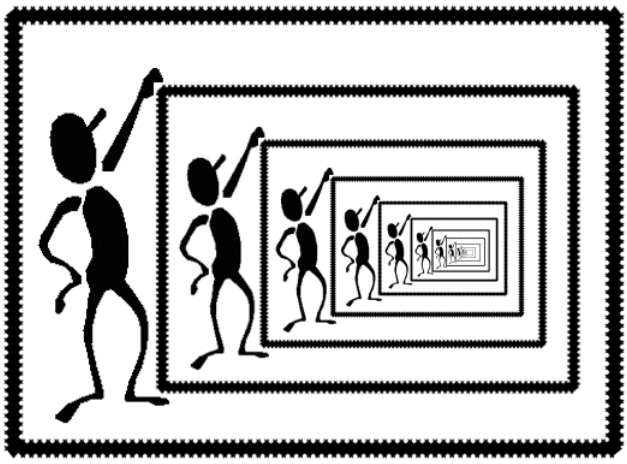
Capítulo 3 : Algoritmos Recorrentes

{ Quando algo é definido em termos de si próprio }



❒ A Utilização da Recorrência

A Função Factorial:

Definição Recorrente:

$$n! = \begin{cases} n \times (n-1)! & \text{se } n > 0 \\ 1 & \text{se } n = 0 \end{cases}$$

Algoritmo Recorrente:

$\forall n \in \mathbb{N}_0$

se n = 0
então n! ← 1
senão n! ← n * (n-1)!

Função Pascal Recorrente:

```
function  factorial (n : intnneg) : intnneg;
begin  if n = 0
       then  factorial := 1
       else  factorial := n * factorial(n-1)
end;
```

Atribuição do Valor ao Identificador da Função (sem o Parâmetro)

Chamada da Função numa Expressão (com o Parâmetro)

Simulação:

factorial(4)
n = 4
factorial ← 4 ∗ **factorial(3)**
n = 3
factorial ← 3 ∗ **factorial(2)**
n=2
factorial ← 2 ∗ **factorial(1)**
n = 1
factorial ← 1 ∗ **factorial(0)**
n = 0
factorial ← **1**
factorial ← 1 ∗ **1** = 1
factorial ← 2 ∗ **1** = 2
factorial ← 3 ∗ **2** = 6
factorial ← 4 ∗ **6** = 24

Comentários:

- Número de Chamadas da Função: n+1
  Número de Multiplicações: n
- Espaço de Memória utilizado:
  Cada Chamada gera uma cópia do Parâmetro n (da Classe Valor);
  O Valor de factorial, calculado por cada Chamada da Função, ocupa um Registo de Memória;
  Total: 2 (n+1)

A utilização da Recorrência no cálculo da Função Factorial:

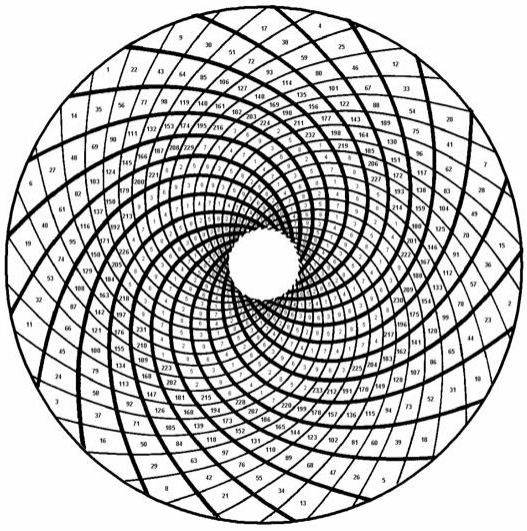
- Conduz a uma solução simples e elegante;
- É pouco eficiente, em particular, em termos da Memória utilizada.

Ainda menos eficiente ...

A Sucessão de Fibonacci

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

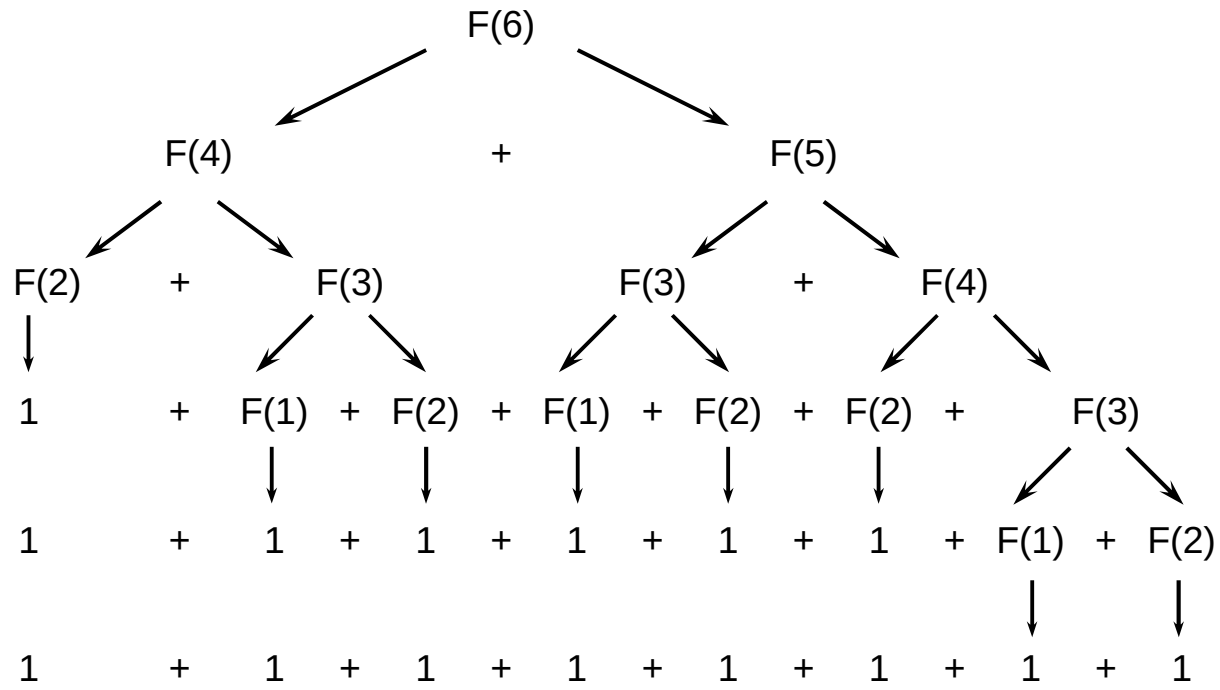
$$F_n = \begin{cases} F_{n-2} + F_{n-1} & \text{se } n > 2 \\ F_1 = F_2 = 1 \end{cases}$$



Função Pascal Recorrente:

```
function fibonacci (n : intpos) : intpos;
         begin if n <= 2
                  then fibonacci:= 1
                  else fibonacci:= fibonacci(n-2) + fibonacci(n-1)
         end;
```

Simulação: F(6) = 8



- Solução simples, clara e elegante;
- Mesmo nada eficiente, nem em termos das Operações realizadas, nem do número de Chamadas, nem da utilização de Memória;
- A definição recorrente pode contudo facilitar a elaboração de uma solução iterativa eficiente.

Exercício: Escrever a versão iterativa, só com 3 variáveis.

- A versão recorrente de um Algoritmo é, por vezes, mais clara e concisa do que a correspondente versão iterativa.

Algoritmo de Euclides

O Algoritmo de Euclides, para o cálculo do Máximo Divisor Comum de dois números inteiros não-negativos, baseia-se nas seguintes Propriedades:

$$mdc(a,b) = mdc(b, a \bmod b) \quad \text{se } b \neq 0$$
$$mdc(a,0) = a$$

Função Pascal Recorrente:

```
function  mdc (a, b : intnneg) : intnneg;
          begin  if b = 0
                 then  mdc:= a
                 else  mdc:= mdc(b, a mod b)
          end;
```

- A solução Recorrente é mais clara do que a versão Iterativa e mais fácil de elaborar, directamente a partir da teoria;
  A perda de eficiência é, neste caso, compensada pela Clareza.

Exercício: Na sua versão original, o Algoritmo de Euclides tinha a seguinte formulação:

"Para calcular o máximo divisor comum de dois números, subtraia o menor ao maior, até ficarem iguais."

Construa a versão Recorrente deste Algoritmo
e a correspondente função em Pascal.

- O modo de utilização de Memória das soluções Recorrentes pode tornar-se bastante útil ...

Inverter uma palavra:

{ Ler uma palavra, terminada por um espaço,
e escrever as suas letras por ordem inversa }

Algoritmo Iterativo:

ler cada letra, até espaço, registando num vector;
escrever letras por ordem inversa.

'R'	'O'	'M'	'A'	' '									...

(Desperdício de espaço de Memória,
provocado pela declaração do vector)

Algoritmo Recorrente:

ler uma letra;
se letra ≠ espaço
então Inverter o resto da palavra;
escrever letra.

Procedimento Pascal Recorrente:

```
procedure inverter;
          var letra : char;
          begin read(letra);
                if letra <> ' '
                then inverter;
                write(letra:1)
          end;
```

Simulação:

```
inverter:  ler('R');
           inverter: ler('O');
                     inverter: ler('M');
                               inverter: ler('A');
                                         inverter: ler(' ');
                                                   escrever(' ');
                                         escrever('A');
                               escrever('M');
                     escrever('O');
           escrever('R');
```

- Espaço de Memória utilizado: # letras + 1
  Cada chamada gera uma cópia da variável local letra, onde são registadas as sucessivas letras da palavra. Neste caso, a versão recorrente é mais eficiente, em termos de utilização de memória.

Inverter um número:

Escrever, por ordem inversa, os dígitos de um dado inteiro positivo.

Algoritmo Recorrente:

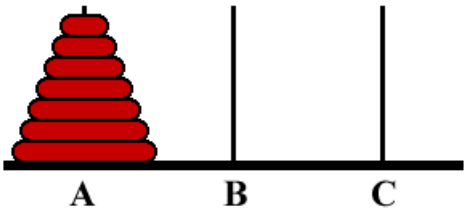
```
escrever o último dígito (unidades);
se restarem dígitos
então Inverter o resto do número.
```

Procedimento Pascal Recorrente:

```
procedure inverter(numero : intpos);
          begin write(numero mod 10 : 1);
                if (numero div 10) <> 0
                then inverter(numero div 10)
          end;
```

- A solução recorrente de um problema é, por vezes, a mais fácil ...

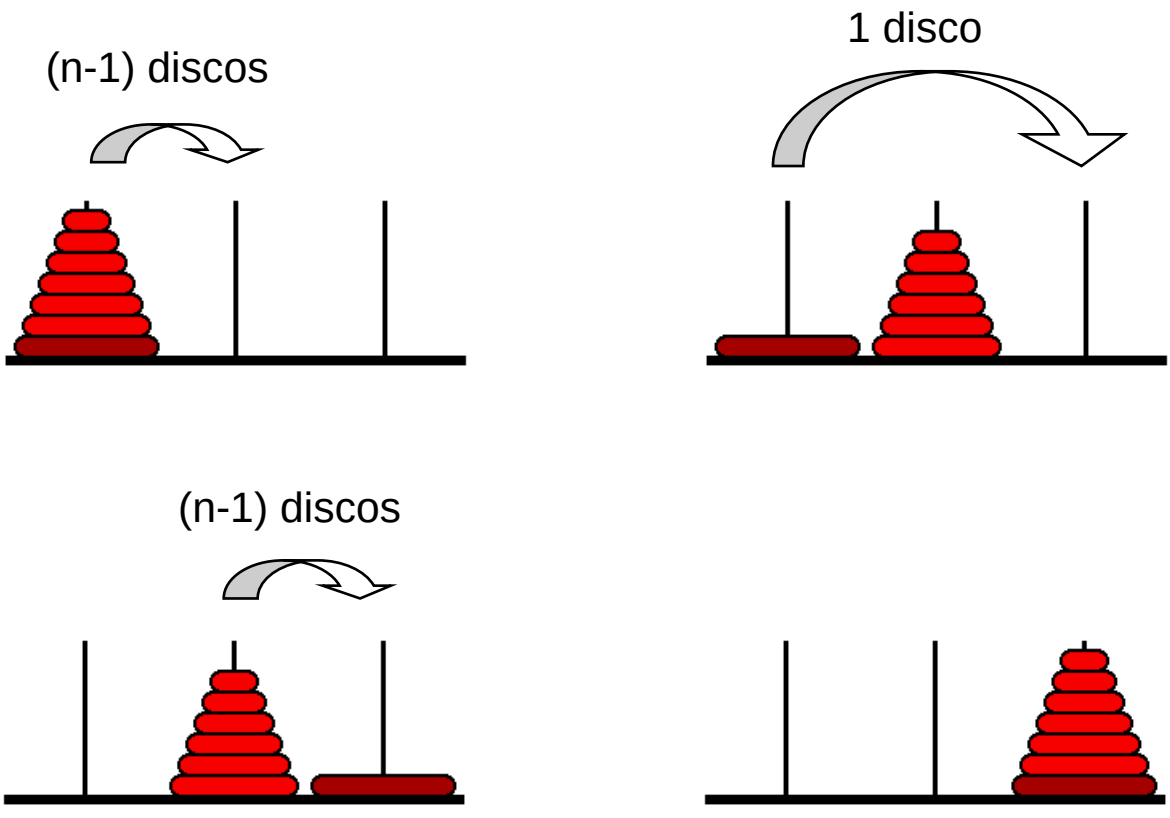
Torres de Hanoi (Edouard Lucas, 1883)



Deslocar todos os discos da Torre A para a Torre C, um de cada vez.
Utilizar a Torre B como auxiliar,
mas nunca colocar nenhum disco, sobre outro menor.

Algoritmo Recorrente:

- n = 1 : sabemos deslocar 1 disco;
- Como deslocaríamos n discos, se soubéssemos deslocar n - 1 discos?



Portanto:

Mover n discos de A para C, usando B como auxiliar:
Mover n-1 discos de A para B, usando C como auxiliar;
Mover 1 disco de A para C;
Mover n-1 discos de B para C, usando A como auxiliar.

Ou, melhor:

MoverTorre(n, origem, auxiliar, destino):
MoverTorre(n-1, origem, destino, auxiliar);
MoverDisco(origem, destino);
MoverTorre(n-1, auxiliar, origem, destino).

Programa Pascal:

```
program TorresdeHanoi(input, output);
type  torre = 'A'..'C';
      intnneg = 0..maxint;
var n : intnneg;

procedure  MoverDisco(desde, para : torre);
           begin writeln(desde:1, '->', para:1)
           end;

procedure  MoverTorre(n : intnneg; origem, auxiliar, destino : torre);
           begin if n>1
                 then begin  MoverTorre(n-1, origem, destino, auxiliar);
                             MoverDisco(origem, destino);
                             MoverTorre(n-1, auxiliar, origem, destino)
                      end
                 else MoverDisco(origem, destino)
           end;

begin writeln('Quantos discos tem esta Torre de Hanoi?');
      readln(n);
      MoverTorre(n, 'A', 'B', 'C')
end.
```

Simulação:

MoverTorre(3, A, B, C):

MoverTorre(2, A, C, B):

MoverTorre(1, A, B, C):

MoverDisco(A, C);

MoverDisco(A, B);

MoverTorre(1, C, A, B):

MoverDisco(C, B);

MoverDisco(A, C);

MoverTorre(2, B, A, C):

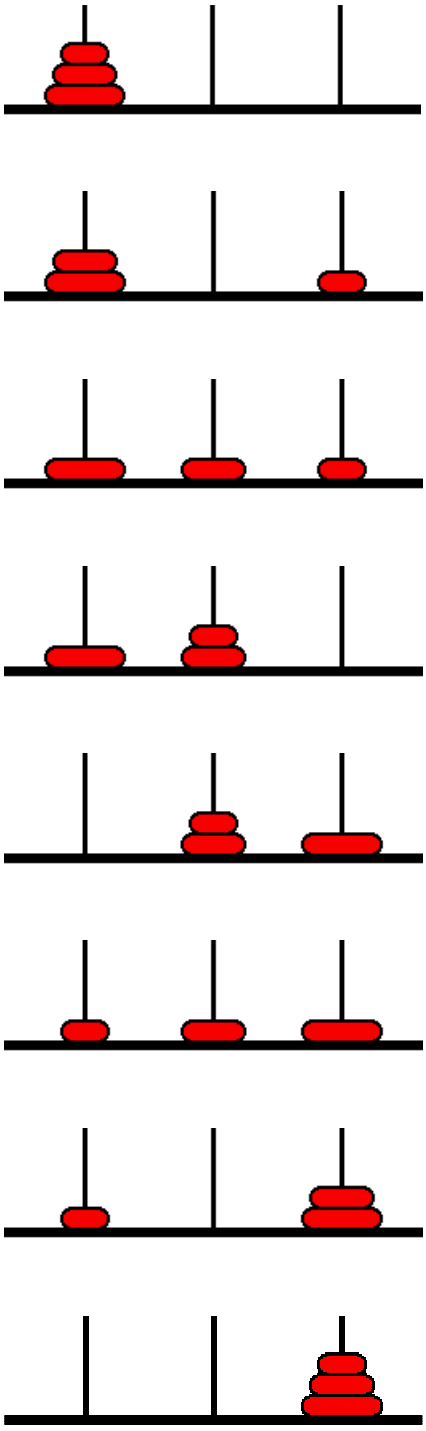
MoverTorre(1, B, C, A):

MoverDisco(B, A);

MoverDisco(B, C);

MoverTorre(1, A, B, C):

MoverDisco(A, C);



Teorema: O número de Movimentos simples necessários para mover uma Torre de Hanoi com n discos:

$$M(n) = 2^n - 1$$

Demonstrar: (Por Indução ...)

Questões associadas à "lenda" das Torres de Hanoi:

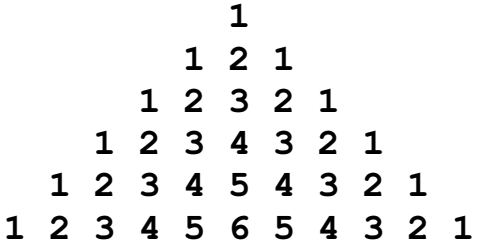
- Se um monge budista demorar um segundo para deslocar um disco, quanto tempo será necessário para mover a Torre completa (64 discos)?
- Qual o valor estimado para a esperança de vida do Sol?
- Construir a solução iterativa do problema das Torres de Hanoi.

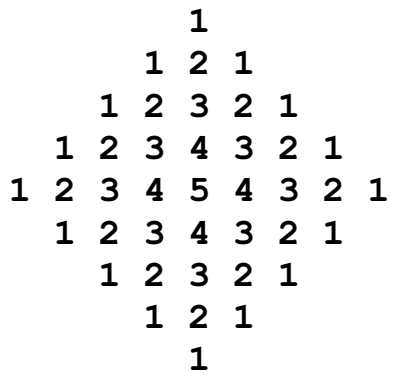
A Construção de Soluções Recorrentes:

1. Como definir o problema em termos de um problema menor e do mesmo tipo?
2. De que modo diminui o tamanho do problema, em cada chamada recorrente?
3. Qual o caso particular do problema, que pode servir para a paragem? (Verificar se o modo como diminui o tamanho do problema, em cada chamada, assegura que o caso base é sempre atingido).

Exercícios:

- Elaborar soluções puramente recorrentes para construir cada uma das figuras seguintes (não utilizar nenhum ciclo).

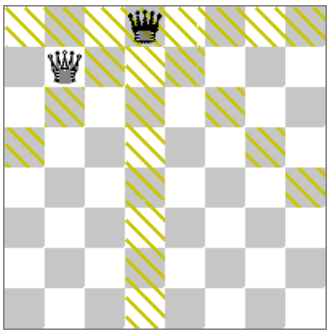




- Em certos casos, quando não é **conhecido** nenhum algoritmo iterativo, é necessário gerar um processo recorrente de Pesquisa Exaustiva (Backtracking).

O Problema da 8 Rainhas:

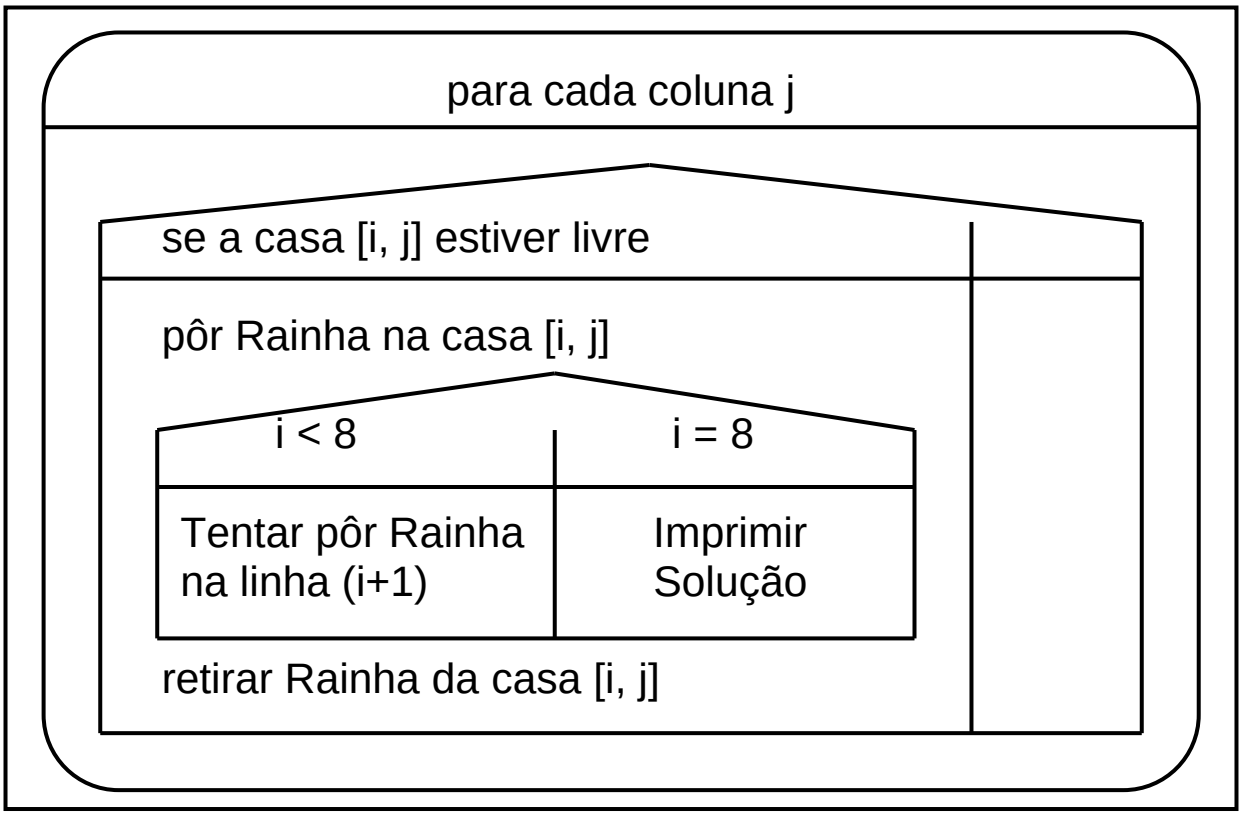
Num tabuleiro de Xadrez, a Rainha domina toda a linha, a coluna e as duas diagonais da casa onde estiver colocada.



Como colocar 8 Rainhas (não dominadas) num tabuleiro? (Existem 92 soluções possíveis)

Processo de Backtracking:

Tentar pôr Rainha na linha (i):



Como assinalar as casas que estão, ou não, livres?

Consideremos um conjunto de vectores do tipo boolean, onde:

true ⇒ casa livre
false ⇒ casa dominada

Para as colunas:

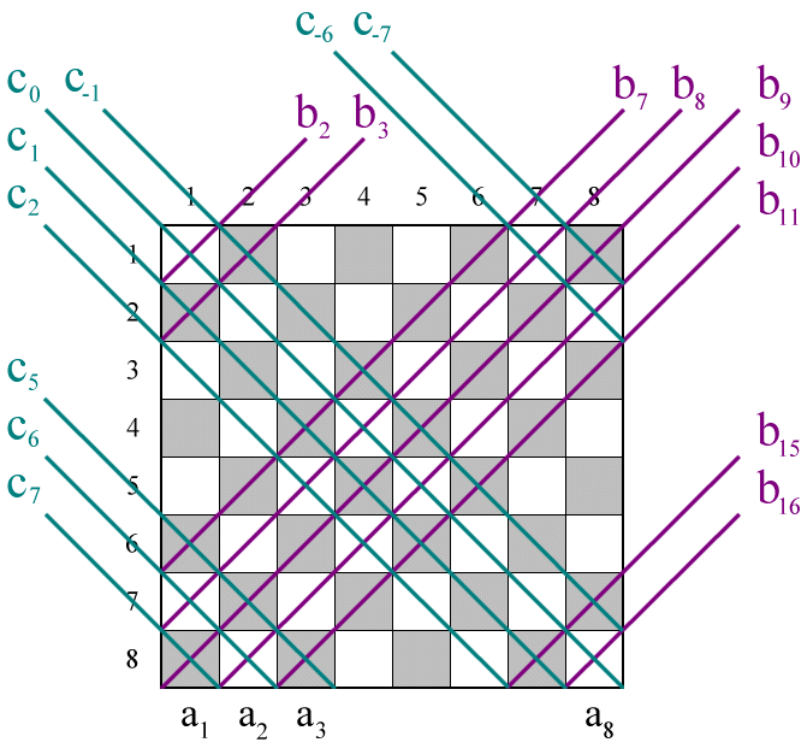
var a : **array** [1..8] of boolean;

Para as diagonais ascendentes (i+j):

var b : **array** [2..16] of boolean;

Para as diagonais descendentes (i-j):

var c : **array** [-7..7] of boolean;



O processo tenta sucessivamente colocar uma Rainha em cada linha.

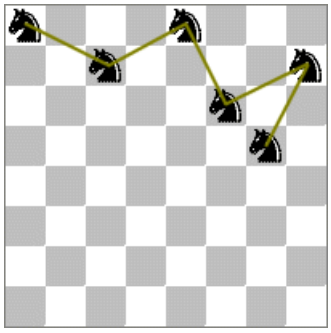
Para registar a posição (coluna) ocupada por cada Rainha basta:

var x : **array** [1..8] of 1..8;

```
program OitoRainhas(output);

type linha, coluna = 1..8;
var a : array [coluna] of boolean;
var b : array [2..16] of boolean;
var c : array [-7..7] of boolean;
var x : array [linha] of coluna;
var  i : integer;
procedure   EscreverSolucao;
            begin    ... { Como? } ...
            end;

procedure  Tentar(i : linha);
            var j : coluna;
            begin  for  j:= 1 to 8 do
                        { Se a casa [i, j] estiver livre }
                        if a[j] and b[i+j] and c[i-j]
                        then  begin { Pôr Rainha em [i, j] }
                                    x[i]:= j;
                                    a[j]:= false;
                                    b[i+j]:= false;
                                    c[i-j]:= false;
                                    if i < 8
                                    then Tentar(i+1)
                                    else EscreverSolucao;
                                    { Retirar Rainha de [i, j] }
                                    x[i]:= 0;
                                    a[j]:= true;
                                    b[i+j]:= true;
                                    c[i-j]:= true
                              end
            end { Tentar };

begin   for i:= 1   to  8    do    a[i]:= true;
        for i:= 2   to  16   do    b[i]:= true;
        for i:= -7  to  7    do    c[i]:= true;
        for i:= 1   to  8    do    x[i]:= 0;
        Tentar(1)
end.
```

Exercício: Circuito de Cavalo

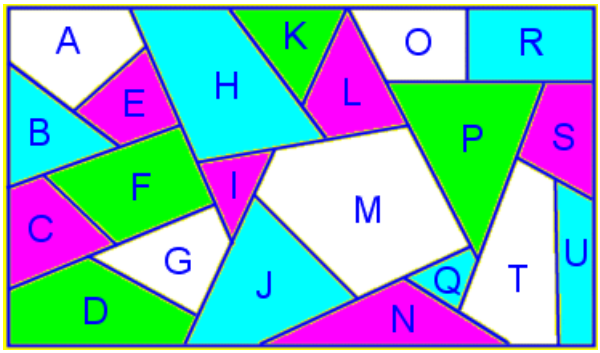
Outro problema clássico associado ao Xadrez consiste em, partindo de uma dada casa e sempre em "Salto de Cavalo", percorrer cada uma das casas do tabuleiro uma só vez, acabando na casa inicial.



Construa uma abordagem do tipo "Backtracking" para este problema.

Exercício: Coloração de Mapas

É sabido, e está já demonstrado, que **quatro cores** são suficientes para colorir qualquer mapa.
Contudo, o processo de **colorir** um dado mapa com quatro cores é um Problema que se demonstra ser **difícil**.



Construa uma abordagem do tipo "Backtracking" para este problema.

- Os Métodos de Pesquisa também admitem Abordagens Recorrentes ...

Pesquisa Linear num Vector (não ordenado):

Uma Estratégia Recorrente:

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

Mas,

- É necessário garantir que o processo **pára** mesmo que o elemento não seja encontrado, isto é, falta identificar o Caso Base da Recorrência.

Algoritmo Recorrente:

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

Contudo,

- Não convém 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 };
```

Pesquisa Binária num Vector ordenado:

Algoritmo Recorrente:

```
procurar  em x[1 .. n]:
          se n ≥ 1
          então  se este = x[meio]
                 então achou
                 senão   se este < x[meio]
                         então  procurar em x[1 .. meio - 1]
                         senão  procurar em x[meio + 1 .. n]
```

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 };
```

❒ A Complexidade dos Algoritmos Recorrentes

• Análise do Pior dos Casos

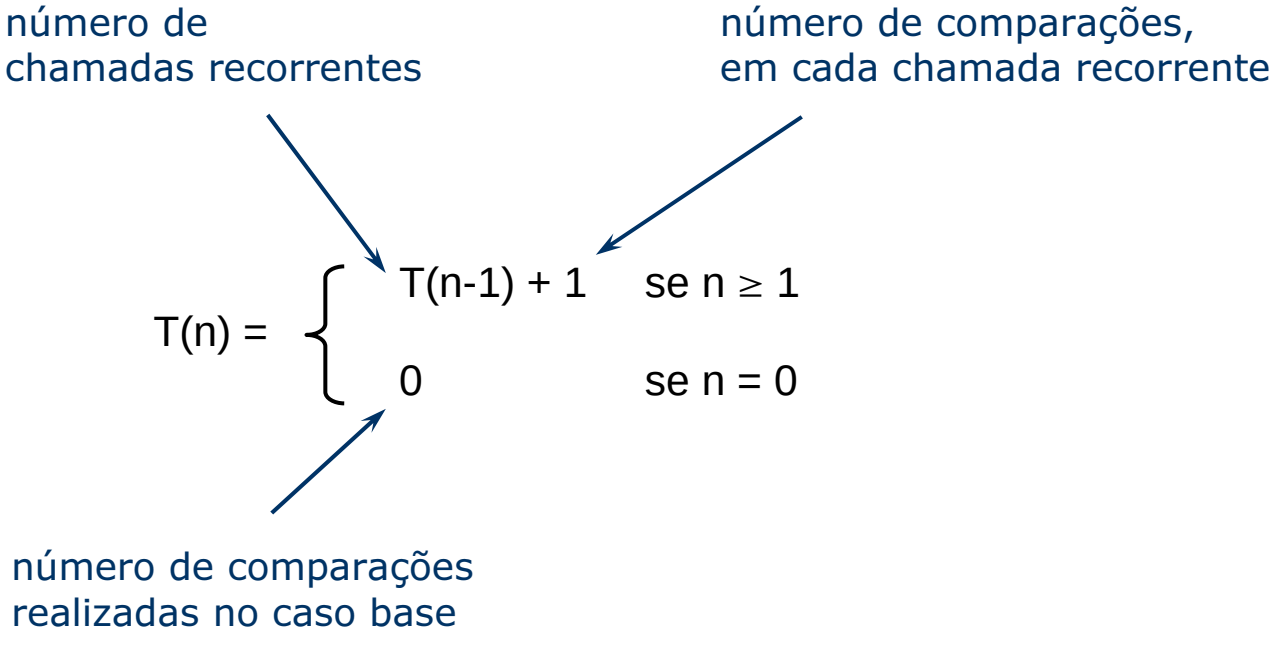
Neste Capítulo vamos analisar apenas o Pior dos Casos da Complexidade de Algoritmos Recorrentes.

Consideremos o caso da **Pesquisa Sequencial**, num vector x[1..n],

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

{ O pior dos casos corresponde às situações este = x[n] ou este ∉ x[1..n], quando todos os elementos são analisados }

Seja T(n) o número de **comparações** necessárias, para realizar uma pesquisa em n elementos.
Este valor vai ser calculado em função de T(n-1), as comparações necessárias, para realizar uma pesquisa em n-1 elementos.



Calculemos,

$$
\begin{aligned}
T(n) &= T(n-1) + 1 \\
&= (T(n-2) + 1) + 1 && = T(n-2) + 2 \\
&= (T(n-3) + 1) + 2 && = T(n-3) + 3 \\
&\ldots \\
T(n) &= T(n - k) + k && \text{para } k = 1, 2, \ldots, n
\end{aligned}
$$

no Pior dos Casos, o processo pára quando $n - k = 0$, ou $k = n$, onde

$$T(n) = T(0) + n = n$$

e, como seria de prever, o número de comparações efectuadas é:

$$T(n) = n.$$

- Uma análise equivalente consistiria em considerar $n = 1$ (a última chamada efectuada) o caso base do processo.

  { Para alguns algoritmos, esta é a abordagem mais conveniente }

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

  Verifique que o resultado obtido é o mesmo.

A complexidade da **Pesquisa Binária**, num vector ordenado x[1..n],

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

Seja T(n) o número de comparações efectuadas, para n elementos. Denotemos por a o número (constante) de comparações realizadas em cada chamada.

$$T(n) = \begin{cases} T(n/2) + a & \text{se } n > 1 \\ a & \text{se } n = 1 \end{cases}$$

Calculemos,

$$T(n) = T(n/2) + a$$
$$= (T(n/2^2) + a) + a \qquad = T(n/2^2) + 2\,a$$
$$= (T(n/2^3) + a) + 2\,a \qquad = T(n/2^3) + 3\,a$$
$$\ldots$$
$$T(n) = T(n/2^k) + k\,a$$

No Pior dos Casos, o processo pára quando $n / 2^k = 1$, ou $k = \log_2 n$, onde $T(n / 2^k) = a$. Portanto,

$$T(n) = T(n/2^k) + k\,a \qquad \text{para } k = 1, 2, \ldots, \log_2 n$$

e quando $k = \log_2 n$,

$$T(n) = a + a \log_2 n = \mathcal{O}(\log_2 n).$$

O Problema das **Torres de Hanoi**, com n discos,

```
MoverTorre(n, origem, auxiliar, destino):
        MoverTorre(n - 1, origem, destino, auxiliar);
        MoverDisco(origem, destino);
        MoverTorre(n - 1, auxiliar, origem, destino).
```

{ O Algoritmo executa um número fixo de operações }

Seja T(n) o número de movimentos simples, necessários para deslocar uma torre com n discos.

$$T(n) = \begin{cases} 2\,T(n-1) + 1 & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

Calculemos,

$$T(n) = 2\,T(n-1) + 1$$

$$= 2\,(2\,T(n-2) + 1) + 1 = 2^2\,T(n-2) + 2 + 1$$

$$= 2^2\,(2\,T(n-3) + 1) + 2 + 1 = 2^3\,T(n-3) + 2^2 + 2 + 1$$

$$= 2^4\,T(n-4) + 2^3 + 2^2 + 2 + 1$$

...

$$T(n) = 2^k\,T(n-k) + 2^{k-1} + 2^{k-2} + \ldots + 2 + 1$$

$$T(n) = 2^k\,T(n-k) + 2^k - 1$$

$$T(n) = 2^k\,(T(n-k) + 1) - 1 \qquad \text{para } k = 1, 2, \ldots, n-1$$

o processo pára quando $k = n - 1$, onde

$$T(n) = 2^{n-1}\,(T(1) + 1) - 1 = 2^n - 1$$

são portanto necessários $2^n - 1$ movimentos simples.

A generalização deste resultado permite detectar a geração de algoritmos de **Complexidade Exponencial**:

$$T(n) = \begin{cases} a\,T(n-c) + d & \text{se } n > 1 \\ b & \text{se } n = 1 \end{cases}$$

Representando um algoritmo com a chamadas recorrentes, onde a grandeza do problema diminui de c unidades, a cada chamada.

Calculemos,

$$\begin{aligned} T(n) &= a\,T(n-c) + d \\ &= a\,(a\,T(n-2c) + d) + d && = a^2\,T(n-2c) + ad + d \\ &= a^2\,(a\,T(n-3c) + d) + ad + d && = a^3\,T(n-3c) + a^2 d + ad + d \end{aligned}$$

...

$$T(n) = a^k\,T(n - kc) + \sum_{i=0}^{k-1} a^i\,d$$

No Pior dos Casos, o processo pára quando $n - k\,c = 1$, ou $k = \frac{n-1}{c}$, onde $T(n - k\,c) = b$.

Há duas situações a considerar:

se a = 1, (uma chamada recorrente)

$$T(n) = b + \frac{n-1}{c}\,d = \mathcal{O}(n)$$

então o algoritmo é Linear.

se a > 1, (a chamadas recorrentes)

$$T(n) = b\,a^{(n-1)/c} + d\,\frac{1-a^{(n-1)/c}}{1-a}$$

$$= \frac{d}{1-a} + (b + \frac{d}{1-a})\quad a^{(n-1)/c} = O(a^{n/c})$$

portanto, o algoritmo tem Complexidade Exponencial, onde a base da exponencial é o próprio número a de chamadas.

- A Abordagem Recorrente permite a construção de Métodos de Ordenamento muito eficientes:

❒ Métodos de Ordenamento Recorrentes

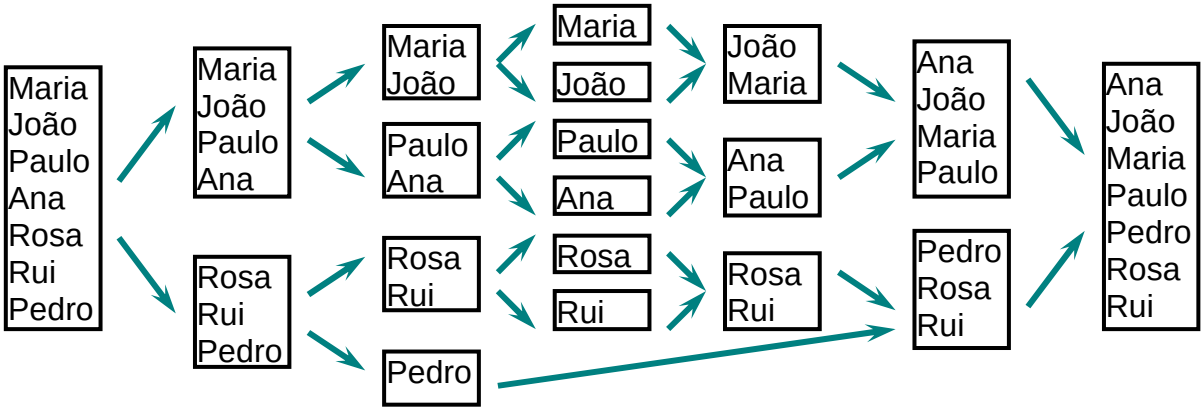
Ordenamento por Fusão:

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.

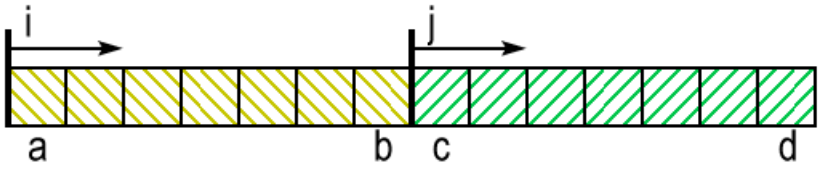
Estratégia Recorrente:

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

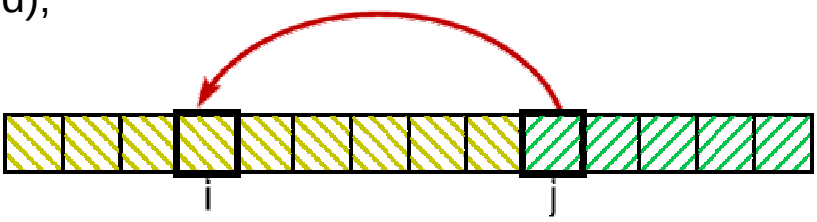


- Portanto, além das duas chamadas recorrentes, resta construir um módulo para a **Fusão** de dois Subvectores já ordenados.

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




```
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);
```



Módulo Recorrente de **Ordenamento por Fusão**:

```
{ Assumindo  var x : vector;  global ao módulo }

procedure OrdFusao (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(n, meio);
                  OrdFusao(meio + 1, m);
                  fundir(n, meio, meio+1, m)
            end
end { OrdFusao };
```

A Complexidade dos Algoritmos Recorrentes de Ordenamento

- Na Análise da Complexidade dos Métodos de Ordenamento devem ser analisadas as duas operações envolvidas: **Comparações** e **Trocas**, entre elementos do vector.

- Já vimos que o valor **mínimo** possível da Complexidade dos Métodos de Ordenamento é $\mathcal{O}(n \log_2 n)$.

A Complexidade do **Ordenamento por Fusão**:

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

Analisemos o número de **comparações** necessárias para ordenar um vector com n elementos. Assumindo que a operação de fusão é $\mathcal{O}(n)$,

$$T(n) = \begin{cases} 2\,T(n/2) + a\,n & \text{se } n > 1 \\ b & \text{se } n = 1 \end{cases}$$

Calculemos,

$$\begin{aligned} T(n) &= 2\,T(n/2) + a\,n \\ &= 2\,(2\,T(n/2^2) + a\,n/2) + a\,n \\ &= 2^2\,T(n/2^2) + 2\,a\,n \\ &= 2^2\,(2\,T(n/2^3) + a\,n/2^2) + 2\,a\,n \\ &= 2^3\,T(n/2^3) + 3\,a\,n \\ &= 2^4\,T(n/2^4) + 4\,a\,n \end{aligned}$$

...

$$T(n) = 2^k T(n / 2^k) + k\ a\ n$$

o processo pára quando $n / 2^k = 1$, ou $k = \log_2 n$, onde $T(n / 2^k) = b$.

$$T(n) = 2^k T(n / 2^k) + k\ a\ n \quad \text{para } k = 0, 1, 2, ..., \log_2 n$$

e quando $k = \log_2 n$,

$$T(n) = b + (\log_2 n)\ a\ n = \mathcal{O}(n \log_2 n).$$

- A generalização deste resultado abrange o cálculo do valor da complexidade de vários algoritmos recorrentes.

Teorema: (Aho + Hopcroft + Ullmann)

Sejam $a, b, c \in \mathbb{N}$ constantes. A fórmula recorrente,

$$T(n) = \begin{cases} a\ T(n/c) + b\ n & \text{se } n > 1 \\ b & \text{se } n = 1 \end{cases}$$

tem como solução:

$$T(n) = \begin{cases} \mathcal{O}(n) & \text{se } a < c \\ \mathcal{O}(n \log_c n) & \text{se } a = c \\ \mathcal{O}(n^{\log_c a}) & \text{se } a > c \end{cases}$$

Exercício: Demonstrar.

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

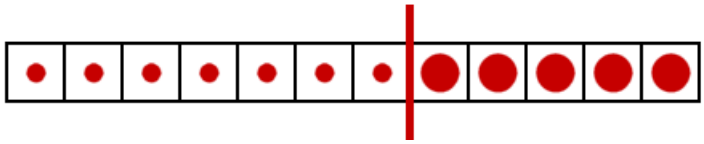
QuickSort (C. A. R. Hoare, 1961):

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 deveria realizar apenas n **div** 2 trocas.

Estratégia:

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



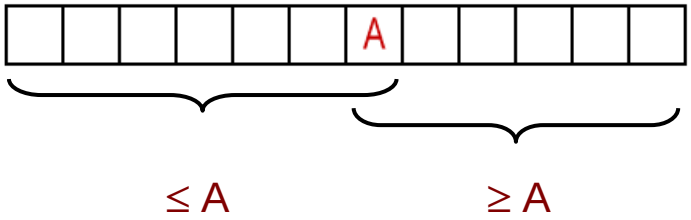
O Processo da Separação é então aplicado a cada uma das partes...

Algoritmo Recorrente:

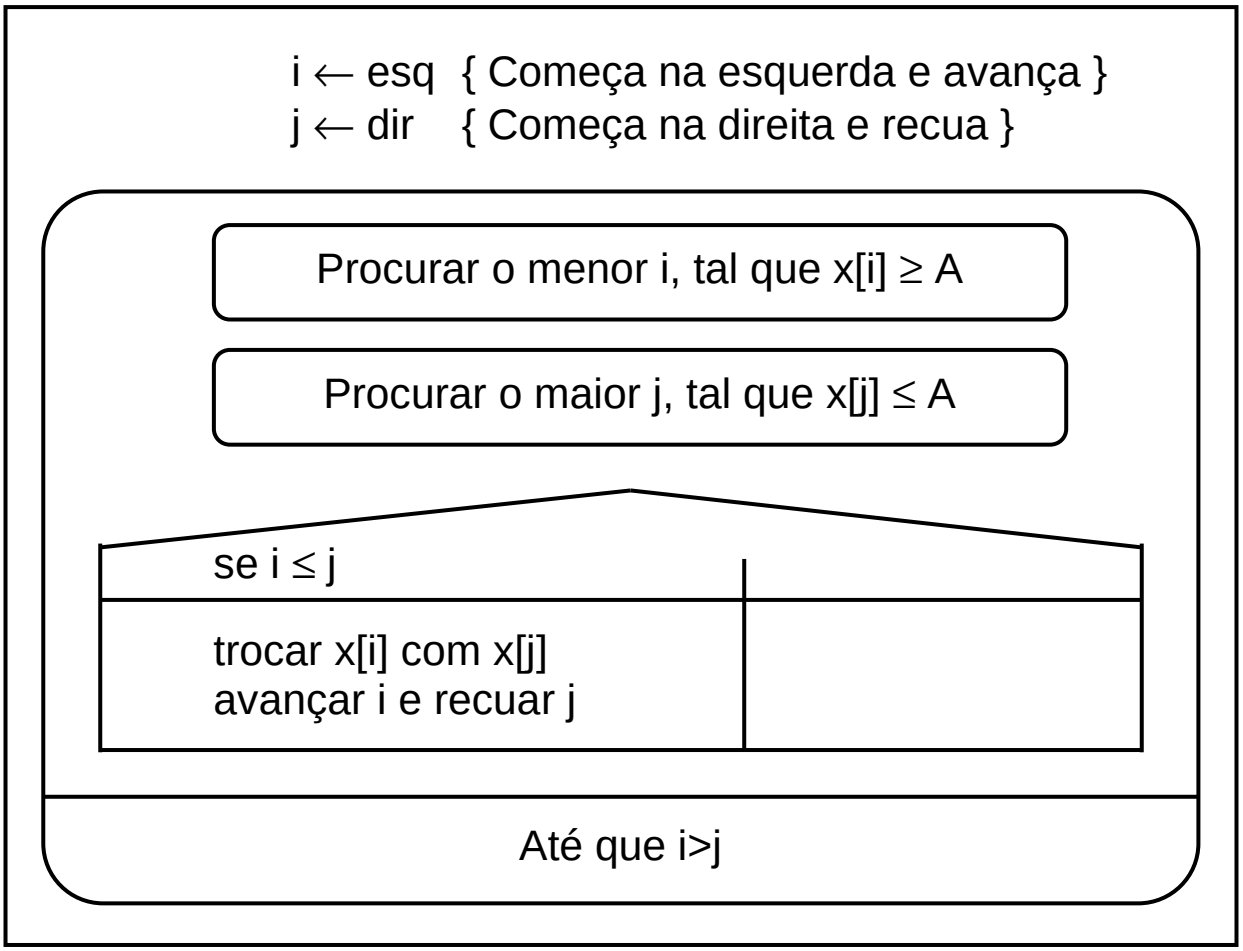
```
ordenar  vector:
         Separar em pequenos e grandes;
         ordenar os pequenos;
         ordenar os grandes.
```

- 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.

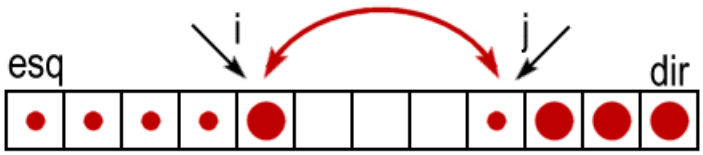


Algoritmo para a Separação do vector x[esq..dir]:

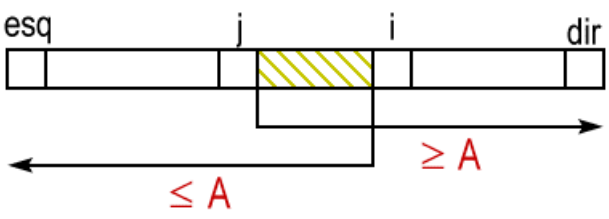


Algoritmo de Separação de Hoare, em Pascal, dados um vector x[esq..dir] e um elemento arbitrário $A \in$ x[esq..dir]:

```
{ 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 }
```

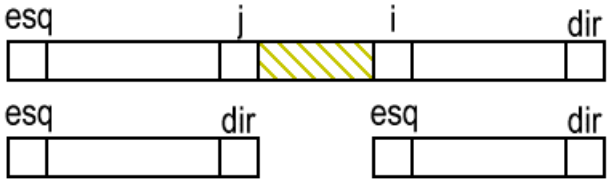



```
        if i<= j
        then  begin { Trocar x[i] com x[j] }
                    aux := x[i];
                    x[i]:= x[j];
                    x[j]:= aux;
                    { Avançar i e recuar j }
                    i:= i +1;
                    j:= j - 1
              end
until i > j;

{ 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

A Complexidade do QuickSort:

- O desempenho do QuickSort pode variar bastante, consoante a grandeza relativa dos dois sub-vectores "separados".

- No Melhor dos Casos, a separação resulta em duas "metades", em todas as chamadas. Assumindo que o processo da Separação é $\mathcal{O}(n)$,

$$T(n) = \begin{cases} 2\,T(n/2) + a\,n & \text{se } n > 1 \\ b & \text{se } n = 1 \end{cases}$$

  donde, pelo Teorema anterior, se obtém $\mathcal{O}(n \log_2 n)$.

- No Pior Caso possível, a cada chamada, o comprimento do sub-vector decresce apenas de uma unidade. O processo consiste na fase da Separação e apenas numa chamada recorrente,

$$T(n) = \begin{cases} 1\,T(n-1) + a\,n & \text{se } n > 1 \\ b & \text{se } n = 1 \end{cases}$$

  Portanto, no Pior dos Casos o QuickSort é $\mathcal{O}(n^2)$.

- Note-se também que o algoritmo efectua trocas desnecessárias como, por exemplo, no caso de o vector já estar (ou quase) ordenado.

- Prova-se que no Caso Médio é também $\mathcal{O}(n \log_2 n)$.

- No artigo original de C.A.R. Hoare, o elemento Arbitrário seleccionado era A:= x[esq]. Verifique que a complexidade do método é a mesma.
  Qual será, nessa versão, o vector que origina o Pior dos Casos?

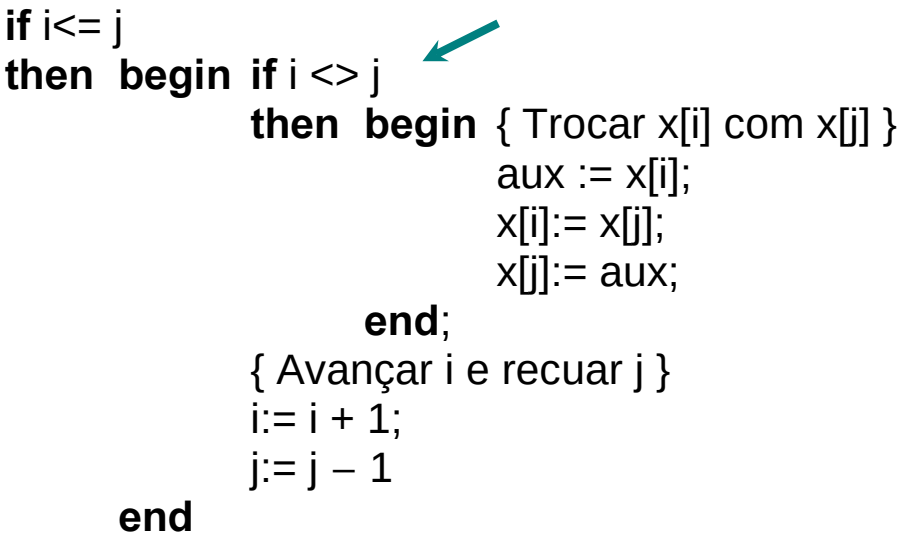
Conclusões:

- O QuickSort é, segundo a Análise da Complexidade teórica, o mais eficiente dos Métodos de Ordenamento; Uma Análise Estatística mostra que essa propriedade também se manifesta na prática, 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 n **div** 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 tipo.

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						

- Problemas cuja definição é recorrente, conduzem naturalmente a soluções recorrentes ...

Notação Polaca (Lukasiewicz, 1878-1956)

Na notação habitual (*infix*) das Expressões Aritméticas, cada operador é colocado entre os dois operandos correspondentes.

a + b * c

Para saber a ordem por que são efectuados os cálculos, é necessário estabelecer um conjunto de Regras de Prioridade entre os operadores e (para alterar essa ordem) é também necessário o uso de Parênteses.

- Na **Notação Polaca** (*postfix*), cada operador é colocado depois dos dois operandos correspondentes.

  A ordem dos cálculos a efectuar fica univocamente definida, sem Regras de Prioridade nem Parênteses.

Exemplos:

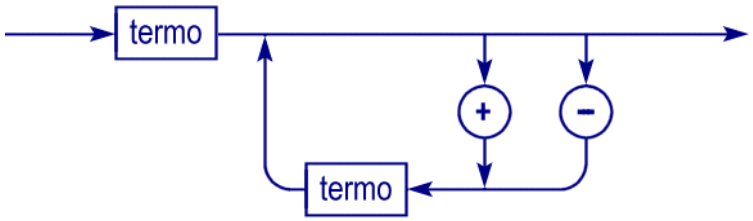
infix	*postfix*
a + b	a b +
(a + b) * c	a b + c *
(a + b) * (c - d)	a b + c d - *
a + b * c - d	a b c * + d -
a + b * (c - d)	a b c d - * +
(a + b) * c - d	a b + c * d -

Problema: Conversão *infix* → *postfix*

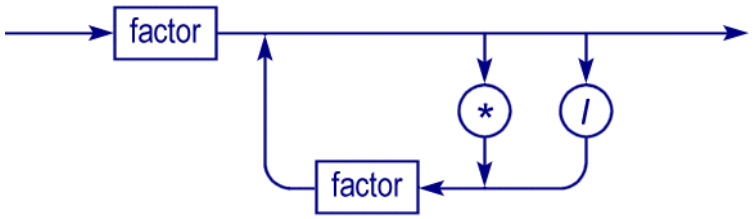
Vamos considerar "uma" Linguagem de Alto Nível, que consiste numa simplificação do Pascal.

Definições:

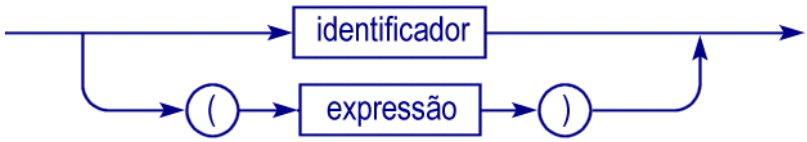
Expressão:



Termo:



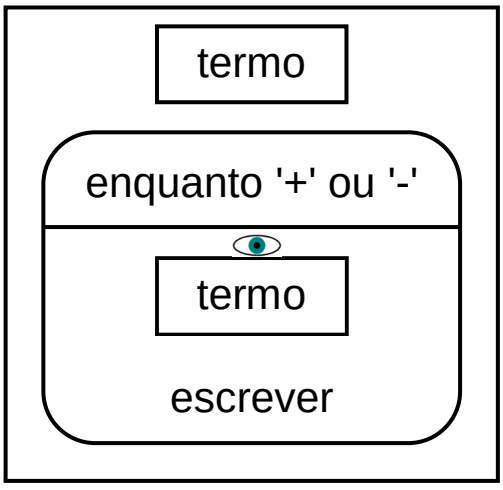
Factor:



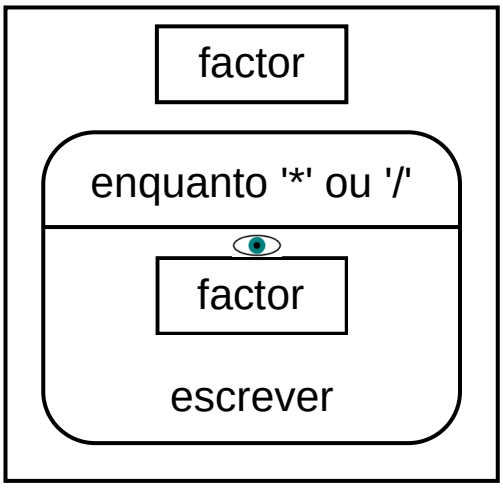
Identificador:

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

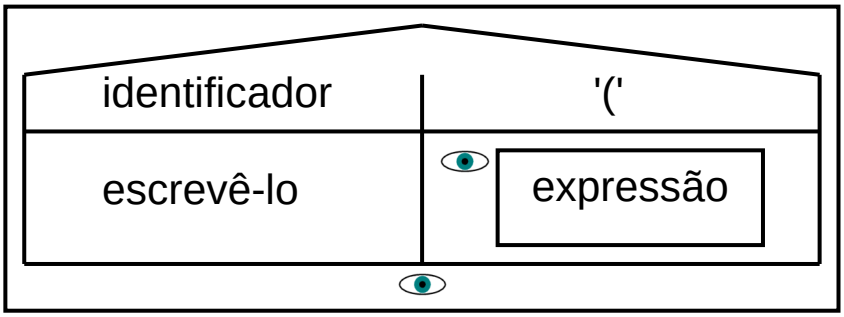
expressão



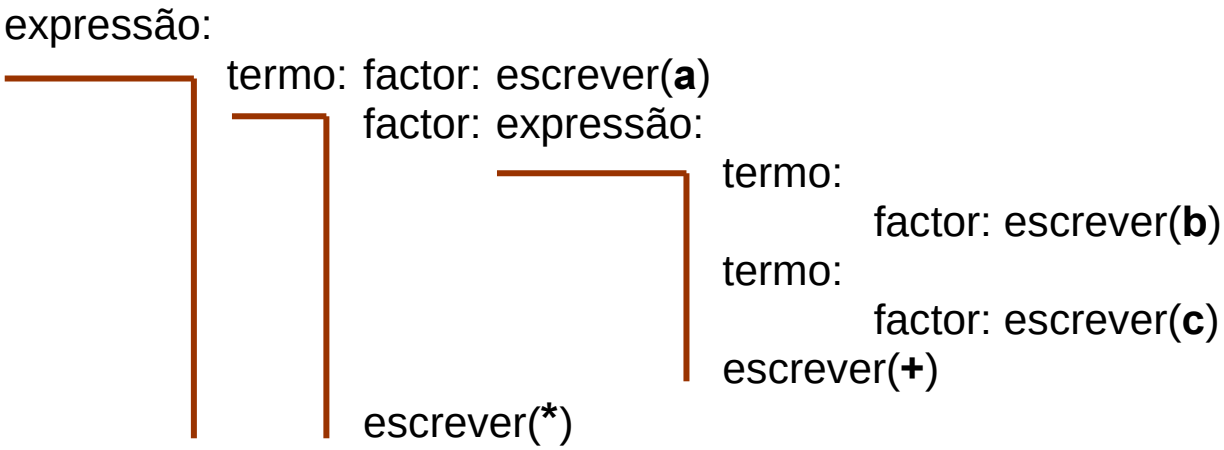
termo



factor



Exemplo: a * (b + c) → a b c + *



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*

Comentários:

- A Notação Polaca é utilizada na Compilação das Expressões Aritméticas de todas as Linguagens de Alto Nível.
- O código produzido pelos módulos recorrentes não é eficiente, não sendo portanto adequado à construção de Compiladores.
- Contudo, a partir da abordagem recorrente, é possível construir a correspondente versão iterativa.

Exercícios:

- Calcular o valor de uma expressão *postfix*.
- Conversão *postfix* $\rightarrow$ *infix*.