

Recursão

MC102

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)
```

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
```

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
3     PA
4     iniciando em a_1 com razão r'''
5     if n == 1:
6         return a_1
7     else:
8         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!

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)
```

```
fib      n: 5  
return 3 + 2
```

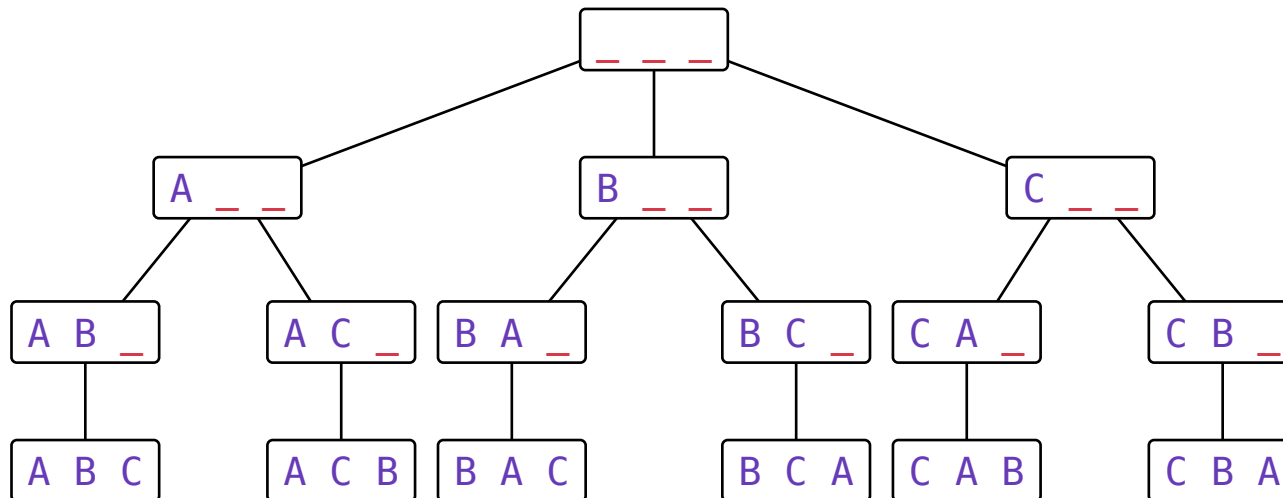
Backtracking

O back e o tracking

Força Bruta: tentar todas as soluções possíveis, selecionando as viáveis.

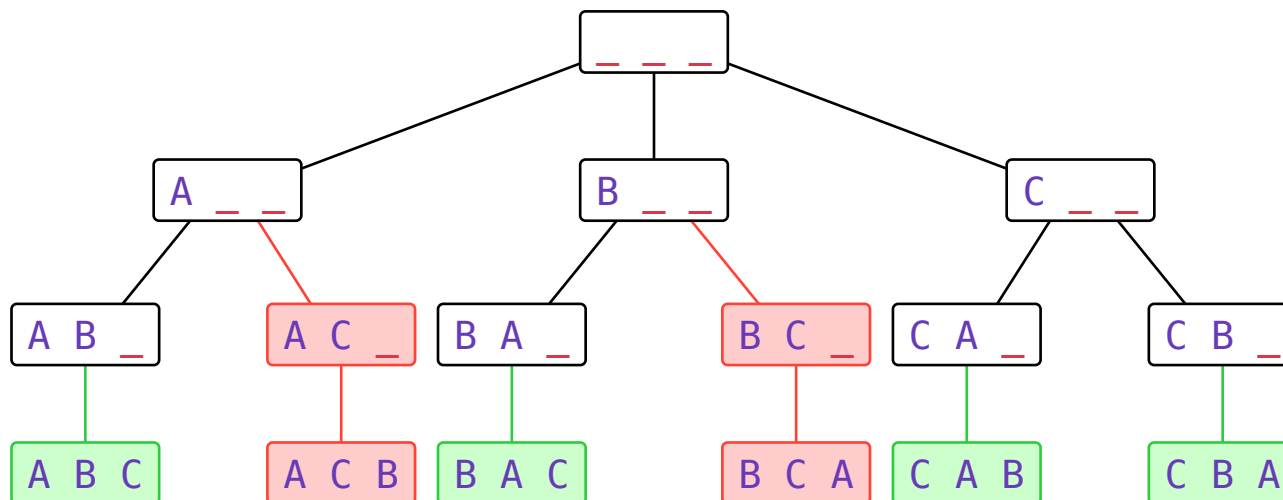
Exemplo: Três estudantes: Alice, Bruno, Carlos que precisam se sentar em 3 cadeiras. Quantas soluções são possíveis? $n! = 3! = 6$

Árvore de Decisão



O back e o tracking

Restrições: Carlos não pode se sentar entre Alice e Bruno.



Ideia central do backtracking

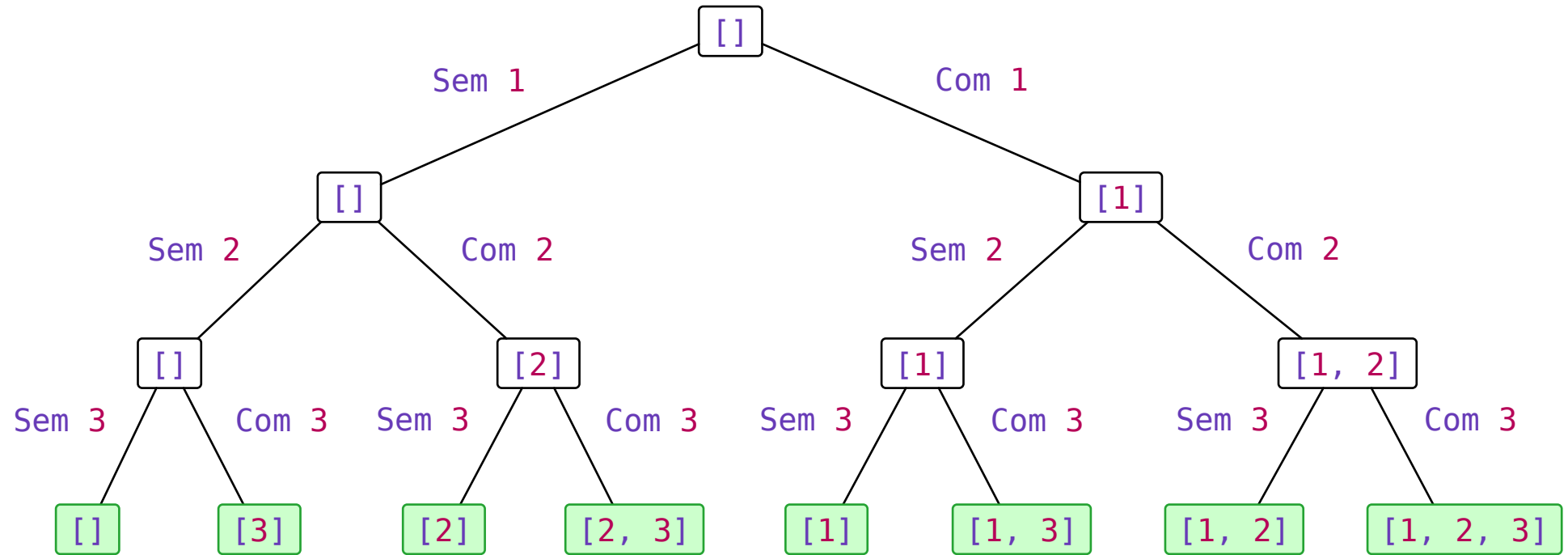
- Gerar soluções parciais
- Verificar se a solução parcial é viável
- Se for, continuar gerando a partir dela
- Se não for, descartar e voltar para a etapa anterior (backtrack)
- Repetir até encontrar a solução completa ou esgotar as possibilidades

Subconjuntos

</> Code

Dado uma lista de elementos, retorne todos os subconjuntos desta lista.

Exemplo para a lista [1, 2, 3]:



Subconjuntos

```
1 solutions = []
2 def subset(lista, index=0, path=[]):
3     if index == len(lista):
4         solutions.append(path)
5         return
6
7     # Não incluir o elemento atual
8     subset(lista, index + 1, path)
9
10    # Incluir o elemento atual
11    subset(lista, index + 1, path + [lista[index]])
```

Ideia central do backtracking

Todos os problemas de backtracking podem ser resolvidos em 4 etapas:

- Caso base
- Escolhas (recursão)
- Restrições (validação)
- Backtrack

Exemplo do labirinto

- Imagine um labirinto representado por uma matriz, onde 0s são caminhos livres
- Queremos encontrar um caminho do ponto A ao ponto B
- Podemos usar backtracking para explorar o labirinto:
 - Começamos no ponto A
 - Em cada passo, tentamos mover para uma direção (cima, baixo, esquerda, direita)
 - Se o movimento for válido (dentro dos limites e sem paredes), continuamos a partir dessa nova posição
 - Se chegarmos ao ponto B, encontramos uma solução
 - Se não houver mais movimentos válidos, voltamos para a posição anterior e tentamos outra direção (backtrack)
 - Repetimos esse processo até encontrar o caminho ou esgotar as possibilidades

Exemplo n rainhas

</> Code

Dado um tabuleiro de xadrez de tamanho $n \times n$, queremos colocar n rainhas no tabuleiro de forma que nenhuma rainha ataque a outra.

- Escolhas: Uma rainha por coluna; precisamos escolher uma linha para cada
- Restrições: Nenhuma rainha pode estar na mesma linha ou diagonal que outra
- Caso base: Se colocamos todas as n rainhas, encontramos uma solução
- Backtrack: Se em algum momento não conseguimos colocar uma rainha sem violar as restrições, voltamos para a coluna anterior e tentamos outra linha para a rainha daquela coluna.

Exemplo n rainhas

```
1  tabuleiro = [ ['. ' for _ in range(n)] for _ in range(n) ]
2
3  def rainha(n, tabuleiro, linha=0):
4      if linha == n:
5          print(tabuleiro)  # Encontramos uma solução
6          return
7
8      for coluna in range(n):
9          if is_valid(tabuleiro, linha, coluna):
10             tabuleiro[linha][coluna] = 'Q'  # Coloca a rainha
11             rainha(n, tabuleiro, linha + 1)  # Vai para a próxima linha
12             tabuleiro[linha][coluna] = '.'  # Remove a rainha (backtrack)
```

Exemplo n rainhas

```
1  def is_valid(tabuleiro, linha, coluna):
2      # Verificar a coluna
3      for i in range(len(tabuleiro)):
4          if tabuleiro[i][coluna] == 'Q':
5              return False
6      # Verificar a diagonal superior esquerda
7      for i, j in zip(range(linha, -1, -1), range(coluna, -1, -1)):
8          if tabuleiro[i][j] == 'Q':
9              return False
10     # Verificar a diagonal superior direita
11     for i, j in zip(range(linha, -1, -1), range(coluna, len(tabuleiro))):
12         if tabuleiro[i][j] == 'Q':
13             return False
14     return True
```

Exemplo n rainhas

- Como paramos na primeira vez que encontrarmos uma solução?
 - Digamos que só estamos interessados em saber se existe.

```
1  def rainha(n, tabuleiro, linha=0) -> bool:
2      if linha == n:
3          print(tabuleiro)  # Encontramos uma solução
4          return True
5      for coluna in range(n):
6          if is_valid(tabuleiro, linha, coluna):
7              tabuleiro[linha][coluna] = 'Q'      # Coloca a rainha
8              if rainha(n, tabuleiro, linha + 1): # Se há solução, show!
9                  return True
10             tabuleiro[linha][coluna] = '.'      # Remove a rainha (backtrack)
11     return False
```

Cuidado com listas!

- Diferente de variáveis comuns, listas são mutáveis
- Se passamos uma lista para a função recursiva e modificamos ela, isso pode afetar outras chamadas recursivas que usam a mesma lista
- Para evitar isso, podemos
 - criar uma cópia da lista para cada chamada recursiva,
 - ou **restaurar** a lista para o estado anterior após a chamada recursiva (backtrack)