BCC202 – Estruturas de Dados I

Departamento de Computação - Universidade Federal de Ouro Preto - MG
Professor: **Pedro Silva** (`www.decom.ufop.br/`)

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

- Submissão com data e hora de entrega disponíveis na plataforma da disciplina. O que vale é o horário do Moodle, e não do *seu*, ou do *meu* relógio!!!

- Clareza, identação 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 o trabalho para complementar a avaliação;

- O trabalho é em grupo de até 2 (duas) pessoas.

- Será aceito trabalhos após a data de entrega, todavia com um decréscimo de 0,05 a cada 10min.

- 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.

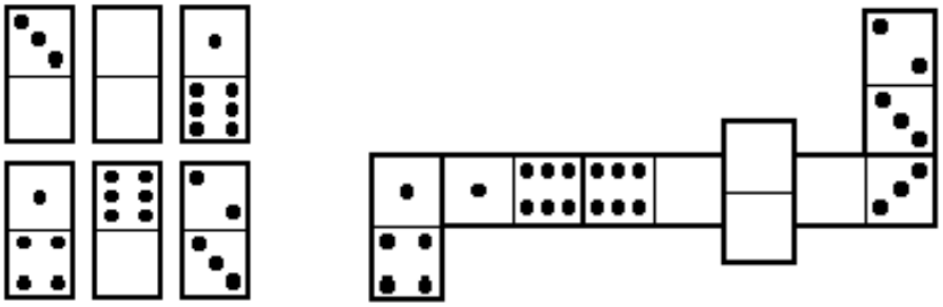
- Procedimento para a entrega:.

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!

Jogo de Dominó

O jogo de dominós é um passatempo clássico, conhecido por sua simplicidade e pelo desafio estratégico que oferece. Nele, peças com dois valores devem ser organizadas em uma sequência, de forma que os valores de peças adjacentes sejam iguais. No entanto, dado um conjunto de peças, surge uma questão interessante: será que é sempre possível formar uma sequência válida com todas elas? Um exemplo de resposta para esta pergunta é visto na Figura 1.



(a) (b)

Figura 1: Exemplo de um conjunto de peças (a) e uma possível solução de sequência válida (b).

Esse problema, que parece simples à primeira vista, envolve conceitos importantes de lógica e manipulação de estruturas de dados, como listas. A habilidade de verificar se uma sequência válida pode ser formada é um exercício que não apenas desafia sua capacidade de raciocínio, mas também conecta conceitos fundamentais da programação a um problema prático e reconhecível do cotidiano.

implementando uma solução para determinar se um conjunto de peças de dominó pode ser organizado em uma formação válida.

Na prática, sua tarefa é dado um conjunto de peças de dominó, escrever um programa que determine se é possível organizar todas as peças recebidas em sequência, obedecendo às regras do jogo de dominó. Lembrando que cada peça tem dois valores X e Y, com X e Y variando de 0 a 6 (X pode ser igual a Y).

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 identado e comentado**).
- Documentação do trabalho (relatório[1]). A documentação deve conter:
  1. **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.
  2. **Impressões gerais**: descreva o seu processo de implementação deste trabalho. Aponte coisas que gostou bem como aquelas que o desagradou. Avalie o que o motivou, conhecimentos que adquiriu, entre outros.

[1]Exemplo de relatório: `https://www.overleaf.com/latex/templates/modelo-relatorio/vprmcsdgmcgd`.

3. **Análise**: deve ser feita uma análise dos resultados obtidos com este trabalho.
4. **Conclusão**: comentários gerais sobre o trabalho e as principais dificuldades encontradas em sua implementação.
5. **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 a data disponível na plataforma de entraga 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 via terminal, e (iii) o relatório em **PDF**.

Detalhes da implementação

Para atingir o seu objetivo, você deverá construir um Tipo Abstrato de Dados `Domino` como representação da **lista** de peças do jogo dominó. O TAD deverá implementar, pelo menos, as seguintes operações:

1. `DominoCria`: aloca um TAD `Domino` que é uma lista.
2. `DominoDestroi`: desaloca um TAD `Domino`.
3. `DominoAdicionaPeca`: adiciona uma peça lida ao TAD `Domino`.
4. `DominoImprime`: função que imprime as peças do TAD `Domino` de acordo com a sua posição.
5. `DominoEhPossivelOrganizar`: função que tenta organizar as peças do TAD `Domino` em uma ordem válida. Ela retorna `true` se for possível e *false*, caso contrário. **Esta função pode ou não ser recursiva!**

A representação de uma peça dentro do TAD `Domino` fica a cargo do aluno, isso pode ser feito por um outro TAD (não deve ser criado outro arquivo), por dois inteiro, vetor, etc. Essa decisão precisa estar detalhada e explicada no relatório. Outras funções também podem ser criadas

Alocação de um ou mais TADs `Domino` fica a critério do aluno. Contudo, o dominó precisa ser representado e manipulado como uma **Lista Encadeada (dupla ou simples)**.

O TAD deve ser implementado utilizando a separação interface no *.h* e implementação *.c* discutida em sala, bem como as convenções de tradução. Caso a 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**.

O código-fonte deve ser modularizado corretamente em três arquivos: *tp.c*, *domino.h* e *domino.c*. O arquivo *tp.c* deve apenas invocar e tratar as respostas das funções e procedimentos definidos no arquivo *domino.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 a redução na nota final. Assim sendo, utilize suas habilidades de programação e de análise de algoritmos para desenvolver um algoritmo correto e rápido!

Entrada

A entrada é dada por meio do terminal e é composta de vários conjuntos de teste. Para facilitar, a entrada será fornecida por meio de arquivos. [2] A primeira linha de um conjunto de testes contém um número inteiro N que indica a quantidade de peças do conjunto. As N linhas seguintes contêm, cada uma, a descrição de uma peça. Uma peça é descrita por dois inteiros X e Y que representam os valores de cada lado da peça. O final da entrada é indicado por $N = 0$.

As restrições do problema são:

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

- $0 \leq X \leq 6$.
- $0 \leq Y \leq 6$.
- $0 \leq N \leq 28$ ($N = 0$ apenas para indicar o final da entrada).

Saída

Para cada conjunto de teste da entrada seu programa deve produzir até quatro linhas na saída. A primeira linha deve conter um identificador do conjunto de teste, no formato "*Teste n*", onde n é numerado a partir de 1. A segunda linha deve conter a expressão "*YES*" se for possível organizar todas as peças em uma formação válida ou a expressão "*NO*" caso contrário. Se caso seja possível organizá-las, haverá uma terceira linha que apresenta a sequência de alocação das peças (separadas por "|"). Se não for possível organizá-las, nada deve ser impresso. A última linha deve ser deixada em branco. A grafia da saída deve ser seguida rigorosamente.

Não é preciso apresentar todas as soluções, somente a primeira válida encontrada!

Exemplo de um caso de teste

Exemplo da saída esperada dada uma entrada:

Entrada	Saída
3 0 1 2 1 2 1 2 1 1 0 0 6 3 0 0 0 1 6 4 1 0 6 2 3 0	Test 1: YES 01|12|21 Test 2: NO Test 3: YES 41|16|60|00|03|32

Entrada	Saída
3 3 0 1 6 1 0 1 1 3 0	Test 1: YES 30|01|16 Test 2: YES 13

Como é um problema combinatorial, a saída válida que você encontrar pode ser diferente da saída que foi reportada como a esperada nos arquivos de saída. Para este trabalho em específico, será fornecido posteriormente um corretor para se adequar a realidade das saídas deste TP.

Diretivas de Compilação

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

```
$ gcc -c domino.c -Wall
$ gcc -c tp.c -Wall
$ gcc domino.o tp.o -o exe
```

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 *.c -Wall
2 valgrind --leak-check=full -s ./exe < casoteste.in
```

Espera-se uma saída com o fim semelhante a:

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

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

Referências

[1] Andrew Ilachinski. *Cellular automata: a discrete universe*. World Scientific, 2001.

[2] Martin Gardner. *Mathematical games: The fantastic combinations of john conway's new solitaire game "life"*. Scientific American, 223(4):120–123, 1970.