

# Introdução

## Conceitos básicos e motivação

Prof. Marcelo de Souza

45EST – Algoritmos e Estruturas de Dados  
Universidade do Estado de Santa Catarina



## Leitura principal:

- ▶ Capítulo 2 de Edelweiss e Galante (2009)<sup>1</sup> – Conceitos básicos.

## Leitura complementar:

- ▶ Capítulo 1 de Pereira (2008)<sup>2</sup> – Introdução.

---

<sup>1</sup>Nina Edelweiss e Renata Galante (2009). *Estruturas de Dados: Volume 18*. Bookman Editora.

<sup>2</sup>Silvio do Lago Pereira (2008). *Estruturas de Dados Fundamentais: Conceitos e Aplicações*.



Algumas definições:

- ▶ **Algoritmo:** sequência de passos para realizar uma tarefa.
- ▶ **Estrutura de dados:** forma sistemática de **organizar** e **acessar** os dados.

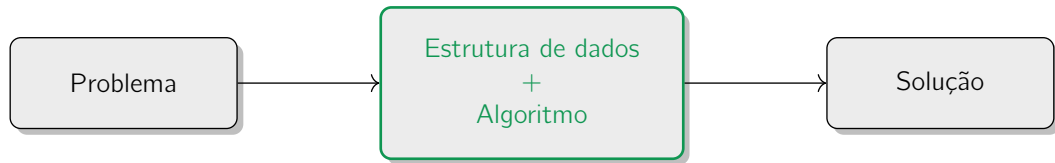


Algumas definições:

- ▶ **Algoritmo:** sequência de passos para realizar uma tarefa.
- ▶ **Estrutura de dados:** forma sistemática de **organizar** e **acessar** os dados.

**Objetivo:** escolher os melhores elementos para resolver um problema. Ou seja, buscando

- ▶ eficácia: resolve o problema;
- ▶ eficiência: usa a menor quantidade de recursos possível.





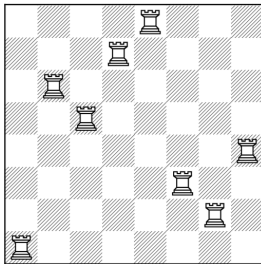
Para isso, precisamos **conhecer**/**dominar**:

- ▶ alguma linguagem de programação (nesta disciplina, Java);
- ▶ orientação a objetos;
- ▶ análise da complexidade de algoritmos e classes de complexidade;
- ▶ estruturas de dados fundamentais (*arrays*, listas encadeadas, ... );
- ▶ estruturas abstratas de dados (filas, pilhas, dicionários, árvores, grafos, ... );
- ▶ principais algoritmos que operam sobre essas estruturas;
- ▶ técnicas algorítmicas (recursividade, divisão e conquista, ... );
- ▶ experiência de aplicações práticas a problemas concretos.

# Exemplo prático (n-rooks)

## Problema das n torres

**Problema:** posicionar  $n$  torres em um tabuleiro de xadrez  $n \times n$ , de tal forma que elas não se ataquem (a torre se movimenta na vertical e na horizontal).



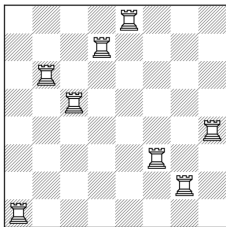
**Tarefa:** dada uma possível solução (posição das torres no tabuleiro), checar quantas violações existem, i.e. quantos ataques ocorrem.

# Exemplo prático (n-rooks)

## Proposta 1 (matriz)

Representar a solução como uma matriz de inteiros  $M = (m_{ij})$  de tamanho  $n \times n$ , tal que  $m_{ij} = 1$  caso haja uma torre posicionada na célula  $(i, j)$ , e  $m_{ij} = 0$ , caso contrário.

### Exemplo:



```
int[] [] board = {  
    {0, 0, 0, 0, 1, 0, 0, 0},  
    {0, 0, 0, 1, 0, 0, 0, 0},  
    {0, 1, 0, 0, 0, 0, 0, 0},  
    {0, 0, 1, 0, 0, 0, 0, 0},  
    {0, 0, 0, 0, 0, 0, 0, 1},  
    {0, 0, 0, 0, 0, 1, 0, 0},  
    {0, 0, 0, 0, 0, 0, 1, 0},  
    {1, 0, 0, 0, 0, 0, 0, 0}};
```

Implementar uma classe `NRooks` que faça a leitura de várias soluções, com diferentes tamanhos de tabuleiro, e conte o número de violações (método `violations`).

Dada a classe principal do projeto (NRooks):

```
1 public class NRooks {  
2     public static void main (String[] args) { /* ... */}  
3 }
```

Podemos compilar e executar essa classe na linha de comando:

```
1 $ javac NRooks.java  
2 $ java NRooks
```

Ou ainda, usando:

```
1 $ javac NRooks.java && java NRooks
```

Ao usar alguma IDE (e.g., VS Code) que compila automaticamente, o bytecode é geralmente copiado para o diretório bin. Podemos executar o programa a partir da raiz do projeto:

```
1 $ java -cp bin/ NRooks
```





Ao executar o programa, podemos passar argumentos na chamada:

```
1 $ java -cp bin/ NRooks Brasil 10
```

E fazer a leitura desses valores no método main (parâmetro args):

```
1 public class NRooks {  
2     public static void main (String[] args) {  
3         System.out.println(args[0]); // imprime a string "Brasil"  
4         System.out.println(args[1]); // imprime a string "10"  
5     }  
6 }
```

Podemos definir um arquivo como entrada padrão do sistema (`System.in`):

```
1 $ java -cp bin/ NRooks < input.txt
```

Ao ler valores (com `Scanner`), o arquivo é lido (em vez do teclado):

```
1 public class NRooks {  
2     public static void main (String[] args) {  
3         Scanner s = new Scanner(System.in);  
4         while(s.hasNext()) System.out.println(s.next());  
5     }  
6 }
```

Também podemos redirecionar a saída do programa para um arquivo:

```
1 $ java -cp bin/ NRooks < input.txt > output.txt
```

O que facilita a comparação de duas saídas (para verificação da correção):

```
1 $ diff output1.txt output2.txt
```

## Exemplo prático (n-rooks)

### Proposta 1 (matriz)

O arquivo `inputMatrix.txt` contém 1000 soluções (uma por linha). Primeiro é fornecido o tamanho  $n \in [8, 500]$ , seguido do valor de cada célula da matriz:

[illegible]



# Exemplo prático (n-rooks)

## Proposta 1 (matriz)

Leitura das soluções e chamada ao método violations:

```
1  public static void evalMatrix() {
2      Scanner s = new Scanner(System.in);
3      while(s.hasNext()) {
4          int n = s.nextInt();
5          int[] [] board = new int[n] [n];
6          for(int i = 0; i < n; i++)
7              for(int j = 0; j < n; j++)
8                  board[i] [j] = s.nextInt();
9          System.out.println(violations(board) + " ");
10     }
11 }
```



# Exemplo prático (n-rooks)

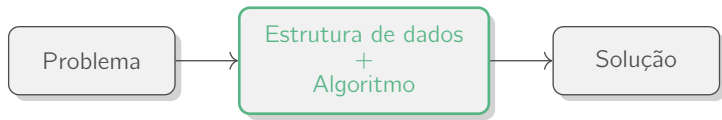
## Proposta 1 (matriz)

O método `violations` conta o número de ataques nas linhas e colunas. Seja  $m$  o número de torres em uma mesma linha/coluna, o número de ataques é dado por  $m(m - 1)/2$ :

```
1  public static int violations(int[] [] board) {
2      int amount = 0;
3      for(int c1 = 0; c1 < board.length; c1++) {
4          int foundRow = 0;
5          int foundCol = 0;
6          for(int c2 = 0; c2 < board[c1].length; c2++) {
7              if(board[c1][c2] == 1) foundRow += 1;
8              if(board[c2][c1] == 1) foundCol += 1;
9          }
10         amount += (foundRow * (foundRow - 1)) / 2;
11         amount += (foundCol * (foundCol - 1)) / 2;
12     }
13     return amount;
14 }
```

# Exemplo prático (n-rooks)

## Proposta 1 (matriz)



Quão boa é nossa solução para o problema (estrutura de dados + algoritmo)?

- ▶ Tempo de execução: ~**25 s**;
- ▶ Espaço de armazenamento: ~**164 MB** (inputMatrix.txt).

# Exemplo prático (n-rooks)

## Proposta 1 (matriz)



Quão boa é nossa solução para o problema (estrutura de dados + algoritmo)?

- ▶ Tempo de execução: ~**25 s**;
- ▶ Espaço de armazenamento: ~**164 MB** (inputMatrix.txt).

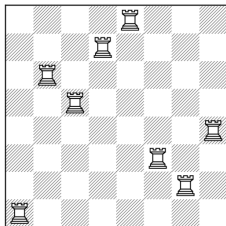
**Podemos fazer melhor?**

# Exemplo prático (n-rooks)

## Proposta 2 (array)

Sim! Assumimos que cada torre ficará em uma linha diferente, e usamos um *array*  $A = (a_i)$  de tamanho  $n$ , tal que  $a_i \in [n]$  indica a coluna onde a torre da linha  $i$  estará posicionada.

### Exemplo:



—————→ `int[] board = {4, 3, 1, 2, 7, 5, 6, 0};`

Com isso, temos uma representação mais simples e com menor possibilidade de violações (não permite torres na mesma linha). Essa estrutura admite algoritmos mais eficientes?





# Exemplo prático (n-rooks)

## Proposta 2 (*array*)

O arquivo `inputArray.txt` contém as mesmas 1000 soluções anteriores (uma por linha). Primeiro é fornecido o tamanho  $n \in [8, 500]$ , seguido do valor de cada posição do *array*:

```
1 360 193 278 299 190 81 280 138 250 121 347 56 146 214 320 279 254 354 256 217 17 115 348 ...
2 309 65 69 182 258 123 297 101 97 53 128 229 269 119 186 155 233 226 50 80 195 9 36 98 ...
3 168 144 67 155 102 24 81 27 136 8 136 136 62 157 143 69 69 122 150 53 94 87 60 125 126 ...
4 147 44 81 44 57 92 85 96 84 119 15 62 16 113 98 123 57 30 32 141 131 57 41 9 18 103 6 ...
5 298 103 235 93 264 85 279 137 40 74 140 14 214 296 163 27 121 48 40 8 58 85 85 191 215 ...
6 28 25 19 7 17 16 8 4 2 23 7 5 3 5 4 19 18 27 20 16 3 11 12 20 27 3 14 11 25
7 54 47 45 16 35 45 32 3 22 53 53 37 44 36 39 33 14 24 41 29 37 39 18 18 43 33 27 22 18 17 ...
8 39 18 30 34 10 4 29 30 35 2 23 18 23 23 34 4 30 6 13 36 38 17 19 32 11 35 18 18 7 37 21 ...
9 180 177 107 8 48 56 99 17 8 92 24 146 142 157 41 155 98 134 174 166 2 25 114 43 14 104 ...
10 285 191 182 116 129 44 105 257 18 202 172 274 129 40 105 126 258 153 114 70 8 226 157 ...
11 289 102 110 94 200 260 252 172 274 283 102 277 177 116 147 196 81 84 155 215 98 141 217 ...
12 ...
```



# Exemplo prático (n-rooks)

## Proposta 2 (array)

Leitura das soluções e chamada ao método violations:

```
1  public static void evalArray(int mode) {
2      Scanner s = new Scanner(System.in);
3      while(s.hasNext()) {
4          int n = s.nextInt();
5          int[] board = new int[n];
6          for(int i = 0; i < n; i++)
7              board[i] = s.nextInt();
8          System.out.print(violations(board) + " ");
9      }
10 }
```



# Exemplo prático (n-rooks)

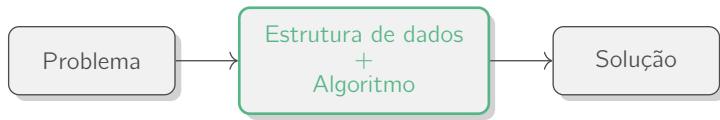
## Proposta 2 (*array*)

O método `violations` conta o número de valores repetidos no *array*, i.e. o número de torres posicionadas em uma mesma coluna (que corresponde ao número de ataques):

```
1  public static int violations(int[] board) {
2      int amount = 0;
3      for(int i = 0; i < board.length - 1; i++)
4          for(int j = i + 1; j < board.length; j++)
5              if(board[i] == board[j])
6                  amount++;
7      return amount;
8  }
```

# Exemplo prático (n-rooks)

## Comparação



Abordagem baseada em **matriz**:

- ▶ Tempo de execução: ~**25 s**;
- ▶ Espaço de armazenamento: ~**164 MB** (`inputMatrix.txt`).

Abordagem baseada em **array**:

- ▶ Tempo de execução: ~**0,2 s**;
- ▶ Espaço de armazenamento: ~**900 kB** (`inputArray.txt`).



# Tipos abstratos de dados

## Definições

Linguagens de programação fornecem:

- ▶ tipos de dados **primitivos**: inteiro, real, lógico, ...
- ▶ tipos de dados **estruturados**: arranjos, registros, sequências, ...



# Tipos abstratos de dados

## Definições

Linguagens de programação fornecem:

- ▶ tipos de dados **primitivos**: inteiro, real, lógico, ...
- ▶ tipos de dados **estruturados**: arranjos, registros, sequências, ...

Com esses recursos, podemos criar um **TAD** (**tipo abstrato de dados**), definindo:

- ▶ os valores e a **organização** dos dados;
- ▶ as **operações** sobre esses dados



# Tipos abstratos de dados

## Definições

Linguagens de programação fornecem:

- ▶ tipos de dados **primitivos**: inteiro, real, lógico, ...
- ▶ tipos de dados **estruturados**: arranjos, registros, sequências, ...

Com esses recursos, podemos criar um **TAD** (**tipo abstrato de dados**), definindo:

- ▶ os valores e a **organização** dos dados;
- ▶ as **operações** sobre esses dados

Um TAD é dito **abstrato** porque sua definição não exige uma implementação concreta, i.e.

- ▶ um TAD define uma API (*application programming interface*);
- ▶ para um TAD, podem existir várias implementações distintas;

Usaremos **interfaces** e **classes** para criar TADs em Java.



# Tipos abstratos de dados

## Representação física

Para criar TADs, usaremos dois tipos de estruturas fundamentais (arranjos ou encadeamento), que diferem uma da outra segundo sua representação física.





# Tipos abstratos de dados

## Representação física

Para criar TADs, usaremos dois tipos de estruturas fundamentais (arranjos ou encadeamento), que diferem uma da outra segundo sua representação física.

### **Arranjos**

- ▶ Exemplos: vetores e matrizes;
- ▶ Contiguidade física: valores armazenados sequencialmente na memória;
- ▶ Vantagem: acesso aleatório (e rápido);
- ▶ Desvantagem: espaço físico estático.



# Tipos abstratos de dados

## Representação física

Para criar TADs, usaremos dois tipos de estruturas fundamentais (arranjos ou encadeamento), que diferem uma da outra segundo sua representação física.

### **Arranjos**

- ▶ Exemplos: vetores e matrizes;
- ▶ Contiguidade física: valores armazenados sequencialmente na memória;
- ▶ Vantagem: acesso aleatório (e rápido);
- ▶ Desvantagem: espaço físico estático.

### **Encadeamento**

- ▶ Exemplos: listas simplesmente e duplamente encadeadas;
- ▶ Alocação dinâmica e não sequencial de memória;
- ▶ Vantagem: maleabilidade;
- ▶ Desvantagem: acesso serial.

