Capítulo 7

Recursão e iteração

'Well, I'll eat it,' said Alice, 'and if it makes me grow larger, I can reach the key; and if it makes me grow smaller, I can creep under the door: so either way I'll get into the garden, and I don't care which happens!'
She ate a little bit, and said anxiously to herself, 'Which way? Which way?'

Lewis Carroll, *Alice's Adventures in Wonderland*

Nos capítulos anteriores considerámos vários aspetos da programação, estudámos o modo de criar funções, utilizámos alguns tipos de dados existentes em Python e estudámos o modo de definir novos tipos de dados. Embora sejamos já capazes de escrever programas com alguma complexidade, o conhecimento que adquirimos ainda não é suficiente para podermos programar de um modo eficiente. Falta-nos saber que funções vale a pena definir e quais as consequências da execução de uma função.

Recordemos que a entidade subjacente à computação é o processo computacional. Na atividade de programação planeamos a sequência de ações a serem executadas por um programa. Para podermos desenvolver programas adequados a um dado fim é essencial que tenhamos uma compreensão clara dos tipos de processos computacionais gerados pelos diferentes tipos de funções. Este aspeto é abordado neste capítulo.

Uma função pode ser considerada como a especificação da evolução local de um processo computacional. Por *evolução local* entenda-se que a função define, em

cada instante, o comportamento do processo computacional, ou seja, especifica como construir cada estágio do processo a partir do estágio anterior. Ao abordar processos computacionais, queremos estudar a evolução global do processo cuja evolução local é definida por uma função[1]. Neste capítulo, apresentamos alguns padrões típicos da evolução de processos computacionais, estudando a ordem de grandeza do número de operações associadas e o "espaço" exigido pela evolução global do processo.

7.1 Recursão linear

Começamos por considerar funções que geram processos que apresentam um padrão de evolução a que se chama recursão linear.

Consideremos a função `fatorial` apresentada na Secção 6.2:

```
def fatorial(n):
    if n == 0:
        return 1
    else:
        return n * fatorial(n - 1)
```

Vimos que durante a avaliação de uma expressão cujo operador é `fatorial`, se gera uma sequência de ambientes locais, todos referentes a esta função (representados na Figura 7.1 para o cálculo de `fatorial(3)`), cada um dos quais contém uma operação adiada: a execução da operação de multiplicação vai sendo sucessivamente adiada até se atingir o valor que corresponde à parte básica.

Este encadeamento de ambientes começa por gerar uma expansão da memória ocupada durante a execução da função, expansão essa que vai ocorrendo até que se encontra o caso terminal da função, sendo seguida por uma fase de contração à medida que os ambientes vão sendo libertados (ver os diagramas de ambientes apresentados nas páginas 182 a 184).

O processo gerado pela função `fatorial` pode ser representado pelo seguinte encadeamento de chamadas à função `fatorial`, em que cada chamada recursiva é escrita mais para a direita:

[1] Note-se que, na realidade, a função também define a evolução global do processo.

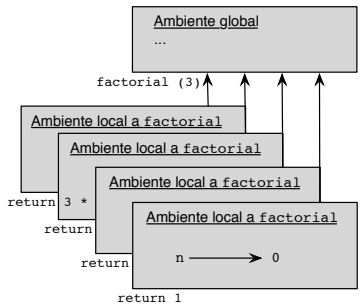

Figura 7.1: Ambientes criados pela avaliação de `fatorial(3)`.

```
fatorial(3)
| fatorial(2)
| | fatorial(1)
| | | fatorial(0)
| | | return 1
| | return 1
| return 2
return 6
```

Consideremos a função `potencia` apresentada na Secção 6.3:

```
def potencia(x, n):
    if n == 0:
        return 1
    else:
        return x * potencia(x, n - 1)
```

Podemos observar que durante a avaliação de uma expressão cujo operador é `potencia`, também se gera uma sequência de ambientes, todos referentes à função `potencia`. Nestes ambientes, a execução da operação de multiplicação vai sendo sucessivamente adiada até se atingir o valor que corresponde à parte básica da função, após o que se efetuam as operações que foram sendo adiadas. Por exemplo, para o cálculo de `potencia(2, 5)`, este processo pode ser representado do seguinte modo:

```
potencia(2, 5)
| potencia(2, 4)
| | potencia(2, 3)
| | | potencia(2, 2)
| | | | potencia(2, 1)
| | | | | potencia(2, 0)
| | | | | return 1
| | | | return 2
| | | return 4
| | return 8
| return 16
return 32
```

Como terceiro exemplo, consideremos a função, `soma_elementos`, para calcular a soma dos elementos de uma lista[2]. Podemos escrever esta função do seguinte modo:

```
def soma_elementos(lst):
    if lst == []:
        return 0
    else:
        return lst[0] + soma_elementos(lst[1:])
```

Ou seja, se a lista não tem elementos, a soma dos seus elementos é zero, em caso contrário, a soma dos seus elementos corresponde ao resultado de somar o primeiro elemento da lista ao resultado de calcular a soma dos restantes elementos da lista.

Podemos considerar a forma do processo gerado pela avaliação de `soma_elementos([1, 2, 3, 4])`:

```
soma_elementos([1, 2, 3, 4])
| soma_elementos([2, 3, 4])
| | soma_elementos([3, 4])
| | | soma_elementos([4])
```

[2]Consideramos que a lista contém números, pelo que não faremos qualquer teste ao seu conteúdo.

```
| | | | soma_elementos([])
| | | | return 0
| | | return 4
| | return 7
| return 9
return 10
```

Voltamos a deparar-nos com um processo em que se gera uma sequência de ambientes, todos referentes à função `soma_elementos`. Nestes ambientes, a execução da operação de adição vai sendo sucessivamente adiada até se atingir o valor que corresponde à parte básica da função `soma_elementos`, após o que se efetuam as operações que foram sendo adiadas.

Todas as funções que apresentámos nesta secção geram processos computacionais que têm um comportamento semelhante: são caraterizados pela criação de uma sequência de ambientes correspondentes à mesma função, o que origina uma expansão da memória necessária para a execução do processo (*fase de crescimento*), seguida por uma diminuição da memória necessária (*fase de contração*). Este padrão de evolução de um processo é muito comum em programação e tem o nome de processo recursivo. Num *processo recursivo* é criada uma sequência de ambientes correspondentes à mesma função.

Tipicamente, mas não obrigatoriamente, num processo recursivo durante a fase de expansão geram-se operações adiadas e na fase de contração essas operações são executadas.

Em todos os casos que apresentámos, o número ambientes gerados cresce linearmente com um determinado valor. Este valor pode corresponder a um dos parâmetros da função (como é o caso do inteiro para o qual se está a calcular o fatorial, do expoente relativamente à potência e do número de elementos da lista), mas pode também corresponder a outras coisas. A um processo recursivo que cresce linearmente com um valor dá-se o nome de processo *recursivo linear*.

7.2 Iteração linear

Nesta secção consideramos os problemas discutidos na secção anterior e apresentamos a sua solução através de funções que geram processos que apresentam um padrão de evolução a que se chama iteração linear.

Consideremos a função `fatorial` apresentada na página 181:

```
def fatorial(n):
    fat = 1
    for i in range(n, 0, -1):
        fat = fat * i
    return fat
```

Nesta função, em cada instante, possuímos toda a informação necessária para saber o que já fizemos (o produto acumulado, `fat`) e o que ainda nos falta fazer (o número de multiplicações a realizar, `i`). Isto significa que podemos interromper o processo computacional em qualquer instante e recomeçá-lo mais tarde, utilizando a informação disponível no instante em que este foi interrompido. Às variáveis que caraterizam o estado de um processo, podendo recomeçá-lo em qualquer instante dá-se o nome de *variáveis de estado*.

A função `potencia` pode ser escrita do seguinte modo:

```
def potencia(x, n):
    pot = 1
    for i in range(1, n + 1):
        pot = pot * x
    return pot
```

Podemos também escrever a seguinte versão da função `soma_elementos`:

```
def soma_elementos(lst):
    soma = 0
    for e in lst:
        soma = soma + e
    return soma
```

As três funções que apresentámos geram processos computacionais que têm um comportamento semelhante. Eles são caraterizados por um certo número de variáveis que fornecem uma descrição completa do estado da computação em cada instante. O padrão de evolução de um processo que descrevemos nesta secção é também muito comum em programação, e tem o nome de *processo iterativo*. Recorrendo ao Wikcionário[3], a palavra "iterativo" tem o seguinte

[3] `http://pt.wiktionary.org/wiki`.

significado "Diz-se do processo que se repete diversas vezes para se chegar a um resultado e a cada vez gera um resultado parcial que será usado na vez seguinte".

Um *processo iterativo* é caraterizado por um certo número de variáveis, chamadas *variáveis de estado*, juntamente com uma regra que especifica como atualizá-las. Estas variáveis fornecem uma descrição completa do estado da computação em cada momento.

Nos nossos exemplos, o número de operações efetuadas sobre as variáveis de estado cresce linearmente com uma grandeza associada à função (o inteiro para o qual se está a calcular o fatorial, o expoente no caso da potência e o o número de elementos da lista).

A um processo iterativo cujo número de operações cresce linearmente com um valor dá-se o nome de processo *iterativo linear*.

7.3 Recursão de cauda

O processo de cálculo utilizado na secção anterior para a escrita da função `fatorial` pode também ser traduzido pela função recursiva `fatorial_rec`, na qual `prod_ac` representa o produto acumulado dos números que já multiplicámos e `prox` representa o próximo número a ser multiplicado:

```
def fatorial_rec(prod_ac, prox):
    if prox == 0:
        return prod_ac
    else:
        return fatorial_rec(prod_ac * prox, prox - 1)
```

A função `fatorial_rec` contém uma chamada recursiva, sendo esta chamada, a *última operação* efetuada pela função. Este padrão de computação tem o nome de *recursão de cauda*[4]. Note-se a diferença em relação à função `fatorial` apresentada na página 212, na qual a última operação realizada pela função é uma multiplicação.

Com base na função `fatorial_rec`, podemos agora escrever a seguinte função para o cálculo de `fatorial`, a qual esconde do exterior a utilização das variáveis

[4]Do Inglês, "tail recursion".

`prod_ac` e `prox`:

```
def fatorial(n):

    def fatorial_rec(prod_ac, prox):
        if prox == 0:
            return prod_ac
        else:
            return fatorial_rec(prod_ac * prox, prox - 1)

    return fatorial_rec(1, n)
```

Durante a avaliação de uma expressão cujo operador é `fatorial`, gera-se uma sequência de ambientes correspondentes à função `fatorial_rec`, não existindo, neste caso, operações adiadas:

```
fatorial(5)
fatorial_rec(1, 5)
| fatorial_rec(5, 4)
| | fatorial_rec(20, 3)
| | | fatorial_rec(60, 2)
| | | | fatorial_rec(120, 1)
| | | | | fatorial_rec(120, 0)
| | | | | return 120
| | | | return 120
| | | return 120
| | return 120
| return 120
return 120
```

Aplicando o mesmo raciocínio ao cálculo de `potencia`, podemos escrever a função `potencia_rec`, na qual o nome `n_mult` corresponde ao número de multiplicações que temos de executar, e o nome `prod_ac` corresponde ao produto acumulado. Ao iniciar o cálculo, estabelecemos que o produto acumulado é um e o número de multiplicações que nos falta efetuar corresponde ao expoente:

```
def potencia(x, n):

    def potencia_rec(x, n_mult, prod_ac):
        if n_mult == 0:
            return prod_ac
        else:
            return potencia_rec(x, n_mult - 1, x * prod_ac)

    return potencia_rec(x, n, 1)
```

Novamente, a função `potencia_rec` utiliza recursão de cauda e gera um processo em que existe uma fase de crescimento seguida por uma fase de contração.

Deixamos como exercício a escrita da função `soma_elementos` usando um raciocínio análogo.

As funções que geram processos iterativos e as funções que utilizam a recursão de cauda partilham a caraterística de utilizarem variáveis de estado. Por esta razão, os processadores de algumas linguagens de programação, por exemplo, o Scheme e o PROLOG, transformam automaticamente funções que utilizam recursão de cauda em funções que geram processos iterativos, sendo comum dizer que este tipo de funções gera processos iterativos.

7.4 Recursão em árvore

Nesta secção vamos considerar um outro padrão da evolução de processos que também é muito comum em programação, a recursão em árvore.

7.4.1 Os números de Fibonacci

Para ilustrar a recursão em árvore vamos considerar a sequência de números descoberta no século XIII pelo matemático italiano Leonardo Fibonacci (c. 1170–c. 1250), também conhecido por Leonardo de Pisa, ao tentar resolver o seguinte problema:

> "Quantos casais de coelhos podem ser produzidos a partir de um único casal durante um ano se cada casal originar um novo casal em cada mês,

o qual se torna fértil a partir do segundo mês; e não ocorrerem mortes."

Fibonacci chegou à conclusão que a evolução do número de casais de coelhos era ditada pela seguinte sequência: 0, 1, 1, 2, 3, 5, 8, 13, 21, ... Nesta sequência, conhecida por *sequência de Fibonacci*, cada termo, exceto os dois primeiros, é a soma dos dois anteriores. Os dois primeiros termos são respetivamente 0 e 1. Os números da sequência de Fibonacci são conhecidos por *números de Fibonacci*, e podem ser descritos através da seguinte definição:

$$fib(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ fib(n-1) + fib(n-2) & \text{se } n > 1 \end{cases}$$

Suponhamos que desejávamos escrever uma função em Python para calcular os números de Fibonacci. Com base na definição anterior podemos escrever a seguinte função:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Vejamos agora qual a forma do processo que é gerado para o cálculo de um número de Fibonacci, por exemplo, `fib(5)`. Assumindo que as sub-expressões numa expressão composta são avaliadas da esquerda para a direita[5], obtemos a seguinte evolução:

```
fib(5)
| fib(4)
| | fib(3)
| | | fib(2)
| | | | fib(1)
| | | | return 1
| | | | fib(0)
```

[5]No caso de uma ordem diferente de avaliação, a "forma" do processo é semelhante.

```
| | | | return 0
| | | return 1
| | | fib(1)
| | | return 1
| | return 2
| | fib(2)
| | | fib(1)
| | | return 1
| | | fib(0)
| | | return 0
| | return 1
| return 3
| fib(3)
| | fib(2)
| | | fib(1)
| | | return 1
| | | fib(0)
| | | return 0
| | return 1
| | fib(1)
| | return 1
| return 2
return 5
```

Ao analisarmos a forma do processo anterior, verificamos que esta não corresponde a nenhum dos padrões já estudados. No entanto, este apresenta um comportamento que se assemelha ao processo recursivo. Tem fases de crescimento em que são criados novos ambientes correspondentes à função `fib`, seguidas por fases de contração em que alguns destes ambientes desaparecem.

Ao contrário do que acontece com o processo recursivo linear, estamos perante a existência de múltiplas fases de crescimento e de contração que são originadas pela dupla recursão que existe na função `fib` (esta refere-se duas vezes a si própria). A este tipo de evolução de um processo dá-se o nome de *recursão em árvore*. Esta designação deve-se ao facto da evolução do processo ter a forma de uma árvore. Cada avaliação da expressão composta cujo operador é a função `fib` dá origem a duas avaliações (dois ramos de uma árvore), exceto para os

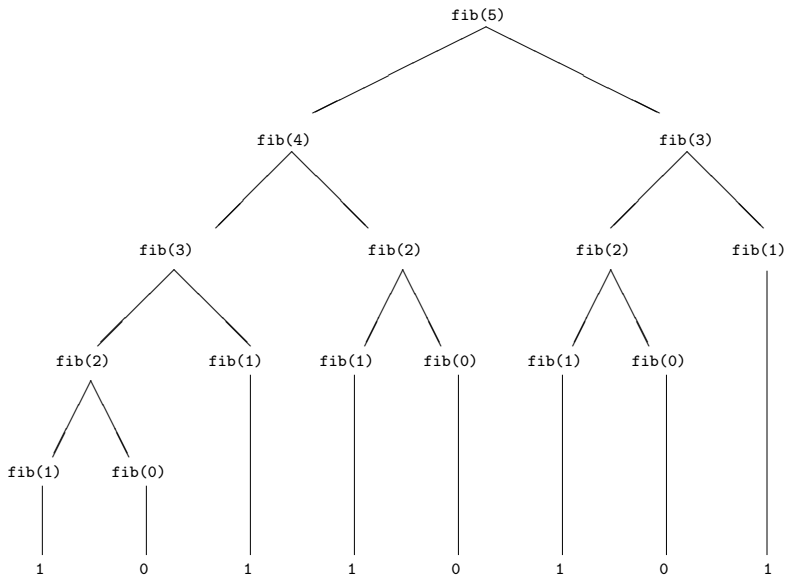

Figura 7.2: Árvore gerada durante o cálculo de `fib(5)`.

dois últimos valores. Na Figura 7.2 apresentamos o resultado da árvore gerada durante o cálculo de `fib(5)`.

Analisando o processo anterior, verificamos que este é muito ineficiente, pois existem cálculos que são repetidos múltiplas vezes. Para além disso, o processo utiliza um número de passos que não cresce linearmente com o valor de n. Demonstra-se que este número cresce exponencialmente com o valor de n.

Para evitar os cálculos repetidos, vamos usar uma função que utiliza a recursão de cauda. Como cada termo da sequência, exceto os dois primeiros, é a soma dos dois anteriores, para calcular um dos termos teremos, pois, de saber os dois termos anteriores. Suponhamos que, no cálculo de um termo genérico f_n, os dois termos anteriores são designados por f_{n-1} e f_{n-2}. Neste caso, o próximo número de Fibonacci será dado por $f_n = f_{n-1} + f_{n-2}$ e a partir de agora os dois últimos termos são f_n e f_{n-1}. Podemos agora, usando o mesmo raciocínio, calcular o próximo termo, f_{n+1}. Para completarmos o nosso processo teremos de explicitar o número de vezes que temos de efetuar estas operações.

Assim, podemos escrever a seguinte função que traduz o nosso processo de raciocínio. Esta função recebe os dois últimos termos da sequência (`f_n_1` e `f_n_2`) e o número de operações que ainda temos de efetuar (`cont`).

```
def fib_aux(f_n_1, f_n_2, cont):
    if cont == 0:
        return f_n_2 + f_n_1
    else:
        return fib_aux(f_n_2 + f_n_1,  f_n_1, cont - 1)
```

Devemos ainda indicar os dois primeiros termos da sequência de Fibonacci:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_aux (1, 0, n - 2)
```

Esta função origina um processo recursivo linear em que não há duplicações de cálculos:

```
fib(5)
fib_aux(1, 0, 3)
| fib_aux(1, 1, 2)
| | fib_aux(2, 1, 1)
| | | fib_aux(3, 2, 0)
| | | return 5
| | return 5
| return 5
return 5
return 5
```

Utilizando um processo iterativo, a função `fib` poderá ser definida do seguinte modo:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        f_n_2 = 0
        f_n_1 = 1
        for i in range(n):
            f_n_2, f_n_1 = f_n_1, f_n_2 + f_n_1
        return f_n_2
```

Deste exemplo não devemos concluir que a recursão em árvore é um processo inútil. Na próxima secção apresentamos um outro problema para o qual a recursão em árvore corresponde ao método ideal para a sua resolução.

7.4.2 A torre de Hanói

Apresentamos um programa em Python para a solução de um *puzzle* chamado a Torre de Hanói. A *Torre de Hanói* é constituída por 3 postes verticais, nos quais podem ser colocados discos de diâmetros diferentes, furados no centro, variando o número de discos de *puzzle* para *puzzle*. O *puzzle* inicia-se com todos os discos num dos postes (tipicamente, o poste da esquerda), com o disco menor no topo e com os discos ordenados, de cima para baixo, por ordem crescente dos respetivos diâmetros, e a finalidade é movimentar todos os discos para um outro poste (tipicamente, o poste da direita), também ordenados por ordem crescente dos respetivos diâmetros, de acordo com as seguintes regras: (1) apenas se pode movimentar um disco de cada vez; (2) em cada poste, apenas se pode movimentar o disco de cima; (3) nunca se pode colocar um disco sobre outro de diâmetro menor.

Este *puzzle* está associado a uma lenda, segundo a qual alguns monges num mosteiro perto de Hanói estão a tentar resolver um destes *puzzles* com 64 discos, e no dia em que o completarem será o fim do mundo. Não nos devemos preocupar com o fim do mundo, pois se os monges apenas efetuarem movimentos perfeitos à taxa de 1 movimento por segundo, demorarão perto de mil milhões de anos

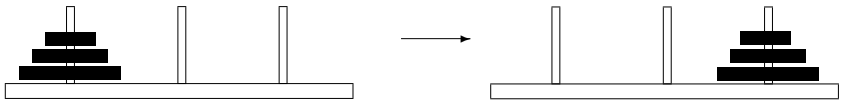

Figura 7.3: Torre de Hanói com três discos.

para o resolver[6].

Na Figura 7.3 apresentamos um exemplo das configurações inicial e final para a Torre de Hanói com três discos.

Suponhamos então que pretendíamos escrever um programa para resolver o *puzzle* da Torre de Hanói para um número n de discos (o valor de n será fornecido pelo utilizador). Para resolver o *puzzle* da Torre de Hanói com n discos ($n > 1$), teremos de efetuar basicamente três passos:

1. Movimentar $n - 1$ discos do poste da esquerda para o poste do centro (utilizado como poste auxiliar);
2. Movimentar o disco do poste da esquerda para o poste da direita;
3. Movimentar os $n - 1$ discos do poste do centro para o poste da direita.

Estes passos encontram-se representados na Figura 7.4 para o caso de $n = 3$.

Com este método conseguimos reduzir o problema de movimentar n discos ao problema de movimentar $n - 1$ discos, ou seja, o problema da movimentação de n discos foi descrito em termos do mesmo problema, mas tendo um disco a menos. Temos aqui um exemplo típico de uma solução recursiva. Quando n for igual a 1, o problema é resolvido trivialmente, movendo o disco da origem para o destino.

Como primeira aproximação, podemos escrever a função `mova` para movimentar n discos. Esta função tem como argumentos o número de discos a mover e uma indicação de quais os postes de origem e destino dos discos, bem como qual o poste que deve ser usado como poste auxiliar:

[6] Ver [Raphael, 1976], página 80.

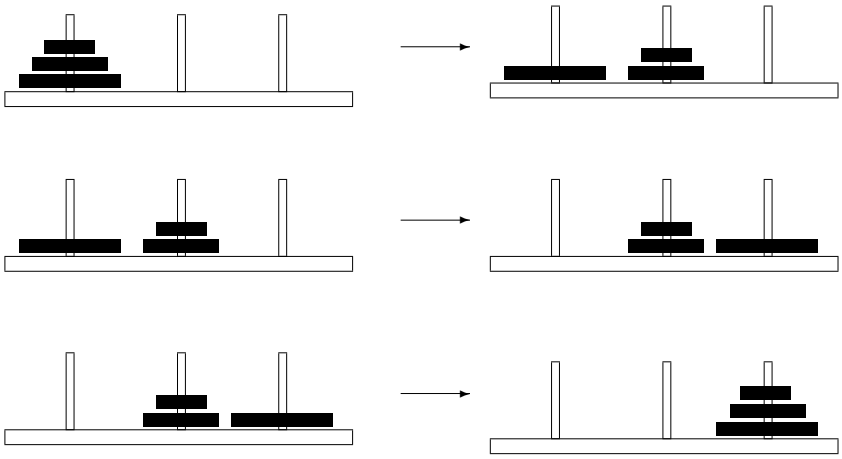

Figura 7.4: Solução da Torre de Hanói com três discos.

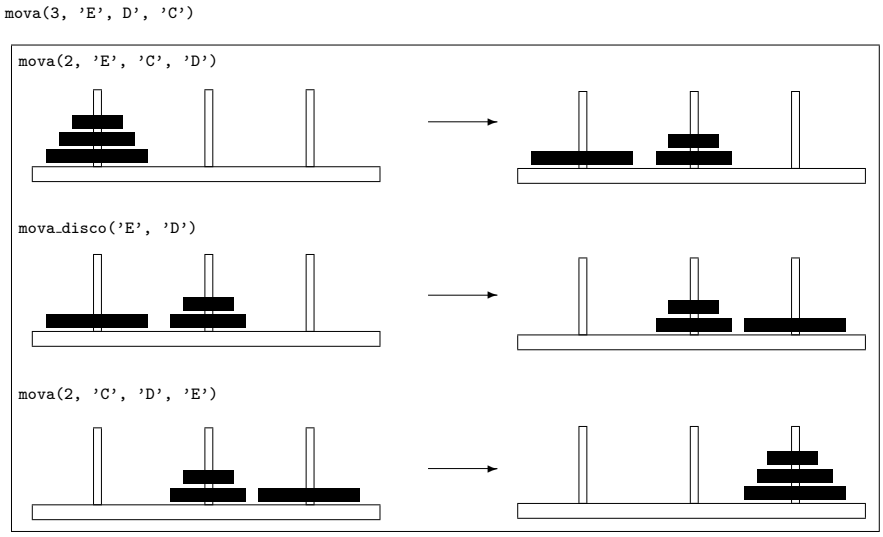

Figura 7.5: Subproblemas gerados por `mova(3, 'E', 'D', 'C')`.

```
def mova(n, origem, destino, aux):
    if n == 1:
        mova_disco(origem, destino)
    else:
        mova(n-1, origem, aux, destino)
        mova_disco(origem, destino)
        mova(n-1, aux, destino, origem)
```

Antes de continuar, convém observar que a instrução composta que corresponde ao `else` é constituída por três instruções. Dentro destas instruções, as duas utilizações da função `mova` têm os argumentos correspondentes aos postes, por ordem diferente, o que corresponde a resolver dois outros *puzzles* em que os postes de origem, de destino e auxiliares são diferentes. Na Figura 7.5 apresentamos as três expressões que são originadas por `mova(3, 'E', 'D', 'C')`, ou seja, mova três discos do poste da esquerda para o poste da direita, utilizando o poste do centro como poste auxiliar, bem como uma representação dos diferentes subproblemas que estas resolvem.

Esta função reflete o desenvolvimento do topo para a base: o primeiro passo

para a solução de um problema consiste na identificação dos subproblemas que o constituem, bem como a determinação da sua inter-relação. Escreve-se então uma primeira aproximação da solução em termos destes subproblemas. No nosso exemplo, o problema da movimentação de n discos foi descrito em termos de dois subproblemas: o problema da movimentação de um disco e o problema da movimentação de $n-1$ discos, e daí a solução recursiva.

Podemos agora escrever a seguinte função que fornece a solução para o *puzzle* da Torre de Hanói para um número arbitrário de discos:

```
def hanoi():

    def mova(n, origem, destino, aux):

        def mova_disco(de, para):
            print(de, '->', para)

        if n == 1:
            mova_disco(origem, destino)
        else:
            mova(n-1, origem, aux, destino)
            mova_disco(origem, destino)
            mova(n-1, aux, destino, origem)

    n = eval(input('Quantos discos deseja considerar?\n? '))
    print('Solução do puzzle:')
    mova(n, 'E', 'D', 'C')
```

Esta função permite originar a seguinte interação:

```
>>> hanoi()
Quantos discos deseja considerar?
? 3
Solução do puzzle:
E -> D
E -> C
D -> C
E -> D
```

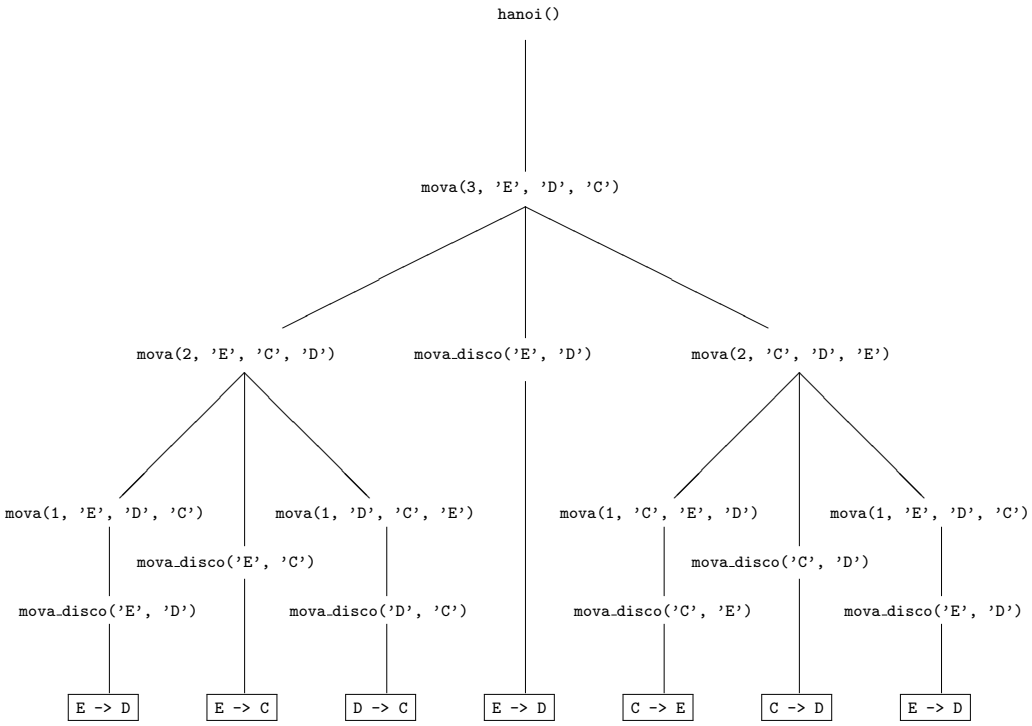

Figura 7.6: Árvore gerada por `hanoi()` com três discos

```
C -> E
C -> D
E -> D
```

Na Figura 7.6 apresentamos a árvore gerada pela execução da função `hanoi()` quando o utilizador fornece o valor 3 para o número de discos. Nesta figura apresentam-se dentro de um retângulo os valores que o Python escreve.

Estamos de novo perante uma função recursiva que origina um processo recursivo em árvore. Este exemplo mostra como a recursão pode dar origem a funções que são fáceis de escrever e de compreender.

7.5 Considerações sobre eficiência

Os exemplos apresentados neste capítulo mostram que os processos gerados por funções podem diferir drasticamente quanto à taxa a que consomem recursos computacionais, mesmo quando calculam a mesma função matemática. Assim, um dos aspetos que vamos ter de levar em linha de conta quando escrevemos programas é a minimização dos recursos computacionais consumidos. Os recursos computacionais que consideramos são o tempo e o espaço. O *tempo* diz respeito ao tempo que o nosso programa demora a executar, e o *espaço* diz respeito ao espaço de memória do computador usado pelo nosso programa.

Dos exemplos apresentados neste capítulo, podemos concluir que os processos recursivos lineares têm ordem de crescimento $O(n)$, quer para o espaço, quer para o tempo, e que os processos iterativos lineares têm ordem de crescimento $O(1)$ para o espaço e $O(n)$ para o tempo. Os processos recursivos em árvore têm ordem de crescimento $O(n)$ para o espaço e $O(k^n)$ para o tempo.

Tendo em atenção as preocupações sobre os recursos consumidos por um processo, apresentamos uma alternativa para o cálculo de potência, a qual gera um processo com ordem de crescimento inferior ao que apresentámos. Para compreender o novo método de cálculo de uma potência, repare-se que podemos definir potência, do seguinte modo:

$$x^n = \begin{cases} x & \text{se } n = 1 \\ x.(x^{n-1}) & \text{se } n \text{ for ímpar} \\ (x^{n/2})^2 & \text{se } n \text{ for par} \end{cases}$$

o que nos leva a escrever a seguinte função para o cálculo da potência:

```
def potencia_rapida(x, n):
    if n == 0:
        return 1
    elif impar(n):
        return x * potencia_rapida(x, n - 1)
    else:
        return quadrado(potencia_rapida(x, n // 2))
```

Nesta função, `quadrado` e `impar` correspondem às funções:

```
def quadrado(x):
    return x * x

def impar(x):
    return (x % 2) == 1
```

Com este processo de cálculo, para calcular x^{2n} precisamos apenas de mais uma multiplicação do que as que são necessárias para calcular x^n. Geramos assim um processo cuja ordem temporal e espacial é $O(log_2(n))$. Para apreciar a vantagem desta função, notemos que, para calcular uma potência de expoente 1 000, a função `potencia` necessita de 1 000 multiplicações, ao passo que a função `potencia_rapida` apenas necessita de 14.

Podemos definir a seguinte versão da função `potencia_rapida` que gera um processo iterativo[7]:

```
def potencia_rapida(x, n):
    res = 1
    while n != 0:
        if impar(n):
            res = res * x
            n = n - 1
        else:
            x = x * x
            n = n // 2
    return res
```

7.6 Notas finais

Neste capítulo apresentámos as motivações para estudar os processos gerados por funções e caraterizámos alguns destes processos, nomeadamente os processos recursivos lineares, iterativos lineares e recursivos em árvore.

[7] Agradeço à Prof. Maria dos Remédios Cravo a sugestão desta função.

7.7 Exercícios

1. Considere a seguinte função:

```
def m_p(x, y):

    def m_p_a(z):
        if z == 0:
            return 1
        else:
            return m_p_a(z - 1) * x

    return m_p_a(y)
```

   (a) Apresente a evolução do processo na avaliação de `m_p(2, 4)`.

   (b) Escreva uma função equivalente que utilize recursão de cauda.

2. Considere a função de Ackermann[8]:

$$A(m,n) = \begin{cases} n+1 & \text{se } m = 0 \\ A(m-1,1) & \text{se } m > 0 \text{ e } n = 0 \\ A(m-1, A(m,n-1)) & \text{se } m > 0 \text{ e } n > 0 \end{cases}$$

   (a) Escreva em Python uma função para calcular o valor da função de Ackermann.

   (b) Siga o processo gerado pelo cálculo de $A(2,2)$.

3. Suponha que em Python não existia o operador de multiplicação, `*`, mas que existiam os operadores de adição, `+`, e divisão inteira, `//`. O produto de dois números inteiros positivos pode ser definido através da seguinte fórmula:

$$x.y = \begin{cases} 0 & \text{se } x = 0 \\ \frac{x}{2} \cdot (y+y) & \text{se } x \text{ é par} \\ y + (x-1) \cdot y & \text{em caso contrário} \end{cases}$$

   (a) Defina a função `produto` que calcula o produto de dois números inteiros positivos, de acordo com esta definição.

   (b) Origina operações adiadas? Justifique.

[8] Tal como apresentada em [Machtey and Young, 1978], página 24.

(c) Se a sua função gerar operações adiadas, escreva uma função equivalente utilizando recursão de cauda; se a sua função utilizar recursão de cauda, escreva uma função equivalente que gera operações adiadas.

4. Considere a função g, definida para inteiros não negativos do seguinte modo:

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

Escreva uma função recursiva em Python para calcular o valor de $g(n)$.

5. O número de combinações de m objetos n a n pode ser dado pela seguinte fórmula:

$$C(m,n) = \begin{cases} 1 & \text{se } n = 0 \\ 1 & \text{se } m = n \\ 0 & \text{se } m < n \\ C(m-1,n) + C(m-1,n-1) & \text{se } m > n,\ m > 0 \text{ e } n > 0 \end{cases}$$

Escreva uma função em Python para calcular o número de combinações de m objetos n a n.