

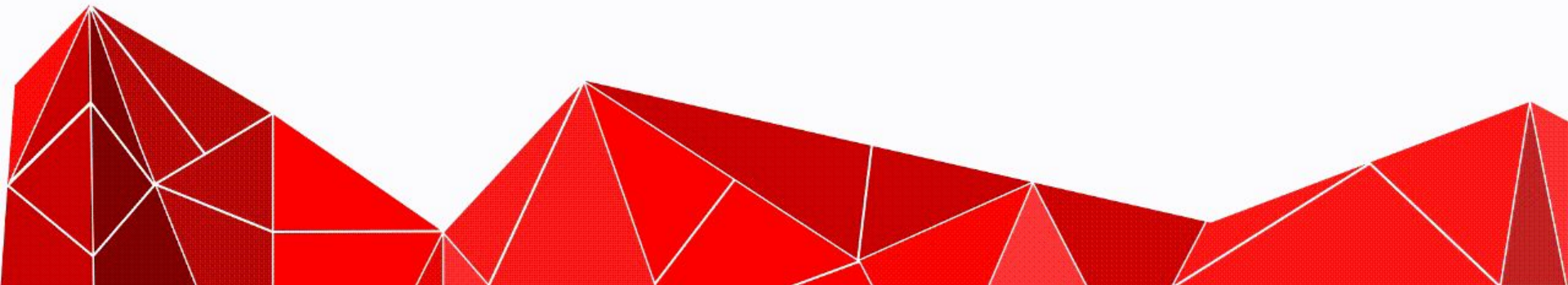


# **CODING**

## **FUNÇÕES - PARTE 03**

**Press -> to continue**

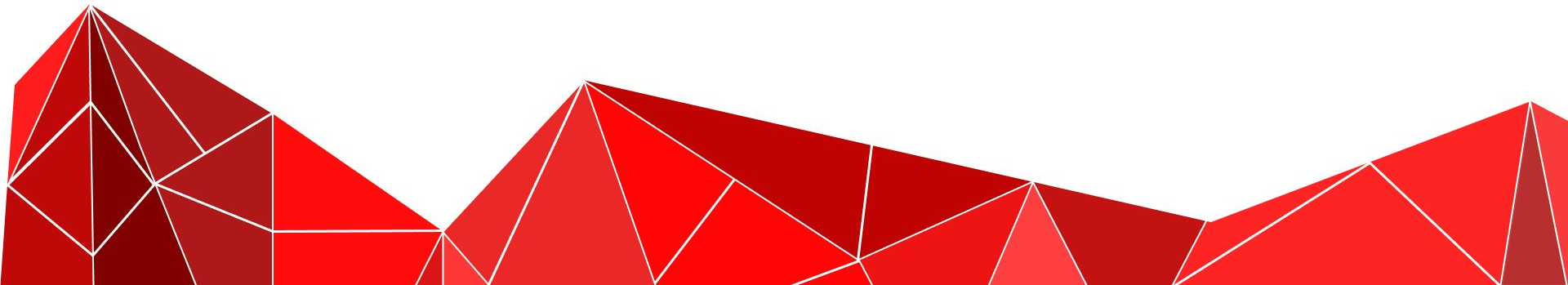
**Powered by PET computação UFC**



O que vamos ver hoje ?

Introdução

Estatégia recursiva





# Introdução



# Introdução

- O processo no qual a função chama a si mesma direta ou indiretamente é chamado de recursão e a função correspondente é chamada de função recursiva.
- Usando algoritmo recursivo, certos problemas podem ser resolvidos com bastante facilidade.



# Introdução

- Assim, função recursiva é uma função que é definida em termos de si mesma.
- Recursividade é o mecanismo básico para repetições nas linguagens funcionais
- De forma bem resumida e coloquial:
  - Uma função recursiva é uma função que chama ela mesma até que um certo problema maior seja dividido em partes menores, até quando não possa mais dividir.

# Introdução

- Vamos considerar um problema que um programador tem para determinar a soma dos primeiros  $n$  números naturais, existem várias maneiras de fazer isso, mas a abordagem mais simples é simplesmente adicionar os números começando de 1 a  $n$ . Então a função simplesmente se parece com:
  - $f(n) = 1 + 2 + 3 + \dots + n$
- Mas podemos resolver o problema de outra maneira:
  - $f(n) = n + f(n-1)$  enquanto  $n > 1$
  - Dessa forma estamos usando recursividade para resolver o problema.

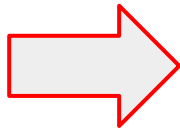
# Introdução

- Existe uma diferença simples entre a abordagem (1) e a abordagem (2):
- É que na abordagem (2) a própria função “  $f()$  ” está sendo chamada dentro da função, então esse fenômeno é chamado de recursão e a função que contém recursão é chamada de função recursiva.
- No final esta é uma ótima ferramenta na mão dos programadores para codificar alguns problemas de maneira muito mais fácil e eficiente.

# Introdução

- Vamos ver um exemplo de maneira mais detalhada:
- Um problema simples que você já sabe como resolver sem o uso de recursão.
- Suponha que você deseja calcular a soma de uma lista de números, tais como: [1,3,5,7,9]. A função usa uma variável acumuladora (theSum) para calcular o total de todos os números da lista iniciando com 0 e somando cada número da lista.

```
1 def listsum(numList):  
2     theSum = 0  
3     for i in numList:  
4         theSum = theSum + i  
5     return theSum  
6  
7 print(listsum([1,3,5,7,9]))
```



```
rafael@rafael-Lenovo-ideapad-330-151KB: ~/Documents  
>>> exec(open('firstProgram.py').read())  
25  
>>> []
```



# Introdução

- Imaginem por um minuto que vocês não tem laços while ou for. Como vocês iriam calcular a soma ?
- Se você fosse um matemático poderia começar recordando que a adição é uma função definida para dois parâmetros, um par de números.
- Para redefinir o problema da adição de uma lista para a adição de pares de números, podemos reescrever a lista como uma expressão totalmente entre parênteses. Tal expressão poderia ser algo como:
  - $((((1+3)+5)+7)+9)$
  - Poderíamos colocar na ordem inversa:
  - $(1+(3+(5+(7+9))))$

# Introdução

- Observe que o par de parênteses mais interno,  $(7+9)$ , é um problema que podemos resolver sem um laço ou qualquer construção especial. Na verdade, pode-se utilizar a seguinte sequência de simplificações para calcular uma soma final:
  - $\text{total} = (1+(3+(5+(7+9))))$   
 $\text{total} = (1+(3+(5+16)))$   
 $\text{total} = (1+(3+21))$   
 $\text{total} = (1+24)$   
 $\text{total} = 25$
- Como podemos usar essa ideia e transformá-la em um programa Python?
- Em primeiro lugar, vamos reformular o problema da soma em termos de listas de Python.



# Introdução

- Poderíamos dizer que a soma da lista numList é a soma do primeiro elemento da lista (numList [0]), com a soma dos números no resto da lista (numList [1:])
- Podemos escrever a função como:
  - `listSum(numList)=first(numList)+listSum(rest(numList))`
- Nesta equação `first(numList)` retorna o primeiro elemento da lista e `rest(numList)` retorna a lista com tudo menos o primeiro elemento.
- Vamos ver a implementação dessa estratégia

# Introdução

- Implementação em python

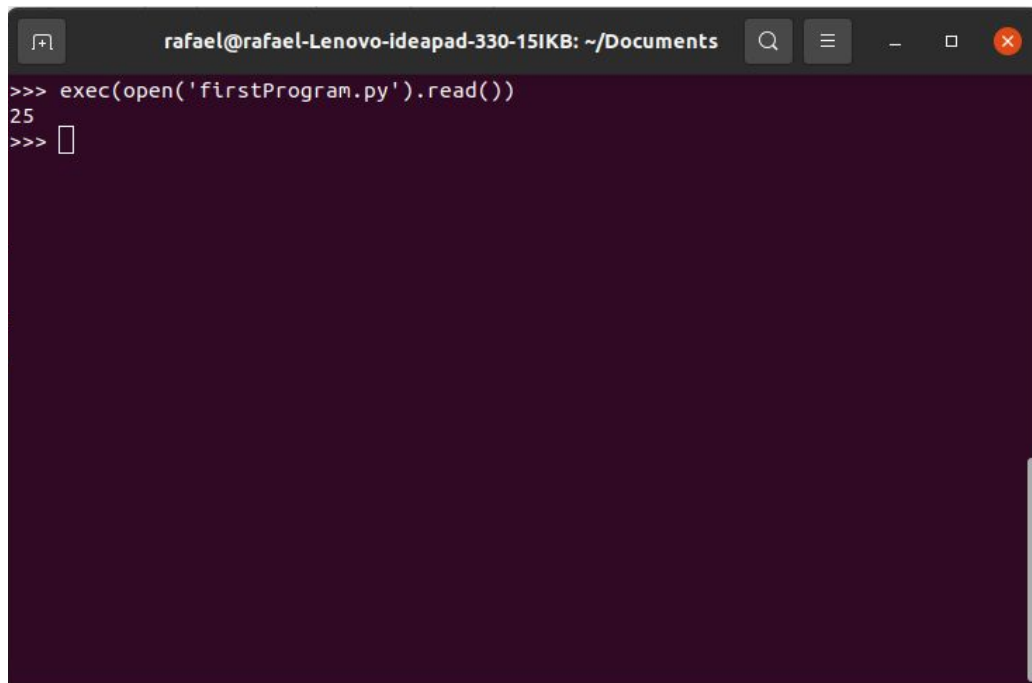


A screenshot of a code editor window titled "firstProgram.py" with the path "~/Documents". The window contains a Python script for a recursive function named "listsum". The code is as follows:

```
1 def listsum(numList):
2     if len(numList) == 1:
3         return numList[0]
4     else:
5         return numList[0] + listsum(numList[1:])
6
7 print(listsum([1,3,5,7,9]))
```

# Introdução

- O resultado é o mesmo:



```
rafael@rafael-Lenovo-ideapad-330-151KB: ~/Documents
>>> exec(open('firstProgram.py').read())
25
>>> 
```

# Introdução

- Existem algumas ideias-chave nesse programa para se estudar.
- Em primeiro lugar, na linha 2 estamos verificando se a lista possui apenas um elemento.

```
2     if len(numList) == 1:  
3         return numList[0]
```

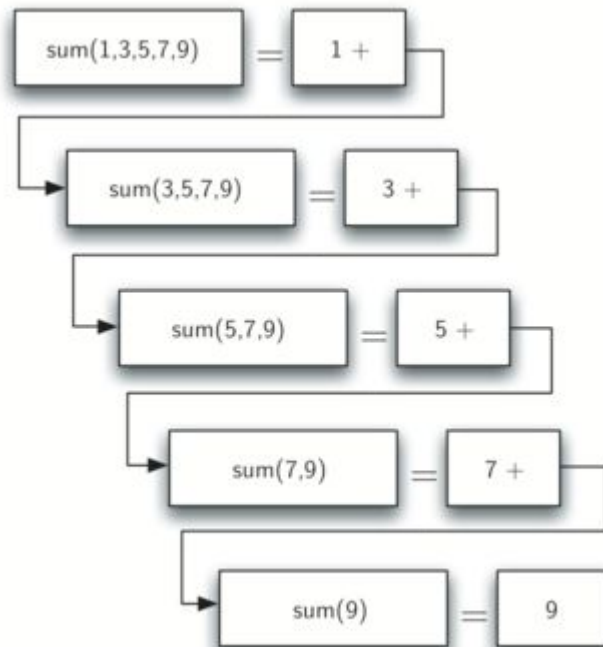
- Esse teste é fundamental e é a nossa cláusula de escape da função. A soma de uma lista de comprimento 1 é trivial; ela é o número na lista.
- Em segundo lugar, na linha 5, nossa função chama a si mesma!

```
4     else:  
5         return numList[0] + listsum(numList[1:])
```

- Vamos ver uma representação mais visual de recursividade

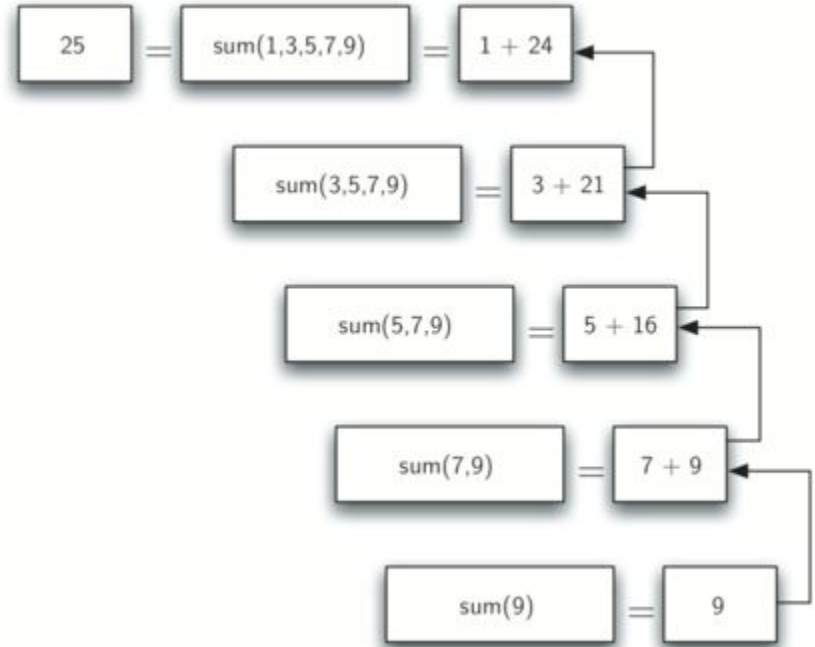
# Introdução

- A Figura mostra a série de chamadas recursivas necessária para somar a lista [1,3,5,7,9]. Você deve pensar nessa série de chamadas como uma série de simplificações.



# Introdução

- Quando chegarmos ao ponto em que o problema é tão simples quanto ele pode ficar, começamos a juntar as soluções de cada um dos pequenos problemas até que o problema inicial seja resolvido.







# **Estatégia Recursiva**



# Estatégia Recursiva

- As três leis da recursão
  - Um algoritmo recursivo deve ter um caso básico
  - Um algoritmo recursivo deve mudar o seu estado e se aproximar do caso básico.
  - Um algoritmo recursivo deve chamar a si mesmo, recursivamente.
- Vamos olhar para cada uma dessas leis com mais detalhes e ver como elas foram utilizadas no algoritmo listsum.
- Em primeiro lugar, um caso básico é a condição que permite que o algoritmo recursivo pare de recorrer.
  - No caso do algoritmo listsum o caso básico é uma lista de comprimento 1.



# Estatégia Recursiva

- Para obedecer a segunda lei, temos de arranjar uma mudança de estado que leve o algoritmo para o caso básico.
- No algoritmo listsum nossa estrutura de dados primária é uma lista, por isso temos de concentrar o nosso esforço de mudança de estado na lista.
- Como o caso básico é uma lista de comprimento 1, uma progressão natural para o caso básico é encurtar a lista.
- Este é exatamente o que acontece na linha 5 do Programa 2 quando chamamos listsum com uma lista mais curta.

# Estatégia Recursiva

- A última lei é que o algoritmo deve chamar a si mesmo.
- Isso ocorre quando o problema ainda não foi dividido ao máximo em problemas menores, ou seja, ainda não chegou no caso básico.

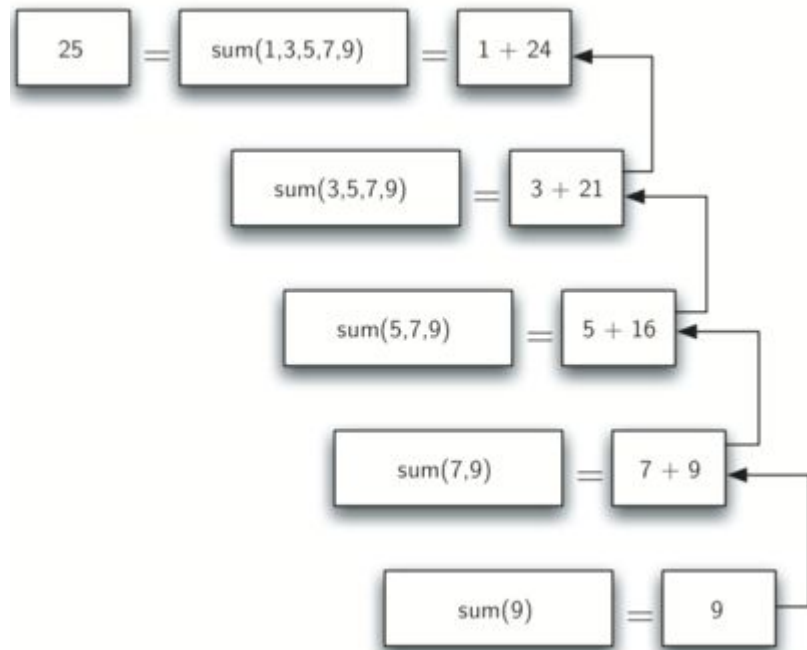


# Uso de memória



# Uso de memória

- Isso é uma propriedade da recursividade que se a função não for bem elaborada, pode consumir muita memória do pc.
- Vejamos o nosso exemplo:
- A cada vez que chamamos a nossa função alocamos um espaço na memória para que ela possa funcionar
- Por isso precisamos elaborar bem nossa função, pois se não ela pode gerar um stack overflow



# Uso de memória

- Overflow vem do inglês e significa “transbordar”
  - Assim, podemos entender stack overflow, como a pilha da memória transbordando, ficar lotada.
- Para evitar o estouro da pilha, as chamadas recursivas precisam parar. Daí a importância da condição de parada da função.
- Apesar de funções recursivas serem mais elegantes e mais fáceis de implementar, elas costumam ser menos eficientes que suas correspondentes iterativas, por causa do overhead de empilhar e desempilhar chamadas de funções.
- Não é tão simples decidir quando usar uma solução recursiva para um problema, mas você vai perceber que alguns problemas são muito mais fáceis e intuitivos de serem resolvidos recursivamente. É nesses casos que a recursão vale a pena.



# Vamos praticar

- Escreva uma função que recebe calcule a potência de uma base  $m$  por um expoente  $n$  sem usar a função `pow` do python. A função recebe a base e o expoente como entradas.





# Vamos praticar

- Defina a função `prod_lista` que recebe como argumento uma lista de inteiros e devolve o produto dos seus elementos..



# Vamos praticar

- Defina a função `prim_alg` que recebe como argumento um número natural e devolve o primeiro algarismo (o mais significativo) na representação decimal de  $n$ .



# Vamos praticar

- Escreva uma função que recebe como argumento um número não negativo  $n$  e devolve True, se o número for primo. Caso contrário, devolve False.

# Vamos praticar

- Defina a função `div` que recebe como argumentos dois números naturais  $m$  e  $n$  e devolve o resultado da divisão inteira de  $m$  por  $n$ . Neste exercício, você não pode recorrer às operações aritméticas de multiplicação, divisão e resto da divisão inteira.



# Vamos praticar

- Faça uma função recursiva que conte as ocorrências de um algarismo em um número. Usando apenas lógica, sem usar strings.

**“Nos tempos mais difíceis,  
esperança é algo que você dá  
a si mesmo. Esse é o  
significado de força  
interior”**

**-Iroh**



The background of the slide features a red grid pattern. The grid consists of horizontal and vertical lines, with additional diagonal lines forming a perspective effect that creates a sense of depth, resembling a floor or ceiling receding into the distance.

**Obrigado!!**  
**Ainda com dúvida ?**  
**Entre em contato**