Exercício Programa 1 (Entregar até 15/09/2019)
MAC 122 – 2. Semestre de 2019 – BMAC - IMEUSP

Sudoku

O jogo Sudoku consiste em preencher os espaços vazios de uma matriz 9x9 com os algarismos de 1 a 9 de tal maneira que em cada linha, cada coluna e cada quadrado interno de 3x3 contenha todos os algarismos de 1 a 9 sem repetição. A matriz dada possui algumas posições já preenchidas com algum algarismo.

O jogo consiste em preencher os espaços vazios e ir testando as possibilidades até que a matriz esteja totalmente preenchida ou que se conclua não haver solução. Quanto mais posições vazias houver, maior o número de possibilidades, portanto mais difícil é o preenchimento. Para uma dada matriz, pode haver mais de uma solução.

O jogo é normalmente apresentado com graduações de dificuldade quando se joga manualmente (Muito Fácil, Fácil, Difícil, Muito Difícil). O nosso objetivo é um programa que preenche a matriz, testando todas as possibilidades de forma consistente, chegando à solução ou às soluções ou concluindo que não há soluções.

Se quiser exercitar suas habilidades no Sudoku, visite o site: https://www.geniol.com.br/logica/sudoku

O algoritmo para resolver o Sudoku é o mesmo que é usado quando o resolvemos manualmente:

- A partir de uma posição:
  - Procure a próxima posição vazia
    - Se não houver mais casas vazias, chegamos a uma solução
    - Mostre a solução e retorne desta chamada
  - Preencha com um algarismo dentre os possíveis candidatos
    - Se não houver mais candidatos, zere essa casa e retorne desta chamada. Assim, novas possibilidades serão testadas nas chamadas anteriores.
  - Retomar o processo, chamando novamente a função Sudoku agora com uma casa a mais já preenchida.

Da maneira descrita acima, o algoritmo é recursivo. Uma função Sudoku desta forma deve ter como parâmetros, a matriz, e a linha e coluna do último elemento que foi preenchido. A primeira chamada desta função, no programa principal seria então:

```
Sudoku(MatrizSudoku, 0, 0)
```

A técnica do algoritmo acima é chamada de **backtracking**. Avançamos no preenchimento das casas guardando as informações necessárias para **retroceder** caso cheguemos a uma impossibilidade de solução ou mesmo quando chegamos a uma solução.

Abaixo, exemplo de Sudoku e a matriz Sudoku correspondente como será fornecida. Casas vazias contém zero, os dígitos estão separados por um ou mais brancos e cada linha da matriz é uma linha de texto.

		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		

```
0 0 4 0 6 0 0 9 0
0 9 0 0 2 7 0 0 0
3 0 0 5 9 0 2 0 1
0 0 0 0 0 6 5 0 0
8 0 5 0 0 0 3 0 9
0 0 2 9 0 0 0 0 0
7 0 3 0 1 4 0 0 5
0 0 0 7 5 0 0 3 0
0 4 0 0 3 0 8 0 0
```

O programa deve:

1. Ler o nome do arquivo de texto: **<arquivo>.txt.**
2. Ler a matriz Sudoku – matriz 9x9. Como a matriz é grande (9 x 9 = 81 elementos), em vez de digitá-la, deve ser lida de um arquivo de texto **<arquivo>.txt**.
3. Fazer as consistências necessárias na matriz lida:
    - Verificar se cada linha tem 9 elementos e são todos int entre 0 e 9
    - Verificar se os valores não zero da matriz não estão repetidos na linha, coluna e quadrado interno.
4. Chamar a função **Sudoku** para efetuar o preenchimento da matriz de acordo com as regras do jogo, indicando todas as soluções possíveis.
5. A cada solução encontrada, verificar se a matriz foi preenchida corretamente. Faça o teste completo na matriz final verificando se cada linha, cada coluna e cada quadrado interno estão preenchidos corretamente.
6. Repetir a partir do passo 1 até que seja digitado **"fim"**.

Arquivos de Teste

Você pode testar o seu programa com qualquer arquivo de texto digitado no formato do exemplo acima (9 linhas por 9 colunas com os algarismos separados por um ou mais brancos). Pode digitar o seu arquivo usando qualquer editor de texto (ex: Bloco de Notas do Windows, ou mesmo o editor de texto do seu compilador). É conveniente que o arquivo de teste esteja no mesmo diretório que o seu programa para evitar ter que dar o caminho completo no open.

Além dos testes que você fará para certificar que o seu programa está funcionando adequadamente, será fornecido também um conjunto de arquivos de teste com nomes pré-definidos:
sudoteste1.txt, sudoteste2.txt, ...

Pode escolher entre carregar esse conjunto de arquivos em seu diretório de programas (download) fazendo acesso local ou acessar diretamente via internet fazendo acesso remoto.

Acesso local

A função LeiaMatrizLocal abaixo mostra como fazer essa leitura se a opção for pelo acesso local. Entenda-a e complete-a:

```
def LeiaMatrizLocal(NomeArquivo):
    # retorna a matriz lida se ok ou uma lista vazia senão

    # abrir o arquivo para leitura
    try:
        arq = open(NomeArquivo, "r")
    except:
        return [] # retorna lista vazia se deu erro

    # inicia matriz SudoKu a ser lida
    mat = [9 * [0] for k in range(9)]

    # ler cada uma das linhas do arquivo
    i = 0
    for linha in arq:
        v = linha.split()
        # verifica se tem 9 elementos e se são todos entre '1' e '9'
```

```
        # . . .
      # transforma de texto para int
        for j in range(len(v)):
            mat[i][j] = int(v[j])
        # faz as consistências iniciais da matriz dada
        # . . .
        i = i + 1
    # fechar arquivo e retorna a matriz lida e consistida
    arq.close()
    return mat
```

Acesso remoto

A função LeiaMatrizRemota abaixo mostra como fazer essa leitura via internet se a opção for pelo acesso remoto. Entenda-a e complete-a:

```
import urllib.request
def LeiaMatrizRemota(NomeArquivo):
    # retorna True se conseguiu ler o arquivo e False caso contrário
    # abrir o arquivo para leitura
    try:
        urlarq = "http://www.ime.usp.br/~mms/mac1222s2019/" + NomeArquivo
        arq = urllib.request.urlopen(urlarq)
    except:
        return [] # retorna lista vazia se der erro

    # inicia matriz SudoKu a ser lida
    mat = [9 * [0] for k in range(9)]
    # ler todo o arquivo e separar em linhas
    arqtotal = arq.read()
    linhas = arqtotal.splitlines()
    # tratar cada uma das linhasdo arquivo
    i = 0
    for linha in linhas:
        v = linha.split()
        # verifica se tem 9 elementos e se são todos entre '1' e '9'
        # . . .
      # transforma de texto para int
        for j in range(len(v)):
            mat[i][j] = int(v[j])
        # faz as consistências iniciais da matriz dada
        # . . .
        i = i + 1
    # fecha arquivo e retorna matriz lida e consistida
    arq.close()
    return mat
```

Funções do programa

Com o objetivo de estruturar melhor o seu programa, construa e use pelo menos as seguintes funções:

`def Sudoku(Mat, Lin, Col):` – função principal que preenche a matriz Sudoku, verificando se chegou ao final de uma solução e retrocedendo sempre que necessário.

`def ImprimaMatriz (Mat):` – imprime a matrix Sudoku **Mat[0..8][0..8].**

`def ProcuraElementoLinha (Mat, L, d):` – procura dígito **d** na linha **L** da matriz **(0 ≤ d ≤ 8)**. Devolve -1 se não encontrou ou índice da coluna onde foi encontrado.

`def ProcuraElementoColuna (Mat, C, d):` – procura dígito **d** na coluna **C** da matriz **(0 ≤ d ≤ 8)**. Devolve -1 se não encontrou ou índice da linha onde foi encontrado.

`def ProcuraElementoQuadrado (Mat, L, C, int d):` – procura o dígito **d** no quadrado interno onde está o elemento **Mat[L][C]** **(1 ≤ d ≤ 9)**. Devolve a dupla (k1, k2) se d está na posição Mat[k1][k2] ou (-1, -1) caso contrário.

`def TestaMatrizPreenchida(Mat):` – devolve True se a matriz Mat está preenchida corretamente. False caso contrário.

`def TestaMatrizLida(Mat):` – devolve True se a matriz lida Mat está com as casas já preenchidas com os valores corretos. Não há repetições na linha, na coluna ou no quadrado interno.

`def LeiaMatriz (Mat):` – conforme exemplo acima ou como achar mais conveniente.

Cronometrar o tempo de execução de cada solução

Para cronometrar o tempo de CPU gasto em cada solução use as funções do módulo time.

```
import time
. . .
tempo1 = time.clock()
# trecho a ser cronometrado
  . . . .
  . . . .
tempo2 = time.clock()
tempo_decorrido = tempo2 - tempo1
```

Contar o número de soluções para uma dada matriz

Conte o número de soluções para cada matriz. Podem haver zero, uma ou mais soluções. Como é uma solução recursiva, é conveniente ter um contador global (**`global contador`**) que é incrementado somente quando se chega a uma solução.

Exemplo de saídas do programa

```
Entre com o nome do arquivo:teste1.txt

* * * Matriz inicial * * *

3  4  0  0  0  7  0  9  0
0  0  8  3  0  9  0  0  0
0  9  0  0  6  0  0  4  3
4  5  0  2  1  0  0  0  0
8  0  0  0  7  0  0  0  1
0  0  0  0  5  3  0  2  0
1  0  2  0  9  0  0  8  0
0  0  0  1  0  0  7  0  0
6  0  0  7  0  2  0  5  9


* * * Matriz Completa

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


* * * Matriz Completa e Consistente
```

```
* * * Matriz Completa

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


* * * Matriz Completa e Consistente

* * * - Tempo decorrido =  1.6073447448421803  segundos

* * * - 2 soluções para esta matriz



Entre com o nome do arquivo:teste2.txt

...
...
...
```