Capítulo 5

Listas

'Who are you?' said the Caterpillar.
This was not an encouraging opening for a conversation.
Alice replied, rather shyly, 'I – I hardly know, Sir, just at present – at least I know who I was when I got up this morning, but I think I must have been changed several times since then.'

Lewis Carroll, *Alice's Adventures in Wonderland*

Neste capítulo abordamos um novo tipo estruturado de dados, a lista. Uma *lista* é uma sequência de elementos de qualquer tipo. Esta definição é semelhante à de um tuplo. Contudo, em oposição aos tuplos, as listas são tipos *mutáveis*, no sentido de que podemos alterar destrutivamente os seus elementos. Na maioria das linguagens de programação existem tipos de certo modo semelhantes às listas do Python, sendo conhecidos por *vetores* ou por *tabelas*[1].

5.1 Listas em Python

Uma *lista*, em Python designada por `list`, é uma sequência de elementos. Em Python, a representação externa de uma lista é definida sintaticamente pelas seguintes expressões em notação BNF[2]:

[1] Em inglês, "array".
[2] Devemos notar que esta definição não está completa e que a definição de ⟨dicionário⟩ é apresentada no Capítulo 7.

Operação	*Tipo dos argumentos*	*Valor*
l_1 `+` l_2	Listas	A concatenação das listas l_1 e l_2.
l `*` i	Lista e inteiro	A repetição i vezes da lista l.
l`[`i_1`:`i_2`]`	Lista e inteiros	A sub-lista de l entre os índices i_1 e $i_2 - 1$.
`del(`els`)`	Lista e inteiro(s)	Em que els pode ser da forma l`[`i`]` ou l`[`i_1`:`i_2`]`. Remove os elemento(s) especificado(s) da lista l.
e `in` l	Universal e lista	`True` se o elemento e pertence à lista l; `False` em caso contrário.
e `not in` l	Universal e lista	A negação do resultado da operação e `in` l.
`list(`a`)`	Tuplo ou dicionário ou cadeia de caracteres	Transforma o seu argumento numa lista. Se não forem fornecidos argumentos, o seu valor é a lista vazia.
`len(`l`)`	Lista	O número de elementos da lista l.

Tabela 5.1: Operações embutidas sobre listas.

⟨lista⟩ ::= `[]` |
`[`⟨elementos⟩`]`

⟨elementos⟩ ::= ⟨elemento⟩ |
⟨elemento⟩`,` ⟨elementos⟩

⟨elemento⟩ ::= ⟨expressão⟩ |
⟨tuplo⟩ |
⟨lista⟩ |
⟨dicionário⟩

As seguintes entidades representam listas em Python `[1, 2, 3]`, `[2, (1, 2)]`, `['a']`, `[]`. A lista `[]` tem o nome de *lista vazia*. Em oposição aos tuplos, uma lista de um elemento não contém a virgula. Tal como no caso dos tuplos, os elementos de uma lista podem ser, por sua vez, outras listas, pelo que a seguinte entidade é uma lista `[1, [2], [[3]]]`.

Sobre as listas podemos efetuar as operações apresentadas na Tabela 5.1. Note-se a semelhança entre estas operações e as operações sobre tuplos e sobre cadeias de caracteres, apresentadas, respetivamente, nas tabelas 4.1 e 4.3, sendo a exceção a operação `del` que existe para listas e que não existe nem para tuplos nem para cadeias de caracteres.

A seguinte interação mostra a utilização de algumas operações sobre listas:

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [[4, 5]]
>>> lst = lst1 + lst2
>>> lst
[1, 2, 3, [4, 5]]
>>> len(lst)
4
>>> lst[3]
[4, 5]
>>> lst[3][0]
4
>>> lst[2] = 'a'
>>> lst
[1, 2, 'a', [4, 5]]
>>> del(lst[1])
>>> lst
[1, 'a', [4, 5]]
>>> del(lst[1:])
>>> lst
[1]
```

Sendo as listas entidades mutáveis, podemos alterar qualquer dos seus elementos. Na interação anterior, atribuímos a cadeia de caracteres `'a'` ao elemento da lista `lst` com índice 2 (`lst[2] = 'a'`), removemos da lista `lst` o elemento com índice 1 (`del(lst[1])`), após o que removemos da lista `lst` todos os elementos com índice igual ou superior a 1 (`del(lst[1:])`).

Consideremos agora a interação:

```
>>> lst1 = [1, 2, 3, 4]
>>> lst2 = lst1
>>> lst1
[1, 2, 3, 4]
>>> lst2
[1, 2, 3, 4]
>>> lst2[1] = 'a'
>>> lst2
[1, 'a', 3, 4]
```

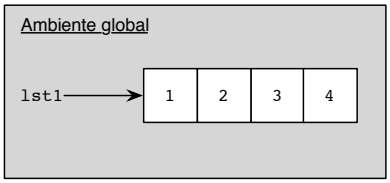

Figura 5.1: Ambiente após a definição da lista `lst1`.

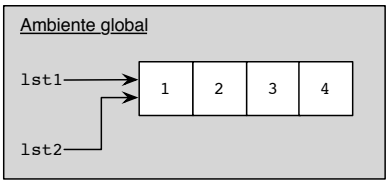

Figura 5.2: Ambiente após a atribuição `lst2 = lst1`.

```
>>> lst1
[1, 'a', 3, 4]
```

Nesta interação, começamos por definir a lista `lst1`, após o que definimos a lista `lst2` como sendo igual a `lst1`. Ao alterarmos a lista `lst2` estamos indiretamente a alterar a lista `lst1`. Este comportamento, aparentemente estranho, é explicado quando consideramos o ambiente criado por esta interação. Ao criar a lista `lst1`, o Python dá origem ao ambiente apresentado na Figura 5.1 (usamos uma representação esquemática para listas que é semelhante à usada para os tuplos). A instrução `lst2 = lst1`, define um novo nome, `lst2`, como sendo o valor de `lst1`. Na realidade, a semântica da instrução de atribuição especifica que ao executar a instrução `lst2 = lst1`, o Python começa por avaliar a expressão `lst1` (um nome), sendo o seu valor a entidade associada ao nome (a localização em memória da lista `[1, 2, 3, 4]`), após o que cria o nome `lst2`, associando-o ao valor desta expressão. Esta instrução origina o ambiente apresentado na Figura 5.2. As listas `lst1` e `lst2` correspondem a *pseudónimos*[3] para a mesma entidade. Assim, ao alterarmos uma delas estamos implicitamente a alterar a outra (Figura 5.3). Esta situação não se verifica com tuplos nem com cadeias de caracteres pois estes são estruturas imutáveis.

[3]Do inglês, "alias".

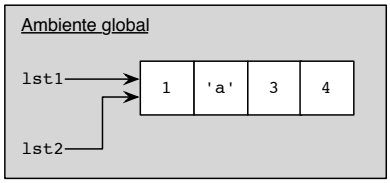

Figura 5.3: Ambiente após a alteração de `lst1`.

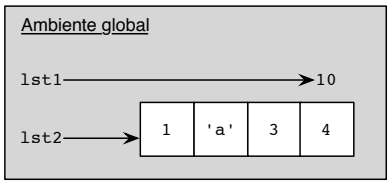

Figura 5.4: Ambiente após nova alteração de `lst1`.

Suponhamos agora que continuávamos a interação anterior do seguinte modo:

```
>>> lst1 = 10
>>> lst1
10
>>> lst2
[1, 'a', 3, 4]
```

Esta nova interação dá origem ao ambiente apresentado na Figura 5.4. A variável `lst1` passa a estar associada ao inteiro `10` e o valor da variável `lst2` mantém-se inalterado. A segunda parte deste exemplo mostra a diferença entre alterar um elemento de uma lista que é partilhada por várias variáveis e alterar uma dessas variáveis.

5.2 Métodos de passagem de parâmetros

Com base no exemplo anterior estamos agora em condições de analisar de um modo mais detalhado o processo de comunicação com funções. Como sabemos, quando uma função é avaliada (ou chamada) estabelece-se uma correspondência entre os parâmetros concretos e os parâmetros formais, associação essa que é

feita com base na posição que os parâmetros ocupam na lista de parâmetros. O processo de ligação entre os parâmetros concretos e os parâmetros formais é denominado *método de passagem de parâmetros*. Existem vários métodos de passagem de parâmetros. Cada linguagem de programação utiliza um, ou vários, destes métodos para a comunicação com funções. O Python utiliza apenas a passagem por valor.

5.2.1 Passagem por valor

Quando um parâmetro é passado por *valor*, o valor do parâmetro concreto é calculado (independentemente de ser uma constante, uma variável ou uma expressão mais complicada), e esse valor é associado com o parâmetro formal correspondente. Ou seja, utilizando a passagem por valor, a função recebe o valor de cada um dos parâmetros e nenhuma informação adicional.

Um parâmetro formal em que seja utilizada a passagem por valor comporta-se, dentro da sua função, como um nome local que é inicializado com o início da avaliação da função.

Para exemplificar, consideremos a função `troca` definida do seguinte modo:

```
def troca(x, y):
    print('antes da troca: x =', x, 'y =', y)
    x, y = y, x  # os valores são trocados
    print('depois da troca: x =', x, 'y =', y)
```

e a seguinte interação que utiliza a função `troca`:

```
>>> x = 3
>>> y = 10
>>> troca(x, y)
antes da troca: x = 3 y = 10
depois da troca: x = 10 y = 3
>>> x
3
>>> y
10
```

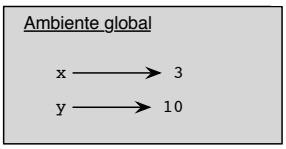

Figura 5.5: Ambiente global antes da invocação de `troca`.

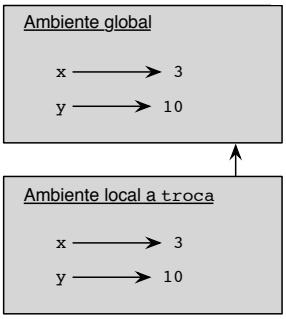

Figura 5.6: Ambiente local criado com a invocação de `troca`.

As duas primeiras linhas desta interação têm como efeito a criação dos nomes `x` e `y` no ambiente global como se mostra na Figura 5.5. Quando o Python invoca a função `troca`, avalia os parâmetros concretos e associa os parâmetros concretos aos parâmetros formais da função `troca`, criando o ambiente local que se mostra na Figura 5.6.

A instrução de atribuição `x, y = y, x` executada pela função `troca`, altera o ambiente local como se mostra na Figura 5.7, não modificando o ambiente global. Com efeito, recorde-se da Secção 2.3 que ao encontrar esta instrução, o Python avalia as expressões à direita do símbolo "`=`", as quais têm, respetivamente, os valores `10` e `3`, após o que atribui estes valores, respetivamente, às variáveis `x` e `y`. Esta instrução tem pois o efeito de trocar os valores das variáveis `x` e `y`. Os valores dos nomes locais `x` e `y` são alterados, mas isso não vai afetar os nomes `x` e `y` que existiam antes da avaliação da função `troca`.

Quando a função `troca` termina a sua execução o ambiente que lhe está asso-

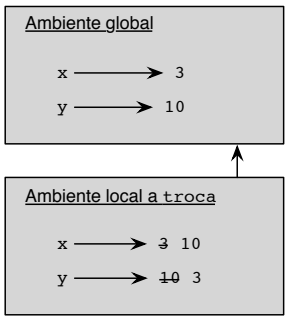

Figura 5.7: Ambientes após a execução da instrução de atribuição.

ciado desaparece, voltando-se à situação apresentada na Figura 5.5. Ou seja, quando se utiliza a passagem por valor, a única ligação entre os parâmetros concretos e os parâmetros formais é uma associação unidirecional de valores. É *unidirecional* porque é feita do ponto de chamada para a função.

5.2.2 Passagem por referência

Quando um parâmetro é passado por *referência*, o que é associado ao parâmetro formal correspondente não é o valor do parâmetro concreto, mas sim a localização na memória do computador que contém o seu valor. Utilizando passagem por referência, os parâmetros formais e os parâmetros concretos vão *partilhar* o mesmo local (dentro da memória do computador) e, consequentemente, qualquer modificação feita aos parâmetros formais reflete-se nos parâmetros concretos. Um parâmetro formal em que seja utilizada a passagem por referência corresponde *à mesma variável* que o parâmetro concreto correspondente, apenas, eventualmente, com outro nome, ou seja, o parâmetro concreto e o parâmetro formal são pseudónimos da mesma localização em memória.

Embora o Python apenas utilize a passagem por valor, o efeito da passagem por referência pode ser parcialmente[4] ilustrado quando se utiliza um parâmetro concreto correspondente a uma estrutura mutável. Consideremos a função `troca_2` que recebe como argumentos uma lista e dois inteiros e que troca os elementos

[4]Parcialmente, porque existem aspetos na passagem por referência que não são simuláveis em Python.

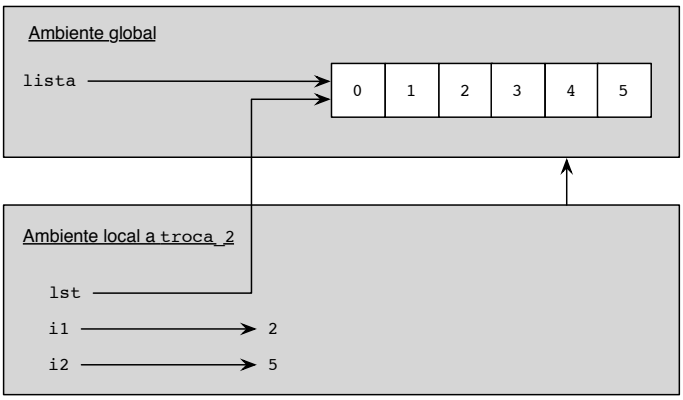

Figura 5.8: Ambientes após o início da execução de `troca_2`.

da lista cujos índices correspondem a esses inteiros:

```
def troca_2(lst, i1, i2):
    lst[i1], lst[i2] = lst[i2], lst[i1]  # os valores são trocados
```

Com esta função podemos originar a seguinte interação:

```
>>> lista = [0, 1, 2, 3, 4, 5]
>>> troca_2(lista, 2, 5)
>>> lista
[0, 1, 5, 3, 4, 2]
```

Neste caso, tendo em atenção a discussão apresentada na página 136, o parâmetro concreto `lst` partilha o mesmo espaço de memória que a variável global `lista`, pelo que qualquer alteração a `lst` reflete-se na variável `lista`. Na Figura 5.8 mostramos os ambientes criados no início da execução da função `troca_2`.

5.3 O Crivo de Eratóstenes

O *Crivo de Eratóstenes* é um algoritmo para calcular números primos que, segundo a tradição, foi criado pelo matemático grego Eratóstenes (c. 285–194

a.C.), o terceiro bibliotecário chefe da Biblioteca de Alexandria. Para um dado inteiro positivo n, o algoritmo calcula todos os números primos inferiores a n. Para isso, começa por criar uma lista com todos os inteiros positivos de 2 a n e seleciona o primeiro elemento da lista, o número 2. Enquanto o número selecionado não for maior que $\sqrt{n}$ executam-se as seguintes ações: (a) removem-se da lista todos os múltiplos do número selecionado; (b) passa-se ao número seguinte na lista. No final do algoritmo, quando o número selecionado for superior a $\sqrt{n}$, a lista apenas contém números primos.

Vamos agora escrever uma função, `crivo` que recebe um inteiro positivo n e calcula a lista de números primos inferiores a n de acordo com o Crivo de Eratóstenes.

A nossa função deve começar por criar a lista com os inteiros entre 2 e n. Para isso, podemos pensar em utilizar as seguintes instruções:

```
lista = []
for i in range(2, n + 1):
    lista = lista + [i]
```

No entanto, sabendo que a função `range(2, n + 1)` devolve a sequência de inteiros de 2 a `n`, podemos recorrer à função embutida `list` para obter o mesmo resultado com a seguinte instrução:

```
lista = list(range(2, n + 1))
```

Após a lista criada, a função percorre a lista para os elementos inferiores ou iguais a $\sqrt{n}$, removendo da lista todos os múltiplos desse elemento. Partindo do princípio da existência de uma função chamada `remove_multiplos` que recebe uma lista e um índice e modifica essa lista, removendo todos os múltiplos do elemento da lista na posição indicada pelo índice, a função `crivo` será:

```
def crivo(n):
    from math import sqrt
    lista = list(range(2, n+1))
    i = 0
    while lista[i] <= sqrt(n):
        remove_multiplos(lista, i)
        i = i + 1
    return lista
```

É importante perceber a razão de não termos recorrido a um ciclo `for` na remoção dos elementos da lista. Um ciclo `for` calcula de antemão o número de vezes que o ciclo será executado, fazendo depois esse número de passagens pelo ciclo com o valor da variável de controlo seguindo uma progressão aritmética. Este é o ciclo ideal para fazer o processamento de tuplos e de cadeias de caracteres e, tipicamente, é também o ciclo ideal para fazer o processamento de listas. No entanto, no nosso exemplo, o número de elementos da lista vai diminuindo à medida que o processamento decorre, e daí a necessidade de fazer o controlo da execução do ciclo de um outro modo. Sabemos que temos que processar os elementos da lista não superiores a $\sqrt{n}$ e é este o mecanismo de controle que usámos no ciclo `while`.

Vamos agora concentrar-nos no desenvolvimento da função `remove_multiplos`. Apesar da discussão que acabámos de apresentar em relação à utilização de uma instrução `while`, nesta função recorremos a um ciclo `for` que percorre os elementos da lista, do final da lista para o início, deste modo podemos percorrer todos os elementos relevantes da lista, sem termos que nos preocupar com eventuais alterações dos índices, originadas pela remoção de elementos da lista. Para exemplificar o nosso raciocínio, suponhamos que estamos a percorrer a lista `[2, 3, 4, 5, 6, 7, 8, 9, 10]`, removendo os múltiplos do seu primeiro elemento. O primeiro elemento considerado é `lst[8]`, cujo valor é `10`, pelo que este elemento é removido da lista, o próximo elemento a considerar é `lst[7]`, cujo valor é `9`, pelo que, não sendo um múltiplo de `2`, este mantém-se na lista, segue-se o elemento `lst[6]`, que é removido da lista e assim sucessivamente. O ciclo termina no segundo elemento da lista, `lst[1]`.

```
def remove_multiplos(l, i):
    el = l[i]
    for j in range(len(l)-1, i, -1):
        if l[j] % el == 0:
            del(l[j])
```

Com a função `crivo` podemos gerar a interação:

```
>>> crivo(60)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59]
```

5.4 Algoritmos de procura

> *'Just look along the road, and tell me if you can see either of them.'*
> *'I see nobody on the road,' said Alice.*
> *'I only wish I had such eyes,' the King remarked in a fretful tone.*
> *'To be able to see Nobody! And at that distance, too! Why, it's as much as I can do to see real people, by this light!'*
>
> Lewis Carroll, *Through the Looking Glass*

A procura de informação é uma das nossas atividades quotidianas. Procuramos páginas com determinadas palavras no Google, palavras em dicionários, livros em bibliotecas, produtos em supermercados, etc. Com o aumento constante da quantidade de informação armazenada por computadores, é natural que a procura de informação se tenha tornado uma das atividades preponderantes num sistema computacional. Embora a localização de um dado elemento entre um conjunto de elementos pareça ser muito simples, o desenvolvimento de programas eficientes de procura levanta muitos problemas. Como exemplo, suponhamos que procurávamos a palavra "Python" no Google, a nossa procura devolve cerca de 198 milhões de resultados, tendo a procura demorado 0.26 segundos. Imagine-se quanto demoraria esta procura se o Google não utilizasse métodos de procura altamente sofisticados.

A procura de informação é frequentemente executada recorrendo a listas. Nesta secção, apresentamos dois métodos para procurar um elemento numa lista, a procura sequencial e a procura binária. Para estes dois tipos de procura, desenvolvemos uma função com o nome `procura` que recebe como argumentos uma lista (`lst`) e um elemento (a que chamamos `chave`). Esta função devolve um

inteiro positivo correspondente ao índice do elemento da lista cujo valor é igual a `chave` (estamos a partir do pressuposto que a lista `lst` não tem elementos repetidos). Convencionamos que se a lista não contiver nenhum elemento igual a `chave`, a função devolve `-1`.

5.4.1 Procura sequencial

Uma das maneiras mais simples de procurar um dado elemento numa lista consiste em começar no primeiro elemento da lista e comparar sucessivamente o elemento procurado com o elemento na lista. Este processo é repetido até que o elemento seja encontrado ou o fim da lista seja atingido. Este método de procura é chamado *procura sequencial*, ou *procura linear*. A função `procura` utilizando a procura sequencial será:

```
def procura(lst, chave):
    for i in range(len(lst)):
        if lst[i] == chave:
            return i
    return -1
```

Com a qual podemos obter a interação:

```
>>> lst = [2, 3, 1, -4, 12, 9]
>>> procura(lst, 1)
2
>>> procura(lst, 5)
-1
```

5.4.2 Procura binária

A procura sequencial é muito simples, mas pode exigir a inspeção de todos os elementos da lista, o que acontece sempre que o elemento que estamos a procurar não se encontra na lista.

A *procura binária* é uma alternativa, mais eficiente, à procura sequencial, exigindo, contudo, que os elementos sobre os quais a procura está a ser efetuada se

encontrem ordenados. Utilizando a procura binária, consideramos o elemento que se encontra no meio da lista:

1. Se o elemento no meio da lista for igual ao elemento que estamos a procurar, então a procura termina;
2. Se o elemento no meio da lista é menor do que o elemento que estamos a procurar então podemos garantir que o elemento que estamos a procurar não se encontra na primeira metade da lista. Repetimos então o processo da procura binária para a segunda metade da lista;
3. Se este elemento é maior do que o elemento que estamos a procurar então podemos garantir que o elemento que estamos a procurar não se encontra na segunda metade da lista. Repetimos então o processo da procura binária para a primeira metade da lista.

Note-se que, em cada passo, a procura binária reduz o número de elementos a considerar para metade (e daí o seu nome).

A seguinte função utiliza a procura binária. Esta função pressupõe que a lista `lst` está ordenada. As variáveis `linf` e `lsup` representam, respetivamente, o menor e o maior índice da gama dos elementos que estamos a considerar.

```
def procura(lst, chave):
    linf = 0
    lsup = len(lst) - 1

    while linf <= lsup:
        meio = (linf + lsup) // 2
        if chave == lst[meio]:
            return meio
        elif chave > lst[meio]:
            linf = meio + 1
        else:
            lsup = meio - 1
    return -1
```

A procura binária exige que mantenhamos uma lista ordenada contendo os elementos em que queremos efetuar a procura, consequentemente, o custo com-

putacional da inserção de um novo elemento na lista é maior se usarmos este método do que se usarmos a procura sequencial. Contudo, como as procuras são normalmente mais frequentes do que as inserções, é preferível utilizar uma lista ordenada.

5.5 Algoritmos de ordenação

Um conjunto de elementos é normalmente ordenado para facilitar a procura. Na nossa vida quotidiana, ordenamos as coisas para tornar a procura mais fácil, e não porque somos arrumados. Em programação, a utilização de listas ordenadas permite a escrita de algoritmos de procura mais eficientes, por exemplo, a procura binária em lugar da procura sequencial. Nesta secção, abordamos o estudo de alguns algoritmos para ordenar os valores contidos numa lista.

Os algoritmos de ordenação podem ser divididos em dois grandes grupos, os algoritmos de ordenação interna e os algoritmos de ordenação externa. Os algoritmos de *ordenação interna* ordenam um conjunto de elementos que estão simultaneamente armazenados em memória, por exemplo, numa lista. Os algoritmos de *ordenação externa* ordenam elementos que, devido à sua quantidade, não podem estar simultaneamente em memória, estando parte deles armazenados algures no computador (num ficheiro, para ser mais preciso). Neste segundo caso, em cada momento apenas estão em memória parte dos elementos a ordenar. Neste livro apenas consideramos algoritmos de ordenação interna.

Na nossa apresentação dos algoritmos de ordenação, vamos supor que desejamos ordenar uma lista de números inteiros. O nosso programa de ordenação, `prog_ordena`, começa por ler os números a ordenar (realizado através da função `le_elementos`), números esses que são guardados numa lista. Em seguida, ordena os elementos, recorrendo à função `ordena` (a qual ainda não foi desenvolvida) e escreve-os por ordem crescente. A ordenação é efetuada pela função `ordena`, que recebe uma lista e devolve, por referência, a lista ordenada.

```
def prog_ordena():
    elementos = le_elementos()
    print('Elementos fornecidos: ', elementos)
    ordena(elementos)
    print('Elementos ordenados : ', elementos)
```

Antes de abordar a função `ordena`, vamos considerar a função `le_elementos`. Esta função solicita ao utilizador a introdução dos elementos a ordenar, separados por espaços em branco, e devolve a lista contendo os inteiros fornecidos. A função contempla a possibilidade do utilizador escrever qualquer número de espaços em branco antes ou depois de cada um dos elementos a ordenar. Após a leitura da cadeia de caracteres (`el`) contendo a sequência de elementos fornecidos pelo utilizador, a função começa por ignorar todos os espaços em branco iniciais existentes em `el`. Ao encontrar um símbolo que não corresponda ao espaço em branco, esta função cria a cadeia de caracteres correspondente aos algarismos do número, transformando depois essa cadeia de caracteres num número recorrendo à função `eval`. Depois de obtido um número, os espaços em branco que o seguem são ignorados. É importante dizer que esta função não está preparada para lidar com dados fornecidos pelo utilizador que não sigam o formato indicado.

```
def le_elementos():
    print('Introduza os elementos a ordenar')
    el = input('separados por espaços\n? ')
    elementos = []
    i = 0
    # Ignora espaços em branco iniciais
    while i < len(el) and el[i] == ' ':
        i = i + 1

    while i < len(el):
        # Transforma um elemento num inteiro
        num = ''
        while i < len(el) and el[i] != ' ':
            num = num + el[i]
            i = i + 1
        elementos = elementos + [eval(num)]

        # Ignora espaços em branco depois do número
        while i < len(el) and el[i] == ' ':
            i = i + 1
    return elementos
```

Com uma função de ordenação apropriada, o programa `prog_ordena` permite gerar a interação:

```
>>> prog_ordena()
Introduza os inteiros a ordenar
separados por espaços
?      45 88775     665443 34 122 1       23
Elementos fornecidos: [45, 88775, 665443, 34, 122, 1, 23]
Elementos ordenados: [1, 23, 34, 45, 122, 88775, 665443]
```

Para apresentar os algoritmos de ordenação, vamos desenvolver várias versões da função `ordena`. Esta função recebe como parâmetro uma lista contendo os elementos a ordenar (correspondendo ao parâmetro formal `lst`), a qual, após execução da função, contém os elementos ordenados. Cada versão que apresentamos desta função corresponde a um algoritmo de ordenação.

5.5.1 Ordenação por borbulhamento

O primeiro algoritmo de ordenação que consideramos é conhecido por ordenação *por borbulhamento*[5]. A ideia subjacente a este algoritmo consiste em percorrer os elementos a ordenar, comparando elementos adjacentes, trocando os pares de elementos que se encontram fora de ordem, ou seja, que não estão ordenados. De um modo geral, uma única passagem pela sequência de elementos não ordena a lista, pelo que é necessário efetuar várias passagens. A lista encontra-se ordenada quando se efetua uma passagem completa em que não é necessário trocar a ordem de nenhum elemento. A razão por que este método é conhecido por "ordenação por borbulhamento" provém do facto de os menores elementos da lista se movimentarem no sentido do início da lista, como bolhas que se libertam dentro de um recipiente com um líquido.

A seguinte função utiliza a ordenação por borbulhamento. Esta função utiliza a variável de tipo lógico, `nenhuma_troca`, cujo valor é `False` se, durante uma passagem pela lista, se efetua alguma troca de elementos, e tem o valor `True` em caso contrário. Esta variável é inicializada para `False` na segunda linha da função de modo a que o ciclo `while` seja executado.

[5] Em inglês, "bubble sort".

Dado que em cada passagem pelo ciclo colocamos o maior elemento da lista na sua posição correta[6], cada passagem subsequente pelos elementos da lista considera um valor a menos e daí a razão da variável `maior_indice`, que contém o maior índice até ao qual a ordenação se processa.

```
def ordena(lst):

    maior_indice = len(lst) - 1
    nenhuma_troca = False # garante que o ciclo while é executado

    while not nenhuma_troca:
        nenhuma_troca = True
        for i in range(maior_indice):
            if lst[i] > lst[i+1]:
                lst[i], lst[i+1] = lst[i+1], lst[i]
                nenhuma_troca = False
        maior_indice = maior_indice - 1
```

5.5.2 Ordenação Shell

Uma variante da ordenação por borbulhamento, a ordenação *Shell*[7] consiste em comparar e trocar, não os elementos adjacentes, mas sim os elementos separados por um certo intervalo. Após uma ordenação completa, do tipo borbulhamento, com um certo intervalo, esse intervalo é dividido ao meio e o processo repete-se com o novo intervalo. Este processo é repetido até que o intervalo seja 1 (correspondendo a uma ordenação por borbulhamento). Como intervalo inicial, considera-se metade do número de elementos a ordenar. A ordenação *Shell* é mais eficiente do que a ordenação por borbulhamento, porque as primeiras passagens que consideram apenas um subconjunto dos elementos a ordenar permitem uma arrumação grosseira dos elementos da lista e as últimas passagens, que consideram todos os elementos, já os encontram parcialmente ordenados. A seguinte função corresponde à ordenação *Shell*:

[6] Como exercício, o leitor deve convencer-se deste facto.
[7] Em inglês, "Shell sort", em honra ao seu criador Donald Shell [Shell, 1959].

```
def ordena(lst):
    intervalo = len(lst) // 2
    while not intervalo == 0:
        nenhuma_troca = False
        while not nenhuma_troca:
            nenhuma_troca = True
            for i in range(len(lst)-intervalo):
                if lst[i] > lst[i+intervalo]:
                    lst[i], lst[i+intervalo] = \
                        lst[i+intervalo], lst[i]
                    nenhuma_troca = False
        intervalo = intervalo // 2
```

5.5.3 Ordenação por seleção

Uma terceira alternativa de ordenação que iremos considerar, a ordenação *por seleção*[8], consiste em percorrer os elementos a ordenar e, em cada passagem, colocar um elemento na sua posição correta. Na primeira passagem, coloca-se o menor elemento na sua posição correta, na segunda passagem, o segundo menor, e assim sucessivamente. A seguinte função efetua a ordenação por selecão:

```
def ordena(lst):
    for i in range(len(lst)):
        pos_menor = i
        for j in range(i + 1, len(lst)):
            if lst[j] < lst[pos_menor]:
                pos_menor = j
        lst[i], lst[pos_menor] = lst[pos_menor], lst[i]
```

[8]Em inglês, "selection sort".

5.6 Exemplo 1

Para ilustrar a utilização de algoritmos de procura e de ordenação, vamos desenvolver um programa que utiliza duas listas, `nomes` e `telefones`, contendo, respetivamente nomes de pessoas e números de telefones. Assumimos que o número de telefone de uma pessoa está armazenado na lista `telefones` na mesma posição que o nome dessa pessoa está armazenado na lista `nomes`. Por exemplo, o número de telefone da pessoa cujo nome é `nomes[3]`, está armazenado em `telefones[3]`. Listas com esta propriedade são chamadas *listas paralelas*. Assumimos também que cada pessoa tem, no máximo, um número de telefone.

O programa utiliza uma abordagem muito simplista, na qual as listas são definidas dentro do próprio programa. Uma abordagem mais realista poderia ser obtida através da leitura desta informação do exterior, o que requer a utilização de ficheiros, os quais são apresentados no Capítulo 9.

O programa começa por ordenar as listas de nomes e de telefones usando a função `ordena_nomes`, após o que interacciona com um utilizador, ao qual fornece o número de telefone da pessoa cujo nome é fornecido ao programa. Esta interação é repetida até que o utilizador forneça, como nome, a palavra `fim`. Para procurar o nome de uma pessoa, o programa utiliza a função `procura` correspondente à procura binária, a qual foi apresentada na secção 5.4.2, e que não é repetida no nosso programa.

O algoritmo para ordenar as listas de nomes e de telefones corresponde à ordenação por seleção apresentada na secção 5.5.3. Contudo, como estamos a lidar com listas paralelas, o algoritmo que usamos corresponde a uma variante da ordenação por seleção. Na realidade, a função `ordena_nomes` utiliza a ordenação por seleção para ordenar os nomes das pessoas, a lista que é usada para procurar os nomes, mas sempre que dois nomes são trocados, a mesma ação é aplicada aos respetivos números de telefone.

```
def lista_tel():

    pessoas = ['Ricardo Saldanha', 'Francisco Nobre', \
               'Leonor Martins', 'Hugo Dias', 'Luiz Leite', \
               'Ana Pacheco', 'Fausto Almeida']
    telefones = [211234567, 919876543, 937659862, 964876347, \
                 218769800, 914365986, 229866450]

    ordena_nomes(pessoas, telefones)

    quem = input('Qual o nome?\n(fim para terminar)\n? ')
    while quem != 'fim':
        pos_tel = procura(pessoas, quem)
        if pos_tel == -1:
            print('Telefone desconhecido')
        else:
            print('O telefone é:', telefones[pos_tel])
        quem = input('Qual o nome?\n(fim para terminar)\n? ')

def ordena_nomes(pessoas, telefs):
    for i in range(len(pessoas)):
        pos_menor = i
        for j in range(i + 1, len(pessoas)):
            if pessoas[j] < pessoas[pos_menor]:
                pos_menor = j
            pessoas[i], pessoas[pos_menor] = \
                pessoas[pos_menor], pessoas[i]
            telefs[i], telefs[pos_menor] = \
                telefs[pos_menor], telefs[i]
```

5.7 Considerações sobre eficiência

Dissemos que a procura binária é mais eficiente do que a procura sequencial, e que a ordenação Shell é mais eficiente do que a ordenação por borbulhamento. Nesta secção, discutimos métodos para comparar a eficiência de algoritmos. O estudo da eficiência de um algoritmo é um aspeto importante em informática,

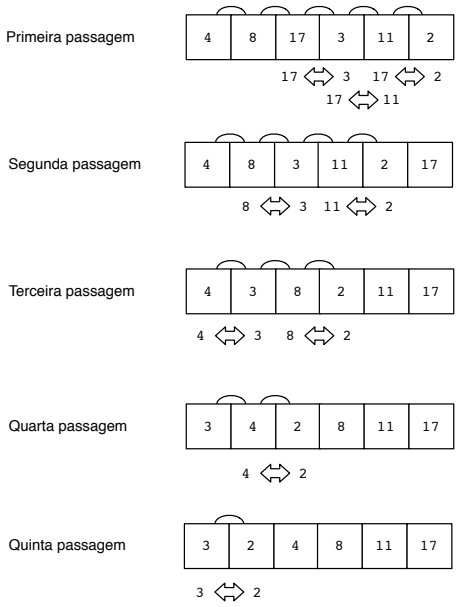

Figura 5.9: Padrão de ordenação utilizando a ordenação por borbulhamento.

porque fornece uma medida grosseira do trabalho envolvido na execução de um algoritmo.

Um dos processos para determinar o trabalho envolvido na execução de um algoritmo consiste em considerar um conjunto de dados e seguir a execução do algoritmo para esses dados. Este aspeto está ilustrado nas figuras 5.9 a 5.11, em que mostramos a evolução da posição relativa dos elementos da lista `[4, 8, 17, 3, 11, 2]`, utilizando os três métodos de ordenação que apresentámos. Nestas figuras, cada linha corresponde a uma passagem pela lista, um arco ligando duas posições da lista significa que os elementos da lista nessas posições foram comparados e uma seta por baixo de duas posições da lista significa que os elementos nessas posições (os quais estão indicados nas extremidades da seta) foram trocados durante uma passagem.

Esta abordagem, contudo, tem a desvantagem do trabalho envolvido na execução de um algoritmo poder variar drasticamente com os dados que lhe são fornecidos (por exemplo, se os dados estiverem ordenados, a ordenação por borbulhamento só necessita de uma passagem). Por esta razão, é importante

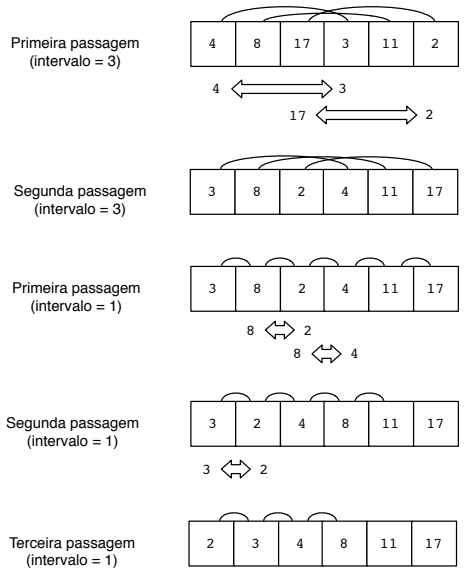

Figura 5.10: Padrão de ordenação utilizando a ordenação *Shell*.

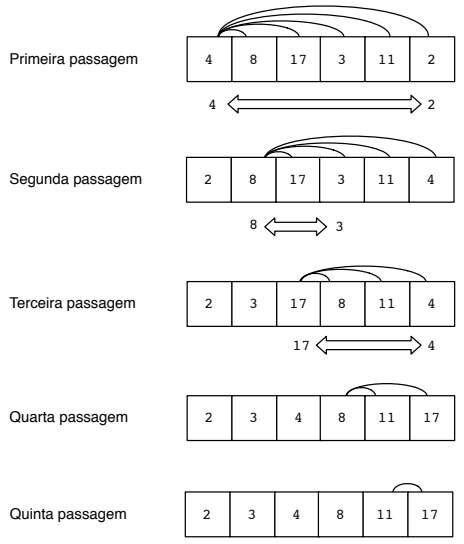

Figura 5.11: Padrão de ordenação utilizando a ordenação por seleção.

encontrar uma medida do trabalho envolvido na execução de um algoritmo que seja independente dos valores particulares dos dados que são utilizados na sua execução.

Este problema é resolvido com outro modo de descrever o trabalho envolvido na execução de um algoritmo, que recorre à noção de *ordem de crescimento*[9] para obter uma avaliação grosseira dos recursos exigidos pelo algoritmo à medida que a quantidade de dados manipulados aumenta. A ideia subjacente a esta abordagem é isolar uma operação que seja fundamental para o algoritmo, e contar o número de vezes que esta operação é executada. Para o caso dos algoritmos de procura e de ordenação, esta operação é a comparação (dados dois elementos, qual deles é o maior?). Segundo este processo, a medida de eficiência de um algoritmo de procura ou de ordenação é o número de comparações que estes efetuam. Dados dois algoritmos, diremos que o algoritmo que efetuar o menor número de comparações é o mais eficiente.

Na procura sequencial, percorremos a sequência de elementos até encontrar o elemento desejado. Se o elemento desejado se encontrar na primeira posição da lista, necessitamos apenas de uma comparação mas poderemos ter de percorrer toda a lista, se o elemento procurado não se encontrar na lista. O número médio de comparações cai entre estes dois extremos, pelo que o número médio de comparações na procura sequencial é:

$$\frac{n}{2}$$

em que n é o número de elementos na lista.

Na procura binária somos capazes de reduzir para metade o número de elementos a considerar sempre que efetuamos uma comparação (ver a função apresentada na página 146). Assim, se começarmos com n elementos, o número de elementos depois de uma passagem é $n/2$; o número de elementos depois de duas passagens é $n/4$ e assim sucessivamente. No caso geral, o número de elementos depois de i passagens é $n/2^i$. O algoritmo termina quando o número de elementos é menor do que 1, ou seja, terminamos depois de i passagens quando

$$\frac{n}{2^i} < 1$$

[9] Do inglês, "order of growth".

ou

$$n < 2^i$$

ou

$$log_2(n) < i.$$

Deste modo, para uma lista com n elementos, a procura binária não exige mais do que $log_2(n)$ passagens. Em cada passagem efetuamos três comparações (uma comparação na instrução `if` e duas comparações para calcular a expressão que controla o ciclo `while`). Consequentemente, o número de comparações exigida pela procura binária é inferior ou igual a $3 \cdot log_2(n)$.

No algoritmo de ordenação por borbulhamento, trocamos pares adjacentes de valores. O processo de ordenação consiste num certo número de passagens por todos os elementos da lista. O algoritmo da página 149 utiliza duas estruturas de repetição encadeadas. A estrutura exterior é realizada através de uma instrução `while`, e a interior com uma instrução `for`. Sendo n o número de elementos da lista, cada vez que o ciclo interior é executado, efetuam-se $n - 1$ comparações. O ciclo exterior é executado até que todos os elementos estejam ordenados. Na situação mais favorável, é executado apenas uma vez; na situação mais desfavorável é executado n vezes, neste caso, o número de comparações será $n \cdot (n-1)$. O caso médio cai entre estes dois extremos, pelo que o número médio de comparações necessárias para a ordenação por borbulhamento será

$$\frac{n \cdot (n-1)}{2} = \frac{1}{2} \cdot (n^2 - n).$$

No algoritmo de ordenação por seleção (apresentado na página 151), a operação básica é a seleção do menor elemento de uma lista. O algoritmo de ordenação por seleção é realizado através da utilização de dois ciclos `for` encadeados. Para uma lista com n elementos, o ciclo exterior é executado $n - 1$ vezes; o número de vezes que o ciclo interior é executado depende do valor da variável `i` do ciclo exterior. A primeira vez será executado $n-1$ vezes, a segunda vez $n-2$ vezes e a última vez será executado apenas uma vez. Assim, o número de comparações exigidas pela ordenação por seleção é

$$(n-1) + (n-2) + ... + 2 + 1$$

Utilizando a fórmula para calcular a soma de uma progressão aritmética, ob-

temos o seguinte resultado para o número de comparações na ordenação por seleção:

$$(n-1) \cdot \frac{(n-1)+1}{2} = \frac{1}{2} \cdot (n^2 - n).$$

5.7.1 A notação do Omaiúsculo

A nossa preocupação com a eficiência está relacionada com problemas que envolvem um número muito grande de elementos. Se estivermos a procurar numa lista com cinco elementos, mesmo o algoritmo menos eficiente nos resolve rapidamente o problema. Contudo, à medida que o número de elementos considerado cresce, o esforço necessário começa a diferir consideravelmente de algoritmo para algoritmo. Por exemplo, se a lista em que efetuamos a comparação tiver 100 000 000 de elementos, a procura sequencial requer uma média de 50 000 000 de comparações, ao passo que a procura binária requer, no máximo, 61 comparações!

Podemos exprimir uma aproximação da relação entre a quantidade de trabalho necessário e o número de elementos considerados, utilizando uma notação matemática conhecida por ordem de magnitude ou *notação do Omaiúsculo* (lido, ó maiúsculo[10]).

A notação do *O*maiúsculo é usada em matemática para descrever o comportamento de uma função, normalmente em termos de outra função mais simples, quando o seu argumento tende para o infinito. A notação do *O*maiúsculo pertence a uma classe de notações conhecidas por notações de Bachmann-Landau[11] ou notações assintóticas. A notação do *O*maiúsculo carateriza funções de acordo com a sua taxa de crescimento, funções diferentes com a mesma taxa de crescimento podem ser representadas pela mesma notação do *O*maiúsculo. Existem outras notações associadas à notação do *O*maiúsculo, usando os símbolos o, Ω, ω e Θ para descrever outros tipos de taxas de crescimento.

Sejam $f(x)$ e $g(x)$ duas funções com o mesmo domínio definido sobre o conjunto dos números reais. Escrevemos $f(x) = O(g(x))$ se e só se existir uma constante positiva k, tal que para valores suficientemente grandes de x, $|f(x)| \leq k \cdot |g(x)|$, ou seja, se e só se existe um número real positivo k e um real x_0 tal que $|f(x)| \leq k \cdot |g(x)|$ para todo o $x > x_0$.

[10] Do inglês, "Big-O".

[11] Em honra aos matemáticos Edmund Landau (1877–1938) e Paul Bachmann (1837–1920).

A ordem de magnitude de uma função é igual à ordem do seu termo que cresce mais rapidamente em relação ao argumento da função. Por exemplo, a ordem de magnitude de $f(n) = n+n^2$ é n^2 uma vez que, para grandes valores de n, o valor de n^2 domina o valor de n (é tão importante face ao valor de n, que no cálculo do valor da função podemos praticamente desprezar este termo). Utilizando a notação do Omaiúsculo, a ordem de magnitude de $n^2 + n$ é $O(n^2)$.

Tendo em atenção esta discussão, podemos dizer que a procura binária é de ordem $O(log_2(n))$, a procura sequencial é de ordem $O(n)$, e tanto a ordenação por borbulhamento como a ordenação por seleção são de ordem $O(n^2)$.

Os recursos consumidos por uma função não dependem apenas do algoritmo utilizado mas também do grau de dificuldade ou dimensão do problema a ser resolvido. Por exemplo, vimos que a procura binária exige menos recursos do que a procura sequencial. No entanto, estes recursos dependem da dimensão do problema em causa (do número de elementos da lista): certamente que a procura de um elemento numa lista de 5 elementos recorrendo à procura sequencial consome menos recursos do que a procura numa lista de 100 000 000 elementos utilizando a procura binária. O *grau de dificuldade* de um problema é tipicamente dado por um número que pode estar relacionado com o valor de um dos argumentos do problema (nos casos da procura e da ordenação, o número de elementos da lista) ou o grau de precisão exigido (como no caso do erro admitido na função para o cálculo da raiz quadrada apresentada na Secção 3.4.5), etc.

De modo a obter uma noção das variações profundas relacionadas com as ordens de algumas funções, suponhamos que dispomos de um computador capaz de realizar 3 mil milhões de operações por segundo (um computador com um processador de 3GHz). Na Tabela 5.2 apresentamos a duração aproximada de alguns algoritmos com ordens de crescimento diferentes, para alguns valores do grau de dificuldade do problema. Notemos que para um problema com grau de dificuldade 19, se o seu crescimento for de ordem $O(6^n)$, para que a função termine nos nossos dias, esta teria que ter sido iniciada no período em que surgiu o *Homo sapiens* e para uma função com crescimento de ordem $O(n!)$ esta teria que ter sido iniciada no tempo dos dinossauros para estar concluída na atualidade. Por outro lado, ainda com base na Tabela 5.2, para um problema com grau de dificuldade 21, a resolução de um problema com ordem $O(log_2(n))$ demoraria na ordem das 4.6 centésimas de segundo ao passo que a resolução de um problema com ordem $O(n!)$ requereria mais tempo do que a idade do

n	$log_2(n)$	n^3	6^n	$n!$
19	0.044 segundos	72.101 segundos	2.031×10^5 anos (Homo sapiens)	4.055×10^7 anos (tempo dos dinossauros)
20	0.045 segundos	84.096 segundos	1.219×10^6 anos	8.110×10^8 anos (início da vida na terra)
21	0.046 segundos	97.351 segundos	7.312×10^6 anos	1.703×10^{10} anos (superior à idade do universo)
22	0.047 segundos	1.866 minutos	4.387×10^7 anos (tempo dos dinossauros)	
23	0.048 segundos	2.132 minutos	2.632×10^8 anos	
24	0.048 segundos	2.422 minutos	1.579×10^9 anos	
25	0.049 segundos	2.738 minutos	9.477×10^9 anos (superior à idade da terra)	
26	0.049 segundos	3.079 minutos	5.686×10^{10} anos (superior à idade do universo)	
100	0.069 segundos	2.920 horas		
500	0.094 segundos	15.208 dias		
1000	0.105 segundos	121.667 dias		

Tabela 5.2: Duração comparativa de algoritmos.

universo.

5.8 Notas finais

Neste capítulo apresentámos o tipo lista, o qual é muito comum em programação, sendo tipicamente conhecido por tipo vetor ou tipo tabela. A lista é um tipo mutável, o que significa que podemos alterar destrutivamente os seus elementos. Apresentámos alguns algoritmos de procura e de ordenação aplicados a listas. Outros métodos de procura e de ordenação podem ser consultados em [Knuth, 1973b] e [Cormen et al., 2009].

As listas existentes em Python diferem dos tipos vetor e tabela existentes em outras linguagens de programação, normalmente conhecidos pela palavra inglesa "array", pelo facto de permitirem remover elementos existentes e aumentar o número de elementos existentes. De facto, em outras linguagens de programação, o tipo "array" apresenta uma caraterística estática: uma vez criado um elemento do tipo "array", o número dos seus elementos é fixo. Os elementos podem ser mudados, mas não podem ser removidos e novos elementos não podem ser introduzidos. O tipo lista em Python mistura as caraterísticas do tipo "array" de outras linguagens de programação com as caraterística de outro tipo de informação correspondente às *listas ligadas*[12], as quais são apresentadas no Capítulo 14.2.

Apresentámos uma primeira abordagem à análise da eficiência de um algoritmo através do estudo da ordem de crescimento e introdução da notação do Omaiúsculo. Para um estudo mais aprofundado da análise da eficiência de algoritmos, recomendamos a leitura de [Cormen et al., 2009], [Edmonds, 2008] ou [McConnell, 2008].

5.9 Exercícios

1. Suponha que a operação `in` não existia em Python. Escreva uma função em Python, com o nome `pertence`, que recebe como argumentos uma lista e um inteiro e devolve `True`, se o inteiro está armazenado na lista, e `False`, em caso contrário. Não pode usar um ciclo `for` pois este recorre à operação `in`. Por exemplo,

   ```
   >>> pertence(3, [2, 3, 4])
   True
   >>> pertence(1, [2, 3, 4])
   False
   ```

2. Escreva uma função chamada `substitui` que recebe uma lista, `lst`, dois valores, `velho` e `novo`, e devolve a lista que resulta de substituir em `lst` todas as ocorrências de `velho` por `novo`. Por exemplo,

   ```
   >>> substitui([1, 2, 3, 2, 4], 2, 'a')
   [1, 'a', 3, 'a', 4]
   ```

[12] Do inglês, "linked list".

3. Escreva uma função chamada `posicoes_lista` que recebe uma lista e um elemento, e devolve uma lista contendo todas as posições em que o elemento ocorre na lista. Por exemplo,

```
>>> posicoes_lista(['a', 2, 'b', 'a'], 'a')
[0, 3]
```

4. Escreva uma função chamada `parte` que recebe como argumentos uma lista, `lst`, e um elemento, `e`, e que devolve uma lista de dois elementos, contendo na primeira posição a lista com os elementos de `lst` menores que `e`, e na segunda posição a lista com os elementos de `lst` maiores ou iguais a `e`. Por exemplo,

```
>>> parte([2, 0, 12, 19, 5], 6)
[[2, 0, 5], [12, 19]]
>>> parte([7, 3, 4, 12], 3)
[[], [7, 3, 4, 12]]
```

5. Escreva uma função chamada `inverte` que recebe uma lista contendo inteiros e devolve a lista com os mesmos elementos mas por ordem inversa.

6. Uma *matriz* é uma tabela bidimensional em que os seus elementos são referenciados pela linha e pela coluna em que se encontram. Uma matriz pode ser representada como uma lista cujos elementos são listas. Com base nesta representação, escreva uma função, chamada `elemento_matriz` que recebe como argumentos uma matriz, uma linha e uma coluna e que devolve o elemento da matriz que se encontra na linha e coluna indicadas. A sua função deve permitir a seguinte interação:

```
>>> m = [[1, 2, 3], [4, 5, 6]]
>>> elemento_matriz(m, 0, 0)
1
>>> elemento_matriz(m, 0, 3)
Índice inválido: coluna 3
>>> elemento_matriz(m, 4, 1)
Índice inválido: linha 4
```

7. Considere uma matriz como definida no exercício anterior. Escreva uma

função em Python que recebe uma matriz e que a escreve sob a forma

$$\begin{array}{llll} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}$$

8. Considere, de novo, o conceito de matriz. Escreva uma função em Python que recebe como argumentos duas matrizes e devolve uma matriz correspondente ao produto das matrizes que são seus argumentos. Os elementos da matriz produto são dados por

$$p_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

9. As funções de ordenação apresentadas neste capítulo correspondem a uma ordenação destrutiva, pois a lista original é destruída e substituída pela lista ordenada. Um processo alternativo consiste em criar uma lista com índices, que representam as posições ordenadas dos elementos da lista. Escreva uma função em Python para efetuar a ordenação por seleção, criando uma lista com índices. A sua função deve permitir a interação:

```
>>> original = [5, 2, 9, 1, 4]
>>> ord = ordena(original)
>>> ord
[3, 1, 4, 0, 2]
>>> original
[5, 2, 9, 1, 4]
>>> for i in range(len(original)):
...     print(original[ord[i]])
...
1
2
4
5
9
```