Prova 2 (2020/02)

Valor: 27 pontos

Ao concordar em fazer esta prova, eu juro que seguirei o **código de honra**:

1. Não ajudarei e nem pedirei ajuda a ninguém durante o período de submissão do exame;
2. Não divulgarei qualquer informação sobre as minhas soluções durante o período de submissão do exame.

É normal não entender completamente uma questão. Se você tiver dúvidas sobre qualquer questão, envie uma mensagem ***PRIVADA*** no Teams ou um e-mail para olmo@dcc.ufmg.br com o assunto **[PROVA2]** e explique a sua dúvida para mim. Tentarei responder o mais rápido possível, mas a resposta pode demorar se a mensagem for enviada fora do horário comercial. **Não responderei mensagens enviadas através do sistema do Moodle!**

Sumário

Esta prova está dividida em três blocos. No primeiro bloco, composto pelas questões **1** a **4**, você vai implementar funções para preencher e localizar blocos quadrados com valores não nulos (isto é, diferentes de zero) em uma matriz quadrada de inteiros. Para isso, você vai precisar usar o tipo de dados `Bloco`. No segundo bloco, composto pela questão **5**, você vai implementar uma função para contar o número de caracteres distintos em uma *string*. Por fim, no terceiro bloco, composto pela questão **6**, você vai implementar uma função para identificar colisões entre retângulos e círculos em um sistema de coordenadas idêntico ao usado pela biblioteca Allegro.

Importante: você pode criar quantas funções extras desejar. Isso é **altamente** recomendável para as questões **3**, **4** e **6**. Dividir é a melhor estratégia para conquistar!

Exercícios

Para todos os exercícios, considere a definição abaixo, que é fornecida para vocês na versão básica da prova:

```
#define MAX_TAM 100
```

Para as questões **2** a **4**, considere o tipo de dados **`Bloco`** descrito abaixo:

```
typedef struct Bloco {
     int linha, coluna, tam;
} Bloco;
```

Essa estrutura armazena as informações sobre um bloco quadrado (número de colunas = número de linhas) localizado (ou a ser inserido) dentro da matriz. Os campos `linha` e `coluna` guardam a **coordenada inicial** do bloco, ou seja, os identificadores da linha e da coluna em que o bloco **começa** (célula do bloco que está mais à esquerda e ao topo). O campo `tam` guarda o número de linhas e colunas que o bloco tem. Por exemplo, considere a matriz abaixo:

0	0	0	0	0
0	0	**1**	**1**	**1**
0	0	**1**	**1**	**1**
0	0	**1**	**1**	**1**
1	0	0	0	0

Exemplo 1

É possível identificar dois blocos na matriz acima. Um bloco é definido pela tupla **(1, 2, 3)**, ou seja, está localizado na `linha`=1 e `coluna`=2, e tem tamanho `tam`=3. O segundo bloco é definido pela tupla **(4, 0, 1)**, ou seja, localizado na `linha`=4 e `coluna`=0, e tem tamanho `tam`=1.

1) Implemente uma função de nome **zeraMatriz** que recebe uma matriz `M` e o seu número `n` de linhas e colunas como parâmetros e atribui **zero** a todas as suas células. Protótipo:

```
void zeraMatriz(int M[][MAX_TAM], int n);
```

Valor: **2 pontos**.

2) Implemente uma função de nome **insereBloco** que recebe como parâmetros uma matriz quadrada **M**, o seu tamanho **n**, e um bloco **b**. Esta função deve atribuir o valor **1** a todas as células da matriz que compõem o bloco. Se **n** for maior ou igual a **2** e **b** é definido pela tupla **(0,0,2)**, então a sua função deve atribuir **1** às células `M[0][0]`, `M[0][1]`, `M[1][0]` e `M[1][1]`. Caso o tamanho do bloco ultrapasse os limites da matriz ou caso a linha ou coluna iniciais do bloco sejam inválidas, nenhuma modificação deve ser feita na matriz e a sua função deve retornar **0**. Caso seja possível inserir o bloco, modifique a matriz e retorne **1**. Protótipo:

```
int insereBloco(int M[][MAX_TAM], int n, Bloco b);
```

Valor: **5 pontos**.

3) Implemente uma função de nome **maiorBloco** que recebe como parâmetros uma matriz quadrada **M**, o seu tamanho **n**, e uma coordenada **i** e **j** de **M**, que representam uma linha e uma coluna dessa matriz, respectivamente. Esta função deve retornar as informações do maior bloco de **M** com coordenada inicial em (linha=**i** e coluna=**j**). Na matriz do **Exemplo 1**, se **i**=1 e **j**=2, a sua função deve retornar o bloco composto pelos campos (`linha`=1, `coluna`=2, `tam`=3). No entanto, considerando essa mesma matriz, se **i**=2 e **j**=3, a sua função deve retornar o bloco composto pelos campos (`linha`=2, `coluna`=3, `tam`=2). Se não for possível formar um bloco a partir da coordenada **(i,j)**, retorne o bloco composto pelos campos (`linha`=**i**, `coluna`=**j**, `tam`=0). Na matriz do **Exemplo 1**, se **i**=1 e **j**=1, a sua função deve retornar o bloco composto pelos campos (`linha`=1, `coluna`=1, `tam`=0). Protótipo:

```
Bloco maiorBloco(int M[][MAX_TAM], int n, int i, int j);
```

Valor: **5 pontos**.

4) Implemente uma função de nome **`encontraBlocos`** que encontra um **conjunto total e maximal** de blocos em uma matriz. Um **conjunto total** de blocos em uma matriz é um conjunto de blocos em que cada célula não nula (diferente de zero) da matriz faz parte de um único bloco do conjunto. Se alguma célula não nula da matriz não fizer parte de nenhum bloco do conjunto, então o conjunto não é **total**. Um **conjunto maxima**l é um conjunto em que é impossível juntar blocos do conjunto para formar um bloco de tamanho maior. Considere os conjuntos abaixo, formados com células da matriz do **Exemplo 1**:

A: { (1, 2, 2), (1, 4, 1), (2, 4, 1), (3, 2, 1), (3, 3, 1), (3, 4, 1), (4, 0, 1) } - **total** e **não maximal**
B: { (1, 2, 1), (2, 3, 2), (4, 0, 1) } - **não total** e **maximal**
C: { (1, 2, 2), (1, 4, 1), (2, 3, 2), (3, 2, 1), (4, 0, 1) } - **não total** e **não maximal**
D: ((1, 2, 3), (4, 0, 1) } - **total** e **maximal**

O conjunto **A** **é total**, pois cada uma das células não nulas da matriz faz parte de um único bloco, mas **não é maximal**, pois se juntarmos os blocos (1, 2, 2), (1, 4, 1), (2, 4, 1), (3, 2, 1), (3, 3, 1) e (3, 4, 1) formamos o bloco (1, 2, 3), de tamanho 3.

O conjunto **B** **não é total**, pois há células não nulas da matriz, como a célula **(1,3)**, que não fazem parte de nenhum bloco de **B**. No entanto, o conjunto **B** **é maximal**, pois não é possível juntar os blocos que o compõem para formar um bloco maior.

O conjunto **C** **não é total**, pois a célula **(2,3)** faz parte de mais de um bloco, do bloco (1, 2, 2) e do bloco (2, 3, 2). O conjunto **C** também **não é maximal**, pois se juntarmos os blocos (1, 2, 2), (1, 4, 1), (2, 3, 2), (3, 2, 1) formamos o bloco (1, 2, 3), de tamanho 3.

Por fim, o conjunto **D** **é total e maximal**.

A sua função deve procurar por blocos maximais e totais a partir da célula **(0,0)** da matriz, percorrendo-a no sentido da esquerda para a direita e de cima para baixo. **Dica**: use a função **maiorBloco** do exercício anterior. Protótipo:

```
void encontraBlocos(int M[][MAX_TAM], int n, Bloco blocos[], int *numBlocos)
```

Valor: **5 pontos**.

5) Implemente uma função de nome **`numCharsDiferentes`** que retorna o número de caracteres distintos da *string* **str** que é passada como parâmetro. A sua função não deve fazer distinção entre letras maiúsculas e minúsculas. Assuma também que **str** não conterá nenhum caractere especial, ou seja, com código ASCII maior que 126. Exemplo: se a string for "*Adoro programar em C!*", a sua função deve retornar **11** (as aspas não fazem parte da *string*). Protótipo:

```
int numCharsDiferentes(char str[]);
```

Valor: **5 pontos**.

6) Para este exercício, considere os tipos de dados abaixo:

```
typedef struct Ponto {
      float x, y;
} Ponto;

typedef struct Circulo {
      Ponto centro;
      float raio;
} Circulo;

typedef struct Retangulo {
      Ponto sup_esq, inf_dir;
} Retangulo;
```

Neste exercício, você deve implementar a função **`colisaoCirculoRetangulo`**, que recebe um `Circulo` e um `Retangulo` como parâmetros e calcula se há uma colisão entre eles. Se há uma colisão, então a função deve retornar **1**. Caso contrário, a função retorna **0**. Um círculo é definido pelas coordenadas **x** e **y** do seu **centro** e também pelo **raio**. Um retângulo é definido pelas coordenadas **x** e **y** do seu vértice superior esquerdo (`sup_esq`) e pelas coordenadas **x** e **y** do seu vértice inferior direito (`inf_dir`). Considere o mesmo sistema de coordenadas da biblioteca *Allegro*, ou seja, o eixo **x** cresce da esquerda para a direita e o eixo **y** cresce de cima para baixo. Protótipo:

```
int colisaoCirculoRetangulo(Circulo cir, Retangulo ret);
```

Valor: **5 pontos**.

Execução no VPL

Para conseguir compilar o seu programa, você deve implementar uma versão sintaticamente correta de todas as funções pedidas. Sugiro fortemente que faça isso antes de pensar nas soluções. Para o exercício **2**, por exemplo, você pode implementar a seguinte função:

```
int insereBloco(int M[][MAX_TAM], int n, Bloco b) { return 0; }
```

Para facilitar, a versão inicial desta prova já vem com essas versões básicas.

Uma forma de testar as suas soluções é através dos testes automáticos disponibilizados para vocês. Para executar os testes automáticos, clique no botão “avaliar” do VPL. Sugiro que faça isso antes de iniciar as soluções para poder visualizar todos os testes disponíveis para esta prova. Você pode avaliar o seu programa quantas vezes quiser. **Importante:** os testes **não** são completos e servem **apenas** para dar uma ideia **inicial** sobre a efetividade da questão. Além disso, eles não são proporcionais aos valores reais de cada questão.

Outra forma de testar as suas soluções no VPL é através do botão “Executar”. Para testar um exercício, inicie a execução no VPL e digite o número da função que gostaria de testar (**1**, **2**, **3**, **4, 5** ou **6**).

Para a função **1**, digite também o tamanho **n** da matriz.

Para a função **2**, digite também o tamanho **n** da matriz e a representação de um bloco (linha, coluna, tamanho). Exemplo: “`2 5 0 0 2`” testa a função **2**, cria uma matriz de tamanho **5** e insere um bloco de tamanho **2** na linha **0** e coluna **0**.

Para a função **3**, digite também o tamanho **n** da matriz e a representação de um bloco (linha, coluna, tamanho) a ser inserido nessa matriz. Por fim, digite a linha e a coluna da matriz em que você deseja procurar pelo maior bloco. Exemplo: “`3 5 0 0 2 1 1`” testa a função **3** na célula (1,1) de uma matriz de tamanho **5** com um bloco de tamanho **2** com coordenada inicial na linha **0** e coluna **0**.

Para a função **4**, digite também o tamanho **n** da matriz e uma sequência de representações de blocos para serem inseridos na matriz. O último bloco desta sequência deve ter tamanho **0**, indicando que a sequência terminou. Exemplo:

```
4 6
0 0 2
0 4 2
2 0 1
0 0 0
```

Neste exemplo, uma matriz de tamanho **6** é criada e os blocos (0, 0, 2), (0, 4, 2) e (2, 0, 1) são inseridos na matriz.

Para a função **5**, digite também a string para ser avaliada. Exemplo:

```
5
É mentira! Eu odeio programar em C!!! :(((((
```

Para a função **6**, digite também as coordenadas do círculo e do retângulo. Exemplo:

```
6
0 0 10
-5 10 5 20
```

Nesse exemplo, são criados um círculo de raio **10** com centro na coordenada **(0,0)** e um retângulo com vértice superior esquerdo na coordenada **(-5, 10)** e com vértice inferior direito na coordenada **(5, 20)**.

Por fim, você pode chamar a função `minha_main()` dando o inteiro **0** de entrada para o programa. Você pode implementar o que quiser nessa função, que serve para você testar qualquer situação particular que você desejar. **Essa função não será avaliada nesta prova!**