Recursão

Neste capítulo

- Você aprenderá recursão. A recursão é uma técnica de programação utilizada em muitos algoritmos. É um assunto importante para a compreensão dos capítulos seguintes.
- Você aprenderá como separar um problema em caso-base e caso recursivo. A estratégia dividir para conquistar (Capítulo 4) usa este conceito simples para resolver problemas complicados.

Estou animado com este capítulo porque trata de *recursão*, uma maneira elegante de solucionar problemas. A recursão é um dos meus tópicos favoritos, mas ela é polêmica. As pessoas ou a amam ou a odeiam, ou elas a odeiam até que aprendam a amá-la alguns anos

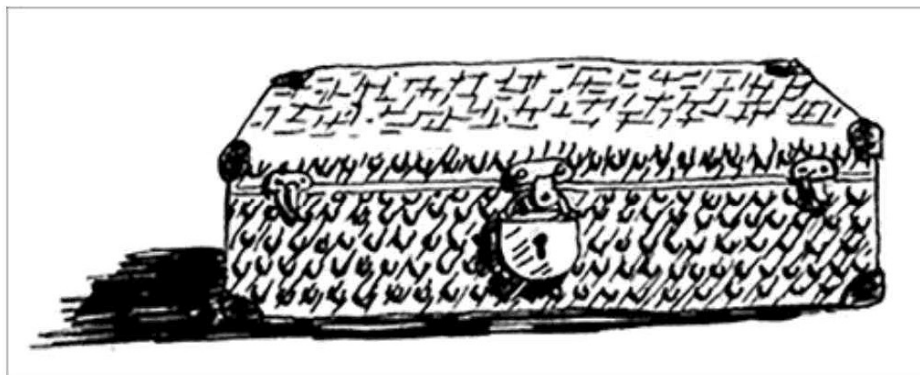
depois. Eu estava nessa terceira situação. Para facilitar as coisas, tenho um conselho:

- Este capítulo apresenta vários exemplos de códigos. Execute-os para ver como eles funcionam.
- Falarei sobre funções recursivas. Pelo menos uma vez, analise uma função recursiva com um papel e uma caneta, algo do tipo "vamos ver, passo o número 5 para a função `fatorial`, e então retorno cinco vezes passando o número 4 para `fatorial`, o que me dá…", e assim por diante. Analisar uma função dessa forma lhe ajudará a entender como funcionam as funções recursivas.

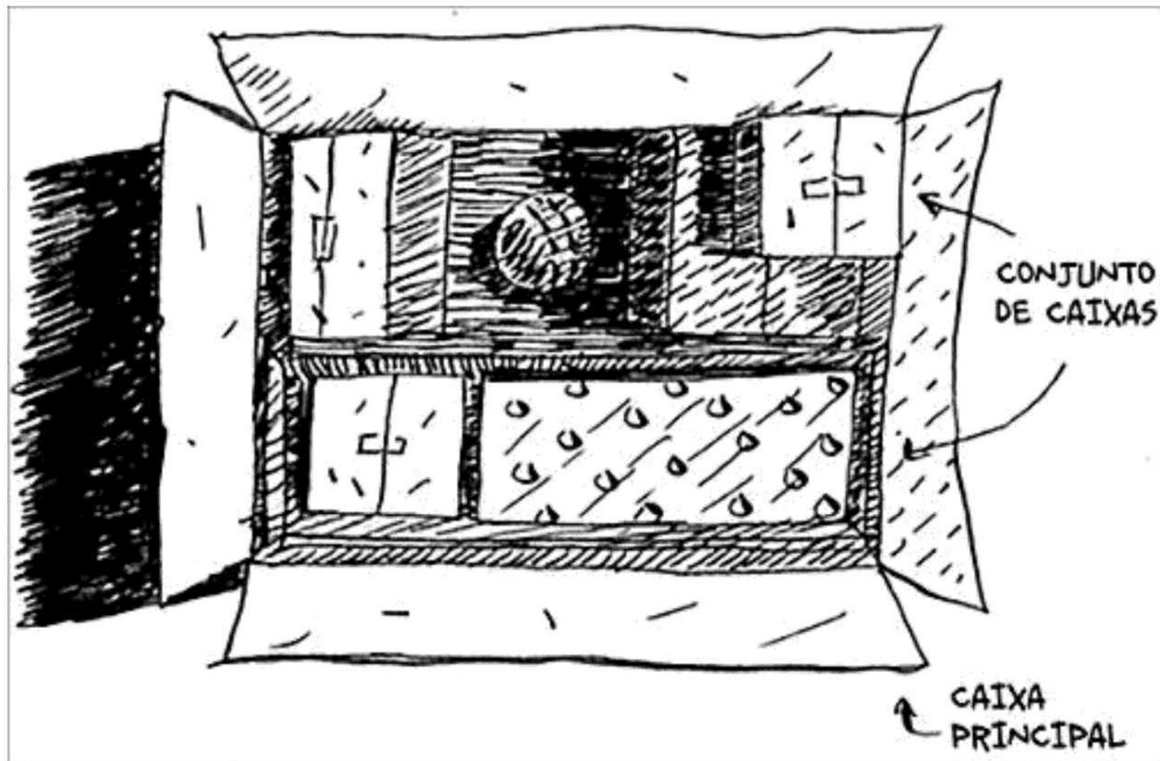
Este capítulo inclui muitos pseudocódigos. *Pseudocódigos* são uma descrição de alto nível de um problema em formato de código. É escrito como um código, mas utiliza linguagem mais próxima da humana.

Recursão

Suponha que você esteja vasculhando o porão de sua avó e encontre uma misteriosa mala trancada.



A sua avó diz que a chave para a mala provavelmente está em uma caixa.



Esta caixa contém mais caixas com mais caixas dentro delas. A chave está em alguma destas caixas. Qual é o seu algoritmo para procurá-la? Pense nisso antes de continuar a leitura.

Aqui está uma abordagem.

1. Monte uma pilha com as caixas que serão analisadas.

2. Pegue uma caixa e olhe o que tem dentro dela.

3. Se você encontrar outra caixa dentro dela, adicione-a a um novo monte para ser verificada mais tarde.

4. Se você encontrar uma chave, terminou!

5. Repita.

Aqui está outra abordagem.

1. Olhe o que tem dentro da caixa.

2. Se encontrar outra caixa, volte ao passo 1.

3. Se encontrar a chave, terminou!

Qual abordagem lhe parece mais fácil? A primeira abordagem utiliza um loop `while` (enquanto, em português). Enquanto o monte existir, pegue uma caixa e olhe o que tem dentro dela:

```
def procure_pela_chave(caixa_principal):
    pilha = main_box.crie_uma_pilha_para_busca()
    while pilha is not vazia:
        caixa = pilha.pegue_caixa()
        for item in caixa:
            if item.e_uma_caixa():
                pilha.append(item)
            elif item.e_uma_chave():
                print "achei a chave!"
```

A segunda maneira utiliza a recursão. *Recursão* é quando uma função chama a si mesma. Veja o pseudocódigo de como isso funciona.

```
def procure_pela_chave(caixa):
    for item in caixa:
        if item.e_uma_caixa():
            procure_pela_chave(item) ❶
        elif item.e_uma_chave():
            print "achei a chave!"
```

❶ Recursão!

Ambas as abordagens cumprem com a mesma proposta, mas a segunda me parece mais objetiva. A recursão é usada para tornar a resposta mais clara. Não há nenhum benefício quanto ao desempenho ao utilizar a recursão. Na verdade, os loops algumas vezes são melhor para o desempenho de um programa. Gosto desta frase de Leigh Caldwell, do Stack Overflow: "Os loops podem melhorar o desempenho do seu programa. A recursão melhora o desempenho do seu programador. Escolha o que for mais importante para a sua situação."[1]

Muitos algoritmos importantes usam a recursão, então é fundamental entender este conceito.

Caso-base e caso recursivo

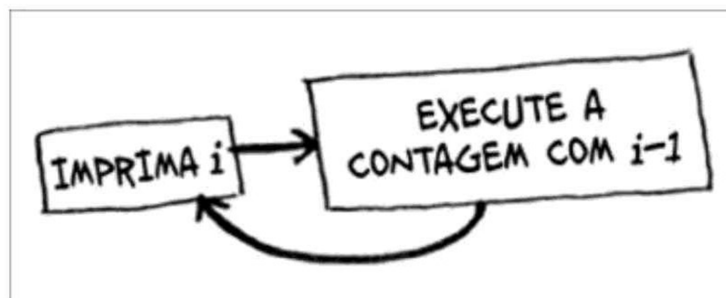
Devido ao fato de a função recursiva chamar a si mesma, é mais fácil escrevê-la erroneamente e acabar em um loop infinito. Por exemplo, suponha que você escreva uma função que imprima uma contagem regressiva, como esta:

```
> 3...2...1
```

Você pode escrever isso de maneira recursiva fazendo o seguinte:

```
def regressiva(i):
    print i
    regressiva(i-1)
```

Escreva este código e execute-o. Você perceberá um problema: essa função ficará executando para sempre!



Loop infinito.

```
> 3...2...1...0...-1...-2...
```

(Pressione Ctrl-C para interromper o seu script.)

Quando você escreve uma função recursiva, deve informar quando a

recursão deve parar. É por isso que *toda função recursiva tem duas partes: o caso-base e o caso recursivo*. O caso recursivo é quando a função chama a si mesma. O caso-base é quando a função não chama a si mesma novamente, de forma que o programa não se torna um loop infinito.

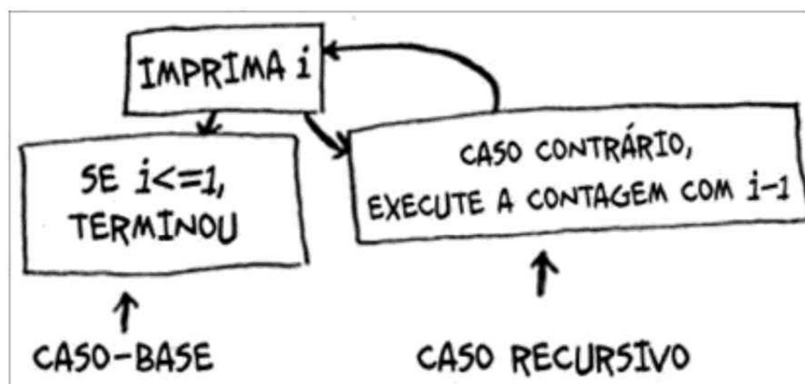
Vamos adicionar o caso-base à função de contagem regressiva:

```
def regressiva(i):
  print i
  if i <= 1: ❶
    return
  else: ❷
    regressiva(i-1)
```

❶ Caso-base.

❷ Caso recursivo.

Agora, a função funciona como esperado. Ela fica mais ou menos assim:



A pilha

Esta seção aborda a *pilha de chamada* (call stack). Isto é um conceito importante em programação e indispensável para entender a recursão.

Suponha que você esteja fazendo um churrasco para os seus amigos. Você tem uma lista de afazeres em forma de uma pilha de notas adesivas.

Você se lembra de que, quando falamos de arrays e listas, também havia uma lista de afazeres? Podia adicionar itens em qualquer lugar da lista ou remover itens aleatórios. A pilha de notas adesivas é bem mais simples. Quando você insere um item, ele é colocado no topo da pilha. Quando você lê um item, lê apenas o item do topo da pilha e ele é retirado da lista. Logo, sua lista de afazeres contém apenas duas ações: *push* (inserir) e *pop* (remover e ler).

Vamos ver como isso funciona na prática.

Esta estrutura de dados é chamada de *pilha*. A pilha é uma estrutura de dados simples. Você a tem usado esse tempo todo sem perceber!

A pilha de chamada

Seu computador usa uma pilha interna denominada *pilha de chamada*. Vamos ver isto na prática. Aqui está um exemplo simples:

```
def sauda(nome):
    print "Olá, " + nome + "!"
    sauda2(nome)
    print "preparando para dizer tchau..."
    tchau()
```

Esta função te cumprimenta e chama outras duas funções:

```
def sauda2(nome):
    print "Como vai " + nome + "?"

def tchau():
    print "ok, tchau!"
```

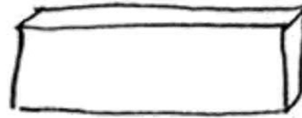
Vamos analisar o que acontece quando você chama uma função.

Nota

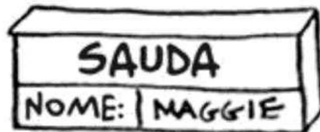
`print` é uma função em Python, mas, para facilitar as coisas, vamos fingir que não é. Entre na brincadeira.

N.R.T.: tecnicamente `print` não é uma função em Python 2, mas uma instrução ou statement.

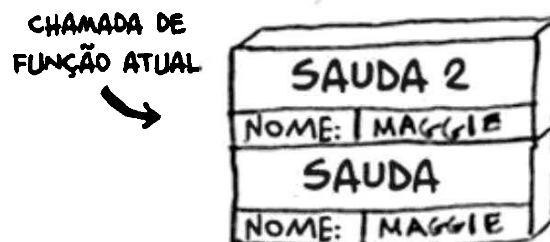
Suponha que você chame `sauda("maggie")`. Primeiro, seu computador aloca uma caixa de memória para essa chamada.



Agora, vamos usar a memória. A variável `nome` é setada para “maggie”. Isso precisa ser salvo.



Cada vez que você faz uma chamada de função, seu computador salva na memória os valores para todas as variáveis. Depois disso, imprime `olá, maggie!`. Então, chama `sauda2("maggie")`.

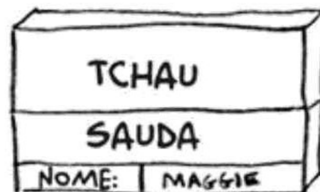


Novamente, seu computador aloca uma caixa de memória para essa chamada de função.

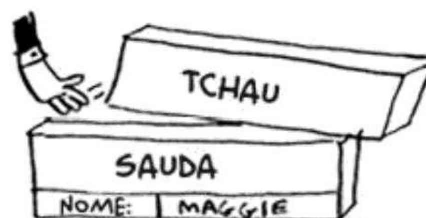
Seu computador está usando uma pilha para estas caixas. A segunda caixa é adicionada em cima da primeira. Você imprime "`como vai maggie?`". Então, retorna da chamada de função. Quando isso acontece, a caixa do topo da pilha é retirada.

Agora, a caixa do topo da pilha aloca os valores da função `sauda`, o que significa que você retornou à função `sauda`. Quando você

chamou a função `sauda2`, a função `sauda` ficou *parcialmente completa*. Esta é a grande ideia por trás desta seção: *quando você chama uma função a partir de outra, a chamada de função fica pausada em um estado parcialmente completo*. Todos os valores das variáveis para aquela função ainda estão armazenados na memória. Agora que você já utilizou a função `sauda2`, você está de volta na função `sauda` e pode continuar de onde parou. Primeiro, imprime "`preparando para dizer tchau...`" e então chama a função `tchau`.



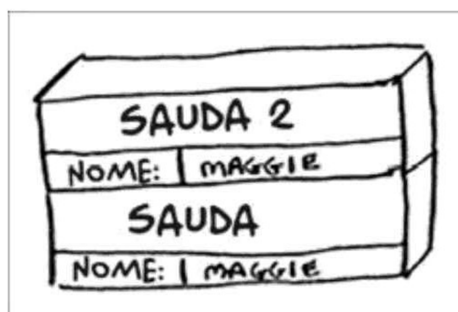
Uma caixa para esta função é adicionada ao topo da pilha. Quando você imprimir `ok, tchau!`, retornará da chamada de função.



Agora, você está de volta à função `sauda`. Não há nada mais a ser feito, e você pode sair da função `sauda` também. Essa pilha usada para guardar as variáveis de múltiplas funções é denominada pilha de chamada.

EXERCÍCIOS

3.1 Suponha que eu forneça uma pilha de chamada como esta:



Quais informações você pode retirar baseando-se apenas nesta pilha de chamada?

Agora, vamos ver esta pilha de chamada sendo executada com uma função recursiva.

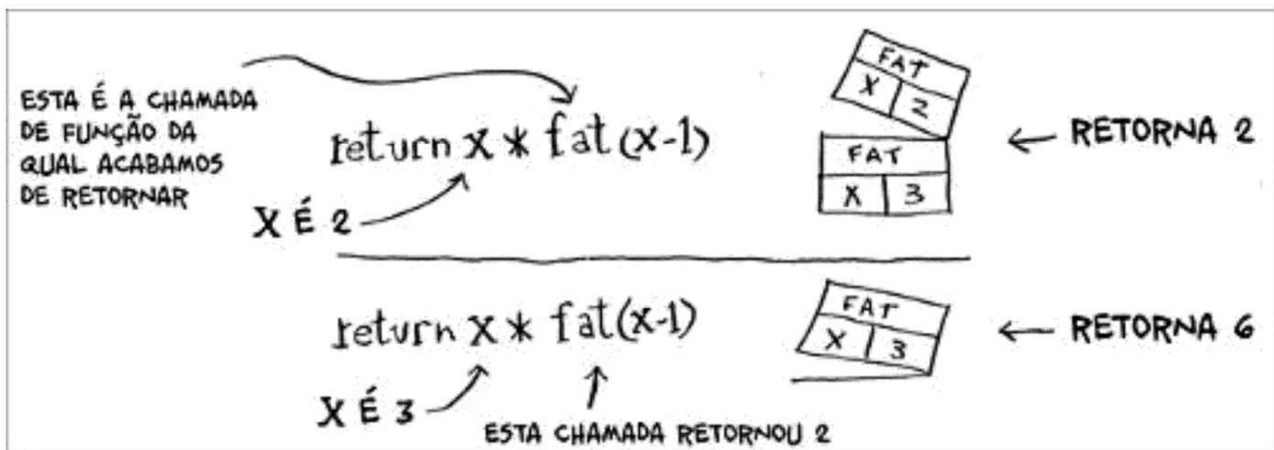
A pilha de chamada com recursão

As funções recursivas também utilizam a pilha de chamada! Vamos analisar isto na prática com a função `fat` (fatorial). `fat(5)` é escrita como 5! e é definida da seguinte forma: 5! = 5 * 4 * 3 * 2 * 1. De forma semelhante, `fat(3)` é 3 * 2 * 1. Aqui está uma função recursiva para calcular a fatorial de um número:

```
def fat(x):
  if x == 1:
    return 1
  else:
    return x * fat(x-1)
```

Agora, você chama a função `fat(3)`. Vamos analisar esta pilha de chamada linha por linha e ver como ela se altera. Lembre-se, a caixa mais próxima ao topo lhe diz em qual chamada a função `fat` se encontra atualmente.

CÓDIGO	PILHA DE CHAMADA	
fat(3)	FAT: x = 3	PRIMEIRA EXECUÇÃO DE fat. X É 3.
if x == 1:	FAT: x = 3	
else:	FAT: x = 3	
return x * fat(x-1) (UMA CHAMADA RECURSIVA!)	FAT: x = 2 FAT: x = 3	
if x == 1: (AGORA, ESTA É A SEGUNDA EXECUÇÃO DE fat. X É 2.)	FAT: x = 2 FAT: x = 3	A CHAMADA DE FUNÇÃO MAIS AO TOPO É A CHAMADA QUE ESTAMOS ATUALMENTE.
else:	FAT: x = 2 FAT: x = 3	PERCEBA QUE AMBAS AS CHAMADAS DE FUNÇÕES POSSUEM UMA VARIÁVEL X, MAS O VALOR DA VARIÁVEL X É DIFERENTE EM CADA UMA.
return x * fat(x-1)	FAT: x = 1 FAT: x = 2 FAT: x = 3	VOCÊ NÃO CONSEGUE ACESSAR ESTA VARIÁVEL X A PARTIR DESTA CHAMADA DE FUNÇÃO E VICE E VERSA.
if x == 1:	FAT: x = 1 FAT: x = 2 FAT: x = 3	
return 1 (NOSSA, NÓS FIZEMOS TRÊS CHAMADAS À FUNÇÃO fat, MAS NÓS NÃO HAVÍAMOS FINALIZADO NENHUMA CHAMADA ATÉ AGORA!)	FAT: x = 1 FAT: x = 2 FAT: x = 3	ESTE É O PRIMEIRO ITEM A SER RETIRADO DA PILHA, O QUE SIGNIFICA QUE ESTA É A PRIMEIRA CHAMADA DA QUAL NÓS RETORNAMOS. RETORNA 1



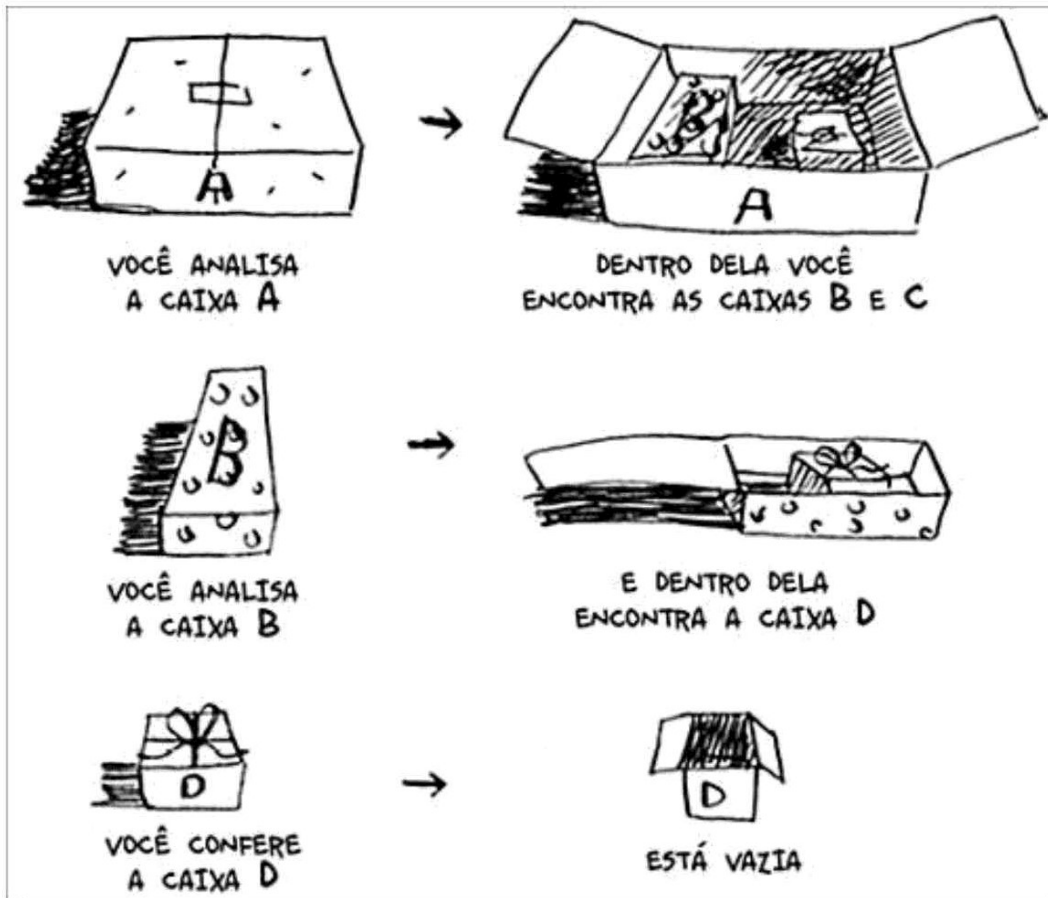
Repare que cada chamada para a função `fat` tem seu próprio valor de x. Você não consegue acessar a mesma função com outro valor de x.

A pilha tem um papel importante na recursão. No primeiro exemplo, mostrei duas abordagens para encontrar a chave. Aqui está a primeira.

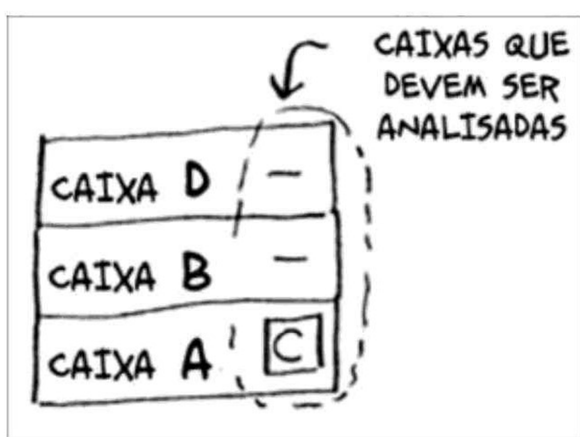
Desta forma, você fez um monte com caixas para analisar, então sabe quais caixas você ainda precisa abrir.

Mas na abordagem recursiva não existem montes.

Se não existem montes, como um algoritmo reconhece quais caixas ele deve procurar? Aqui está um exemplo.



Neste ponto, a pilha de chamada se parece com isto:



O “monte de caixas” é salvo na pilha! Esta é uma pilha com as funções de chamada completadas até a metade, cada uma com a sua lista de caixas, também completadas até a metade, para ser analisadas. Utilizar a pilha é conveniente porque você não precisa acompanhar o monte de caixas – a pilha faz isso para você. Usar a pilha é bom, porém, existe um custo: salvar toda essa informação pode ocupar muita memória. Cada uma destas funções de chamada ocupa um pouco de memória, e quando a sua pilha está muito cheia

é sinal de que seu computador está salvando informação para muitas chamadas de funções. Para esta situação, você tem duas opções:

- Reescrever seu código utilizando loops.
- Utilizar o que chamamos de tail recursion (*recursão de cauda*). Isto é um tópico avançado em recursão e está fora do escopo deste livro. Esta técnica também não é suportada por todas as linguagens de programação.

EXERCÍCIO

3.2 Suponha que você acidentalmente escreva uma função recursiva que fique executando infinitamente. Como você viu, seu computador aloca memória na pilha para cada chamada de função. O que acontece com a pilha quando a função recursiva fica executando infinitamente?

Recapitulando

- Recursão é quando uma função chama a si mesma.
- Toda função recursiva tem dois casos: o caso-base e o caso recursivo.
- Uma pilha tem duas operações: push e pop.
- Todas as chamadas de função vão para a pilha de chamada.
- A pilha de chamada pode ficar muito grande e ocupar muita memória.