dispose(p)
              end;
end;
```

4.

(a) O total das declarações pretende definir uma estrutura de dados, *Vector*, que é uma lista estática (tabela) de números complexos. De acordo com a definição matemática de número complexo, este pode ser um real puro ou complexo com parte imaginária, que diremos **completo**. Assim, cada complexo *completo* é representado por um registo, *Complexo*, com 2 campos do tipo *real*, que correspondem aos coeficientes das partes imaginária e real do complexo. Cada número é representado, então, por um registo do tipo *Numero*, com parte variável, permitindo que um real puro seja representado apenas por um campo do tipo *real* e um complexo completo por um campo do tipo *Complexo*.

(b) Dados dois quaisquer números, para efectuar a sua soma é necessário saber quais os respectivos tipos. Assim, temos três hipóteses possíveis:

i. São dados 2 reais puros. Soma-se os respectivos valores para uma nova variável do mesmo tipo.

ii. São dados 2 complexos. Neste caso é necessário somar as respectivas partes reais e, separadamente, as respectivas partes imaginárias, guardando os resultados numa variável do mesmo tipo.

iii. É dado um número de cada tipo. Devemos então tratar o número real como um complexo com coeficiente da parte imaginária nulo, e proceder como em (ii), devolvendo o resultado numa variável do tipo *Complexo*.

Apresentamos, seguidamente, uma implementação para o algoritmo anterior:

Dados: dois números;

Resultados: a respectiva soma;

Implementação:

```
procedure soma(x, y: Numero; var res: Numero);
begin
   if X.tipo = Y.tipo
   then if X.tipo = NReal
        then begin
                res.tipo := NReal;
                res.re := X.re + Y.re;
             end
        else begin
                res.tipo := NComplexo;
```

```
                    res.z.re := X.z.re + Y.z.re;
                    res.z.im := X.z.im + Y.z.im;
                end
        else begin
                res.tipo := NComplexo;
                if X.tipo = NReal
                then begin
                        res.z.re := X.re + Y.z.re;
                        res.z.im := Y.z.im;
                    end
                else begin
                        res.z.re := X.z.re + Y.re;
                        res.z.im := X.z.im;
                    end
            end
end;
```

Note-se que, um aperfeiçoamento possível para o algoritmo e implementação anteriores, será o de verificar se a soma dos coeficientes das partes imaginárias de complexos completos se anulam. Neste caso, o resultado deverá ser um real puro.

(c) Para efectuar a alteração pedida no enunciado, basta proceder às seguintes modificações, **na ordem apresentada**:

```
Vector  = ^Caixa;
Caixa = record
            num : Numeros;
            prox: Vector;
        end;
```

(d) Vantagens em usar uma lista dinâmica (com ponteiros) em vez de uma lista estática (com a tabela):

- Número variável de elementos na lista, limitado apenas pela memória disponível e correspondente ao estritamente necessário em cada momento.
- Inserções e remoções na lista não implicam deslocamentos dos restantes elementos.

Desvantagens:

- Acesso apenas sequencial.
- Pesquisa apenas sequencial, mesmo em listas ordenadas.

(e) O algoritmo básico para efectuar a soma pretendida é, no seu essencial, igual ao apresentado na alínea (1.a). A única diferença substancial consiste no facto de o número de elementos da lista ser desconhecido. Assim, teremos que testar, em cada momento, se a lista tem ou não mais elementos. Para a soma propriamente dita vamos usar o procedimento criado na alínea (4.b).

Dados: uma lista ligada de números;

Resultados: a soma de todos os números da lista;

Implementação:

```
procedure soma_lista(lista: Vector; var res: Numeros);
var aux   : Vector;
begin
   if lista <> nil   { se existe, pelo menos, 1 elemento }
   then begin
           res := lista^.num; lista := lista^.prox,
           while lista <> nil do soma(res, lista^.num, res);
        end
   else begin  { ERRO }
           res.tipo := NComplexo;
           res.z.re := INFINITO;
           res.z.im := INFINITO;
           { possivel mensagem de erro aqui}
        end
end;
```

Note-se que considerámos a declaração prévia de uma constante (*INFINITO*) como sendo o maior valor do tipo *real* representável.

5. Neste grupo, e de acordo com o enunciado, consideraremos pré-declarado o seguinte tipo:

```
typedef Vector = array [1..LIMITE] of integer;
```

sendo LIMITE uma constante anteriormente definida com valor considerado adequado.

Este tipo define uma estrutura de dados do tipo tabela de inteiros, com a propriedade de que *os elementos de um Vector são todos distintos.*

(a) Vamos pesquisar os dois menores elementos para um dado $v[1..n]$ do tipo *Vector* que, devido à propriedade acima, serão, necessariamente, diferentes. Assim, será devolvido em primeiro lugar o índice do elemento mínimo e, a seguir, o índice do segundo menor elemento. O algoritmo começará por estabelecer qual é o menor dos dois primeiros elementos e, em seguida, terá de efectuar uma pesquisa exaustiva em toda a dimensão do vector, para determinar com toda a certeza, quais são os índices pretendidos. Teremos, portanto, uma implementação como a seguinte:

Dados: uma tabela do tipo *Vector* e respectiva dimensão;

Resultados: os índices dos dois menores elementos da tabela (por ordem inversa de grandeza);

Implementação:

```
procedure pesquisa2min(v: Vector; n: integer; var ind1, ind2: integer);
var i: integer;
begin
   if n > 1
   then begin
           if v[1] < v[2]
           then begin
                   ind1 := 1; ind2 := 2
                end
           else begin
```

```
                ind1 := 2; ind2 := 1
            end;
        for i:= 3 to n do
        begin
           if v[i] < v[ind1]
           then begin
                   ind2 := ind1; ind1 := i
                end
           else if v[i] < v[ind2] then ind2 := i;
        end
    end
  else  { possivel mensagem de erro aqui}
end;
```

(b) Dado um vector de dimensão n, é sempre feita uma comparação para inicializar os índices pretendidos. A única variação (a existir) será na contagem de comparações no ciclo, uma vez que, consoante o valor de verdade na comparação do *if* interno ao ciclo, poderá fazer-se, ou não, a comparação do *else*. Assim, e para que sejam feita o máximo de comparações possível, é necessário obrigar a que o valor no *if* seja falso. Para que isso aconteça, é necessário que, $\forall i \in \{3,4,\ldots,n\}$, $v[i] > v[ind1]$. Ou seja, o pior dos casos acontece sempre que o elemento mínimo é um dos dois primeiros elementos do vector.

Neste caso, o número de comparações será, então,

$$C_p(n) = 1 + \sum_{i=3}^{n} 2 = 1 + 2(n-3+1) = 2n-3\,.$$

(c) O algoritmo que usa o procedimento da alínea 5.a) para ordenar um dado vector é, simplesmente uma variação do método de ordenamento por *Selecção Linear* onde, ao invés da pesquisa e colocação na posição respectiva de um elemento extremo, são pesquisados (e colocados) na sua posição 2 elementos de cada vez.

Para facilitar a pesquisa, vamos alterar o procedimento de 5.a) para receber não só o índice do extremo direito do vector mas, também, o índice do extremo esquerdo, para saber onde começa a procura.

Para além deste procedimento, incluiremos também um procedimento para efectuar a troca de 2 elementos de um dado vector, dados os respectivos índices.

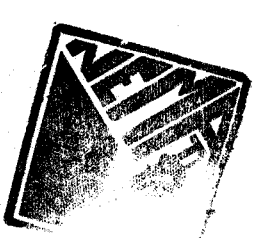
Dados: uma tabela do tipo *Vector* e respectiva dimensão;

Resultados: a tabela dada, ordenada por ordem crescente dos seus elementos;

Implementação:

```
procedure ordena( var v: Vector, n: integer);
var i1, i2, esq: integer;

procedure troca( var v:Vector; i, j: integer);
var aux: integer;
begin
   aux := v[i]; v[i] := v[j]; v[j] := aux;
```



```
end;{ troca }

procedure pesquisa2min(v: Vector; e,d: integer; var ind1, ind2: integer);
var i: integer;
begin
   if d-e > 1
   then begin
          if v[e] < v[e+1]
          then begin
                 ind1 := e; ind2 := e+1
               end
          else begin
                 ind1 := e+1; ind2 := e
               end;
          for i:= e+2 to d do
          begin
            if v[i] < v[ind1]
            then begin
                   ind2 := ind1; ind1 := i
                 end
            else if v[i] < v[ind2] then ind2 := i;
          end
        end
    else  { possivel mensagem de erro aqui}
end;{ pesquisa2min }

begin
   esq := 1;
   while esq < n do
   begin
     pesquisa2min(v,esq,n,i1,i2);
     if i1 <> esq then troca(v,esq,i1);
     if i2 <> esq+1 then troca(v,esq+1,i2);
     esq := esq+2;
   end
end;{ordena}
```

(d) Se considerarmos apenas comparações entre elementos do vector teremos, então, que, sendo $C_p()$ o número de comparações efectuadas na chamada ao procedimento de pesquisa,

passagem		no. de comparações
1	—	$C_p(n)$
2	—	$C_p(n-2)$
3	—	$C_p(n-4)$
	⋮	
k	—	$C_p(n-2(k-1))$

Note-se que, em cada volta do ciclo, são colocados 2 elementos na sua posição final e, na próxima volta, serão, portanto, pesquisados menos 2 elementos que na volta anterior.
Assim, o número total de comparações, $C(n)$, será:

$$C(n) = C_p(n) + C_p(n-2) + \ldots + C_p(n-2(k-1)).$$

O ciclo pára quando o limite esquerdo atingir (ou ultrapassar) o limite direito mas, uma vez que também só se efectua a pesquisa se a diferença entre os dois valores for superior a 1, isto significa que a última volta com comparações acontece quando $n-2(k-1)=2$. Logo, o último valor para contagem acontece na volta $k = n/2$. Logo, vem,

$$C(n) = \sum_{k=1}^{n/2} C_p(n-2(k-1)) = \sum_{k=1}^{n/2} C_p(n-2k+2)$$

e, como, pela alínea 5.b), $C_p(i) = 2i-3$, então, temos,

$$\begin{aligned} C(n) = \sum_{k=1}^{n/2} 2(n-2k+2) = \sum_{k=1}^{n/2}(2n+2) - \sum_{k=1}^{n/2} 4k = 2(n+1)\frac{n}{2} - 4\frac{1+n/2}{2}\frac{n}{2} \\ = n(n+1) - n(1+\frac{n}{2}) = \frac{3}{2}n(n-1) \end{aligned}$$

(e) Este algoritmo tem ordem de complexidade quadrática, pelo que pertence à classe das ordens mais elevadas de complexidade para algoritmos de ordenamento. As melhores ordens conhecidas são da classe de $\mathcal{O}(\log_2 n)$. Assim, este algoritmo só deverá ser escolhido para o ordenamento de tabelas com poucos elementos, sob pena de o ordenamento se tornar demasiado pesado. Mais ainda, uma vez que o algoritmo faz, obrigatoriamente, ciclos fixos de pesquisa exaustiva, apenas se deve usar se houver uma forte suspeita (ou mesmo a certeza absoluta) que os elementos da tabela estão, de facto, bastante desarrumados.