Processo Seletivo CrossBots

Documentação - Python

Muito fácil - 1)

def `rad_deg(angle_rad)`

Recebe por parâmetro um ângulo em rad (número inteiro), o converte para ângulo em grau e retorna esse número.

Muito fácil - 2)

def `convert(hour)`

Recebe por parâmetro uma quantidade de horas (número inteiro), converte essa quantidade para minutos, segundos, milissegundos, semanas e meses e retorna uma lista contendo esses valores.

Fácil - 1)

def `repeat(item, n)`

Recebe por parâmetro um item (*string*) e uma quantidade N (número inteiro), cria uma lista e insere o item na lista N vezes e retorna essa lista.

Fácil - 2)

def `conta_uns(num)`

Recebe por parâmetro um número (inteiro), o converte para binário e retorna a quantidade de números 1 que existe no número binário em questão.

Médio - 1)

def `dist(A, B)`

Recebe por parâmetro um dicionário A e outro B. Ambos possuem como chave os números inteiros x e y, que correspondem a posição de um ponto no plano cartesiano. A função calcula a distância entre os pontos A e B e retorna esse número.

Médio - 2)

def `mult_3(num)`

Recebe por parâmetro um número, verifica se esse número é múltiplo de 3 e caso seja, retorna o valor booleano *True*.

def `mult_5(num)`

Recebe por parâmetro um número, verifica se esse número é múltiplo de 5 e caso seja, retorna o valor booleano *True*.

def `crossbots(num)`

Recebe por parâmetro um número, chama as funções acima para verificar se tal número é múltiplo de 3, 5, ambos ou nenhum e retorna uma *string* diferente para cada caso, ou retorna o próprio número caso não seja múltiplo de 3 e nem de 5.

Médio - 3)

def `getEnemy(distance)`

Recebe por parâmetro um número, verifica se esse número é menor que 300, e retorna o valor dele. Caso contrário, retorna o valor booleano *False*.

def `getBorder(light)`

Recebe por parâmetro um número, verifica se esse número é menor que 0.25, e retorna o valor booleano *True*. Caso contrário, retorna o valor booleano *False*.

def `control(inputs)`

Recebe por parâmetro as entradas dos sensores do robô (*dict*) e controla o estado (parado, andando pra frente, fazendo curva, procurando inimigo) e a velocidade do robô, baseada nos retornos das funções acima. Retorna um *dict* contendo as velocidades das 2 rodas e um log.

Difícil - 1)

def `newMatrix(N)`

Recebe por parâmetro um número e cria uma matriz quadrada N x N, recebendo o valor de cada posição do usuário, e retorna essa matriz.

def `bfs(visited, queue, matriz, N)`

Recebe por parâmetro as posições visitadas (lista), a fila (lista), o mapa (lista de lista) e o tamanho dele (número inteiro). Enquanto existir algo na fila, a função verifica todas as casas adjacentes da posição atual e caso alguma delas seja montanha e ainda não tenha sido visitada, ela será inserida na fila e nas visitadas e a posição atual é inserida

em um *dict* que corresponde aos caminhos encontrados, em que a *key* é a posição adjacente válida.

Quando não houver nenhum item na fila, verifica se existe uma posição final no *dict* de caminhos e caso exista, retorna o número de casas entre a posição final e inicial do mapa. Caso contrário, retorna SI. Solução baseada no algoritmo Breadth First Search.

Difícil - 2)

def `newMatrix(N = 9)`

Não recebe valor por parâmetro, apenas solicita do usuários os valores do tabuleiro de sudoku não resolvido, onde o número 0 representa o valor em branco.

def `printSudoku(matriz)`

Recebe por parâmetro o tabuleiro de sudoku (lista de lista) e apenas imprime a matriz com uma formatação adequada para apresentação de um sudoku.

def `getEmpty(matriz)`

Recebe por parâmetro o tabuleiro de sudoku (lista de lista), verifica em qual posição se encontra o primeiro número 0 (equivalente ao espaço em branco) e retorna essa posição como um tuple. Caso não encontre nenhum 0, retorna o valor booleano False.

def `isValid(matriz, num, pos)`

Recebe por parâmetro o tabuleiro de sudoku (lista de lista), o número atual e a posição atual. A função verifica se o número atual é válido na posição atual, checando se há um número igual na linha inteira, na coluna inteira e na seção correspondente e retorna o valor booleano *True* ou *False*, para válido ou não, respectivamente.

def `solve(matriz)`

Recebe por parâmetro o tabuleiro de sudoku (lista de lista), verifica se existe algum número 0 no tabuleiro, caso não exista retorna o valor booleano *True*, e caso exista, atribui a posição retornada à posição atual e testa todos os número de 1 a 9 nessa posição. Caso algum seja válido, ativa a recursividade, se chamando novamente e repetindo até alguma posição não possuir nenhum número válido. Então retornará o valor booleano *False* e o valor dessa posição volta pra 0 e tenta de novo a partir da última posição válida com outro número e assim sucessivamente. Solução baseada no algoritmo de Backtracking.

Difícil - 3)

def `getPi(dotsTotal)`

Recebe por parâmetro a quantidade total de pontos gerados (número inteiro) e os distribui em posições aleatórias (coordenadas x e y são números gerados aleatoriamente entre 0 e 1 para cada ponto). A função calcula a distância entre cada ponto e a origem, e conta quantos pontos estão a uma distância menor ou igual a 1, da origem,

que corresponde a quantidade de pontos dentro de um quarto de círculo de raio 1 inscrito em um quadrado. Então a função define pi como 4 vezes a quantidade de pontos no quarto de círculo (área desse objeto) sobre a quantidade total de pontos (área do quadrado) e retorna esse valor.