Estruturas de Dados e Introdução à Projeto e Análise de Algoritmos - EDPAA
Laboratório 01 - Recursividade (Sudoku)
Prof. Luis Souza

I. Contexto

O **Sudoku** descreve de um jogo de raciocínio lógico e experimentação onde em sua versão mais comum, o jogador é desafiado a encaixar algarismos de 1 a 9 em uma grade 9 × 9, com 81 células distribuídas em 9 linhas, 9 colunas e 9 blocos 3 × 3, utilizando para isso, alguns algarismos fornecidos como dados iniciais (pistas do jogo) e com uma única regra restritiva: “Não repetir elementos nas linhas, colunas e blocos”. Neste jogo, precisa-se apenas verificar se o elemento em questão já foi ou não utilizado (na linha, coluna ou bloco), o que nos permite substituir os 9 dígitos (algarismos de 1 a 9), por 9 letras, ou 9 símbolos. O Sudoku como hoje conhecemos, foi criado em 1979, por Howard Garns, um arquiteto americano aposentado de 74 anos de idade e construtor de quebra cabeças, provavelmente utilizando como base o quadrado latino, assim denominado por Leonhard Euler (1707 - 1783) pelo fato de utilizar letras latinas em seus estudos. A semelhança entre o Sudoku e um quadrado latino de ordem 9 é notável, pois a solução de um quadrado latino, consistia em distribuir os elementos em linhas e colunas, sem que ocorresse repetição de elementos.

				1	9			6
			3		6			5
					4		2	9
4		5	1			9		
	3						5	
		1			5	2		8
2	1		9					
6			7		1			
5			6	2				

		4		6			9	
	9			2	7			
3			5	9		2		1
					6	5		
8		5				3		9
		2	9					
7		3		1	4			5
			7	5			3	
	4			3		8		

Figura 1. Ilustração de um Sudoku com grade 3x3.

II. Implementação

Para a resolução do Sudoku de forma computacional, temos as seguintes informações:

- Para cada posição vazia, pode haver mais de um candidato. Para resolver o problema então basta colocar os candidatos em cada uma das posições até encontrar uma combinação de candidatos que preencha toda a matriz. Pode haver mais de uma solução ou eventualmente nenhuma.

- Na solução convencional vamos experimentando o preenchimento de cada posição por tentativa e erro. O mesmo princípio pode ser usado num algoritmo.

A solução convencional pode ser expressa da seguinte forma:

Preencher a matriz S:

1. Procure uma casa livre – seja S[lin][col] essa posição
   Se não há posição livre já temos uma solução
2. Para cada um dos candidatos a essa posição:
   a. Faça S[lin][col] = candidato
   b. Prencher a matriz S (agora com mais uma posição preenchida)
3. Faça S[lin][col] = 0 (não há candidato que atenda a essa posição – retorne dessa chamada)

Veja que estamos invocando o preenchimento dentro do preenchimento, mas com pelo menos uma posição a mais já preenchida.

Detalhando um pouco mais o algoritmo, vamos a partir da posição S[0][0], com o seguinte algoritmo:

```
# Chamada inicial da função
PreencheSudoku(S)

# Preenche a matriz Sudoku
def PreencheSudoku(S):
     Procure uma posição livre a partir de S
     Se não há:
          # A matriz S está completa.
          retorne desta chamada
     Seja S[lin][col] a posição livre.
     # Podem haver zero ou mais candidatos a ocupar essa
     posição.
     # Os candidatos não podem estar na linha lin, na coluna
     col ou no quadrado correspondente.
     Para cada candidato:
          S[lin][col] = candidato.
          PreencheSudoku(S)
     # Neste ponto não há candidato para a posição S[lin][col]
     # Libera e retorna para que chamadas anteriores da função
     # possam tentar o preenchimento
     S[lin][col] = 0
```

```
retorne desta chamada
```

Passos:

1. **Inicializar o Grid (Tabuleiro)**
   Nesta etapa inicializa-se o “quadro” com um conjunto de números fornecidos pelo problema. Por convenção, células vazias são inicializadas com um valor ‘0’. O tipo de dados para manter os valores dos dígitos do quebra-cabeça Sudoku é uma lista bidimensional de tamanho 9x9 (ou NxN).
   A matriz inicial do Sudoku é fornecida em “hardcode”, sendo preenchida manualmente - não estamos interessados em preencher ela durante a execução do código.

2				8		3		
	6			7			8	4
	3		5			2		9
			1		5	4		8
4		2	7		6			
3		1			7		4	
7	2			4			6	
		4		1				3

Figura 2. Exemplo de solução inicial do Sudoku.

2. **Busca de nova posição “vazia” ou disponível**
   A busca é bastante simples: uma referência ao tabuleiro é passada como parâmetro e então a função o percorre até encontrar a primeira célula vazia que é retornada (conteúdo igual a 0). Caso não encontre células vazias, o problema está resolvido (todas as células foram preenchidas), portanto retorna o valor ‘Nenhum’.

3. **Verificação da validade de um candidato para uma dada posição do grid**
   Uma função deve confirmar se um determinado número candidato é uma opção válida para uma determinada célula do quadro. Existem 3 requisitos que devem ser atendidos para que o valor seja aceito na posição corrente:

   - Nenhuma das células da mesma linha deve conter o número examinado.

- Nenhuma das células da mesma coluna deve conter o número examinado.
- Nenhuma das células do bloco deve conter o número examinado

Se todos os itens acima forem atendidos, a função retornará True, o que acionará a inserção do novo valor no grid.

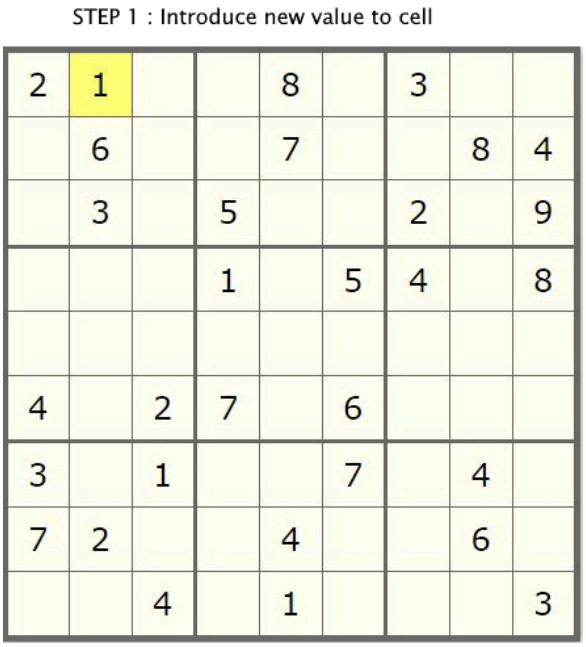

Figura 3. Exemplo de sucesso na inserção de um número candidato em uma posição do Grid.

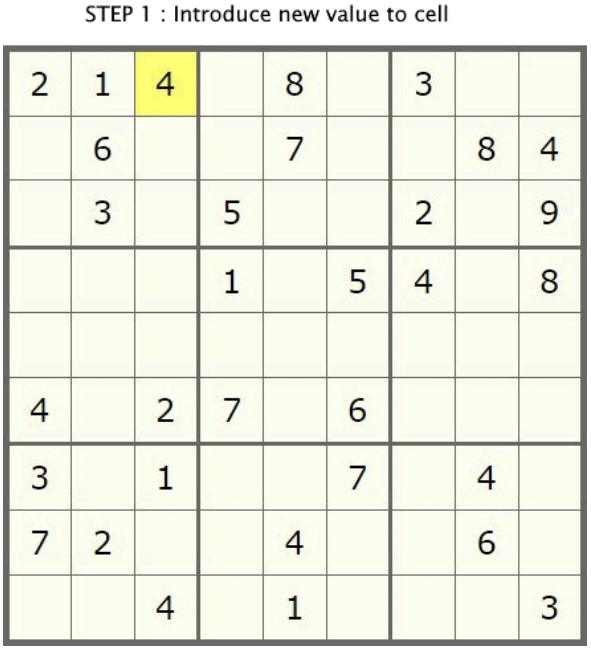

Figura 3. Exemplo de falha na inserção de um número candidato em uma posição do Grid.

4. **Implementação do algoritmo de solução do Sudoku (que utiliza backtracking)**

   A próxima função é o núcleo principal da solução, onde o backtracking é introduzido: o algoritmo pode ser descrito pelas seguintes etapas:

   1. Procure a próxima célula vazia. **Se nada for encontrado, então o problema já foi resolvido**; caso contrário, passamos para a etapa 2.
   2. Adivinhe o número correto iterando os números de 1 a 9 (ou a N, dependendo do tamanho do seu Grid) e examine se é uma solução válida, dados os números já estabelecidos no quadro.
   3. Se um número válido for encontrado, **chame a função de solução do Sudoku (recursivamente) para a próxima célula vazia.**
   4. Se nenhum dos números de 1 a 9 for válido, a chamada anterior da função redefinirá o valor da célula para 0 e continuará a iteração para encontrar o próximo número válido.

III. Backtracking

Backtracking é um algoritmo genérico que busca, por força bruta, soluções possíveis para problemas computacionais (tipicamente problemas de satisfações à restrições).

De maneira incremental, busca por candidatos a soluções e abandona cada candidato parcial C quando C não pode resultar em uma solução válida. Quando sua busca chega a uma extremidade da estrutura de dados, como um nó terminal de uma árvore, o algoritmo realiza um retrocesso tipicamente implementado através de uma recursão.

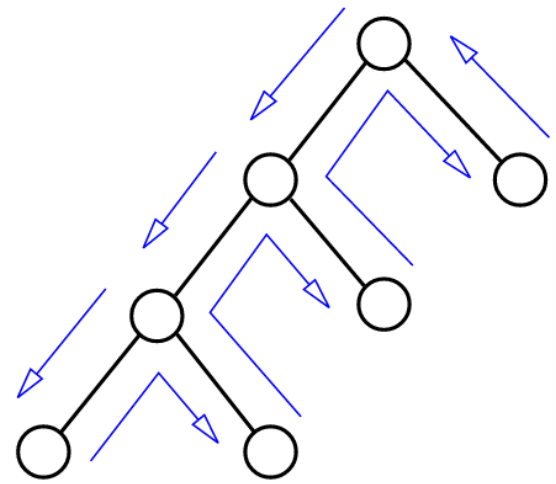

Figura 5. Exemplo de busca por backtracking.

Para o problema do Sudoku, um algoritmo de força bruta visita todas as células vazias, preenchendo os números quando foram válidos. Caso o algoritmo encontre alguma inconformidade, ele pode descartar todos os casos testados anteriormente.

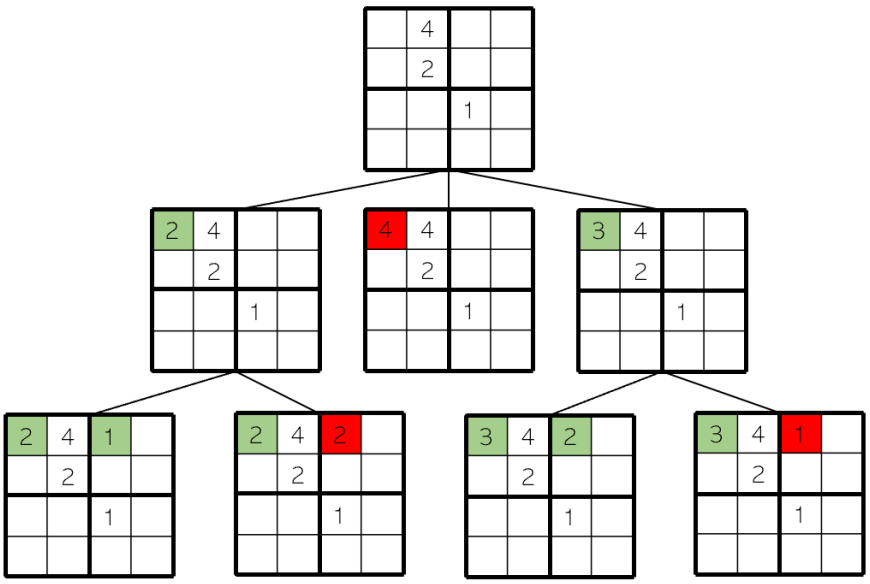

Figura 6. Ilustração da aplicação de backtracking na solução do problema do Sudoku. na busca por soluções, quando um candidato é encontrado e não satisfaz as regras do problema (para o caso específico do Sudoku, não é único para linha, coluna e grid *n* x *n*) toda a possível solução até aquele ponto é descartada.

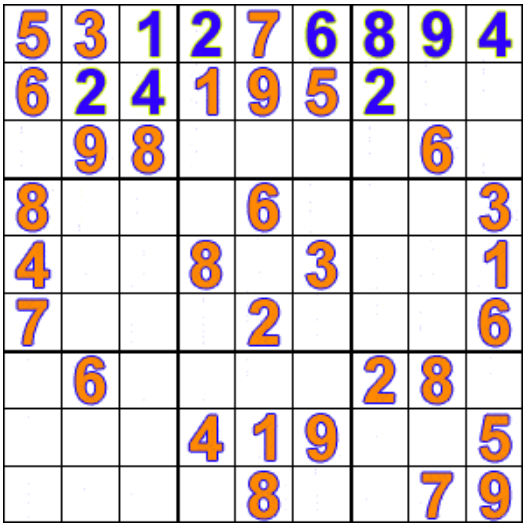

Figura 7. Ilustração da solução do Sudoku por backtracking.

Tarefa: Implementar o Sudoku em Python.
Para pensar: Como é o desempenho deste algoritmo? Pode ser melhorado?

Exemplo prático do funcionamento do algoritmo:

Entradas:

- grid = [
  [1, 0, 0, 0],
  [0, 0, 3, 4],
  [0, 0, 0, 0],
  [0, 3, 0, 1]
  ]
- dimSudoku = 4 → blocos de tamanho 2x2 (porque $\sqrt{4}$ = 2)

Chamadas de resolveSudoku:

1.

- Encontra primeira posição vazia (0,1) (linha 0, coluna 1)
- Teste de 1 a 4 para esta posição
  - 1 já está na linha
  - 2 é válido
- Grid atualizado: [1, 2, 0, 0]
  [0, 0, 3, 4]
  [0, 0, 0, 0]
  [0, 3, 0, 1]

2.

- Encontra posição vazia (0,2) (linha 0, coluna 1)

- Testa 1, 2, 3 e 4-> 4 ninguém é válido
- Grid atualizado: [1, 0, 0, 0]
  [0, 0, 3, 4]
  [0, 0, 0, 0]
  [0, 3, 0, 1]

- Avalia próximo candidato para (0,1) (linha 0, coluna 1)
- Testa 3 e 4 -> 4 é válido
- Grid atualizado: [1, 4, 0, 0]
  [0, 0, 3, 4]
  [0, 0, 0, 0]
  [0, 3, 0, 1]

3.
- Avalia próximo candidato para (0,2) (linha 0, coluna 2)
- Testa 1 e 2 -> 2 é válido
- Grid atualizado: [1, 4, 2, 0]
  [0, 0, 3, 4]
  [0, 0, 0, 0]
  [0, 3, 0, 1]

4.
- Avalia próximo candidato para (0,3) (linha 0, coluna 3)
- Testa 1, 2, 3 e 4 -> ninguém é válido -> logo, não tem solução