

□ Nota breve sobre cálculos de somas parciais

• Séries Aritméticas

São da forma,

$$\{a_k\}, k = 1, 2, 3, \dots \text{ com } a_{k+1} - a_k = d$$

e portanto

$$a_k = a_{k-1} + d = a_{k-2} + 2d = \dots = a_1 + d(k-1)$$

Soma dos n primeiros termos:

$$\begin{aligned} S_n &= a_1 + \dots + a_n = \frac{1}{2} n (a_1 + a_n) \\ &= \frac{1}{2} n (2 a_1 + (n - 1) d) \end{aligned}$$

$$\begin{aligned} \text{Exemplos: } 1 + 2 + 3 + 4 + \dots + n &= \frac{1}{2} n (n+1) \\ 1 + 3 + 5 + 7 + \dots + (2n - 1) &= n^2 \\ 1 + 7 + 19 + 37 + \dots + (3n^2 - 3n + 1) &= n^3 \end{aligned}$$

• Séries Geométricas

São da forma,

$$\{a_k\}, k = 0, 1, 2, \dots \text{ com } a_{k+1} / a_k = r \quad (r \neq 1)$$

e portanto

$$a_k = a_{k-1} r = a_{k-2} r^2 = \dots = a_0 r^k$$

Soma dos n primeiros termos:

$$\begin{aligned} S_n &= a_0 + \dots + a_{n-1} = (a_0 - r a_{n-1}) / (1 - r) \\ &= a_0 (1 - r^n) / (1 - r) \end{aligned}$$

Se $|r| < 1$ então, quando $n \rightarrow \infty$,

$$a_0 + a_0 r + a_0 r^2 + a_0 r^3 + \dots = a_0 / (1 - r)$$

Exemplos:

$$\begin{aligned} 1 + 2 + 4 + 8 + \dots + 2^{n-1} &= 2^n - 1 \\ 1 + 2 + 4 + 8 + \dots + 2^{\lfloor \log_2 n \rfloor} &= 2^{\lfloor \log_2 n \rfloor + 1} - 1 \end{aligned}$$

- **A Série Aritmético-Geométrica de razão 2**

$$\begin{aligned} \sum_{k=0}^n k 2^k &= 0 + 2 + 2 \cdot 2^2 + 3 \cdot 2^3 + \dots + n \cdot 2^n \\ &= 2 + 2^n (2n - 2) \end{aligned}$$

- **A Série Harmónica**

$$\lim_{n \rightarrow \infty} (1 + 1/2 + 1/3 + 1/4 + \dots + 1/n - \ln(n)) = \gamma$$

onde $\gamma = 0.57721\dots$ é a constante de Euler.

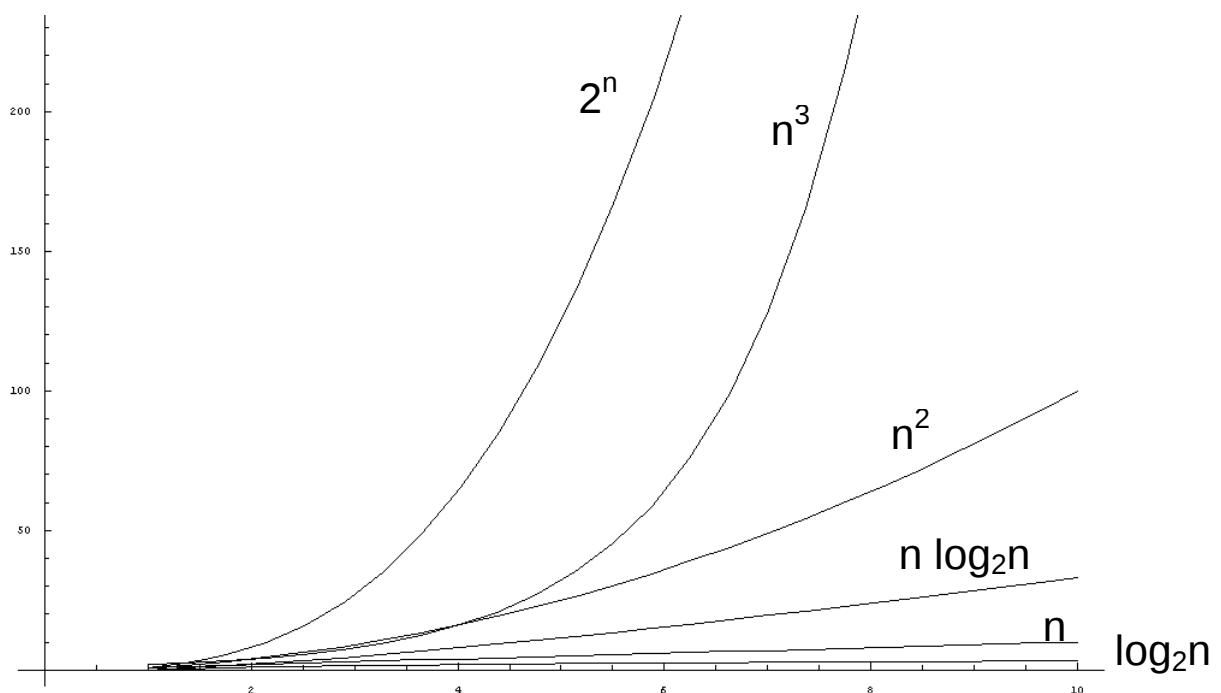
Capítulo 2 : Complexidade Algorítmica

{Uma Medida dos Recursos computacionais utilizados por um Algoritmo}

Complexidade { **Temporal** (Tempo de Execução)
Espacial (Espaço de Memória)

{Vamos analisar apenas a Complexidade Temporal}

□ Breve nota sobre o Comportamento Assimptótico de sucessões divergentes



Alguns exemplos de **Tempos de Execução** de Programas, em função da **Dimensão n** da **Instância** do **Problema** a resolverem:

	10	100	10^3	10^4	10^5	10^6
$\log_2 n$	3	6	9	13	16	19
n	10	100	1000	10^4	10^5	10^6
$n \log_2 n$	30	664	9965	10^5	10^6	10^7
n^2	100	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{300}	10^{3000}	10^{30000}	10^{300000}

(com $2^{10} \approx 10^3$)

1 ano = $365 \times 24 \times 60 \times 60 \approx 3 \times 10^7$ segundos

1 século $\approx 3 \times 10^9$ segundos

1 milénio $\approx 3 \times 10^{10}$ segundos

- Algoritmos de Complexidade Logarítmica são os mais eficientes;
- Algoritmos de Complexidade Exponencial geram tempos de execução inportáveis;
- Mesmo para um (hipotético) Computador 100 vezes mais rápido, os Tempos de Execução seriam igualmente impraticáveis;
- Um algoritmo só é viável se puder ser calculado em “tempo útil”;
- Só “interessam” algoritmos de Complexidade Polinomial, ou inferior.

❑ Problema: Pesquisa de um Elemento num Vector

{Verificar se um dado valor pertence a um vector $a[1..n]$ e, caso pertença, qual a sua posição}

Algoritmo de Pesquisa Sequencial ou Linear

1ª Versão:

```
...
indice:= 0; {Fora do domínio dos índices}

for i:=1 to n do
    if a[i] = valor
    then indice:= i;

if indice = 0
then writeln(valor, 'Não pertence ao vector.')
else writeln(valor, 'Está na posição ', indice, ' do vector.');
```

- Caso ocorram vários elementos iguais ao valor pretendido, será identificado o último.
- Se se pretender identificar apenas a primeira ocorrência, ou se os elementos forem todos distintos, a pesquisa deverá parar logo que detecte a primeira igualdade.

2ª Versão: { a pesquisa pára quando encontra a primeira ocorrência;
com utilização de uma variável lógica }

```
var achou : boolean;    { achou = ( valor ∈ a[1..n] ) }
...
```

```
...
indice:= 0;
achou:= false;
while not achou and (indice < n) do
    begin indice:= indice+1;
           achou:= a[indice] = valor
    end;
{ Aqui achou ou indice = n }

if achou
then writeln(valor, 'Está na posição ', indice, ' do vector.')
else writeln(valor, 'Não pertence ao vector.');
```

3ª Versão: { um Módulo de Pesquisa Sequencial }

```
const lim = ...;
type elemento = ... ;
      vector = array[1..lim] of elemento;
...
```

```
procedure PesqSeq (a : vector; n : integer; valor : elemento;
                   var achou : boolean; var indice : integer);
var i : integer;
begin i:= 1;
      while (a[i] <> valor) and (i < n) do
          i:= i +1;
      { Aqui (a[i] = valor) or (i = n) }

      { Cálculo dos Parâmetros de Saída }
      achou:= a[i] = valor;

      if achou
      then indice:= i
      else indice:= 0
end;
```

Pesquisa Sequencial num Vector Ordenado

- Se os elementos do vector estiverem dispostos por ordem crescente, a pesquisa deverá parar quando encontre o elemento ou a posição que ocuparia, caso lá estivesse.



```

var achou : boolean;
{ achou ≡ (achou o valor ou a sua posição, se lá estivesse) }
...

...
indice:= 0;
achou:= false;
while not achou and (indice<n) do
    begin indice:= indice+1;
        achou:= a[indice] >= valor
    end;
{ Aqui achou ou indice = n }

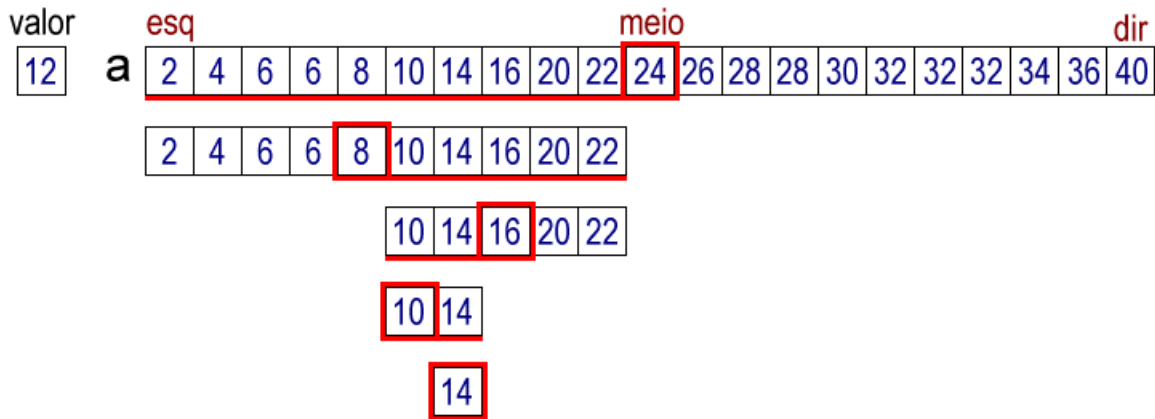
{ achou = ( a[indice] >= valor ) }

if achou
then if a[indice] = valor
    then writeln(valor, 'Está na posição ', indice, ' do vector.')
    else writeln(valor, 'Não pertence ao vector.
                Deveria estar na posição ', indice)
else writeln(valor, 'É superior a qualquer dos', n,
                ' elementos do vector.');
```

...

Pesquisa Binária num Vector Ordenado

- Tirando partido do ordenamento do vector, a pesquisa de um valor pode ser feita de um modo muito mais eficiente:



```
...
var achou : boolean;    { achou = ( valor ∈ a[1..n] ) }
```

```
...
```

```
...
esq:= 1; dir:= n;
achou:= false;
while not achou and (esq <= dir) do
    begin meio:= (esq+dir) div 2;
        if valor = a[meio]
            then achou:= true
            else if valor < a[meio]
                then dir:= meio-1
                else esq:= meio+1
        end;
    { Aqui achou ou esq>dir }
```

```
{ achou = ( a[meio] = valor ) }
```

```
if achou
then writeln(valor, 'Está na posição ', meio, ' do vector.')
else writeln(valor, 'Não pertence ao vector.');
```

```
...
```

□ Análise da Complexidade Temporal de Algoritmos

|| Uma Medida do Número de Operações realizadas por um dado Algoritmo, na resolução de um dado Problema

Para um dado Problema e um dado Algoritmo, sejam:

D_n = o domínio das particularizações concretas de Dados do Problema, com dimensão n ;

$I \in D_n$ = uma instância (particularização) do Problema;

$p(I)$ = a probabilidade de ocorrência de I em D_n ;

$t(I)$ = uma medida das operações realizadas pelo Algoritmo, na resolução de I .

A Instância concreta do Problema a resolver pode ser o Caso Mais Favorável, ou **Melhor Caso**, para o Algoritmo. Assim:

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

ou, a Instância em causa pode constituir o **Pior Caso** do Algoritmo:

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

uma justa **Análise da Complexidade** do Algoritmo deve ser efectuada em termos do **Caso Médio**:

$$A(n) = \sum_{I \in D_n} p(I) t(I)$$

Um Exemplo:

Problema: Procurar um valor x num vector $a[1..n]$;

Algoritmo: Pesquisa Sequencial;

(para simplificar, assumimos que $x \in a[1..n]$)

Que **Operações** importa considerar?

Tratando-se de um Problema de Pesquisa, as Operações mais relevantes são a **comparações** de x com os elementos do vector.

Na 1ª Versão do Algoritmo (ciclo **for**), o número de comparações realizadas é sempre o mesmo, para qualquer Instância, i.e.,

$$B(n) = W(n) = A(n) = n$$

Na 2ª Versão do Algoritmo, o número de comparações depende da localização (índice) de x em $a[1..n]$.

No Melhor dos Casos, $x = a[1]$,

$$B(n) = 1$$

No Pior dos Casos, $x = a[n]$,

$$W(n) = n$$

(note-se que este é também o caso de $x \notin a[1..n]$)

Para uma Análise do Caso Médio, começamos por identificar o Número de Casos Possíveis, e respectivas operações:

I_i com $i \in [1..n]$, representa o caso $x = a[i]$, com $t(I_i) = i$

Se não existir qualquer informação sobre a localização de x ,

$$p(I_i) = P(x = a[i]) = 1/n, \text{ para todo o } i \in [1..n].$$

Calculando:

$$A(n) = \sum_{i=1}^n p(I_i) t(I_i) = \sum_{i=1}^n i/n = (n+1)/2$$

ou seja, em média, metade dos elementos do vector são analisados.

Dizemos que:

|| O Algoritmo da Pesquisa Sequencial é
da ordem de n ou Linear em n .

Outro Exemplo:

Problema: Procurar um valor x num vector ordenado $a[1..n]$;

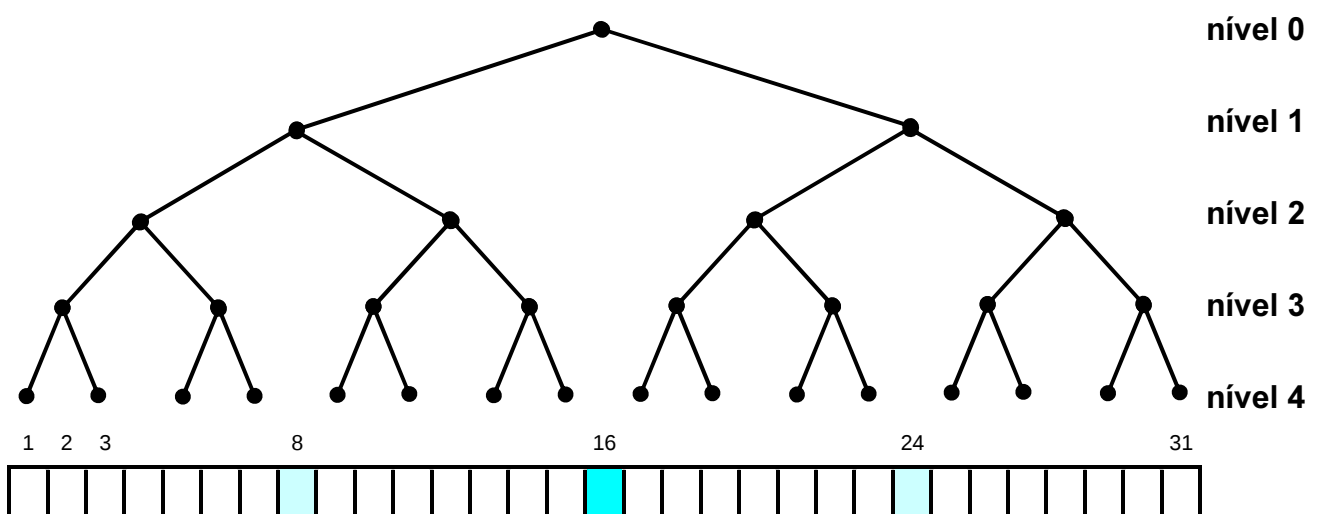
Algoritmo: Pesquisa Binária;

(continuando a assumir que $x \in a[1..n]$)

Como identificar o Melhor e o Pior dos Casos?

Quantos são os casos a considerar?

Analisemos um vector com $n = 2^5 - 1 = 31$ elementos:



Assim, o Melhor dos Casos corresponde a $x = a[16]$ com,

$$B(n) = 1$$

No Pior dos Casos, $x = a[1]$, $x = a[3]$, $x = a[5]$, ..., ou $x = a[31]$,

$$W(n) = 5$$

(equivalente ao caso de $x \notin a[1..n]$)

Análise do Caso Médio:

Os Casos Possíveis correspondem aos níveis da árvore binária.

nível	# elementos	probabilidade	# comparações
0	1 = 2^0	$2^0/n$	1
1	2 = 2^1	$2^1/n$	2
2	4 = 2^2	$2^2/n$	3
3	8 = 2^3	$2^3/n$	4
4	16 = 2^4	$2^4/n$	5
k	2^k	$2^k/n$	k+1

Para um n arbitrário, qual será a altura k da árvore?

Como a soma de todos os elementos tem de ser n ,

$$1 + 2 + 4 + 8 + \dots + 2^k = n$$

e então, $2^{k+1} - 1 = n$ ou $k + 1 = \log_2(n+1)$

$$k = \log_2(n+1) - 1$$

que podemos aproximar por,

$$k \approx \lfloor \log_2 n \rfloor$$

Calculando:

$$\begin{aligned}
 A(n) &= \sum_{i=0}^k 2^i/n (i+1) = 1/n \left(\sum_{i=0}^k i 2^i + \sum_{i=0}^k 2^i \right) \\
 &= 1/n (2 + 2^{k+1} (k-1) + 2^{k+1} - 1) \\
 &= 1/n (1 + 2 \cdot 2^{k+1} k) \quad \text{com } k \approx \lfloor \log_2 n \rfloor
 \end{aligned}$$

portanto,

$$A(n) \approx 1/n (1 + 2 n \lfloor \log_2 n \rfloor)$$

assim como podemos calcular o valor exacto de:

$$\begin{aligned}
 W(n) &= k+1 \quad \text{com } k = \log_2(n+1) - 1 \\
 &= \log_2(n+1)
 \end{aligned}$$

Dizemos que:

|| O Algoritmo da Pesquisa Binária é
da ordem de $\log_2 n$ ou **Logarítmico em n**.

Comparação da ordem de crescimento dos tempos de execução,
entre um Algoritmo Logarítmico e um Algoritmo Linear:

log₂n	3	6	9	13	16	19	23	26	29	33	36
n	10	10 ²	10 ³	10 ⁴	10 ⁵	10 ⁶	10 ⁷	10 ⁸	10 ⁹	10 ¹⁰	10 ¹¹

{tente desenhar o gráfico ...}

A notação O :

Sejam f e g duas funções de domínio $\mathbb{N}$.

Dizemos que f é **da ordem de** g ou **$f(n) = O(g(n))$** se,

$$\exists c, n_0 : f(n) \leq c g(n), \forall n \geq n_0$$

Para comparar o comportamento assintótico de duas funções, basta verificar se a sucessão $\{f(n)/g(n)\}$ é limitada, como por exemplo:

$$(n+1) / 2 = O(n)$$

$$2n^3 + 4n^2 + 3 = O(n^3)$$

$$n^2 + 1/n + 3 = O(n^2)$$

$$(n^2 + 1)(n + 1) = O(n^3)$$

$$n \log_2 n + n + 2 = O(n \log_2 n)$$

$$\ln n = O(\log_2 n)$$

$$\{ \log_b a = \ln a / \ln b \}$$

Assim, podemos definir **Classes de Complexidade Algorítmica**:

$O(1)$	Complexidade Constante
$O(\log n)$	Complexidade Logarítmica
$O(n)$	Complexidade Linear
$O(n \log n)$	Complexidade Log-Linear
$O(n^2)$	Complexidade Quadrática
$O(n^3)$	Complexidade Cúbica
$O(n^k)$	Complexidade Polinomial
$O(a^n)$	Complexidade Exponencial
$O(n!)$	Complexidade Factorial

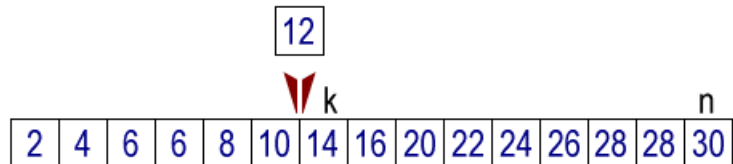
- Algoritmos de Complexidade Exponencial (ou Factorial) só podem ser executados “em tempo útil” para instâncias de muito pequenas dimensões. Portanto, no sentido restrito de **Solução de um Problema**, o termo **Algoritmo** só deve ser aplicado quando a Complexidade é Polinomial (ou inferior).
- Métodos de Complexidade Exponencial correspondem a abordagens do tipo **pesquisa exaustiva** (ou força-bruta) quando **não existe Algoritmo** para a resolução de um Problema.
- Os Problemas de Pesquisa num Vector são resolúveis por Algoritmos de Complexidades entre $O(\log_2 n)$ e $O(n)$.
- O Problema do Ordenamento de um Vector é resolúvel por Algoritmos entre $O(n \log_2 n)$ e $O(n^2)$. Demonstra-se que a Complexidade Mínima é $O(n \log_2 n)$.
- Os Problemas relacionados com operações Matriciais exigem, em princípio, Algoritmos de Complexidade $O(n^3)$. Não existe (ainda) um valor mínimo teoricamente estabelecido.

Observação:

Toda esta Análise de Complexidade tem por objecto o Algoritmo como entidade teórica e não a sua implementação concreta, numa dada linguagem num dado Computador. Assim, são desprezados os tempos envolvidos na execução de operações tais como: chamadas de sub-programas, leituras e escritas, acessos a elementos de vectores ou matrizes, ...

❑ Problema: **Inserção Ordenada num Vector**

{Dado um vector ordenado $a[1..n]$, inserir um novo elemento}



1ª Versão: Procurar o menor k tal que $\text{novo} < a[k]$;
 Deslocar o sub-vector $a[k..n]$ para a direita;
 Colocar o novo elemento em $a[k]$.

```

...
{ Procurar o menor  $k$  tal que  $\text{novo} < a[k]$  }
 $k := 1$ ;
while ( $\text{novo} \geq a[k]$ ) and ( $k < n$ ) do
     $k := k + 1$ ;
{ Aqui  $\text{novo} < a[k]$  ou  $k = n$  }

if  $\text{novo} < a[k]$ 
then begin { Deslocar o sub-vector  $a[k..n]$  para a direita }
    for  $i := n$  downto  $k$  do
         $a[i+1] := a[i]$ ;
    { Colocar o novo elemento em  $a[k]$  }
     $a[k] := \text{novo}$ 
end
else {  $\text{novo} \geq a[k]$  e  $k = n$  }
    { O novo elemento era maior que todos os outros }
     $a[n+1] := \text{novo}$ ;

{ A dimensão do vector aumentou }
 $n := n + 1$ ;
...

```

2ª Versão: Começando pelo último elemento,
ir deslocado cada elemento para a direita,
enquanto forem superiores ao novo;
Colocar o novo elemento.

...

{ Começando pelo último elemento }

k:= n;

{ Enquanto forem superiores ao novo }

```
while (k >= 1) and (a[k] > novo) do  
  begin { Deslocar a[k] para a direita }  
    a[k+1]:= a[k];  
    { Analisar outro }  
    k:= k-1
```

```
  end;
```

{ Aqui k = 0 ou a[k] <= novo }

{ Colocar o novo elemento em a[k+1] }

a[k+1]:= novo

n:= n+1;

...

Exercício:

Estude a Complexidade de cada um dos Algoritmos, em termos das duas operações: **Comparações** e **Deslocamentos** de elementos do vector.

Identifique o **Melhor** e o **Pior** Caso de cada Algoritmo, para cada uma das operações.

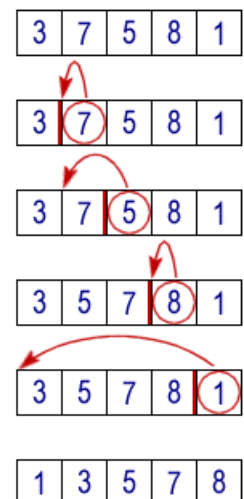
Compare os resultados das Análises dos **Casos Médios**.

Exercício: { Ordenamento por Inserção Sequencial }

Generalizando o Algoritmo anterior, é possível **ordenar** os elementos de um dado vector, com base na seguinte

Estratégia:

Se um sub-vector (formado pelos primeiros elementos do vector dado) já estiver ordenado, basta inserir ordenadamente o elemento seguinte nesse sub-vector.



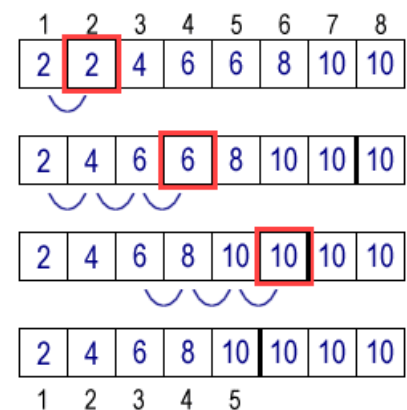
Estude a **Complexidade** do Algoritmo da Ordenamento por Inserção Sequencial, em termos das **Comparações** e dos **Deslocamentos** de elementos do vector, no três Casos: **Melhor**, **Pior** e **Médio**.

Identifique as Instâncias que geram o Melhor e o Pior Caso do Algoritmo.

❑ Problema: **Remoção Ordenada num Vector**

{Dado um vector ordenado, **remover as repetições** de elementos}

1ª Versão: Percorrendo o Vector da esquerda para a direita;
se o elemento $a[i]$ for igual ao anterior
então remover $a[i]$
senão, avançar.



...

```

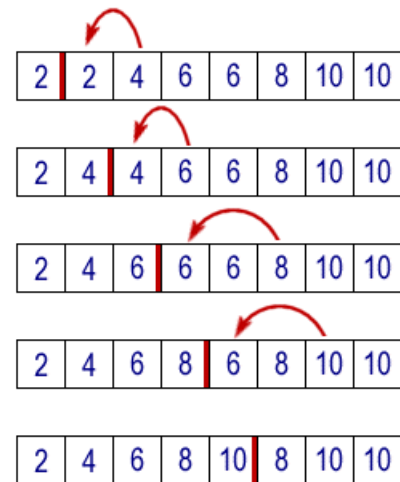
i := 2;
while i <= n do
  if a[i-1] <> a[i]
  then i := i+1
  else begin { Remover a[i] }
        for k:=i to n-1 do
          a[k] := a[k+1];
        { Actualizar a dimensão do Vector }
        n := n-1
      end;

```

...

2ª Versão: Percorrendo o Vector da esquerda para a direita; se um elemento for diferente dos anteriores, acrescentá-lo ao sub-vector de elementos distintos.

{Basta um Ciclo, que pode ser um ciclo **for**}



...

fim := 1; { Número de elementos distintos }

```

for k := 2 to n do
  if a[fim] <> a[k]
  then begin { Mais um elemento distinto }
    fim := fim+1;
    a[fim] := a[k]
  end

```

n:= fim;

...

Análise comparativa das Complexidades das duas versões:

Número de deslocamentos:

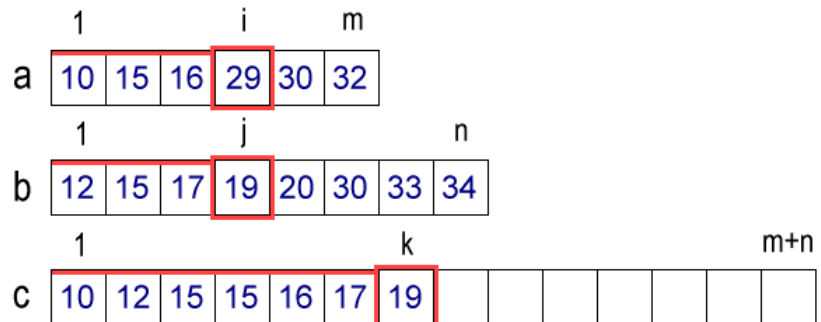
1ª Versão	2ª Versão
Melhor Caso: (elementos todos distintos) $B_D(n) = 0$ Pior Caso: (elementos todos iguais) $W_D(n) = (n-2) + (n-3) + \dots + 1$ $= \sum_{k=2}^{n-1} (n - k) = \frac{1}{2} (n-1)(n-2)$ Caso Médio: (cálculo aproximado) $A_D(n) = \sum_{k=2}^{n-1} \frac{1}{2} (n-k) = (n-1)(n-2)/4$ Complexidade $O(n^2)$	Melhor Caso: (elementos todos iguais) $B_D(n) = 0$ Pior Caso: (elementos todos distintos) $W_D(n) = \sum_{k=2}^n 1 = n - 1$ Caso Médio: $A_D(n) = \sum_{k=2}^n \frac{1}{2} 1 = (n-1) / 2$ Complexidade $O(n)$ {Algoritmo mais eficiente, apesar de efectuar atribuições de elementos a si próprios}

Exercícios:

- Alterar a 2ª versão do Algoritmo, por forma a que não sejam realizadas atribuições inúteis.
- Analisar o Número de Comparações efectuadas por cada versão.

❑ Problema: Fusão de Vectors Ordenados

{Dados dois vectores ordenados $a[1..m]$ e $b[1..n]$, construir o vector ordenado $c[1..m+n]$ com a totalidade dos elementos de a e de b }



```

...
i:= 1; j:= 1; k:= 1;
{ Enquanto não chegar ao fim de nenhum vector}
while (i <= m) and (j <= n) do
    begin { Copiar o menor elemento para c }
        if a[i] < b[j]
            then begin c[k]:= a[i];
                        i:= i + 1
                    end
            else begin c[k]:= b[j];
                        j:= j + 1
                    end;
                k:= k+1
    end;
    → { Acabar de copiar o resto de a, se existir }
    for i:= i to m do
        begin c[k]:= a[i];
            k:= k+1
        end;
    → { Acabar de copiar o resto de b, se existir }
    for j:= j to n do
        begin c[k]:= b[j];
            k:= k+1
        end;

```

Exercício: Verificar que a Fusão Ordenada é Linear em $O(m+n)$.

❑ Problema: Ordenar os Elementos de um Vector

{Colocar os elementos de um dado vector $a[1..n]$
por ordem não decrescente}

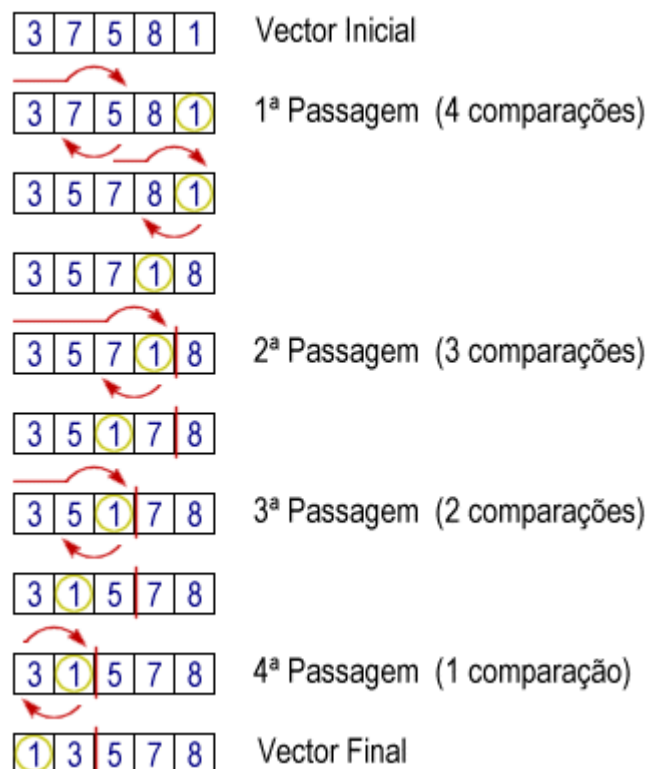
Método do Borbulhamento (BubbleSort)

Estratégia:

Os elementos de um vector $a[1..n]$ estão dispostos por ordem não decrescente quando,

$$\forall i \in [1..n-1] \Rightarrow a[i] \leq a[i+1].$$

Basta portanto identificar todos pares de elementos contíguos tais que $a[i] > a[i+1]$ e trocá-los.



- No final de cada “passagem”, ao longo do sub-vector $a[1..\text{limite}]$, (apenas) o “último” elemento fica na sua posição definitiva;
- Portanto são necessárias $(n-1)$ “passagens”,

1ª Versão:

```

...
for limite:= n downto 2 do
  for i:= 1 to limite-1 do
    if  $a[i] > a[i+1]$ 
      then begin aux:=  $a[i]$ ;
                 $a[i]:= a[i+1]$ ;
                 $a[i+1]:= aux$ 
      end;
...

```

- Como o Algoritmo envolve apenas ciclos **for**, a Análise de Complexidade é simples;

Contagem das Comparações (entre elementos do vector):

$$B_C(n) = W_C(n) = A_C(n) = \sum_{k=2}^n (k-1) = n(n-1) / 2$$

Contagem das Trocas (de elementos do vector):

No Melhor Caso (vector já ordenado)

$$B_T(n) = 0$$

No Pior Caso (vector por ordem inversa)

$$W_T(n) = \sum_{k=2}^n (k-1) = n(n-1) / 2$$

No Caso Médio, assumindo que é igualmente provável a ocorrência (ou não) de uma troca,

$$A_T(n) = \sum_{k=2}^n \frac{1}{2} (k-1) = n(n-1) / 4$$

O BubbleSort é $O(n^2)$ tanto em Trocas como em Comparações.

- O BubbleSort é o mais simples dos Métodos de ordenamento, mas também o menos eficiente.
- Admite contudo vários melhoramentos e é também uma boa base para a construção de métodos mais elaborados.

2ª Versão:

- O melhoramento mais óbvio do BubbleSort consiste em **parar** o processo logo que se detecte que, ao longo de uma passagem, não foram efectuadas quaisquer trocas.

```

...
limite:= n+1;
houvetrocas:= true;

while houvetrocas do
  begin
    limite:= limite-1;
    houvetrocas:= false;    ←

    for i:= 1 to limite-1 do
      if a[i] > a[i+1]
      then begin
        aux:= a[i];
        a[i]:= a[i+1];
        a[i+1]:= aux;
        houvetrocas:= true  ←
      end
    end;
  end;
...

```

- A Análise da Complexidade desta versão já não é tão simples;

No Melhor Caso (vector já ordenado) basta uma “passagem”:

$$B_C(n) = n-1 \quad B_T(n) = 0$$

No Pior Caso (vector por ordem inversa), esta versão é equivalente à anterior:

$$W_C(n) = W_T(n) = \sum_{k=2}^n (k-1) = n(n-1) / 2$$

A Análise do Caso Médio é bastante complicada, envolvendo conceitos de Análise Combinatória. Prova-se que:

$$A_C(n) \approx A_T(n) \approx n(n-1) / 4$$

- De qualquer modo, o BubbleSort é $O(n^2)$ tanto nas Comparações como nas Trocas de elementos do Vector.
- Note-se contudo que, devido à sua simplicidade, o BubbleSort é um método de grande aplicabilidade prática, devendo ser usado quando o vector original está “quase” ordenado, ou para vectores de “pequenas” dimensões.

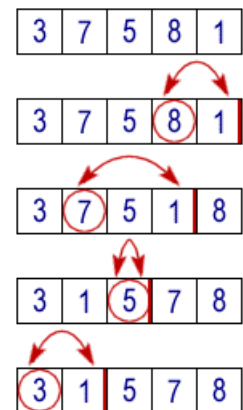
n	10	10 ²	10 ³	10 ⁴	10 ⁵	10 ⁶	10 ⁷	10 ⁸	10 ⁹	10 ¹⁰
n log ₂ n	3×10	7×10 ²	1×10 ⁴	1×10 ⁵	2×10 ⁶	2×10 ⁷	2×10 ⁸	3×10 ⁹	3×10 ¹⁰	3×10 ¹¹
n ²	10 ²	10 ⁴	10 ⁶	10 ⁸	10 ¹⁰	10 ¹²	10 ¹⁴	10 ¹⁶	10 ¹⁸	10 ²⁰

Método da Selecção Linear

Estratégia:

Uma simples observação do comportamento do BubbleSort revela que, ao fim de cada passagem, apenas o “último” elemento de cada sub-vector é colocado na posição correcta.

O método da Selecção Linear consiste em **seleccionar o maior elemento** de cada sub-vector e **trocá-lo com o “último”**.



1ª Versão:

```

...
for ultimo:= n downto 2 do

    begin (* Procurar o maior elemento de a[1..ultimo] *)
        imax:= 1;
        for i:= 2 to ultimo do
            if a[i] > a[imax]
            then imax:= i;

        (* Trocar o maior com o último *)
        aux:= a[imax];
        a[imax]:= a[ultimo];
        a[ultimo]:= aux;

    end;
...

```

Versões alternativas:

- Procurar o menor e trocá-lo com o “primeiro”.
- Procurar o menor e também o maior e trocá-los, respectivamente, com o “primeiro” e com o “último”.

Análise da Complexidade da versão básica da Selecção Linear:

Tanto o número de Comparações como o número de Trocas são fixas, para qualquer instância,

$$B_C(n) = W_C(n) = A_C(n) = \sum_{k=2}^n (k-1) = n(n-1) / 2 = O(n^2)$$

$$B_T(n) = W_T(n) = A_T(n) = n-1 = O(n)$$

Observação:

- O Algoritmo realiza sempre o mesmo número de Trocas, mesmo para um vector já ordenado;
- Vejamos porquê:

```
...  
for k:= n downto 2 do  
  
    begin imax:= indiceMax(1, k);  
          trocar(a[imax], a[k])  
    end;  
...
```

portanto, se pretendermos evitar trocas inúteis,

2ª Versão:

```

...
for k:= n downto 2 do

    begin imax:= indiceMax(1, k);
          if imax <> k
            then trocar(a[imax], a[k])
          end;
    end;
...

```

assim, para cada $k \in [2..n]$, só ocorrerá uma Troca **se** $\text{imax} \neq k$ ou seja, em $k-1$ dos casos possíveis.

Por outro lado, $P(\text{imax} = i) = 1/k$ para todo o $i \in [1..k]$.

Logo, em Média:

$$\begin{aligned}
 A_T(n) &= \sum_{k=2}^n \sum_{i=1}^{k-1} 1/k = \sum_{k=2}^n (k-1) / k \\
 &= \sum_{k=2}^n 1 - \sum_{k=2}^n 1/k = (n-1) - \sum_{k=2}^n 1/k
 \end{aligned}$$

e atendendo a que, $\sum_{k=2}^n 1/k = H(n) - 1 \approx \ln n - 1$

$$A_T(n) \approx n - \ln n = O(n)$$

Ou seja, apesar de conseguirmos $B_T(n) = 0$, o Caso Médio das Trocas realizadas pelo Algoritmo permanece Linear.

❑ A Complexidade dos Algoritmos de Ordenamento

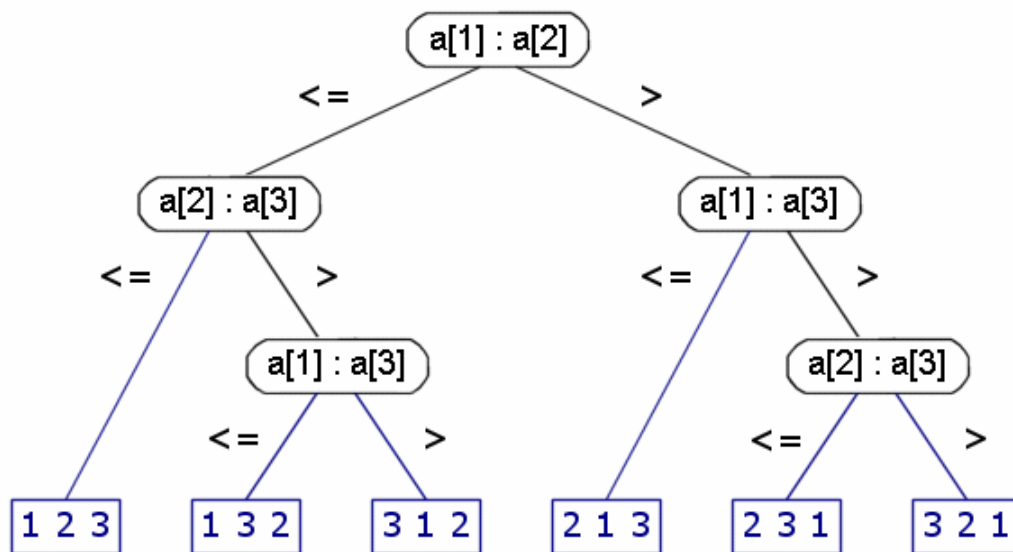
Consideremos os Algoritmos de Ordenamento (baseados em trocas de pares de elementos) e analisemos o **número de comparações** realizadas entre pares de elementos.

- **Um exemplo:**

Seja $a[1..3]$ um vector cujos elementos são $\{1, 2, 3\}$.

Um Algoritmo de Ordenamento deverá efectuar comparações (e consequentes trocas) de modo a obter a permutação $[1\ 2\ 3]$.

Analisemos todas as possibilidades, na **Árvore de Decisão**:



A Árvore de Decisão tem **6 folhas**, que correspondem às $3! = 6$ **permutações** dos inteiros $\{1, 2, 3\}$.

Um Algoritmo é "eficiente" se for capaz de encontrar qualquer permutação efectuando um número "mínimo" de comparações.

Cada **vértice interno** corresponde a uma **comparação**.

O **número de comparações** realizadas para encontrar uma permutação corresponde ao **comprimento de um caminho** desde a **raiz** até uma **folha**.

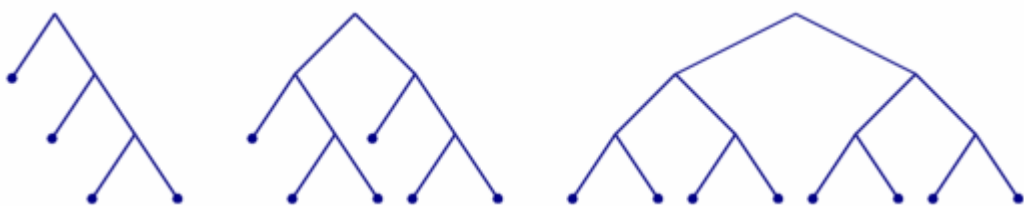
A Árvore de Decisão é uma **Árvore Binária**.

Chama-se **altura** de uma Árvore Binária ao comprimento do caminho mais longo da raiz até uma folha.

- **Um Lema:**

Uma Árvore Binária de altura $\underline{h}$ tem, no máximo, $\underline{2^h}$ folhas.

Ex: para $h = 3$, $\# \text{folhas} \leq 2^3 = 8$



{ Demonstrar por Indução }

- **Portanto:**

Numa Árvore Binária com $\underline{n}$ folhas e altura $\underline{h}$, $n \leq 2^h$
 $h \geq \log_2 n$

Altura de uma Árvore Binária com $\underline{n}$ folhas $\geq \log_2 n$.

Mas a Árvore de Decisão tem $\underline{n!}$ folhas ...

- **Outro Lema:**

Altura de uma Árvore Binária com $n!$ folhas $\geq \log_2 n!$.

$$\begin{aligned}
 \text{Calculemos, } \ln n! &= \ln(1 \times 2 \times \cdots \times n) \\
 &= \ln 1 + \ln 2 + \cdots + \ln n \\
 &= \sum_{k=1}^n \ln k \\
 &\approx \int_1^n \ln x dx \\
 &= [x \ln x - x]_1^n \\
 &= n \ln n - n + 1
 \end{aligned}$$

$$\text{Assim, } \log_2 n! = \frac{\ln n!}{\ln 2} \approx \frac{n \ln n - n + 1}{\ln 2} \approx n \log_2 n - n$$

Portanto, o limite inferior da altura de uma Alvore Binária com $n!$ folhas pode ser aproximado por $n \log_2 n - n$.

*O "mais eficiente" Algoritmo de Ordenamento deverá atingir esse valor "mínimo" comparações, mesmo no seu **Pior Caso**.*

- **Teorema**

Para os Algoritmos de Ordenamento baseados em trocas, sendo $C(n)$ o número de Comparações realizadas,

$$C(n) \geq O(n \log_2 n)$$