Recursão ... novamente!?. Soluções aos exercícios

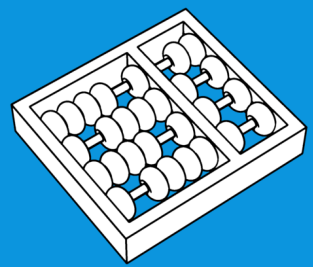
MC102 - Algoritmos e Programação de Computadores

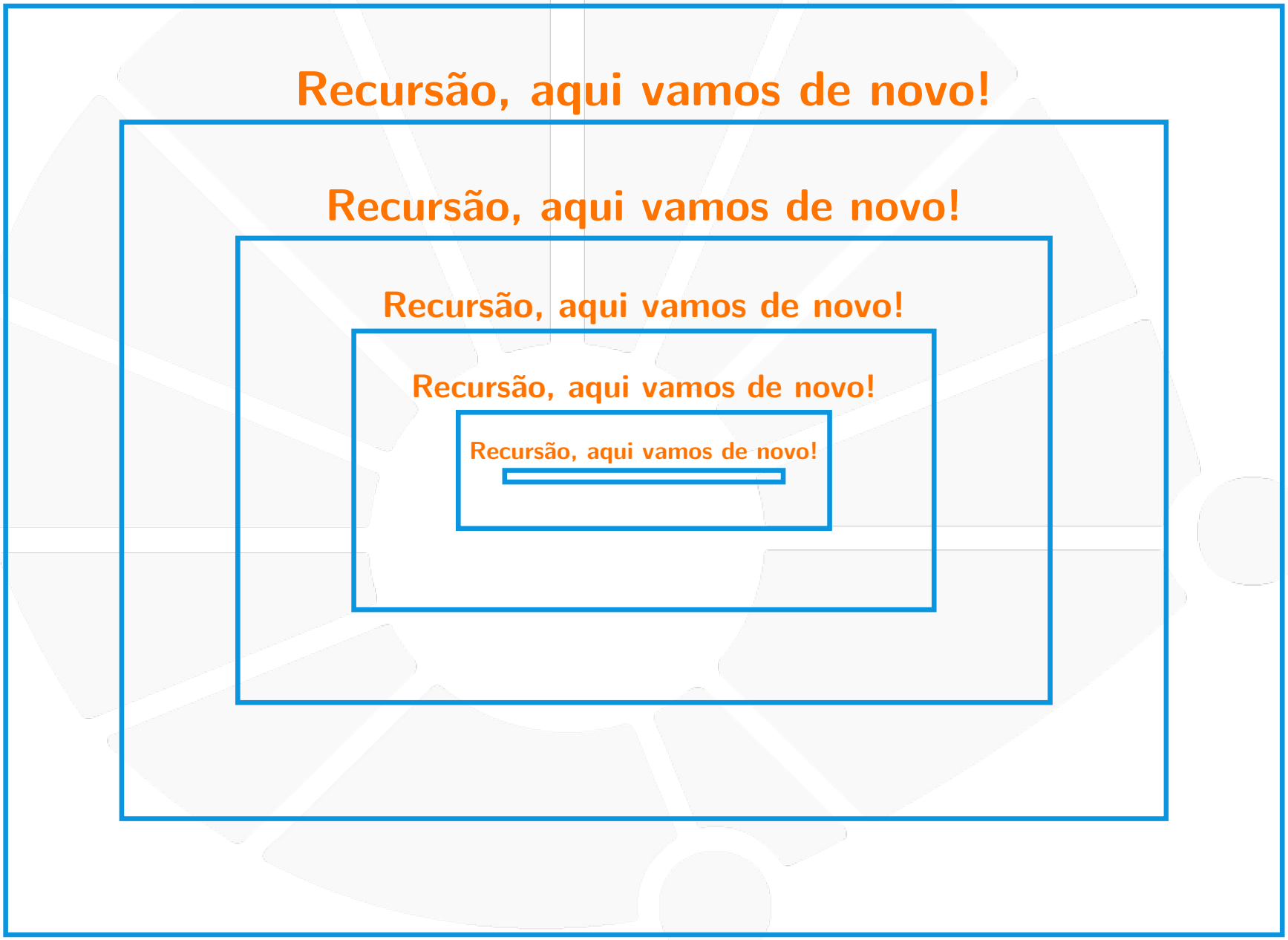
06/25

25

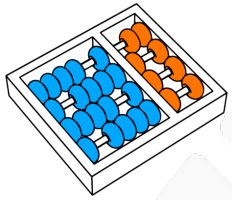
Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

UNICAMP



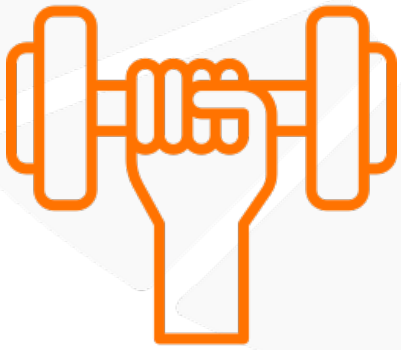

Recursão, aqui vamos de novo!
Recursão, aqui vamos de novo!
Recursão, aqui vamos de novo!
Recursão, aqui vamos de novo!
Recursão, aqui vamos de novo!
Recursão, aqui vamos de novo!
Recursão, aqui vamos de novo!

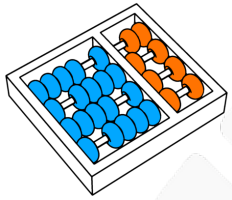
Exercícios



Recursão

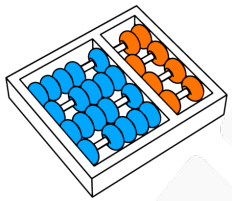
Soluções para os exercícios!





Exercícios

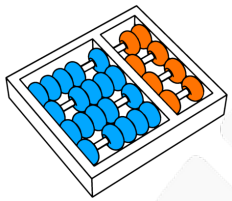
1. Faça uma função recursiva que, dado um inteiro positivo n, imprime todas as permutações de elementos de 1 a n.
2. Faça uma função recursiva que dado um inteiro n, imprime todas as combinações com n pares de parênteses balanceados.
3. Faça uma função recursiva que, dada uma lista l e um inteiro positivo r, imprime todas as combinações de r elementos de l.
4. Dado um jogo de Sudoku, encontre uma solução se ela existir. Considere que as casas não preenchidas tem o valor **0**.
5. Dado um inteiro **n**, determine se é possível colocar **n** rainhas (do xadrez) em um tabuleiro **n**×**n**, sem que elas se ameacem.



Soluções

Permutações de elementos de 1 a *n*:

```
1  def permutacoes_rec(mem, n, usados):
2      if len(mem) == n:  # temos uma permutação completa
3          print(' '.join(mem))
4      else:
5          for i in range(1, n + 1):
6              if not usados[i]: # adiciona se o número não foi usado
7                  mem.append(str(i))  # coloca o número no final
8                  usados[i] = True # indica que agora está usado
9                  permutacoes_rec(mem, n, usados)
10                 mem.pop()  # retira o número do final do prefixo
11                 usados[i] = False # o número não está mais na memória
12
13 def permutacoes(n):
14     mem = []
15     usados = [False] * n # inicialmente nenhum número é usado
16     permutacoes_rec(mem, n, usados)
```

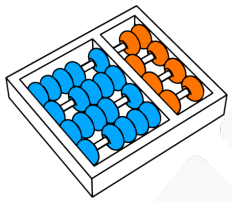


Soluções. Continuação

Parênteses balanceados:

```
1  def parenteses_rec(mem, n, balanco):
2      if len(mem) == 2 * n:  # temos uma combinação de n pares balanceados
3          print(' '.join(mem))
4      else:
5          if balanco + len(mem) < 2 * n: # se houver espaço para fechar parênteses abertos
6              mem.append('(') # é possível adicionar um aberto
7              parenteses_rec(mem, n, balanco + 1) # há mais um aberto para ser fechado
8              mem.pop() # removemos o parêntese
9          if balanco > 0: # se houver abertos sem fechar
10             mem.append(')') # é possível adicionar um fechado
11             parenteses_rec(mem, n, balanco - 1) # há menos um aberto sem fechar
12             mem.pop() # removemos o parêntese
13
14 def parenteses(n):
15     mem = []
16     parenteses_rec(mem, n, 0)
```

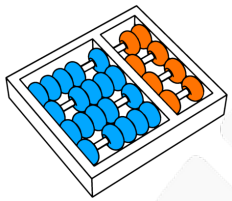
A variável balanco é usada para saber quantos abertos (sem fechar) há. Assim, por cada aberto que adicionamos no prefixo gerado, aumentamos o valor e por cada fechado o diminuímos. Só podemos adicionar abertos, se houver espaço para fechar todos os abertos que foram adicionados sem seu par fechado. Só podemos adicionar fechados se houver antes algum aberto sem par.



Soluções. Continuação

Combinações de *r* elementos em lista:

```
1  def combinacoes_rec(l, r, mem, usados):
2      if len(mem) == r:  # temos uma combinação de r elementos
3          print(' '.join(mem))
4      else:
5          for i in range(len(l)):
6              if not usados[i]: # adiciona se o elemento não foi usado
7                  mem.append(str(l[i]))  # coloca o número no final
8                  usados[i] = True # indica que agora está usado
9                  combinacoes_rec(l, r, mem, usados)
10                 mem.pop()  # retira o elemento do final do prefixo
11                 usados[i] = False # o elemento da posição i não está mais no prefixo
12
13 def combinacoes(l, r):
14     mem = []
15     usados = [False] * len(l) # inicialmente nenhum elemento é usado
16     combinacoes_rec(l, r, mem, usados)
```

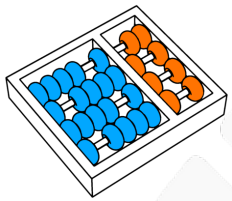


Soluções. Continuação

Sudoku:

```
1
2  def sudoku_rec(matriz, l, c):
3      if l == 9: # se chegamos na linha 9, então preenchemos o Sudoku (última posição é [8][8])
4          for linha in matriz:
5              print(*linha)
6          return True
7      prox_l, prox_c = proxima_pos(l, c)
8      if matriz[l][c] > 0: # a matriz veio preenchida nessa posição
9          return sudoku_rec(matriz, prox_l, prox_c)
10     for v in range(1, 10): # tentamos colocar cada possível valor
11         if pode_inserir(matriz, l, c, v):
12             matriz[l][c] = v
13             if sudoku_rec(matriz, prox_l, prox_c):
14                 return True
15             matriz[l][c] = 0 # Se não deu certo, colocamos 0
16     return False
17
18 def sudoku(matriz):
19     return sudoku_rec(matriz, 0, 0)
```

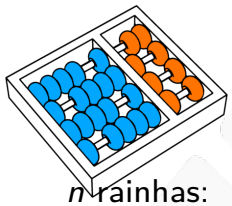
A solução proposta usa duas funções auxiliares: "proxima_pos", que indica a próxima posição para analisar (a proposta é varrer linha por linha); "pode_inserir" que verifica se um valo "v" pode ser inserido na posição "l,c" seguindo as regras do Sudoku (não pode estar na linha "l", na coluna "c" ou no bloque de tamanho 3×3 que contém essa posição). Essas funções são dadas no próximo slide.



Soluções. Continuação

Funções auxiliares para Sudoku:

```
 1  def pode_inserir(matriz, l, c, v):
 2      for i in range(10):
 3          if m[l][i] == v or matriz[i][c] == v: # se v está na linha l ou na coluna c, não pode
 4              return False
 5      # pegamos o início do bloco 3x3 em que queremos inserir (linha e coluna)
 6      l_bloco = 3 * (l // 3)
 7      c_bloco = 3 * (c // 3)
 8      for i in range(l_bloco, l_bloco + 3):
 9          for j in range(c_bloco, c_bloco + 3):
10              if matriz[i][j] == v: # se v aparece no bloco, não pode
11                  return False
12      return True
13
14  def proxima_pos(l, c):
15      if c < 8: # se não chegou no fim da linha, vai para a próxima casa da linha
16          return l, c + 1
17      else: # senão, vai para o começo da próxima linha
18          return l + 1, 0
19
```



Soluções. Continuação

n rainhas:

```
1  def pode_inserir(matriz, l, c):
2      # só verifica linhas anteriores, pois a atual e as próximas não foram preenchidas
3      for i in range(l):
4          if matriz[i][c] == 1: # há rainha na coluna?
5              return False
6          if c + i < len(matriz) and matriz[l - i][c + i] == 1: # na diagonal direita?
7              return False
8          if c - i < len(matriz) and matriz[l - i][c - i] == 1: # na diagonal esquerda?
9              return False
10     return True
11
12 def rainhas_rec(matriz, l):
13     if l == len(matriz): # colocamos len(n) rainhas
14         for linha in matriz:
15             print(*linha)
16         return True
17     for c in range(len(matriz)): # tentamos colocar a rainha em cada posição da linha
18         if pode_inserir(matriz, l, c):
19             matriz[l][c] = 1
20             if rainhas_rec(matriz, l + 1):
21                 return True
22             matriz[l][c] = 0 # Se não deu certo, colocamos 0
23     return False
24
25 def rainhas(n):
26     matriz = [[0] * n for i in range(n)]
27     return rainhas_rec(matriz, 0)
```

Recursão ... novamente!?. Soluções aos exercícios

MC102 - Algoritmos e Programação de Computadores

06/25

25

Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

UNICAMP