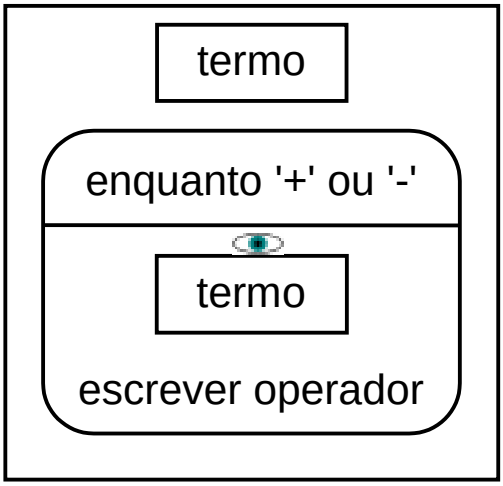
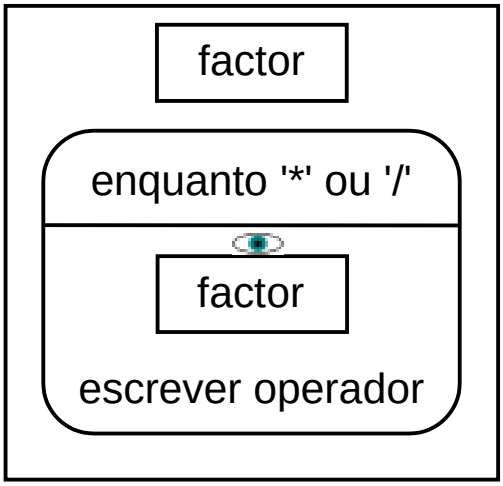
Para resolver o problema, basta construir um Módulo (procedimento) para cada uma das definições:

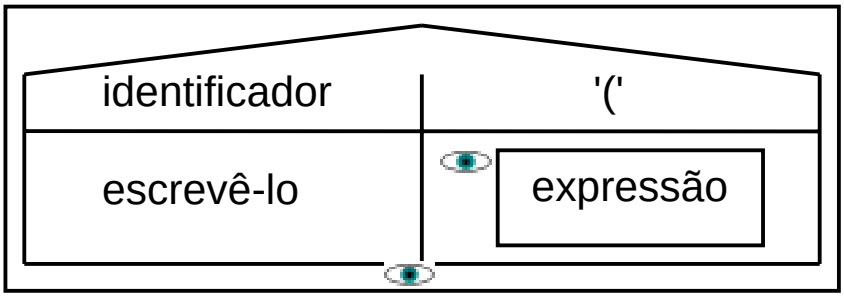
expressão



termo



factor



Exemplo: a * (b + c) → a b c + *

expressão:
- termo: factor: escrever(**a**)
- factor: expressão:
  - termo:
    - factor: escrever(**b**)
  - termo:
    - factor: escrever(**c**)
  - escrever(**+**)
- escrever(*****)

Programa Pascal:

```
program InfixPostfix(input, output);
var car : char;

procedure procurar;
          begin repeat read(car);
                until (car <> ' ') or eoln(input)
          end (* procurar *);

procedure expressao;
          var operaditivo : char;

          procedure termo;
                    var opermult : char;

                    procedure factor;
                              begin (* factor *)
                                    if car = '('
                                    then begin procurar;
                                               expressao
                                         end
                                    else write(car);
                                    procurar
                              end (* factor *);

                    begin (* termo *)
                          factor;
                          while (car = '*') or (car = '/') do
                                begin opermult:= car;
                                      procurar;
                                      factor;
                                      write(opermult)
                                end
                    end (* termo *);
```

```
        begin (* expressao *)
              termo;
              while (car = '+') or (car = '-') do
                    begin operaditivo:= car;
                          procurar;
                          termo;
                          write(operaditivo)
                    end
        end (* expressao *);

begin (* Programa Principal *)
      procurar;
      repeat expressao;
             writeln
      until car = '.'
end (*InfixPostfix *).
```

Alguns resultados:

input	output
(a+b)*(c -d)	ab+cd-*
a+b*c-d	abc*+d-
(a +b)* c-d	ab+c*d-
a+b*(c-d)	abcd-*+
a *a *a*a	aa*a*a*
b+c *(d+c*a*a) *b + a	bcdca*a*+*b*+a+
(a* b) / c	ab*c/
a*b/c	ab*c/
a*(b/c)	abc/*
a*a/a*a/a*a .	aa*a/a*a/a*

Problemas:

- **Calcular o valor de uma expressão *postfix*.**
- **Conversão *postfix* → *infix*.**

Os Métodos de Pesquisa também admitem Abordagens Recorrentes ...

Ex13: Pesquisa Linear num Vector (não ordenado):

Versão Iterativa de um Módulo Completo de Pesquisa:

```
procedure pesquisar (x : vector; n : integer; este : elemento;
                                  var achou : boolean; var : indice : integer);
var i : integer;
begin  i:= 1;
       while  (x[i] <> este) and (i<n) do
                i:= i +1;
       (* Aqui (x[i] = este) or (i>=n) *)
       achou:= x[i] = este;
       if achou
       then  indice:= i
       else  indice:= 0
end;
```

Uma Estratégia Recorrente:

procurar em x[1..n]:
se este = x[n]
então achou
senão procurar em x[1..n-1]

Contudo,

- **É necessário garantir que o processo pára mesmo que o elemento não seja encontrado, isto é, falta identificar o Caso Trivial da Recorrência.**
- **Não convem que o procedimento Recorrente contenha a lista completa dos Parâmetros pois, por exemplo** (x : vector) **iria gerar uma cópia de todo o vector para cada Chamada Recorrente.**

- **Uma solução consiste no Encapsulamento da Recorrência, isto é, na utilização de um procedimento Recorrente interno ao Módulo de Pesquisa pretendido.**

Módulo de Pesquisa com Recorrência Encapsulada:

```
procedure pesquisar (x : vector; n : integer; este : elemento;
                     var achou : boolean; var : indice : integer);

        procedure pesq(n : integer);
                  begin if n>=1
                        then begin if este = x[n]
                                   then indice := n
                                   else pesq(n-1)
                             end
                        else indice:= 0
                  end (* pesq *);

begin pesq(n);
      achou:= indice<>0
end (* pesquisar *);
```

- **O subprograma Recorrente pode também ser uma Função:**

```
procedure pesquisar (x : vector; n : integer; este : elemento;
                     var achou : boolean; var : indice : integer);

        function pesq(n : integer) : integer;
                 begin if n>=1
                       then begin if este = x[n]
                                  then pesq:= n
                                  else pesq:= pesq(n-1)
                            end
                       else pesq:= 0
                 end (* pesq *);

begin indice:= pesq(n);
      achou:= indice<>0
end (* pesquisar *);
```

Ex14: Pesquisa num Vector ordenado:

Módulo de Pesquisa Binária com Recorrência Encapsulada:

```
procedure pesquisar (x : vector; n : integer; este : elemento;
                     var achou : boolean; var : indice : integer);

  function  pesq(esq, dir : integer) : integer;
            var meio : integer;
            begin  if esq <= dir
                   then  begin  meio:= (esq+dir) div 2;
                                if este = x[meio]
                                then  pesq := meio
                                else  if este < x[meio]
                                      then  pesq:= pesq(esq, meio-1)
                                      else  pesq:= pesq(meio+1, dir)
                         end
                   else pesq:= 0
            end (* pesq *);

begin  indice:= pesq(1, n);
       achou:= indice<>0
end (* pesquisar *);
```

Exercício: Identificar o <u>maior</u> elemento de um vector, utilizando a seguinte Estratégia Recorrente:

procurar máximo (de todo o vector):
procurar máximo (da primeira metade);
procurar máximo (da segunda metade);
escolher o maior.

E também os Métodos de Ordenamento ...

Ex15: Ordenamento por Inserção (Ver Cap. V, pág. 22):

Uma Estratégia Recorrente:

```
ordenar x[1..n]:
        ordenar x[1..n-1];
        inserir x[n] em x[1..n-1].
```

Módulo Recorrente de Ordenamento por Inserção:

```
procedure OrdInsercao (var x : vector; n : integer);

    procedure inserir(novo : elemento; k : integer);
              (* Ver Cap. V, pág. 15 *)
              var i : integer;

              begin if novo >= x[k]
                    then x[k+1]:= novo
                    else begin i:= k;
                               while (i>=1) and (x[i]>novo) do
                                     begin x[i+1]:= x[i];
                                           i:= i-1
                                     end;
                               x[i+1]:= novo
                         end;
              end (* inserir *);

begin if n>1
      then begin OrdInsercao(x, n-1);
                 inserir(x[n], n-1)
           end
end (* OrdInsercao *);
```

A Abordagem Recorrente permite a construção de Métodos de Ordenamento muito eficientes ...

Ex16: Ordenamento por Fusão:

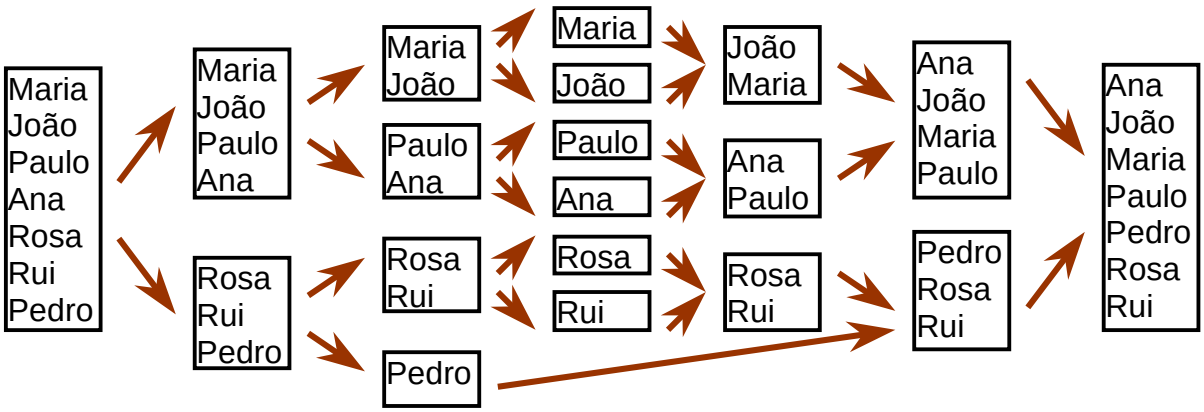
Observações:

- **O Problema da Fusão de dois Vectores, já ordenados, é menos complexo do que o Problema do Ordenamento;**
- **A Recorrência permite transformar (reduzir) o processo do Ordenamento num processo de Fusão Ordenada.**

Estratégia Recorrente:

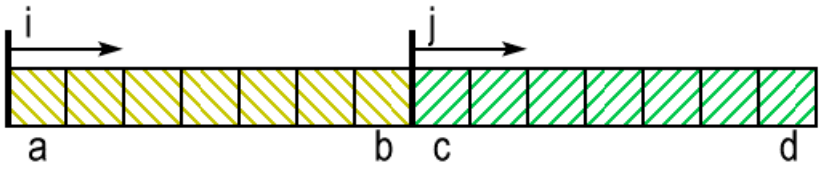
ordenar vector:
ordenar a primeira metade;
ordenar a segunda metade;
fundir as duas metades.

Ex.:



- **Basta portanto construir um módulo para a Fusão de dois SubVectores já ordenados.**

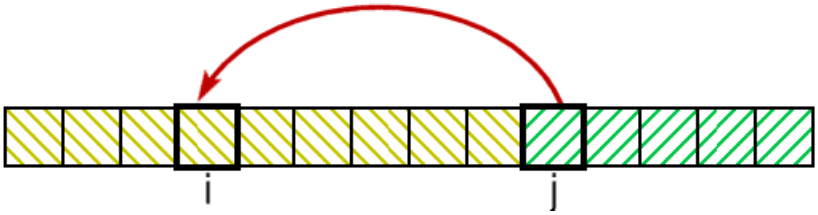
Problema: Ordenar o vector x[a..d], sabendo que os subvectores x[a..b] e x[c..d], com b+1=c, estão já ordenados.



Procedimento Pascal:

```
procedure fundir (a, b, c, d : integer);
var   i, j, k : integer;
      aux : elemento;

begin  i:= a;
       j:= c;
       repeat  (* Inserir x[j] *)
               if x[i] > x[j]
               then  begin  aux := x[j];
                            for  k:= j downto (i+1) do
                                 x[k]:= x[k-1];
                            x[i]:= aux;
                            j:= j+1
                     end
               else i:= i+1
       until (i>=j) or (j>d)
end (* fundir *);
```



Módulo Recorrente de Ordenamento por Fusão:

```
procedure OrdFusao (var x : vector; n, m : integer);
          var meio : integer;

          procedure fundir (a, b, c, d : integer);
          var  i, j, k : integer;
               aux : elemento;

          begin i:= a;
                j:= c;
                repeat (* Inserir x[j] *)
                       if x[i] > x[j]
                       then  begin aux := x[j];
                                   for k:= j downto (i+1) do
                                       x[k]:= x[k-1];
                                   x[i]:= aux;
                                   j:= j+1
                                   end
                       else i:= i+1
                until (i>=j) or (j>d)
          end (* fundir *);

begin if m-n >= 1
      then  begin meio:=(n+m) div 2;
                  OrdFusao(x, n, meio);
                  OrdFusao(x, meio+1, m);
                  fundir(n, meio, meio+1, m)
            end
end (* OrdFusao *);
```

A Recorrência está na origem daquele que é habitualmente considerado o mais rápido dos Métodos de Ordenamento ...

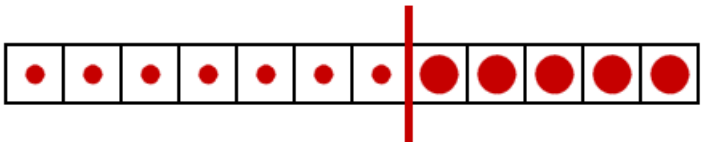
Ex17: **QuickSort** (C. A. R. Hoare, 1961)**:**

Observações:

- **Ao Ordenar um Vector, as Trocas mais produtivas são as efectuadas entre elementos mais afastados;**
- **Se o Vector inicial estiver por ordem inversa da pretendida, um bom Método de Ordenamento deverá realizar apenas n div 2 trocas.**
  (Ver Cap.V, pág.7)

- **A ideia fundamental do QuickSort, consiste em:**

Separar os elementos do vector dado, de modo que todos os "pequenos" fiquem na parte esquerda e todos os "grandes" na parte direita.



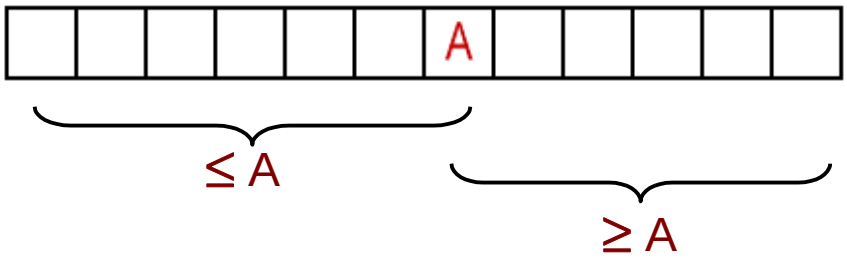
- **O Processo da Separação é então aplicado a cada uma das partes...**

Estratégia Recorrente:

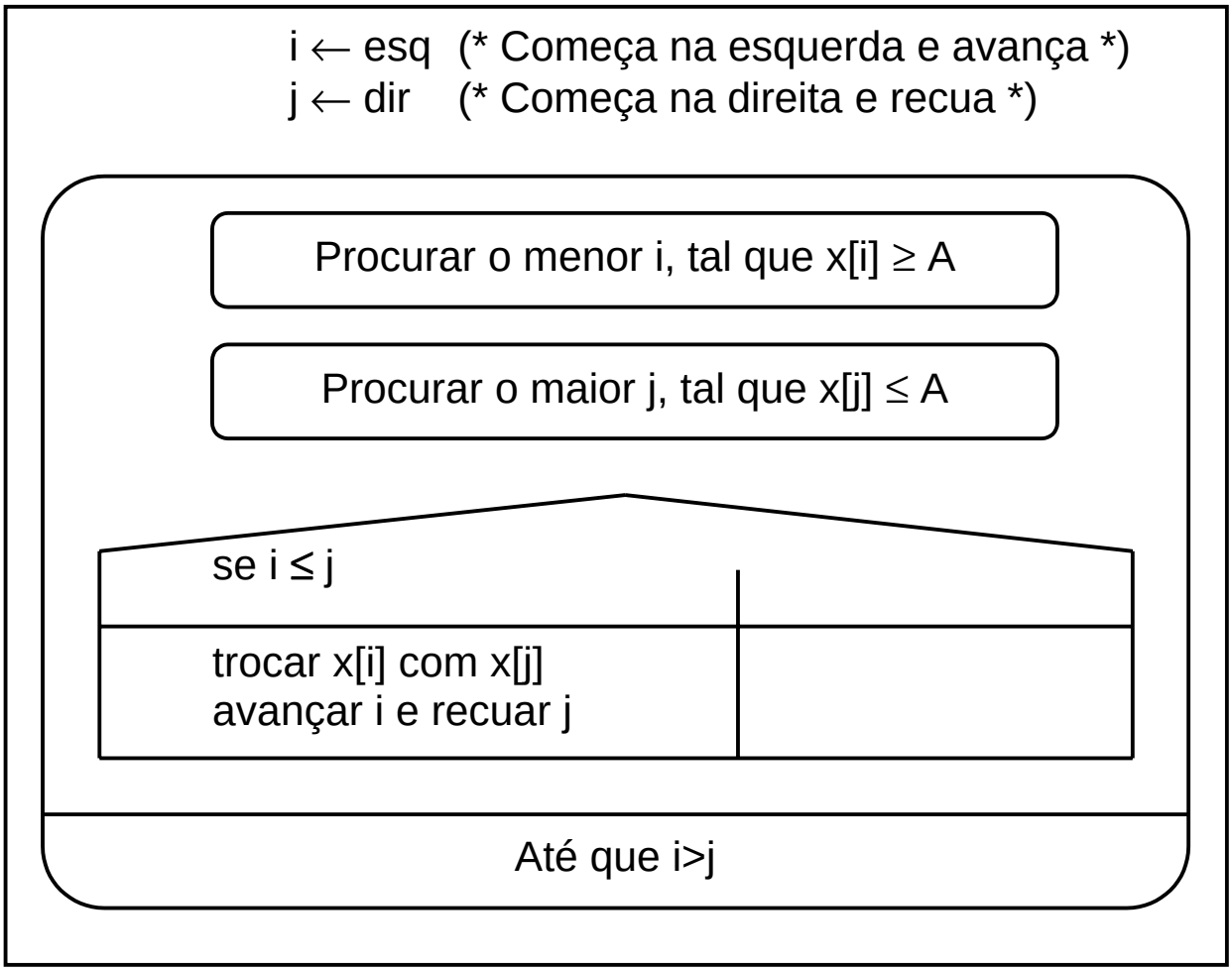
```
ordenar vector:
        Separar em pequenos e grandes;
        ordenar os pequenos;
        ordenar os grandes.
```

- **Resta assim estabelecer uma definição adequada para elementos "pequenos" e "grandes", bem como construir um Algoritmo para a Separação.**

Problema: **Dado um vector** x[1..n] **e um elemento arbitrário** A**, Separar os elementos de** x**, de modo que os inferiores fiquem à esquerda e os superiores à direita de** A**.**



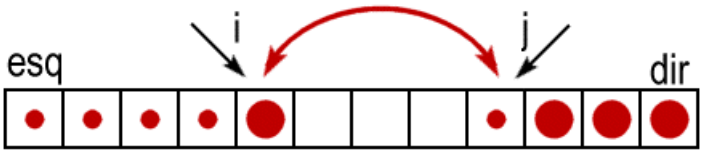
Algoritmo para a Separação do vector x[esq..dir]**:**



Algoritmo de Separação de Hoare, em Pascal, dados um vector x[esq..dir] e um elemento arbitrário A ∈ x[esq..dir]:

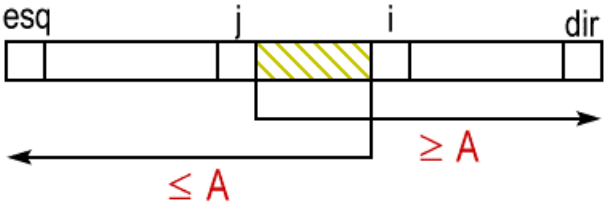
```
(* Inicializar i e j *)
i:= esq;
j:= dir;
repeat  (* Procurar o menor i, tal que x[i] ≥ A *)
        while x[i]<A do i:= i+1;
        (* Aqui x[i] ≥ A *)

        (* Procurar o maior j, tal que x[j] ≤ A *)
        while x[j]>A do j:= j-1;
        (* Aqui x[j] ≤ A *)
```




```
        if i<= j
        then  begin  (* Trocar x[i] com x[j] *)
                     aux := x[i];
                     x[i]:= x[j];
                     x[j]:= aux;
                     (* Avançar i e recuar j *)
                     i:= i+1;
                     j:= j-1
              end
until i>j;

(* Aqui x[esq..i-1] ≤ A and x[j+1..dir] ≥ A *)
```



Módulo Recorrente para o QuickSort:

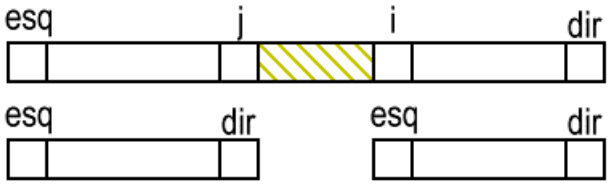
```
procedure  QuickSort (var x : vector; n : integer);

        procedure Sort (esq, dir : integer);
        var   i, j : integer;
              A, aux : elemento;

        begin  (* Fase da Separação *)
               i:= esq; j:= dir;
               (* O elemento Arbitrário pode ser o do meio *)
               A:= x[(esq+dir) div 2];

               repeat  while x[i]<A do i:= i+1;  (* Aqui x[i] ≥ A *)
                       while x[j]>A do j:= j-1;   (* Aqui x[j] ≤ A *)

                       if i<= j
                       then  begin  (* Trocar x[i] com x[j] *)
                                    aux :=x[i]; x[i]:=x[j]; x[j]:=aux;
                                    (* Avançar i e recuar j *)
                                    i:= i+1; j:= j-1
                             end
               until i>j;
               (* Aqui x[esq..i-1] ≤ A and x[j+1..dir] ≥ A *)
```

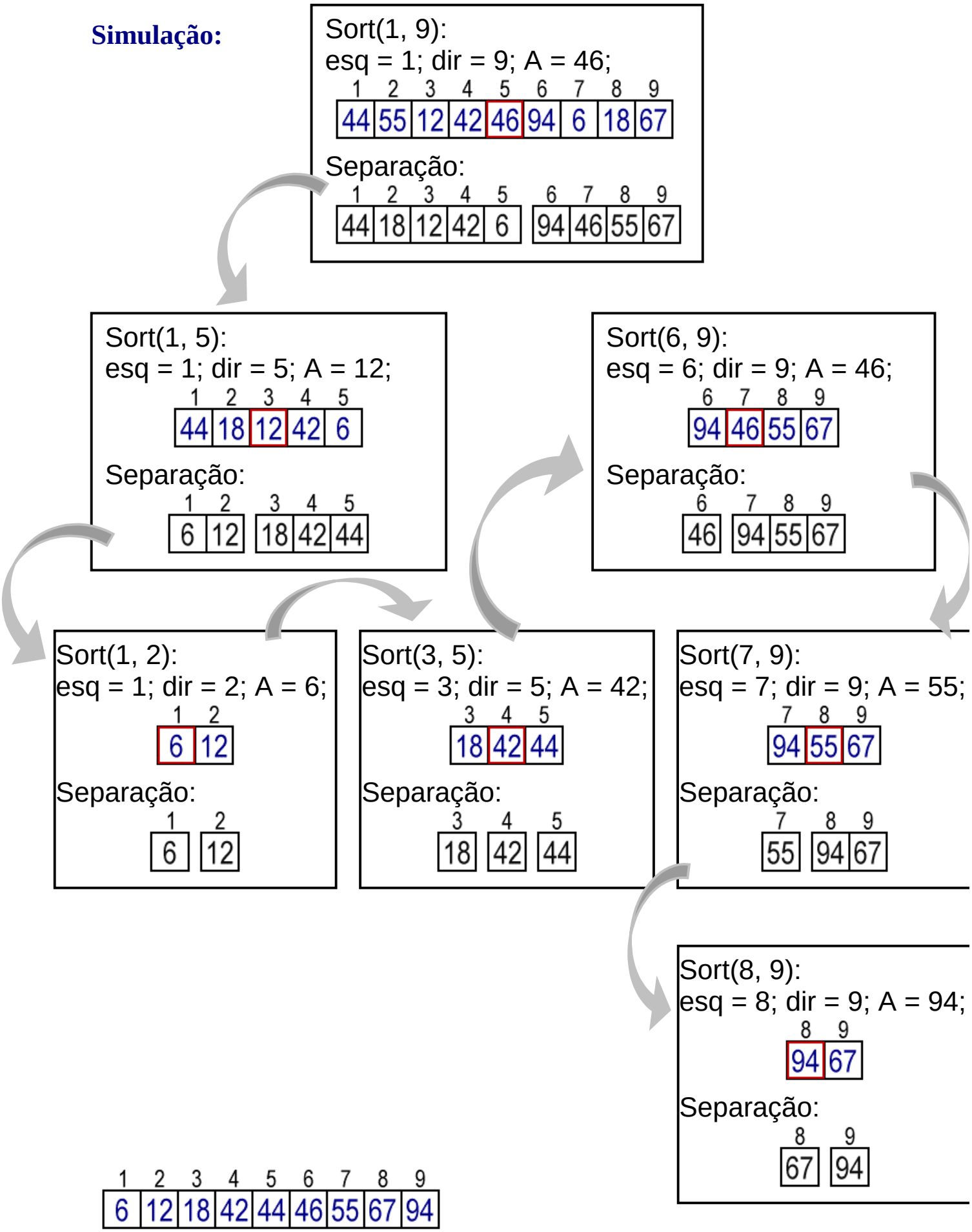



```
               (* Chamadas Recorrentes *)
               if esq < j  then Sort(esq, j);
               if i < dir    then Sort(i, dir)

        end (* Sort *);

begin  Sort(1, n)
end (*QuickSort *);
```

Simulação:


Sort(1, 9):
esq = 1; dir = 9; A = 46;
1 2 3 4 5 6 7 8 9
44 55 12 42 46 94 6 18 67
Separação:
1 2 3 4 5 6 7 8 9
44 18 12 42 6 94 46 55 67
Sort(1, 5):
esq = 1; dir = 5; A = 12;
1 2 3 4 5
44 18 12 42 6
Separação:
1 2 3 4 5
6 12 18 42 44
Sort(6, 9):
esq = 6; dir = 9; A = 46;
6 7 8 9
94 46 55 67
Separação:
6 7 8 9
46 94 55 67
Sort(1, 2):
esq = 1; dir = 2; A = 6;
1 2
6 12
Separação:
1 2
6 12
Sort(3, 5):
esq = 3; dir = 5; A = 42;
3 4 5
18 42 44
Separação:
3 4 5
18 42 44
Sort(7, 9):
esq = 7; dir = 9; A = 55;
7 8 9
94 55 67
Separação:
7 8 9
55 94 67
Sort(8, 9):
esq = 8; dir = 9; A = 94;
8 9
94 67
Separação:
8 9
67 94
1 2 3 4 5 6 7 8 9
6 12 18 42 44 46 55 67 94

- **Uma Análise da Complexidade Teórica, demonstra que o QuickSort é efectivamente o mais eficiente dos Métodos de Ordenamento;**

- **Um Estudo Estatístico mostra que essa propriedade se manifesta, muito em particular em vectores de grandes dimensões, bem como nos casos mais "difíceis";**

- **Por outro lado nos casos "fáceis", como no caso do vector dado já se encontrar (ou quase) ordenado, o QuickSort realiza operações inúteis (troca de elementos por si próprios);**

- **Para vectores pequenos e/ou quase ordenado, os métodos mais simples são os mais convenientes;**

- **Consideremos o seguinte "melhoramento" do QuickSort, destinado a evitar o caso da troca de um elemento por si próprio:**

```
if i<= j
then  begin  if i <> j
                    then  begin  (* Trocar x[i] com x[j] *)
                                 aux := x[i];
                                 x[i]:= x[j];
                                 x[j]:= aux;
                          end;
                    (* Avançar i e recuar j *)
                    i:= i+1;
                    j:= j-1
             end
```

- **Note-se contudo que, para evitar alguns (raros) casos de trocas inúteis, foi necessário introduzir uma nova comparação.**

- **Com esta alteração, o QuickSort realiza efectivamente $\frac{n}{2}$ trocas, no caso do vector dado se encontrar por ordem inversa. Verifique.**

- **O algoritmo da Separação de Hoare, só por si, pode ainda fornecer uma solução simples para outros tipos de problemas.**

  Por exemplo: **Dado um vector de elementos inteiros e positivos, separar os números pares dos ímpares, indicando quantos existem de cada.**

Procedimento Pascal:

```
procedure SepararParesImpares (var x : vector; n : integer);
var  i, j : integer;
     aux : integer;

begin  i:= 1; j:= n;
       repeat  while odd(x[i]) do i:= i+1;
               (* x[1..i-1] impares and x[i] par *)
               while not odd(x[j]) do j:= j-1;
               (* x[j] impar and x[j+1..n] pares *)
               if i<=j
               then  begin  aux :=x[i]; x[i]:=x[j]; x[j]:=aux;
                            i:= i+1; j:= j-1
                     end
       until i>j;
       (* x[1..i-1] impares and x[j+1..n] pares *)

       writeln('Existem ', i-1, ' números ímpares e ', n-j, ' pares.')
end (* SepararParesImpares *);
```

Ex.:

1	2	3	4	5	6	7	8	9	10	11	12
2	3	4	5	6	7	8	9	10	11	12	14

Existem 5 números ímpares e 7 pares.

1	2	3	4	5	6	7	8	9	10	11	12
11	3	9	5	7	6	8	4	10	2	12	14
				j	i						