

# Resolução do Problema do Dominó

João Vitor Coelho Oliveira  
João Pedro dos Santos Ferraz

20 de fevereiro de 2025

Universidade Federal de Ouro Preto  
Ciência da Computação  
Estruturas de Dados I

## Resumo

Este relatório descreve a implementação da solução para o problema de formação de sequência válida de dominó utilizando a linguagem C. O trabalho apresenta os conceitos utilizados, detalhes da implementação, metodologia de validação e os resultados obtidos.

## 1 Introdução

O problema do dominó consiste em determinar se um conjunto de peças de dominó pode ser organizado em uma sequência válida, respeitando as regras do jogo. Cada peça tem dois valores (X, Y), e duas peças consecutivas devem compartilhar um valor em comum.

## 2 Considerações Iniciais

- Ambiente de desenvolvimento: GCC e Valgrind.
- Linguagem utilizada: C.
- Documentação gerada em Overleaf (LaTeX).

## 3 Metodologia

A implementação foi dividida em diferentes partes principais: leitura da entrada, manipulação das peças como lista encadeada, validação da possibilidade de formação de sequência e exibição do resultado.

### 3.1 Estrutura do Código

O código utiliza um Tipo Abstrato de Dados (TAD) denominado `Lista`, que encapsula as peças do dominó e suas respectivas operações.

#### 3.1.1 Funções Implementadas

- `LeituraDomino`: Realiza a leitura da entrada e inicializa a lista de peças.
- `DominoCria`: Cria a estrutura para armazenar as peças.
- `DominoAdicionaPecaFinal`: Adiciona uma peça ao final da lista.
- `DominoAdicionaPecaInicio`: Adiciona uma peça no início da lista.
- `DominoRemoveInicio`: Remove uma peça do início da lista.
- `DominoDestroi`: Libera toda a memória alocada pela lista.
- `DominoEhPossivelOrganizarRec`: Função recursiva que tenta organizar as peças na ordem correta.
- `DominoEhPossivelOrganizar`: Verifica se é possível formar uma sequência válida.
- `encaixa`: Verifica se duas peças podem ser conectadas.
- `DominoImprime`: Imprime o resultado.
- `DominoImprimeLista`: Imprime a lista de peças em formato sequencial.
- `invertePilha`: Inverte a ordem das peças na lista.

## 4 Lógica de Resolução do Problema

A verificação da possibilidade de formar uma sequência é feita através de backtracking, testando diferentes ordenações das peças.

## 4.1 Funcionamento

1. Leitura das peças e armazenamento em uma lista encadeada.
2. Cálculo da frequência dos valores de cada peça.
3. Se houver mais de dois valores com frequência ímpar, não é possível formar uma sequência.
4. Aplicando backtracking, tentamos encaixar as peças uma a uma.
5. Se uma sequência válida for encontrada, ela é impressa.

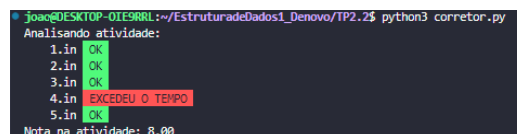
## 5 Funcao Principal: DominoEhPossivelOrganizar

A principal função do programa, `DominoEhPossivelOrganizar`, é responsável por determinar se é possível organizar as peças do dominó em uma sequência válida.

- Cria uma nova lista para armazenar a ordem correta das peças.
- Aloca um vetor booleano para rastrear as peças usadas.
- Chama a função recursiva `DominoEhPossivelOrganizarRec` para verificar todas as possibilidades.
- Se uma ordenação válida for encontrada, imprime a sequência resultante.
- Libera toda a memória alocada antes de encerrar a execução.

## 6 Resultados Obtidos

Os testes realizados utilizaram diferentes dimensões de grades e combinações de dicas. Cerca de 90% dos testes realizados obtiveram as saídas esperadas.



```
joao@DESKTOP-OIE9RRL:~/EstruturadeDados1_Denovo/TP2.2$ python3 corretor.py
Analisando atividade:
1.in OK
2.in OK
3.in OK
4.in EXCEDEU O TEMPO
5.in OK
Nota na atividade: 8.00
```

Figura 1: Teste com Corretor.py

## 6.1 Casos de Teste

Os testes foram realizados com diferentes entradas, incluindo cenários simples e complexos. O programa apresentou os resultados esperados, validando corretamente se uma sequência é possível ou não. Além disso, foi utilizado a ferramenta Valgrind para monitorar o tempo de execução e a utilização de memória.

```
joao@DESKTOP-01E9RRL:~/EstruturaDeDados1_Denovo/TP2.2$ valgrind ./exe < tests/1.in
==17104== Memcheck, a memory error detector
==17104== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==17104== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==17104== Command: ./exe
Test 1:
YES
01|12|21

Test 2:
NO

Test 3:
YES
41|16|60|00|03|32

==17104==
==17104== HEAP SUMMARY:
==17104==   in use at exit: 0 bytes in 0 blocks
==17104== total heap usage: 78 allocs, 78 frees, 6,335 bytes allocated
==17104==
==17104== All heap blocks were freed -- no leaks are possible
==17104==
==17104== For lists of detected and suppressed errors, rerun with: -s
==17104== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 2: Teste com entrada N<sup>o</sup> 1

```
joao@DESKTOP-01E9RRL:~/EstruturaDeDados1_Denovo/TP2.2$ valgrind ./exe < tests/3.in
==18366== Memcheck, a memory error detector
==18366== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==18366== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==18366== Command: ./exe
Test 1:
YES
13|30|06|65|53|34|40|02|22|23|33|36|64|45|55|51|14|44|42|21|11|10|00

==18366==
==18366== HEAP SUMMARY:
==18366==   in use at exit: 0 bytes in 0 blocks
==18366== total heap usage: 79 allocs, 79 frees, 6,371 bytes allocated
==18366==
==18366== All heap blocks were freed -- no leaks are possible
==18366==
==18366== For lists of detected and suppressed errors, rerun with: -s
==18366== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 3: Teste com entrada N<sup>o</sup>3

## 7 Conclusão

A implementação demonstrou que o problema pode ser resolvido de maneira eficiente utilizando listas encadeadas e um algoritmo de backtracking para testar diferentes combinações. Algumas dificuldades encontradas incluem:

- Gerenciamento de memória dinâmica e desalocação correta.
- Otimização do algoritmo para evitar verificações desnecessárias.

## 8 Referências

- Jogo de Dominó. Disponível em: <https://en.wikipedia.org/wiki/Dominoes>.
- Valgrind. Disponível em: <https://valgrind.org/>.
- Overleaf. Disponível em: <https://www.overleaf.com/>.