

MC102 — Recursão

Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

Atualizado em: 2023-06-06 09:48

Recursão

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

Recursão

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

- Sabemos resolver instâncias pequenas do problema

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**

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

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**

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**

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

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**

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

- **Base**: Sabemos resolver $0!$, pois $0! = 1$

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**

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)
```

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?

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

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**

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

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

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)
```

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 * fatorial(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)
```

```
fatorial      n: 4
```

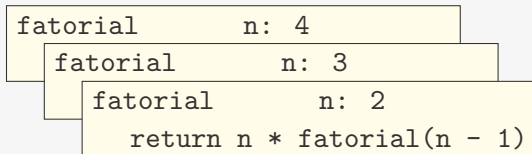
```
fatorial      n: 3
```

```
    return n * fatorial(n - 1)
```

Exemplo: Fatorial

Calculando $n!$ recursivamente:

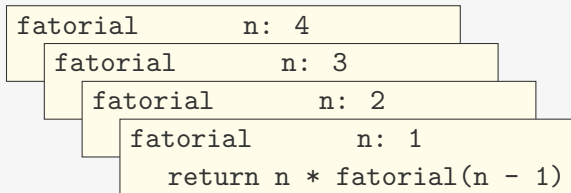
```
1 def fatorial(n):  
2     if n == 0:  
3         return 1  
4     else:  
5         return n * fatorial(n - 1)
```



Exemplo: Fatorial

Calculando $n!$ recursivamente:

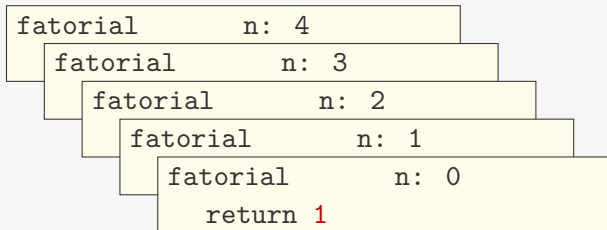
```
1 def fatorial(n):  
2     if n == 0:  
3         return 1  
4     else:  
5         return n * fatorial(n - 1)
```



Exemplo: Fatorial

Calculando $n!$ recursivamente:

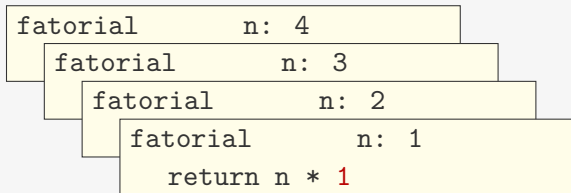
```
1 def fatorial(n):  
2     if n == 0:  
3         return 1  
4     else:  
5         return n * fatorial(n - 1)
```



Exemplo: Fatorial

Calculando $n!$ recursivamente:

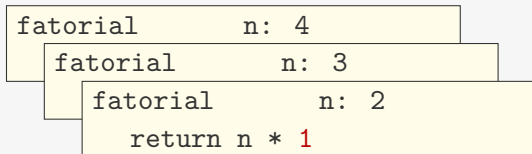
```
1 def fatorial(n):  
2     if n == 0:  
3         return 1  
4     else:  
5         return n * fatorial(n - 1)
```



Exemplo: Fatorial

Calculando $n!$ recursivamente:

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



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

fatorial n: 3

return n * 2

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
```

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
```

Vamos depurar!

Exemplo: Último termo da PA

Uma Progressão Aritmética (PA)

Exemplo: Último termo da PA

Uma Progressão Aritmética (PA)

- é uma sequência de números (a_1, a_2, \dots, a_n)

Exemplo: 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

Exemplo: 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 $1 < i \leq n$

Exemplo: 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

Exemplo: 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 $n = 1$, $a_n = a_1$

Exemplo: 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 $n > 1$, $a_n = a_{n-1} + r$

Exemplo: 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

Exemplo: 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:

Exemplo: 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     '''Calcula o n-ésimo termo de uma PA
3     iniciando em a_1 com razão r'''
4     if n == 1:
5         return a_1
6     else:
7         return r + termo(a_1, r, n - 1)
```

Exemplo: Progressão aritmética com impressão

E se quisermos imprimir a progressão aritmética?

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
```

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!

Exemplo: Fibonnaci

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

Exemplo: Fibonnaci

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

- Começa com 1, 1

Exemplo: Fibonnaci

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

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

Exemplo: Fibonnaci

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

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

Ou seja,

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

Exemplo: Fibonnaci

A sequência de Fibonnaci 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**

Exemplo: Fibonnaci

A sequência de Fibonnaci 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

Exemplo: Fibonnaci

A sequência de Fibonnaci 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

Exemplo: Fibonnaci

A sequência de Fibonnaci 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

Exemplo: Fibonnaci

A sequência de Fibonnaci 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)
```

Exemplo: Fibonacci

```
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 fib(n - 1) + fib(n - 2)
```

Exemplo: Fibonacci

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

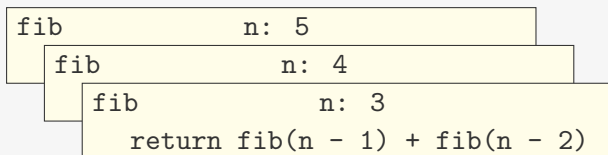
```
fib          n: 5
```

```
fib          n: 4
```

```
return fib(n - 1) + fib(n - 2)
```

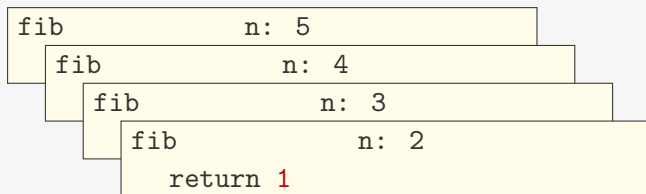
Exemplo: Fibonacci

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



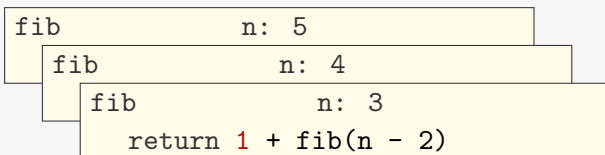
Exemplo: Fibonnaci

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



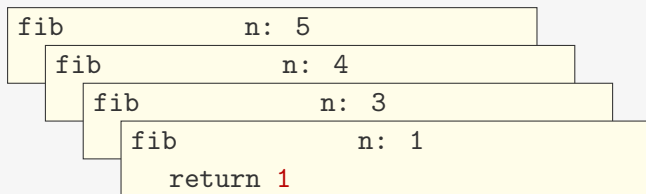
Exemplo: Fibonnaci

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



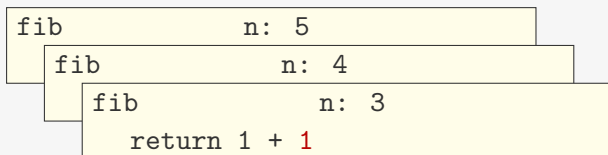
Exemplo: Fibonnaci

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



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)
```



Exemplo: Fibonacci

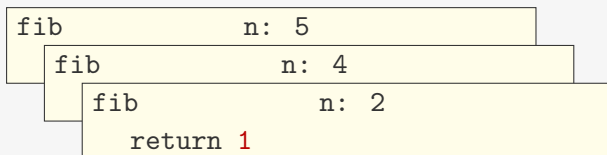
```
1 def fib(n):
2     if n == 1 or n == 2:
3         return 1
4     else:
5         return fib(n - 1) + fib(n - 2)
```

```
fib          n: 5
```

```
fib          n: 4
return 2 + fib(n - 2)
```

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)
```



Exemplo: Fibonacci

```
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
```

```
fib          n: 4
```

```
return 2 + 1
```

Exemplo: Fibonacci

```
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 + fib(n - 2)
```

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)
```

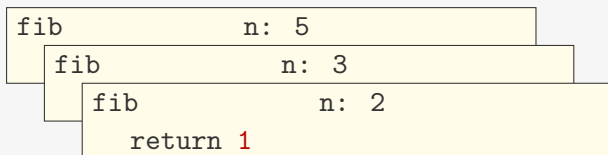
```
fib          n: 5
```

```
fib          n: 3
```

```
    return fib(n - 1) + fib(n - 2)
```

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)
```



Exemplo: Fibonacci

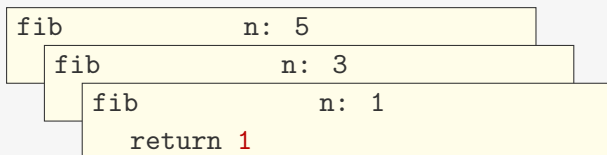
```
1 def fib(n):  
2     if n == 1 or n == 2:  
3         return 1  
4     else:  
5         return fib(n - 1) + fib(n - 2)
```

```
fib          n: 5
```

```
fib          n: 3  
return 1 + fib(n - 2)
```


Exemplo: Fibonacci

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



Exemplo: Fibonacci

```
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
```

```
fib          n: 3
```

```
return 1 + 1
```

Exemplo: Fibonacci

```
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
```

Exemplo: Algoritmo de Euclides

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

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 $mdc(a, b)$ é o maior inteiro que divide a e b simultaneamente

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 $mdc(a, b)$ é o maior inteiro que divide a e b simultaneamente

Qual o $mdc(a, 0)$?

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 $mdc(a, b)$ é o maior inteiro que divide a e b simultaneamente

Qual o $mdc(a, 0)$?

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

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 $mdc(a, b)$ é o maior inteiro que divide a e b simultaneamente

Qual o $mdc(a, 0)$?

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

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

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 $a \bmod b$

- $a = k \cdot b + r$ e, portanto, $a \bmod b = r = a - kb$

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 $mdc(a, b)$ é o maior inteiro que divide a e b simultaneamente

Qual o $mdc(a, 0)$?

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

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

- $a = k \cdot b + r$ e, portanto, $a \bmod b = r = a - kb$
- $a = q_a \cdot x$ e $b = q_b \cdot x$

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 $mdc(a, b)$ é o maior inteiro que divide a e b simultaneamente

Qual o $mdc(a, 0)$?

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

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

- $a = k \cdot b + r$ e, portanto, $a \bmod b = r = a - kb$
- $a = q_a \cdot x$ e $b = q_b \cdot x$
- $a \bmod b = a - k \cdot b = q_a \cdot x - k \cdot q_b \cdot x = (q_a - k \cdot q_b) \cdot x$

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 $mdc(a, b)$ é o maior inteiro que divide a e b simultaneamente

Qual o $mdc(a, 0)$?

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

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

- $a = k \cdot b + r$ e, portanto, $a \bmod b = r = a - kb$
- $a = q_a \cdot x$ e $b = q_b \cdot x$
- $a \bmod b = a - k \cdot b = q_a \cdot x - k \cdot q_b \cdot x = (q_a - k \cdot q_b) \cdot x$

Assim:

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 $mdc(a, b)$ é o maior inteiro que divide a e b simultaneamente

Qual o $mdc(a, 0)$?

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

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

- $a = k \cdot b + r$ e, portanto, $a \bmod b = r = a - kb$
- $a = q_a \cdot x$ e $b = q_b \cdot x$
- $a \bmod b = a - k \cdot b = q_a \cdot x - k \cdot q_b \cdot x = (q_a - k \cdot q_b) \cdot x$

Assim:

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

Exemplo: Algoritmo de Euclides

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

Exemplo: Algoritmo de Euclides

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

Temos um caso base e um caso geral:

Exemplo: Algoritmo de Euclides

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

Temos um caso base e um caso geral:

- Para usar recursão, temos que ir do maior para o menor

Exemplo: Algoritmo de Euclides

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

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

Exemplo: Algoritmo de Euclides

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

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 $a \bmod b < b$

Exemplo: Algoritmo de Euclides

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

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 $a \bmod b < b$
 - Ou seja, estamos sempre diminuindo o segundo termo

Exemplo: Algoritmo de Euclides

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

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 $a \bmod b < b$
 - 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 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
```

```
mdc      a: 70      b: 40
```

```
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

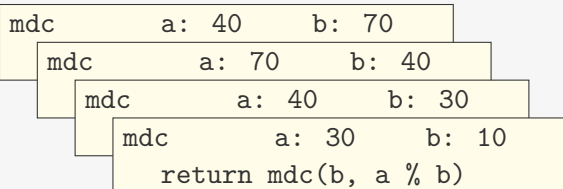
mdc a: 70 b: 40

mdc a: 40 b: 30

return mdc(b, a % b)

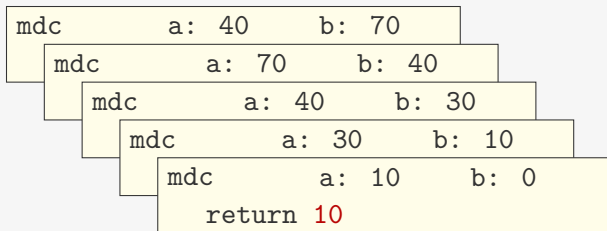
Exemplo: Algoritmo de Euclides

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



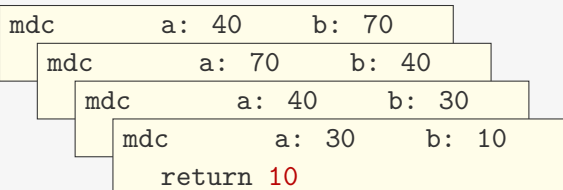
Exemplo: Algoritmo de Euclides

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



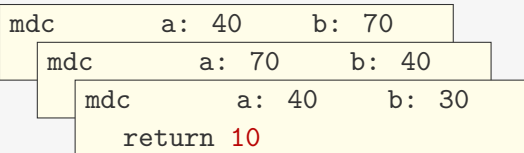
Exemplo: Algoritmo de Euclides

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



Exemplo: Algoritmo de Euclides

```
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
```

```
mdc      a: 70      b: 40
```

```
return 10
```

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
```

Um mal exemplo de recursão: $3n + 1$

Considere a seguinte sequência de números:

Um mal exemplo de recursão: $3n + 1$

Considere a seguinte sequência de números:

- Comece com um número a_0 a sua escolha

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

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 $a_{i+1} = a_i/2$

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 $a_{i+1} = 3 \cdot a_i + 1$

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

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 $a_i = 1$

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

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...

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...

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!

Dicas

- Você sempre precisa ter pelo menos um caso base!
 - Senão, você alcança um caso pequeno que não sabe resolver

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!

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

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!

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!

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:

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)
```


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 $n = 2$?

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)`...

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, dada um número inteiro positivo, acha o seu valor em binário (em uma string).

Recursão com listas

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

Recursão com listas

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

- Fibonnaci, mdc, PA, etc

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

Recursão com listas

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

- Fibonacci, mdc, PA, etc

Mas podemos usar recursão para diversas tarefas computacionais

- Como, por exemplo, algoritmos que lidam com listas

Recursão com listas

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

- Fibonacci, 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:

Recursão com listas

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

- Fibonacci, 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

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

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:])
```

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

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

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

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:

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()
```

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:

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**

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

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**

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

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
 - e imprime o último elemento

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
 - e imprime o último elemento

Vamos depurar!

Exercícios

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.

Busca Binária Iterativa

Relembrando:

```
1 def busca_binaria(l, x):
2     e = 0
3     d = len(l) - 1
4     while e <= d:
5         m = (e + d) // 2
6         if l[m] == x:
7             return m
8         elif l[m] < x:
9             e = m + 1
10        else:
11            d = m - 1
12    return -1
```

Se **x** estiver em **l**, está entre **e** e **d**

- Essa propriedade é mantida no **while** da linha 4

Busca Binária Recursiva

```
1 def busca_binaria_rec(l, x, e, d):
2     if e > d:
3         return -1
4     m = (e + d) // 2
5     if l[m] == x:
6         return m
7     if l[m] < x:
8         return busca_binaria_rec(l, x, m + 1, d)
9     else:
10        return busca_binaria_rec(l, x, e, m - 1)
11
12 def busca_binaria(l, x):
13     return busca_binaria_rec(l, x, 0, len(l) - 1)
```

Busca Binária Recursiva

```
1 def busca_binaria_rec(l, x, e, d):
2     if e > d:
3         return -1
4     m = (e + d) // 2
5     if l[m] == x:
6         return m
7     if l[m] < x:
8         return busca_binaria_rec(l, x, m + 1, d)
9     else:
10        return busca_binaria_rec(l, x, e, m - 1)
11
12 def busca_binaria(l, x):
13     return busca_binaria_rec(l, x, 0, len(l) - 1)
```

A recursão é em e e d e $d - e$ sempre diminuí:

Busca Binária Recursiva

```
1 def busca_binaria_rec(l, x, e, d):
2     if e > d:
3         return -1
4     m = (e + d) // 2
5     if l[m] == x:
6         return m
7     if l[m] < x:
8         return busca_binaria_rec(l, x, m + 1, d)
9     else:
10        return busca_binaria_rec(l, x, e, m - 1)
11
12 def busca_binaria(l, x):
13     return busca_binaria_rec(l, x, 0, len(l) - 1)
```

A recursão é em e e d e $d - e$ sempre diminuí:

- **Base:** Se $d - e < 0$, a faixa que estamos buscando é vazia

Busca Binária Recursiva

```
1 def busca_binaria_rec(l, x, e, d):
2     if e > d:
3         return -1
4     m = (e + d) // 2
5     if l[m] == x:
6         return m
7     if l[m] < x:
8         return busca_binaria_rec(l, x, m + 1, d)
9     else:
10        return busca_binaria_rec(l, x, e, m - 1)
11
12 def busca_binaria(l, x):
13     return busca_binaria_rec(l, x, 0, len(l) - 1)
```

A recursão é em e e d e $d - e$ sempre diminuí:

- Base: Se $d - e < 0$, a faixa que estamos buscando é vazia
- Base: Se $l[m] == x$, encontramos o elemento x

Busca Binária Recursiva

```
1 def busca_binaria_rec(l, x, e, d):
2     if e > d:
3         return -1
4     m = (e + d) // 2
5     if l[m] == x:
6         return m
7     if l[m] < x:
8         return busca_binaria_rec(l, x, m + 1, d)
9     else:
10        return busca_binaria_rec(l, x, e, m - 1)
11
12 def busca_binaria(l, x):
13     return busca_binaria_rec(l, x, 0, len(l) - 1)
```

A recursão é em e e d e $d - e$ sempre diminui:

- **Base:** Se $d - e < 0$, a faixa que estamos buscando é vazia
- **Base:** Se $l[m] == x$, encontramos o elemento x
- **Geral:** Se x está em l entre e e d e $l[m] < x$, então x está em l entre $m + 1$ e d

Busca Binária Recursiva

```
1 def busca_binaria_rec(l, x, e, d):
2     if e > d:
3         return -1
4     m = (e + d) // 2
5     if l[m] == x:
6         return m
7     if l[m] < x:
8         return busca_binaria_rec(l, x, m + 1, d)
9     else:
10        return busca_binaria_rec(l, x, e, m - 1)
11
12 def busca_binaria(l, x):
13     return busca_binaria_rec(l, x, 0, len(l) - 1)
```

A recursão é em e e d e $d - e$ sempre diminui:

- **Base:** Se $d - e < 0$, a faixa que estamos buscando é vazia
- **Base:** Se $l[m] == x$, encontramos o elemento x
- **Geral:** Se x está em l entre e e d e $l[m] < x$, então x está em l entre $m + 1$ e d
- **Geral:** Se x está em l entre e e d e $l[m] > x$, então x está em l entre e e $m - 1$

Enumerando

Recursão pode ser útil também para enumerar:

Enumerando

Recursão pode ser útil também para enumerar:

- Gerar todas as possíveis senhas de acordo com uma regra

Enumerando

Recursão pode ser útil também para enumerar:

- Gerar todas as possíveis senhas de acordo com uma regra
- Gerar todas as permutações de elementos

Enumerando

Recursão pode ser útil também para enumerar:

- Gerar todas as possíveis senhas de acordo com uma regra
- Gerar todas as permutações de elementos
- Gerar todos os subconjuntos de um conjunto

Enumerando

Recursão pode ser útil também para enumerar:

- Gerar todas as possíveis senhas de acordo com uma regra
- Gerar todas as permutações de elementos
- Gerar todos os subconjuntos de um conjunto
- Etc...

Enumerando

Recursão pode ser útil também para enumerar:

- Gerar todas as possíveis senhas de acordo com uma regra
- Gerar todas as permutações de elementos
- Gerar todos os subconjuntos de um conjunto
- Etc...

Para tanto, precisamos construir a informação passo a passo

Enumerando

Recursão pode ser útil também para enumerar:

- Gerar todas as possíveis senhas de acordo com uma regra
- Gerar todas as permutações de elementos
- Gerar todos os subconjuntos de um conjunto
- Etc...

Para tanto, precisamos construir a informação passo a passo

- Armazenando ela em alguma estrutura de dados

Exemplo: Senhas numéricas

Como gerar todas as senhas numéricas de n dígitos?

Exemplo: Senhas numéricas

Como gerar todas as senhas numéricas de n dígitos?

- **Base:** Se $n = 0$, não temos nada a fazer

Exemplo: Senhas numéricas

Como gerar todas as senhas numéricas de n dígitos?

- **Base:** Se $n = 0$, não temos nada a fazer
- **Geral:** Para cada $i \in \{0, \dots, 9\}$, gere todas as senhas de $n - 1$ dígitos colocando i na frente

Exemplo: Senhas numéricas

Como gerar todas as senhas numéricas de n dígitos?

- **Base:** Se $n = 0$, não temos nada a fazer
- **Geral:** Para cada $i \in \{0, \dots, 9\}$, gere todas as senhas de $n - 1$ dígitos colocando i na frente

```
1 def senhas_rec(mem, n):
2     if len(mem) == n: # temos n digitos
3         print(''.join(mem), end=' ')
4     else:
5         for i in range(10):
6             mem.append(str(i)) # coloca o dígito na memória
7             senhas_rec(l, n)
8             mem.pop() # retira o dígito da memória
9
10 def senhas(n):
11     mem = []
12     senhas_rec(mem, n)
```

Exemplo: Senhas numéricas

Como gerar todas as senhas numéricas de n dígitos?

- **Base:** Se $n = 0$, não temos nada a fazer
- **Geral:** Para cada $i \in \{0, \dots, 9\}$, gere todas as senhas de $n - 1$ dígitos colocando i na frente

```
1 def senhas_rec(mem, n):
2     if len(mem) == n: # temos n digitos
3         print(''.join(mem), end=' ')
4     else:
5         for i in range(10):
6             mem.append(str(i)) # coloca o dígito na memória
7             senhas_rec(l, n)
8             mem.pop() # retira o dígito da memória
9
10 def senhas(n):
11     mem = []
12     senhas_rec(mem, n)
```

Vamos depurar!

Exercícios

1. Faça uma função recursiva que, dado um inteiro positivo n , imprime todas as permutações de elementos de 1 a n .
2. Faça uma função recursiva que dada uma string s , diz se s é palíndromo ou não. Uma string s é um palíndromo se a reversão de s é a própria s (ex: ' ana ' e ' $arara$ ').
3. Faça uma função recursiva que verifica se duas listas são iguais. Porém, os elementos podem também ser listas e você não deve usar a comparação de listas pronta do Python.
4. Faça uma função recursiva que, dada uma lista l e um inteiro positivo r , imprime todas as combinações de r elementos de l .