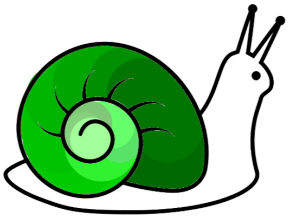
MC202E - ESTRUTURAS DE DADOS

Lab01: Matriz Caracol de Willian

Willian é uma criança de muita imaginação. Hoje ele aprendeu sobre matrizes especiais em sua escola, tais como a matriz identidade, matriz escada, etc. Na hora do recreio, ao brincar no parque do Caracol, ele resolveu criar sua própria matriz especial, a matriz caracol. Uma matriz é dita ser *caracol* se forma uma **espiral decrescente** de fora para dentro no sentido anti-horário, começando no canto inferior esquerdo. Veja o exemplo a seguir.

```
6.9 7.1 7.2 7.8
6.6 0.8 2.1 8.0
6.2 0.5 2.2 8.5
5.8 5.7 3.0 9.1
```



Tarefa

Escreva um programa em C que, dado como entrada dois inteiros m e n e um vetor **ordenado de forma decrescente** de números reais de tamanho $m*n$, imprima na saída padrão sua forma caracol, i.e., a matriz caracol de Willian.

Entrada

A entrada é composta por duas linhas. A primeira linha da entrada contém dois inteiros positivos m e n, que correspondem às dimensões da matriz caracol. A linha seguinte contém $m*n$ números reais.

Restrição

❑ $3 \leq m, n \leq 100$

Saída

A saída do seu programa deverá conter m linhas, com n números reais (com precisão de uma casa decimal) em cada linha, correspondente à matriz caracol obtida a partir do vetor dado como entrada.

Observações

❑ Para a **precisão de uma casa decimal**, você pode usar o `printf("%.1f", x)`.

☞ **Aqui** você pode aprender um pouco mais sobre os especificadores de formato.

❑ O final de cada uma das ***m*** linhas da saída **não deve conter** um espaço em branco.
❑ A última linha da saída deve estar em branco, i.e., deve conter o caracter **`'\n'`**.

Exemplos

A grafia da saída abaixo deve ser seguida rigorosamente por seu programa, inclusive a impressão da linha em branco no final da saída.

#1 Entrada

```
4 4
9.1 8.5 8.0 7.8 7.2 7.1 6.9 6.6 6.2 5.8 5.7 3.0 2.2 2.1 0.8 0.5
```

#1 Saída

```
6.9 7.1 7.2 7.8
6.6 0.8 2.1 8.0
6.2 0.5 2.2 8.5
5.8 5.7 3.0 9.1

```

#2 Entrada

```
3 5
8.5 8.0 7.8 7.2 7.1 6.9 6.6 6.2 5.8 5.7 3.0 2.2 2.1 0.8 0.5
```

#2 Saída

```
6.6 6.9 7.1 7.2 7.8
6.2 0.5 0.8 2.1 8.0
5.8 5.7 3.0 2.2 8.5

```

Critérios específicos

Os seguintes critérios específicos sobre o envio, implementação, compilação e execução devem ser satisfeitos:

i. Submeter no *SuSy* apenas o arquivo:
⇒ **`lab01.c`**: programa principal.

ii. É obrigatório utilizar vetor e matriz.

iii. Flags de compilação:
`-std=c99 -Wall -Werror -g -lm`

iv. Tempo máximo de execução: 1 segundo.

Observações gerais

No decorrer do semestre haverá 3 tipos de tarefas no *SuSy* (descritas logo abaixo). As tarefas possuirão os mesmos casos de testes abertos e fechados, no entanto o número de submissões permitidas e prazos são diferentes. As seguintes tarefas estão disponíveis no *SuSy*:

- **Lab01-AmbienteDeTeste**: Esta tarefa serve para testar seu programa no *SuSy* antes de submeter a versão final. Nessa tarefa, tanto o prazo quanto o número de submissões são ilimitados, porém os arquivos submetidos aqui **não serão corrigidos**.
- **Lab01-Entrega**: Esta tarefa tem limite de uma **única** submissão e serve para entregar a **versão final** dentro do prazo estabelecido para o laboratório. Não use essa tarefa para testar o seu programa e submeta aqui apenas quando não for mais fazer alterações no seu programa.
- **Lab01-ForaDoPrazo**: Esta tarefa tem limite de uma **única** submissão e serve para entregar a versão final fora prazo estabelecido para o laboratório. Esta tarefa irá substituir a nota obtida na tarefa **Lab01-Entrega** apenas se o aluno tiver realizado as correções sugeridas no *feedback* ou caso não tenha enviado anteriormente na tarefa **Lab01-Entrega**.

Avaliação

Este laboratório será avaliado da seguinte maneira: se o seu programa apresentar resposta correta para todos os casos de teste do *SuSy*, então é nota 10; caso contrário, é nota 0 (i.e, seu programa apresentou resposta incorreta para pelo menos um caso de teste aberto ou fechado).

Testando seu programa

Para compilar usando o `Makefile` fornecido e testar se a solução do seu programa está correta, basta seguir o exemplo abaixo.

```
make
./lab01 < testes_abertos/arq01.in > testes_abertos/arq01.out
diff testes_abertos/arq01.out testes_abertos/arq01.res
```

O `arq01.in` é a entrada (caso de teste disponível no *SuSy*) e `arq01.out` é a saída do seu programa. O `Makefile` também contém regras para baixar e testar todos os testes de uma única vez; nesse caso, basta digitar conforme o exemplo a seguir.

```
make baixar_abertos
```

```
make testar_abertos
```

A primeira instrução irá baixar os casos de teste abertos para a pasta **testes_abertos**, criada automaticamente, e a segunda instrução irá testar o seu programa com os casos de teste abertos. Após o prazo, os casos de teste fechados serão liberados e podem ser baixados e testados da mesma forma que os testes abertos. Para isso, basta trocar `"_abertos"` por `"_fechados"` (e.g., `make baixar_fechados`).

Praticando

Obs.: O "exercício" a seguir é opcional e não valerá ponto para a avaliação desta tarefa.

À medida que resolvemos problemas maiores e complexos, fica difícil identificar erros em um programa. Na verdade, é muito comum que os programas contenham erros nas primeiras implementações. Neste sentido, aprenderemos a seguir como utilizar o depurador de programas chamado GDB (programa que permite a um programador visualizar o que acontece no interior de um outro programa durante sua execução).

Leia primeiro o **tutorial de GBD** disponível na página da disciplina.

Após ter implementado o `lab01.c` e compilado com sucesso, abra o terminal execute os comandos (apresentados na cor verde) conforme ilustrado a seguir.

```
gdb lab01
...
Reading symbols from lab01...done.
(gdb) break main
Breakpoint 1 at 0x775: file lab01.c, line 12.
(gdb) run < arq02.in
(gdb) print V
$1 = {-1.03697632e+34, 4.59163468e-41, -1.03806422e+34,
4.59163468e-41, 0, 0, 0, 0, 0, 0, 3.02202344e+32, 0,
1.26116862e-44, 0, -8.98358231e+33}
(gdb) print M
$2 = {{-nan(0x7fdb68), 4.59163468e-41, 2.2105807e-38, 0,
1.40129846e-45}, {0, 1.46575794e+13, 3.0611365e-41,
-9.01959151e+33, 4.59163468e-41}, {0, 0, 1.46574986e+13,
3.0611365e-41, 1.46561397e+13}}
(gdb) next
17      scanf("%d %d", &m, &n);
```

Obs.: Nessa implementação temos que **`V`** e **`M`** correspondem respectivamente ao vetor e a matriz utilizados durante a execução.

Tente responde: Por que foi impresso valores estranhos?

Agora, use o comando **`next`** e **`print`** no decorrer da leitura dos elementos que compõem o vetor de entrada para ver como ele é preenchido.

```
(gdb) next
...
(gdb) print V
$3 = {8.5, 4.59163468e-41, -1.03806422e+34, 4.59163468e-41, 0, 0,
0, 0, 0, 0, 3.02202344e+32, 0, 1.26116862e-44, 0, -8.98358231e+33}
...
(gdb) next
...
(gdb) print V
$4 = {8.5, 8, -1.03806422e+34, 4.59163468e-41, 0, 0, 0, 0, 0, 0,
3.02202344e+32, 0, 1.26116862e-44, 0, -8.98358231e+33}
```

Após a leitura da entrada, veja se seu programa está preenchendo corretamente a matriz de forma análoga a leitura do vetor.

```
(gdb) next
...
(gdb) print M
$5 = {{-nan(0x7fdb68), 4.59163468e-41, 2.2105807e-38, 0,
1.40129846e-45}, {0, 1.46581666e+13, 3.0611365e-41,
-9.01959151e+33, 4.59163468e-41}, {0, 0, 1.46580858e+13,
3.0611365e-41, 8.5}}
...
(gdb) next
...
(gdb) print M
$6 = {{-nan(0x7fdb68), 4.59163468e-41, 2.2105807e-38, 0,
1.40129846e-45}, {0, 1.46581666e+13, 3.0611365e-41,
-9.01959151e+33, 8}, {0, 0, 1.46580858e+13, 3.0611365e-41, 8.5}}
```

Coloque *breakpoints* na leiteira dos elementos do vetor e na inicialização dos elementos da matriz. Agora, tente combinar com o comando **`continue`**. Além disso, tente simplificar a impressão usando o comando **`display`**.