



### Trabalho Prático III (TP III) - 10 pontos, peso 1.

- Data de entrega: 20/08/2023 até 23:55. O que vale é o horário do *Moodle*, e não do *seu*, ou do *meu* relógio!!!
- Clareza, identificação e comentários no código também vão valer pontos. Por isso, escolha cuidadosamente o nome das variáveis e torne o código o mais legível possível.
- O padrão de entrada e saída deve ser respeitado exatamente como determinado no enunciado. Parte da correção é automática, não respeitar as instruções enunciadas pode acarretar em perda de pontos.
- Durante a correção, os programas serão submetidos a vários casos de testes, com características variadas.
- A avaliação considerará o tempo de execução e o percentual de respostas corretas.
- Eventualmente serão realizadas entrevistas sobre os estudos dirigidos para complementar a avaliação;
- O trabalho é em grupo de até 2 (duas) pessoas.
- Entregar um relatório.
- Os códigos fonte serão submetidos a uma ferramenta de detecção de plágios em software.
- Códigos cuja autoria não seja do aluno, com alto nível de similaridade em relação a outros trabalhos, ou que não puder ser explicado, acarretará na perda da nota.
- Códigos ou funções prontas específicas de algoritmos para solução dos problemas elencados não são aceitos
- Não serão considerados algoritmos parcialmente implementados.
- Procedimento para a entrega:
  1. Submissão: via *Moodle*.
  2. Os nomes dos arquivos e das funções devem ser especificados considerando boas práticas de programação.
  3. Funções auxiliares, complementares aquelas definidas, podem ser especificadas e implementadas, se necessário.
  4. A solução deve ser devidamente modularizada e separar a especificação da implementação em arquivos *.h* e *.c* sempre que cabível.
  5. Os arquivos a serem entregues, incluindo aquele que contém *main()*, devem ser compactados (*.zip*), sendo o arquivo resultante submetido via *Moodle*.
  6. Você deve submeter os arquivos *.h*, *.c* e o *.pdf* (relatório) na raiz do arquivo *.zip*. Use os nomes dos arquivos *.h* e *.c* exatamente como pedido.
  7. Caracteres como acento, cedilha e afins não devem ser utilizados para especificar nomes de arquivos ou comentários no código.
- **Bom trabalho!**

## Ajudando um rato a sair do labirinto

Um conjunto de caminhos intercalados que são criados com o intuito de desorientar quem o está percorrendo é chamado de labirinto. Por definição, um labirinto é uma “construção de muitas passagens ou divisões, dispostas tão confusamente que com dificuldade se acha a saída” ([dicio.com.br](http://dicio.com.br)).

A Figura 1 ilustra um exemplo de labirinto.

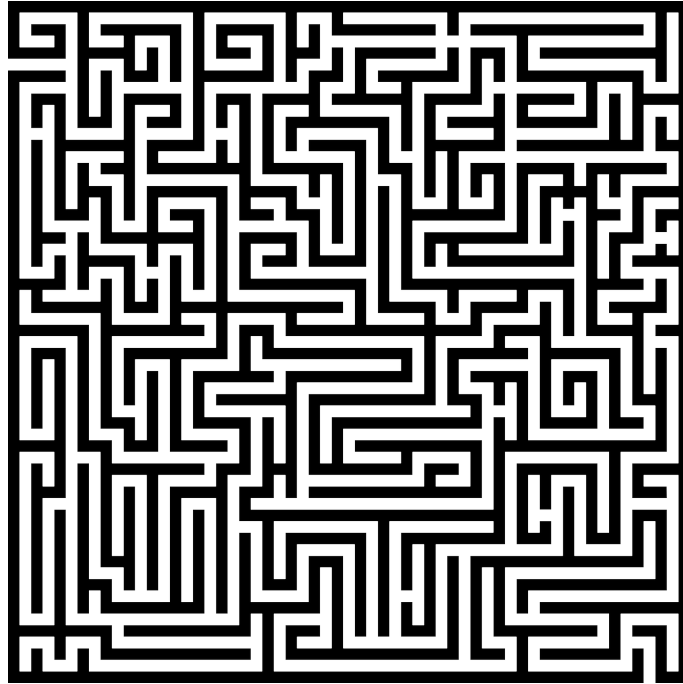


Figura 1: Exemplo de labirinto.

O objetivo deste trabalho é, dado um labirinto qualquer e a posição inicial de um rato, encontrar todos os caminhos possíveis até a saída. Para cada posição, o rato tem até quatro ações (desde que uma parede não o impeça): ir para direita, esquerda, para cima ou para baixo.

## Imposições e comentários gerais

Neste trabalho, as seguintes regras devem ser seguidas:

- Seu programa não pode ter *memory leaks*, ou seja, toda memória alocada pelo seu código deve ser corretamente liberada antes do final da execução. (Dica: utilize a ferramenta *valgrind* para se certificar de que seu código libera toda a memória alocada).
- Um grande número de *Warnings* ocasionará a redução na nota final.

## O que deve ser entregue

- Código fonte do programa em C (**bem indentado e comentado**).
- Documentação do trabalho (relatório<sup>1</sup>). A documentação deve conter:
  1. **Introdução:** descrição sucinta do problema a ser resolvido e visão geral sobre o funcionamento do programa.
  2. **Implementação:** descrição sobre a implementação do programa. **Não faça “print screens”** de telas. Ao contrário, procure resumir ao máximo a documentação, fazendo referência ao que julgar mais relevante. É importante, no entanto, que seja descrito o funcionamento das principais funções e procedimentos utilizados, bem como decisões tomadas relativas aos casos e detalhes de especificação que porventura estejam omissos no enunciado. Muito importante: os códigos utilizados na implementação devem ser inseridos na documentação.

---

<sup>1</sup>Exemplo de relatório: <https://www.overleaf.com/latex/templates/modelo-relatorio/vprmcsgmcdg>.

3. **Estudo de Complexidade:** estudo da complexidade do tempo de execução dos procedimentos implementados e do programa como um todo (notação  $O$ ), considerando entradas de tamanho  $n$ .
4. **Testes:** descrição dos testes realizados e listagem da saída (não edite os resultados).
5. **Análise:** deve ser feita uma análise dos resultados obtidos com este trabalho. Por exemplo, avaliar o tempo gasto de acordo com o tamanho do problema.
6. **Conclusão:** comentários gerais sobre o trabalho e as principais dificuldades encontradas em sua implementação.
7. **Bibliografia:** bibliografia utilizada para o desenvolvimento do trabalho, incluindo sites da Internet se for o caso.
8. **Formato:** PDF ou HTML.

## Como deve ser feita a entrega

Verifique se seu programa compila e executa na linha de comando antes de efetuar a entrega. Quando o resultado for correto, entregue via *Moodle* até 20/08/2023 até 23:55, um arquivo **.ZIP** com o nome e sobrenome do aluno. Esse arquivo deve conter: (i) os arquivos *.c* e *.h* utilizados na implementação, (ii) instruções de como compilar e executar o programa no terminal, e (iii) o relatório em **PDF**.

## Detalhes da implementação

Para atingir o seu objetivo, você deverá construir um Tipo Abstrato de Dados (TAD) **Labirinto** como representação de um labirinto que você quer analisar. O TAD **Labirinto** deverá implementar, pelo menos, as seguintes operações:

1. **alocarLabirinto:** aloca um (ou mais) TAD **Labirinto**.
2. **desalocarLabirinto:** desaloca um TAD **Labirinto**.
3. **leLabirinto:** inicializa o TAD **Labirinto** a partir de dados do **terminal**.
4. **acharSaida:** função **recursiva** que retorna o percurso que deve ser feito para sair do labirinto presente no TAD **Labirinto**.
5. **imprimePercursoNoLabirinto:** imprime o labirinto com o percurso encontrado.

Além do TAD **Labirinto**, deve ser implementado o TAD **Posicao** que armazena as coordenadas  $(x, y)$ . O TAD **Posicao** deve ser usado como parte do TAD **Percurso** que armazena a sequência de posições (TAD **Posicao**) avaliadas até a saída e um inteiro com o comprimento da sequência de posições. As funções do TAD **Posicao** e do TAD **Percurso** ficam a cargo do aluno.

**Para armazenar todos os possíveis caminhos, você deve usar um TAD Árvore.** Essa árvore não será binária, mas sim terá pelo menos quatro nós filhos (baixo, direita, cima e esquerda). A implementação de outros TADs fica a critério do aluno.

## Considerações

Você deve tomar cuidado para não adicionar a fila/pilha posições que já foram visitadas. Para isso, você deve usar o TAD **Labirinto** tanto para determinar quais as novas posições a serem visitadas quanto para evitar que sejam inseridas células já visitadas nas estruturas de dados. Fica a critério do aluno determinar a melhor forma de implementar isso. Também fica a critério do aluno criar uma estratégia de como gerenciar os caminhos conforme são feitas as chamadas para a fila e para a pilha.

Alocação de um ou mais TADs **Labirinto**, **Posicao**, **Percurso**, **Arvore** e outros TADs fica a critério do aluno. Os TADs devem ser implementados utilizando a separação interface no *.h* e implementação *.c* discutida em sala, bem como as convenções de tradução. Caso uma operação possa dar errado, devem ser definidos retornos com erro, tratados no corpo principal. **A alocação da TAD necessariamente deve ser feita de forma dinâmica.**

Ao avaliar os caminhos utilize a ordem direita, baixo, esquerda e cima para a estratégia com fila, pilha e recursão.

O código-fonte deve ser modularizado corretamente em pelo menos cinco arquivos: *tp.c*, *labirinto.h*, *labirinto.c*, *arvore.h* e *arvore.c*. O arquivo *tp.c* deve apenas invocar e tratar as respostas das funções e procedimentos definidos no arquivo *labirinto.h* e *arvore.h*. A separação das operações em funções e procedimentos está a cargo do aluno, porém, **não deve haver acúmulo** de operações dentro de uma mesma função/procedimento.

O limite de tempo para solução de cada caso de teste é de apenas **um segundo**. Além disso, o seu programa não pode ter *memory leaks*, ou seja, toda memória alocada pelo seu código deve ser corretamente liberada antes do final da execução. (Dica: utilize a ferramenta *valgrind* para se certificar de que seu código libera toda a memória alocada). *Warnings* ocasionará redução na nota final. Assim sendo, utilize suas habilidades de programação e de análise de algoritmos para desenvolver um algoritmo correto e rápido!

**Utilize o mesmo código do TP I para percorrer o labirinto. A árvore será usada para manter todos os caminhos percorridos.**

## Entrada

A entrada é dada por meio do terminal. Para facilitar, a entrada será fornecida por meio de arquivos.<sup>2</sup> A primeira linha especifica as dimensões do labirinto que não necessariamente é quadrado, logo serão informados dois valores: o número de linhas e o número de colunas respectivamente. **A saída do labirinto sempre será o canto inferior direito.** Após essas duas linhas, um caractere é fornecido para definir qual estratégia será usada, onde ‘g’ é usada para que seja impresso o maior caminho encontrado e ‘s’ para mostrar o menor. Por fim, é apresentada uma matriz de caracteres que reproduz o labirinto a ser analisado, sendo o caractere ‘\*’ representando as paredes do labirinto, ‘ ’ (espaço) um caminho e ‘M’ a origem de onde se sai. Abaixo um exemplo de entrada.

```
1 5 7
2 g
3 *****
4 *  M*  *
5 *  ***  *
6 *
7 *****
```

## Saída

A saída informada deve ser o tamanho do percurso até a saída e a impressão do labirinto com o caminho percorrido, para isso, utilize o caractere ‘.’ para informar os caminhos percorridos. Também pode ocorrer que não tenha saída no labirinto dada a posição que você se encontrava. Quando isso ocorrer, apresente a mensagem “EPIC FAIL!”.

## Exemplo de casos de teste

Exemplos de saídas esperadas dada diferentes entradas:

Entrada	Saída
<pre>7 11 g ***** *           * ***** * *M*   * * * * *** * *** *           * *****</pre>	EPIC FAIL!

<sup>2</sup>Para usar o arquivo como entrada no terminal, utilize `./executavel < nome_do_arquivo_de_teste`.

Entrada	Saída
<pre> 7 11 s ***** *           * ***** * *M*   *   * * *** * * * * ***** </pre>	<pre> 11 ***** *           * ***** * *M*   *   * *.*** * * * *..... ***** </pre>

Entrada	Saída
<pre> 7 11 g ***** *           * ***** * *M*   *   * * *** * * * * ***** </pre>	<pre> 15 ***** *           * ***** * *M*   *...* *.*** *.*. *..... ***** </pre>

A SAÍDA DA SUA IMPLEMENTAÇÃO DEVE SEGUIR EXATAMENTE A SAÍDA PROPOSTA.

## Diretivas de Compilação

As seguintes diretivas de compilação devem ser usadas (essas são as mesmas usadas no *Moodle*).

```

$ gcc -c arvore.c -Wall
$ gcc -c labirinto.c -Wall
$ gcc -c main.c -Wall
$ gcc arvore.o labirinto.o main.o -o exe -lm

```

## Avaliação de *leaks* de memória

Uma forma de avaliar se não há *leaks* de memória é usando a ferramenta *valgrind*. O *valgrind* é um *framework* de instrumentação para análise dinâmica de um código e é muito útil para resolver dois problemas em seus programas: **vazamento de memória e acesso a posições inválidas de memória** (o que pode levar a *segmentation fault*). Um exemplo de uso é:

```

1 gcc -g -o exe arquivo1.c arquivo2.c -Wall
2 valgrind --leak-check=full -s ./exe < casoteste.in

```

Espera-se uma saída com o fim semelhante a:

```

1 ==xxxxxx== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Para instalar no Linux, basta usar: `sudo apt install valgrind`.

O SEU CÓDIGO SERÁ TESTADO NOS COMPUTADORES DO LABORATÓRIO  
(AMBIENTE LINUX)

## PONTO EXTRA

Será concedido 0,1 extra para quem imprimir todos os nós da árvore por nível. Para esse caso será fornecido o caractere 'f' ao invés do 'g' e do 's'. Abaixo um exemplo:

Entrada	Saída
<pre> 7 11 f ***** *           * ***** * *M*      *  * * *** * * * * ***** </pre>	<pre> Nivel 1: (3, 1) Nivel 2: (4, 1) Nivel 3: (5, 1) Nivel 4: (5, 2) Nivel 5: (5, 3) Nivel 6: (5, 4) Nivel 7: (5, 5) Nivel 8: (4, 5) (5, 6) Nivel 9: (3, 5) (5, 7) Nivel 10: (3, 4) (4, 7) (5, 8) Nivel 11: (3, 3) (3, 7) (5, 9) Nivel 12: (3, 8) (4, 9) (5, 10) Nivel 13: (3, 9) (3, 9) Nivel 14: (4, 9) (2, 9) (2, 9) (3, 8) Nivel 15: (5, 9) (1, 9) (1, 9) (3, 7) Nivel 16: (5, 10) (5, 8) (1, 8) (1, 8) (4, 7) Nivel 17: (1, 7) (1, 7) Nivel 18: (1, 6) (1, 6) Nivel 19: (1, 5) (1, 5) Nivel 20: (1, 4) (1, 4) Nivel 21: (1, 3) (1, 3) Nivel 22: (1, 2) (1, 2) Nivel 23: (1, 1) (1, 1) </pre>