1. Análise algorítmica

A análise algorítmica inclui o estudo de todos os aspectos que interferem no rendimento da resolução computacional de problemas, desde a formulação preliminar, passando pelos passos da programação, até à tarefa final ou à interpretação dos resultados obtidos.

1.1. Definição de algoritmo

Um algoritmo é um processo que consiste de um conjunto finito de regras objectivas (não ambíguas), que traduzem uma sequência finita de operações que produzem a solução de um problema ou de uma classe de problemas. Desta definição tiram-se as seguintes notas importantes :

1. Cada iteração de um algoritmo deverá ser objectiva e definida com exactidão. As acções a executar deverão ser especificadas rigorosamente para cada caso.
2. Um algoritmo deverá sempre chegar a uma solução para o problema, após um número finito de iterações. De facto, na prática, a condição de paragem não é suficiente, uma vez que o número de iterações necessário para resolver um problema, embora finito, pode ser demasiado grande para ser praticável computacionalmente. Um algoritmo eficiente deve exigir, não apenas um número finito, mas um número razoável de iterações.
3. Qualquer algoritmo com significado contém 0 ou mais dados de entrada e fornece 1 ou mais dados de saída (resultados). Os dados de entrada podem ser definidos como quantidades a serem introduzidas inicialmente no algoritmo (antes de ser executado), e os resultados como quantidades que têm uma relação específica com os dados de entrada e que são devolvidos no final da execução do algoritmo.
4. É preferível que o algoritmo seja apropriado a todos os membros de uma classe de problemas do que apenas a um único problema. Esta propriedade de generalidade, embora não seja uma necessidade, é certamente um atributo pretendido de um algoritmo bem construído.

Embora o conceito de algoritmo seja muito vasto, apenas irão ser estudados aqueles que podem ser executados no computador. Estes algoritmos são incluídos num programa para computador (ou num conjunto de programas), o que é feito recorrendo a uma linguagem de programação, como por exemplo o Delphi Pascal — implementação computacional.

1.2. Medidas de eficiência

Inventar algoritmos é relativamente fácil. Na prática, no entanto, não se pretende apenas algoritmos, mas sim bons algoritmos. Assim, o objectivo é inventar algoritmos e provar que eles são mesmo bons.

A qualidade de um algoritmo pode ser avaliada utilizando vários critérios. Um dos critérios mais importantes é o tempo despendido na execução do algoritmo. Existem vários aspectos a considerar em cada critério de tempo. Um está relacionado com o tempo de execução requerido pelos diferentes algoritmos, para encontrar a solução final de um problema ou cálculo particular.

No entanto, cada uma das medidas empíricas está fortemente dependente, tanto do programa como da máquina utilizada para implementar o algoritmo. Assim, a alteração num programa pode não representar uma alteração significativa no algoritmo básico, mas pode, no entanto, afectar a velocidade de execução. Além disso, se dois programas são comparados, primeiro numa máquina e depois noutra, as comparações podem conduzir a diferentes conclusões. Assim, embora a comparação de programas completos a serem executados em computadores seja uma importante fonte de informação, os resultados são inevitavelmente afectados pelo estilo de programação e pelas características da máquina.

Uma boa alternativa às medidas empíricas, é a análise matemática da dificuldade intrínseca de resolver computacionalmente um problema. Esta análise fornece um importante meio de avaliação do custo de execução do algoritmo, quando bem usada.

A eficiência do tempo de execução de um algoritmo é uma função do tamanho do problema a ser resolvido computacionalmente. No entanto, supondo que o programa termina, a resolução de um problema específico requer apenas tempo e espaço de armazenamento suficientes. Os algoritmos com mais interesse geral, são os que podem ser aplicados a classes de problemas de um certo tipo. Para estes algoritmos, o tempo e o espaço de memória requeridos por um programa varia com o problema particular a ser resolvido.

1.3. Algoritmos e complexidade

De modo a medir-se o custo de execução de um programa, é normal definir-se uma função de complexidade (ou custo) F, onde F(N) é uma medida tanto do tempo requerido para executar o algoritmo associado a um problema de tamanho N, como do espaço de memória requerido para essa execução. Pode-se falar, tanto de função de complexidade do tempo de execução como do espaço de armazenamento do algoritmo ou, simplesmente, de função de custo de um algoritmo.

A complexidade computacional de um algoritmo é usada para estimar o poder computacional requerido para resolver o problema, e é medida pelo número de operações aritméticas (produtos, somas, etc.) e lógicas (comparações) requeridas.

De uma maneira geral, o custo na obtenção duma solução aumenta com o tamanho do problema (N). Se o valor de N é suficientemente pequeno, até mesmo um algoritmo ineficiente não terá um grande custo de execução; consequentemente, a escolha de um algoritmo para um problema pequeno não é crítico (a menos que o problema seja resolvido muitas vezes).

Na maioria dos casos de interesse prático, pretende-se comparar a eficiência ("performance") dos algoritmos para valores de N relativamente grandes. Neste contexto, é conveniente falar-se do comportamento assimptótico (próximo do ideal) da função de complexidade. A função de complexidade assimptótica (tempo ou espaço) determina fundamentalmente o tamanho do problema que pode ser resolvido pelo algoritmo. Em termos de terminologia associada com a noção de complexidade, deve dizer-se, por exemplo, que a complexidade (tempo) do algoritmo é O(ln(N)), lê-se "ordem ln(N)", se o processamento, pelo algoritmo, de entradas de tamanho N requer o tempo de C.ln(N), para qualquer constante C, quando N tende para o infinito. Geralmente, em relação ao dizer-se a complexidade do tempo de um algoritmo particular, qualquer uma das seguintes terminologias serão usadas com o mesmo significado :

- **a)** a complexidade de tempo de um algoritmo é da ordem de ln(N); isto também pode ser escrito como "O(ln(N))",
- **b)** o algoritmo é executado em O(ln(N)) vezes,
- **c)** a quantidade de trabalho requerido pelo algoritmo é proporcional a ln(N) ou é "O(ln(N))".

Quando apropriado, o termo "unidade de tempo" é usado com o mesmo significado do termo "uma operação básica". Consequentemente, a seguinte terminologia adicional será usada com o mesmo significado de a), b) e c) :

- **d)** o algoritmo requer um número de operações básicas proporcional a ln(N).

1.4. Análise de complexidade

A terminologia utilizada é a da alínea **d**, isto é, a análise do número de operações básicas (C + A, em que C = nº de comparações e A = nº de atribuições) realizadas por um determinado algoritmo; os números C e A são determinados em função do tamanho do problema (N). O cálculo do número de operações pode ser feito mediante a análise do pior dos casos ou análise do caso médio.
Sejam :

D_N = { conjuntos de dados de grandeza N }

$I \in D_N$

p(I) = probabilidade de ocorrência de I em D_N

t(I) = número de operações realizadas pelo algoritmo sobre I

Análise do pior dos casos :

$W(N) = \max t(I),\ I \in D_N$

Análise do caso médio :

$A(N) = \sum p(I) \cdot t(I),\ I \in D_N$

Exemplo : Pesquisa exaustiva de X em L[1 . . N] (ver 2.7.1, pág. 9) : quantas operações (comparações) são realizadas ? (Pode-se só contabilizar o número de comparações, uma vez que o número de atribuições, para este caso específico, é 1.)

Há N + 1 casos a considerar :

I_i com $i \in [1 . . N]$ representa o caso $X = L[i]$

I_{N+1} representa o caso $X \notin L[1 . . N]$

No pior dos casos :

Como $t(I_N) = t(I_{N+1}) = N$ { comparações }

Então $W(N) = N$

No caso médio :

Seja $q = P(X \in L[1 . . N])$.

Então $P(X \notin L[1 . . N]) = 1 - q$

E, assumindo ser igualmente provável que

$\forall i \in [1 . . N] : X = L[i]$

então

$$P(X = L[i]) = \frac{q}{N},\ \forall i \in [1 . . N].$$

Logo,

$$A(N) = \sum_{i=1}^{N+1} p(I_i) \cdot t(I_i) = \sum_{i=1}^{N} \frac{q}{N} \cdot i + (1-q) \cdot N,\ \ i = t(X = L[i])$$

$$A(N) = \frac{q}{N} \cdot \sum_{i=1}^{N} i + (1-q) \cdot N = \frac{q}{N} \cdot \frac{N(N+1)}{2} + (1-q) \cdot N = \frac{q}{2} \cdot (N+1) + (1-q) \cdot N$$

Se for conhecido à partida que $X \in L[1 . . N]$ então q = 1 e :

$$A(N) = \frac{N+1}{2}$$ { *em regra, metade dos elementos são analisados* }

Se for igualmente provável que X pertença ou não a L, isto é, que q = 1/2 , então :

$$A(N) = \frac{N+1}{4} + \frac{N}{2} \approx \frac{3}{4} \cdot N$$

Para ambos os casos, o algoritmo é Linear ou O(N) (da ordem de N).

Def. : Sejam f e g duas funções de domínio $\aleph$. Diz-se que f é da ordem de g ou f(n) = O(g(n)) se

$\exists\ c, n_0 > 0 \quad : \quad f(n) \le c \cdot g(n_0)\ , \quad \forall\ n > n_0$.

NOTA : Nos algoritmos que se irão estudar neste texto, apenas se irá fazer a análise de complexidade associada ao pior dos casos.

2. Listas

2.1. Conceito

Uma LISTA é uma sequência linear de tipos de dados iguais com as seguintes propriedades :

- existe um primeiro elemento (a cabeça) com um único sucessor (o próximo),
- existe um último elemento (a cauda) sem sucessor,
- qualquer outro elemento tem um sucessor (próximo) e um antecessor (anterior);

sobre a qual se pode realizar qualquer das operações seguintes :

1. Verificar se uma lista está vazia (não tem elementos)
2. Verificar se uma lista está cheia (atingiu o máximo de elementos possível)
3. Pesquisar um elemento numa lista (procurar a posição do elemento na lista)
4. Inserir um elemento numa lista (no fim, no início ou por ordem)
5. Remover/eliminar um elemento duma lista
6. Ordenar uma lista

Uma LISTA pode ser implementada como dados estáticos (fixos) ou dinâmicos, utilizando um “array” ou ponteiros, respectivamente.

2.2. Implementação utilizando “arrays”

Ao implementar uma lista num espaço de armazenamento contínuo, todos os dados na estrutura são mantidos num “array”. Os “arrays” tem que ser declarados com um tamanho fixado quando o programa é escrito, e não pode ser alterado enquanto o programa está a ser executado.

Quando se está a escrever um programa, tem que se decidir qual é o limite máximo de memória que vai ser necessário para os “arrays” e estabelecer este limite nas declarações. Se o programa for executado com um pequeno conjunto de dados, então muito deste espaço nunca vai ser utilizado. Se pelo contrário, o programa for executado com um grande conjunto de dados, então pode-se esgotar o espaço estabelecido na declaração e encontrar uma situação de *overflow*. Isto pode ocorrer mesmo no caso da memória do computador não estar totalmente usada, devido aos limites iniciais do “array” serem muito pequenos.

Numa lista cada um dos elementos consiste no seguinte :

1. Dados
2. Chave — campo pelo qual a lista está ordenada (opcional)
3. Localização (índice) na lista

Numa lista :

- o primeiro elemento tem índice 1 e não tem antecessor
- o último elemento tem índice N e não tem sucessor.

É importante fazer uma distinção entre estes dois conceitos. Numa lista de N elementos, pode ser inserido ou eliminado um elemento, donde, se $N = 3$ então a lista contém somente 3 elementos, se $N = 19$ então a lista contém 19 elementos — o tamanho da lista é variável.

Uma lista é uma estrutura de dados dinâmica porque o seu tamanho pode variar. Um “array” é uma estrutura de dados fixa porque o seu tamanho é fixo, sendo quando o programa é compilado.

Os conceitos de lista e “array” estão relacionados uma vez que uma lista de tamanho variável pode ser implementada usando um “array” com tamanho fixo (com algumas posições do “array” não sendo utilizadas).

Existem várias formas diferentes de implementar uma lista e não se deve confundir decisões de implementação com decisões de escolha e especificação de estruturas de dados.

Por exemplo : uma lista com 1000 elementos, os quais são valores inteiros.

```
LISTA = array [ 1 . . 1000 ] of integer ;
```

1	2	3	4	5	6		999	1000
9	14	18	27	35	51	…	976	982

Características fundamentais :

Comprimento : constante (= 1000) ⇒ ineficiência
Acesso : directo
Reordenamento ⇒ trocas (ou vector auxiliar)
Inserção (e remoção) de elementos ⇒ deslocamento de todos os elementos seguintes
Tamanho actual : necessário actualizar e testar eventual transbordo (> 1000).

2.3. Definição

Os elementos de uma lista podem ser definidos como um tipo qualquer simples (por exemplo, inteiro, real ou carácter) ou estruturado (por exemplo, registo). Como se disse, uma lista pode ser definida como um vector ("array"). Portanto, antes de se definir uma lista tem que se definir os elementos da lista.

Considere-se, por exemplo, as seguintes definições em Delphi Pascal :

```
const  Max = 50 ;      { número máximo de elementos que a lista pode receber }
type  Elemento = record
                   Nome, Morada : string ;
                   Idade : byte ;
                   BI : integer ;
                   Sexo : (Masculino, Feminino) ;
              end ;
              Lista = array [ 1 . . Max ] of Elemento ;
```

Nesta lista, cada um dos elementos consiste no seguinte :

- Dados — informação associada (Nome, Morada, Idade, BI e Sexo)
- Chave — campo que servirá de base para ordenar a lista — por exemplo, o Nome
- Localização — é o índice que lhe está associado (entre 1 e Max).

Para controlar o tamanho da lista, utiliza-se uma variável global do tipo inteiro. Considere-se, por exemplo, a seguinte declaração em Delphi Pascal :

```
var  N : integer ;      { número de elementos da lista — tamanho }
```

Nos capítulos seguintes, sempre que se referir à ordenação da lista, é suposto ser por ordem crescente do campo Chave, se nada se disser em contrário.

2.4. Lista vazia e lista cheia

Uma lista está vazia quando não tem qualquer elemento e, está cheia quando não pode receber mais elementos. Para se verificar se a lista está vazia ou cheia, basta analisar o valor da variável N, da seguinte forma :

N = 0 ⇒ lista vazia
N = Max ⇒ lista cheia

2.5. Inserir um elemento numa lista

Na resolução deste problema tem de ser analisadas três situações distintas, consoante a forma como se pretende inserir o elemento : no inicio, no fim ou com a lista ordenada.

2.5.1. Inserir no inicio

Neste caso tem que se fazer avançar uma posição a todos os elementos para que a posição 1 fique liberta, para que esta possa receber o novo elemento.

Um possível algoritmo é o seguinte :

Dados de entrada : o elemento X, a lista L e o tamanho da lista (N)
Dados de saída : a lista L e o seu tamanho (N) actualizados
Algoritmo :

```
Avançar uma posição todos os elementos da lista L
Inserir X na posição 1
Actualizar o tamanho da lista, incrementando em um valor a N
```

Uma rotina que traduza o algoritmo é, por exemplo, o procedimento que se segue, que recebe o elemento a inserir (X), uma lista (L) e o tamanho da lista (N) e, devolve a lista L e o seu tamanho N actualizados :

```
procedure  Inserir_Inicio ( X : Elemento ; var L : Lista ; var N : integer ) ;
var i : integer ;
begin
    for i := N downto 1 do
        L [i +1] := L [ i ] ;      { avançar todos os elementos da lista uma posição }
    L [ 1 ] := X ;                 { colocar o elemento X na posição 1 }
    N := N + 1                     { actualizar o tamanho da lista, incrementando um valor }
end ;
```

Ordem de complexidade (pior dos casos) : O(N) — C = 0 e A = (N–1) + 2 = N + 1; logo, C + A = N + 1.

2.5.2. Inserir no fim

Aqui apenas é necessário introduzir o novo elemento uma posição à frente do último elemento da lista, passando a ser este novo elemento o último da lista.

Um possível algoritmo é o seguinte :

Dados de entrada : o elemento X, a lista L e o tamanho da lista (N)
Dados de saída : a lista L e o seu tamanho (N) actualizados
Algoritmo :

```
Inserir X na posição N + 1 (uma posição à frente do último elemento da lista)
Actualizar o tamanho da lista, incrementando em um valor a N
```

Uma rotina que traduza o algoritmo é, por exemplo, o procedimento que se segue, que recebe o elemento a inserir (X), uma lista (L) e o tamanho da lista (N) e, devolve actualizados a lista L e o seu tamanho N :

```
procedure  Inserir_Fim ( X : Elemento ; var L : Lista ; var N : integer ) ;
var i : integer ;
begin
    L [N + 1] := X ;  { colocar o elemento X na posição N+1 }
    N := N + 1        { actualizar o tamanho da lista, incrementando um valor }
end ;
```

Este algoritmo é da ordem de complexidade de 0, O(0) : C = 0 e A = 2.

2.5.3. Inserir por ordem

Este é o caso mais utilizado, começando-se por pesquisar a posição onde inserir o elemento, para depois se fazer avançar uma posição todos os elementos que se encontram depois da posição pesquisada, para que esta posição fique liberta para receber o novo valor.

Um possível algoritmo é o seguinte :

Dados de entrada : o elemento X, a lista L e o tamanho da lista (N)
Dados de saída : a lista L e o seu tamanho (N) actualizados
Algoritmo : (*lista ordenada crescentemente*)

> Determinar a posição onde inserir o elemento X, **k**
> Avançar uma posição todos os elementos da lista L que estão depois da posição **k**
> Inserir X na posição **k**
> Actualizar o tamanho da lista, incrementando N em um valor

Uma rotina que traduza o algoritmo é, por exemplo, o procedimento que se segue, que recebe o elemento X a inserir, a lista L e o tamanho da lista (N) e, devolve actualizados a lista L e o seu tamanho N :

```
procedure  Inserir_Ordem ( X : Elemento ; var L : Lista ; var N : integer ) ;
var i, k : integer ;
begin
    k := N + 1 ;            { para previr que o elemento X seja inserido no fim da lista }
    for i := 1 to N do
        if ( L[ i ] . Nome > X . Nome ) then
        begin               { foi encontrado o primeiro elemento superior a X }
            k := i ;        { k = posição onde inserir o elemento X }
            Break           { terminar pesquisa, pois já foi encontrada a posição }
        end ;
    for i := N downto k do
        L [i +1] := L [ i ] ;      { avançar uma posição os elementos de L depois da posição k }
    L [ k ] := X ;          { colocar o elemento X na posição k }
    N := N + 1              { actualizar o tamanho da lista, incrementando um valor }
end ;
```

Ordem de complexidade (pior dos casos) : O(N) — C = N e A = 1 + k + (N–k) + 2 = 3 + N; logo, C + A = N + 3 + N = 3 + 2 x N.

2.6. Remover um elemento duma lista

Este problema consiste em remover (eliminar) um dado elemento de uma lista, a qual pode ou não estar ordenada. No entanto, apesar destas duas situações serem diferentes, em termos algorítmicos essa diferença encontra-se apenas na forma de pesquisar a posição do elemento a remover. Para tal, basta apenas aplicar um dos métodos existentes para o efeito (ver Lista não ordenada (pesquisa exaustiva) – pág. 9), o que mais se adequar à lista : ordenada ou não ordenada.

Desta forma, um dos possíveis algoritmos é o seguinte :

Dados de entrada : o elemento X, uma lista L e o tamanho da lista (N)
Dados de saída : a lista L e o seu tamanho (N) actualizados
Algoritmo :

> k ← posição do elemento X na lista L (utilizar um dos métodos de pesquisa)
> Recuar uma posição todos os elementos que se encontrem à frente do elemento de posição **k**
> Actualizar o tamanho da lista, decrementando N em um valor

Uma rotina que traduza o algoritmo é, por exemplo, o procedimento que se segue, que recebe o elemento a remover (X), uma lista ordenada (L) e o tamanho da lista (N) e, devolve actualizados a lista L e o seu tamanho N :

```
procedure  Remover_Ordenada ( X : Elemento ; var L : Lista ; var N : integer ) ;
var i, k : integer ;
begin
    k := Pesquisa_Binaria_Rec (X, L, N);        { lista ordenada }
    if ( k = 0 ) then     { o elemento a remover, X, não pertence à lista L }
        Exit ;
    for i := k to N–1 do
        L [ i ] := L [ i + 1 ] ;
    N := N – 1        { actualizar o tamanho da lista, decrementando a N um valor }
end ;
```

Uma rotina que traduza o algoritmo é, por exemplo, o procedimento que se segue, que recebe o elemento a remover (X), uma lista não ordenada (L) e o tamanho da lista (N) e, devolve actualizados a lista L e o seu tamanho N :

```
procedure  Remover_Nao_Ordenada ( X : Elemento ; var L : Lista ; var N : integer ) ;
var i, k : integer ;
begin
    k := Pesquisa_Nao_Ordenada (X, L, N);          { lista não ordenada }
    if ( k = 0 ) then     { o elemento a remover, X, não pertence à lista L }
        Exit ;
    for i := k to N–1 do
        L [ i ] := L [ i + 1 ] ;
    N := N – 1        { actualizar o tamanho da lista, decrementando a N um valor }
end ;
```

No entanto, a forma mais comum de tratar este problema não é fornecer o elemento a remover, mas sim apenas a posição daquele elemento, sendo o trabalho de pesquisa da posição do referido elemento, assim como a verificação se aquele elemento pertence ou não à lista, feito antes da remoção. Isto é, antes de se remover um elemento duma lista, verifica-se se aquele elemento pertence à lista e, caso pertença, em que posição se encontra, o que se pode fazer recorrendo a um dos métodos de pesquisa existentes (ver Pesquisa de um elemento numa lista, pág. 9).

Assim sendo, um possível algoritmo é o seguinte :

Dados de entrada : a posição (k) do elemento a remover, uma lista L e o tamanho da lista (N)
Dados de saída : a lista L e o seu tamanho (N) actualizados
Algoritmo :

```
Recuar uma posição todos os elementos que se encontrem à frente do elemento de posição k
Actualizar o tamanho da lista, decrementando N em um valor
```

Uma rotina que traduza o algoritmo é, por exemplo, o procedimento que se segue, que recebe o elemento a remover (X), uma lista (L) e o tamanho da lista (N) e, devolve actualizados a lista L e o seu tamanho N :

```
procedure  Remover ( k : integer ; var L : Lista ; var N : integer ) ;
var i : integer ;
begin
    for i := k to N–1 do
        L [ i ] := L [ i + 1 ] ;        { recuar uma posição os elementos desde k }
    N := N – 1        { actualizar o tamanho da lista, decrementando a N um valor }
end ;
```

Ordem de complexidade (pior dos casos) : O(N) — C = 0 e A = (N – 1) + 1 = N; logo, C + A = N.

2.7. Pesquisa de um elemento numa lista

Dado uma lista, pretende-se determinar a posição de um elemento na lista, caso pertença à lista. O estudo deste problema, deve contemplar duas situações, consoante a lista está ou não ordenada.

2.7.1. Lista não ordenada (pesquisa exaustiva)

Neste caso, a pesquisa do elemento terá que ser feita exaustivamente; isto é, percorrendo e analisando todos os elementos da lista.

Um possível algoritmo para este caso é o seguinte :

Dados de entrada : o elemento X, a lista L não ordenada e o tamanho da lista (N)

Dados de saída : a posição de X na lista L (se $X \in L$)

Algoritmo :

```
Para  cada elemento de L, L[i] com i = 1, …, N,  Fazer
    Se  X . Chave = L[ i ] . Chave  Então
        X encontra-se na posição i
        STOP – terminar pesquisa
X ∉ L  —  não foi encontrado qualquer elemento igual a X (relativamente à Chave)
```

Uma rotina que traduza o algoritmo é, por exemplo, a função que se segue, que devolve a posição do elemento X (se $X \in L$) ou zero (se $X \notin L$) :

```
function  Pesquisa_Nao_Ordenada ( X : Elemento ; L : Lista ; N : integer ) : integer ;
var   i : integer ;
begin
    for i := 1 to N do
        if ( X . Nome = L[i] . Nome ) then
        begin
            Pesquisa_Nao_Ordenada := i ;
            Exit
        end ;
    Pesquisa_Nao_Ordenada := 0
end ;
```

Ordem de complexidade (pior dos casos) : O(N) — ver exemplo em Análise de complexidade, pág. 2.

2.7.2. Lista ordenada

Para resolver este problema, pode-se aplicar vários métodos, entre os quais destacam-se os seguintes : pesquisa Sequencial e pesquisa Binária.

2.7.2.1. Pesquisa Sequencial

Neste método, a pesquisa do elemento é feita a partir do inicio da lista e sequencialmente, terminando quando o elemento for encontrado ou quando o valor da Chave do elemento a pesquisar for superior ao da Chave do elemento a analisar.
Um possível algoritmo é o seguinte :

Dados de entrada : o elemento X, a lista L e o tamanho da lista L (N)
Dados de saída : a posição de X na lista L (se $X \in L$) ou zero (se $X \notin L$)
Algoritmo :

```
Para cada elemento de L, L[i] com i = 1, …, N, Fazer
    Se X . Chave = L[i] . Chave Então
        X encontra-se na posição i
        STOP – terminar pesquisa
    Senão
        Se X . Chave < L[i] . Chave Então
            X ∉ L
            STOP – terminar pesquisa      { a partir daqui os elementos são superiores a X }
X ∉ L — todos os elementos de L são inferiores a X (relativamente à Chave)
```

Uma rotina que traduza o algoritmo é, por exemplo, a função que se segue, que devolve a posição do elemento X (se $X \in L$) ou zero (se $X \notin L$) :

```
function Pesquisa_Sequencial ( X : Elemento ; L : Lista ; N : integer ) : integer ;
var i : integer ;
begin
    for i := 1 to N do
        if ( X . Nome = L[i] . Nome ) then
        begin
            Pesquisa_ Sequencial := i ;
            Exit
        end
        else
            if ( X . Nome < L[i] . Nome ) then
            begin
                Pesquisa_ Sequencial := 0 ; { pode-se concluir que X ∉ L, pois a partir }
                Exit                        { daqui só existem elementos superiores a X }
            end ;
    Pesquisa_ Sequencial := 0     { todos os elementos de L são diferentes de X }
end ;
```

Ordem de complexidade (pior dos casos) : O(N) — $C_1 = N$, $C_2 = N$ e $A = 1$; logo, $C + A = 2N + 1$

2.7.2.2. Pesquisa Binária

Neste método, a pesquisa do elemento na lista é feita através da análise de sub-listas. Isto é, pega-se no elemento que se encontra ao meio da lista e verifica-se se o elemento a pesquisar está para a direita ou para a esquerda deste elemento; se está à esquerda, vai-se pesquisar apenas a sub-lista à esquerda deste elemento, caso contrário, pesquisa-se apenas a sub-lista à direita. O processo termina quando o elemento for encontrado ou quando a sub-lista onde se vai pesquisar o elemento não tem elementos, o que significa que o elemento a pesquisar não pertence à lista.

A implementação deste método pode ser feita de duas formas distintas ou versões : iterativa e recursiva.

Um possível algoritmo para a versão iterativa é o seguinte :

Dados de entrada : o elemento X, a lista L e o tamanho da lista L (N)

Dados de saída : a posição de X na lista L (se $X \in L$) ou zero (se $X \notin L$)

Algoritmo :

```
esq ← 1
dir ← N
Enquanto ( esq ≤ dir ) Fazer     { esq ≤ dir  ⇒  L ≠ ∅ }
    meio ← posição do elemento que está ao meio da lista : (esq + dir) / 2
    Se X . Chave = L [Meio] . Chave Então
        X encontra-se na posição meio
        STOP – terminar pesquisa
    Senão
        Se X . Chave < L [Meio] . Chave Então
            dir ← meio – 1
        Senão
            esq ← meio + 1
X ∉ L  ——  todos os elementos de L são inferiores a X (relativamente à Chave)
```

Uma rotina que traduza o algoritmo é, por exemplo, a função que se segue, que devolve a posição do elemento X (se $X \in L$) ou zero (se $X \notin L$) :

```
function Pesquisa_Binaria_It ( X : Elemento ; L : Lista ; N : integer ) : integer ;
var  i, esq, dir, meio : integer ;
begin
    esq := 1 ;
    dir := N ;
    while ( esq <= dir ) do
    begin
        meio := ( esq + dir ) div 2 ;
        if X . Nome = L [meio] . Nome then
        begin
            Pesquisa_Binaria_It := meio ;
            Exit
        end
        else
            if X . Nome < L [meio] . Nome then
                dir := meio – 1
            else
                esq := meio + 1
    end ;
    Pesquisa_Binaria_It := 0        { X ∉ L }
end ;
```

Ordem de complexidade (pior dos casos – $X \notin L$ e $X < L[1]$) : $O(\log_2 N)$.

1^a comparação : $[1 . . N] = [1 . . N/2^0]$

2^a comparação : $[1 . . N/2] = [1 . . N/2^1]$

3^a Comparação : $[1 . . (N/2)/2] = [1 . . N/4] = [1 . . N/2^2]$

. . .

i-ésima comparação : $[1 . . N/2^{i-1}]$

Na última comparação : [1 . . 1]. Ou seja, $N/2^{i-1} = 1$.

Logo, $2^{i-1} = N \Rightarrow i - 1 = \log_2 N \Leftrightarrow i = \log_2 N + 1$

Assim sendo, o número de comparações é o seguinte : $C = \log_2 N + 1$.

O número de atribuições está associado ao número de comparações : por cada comparação é efectuados 2 atribuições (meio e dir/esq). Logo, A = 2 . ($\log_2$ N + 1)

Concluindo,

C + A = ($\log_2$ N + 1) + (2 . ($\log_2$ N + 1)) = 3 . $\log_2$ N + 3.

Ou seja, o algoritmo é da ordem de complexidade de $\log_2$ N.

Um possível algoritmo para a versão recursiva é o seguinte :

Dados de entrada : o elemento X, a lista L e as posições dos elementos mais à esquerda (esq) e mais à direita (dir) na lista L

Dados de saída : a posição de X na lista L (se X ∈ L) ou zero (se X ∉ L)

Algoritmo :

```
Se  esq > dir  Então      { esq > dir  ⇒  L = ∅ }
    X ∉ L — pesquisa de X numa lista vazia
    STOP — terminar pesquisa
meio ← posição do elemento que está ao meio da lista : (esq + dir) / 2
Se X . Chave = L [Meio] . Chave  Então
    X encontra-se na posição meio
Senão
    Se X . Chave < L [Meio] . Chave  Então
        Pesquisar X na lista L entre as posições esq e meio–1
    Senão
        Pesquisar X na lista L entre as posições meio+1 e dir
```

Uma rotina que traduza o algoritmo é, por exemplo, a função que se segue (Pesquisa_Binaria_R), que devolve a posição do elemento X (se X ∈ L) ou zero (se X ∉ L) :

```
function Pesquisa_Binaria ( X : Elemento ; L : Lista ; N : integer ) : integer ;
    function Pesquisa_Binaria_R ( X : Elemento ; L : Lista ; esq, dir : integer ) : integer ;
    var  i, meio : integer ;
    begin
        if  ( esq > dir )  then
        begin
            Pesquisa_Binaria_R := 0 ;      { pesquisar X numa lista vazia  ⇒ X ∉ L }
            Exit
        end ;
        meio := ( esq + dir ) div 2 ;
        if  X . Nome = L [meio] . Nome  then
            Pesquisa_Binaria_R := meio
        else
            if  X . Nome < L [meio] . Nome  then
                Pesquisa_Binaria_R := Pesquisa_Binaria_R ( X, L, esq, meio – 1 )
            else
                Pesquisa_Binaria_R := Pesquisa_Binaria_R ( X, L, meio + 1, dir )
    end ;
begin
    Pesquisa_Binaria := Pesquisa_Binaria_R ( X, L, 1, N )
end ;
```

Tal como na versão iterativa, e fazendo a mesma análise, conclui-se que a ordem de complexidade (pior dos casos) : O($\log_2$ N).

2.8. Ordenar uma lista

Este problema consiste em ordenar uma lista que, em principio, se encontra desordenada. Existem vários métodos para conseguir atingir tal objectivo, dos quais se destacam os seguintes : Selecção Linear, Bubblesort, Quicksort e Por Inserção. Os algoritmos associados aos dois primeiros são versões iterativas e os associados aos dois restantes são versões recursivas.

No estudo destes métodos vai-se admitir, tal como acontece nos anteriores, que se pretende ordenar a lista crescentemente, relativamente ao campo Chave (que será o campo **Nome**).

2.8.1. Selecção Linear

Este método consiste em determinar o menor elemento e colocá-lo na posição 1, depois determinar o segundo menor elemento e colocá-lo na posição 2, e assim sucessivamente até colocar o maior elemento na posição N.

Um possível algoritmo é o seguinte :

Dados de entrada : uma lista L e o tamanho da lista (N)

Dados de saída : a lista L modificada (apesar do tamanho e dos elementos se manter, alguns elementos estão em posições diferentes)

Algoritmo :

Para cada posição da lista L (i = 1, …, N) **Fazer**
    menor ← posição do i-ésimo menor elemento da lista L
    Trocar o elemento da posição **menor** com o da posição **i**

Uma rotina que traduza o algoritmo é, por exemplo, o procedimento que se segue, que recebe uma lista não ordenada (L) e o tamanho da lista (N) e, devolve actualizado a lista L (apesar do tamanho ser o mesmo, alguns dos elementos mudaram de posição) :

```
procedure  Ordenar_Seleccao_Linear ( var L : Lista ; N : integer ) ;
var i, j, menor : integer ;
    aux : Elemento ;
begin
    for i := 1 to N–1 do
    begin
        menor := i ;      { inicialmente o menor elemento está na posição i }
        for j := i+1 to N do    { percorrer os elementos desde a posição i + 1 }
            if ( L [menor] . Nome > L [ i ] . Nome ) then
                menor := j ;      { actualizar o menor, pois encontrado um elemento menor }
        aux := L [menor] ;        { trocar o elemento da posição menor com o da posição i }
        L [menor] := L [ i ] ;
        L [ i ] := aux ;
    end ;
end ;
```

Ordem de complexidade (pior dos casos) : $O(N^2)$.

$i = 1 ; j = 2, \ldots, N \Rightarrow N - 1$ comparações

$i = 2 ; j = 3, \ldots, N \Rightarrow N - 2$ comparações

. . .

$i = N - 1 ; j = N, \ldots, N \Rightarrow N - (N - 1) = 1$ comparação

Logo, o número de comparações é o seguinte :

$$C = (N-1) + (N-2) + \ldots + 1 = \frac{(N-1)+1}{2} \cdot (N-1) = \frac{N \cdot (N-1)}{2} = \frac{N^2 - N}{2}$$

Por outro lado, o número de atribuições é o seguinte :

$A = 4 \times (N - 1) + C$ { for i + for j }

Concluindo, temos que o número total de operações é o seguinte :

$$C + A = \frac{N^2 - N}{2} + 4 \times (N - 1) + \frac{N^2 - N}{2} = 2 \times \frac{N^2 - N}{2} + 4N - 4 = N^2 + 3N - 4$$

Ou seja, o algoritmo é da ordem de complexidade de N^2.

2.8.2. Bubblesort

Este método consiste em fazer trocas sucessivas entre dois elementos seguidos (vizinhos), de forma que o maior fique depois do menor. Este processo (iteração) é repetido percorrendo todos os elementos da lista as vezes que for necessário, até que não se faça qualquer troca, o que acontece quando a lista está ordenada. Note-se que após a iteração k se concluir, o k últimos elementos ficam ordenados (após a conclusão da 1ª iteração, o maior elemento fica na N-ésima (última) posição, e assim sucessivamente); portanto, cada iteração apenas vai analisar até uma determinada posição.

Um possível algoritmo é o seguinte :

Dados de entrada : uma lista L e o tamanho da lista (N)

Dados de saída : a lista L modificada (apesar do tamanho e dos elementos se manter, alguns elementos estão em posições diferentes)

Algoritmo :

```
limite ← N        { inicialmente vai-se analisar até à posição N }
Repetir
    trocas ← 0
    Para cada posição da lista L (i = 1, …, limite–1) Fazer
        Se L[i + 1] . Chave < L [i] . Chave Então
            Trocar o elemento da posição i+1 com o da posição i
            trocas ← trocas + 1
    Como o elemento que está na posição limite está ordenado : limite ← limite – 1
Até (trocas = 0)      { isto acontece quando o teste "Se…Então…" nunca for verdadeiro }
```

Uma rotina que traduza o algoritmo é, por exemplo, o procedimento que se segue, que recebe uma lista não ordenada (L) e o tamanho da lista (N) e, devolve actualizado a lista L (apesar do tamanho ser o mesmo, alguns dos elementos mudaram de posição) :

```
procedure  Ordenar_Bubblesort ( var L : Lista ; N : integer ) ;
var i, limite, trocas : integer ;
    aux : Elemento ;
begin
    limite := N ;      { inicialmente vai-se analisar até à posição N }
    repeat
        trocas := 0 ;
        for i := 1 to limite – 1 do
            if ( L [ i + 1 ] . Nome < L [ i ] . Nome ) then
            begin
                aux := L [ i + 1 ] ;
                L [ i + 1] := L [ i ] ;
                L [ i ] := aux ;
                trocas := trocas + 1
            end ;
        { o elemento que está na posição limite está na posição correcta }
        limite := limite – 1
    until ( trocas = 0 )
end ;
```

Ordem de complexidade (pior dos casos — lista ordenada pela ordem inversa) : $O(N^2)$.

1ª iteração : i = 1, . . ., N – 1 $\Rightarrow$ N – 1 comparações
2ª iteração : i = 1, . . ., N – 2 $\Rightarrow$ N – 2 comparações
. . .
k-ésima iteração : i = 1, . . ., N – k $\Rightarrow$ N – (N – k) = k comparações
. . .
última iteração : i = 1, . . ., 1 $\Rightarrow$ N – (N – 1) = 1 comparação
Logo, o número de comparações é o seguinte :

$$C = (N-1) + (N-2) + \ldots + 1 = \frac{(N-1)+1}{2} \cdot (N-1) = \frac{N \cdot (N-1)}{2} = \frac{N^2 - N}{2}$$

Por outro lado, o número de atribuições é o seguinte :

$A = (N – 1) + 4 \times C$ { repeat + for i }

Concluindo, temos que o número total de operações é o seguinte :

$$C + A = \frac{N^2 - N}{2} + (N-1) + 4 \times \frac{N^2 - N}{2} = 5 \times \frac{N^2 - N}{2} + N - 1 = \frac{5N^2 - 3N - 2}{2}$$

Ou seja, o algoritmo é da ordem de complexidade de N^2.

2.8.3. Quicksort

Este método consiste em considerar o elemento que está ao meio da lista e trocar os elementos que estão à sua esquerda que são maiores que ele, com os elementos que estão à sua direita que são menores. Este processo é aplicado recursivamente às sub-listas que ainda não estejam ordenadas.

Um possível algoritmo é o seguinte :

Dados de entrada : uma lista L e as posições do elemento mais à esquerda (esq) e do elemento mais à direita (dir)

Dados de saída : a lista L modificada (apesar do tamanho e dos elementos se manter, alguns elementos estão em posições diferentes)

Algoritmo :

```
meio ← elemento que está ao meio da lista – L [ (esq + dir) / 2 ]
Repetir
    Procurar um elemento à esquerda de meio que seja maior que o elemento ao meio
    Procurar um elemento à direita de meio que seja menor que o elemento ao meio
    Trocar os dois elementos encontrados antes, entre si
Até não ser possível fazer qualquer troca
Se existe alguma sub-lista à esquerda do meio que não está ordenada Então
    Aplicar recursivamente o método àquela sub-lista
Se existe alguma sub-lista à direita do meio que não está ordenada Então
    Aplicar recursivamente o método àquela sub-lista
```

Uma rotina que traduza o algoritmo é, por exemplo, o procedimento que se segue, que recebe uma lista não ordenada (L) e o tamanho da lista (N) e, devolve actualizado a lista L (apesar do tamanho ser o mesmo, alguns dos elementos mudaram de posição) :

```
procedure Ordenar_Quicksort ( var L : Lista ; N : integer ) ;
    procedure Quicksort (var L : Lista ; esq, dir : integer ) ;
    var i, j : integer ;
        Meio, aux : Elemento ;
    begin
        i := esq ;
        j := dir ;
        Meio := L [ (esq + dir) div 2 ] ;
        repeat
            while L [ i ] . Nome < Meio . Nome do
                i := i + 1 ;
            while L [ j ] . Nome > Meio . Nome do
                j := j – 1 ;
            if ( i <= j ) then
            begin
                aux := L [ i ] ;
                L [ i ] := L [ j ] ;
                L [ j ] := aux ;
                i := i + 1 ;
                j := j – 1
            end
        until ( i > j ) ;
        if ( esq < j ) then
            Quicksort ( esq, j ) ;
        if ( i < dir ) then
            Quicksort ( i, dir )
    end ;
begin
    Quicksort ( 1, N )
end ;
```

2.8.4. Por Inserção

Este método consiste em tomar um dos elementos da lista, ordenar a sub-lista composta pelos restantes elementos e depois inserir o elemento tomada antes na sub-lista ordenada. Este processo é executado recursivamente e utilizando o algoritmo para inserir um elemento numa lista ordenada estudado antes (ver Inserir um elemento numa lista– pág. 5).

Um possível algoritmo é o seguinte :

Dados de entrada : uma lista L e o tamanho da lista (N)

Dados de saída : a lista L ordenada (apesar do tamanho e dos elementos se manter, alguns elementos estão em posições diferentes)

Algoritmo :

```
Se  N > 1  Então
    Ordenar a sub-lista de L com tamanho N–1 ( da posição 1 à posição N–1 )
    Inserir o elemento da posição N, L [N], na sub-lista ordenada no passo anterior
```

Uma rotina que traduza o algoritmo é, por exemplo, o procedimento que se segue, que recebe uma lista não ordenada (L) e o tamanho da lista (N) e, devolve actualizado a lista L (apesar do tamanho ser o mesmo, alguns dos elementos mudaram de posição) :

```
procedure  Ordenar_Por_Insercao  ( var  L  :  Lista ;  N  :  integer ) ;
var  k  :  integer ;
begin
    if  ( N  >  1 )  then
    begin
        k  :=  N – 1 ;
        Ordenar_Por_Insercao ( L, k ) ;
        Inserir_Ordem ( L [N], L, k )
    end
end ;
```

ÍNDICE