Recursividade

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Nós vimos como a definição de tipos de dados adequadas é importante no projeto de programas.

Agora vamos explorar como a forma da definição do tipo de dado pode nos ajudar a escrever o corpo das funções.

Considere a seguinte definição de número natural:

- 0 é um número natural;
- Se n é um número natural, então $n + 1$ é um número natural.

O que esta definição tem de diferente?

No segundo caso, um número natural é definido em termos de outro número natural! Como isso é possível!?

Definições recursivas

Como é possível definir uma coisa em termos dela mesmo?

Este tipo de definição é chamada de **definição recursiva** (ou definição indutiva) e é muito utilizada na computação e matemática.

Para ser válida, uma definição recursiva precisa de

- Pelo menos um caso base (que não depende da própria definição)
- Pelo menos um caso com autorreferência (que depende da própria definição para elementos “menores”)

A partir do(s) caso(s) base, os outros elementos são definidos de forma indutiva pelos casos com autorreferência.

Definições recursivas

Definição de número natural:

- 0 é um número natural;
- Se n é um número natural, então $n + 1$ é um número natural.

O número 4 é natural? Vamos verificar

- Como 4 não é zero, para ele ser natural, o 3 tem que ser natural
- Como 3 não é zero, para ele ser natural, o 2 tem que ser natural
- Como 2 não é zero, pare ele ser natural, o 1 tem que ser natural
- Como 1 não é zero, pare ele ser natural, o 0 tem que ser natural
- Por definição, 0 é natural

Portanto, o 4 é natural.

Note que foi preciso decompor o 4 até chegar no caso base.

Assim como temos definições recursivas, também podemos ter funções recursivas.

Uma **função recursiva** é aquela que chama a si mesmo.

Assim como para definições recursivas, para estar bem formada uma função recursiva precisa de

- Pelo menos um caso base (o valor da função é calculado diretamente)
- Pelo menos um caso com chamada recursiva (depende do valor da função para entradas menores)

Como projetar funções recursivas?

Existem várias técnicas de projeto de funções recursivas, nós vamos explorar uma delas, chamada de diminuição e conquista.

A ideia é diminuir o tamanho do problema original, conquistar o problema menor – diretamente ou recursivamente, e estender a solução do problema menor para o problema original.

No início, para diminuir o problema original, nós vamos explorar a relação entre autorreferência na definição do tipo de dado e a chamada recursiva na função que processa o tipo de dado.

Exemplo: soma naturais

Projete uma função recursiva que some todos os números naturais menores ou iguais que um determinado n.

Exemplo: soma naturais

```
def soma_naturais(n: int) -> int:
    '''
    Soma todos os número naturais menores
    ou iguais que *n*.
    Requer que n >= 0.
    Exemplos
    >>> soma_naturais(0)
    0
    >>> soma_naturais(1)
    1
    >>> soma_naturais(2)
    3
    >>> soma_naturais(3)
    6
    >>> soma_naturais(4)
    10
    '''
    return 0
```

Como a definição de número natural tem dois casos, vamos começar a implementação da função com dois casos.

```
if n == 0:
    ...
else:
    n ...
```

Como o segundo caso da definição de número natural tem uma autorreferência, vamos colocar uma chamada recursiva no segundo caso da função.

```
if n == 0:
    ...
else:
    n ... soma_naturais(n - 1)
```

Exemplo: soma naturais

```python
def soma_naturais(n: int) -> int:
    '''
    Soma todos os número naturais menores
    ou iguais que *n*.
    Requer que n >= 0.
    Exemplos
    >>> soma_naturais(0)
    0
    >>> soma_naturais(1)
    1
    >>> soma_naturais(2)
    3
    >>> soma_naturais(3)
    6
    >>> soma_naturais(4)
    10
    '''
```

```python
def soma_naturais(n: int) -> int:
    if n == 0:
        # Qual é a soma dos naturais
        # até n == 0?
        soma = ...
    else:
        # Tendo a soma dos naturais
        # até n - 1 e o natural n,
        # como obter a soma para os
        # naturais até n?
        soma = n ... soma_naturais(n - 1)
    return soma
```

Exemplo: soma naturais

```python
def soma_naturais(n: int) -> int:
    '''
    Soma todos os número naturais menores
    ou iguais que *n*.
    Requer que n >= 0.
    Exemplos
    >>> soma_naturais(0)
    0
    >>> soma_naturais(1)
    1
    >>> soma_naturais(2)
    3
    >>> soma_naturais(3)
    6
    >>> soma_naturais(4)
    10
    '''
```

```python
def soma_naturais(n: int) -> int:
    if n == 0:
        # Qual é a soma dos naturais
        # até n == 0?
        soma = 0
    else:
        # Tendo a soma dos naturais
        # até n - 1 e o natural n,
        # como obter a soma para os
        # naturais até n?
        soma = n + soma_naturais(n - 1)
    return soma
```

Exemplo: exponencial

Projete uma função recursiva que receba como entrada um número $a \neq 0$ e um número natural n e calcule o valor a^n.

Exemplo: exponencial

```
def potencia(a: float, n: int) -> float:
    '''
    Calcula *a* elevado a *n*.
    Requer que a != 0 e n >= 0.
    Exemplos
    >>> potencia(2.0, 0)
    1.0
    >>> potencia(2.0, 1)
    2.0
    >>> potencia(2.0, 2)
    4.0
    >>> potencia(2.0, 3)
    8.0
    >>> potencia(3.0, 3)
    27.0
    >>> potencia(3.0, 4)
    81.0
    '''
    return 0.0
```

Como a definição de número natural tem dois casos, vamos começar a implementação da função com dois casos.

```
if n == 0:
    a ...
else:
    a ... n ...
```

Como o segundo caso da definição de número natural tem uma autorreferência, vamos colocar uma chamada recursiva no segundo caso da função.

```
if n == 0:
    a ...
else:
    a ... n ... potencia(a, n - 1)
```

Exemplo: exponencial

```python
def potencia(a: float, n: int) -> float:
    '''
    Calcula *a* elevado a *n*.
    Requer que a != 0 e n >= 0.
    Exemplos
    >>> potencia(2.0, 0)
    1.0
    >>> potencia(2.0, 1)
    2.0
    >>> potencia(2.0, 2)
    4.0
    >>> potencia(2.0, 3)
    8.0
    >>> potencia(3.0, 3)
    27.0
    >>> potencia(3.0, 4)
    81.0
    '''
```

```python
def potencia(a: float, n: int) -> float:
    if n == 0:
        # Qual é o valor de a^n
        # quando n == 0?
        an = a ...
    else:
        # Tendo a potência a^(n - 1),
        # o valor de a e n, como
        # calcular a^n?
        an = a ... n ... potencia(a, n - 1)
    return an
```

Exemplo: exponencial

```python
def potencia(a: float, n: int) -> float:
    '''
    Calcula *a* elevado a *n*.
    Requer que a != 0 e n >= 0.
    Exemplos
    >>> potencia(2.0, 0)
    1.0
    >>> potencia(2.0, 1)
    2.0
    >>> potencia(2.0, 2)
    4.0
    >>> potencia(2.0, 3)
    8.0
    >>> potencia(3.0, 3)
    27.0
    >>> potencia(3.0, 4)
    81.0
    '''
```

```python
def potencia(a: float, n: int) -> float:
    if n == 0:
        # Qual é o valor de a^n
        # quando n == 0?
        an = 1.0
    else:
        # Tendo a potência a^(n - 1),
        # o valor de a e n, como
        # calcular a^n?
        an = a * potencia(a, n - 1)
    return an
```

Exemplo: fatorial

Projete uma função recursiva que calcule o fatorial de n, isto é, o produto dos n primeiros números naturais maiores que 0.

Exemplo: fatorial

```
def fatorial(n: int) -> int:
    '''
    Calcula o fatorial de *n*, isto é,
    o produto dos *n* primeiros números
    naturais maiores que 0.
    Requer que n >= 0.
    Exemplos
    >>> fatorial(0)
    1
    >>> fatorial(1)
    1
    >>> fatorial(2)
    2
    >>> fatorial(3)
    6
    >>> fatorial(4)
    24
    '''
    return 0
```

Como a definição de número natural tem dois casos, vamos começar a implementação da função com dois casos.

```
if n == 0:
    ...
else:
    n ...
```

Como o segundo caso da definição de número natural tem uma autorreferência, vamos colocar uma chamada recursiva no segundo caso da função.

```
if n == 0:
    ...
else:
    n ... fatorial(n - 1)
```

Exemplo: fatorial

```
def fatorial(n: int) -> int:
    '''
    Calcula o fatorial de *n*, isto é,
    o produto dos *n* primeiros números
    naturais maiores que 0.
    Requer que n >= 0.
    Exemplos
    >>> fatorial(0)
    1
    >>> fatorial(1)
    1
    >>> fatorial(2)
    2
    >>> fatorial(3)
    6
    >>> fatorial(4)
    24
    '''
```

```
def fatorial(n: int) -> int:
    if n == 0:
        # Qual é o fatorial de 0,
        # isto é, o produto dos
        # primeiros n == 0
        # naturais maiores que 0?
        fat = ...
    else:
        # Tendo o fatorial de n - 1
        # e o natural n, como obter
        # o fatorial de n?
        fat = n ... fatorial(n - 1)
    return fat
```

Exemplo: fatorial

```
def fatorial(n: int) -> int:
    '''
    Calcula o fatorial de *n*, isto é,
    o produto dos *n* primeiros números
    naturais maiores que 0.
    Requer que n >= 0.
    Exemplos
    >>> fatorial(0)
    1
    >>> fatorial(1)
    1
    >>> fatorial(2)
    2
    >>> fatorial(3)
    6
    >>> fatorial(4)
    24
    '''
```

```
def fatorial(n: int) -> int:
    if n == 0:
        # Qual é o fatorial de 0,
        # isto é, o produto dos
        # primeiros n == 0
        # naturais maiores que 0?
        fat = 1
    else:
        # Tendo o fatorial de n - 1
        # e o natural n, como obter
        # o fatorial de n?
        fat = n * fatorial(n - 1)
    return fat
```

Aspectos importantes no projeto de funções recursivas

Quando estamos projetando funções recursivas, temos que considerar dois aspectos:

- A chamada recursiva deve ser feita para uma **entrada "menor"**, dessa forma temos a certeza que o caso base será alcançado e a função terminará.
- Devemos **confiar na chamada recursiva**, isto é, que ela produz a resposta correta, e nos preocuparmos apenas em como utilizar essa resposta para calcular o resultado da função.

Podemos projetar funções recursivas que operam em listas de forma similar a funções que operam com números naturais. Considere a seguinte definição de lista:

Uma lista é:

- Vazia; ou
- Um elemento seguido de uma lista (resto da lista)

Assim como a definição de número natural, essa definição de lista também tem autorreferência (é indutiva).

Portanto, para implementar uma função que processa uma lista, podemos usar a mesma estratégia que usamos para implementar funções recursivas que processam números naturais.

Vamos ver alguns exemplos.

Exemplo: soma

Projete uma função recursiva que some os elementos de uma lista.

Exemplo: soma

```
def soma(lst: list[int]) -> int:
    '''
    Soma os elementos de *lst*.
    Exemplos
    >>> soma([])
    0
    >>> soma([6])
    6
    >>> soma([3, 6])
    9
    >>> soma([7, 3, 6])
    16
    '''
    return 0
```

Como a definição de lista tem dois casos, vamos começar a implementação da função com dois casos.

```
if lst == []:
    ...
else:
    # as partes de lst
    lst[0] ... lst[1:]
```

Como o segundo caso da definição de lista tem uma autorreferência, isto é, `lst[1:]` é uma lista, vamos fazer uma chamada recursiva para `lst[1:]`.

```
else:
    lst[0] ... soma(lst[1:])
```

Exemplo: soma

```
def soma(lst: list[int]) -> int:
    '''
    Soma os elementos de *lst*.
    Exemplos
    >>> soma([])
    0
    >>> soma([6])
    6
    >>> soma([3, 6])
    9
    >>> soma([7, 3, 6])
    16
    '''
```

```
def soma(lst: list[int]) -> int:
    if lst == []:
        # Qual é a soma dos elementos
        # de uma lista vazia?
        s = ...
    else:
        # Sabendo a soma do resto da lista
        # e o valor do primeiro elemento,
        # como obter a soma da lista?
        s = lst[0] ... soma(lst[1:])
    return s
```

Exemplo: soma

```
def soma(lst: list[int]) -> int:
    '''
    Soma os elementos de *lst*.
    Exemplos
    >>> soma([])
    0
    >>> soma([6])
    6
    >>> soma([3, 6])
    9
    >>> soma([7, 3, 6])
    16
    '''
```

```
def soma(lst: list[int]) -> int:
    if lst == []:
        # Qual é a soma dos elementos
        # de uma lista vazia?
        s = 0
    else:
        # Sabendo a soma do resto da lista
        # e o valor do primeiro elemento,
        # como obter a soma da lista?
        s = lst[0] + soma(lst[1:])
    return s
```

Exemplo: contagem

Projete uma função recursiva que conte quantas vezes um determinado número aparece em uma lista de números.

Exemplo: contagem

```
def freq(v: int, lst: list[int]) -> int:
    '''
    Conta quantas vezes *v* aparece
    em *lst*.

    Exemplos
    >>> freq(1, [])
    0
    >>> freq(1, [7])
    0
    >>> freq(1, [1, 7, 1])
    2
    >>> freq(4, [4, 1, 7, 4, 4])
    3
    '''
    return 0
```

Como a definição de lista tem dois casos, vamos começar a implementação da função com dois casos.

```
if lst == []:
    v ...
else:
    v ... lst[0] ... lst[1:]
```

Como o segundo caso da definição de lista tem uma autorreferência, isto é, `lst[1:]` é uma lista, vamos fazer uma chamada recursiva para `lst[1:]`.

```
else:
    v ... lst[0] ... freq(v, lst[1:])
```

Exemplo: contagem

```
def freq(v: int, lst: list[int]) -> int:
    '''
    Conta quantas vezes *v* aparece
    em *lst*.

    Exemplos
    >>> freq(1, [])
    0
    >>> freq(1, [7])
    0
    >>> freq(1, [1, 7, 1])
    2
    >>> freq(4, [4, 1, 7, 4, 4])
    3
    '''
```

```
def freq(v: int, lst: list[int]) -> int:
    if lst == []:
        # Quantas vezes v aparece
        # na lista vazia?
        cont = v ...
    else:
        # Sabendo a quantidade de vezes
        # que v aparece em lst[1:],
        # como determinamos a quantidade
        # de vezes que v aparece em lst?
        cont = ...
        v... lst[0] ... freq(v, lst[1:])
    return cont
```

Exemplo: contagem

```
def freq(v: int, lst: list[int]) -> int:
    '''
    Conta quantas vezes *v* aparece
    em *lst*.

    Exemplos
    >>> freq(1, [])
    0
    >>> freq(1, [7])
    0
    >>> freq(1, [1, 7, 1])
    2
    >>> freq(4, [4, 1, 7, 4, 4])
    3
    '''
```

```
def freq(v: int, lst: list[int]) -> int:
    if lst == []:
        # Quantas vezes v aparece
        # na lista vazia?
        cont = ...
    else:
        # Sabendo a quantidade de vezes
        # que v aparece em lst[1:],
        # como determinamos a quantidade
        # de vezes que v aparece em lst?
        if v == lst[0]:
            cont = 1 + freq(v, lst[1:])
        else:
            cont = freq(v, lst[1:])
    return cont
```

Exemplo: ordem

Projete uma função recursiva que verifique se os elementos de uma lista estão em ordem não decrescente.

Diminuição lógica

Apesar das funções `soma`, `freq` e `em_ordem` funcionarem corretamente, elas não são eficientes.

Isto porque a operação de slice `lst[1:]` cria uma nova lista copiando todos os elementos de `lst` a partir do índice 1.

Para resolver esse problema, podemos diminuir `lst` de **forma lógica ao invés de forma física** (com o slice).

A ideia é usar um parâmetro extra `i` que indica de onde a soma deve começar. Na primeira chamada `i = 0` e na chamada recursiva `i + 1`. O caso base é atingindo quando `i == len(lst)`.

Exemplo: soma com índice incrementando

```
def soma_inc(lst: list[int], i: int) -> int:
    '''
    Soma os elementos de *lst* a partir
    de *i*, isto é, soma os elementos
    de *lst[i:]*.
    Requer que 0 <= i <= len(lst).
    >>> soma_inc([7, 3, 6], 0)
    16
    >>> soma_inc([7, 3, 6], 1)
    9
    >>> soma_inc([7, 3, 6], 2)
    6
    >>> soma_inc([7, 3, 6], 3)
    0
    '''
```

```
def soma_inc(lst: list[int], i: int) -> int:
    if i >= len(lst):
        # Qual é a soma dos elementos
        # de lst a partir de i?
        s = 0
    else:
        # Tendo a soma dos elementos de
        # lst a partir de i + 1 (chamada
        # recursiva) e lst[i], como
        # obter a soma dos elementos
        # de lst a partir de i?
        s = lst[i] + soma_inc(lst, i + 1)
    return s
```

Exemplo: soma com índice decrementando

Ao invés de começar o índice com 0 e incrementar na chamada recursiva, podemos começar o índice com `len(lst)` e decrementar o índice na chamada recursiva.

Desse forma, o índice funciona como um **tamanho lógico** para `lst` e podemos pensar em recursão com lista como pensamos para recursão com número natural.

Vamos chamar o argumento de `n` ao invés de `i` para destacar a relação com o tamanho.

Exemplo: soma com índice decrementando

```python
def soma_dec(lst: list[int], n: int) -> int:
    '''
    Soma os primeiro *n* elementos de *lst*,
    isto é, soma os elementos de *lst[:n]*.
    Requer que 0 <= n <= len(lst)
    >>> soma_dec([7, 3, 6], 0)
    0
    >>> soma_dec([7, 3, 6], 1)
    7
    >>> soma_dec([7, 3, 6], 2)
    10
    >>> soma_dec([7, 3, 6], 3)
    16
    '''
```

```python
def soma_dec(lst: list[int], n: int) -> int:
    if n == 0:
        # Qual é a soma de lst, sendo
        # que o "tamanho" (n) de lst é 0?
        s = 0
    else:
        # Tendo a soma dos elementos
        # de lst sem o "último" elemento
        # de lst (chamada recursiva)
        # e o último elemento (lst[n-1]),
        # como obtemos a soma de todos os
        # elementos de lst[:n]?
        s = lst[n - 1] + soma_dec(lst, n - 1)
    return s
```

Esta técnica sempre funciona? Ou seja, se eu aplicar a ideia de diminuir e conquistar eu consigo projetar um algoritmo para resolver qualquer problema?

Não!

Mas então, quando podemos aplicar a técnica de projeto diminuição e conquista?

- Quando conseguimos resolver o caso base
- Quando a solução do problema menor pode ser estendida para a solução do problema original

Se quisermos encontrar todos os divisores de um número n, podemos aplicar essa técnica?

Conseguimos resolver o caso base? Sim.

Tendo os divisores de $n - 1$, podemos encontrar os divisores de n? Sabendo os divisores de $9(1, 3, 9)$, podemos determinar os divisores de $10(1, 2, 5, 10)$? Não!

Então essa técnica não é adequada para esse problema.