

Recursividade

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Uma função é **recursiva** quando ela chama a si mesmo de forma direta ou indireta.

A recursividade é uma técnica muito poderosa e bastante utilizada na Computação e Matemática.

De certa a forma a recursividade é um caso especial da decomposição de problemas.

De forma geral podemos resolver um problema decompondo-o em subproblemas mais simples, resolvendo os subproblemas e combinando as soluções para obter a solução do problema inicial.

A recursividade surge quando decomponemos um problema em subproblemas do *mesmo tipo*, pois nesses casos podemos utilizar *o mesmo processo* para resolver o problema inicial e os subproblemas. Note que para que o processo funcione, devemos definir situações limites em que o problema seja resolvido diretamente, sem precisar ser decomposto, que são os casos bases.

Então, para que possamos aplicar a recursividade é necessário decompor um problema em subproblemas do mesmo tipo. Mas como fazer a decomposição?

- Para algumas problemas pode ser necessário um momento “eureka” e inventar uma forma de fazer a decomposição, o que requer experiência.
- Mas para a maioria dos problemas podemos fazer uma decomposição “direta”, baseada na definição com autorreferência do dado (estrutura) que representa o problema.

A primeira forma gera **funções recursivas generativas**, já a segunda forma gerar **funções recursivas estruturais**. Vamos explorar agora essa segunda forma.

Para escrever os próximos exemplos não vamos usar

- Arranjos; e
- Laços de repetição

Como representar uma quantidade arbitrária de dados sem arranjos?

- Usando encadeamento

Definição de lista

A definição para nó que utilizamos foi:

```
@dataclass
class No:
    item: int
    prox: No | None
```

Para facilitar o projeto e entendimento das próximas funções vamos utilizar a seguinte definição:

```
@dataclass
class No:
    primeiro: int
    resto: Lista
```

```
Lista = No | None
```

De maneira formal, uma **Lista** é:

- Vazia (**None**); ou
- Um nó (**No**) com um elemento e o resto, que é uma **Lista**.

Para implementar funções que processam uma Lista, vamos explorar a relação entre autorreferência (na definição) e recursividade (na função):

```
def fn_para_lista(lst: Lista) -> ...:
    if lst is None:
        return ...
    else:
        return lst.primeiro ... \
            fn_para_lista(lst.resto)
```

Projete uma função que some os elementos de uma lista.

```
def soma(lst: Lista) -> int:
```

```
    '''Soma os elementos de *lst*.
```

```
        lst
    /-----\
soma(No(10, No(4, No(3, None)))) -> 17
    | \-----/
primeiro soma(resto)
    10      7
```

Como computar soma(lst) a partir de

lst.primeiro e soma(lst.resto)?

```
    ...
```

```
if lst is None:
```

```
    return ...
```

```
else:
```

```
    return lst.primeiro ... soma(lst.resto)
```

```
def soma(lst: Lista) -> int:
    '''Soma os elementos de *lst*.
```

```

                lst
            /-----\
soma(No(10, No(4, No(3, None)))) -> 17
    | \-----/
primeiro soma(resto)
    10      7

```

Como computar soma(lst) a partir de
lst.primeiro e soma(lst.resto)?

```
'''
if lst is None:
    return 0
else:
    return lst.primeiro + soma(lst.resto)
```

```
def soma(lst: Lista) -> int:
    '''Soma os elementos de *lst*.
```

```

                lst
            /-----\
soma(No(10, No(4, No(3, None)))) -> 17
                \-----/ | |
                    s   p.primeiro |
                    14   p.resto

```

Como inicializar s e p?
Como atualizar s e p?

```
'''
s = 0
p = lst
while p is not None:
    s += p.primeiro
    p = p.resto
return s
```


Projete uma função que determine a quantidade de itens em uma lista.

Número de itens

```
def num_itens(lst: Lista) -> int:  
    '''Devolve a quantidade de itens em *lst*.
```

```
                                lst  
                            /-----\  
num_itens(No(10, No(4, No(3, None)))) -> 3  
    |   \-----/\  
primeiro  num_itens(resto)  
    10      2
```

Como computar `num_itens(lst)` a partir de
`lst.primeiro` e `num_itens(lst.resto)`?

```
...
```

```
if lst is None:  
    return ...  
else:  
    return lst.primeiro ... num_itens(lst.resto)
```

Número de itens

```
def num_itens(lst: Lista) -> int:  
    '''Devolve a quantidade de itens em *lst*.
```

```
                lst  
            /-----\  
num_itens(No(10, No(4, No(3, None)))) -> 3  
    | \-----/  
primeiro  num_itens(resto)  
    10      2
```

Como computar `num_itens(lst)` a partir de `lst.primeiro` e `num_itens(lst.resto)`?

```
'''
```

```
if lst is None:  
    return 0  
else:  
    return 1 + num_itens(lst.resto)
```

```
def num_itens(lst: Lista) -> int:  
    '''Devolve a quantidade de itens em *lst*.
```

```
                lst  
            /-----\  
num_itens(No(10, No(4, No(3, None)))) -> 3  
                \-----/  
                num  p.primeiro |  
                2      p.resto
```

Como inicializar `num` e `p`?

Como atualizar `num` e `p`?

```
'''
```

```
num = 0  
p = lst  
while p is not None:  
    num += 1  
    p = p.resto  
return num
```

Projete uma função que verifique se todos os elementos de uma lista são pares.

Todos pares

```
def todos_pares(lst: Lista) -> bool:  
    '''Devolve True se todos os elementos  
    de *lst* são pares, False caso contrário.
```

```
                lst  
            /-----\  
todos_pares(No(10, No(4, No(6, None)))) -> True  
    | \-----/  
primeiro  todos_pares(resto)  
    10          True
```

Como computar `todos_pares(lst)` a partir de
`lst.primeiro` e `todos_pares(lst.resto)`?

```
'''
```

Todos pares

```
def todos_pares(lst: Lista) -> bool:  
    '''Devolve True se todos os elementos  
    de *lst* são pares, False caso contrário.
```

```
                lst  
            /-----\  
todos_pares(No(11, No(4, No(6, None)))) -> False  
    | \-----/  
primeiro  todos_pares(resto)  
    11          True
```

Como computar `todos_pares(lst)` a partir de
`lst.primeiro` e `todos_pares(lst.resto)`?
'''

Todos pares

```
def todos_pares(lst: Lista) -> bool:
    '''Devolve True se todos os elementos
    de *lst* são pares, False caso contrário.
```

```

                lst
            /-----\
todos_pares(No(10, No(4, No(3, None)))) -> False
        |   \-----/
    primeiro todos_pares(resto)
        10         False
```

Como computar `todos_pares(lst)` a partir de
`lst.primeiro` e `todos_pares(lst.resto)`?

```
'''
```

```
if lst is None:
    return ...
else:
    return lst.primeiro ... \
           todos_pares(lst.resto)
```

Todos pares

```
def todos_pares(lst: Lista) -> bool:
    '''Devolve True se todos os elementos
    de *lst* são pares, False caso contrário.
```

```

                lst
            /-----\
todos_pares(No(10, No(4, No(3, None)))) -> False
        |   \-----/
    primeiro todos_pares(resto)
        10         False
```

Como computar `todos_pares(lst)` a partir de
`lst.primeiro` e `todos_pares(lst.resto)`?

```
'''
```

```
if lst is None:
    return True
else:
    return lst.primeiro % 2 == 0 and \
           todos_pares(lst.resto)
```


Todos pares

```
def todos_pares(lst: Lista) -> bool:
    '''Devolve True se todos os elementos
    de *lst* são pares, False caso contrário.
```

```

                lst
            /-----\
todos_pares(No(10, No(4, No(3, None)))) -> False
            | \-----/
        primeiro todos_pares(resto)
            10      False
```

Como computar todos_pares(lst) a partir de
lst.primeiro e todos_pares(lst.resto)?
'''

```
return lst is None or \
        lst.primeiro % 2 == 0 and \
            todos_pares(lst.resto)
```

```
def todos_pares(lst: Lista) -> bool:
    '''Devolve True se todos os elementos
    de *lst* são pares, False caso contrário.
```

```

                lst
            /-----\
todos_pares(No(10, No(4, No(3, None)))) -> False
                \-----/ | |
                pares p.primeiro |
                True      p.resto
```

Como inicializar pares e p?
Como atualizar pares e p?

```
'''
pares = True
p = lst
while pares and p is not None:
    pares = p.primeiro % 2 == 0
    p = p.resto
return pares
```

Todos pares

```
def todos_pares(lst: Lista) -> bool:
    '''Devolve True se todos os elementos
    de *lst* são pares, False caso contrário.
```

```

                lst
            /-----\
todos_pares(No(10, No(4, No(3, None)))) -> False
            | \-----/
        primeiro todos_pares(resto)
            10      False
```

Como computar todos_pares(lst) a partir de
lst.primeiro e todos_pares(lst.resto)?
'''

```
return lst is None or \
        lst.primeiro % 2 == 0 and \
        todos_pares(lst.resto)
```

```
def todos_pares(lst: Lista) -> bool:
    '''Devolve True se todos os elementos
    de *lst* são pares, False caso contrário.
```

```

                lst
            /-----\
todos_pares(No(10, No(4, No(3, None)))) -> False
            \-----/ | |
                pares p.primeiro |
                True      p.resto
```

Como inicializar pares e p?
Como atualizar pares e p?
'''

```
p = lst
while p is not None:
    if p.primeiro % 2 != 0:
        return False
    p = p.resto
return True
```

Todos pares

```
def todos_pares(lst: Lista) -> bool:
    '''Devolve True se todos os elementos
    de *lst* são pares, False caso contrário.
```

```

                lst
            /-----\
todos_pares(No(10, No(4, No(3, None)))) -> False
            | \-----/
        primeiro todos_pares(resto)
            10      False
```

Como computar todos_pares(lst) a partir de lst.primeiro e todos_pares(lst.resto)?

```
'''
return lst is None or \
        lst.primeiro % 2 == 0 and \
            todos_pares(lst.resto)
```

```
def todos_pares(lst: Lista) -> bool:
    '''Devolve True se todos os elementos
    de *lst* são pares, False caso contrário.
```

```

                lst
            /-----\
todos_pares(No(10, No(4, No(3, None)))) -> False
            \-----/ | |
                pares p.primeiro |
                True      p.resto
```

Como inicializar pares e p?

Como atualizar pares e p?

```
'''
while lst is not None and lst.primeiro % 2 == 0:
    lst = lst.resto
return lst is None
```

Projete uma função que verifique se um item está em uma lista.

```
def contem(lst: Lista, v: int) -> bool:  
    '''Devolve True se *v* está em *lst*,  
    False caso contrário.
```

```
                lst                v  
            /-----\ |  
contem(No(10, No(4, No(3, None))), 4) -> True  
    | \-----/ |  
primeiro  contem(resto, v)  
    10          True
```

Como computar `contem(lst, v)` a partir de
`lst.primeiro` e `contem(lst.resto, v)`?

```
'''  
  
if lst is None:  
    return ... v  
else:  
    return v ... lst.primeiro ... \  
        contem(lst.resto, v)
```

```
def contem(lst: Lista, v: int) -> bool:  
    '''Devolve True se *v* está em *lst*,  
    False caso contrário.
```

```
                lst                v  
            /-----\ |  
contem(No(10, No(4, No(3, None))), 4) -> True  
    | \-----/ |  
primeiro  contem(resto, v)  
    10          True
```

Como computar `contem(lst, v)` a partir de
`lst.primeiro` e `contem(lst.resto, v)`?

```
'''  
  
if lst is None:  
    return False  
else:  
    return v == lst.primeiro or \  
           contem(lst.resto, v)
```

```
def contem(lst: Lista, v: int) -> bool:
    '''Devolve True se *v* está em *lst*,
    False caso contrário.
```

```

                lst                v
            /-----\ |
contem(No(10, No(4, No(3, None))), 4) -> True
            | \-----/
        primeiro  contem(resto, v)
            10          True
    
```

Como computar `contem(lst, v)` a partir de `lst.primeiro` e `contem(lst.resto, v)`?

```
'''
```

```
return lst is not None and \
    (v == lst.primeiro or
     contem(lst.resto, v))
```

```
def contem(lst: Lista, v: int) -> bool:
    '''Devolve True se *v* está em *lst*,
    False caso contrário.
```

```

                lst                v
            /-----\ |
contem(No(10, No(4, No(3, None))), 4) -> True
                \-----/ | |
                achou p.primeiro |
                False          p.resto
    
```

Como inicializar `achou` e `p`?

Como atualizar `achou` e `p`?

```
'''
```

```
achou = False
p = lst
while not achou and p is not None:
    achou = v == p.primeiro
    p = p.resto
return False
```

Contém

```
def contem(lst: Lista, v: int) -> bool:
    '''Devolve True se *v* está em *lst*,
    False caso contrário.
```

```

                lst                v
            /-----\             |
contem(No(10, No(4, No(3, None))), 4) -> True
            | \-----/
        primeiro  contem(resto, v)
            10          True
```

Como computar `contem(lst, v)` a partir de `lst.primeiro` e `contem(lst.resto, v)`?

```
'''
return lst is not None and \
    (v == lst.primeiro or
     contem(lst.resto, v))
```

```
def contem(lst: Lista, v: int) -> bool:
    '''Devolve True se *v* está em *lst*,
    False caso contrário.
```

```

                lst                v
            /-----\             |
contem(No(10, No(4, No(3, None))), 4) -> True
                \-----/      |      |
                achou p.primeiro |
                False           p.resto
```

Como inicializar `achou` e `p`?

Como atualizar `achou` e `p`?

```
'''
```

```
while lst is not None and v != lst.primeiro:
    lst = lst.resto
return lst is not None
```


Projete uma função que modifique uma lista somando 1 em cada um dos seus elementos.

```
def soma1(lst: Lista):
    '''Modifica *lst* somando 1 a cada elemento
    de *lst*.
```

```

                lst
            /-----\
soma1(No(10, No(4, No(3, None))))
    |   \-----/
primeiro soma1(resto)
    10   No(5, No(4, None))

```

Como implementar soma1(lst) usando
 lst.primeiro e soma1(lst.resto)?
 ...

```
if lst is None:
    ...
else:
    lst.primeiro ... \
        soma1(lst.resto)
```

```
def soma1(lst: Lista):
    '''Modifica *lst* somando 1 a cada elemento
    de *lst*.
```

```

                lst
            /-----\
soma1(No(10, No(4, No(3, None))))
      | \-----/
primeiro soma1(resto)
      10  No(5, No(4, None))

```

Como implementar soma1(lst) usando
 lst.primeiro e soma1(lst.resto)?
 ...

```
if lst is None:
    return
else:
    lst.primeiro += 1
    soma1(lst.resto)
```

```
def soma1(lst: Lista):
    '''Modifica *lst* somando 1 a cada elemento
    de *lst*.
```

```

                lst
            /-----\
soma1(No(10, No(4, No(3, None))))
      | \-----/
primeiro soma1(resto)
      10  No(5, No(4, None))

```

Como implementar `soma1(lst)` usando
`lst.primeiro` e `soma1(lst.resto)`?

```
'''
if lst is not None:
    lst.primeiro += 1
    soma1(lst.resto)
```

```
def soma1(lst: Lista):
    '''Modifica *lst* somando 1 a cada elemento
    de *lst*.
```

```

                lst
            /-----\
soma1(No(10, No(4, No(3, None))))
      \-----/ | |
        11   5 p.primeiro |
                          |
                          p.resto

```

Como modificar `p.primeiro` e atualizar `p`?

```
'''
p = lst
while p is not None:
    p.primeiro += 1
    p = p.resto
```

```
def soma1(lst: Lista):
    '''Modifica *lst* somando 1 a cada elemento
    de *lst*.
```

```

                lst
            /-----\
soma1(No(10, No(4, No(3, None))))
      | \-----/
primeiro soma1(resto)
      10  No(5, No(4, None))

```

Como implementar soma1(lst) usando
lst.primeiro e soma1(lst.resto)?
'''

```
if lst is not None:
    lst.primeiro += 1
    soma1(lst.resto)
```

```
def soma1(lst: Lista):
    '''Modifica *lst* somando 1 a cada elemento
    de *lst*.
```

```

                lst
            /-----\
soma1(No(10, No(4, No(3, None))))
      \-----/ | |
        11   5 p.primeiro |
                                   |
                                   p.resto

```

Como modificar p.primeiro e atualizar p?
'''

```
while lst is not None:
    lst.primeiro += 1
    lst = lst.resto
```

Projete uma função que modifique uma lista criando uma cópia de cada item da lista (que deve ficar após o item que foi copiado).

```
def duplica(lst: Lista):  
    ...  
    Modifica *lst* criando uma cópia de cada nó  
    que é colocado após o nó copiado.  
  
    Como implementar duplica(lst) usando  
    lst.primeiro e duplica(lst.resto)?  
    ...  
  
    if lst is None:  
        return ...  
    else:  
        lst.primeiro  
        duplica(lst.resto)
```

```
def duplica(lst: Lista):  
    ...  
    Modifica *lst* criando uma cópia de cada nó  
    que é colocado após o nó copiado.  
  
    Como implementar duplica(lst) usando  
    lst.primeiro e duplica(lst.resto)?  
    ...  
  
    if lst is None:  
        return  
    else:  
        duplica(lst.resto)  
        lst.resto = No(lst.primeiro, lst.resto)
```



```
def duplica(lst: Lista):  
    '''  
    Modifica *lst* criando uma cópia de cada nó  
    que é colocado após o nó copiado.  
  
    Como implementar duplica(lst) usando  
    lst.primeiro e duplica(lst.resto)?  
    '''  
  
    if lst is not None:  
        duplica(lst.resto)  
        lst.resto = No(lst.primeiro, lst.resto)
```

```
def duplica(lst: Lista):  
    '''  
    Modifica *lst* criando uma cópia de cada nó  
    que é colocado após o nó copiado.  
  
    Como duplicar um nó p e atualizar p?  
    '''  
  
    p = lst  
    while p is not None:  
        p.resto = No(p.primeiro, p.resto)  
        p = p.resto.resto
```

Projetar uma função recursiva pode ser um desafio se for preciso “inventar” uma forma de decompor o problema.

No entanto, se fizermos a decomposição estrutural, isto é, decompor o problema conforme o dado que descreve o problema, então o projeto de funções recursivas se torna um processo mais sistemático.

Podemos aplicar a o processo de projeto de funções recursivas baseada na decomposição estrutural em dados que não sejam listas? Sim, podemos aplicar em qualquer dado que tenha autorreferência!

Como definir um número natural? Usando autorreferência.

Um **número natural** é:

- 0; ou
- $n + 1$, onde n é um **número natural**.

A partir dessa definição podemos criar um modelo de função para processar número naturais (que precisam ser decompostos):

```
def fn_para_n(n: int) -> ...:
    if n == 0:
        return ...
    else:
        return n ... fn_para_n(n - 1)
```

Projete uma função que some todos os número naturais até um dado n .

```
def soma(n: int) -> int:
    ...

    Devolve a soma de todos os números
    naturais até *n*. Requer que n >=0.
    Exemplos
    >>> soma(0)
    0
    >>> soma(4)
    10
    ...

    if n == 0:
        return 0
    else:
        return n + soma(n - 1)
```

Como definir um número natural? Usando autorreferência.

Um **número natural** é:

- 0; ou
- $n + 1$, onde n é um **número natural**.

A partir dessa definição podemos criar um modelo de função para processar número naturais (que precisam ser decompostos):

```
def fn_para_n(n: int) -> ...:  
    if n == 0:  
        return ...  
    else:  
        return n ... fn_para_n(n - 1)
```

Projete uma função que receba como parâmetro um número natural n e crie um arranjo $[1, 2, \dots, n]$.

```
def lista_n(n: int) -> list[int]:  
    ...  
    Devolve a lista [1, 2, ..., *n*].  
    Requer que n >= 0.  
  
>>> lista_n(0)  
[]  
>>> lista_n(3)  
[1, 2, 3]  
...  
  
if n == 0:  
    return []  
else:  
    return lista_n(n - 1) + [n]
```

Como definir um número natural? Usando autorreferência.

Um **número natural** é:

- 0; ou
- $n + 1$, onde n é um **número natural**.

A partir dessa definição podemos criar um modelo de função para processar número naturais (que precisam ser decompostos):

```
def fn_para_n(n: int) -> ...:
    if n == 0:
        return ...
    else:
        return n ... fn_para_n(n - 1)
```

Projete uma função que receba como parâmetro um número natural n e crie um arranjo $[1, 2, \dots, n]$.

```
def lista_n(n: int) -> list[int]:
    ...
    Devolve a lista [1, 2, ..., *n*].
    Requer que n >= 0.
    >>> lista_n(0)
    []
    >>> lista_n(3)
    [1, 2, 3]
    ...

    if n == 0:
        return []
    else:
        lst = lista_n(n - 1)
        lst.append(n)
        return lst
```

Recursão com arranjos

Podemos usar recursão estrutural com arranjos? Sim e não!

Tentar definir um arranjo usando autorreferência pode ser um pouco confuso... Mas podemos pensar que um arranjo é vazio, ou tem um primeiro elemento e o restante dos elementos.

Dessa forma, podemos definir o seguinte modelo:

```
def fn_para_array(lst: list[int]) -> ...:
    if lst == []:
        return ...
    else:
        return lst[0] ... fn_para_array(lst[1:])
```

Projete uma função que some todos os elementos de um arranjo.

```
def soma(lst: list[int]) -> int:
    ...
    Soma todos os elementos de *lst*.
    ...

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

Qual o problema com essa estratégia?

O *slice* cria um novo arranjo a cada chamada, o que é custoso.

Podemos fazer melhor? Sim!

Recursão com arranjos

Ao invés de “diminuir” o arranjo do início, vamos diminuir do fim usando um “tamanho virtual”.

Junto com o arranjo passamos também um valor n , que representa quantos elementos a partir do início do arranjo devem ser considerados. Na chamada recursiva, passamos o arranjo inalterado e o valor $n - 1$, que representa a diminuição do arranjo. O modelo fica assim:

```
def fn_para_array(lst: list[int], n: int) -> ...:
    if n == 0:
        return ...
    else:
        return lst[n - 1] ... \
            fn_para_array(lst, n - 1)
```

Projete uma função que some todos os elementos de um arranjo.

```
def soma(lst: list[int], n: int) -> int:
    '''
    Soma os primeiros *n* elementos de *lst*.
    Requer que 0 <= n <= len(lst)
    >>> soma([5, 1, 4, 2, 3], 3)
    10
    '''
    if n == 0:
        return 0
    else:
        return lst[n - 1] + soma(lst, n - 1)
```

Não parece melhor que um laço de repetição... Além disso, a função precisa de um argumento extra!

Esse exemplo de função recursiva com arranjo é ilustrativa e de fato não é muito útil.

Na prática, recursividade em arranjo é feita em subarranjos quaisquer, e não em um subarranjo sem o último elemento.

Nesse caso, a função receba como parâmetro além do arranjo um índice de início e outro de fim, que define o subarranjo que vai ser processado.

Note que dessa forma não temos mais recursão estrutural e sim recursão generativa. É preciso determinar uma forma específica para o subarranjo.

Projete uma função recursiva que determine se um arranjo de números é palíndromo, isto é, tem os mesmos elementos quando lido da direita para e esquerda e da esquerda para a direita.

Para esse problema o principal desafio é definir como decompor o problema em subproblema(s) da mesma natureza.

Por exemplo, para o arranjo [4, 1, 3, 3, 1, 4], que subproblema (subarranjo) podemos resolver de forma recursiva que nos ajude a resolver o problema para o arranjo todo?

Se determinamos se [1, 3, 3, 1] (arranjo original sem o primeiro e último) é palíndromo, então podemos utilizar esse fato para determinar se o arranjo original é palíndromo verificando se o primeiro e último elementos são iguais.

Em que situação não precisamos decompor o problema original? Se o subarranjo é vazio ou tem apenas um elemento.

```
def palindromo(lst: list[int], ini: int, fim: int) -> bool:
    ...

    Devolve True se o subarranjo *lst[ini:fim+1]* é palíndromo,
    isto é, o subarranjo tem os mesmos elementos quando visto
    da direita para esquerda e da esquerda para a direita.
    Requer que 0 <= ini < len(lst) e 0 <= fim < len(lst)
    Exemplos
    >>> palindromo([1, 1, 3, 4, 3, 1], 1, 5)
    True
    >>> palindromo([1, 1, 3, 4, 3, 1], 0, 5)
    False
    ...

    assert 0 <= ini < len(lst)
    assert 0 <= fim < len(lst)
    if fim <= ini:
        return True
    else:
        return lst[ini] == lst[fim] and palindromo(lst, ini + 1, fim - 1)
```

```
def palindromo(lst: list[int], ini: int, fim: int) -> bool:
    '''
    Devolve True se o subarranjo *lst[ini:fim+1]* é palíndromo,
    isto é, o subarranjo tem os mesmos elementos quando visto
    da direita para esquerda e da esquerda para a direita.
    Requer que 0 <= ini < len(lst) e 0 <= fim < len(lst)
    Exemplos
    >>> palindromo([1, 1, 3, 4, 3, 1], 1, 5)
    True
    >>> palindromo([1, 1, 3, 4, 3, 1], 0, 5)
    False
    '''
    assert 0 <= ini < len(lst)
    assert 0 <= fim < len(lst)
    return fim <= ini or \
           lst[ini] == lst[fim] and \
           palindromo(lst, ini + 1, fim - 1)
```

Quais os problemas dessa implementação?

- Requer argumentos extras;
- Verifica a validade dos parâmetros em todas as chamadas.

Como podemos melhorar?

Vamos criar um função auxiliar interna que recebe o início e o fim e deixar a função principal recebendo apenas um argumento.

```
def palindromo(lst: list[int]) -> bool:
    ...

    Devolve True se o lst é palíndromo, isto é, tem os mesmos elementos quando
    visto da direita para esquerda e da esquerda para a direita.
    Exemplos
    >>> palindromo([1, 1])
    True
    >>> palindromo([2, 1, 0, 1, 2])
    True
    >>> palindromo([2, 1, 0, 1, 1])
    False
    ...

def _palindromo(lst: list[int], ini: int, fim: int) -> bool:
    return fim <= ini or \
           lst[ini] == lst[fim] and \
           _palindromo(lst, ini + 1, fim - 1)

return _palindromo(lst, 0, len(lst) - 1)
```

Exemplos de execução passo a passo no [PythonTutor](#):

- [Soma](#) dos números naturais menores que um n .
- [Soma](#) dos elementos de uma lista encadeada.
- [Soma](#) dos elementos de um arranjo.
- [Palíndromo](#) de arranjo.