Processo Seletivo Crossbots 2022.2: Programação

Candidato 017 - Documentação dos códigos

Problema muito simples 1:

Função:

`distancia`: recebe as coordenadas de dois pontos e, através de uma fórmula que deriva do Teorema de Pitágoras, retorna a distância entre elas.

Problema muito simples 2:

Função:

`converte_temperatura`: recebe a temperatura em Fahrenheit e, por meio da fórmula da conversão de Fahrenheit para Celsius, retorna a temperatura em Celsius.

Problema simples 1:

Não possui funções, mas é importante ressaltar que a entrada para o vetor deve ser no formato “lista = [a, b, c, d, e, f, g, h, i, j]” para o funcionamento adequado do código.

Problema simples 2:

Função:

`verifica_perfeito`: recebe um número inteiro (n), soma todos os seus divisores (números para os quais o resto é zero na divisão desse número n por eles), e então verifica se essa soma é igual ao valor do próprio número n, visto que essa é a definição de um número perfeito. Se isso for verdadeiro, retorna 1, se não, retorna 0.

Problema simples 3:

Função:

`encontra_primo`: recebe um vetor (vtr[]) e o “endereço” de seu tamanho (*n), analisa qual será o tamanho da lista com números primos (números naturais com somente dois divisores, o que é verificado no loop), armazena a lista com somente os números primos em um vetor auxiliar (vtr_final[]), altera o tamanho do vetor no código fonte para o novo tamanho e, por fim, retorna o vetor auxiliar, o qual contém o conteúdo desejado.

Problema simples 4:

Função:

`encontra_letra`: recebe um caracter e uma string e verifica, caracter a caracter, quantos caracteres na string são iguais ao caracter que foi recebido, retornando esse valor.

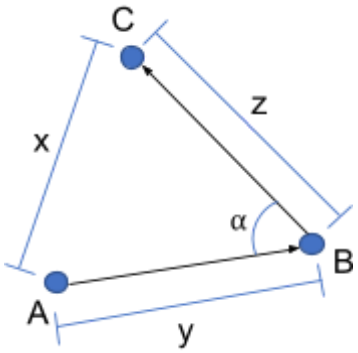
Problema intermediário 1:

Função:

`angle_ABC`: recebe as coordenadas dos pontos A, B e C e retorna o ângulo ABC através de uma fórmula que será demonstrada abaixo.

Base teórica:

Para resolução desse problema, foi considerado ângulo formado entre os vetores $\vec{AB}$ e $\vec{BC}$, sendo feita a seguinte análise:



Coordenadas dos pontos:
A: (Ax, Ay)
B: (Bx, By)
C: (Cx, Cy)

Utilizando a Lei dos Cossenos, é possível concluir que

$$x^2 = y^2 + z^2 - 2.y.z.cos(\alpha)$$

Os valores de x, y e z são obtidos pelas seguintes equações:

$$x = [(Cx - Ax)^2 + (Cy - Ay)^2]^{1/2}$$

$$y = [(Bx - Ax)^2 + (By - Ay)^2]^{1/2}$$

$$z = [(Cx - Bx)^2 + (Cy - By)^2]^{1/2}$$

Portanto, fazendo a substituição dessas equações na fórmula anteriormente citada, temos que

$$\alpha = arccos\left(\frac{Bx^2+By^2-Ax.Bx+Ax.Cx-Bx.Cx-Ay.By+Ay.Cy-By.Cy}{[(Bx-Ax)^2+(By-Ay)^2]^{1/2}.[(Cx-Bx)^2+(Cy-By)^2]^{1/2}}\right)$$

Enfim, para que α seja transformado de radianos para graus, basta multiplicá-lo por $180/\pi$.

Problema intermediário 2:

Função:

`tira_repeticoes`: recebe um vetor (lista[]) e o “endereço” de seu tamanho (*n), analisa qual será o tamanho da lista sem repetições por meio do primeiro loop (ele faz isso verificando, número a número, quais não possuem repetições “a sua esquerda” no vetor, os quais são contados, o que faz com que os elementos contados sejam unicamente aqueles que

não são repetições de números anteriormente contados), armazena a lista sem repetições em um vetor auxiliar (lista_final[]) através do segundo loop (usa a mesma lógica de verificação do primeiro loop), altera o tamanho do vetor no código fonte para o novo tamanho e, por fim, retorna o vetor auxiliar, o qual contém o conteúdo solicitado.

Problema difícil 1:

Para resolução desse problema foi utilizada uma struct (strings) para que fosse possível armazenar os casos teste em um "vetor de strings". Além disso, foi criada a função verifica_diamantes para contar os diamantes de cada caso teste.

Função:

`verifica_diamantes`: função recursiva que recebe a string do caso teste (str[]) e o "endereço" número de diamantes (*n), conta o número de diamantes e o altera no código fonte.

A lógica dessa função é aplicar diretamente o que o enunciado disse a respeito dos diamantes, assim, no loop "externo", de elemento em elemento da string, é verificado se há o caracter "<", e se há, é feita uma verificação dos elementos seguintes da string no loop "interno", em que, se é encontrado "<", o loop "interno" é interrompido, e se é encontrado ">", o número de diamantes é adicionado em uma unidade, visto que um diamante foi encontrado (com ou sem areia no meio), os elementos da string nos quais estavam "<" e ">" se transformam em ".", para que não sejam mais considerados na contagem, e então o loop "interno" é interrompido. Ademais, foi utilizada uma variável auxiliar (aux) que é somada em uma unidade cada vez que é encontrado um diamante, para que a função verifica_diamante seja chamada novamente até não for possível encontrar outro diamante, ou seja, até que a variável auxiliar seja igual a zero.

Problema difícil 2:

Para resolução desse problema, foi feita a subdivisão do código em um número razoável de funções, o que facilita o entendimento do programa. Nele, foram usadas funções para analisar todas as peças do tabuleiro (com exceção dos reis), verificando se alguma está fazendo com que algum dos reis esteja em cheque.

Ademais, foi criada uma struct ("caracteres") para facilitar a manipulação da entrada, sendo tida como um vetor de strings, em que cada vetor é uma linha da entrada e cada posição das strings é uma coluna dela, o que permite ver a entrada como uma matriz de caracteres, e é assim que ela foi considerada na explicação das funções.

Funções:

`condicoes`: recebe uma posição da "matriz" (linha e coluna) e o tamanho dela e verifica se essa posição (que representa uma casa) está dentro ou fora do tabuleiro, retornando 1 se está e 0 se não. Usada para que não sejam feitas verificações em elementos fora da "matriz".

`verifica_peao`: recebe a "matriz" e seu tamanho, verifica se há peões no tabuleiro, analisa pontualmente todas as possibilidades que cada peão tem de atacar e retorna, de acordo exclusivamente com o peão, "B" se o rei branco está em cheque, "P" se o rei preto está em cheque ou "N" se nenhum está.

`verifica_cavalo`: recebe a “matriz” e seu tamanho, verifica se há cavalos no tabuleiro, analisa pontualmente todas as possibilidades que cada cavalo tem de atacar e retorna, de acordo exclusivamente com o cavalo, "B" se o rei branco está em cheque, "P" se o rei preto está em cheque ou "N" se nenhum está.

`verifica_torre`: recebe a “matriz” e seu tamanho, verifica se há torres no tabuleiro, analisa, através de laços for, todas as possibilidades que cada torre tem de atacar e retorna, de acordo exclusivamente com a torre, "B" se o rei branco está em cheque, "P" se o rei preto está em cheque ou "N" se nenhum está.

`verifica_bispo`: recebe a “matriz” e seu tamanho, verifica se há bispos no tabuleiro, analisa, através de laços for e uma variável auxiliar, todas as possibilidades que cada torre tem de atacar e retorna, de acordo exclusivamente com o bispo, "B" se o rei branco está em cheque, "P" se o rei preto está em cheque ou "N" se nenhum está.

`verifica_rainha`: recebe a “matriz” e seu tamanho, verifica se há rainhas no tabuleiro, analisa, através de laços for e uma variável auxiliar, todas as possibilidades que cada rainha tem de atacar e retorna, de acordo exclusivamente com a rainha, "B" se o rei branco está em cheque, "P" se o rei preto está em cheque ou "N" se nenhum está. Para análise de cada rainha, foi feita uma junção das análises da torre e do bispo, visto que as possibilidades de ataque dela são uma mistura das possibilidades dos dois.

`verifica_tabuleiro`: recebe a “matriz” e seu tamanho e retorna 1 se todos os elementos dela forem iguais a “.” e 0 se não. Usada para que quando um tabuleiro sem nenhuma peça for colocado na entrada o laço while seja interrompido e, assim, o programa finalize.