

RECURSÃO

MC102 - Algoritmos e
Programação de
Computadores

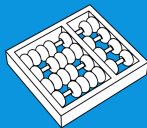
Santiago Valdés Ravelo
[https://ic.unicamp.br/~santiago/
ravelo@unicamp.br](https://ic.unicamp.br/~santiago/ravelo@unicamp.br)

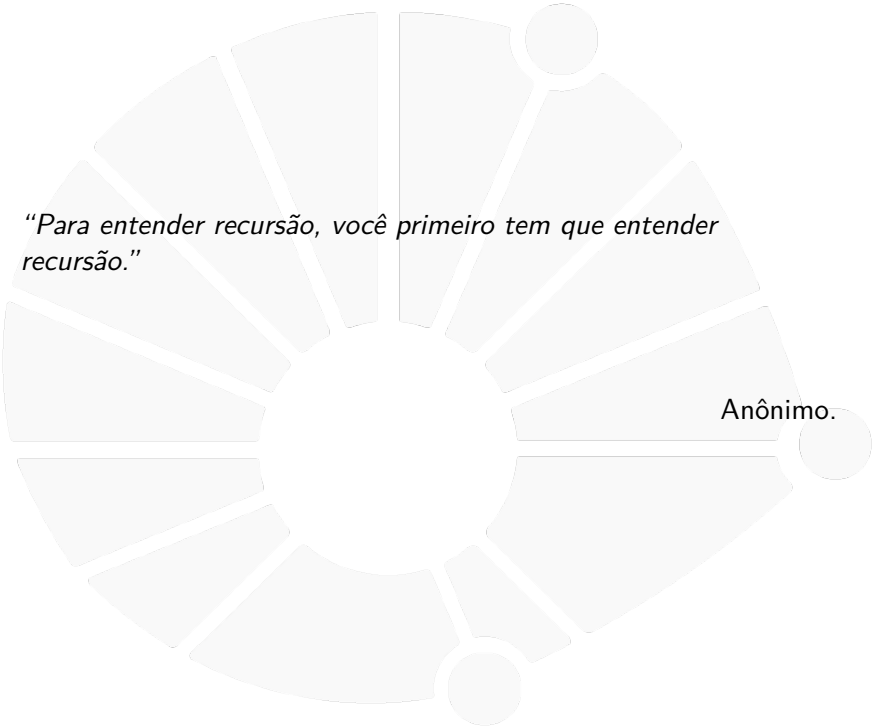
06/25

23



UNICAMP



A circular diagram composed of 12 light gray segments arranged in a ring, separated by white gaps. Three small light gray circles are positioned at the top, right, and bottom of the ring. The text "Para entender recursão, você primeiro tem que entender recursão." is written in the center of the ring.

“Para entender recursão, você primeiro tem que entender recursão.”

Anônimo.



DÚVIDAS DA AULA ANTERIOR

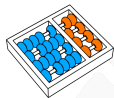


Dúvidas selecionadas

- ▶ No caso em que a lista está desorganizada, qual opção tem melhor desempenho: fazer uma busca linear ou organizar a lista com um dos algoritmos da aula passada e fazer uma busca binária?
- ▶ Como funciona a busca em conjuntos?
- ▶ Quanto ao número de divisões que fazemos em uma lista ordenada (na busca binária, duas divisões) existe um número de divisões mais eficiente levando em conta o tempo que os operadores lógicos consomem? Tem a ver com o comprimento da lista da busca?
- ▶ Python possui alguma biblioteca built-in com uma função pronta para fazer a busca binária sem ter que a criar do zero?
- ▶ Os algoritmos de ordenação funcionariam com números repetidos?
- ▶ Se a lista for bidimensional ou superior, é necessário primeiro linearizá-la para depois aplicar os algoritmos de busca sobre seus elementos?
- ▶ O algoritmo de busca binária funcionaria em uma lista preenchida por um único número, diferente do valor que estou buscando? Por exemplo se estiver buscando 3 em `[1, 1, 1, 1]`.
- ▶ Quando é melhor usar a busca sequencial mesmo com a lista ordenada, em vez da busca binária? Somente em listas pequenas ou quando queremos a primeira ocorrência do elemento?
- ▶ Existem casos em que a busca sequencial se torna mais eficiente do que a binária? Se sim, como é feita essa avaliação?
- ▶ O algoritmo de busca binária funciona apenas em listas ordenadas?
- ▶ Como as estratégias de busca funcionariam em listas com mais de um tipo?
- ▶ Então para desfrutar do tempo logarítmico da busca binária em algum programa, sem o custo de ordenar a lista na hora, o ideal é manter uma lista sempre ordenada? (Isto é, inserindo novos elementos sempre na posição correta ao invés de appendar)
- ▶ Os algoritmos feitos para os métodos de cada linguagem costumam ser parecidos?



INTRODUÇÃO



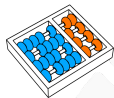
Motivação

- ▶ É necessário solucionar um problema para uma entrada grande.
- ▶ O problema pode ser reduzido para um caso menor.
- ▶ Solucionar o caso menor e a partir dele obter a solução do maior.

O que é recursão?

TÉCNICA em que pelo menos um dos passos de um certo procedimento envolve a **REPETIÇÃO** de todo o procedimento.

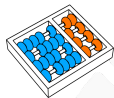
TÉCNICA em que um subprograma (**FUNÇÃO**) chama a si mesmo.



Recursão

Recursão é o conceito de uma função chamar a si mesma para resolver algum problema computacional:

- ▶ Sabemos resolver instâncias pequenas do problema
 - ▶ Chamamos de **CASO BASE**.
- ▶ Sabemos resolver uma instância maior a partir das menores.
 - ▶ Chamamos de **CASO GERAL**.



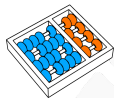
Algoritmos recursivos

Se: a instância é pequena, **então:**

- ▶ Resolva o problema diretamente.

Senão: ▶ Reduza-a a uma (ou várias) instância(s) menor(es) do mesmo problema.

- ▶ Aplique o algoritmo à(s) instância(s) menor(es).
- ▶ Volte à instância original e resolva-a.

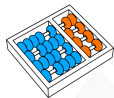


Fim da recursão

IMPORTANTE: toda função recursiva deve **ENCERRAR** a recursão!

Para encerrar a recursão se realiza um teste (**CONDIÇÃO DE PARADA**).

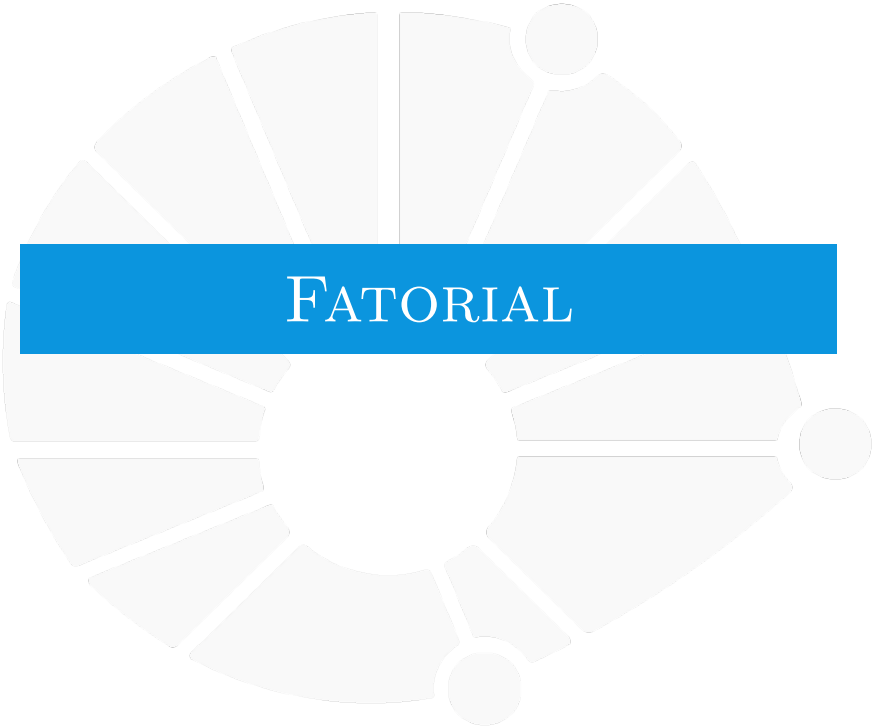
Sem a condição de parada, a recursão se repete eternamente (até dar um erro de memória (**stack**) cheia).



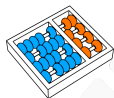
Forma geral

Algoritmo: FUNÇÃO(parâmetros)

- 1 **se** *condição de parada*
 - 2 | resolver **caso base** e/ou finalizar
 - 3 **senão**
 - 4 | FUNÇÃO(parâmetros menores) ▷ chamado recursivo
 - 5 | solucionar o problema para parâmetros
-



FATORIAL

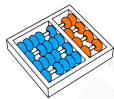


Calculando Fatorial

Por exemplo, para calcular $n! = \prod_{i=1}^n i$:

- ▶ **Base:** Sabemos resolver $0!$, pois $0! = 1$.
- ▶ **Geral:** Sabemos resolver $n!$ a partir de $(n-1)!$ pois:

$$n! = \prod_{i=1}^n i = n \prod_{i=1}^{n-1} i = n(n-1)!$$



Exemplo: Fatorial

Calculando $n!$ recursivamente:

```
1 def fatorial(n):  
2     if n == 0:  
3         return 1  
4     else:  
5         return n * fatorial(n - 1)
```

Mas como isso pode funcionar se a função chama a si mesma?

O Python sabe a linha de código que fez a chamada de função:

- ▶ Isso gera a **pilha de chamadas**.
- ▶ Quando uma função é chamada, ela vai “em cima” da atual.

O Python sabe também qual era o valor das variáveis locais:

- ▶ Ele consegue restaurar esses valores quando voltar.



Exemplo: Fatorial

Calculando $n!$ recursivamente:

```
1 def fatorial(n):  
2     if n == 0:  
3         return 1  
4     else:  
5         return n * fatorial(n - 1)
```

fatorial n: 4

return n * 6

return n * 2

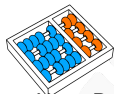
return n * 1

return n * 1

return 1



PROGRESSÃO ARITMÉTICA



Último termo da PA

Uma Progressão Aritmética (PA):

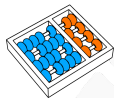
- ▶ É uma sequência de números $(a_1, a_2, \dots, a_n)$,
- ▶ onde existe um número r tal que:
- ▶ $a_i = a_{i-1} + r$, para todo $1 < i \leq n$.

Queremos um algoritmo recursivo para calcular a_n :

- ▶ **Base:** Se $n = 1$, $a_n = a_1$.
- ▶ **Geral:** Se $n > 1$, $a_n = a_{n-1} + r$.
- ▶ Estamos indo de uma instância maior para uma menor.

Solução:

```
1 def termo(a_1, r, n):  
2     if n == 1:  
3         return a_1  
4     else:  
5         return r + termo(a_1, r, n - 1)
```

Exemplo: Progressão aritmética com impressão

E se quisermos imprimir a progressão aritmética?

```
1 def termo(a_1, r, n):  
2     if n == 1:  
3         print(a_1)  
4         return a_1  
5     else:  
6         atual = r + termo(a_1, r, n - 1)  
7         print(atual)  
8         return atual
```

Vamos depurar!

A light gray background featuring a large, faint Fibonacci spiral. The spiral is composed of many thin, light gray rectangular segments that curve inward, creating a circular pattern. Three small, solid light gray circles are positioned at the outer ends of the spiral, one at the top, one on the right, and one at the bottom.

FIBONACCI



Exemplo: Fibonacci

A sequência de Fibonacci é: **1, 1, 2, 3, 5, 8, ...**

- ▶ Começa com **1, 1**.
- ▶ Cada elemento a seguir é a soma dos dois anteriores.

Ou seja,

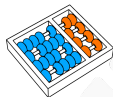
$$f(n) = \begin{cases} f(n-1) + f(n-2), & \text{se } n > 2 \\ 1, & \text{se } n = 1 \text{ ou } n = 2 \end{cases}$$

É o que chamamos na matemática de **recorrência**:

- ▶ Uma função definida recursivamente.
- ▶ **$n!$** é outro exemplo de recorrência.

Note que temos dois casos bases e um caso geral:

- ▶ E que resolvemos uma instância a partir de duas menores.



Exemplo: Fibonnaci

```
1 def fib(n):  
2     if n == 1 or n == 2:  
3         return 1  
4     else:  
5         return fib(n - 1) + fib(n - 2)
```

```
fib          n: 5
```

```
    return 3 + 2
```

```
        return 1 + 1
```

```
            return 1
```

```
                return 1
```

The background features a large, light gray circular design. This design is composed of approximately 16 wedge-shaped segments of varying sizes, arranged in a ring. Three small, solid light gray circles are positioned at the top, bottom, and right edges of the ring, acting as spacers or decorative elements.

EUCLIDES



Exemplo: Algoritmo de Euclides

x é um **divisor** de y se existe um inteiro k tal que $y = k \cdot x$.

- ▶ O máximo divisor comum de a e b , denotado por $\text{mdc}(a, b)$ é o maior inteiro que divide a e b simultaneamente.

Qual o $\text{mdc}(a, 0)$?

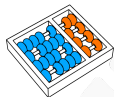
- ▶ Qualquer número x é divisor de 0 , pois $0 = 0 \cdot x$.

Além disso, se x divide a e b , então x divide $a \% b$:

- ▶ $a = q_a \cdot x$ e $b = q_b \cdot x$.
- ▶ $a = k \cdot b + r$ e, portanto, $a \% b = r = a - k \cdot b$.
- ▶ $a \% b = a - k \cdot b = q_a \cdot x - k \cdot q_b \cdot x = (q_a - k \cdot q_b) \cdot x$.

Assim:

$$\text{mdc}(a, b) = \begin{cases} \text{mdc}(b, a \% b), & \text{se } b \neq 0 \\ a, & \text{se } b = 0 \end{cases}$$



Exemplo: Algoritmo de Euclides

$$\text{mdc}(a, b) = \begin{cases} \text{mdc}(b, a \% b), & \text{se } b \neq 0 \\ a, & \text{se } b = 0 \end{cases}$$

Temos um caso base e um caso geral:

- ▶ Para usar recursão, temos que ir do maior para o menor.
 - ▶ E chegarmos na base em algum momento.
- ▶ Note que $a \% b < b$.
 - ▶ Ou seja, estamos sempre diminuindo o segundo termo.

```
1 def mdc(a, b):  
2     if b == 0:  
3         return a  
4     else:  
5         return mdc(b, a % b)
```



Exemplo: Algoritmo de Euclides

```
1 def mdc(a, b):  
2     if b == 0:  
3         return a  
4     else:  
5         return mdc(b, a % b)
```

mdc a: 40 b: 70

return 10

return 10

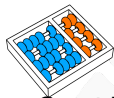
return 10

return 10

return 10



CONSIDERAÇÕES



Um mal exemplo de recursão: $3n + 1$

Considere a seguinte sequência de números:

- ▶ Comece com um número a_0 a sua escolha.
- ▶ Se $a_i = 1$, pare.
- ▶ Se a_i é par, então $a_{i+1} = a_i/2$.
- ▶ Se a_i é ímpar, então $a_{i+1} = 3 \cdot a_i + 1$.

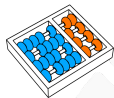
A **Conjectura de Collatz** é que, para qualquer a_0 escolhido, essa sequência sempre termina:

- ▶ Isto é, chegamos em $a_i = 1$.

Podemos fazer um código em Python que gera essa sequência:

- ▶ Mas não temos certeza se a execução terminaria...
- ▶ Afinal, não sabemos se a conjectura é verdadeira...

O problema é que não estamos indo de uma instância maior para uma menor quando a_i é ímpar!



Dicas

- ▶ Você sempre precisa ter pelo menos um caso base!
 - ▶ Senão, você alcança um caso pequeno que não sabe resolver.
- ▶ Você precisa sempre chegar em algum caso base!
 - ▶ Você precisa ir da instância maior para a menor.
 - ▶ Se crescer pode nunca chegar na base!
 - ▶ Se continuar igual, irá ciclar!

Veja esse exemplo:

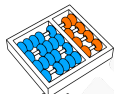
```
1 def fib_defeituoso(n):  
2     if n == 1:  
3         return 1  
4     else:  
5         return fib_defeituoso(n - 1) + fib_defeituoso(n - 2)
```

O que acontece para $n = 2$?

- ▶ Nunca atingimos uma base para `fib_defeituoso(0)`...



RECURSÃO COM LISTAS



Recursão com listas

Até o momento, vimos o uso de recursão para operações matemáticas:

- ▶ Fibonnaci, mdc, PA, etc.

Mas podemos usar recursão para diversas tarefas computacionais:

- ▶ Como, por exemplo, algoritmos que lidam com listas.

Digamos que queremos imprimir uma lista:

- ▶ **Base:** Se a lista for vazia, não temos nada para imprimir.
- ▶ **Geral:** Imprimimos o primeiro elemento e imprimimos o resto recursivamente.

```
1 def imprime(l):  
2     if len(l) == 0:  
3         print()  
4     else:  
5         print(l[0], end=' ')  
6         imprime(l[1:])
```



Exemplo: Imprimindo uma lista

```
1 def imprime(l):
2     if len(l) == 0:
3         print()
4     else:
5         print(l[0], end=' ')
6         imprime(l[1:])
```

Esse código é ruim porque estamos usando slices:

- ▶ Cada slice é uma nova cópia da lista.
- ▶ O que gasta espaço na memória e tempo para a cópia.

Uma versão melhor:

```
1 def imprime_rec(l, n):
2     if n > 0: # a base n == 0 está implícita
3         imprime_rec(l, n - 1)
4         print(l[n - 1], end=' ')
5
6 def imprime(l):
7     imprime_rec(l, len(l))
8     print()
```



Exemplo: Imprimindo uma lista

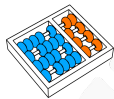
```
1 def imprime_rec(l, n):  
2     if n > 0: # a base n == 0 está implícita  
3         imprime_rec(l, n - 1)  
4         print(l[n - 1], end=' ')  
5  
6 def imprime(l):  
7     imprime_rec(l, len(l))  
8     print()
```

A recursão é no tamanho **n** da sublista a ser impressa:

- ▶ **Base:** Temos **n == 0**.
 - ▶ Ou seja, nada a fazer.
- ▶ **Geral:** Temos **n > 0**:
 - ▶ Imprime recursivamente o começo da lista.
 - ▶ Imprime o último elemento.



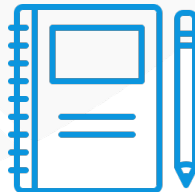
EXERCÍCIOS

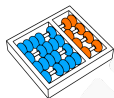


Sobre recursão



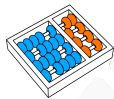
Vamos fazer alguns exercícios?





Exercícios

1. Faça uma função recursiva que calcula a soma dos números naturais menores ou iguais a n .
2. Faça uma função recursiva que calcula a soma dos números naturais ímpares menores ou iguais a n .
3. Faça uma função recursiva que calcula a soma de uma PA com valor inicial a_1 , razão r e n termos.
4. Faça uma função recursiva para contar quantos dígitos um número inteiro positivo tem na representação decimal.
5. Faça uma função recursiva que, dada uma string representando um número inteiro positivo em binário, acha o seu valor em decimal.
6. Faça uma função recursiva que, dado um número inteiro positivo, acha o seu valor em binário (em uma string).



Exercícios com listas

1. Faça uma função recursiva que calcula a soma dos elementos de uma lista.
2. Faça uma função recursiva que encontra o máximo de uma lista.
3. Faça uma função recursiva que busca um elemento em uma lista não ordenada.
4. Faça uma função recursiva que recebe uma lista e devolve uma copia da lista invertida.
5. Faça uma função recursiva que checa se duas listas dadas são iguais.

RECURSÃO

MC102 - Algoritmos e
Programação de
Computadores

Santiago Valdés Ravelo
[https://ic.unicamp.br/~santiago/
ravelo@unicamp.br](https://ic.unicamp.br/~santiago/ravelo@unicamp.br)

06/25

23

