

# Material de Apoio – Aula 1

---

## 1. Introdução ao Coding Tank

- Objetivo do encontro: revisar rapidamente ferramentas de ordenação & prioridade para que você já saia pronto(a) para o workshop de amanhã.
- Expectativa: não escreveremos código; foco em ideias, trade-offs e artefatos visuais que guiarão seu projeto.
- Mapa dos 3 dias:
  - **1** Conceitos & prática guiada → **2** Prototipagem em grupo → **3** Apresentação e Perguntas (dos avaliadores).

## 2. Algoritmos de Ordenação

**\*\*Estrutura dos blocos (para cada algoritmo)\*\***

Descrição → Análise → Pros/Cons → Cenários → Pseudocódigo → Diagrama ASCII

### 2.1 Insertion Sort

#### 2.1.1 Breve descrição

Insere cada elemento na posição correta de uma porção já ordenada.

#### 2.1.2 Análise

- Tempo:  $O(n^2)$  worst/avg,  $O(n)$  best (lista quase ordenada)
- Memória:  $O(1)$  in-place

#### 2.1.3 Pros/Cons

✓ Prós

✗ Contras

---

Simples, estável

Lento para  $n$  grande

---

Ótimo p/ listas quase ordenadas

Não paralelizável

#### 2.1.4 Cenários

- Ordenar pequenos buckets depois de Quick Sort (Timsort).
- Autocomplete por frequência recente (lista curta).

#### 2.1.5 Passo-a-passo

1. Considere o primeiro elemento do vetor como "sublista ordenada" (tamanho 1).
2. Para cada posição  $i$  de 1 até  $n - 1$ :

1. Armazene  $A[i]$  em uma variável *chave*.
  2. Compare *chave* com os elementos à esquerda, começando por  $A[i-1]$ .
  3. Enquanto  $j \geq 0$  e  $A[j] > \text{chave}$ , desloque  $A[j]$  uma posição à direita ( $A[j+1] = A[j]$ ) e decamente  $j$ .
  4. Quando encontrar posição correta, insira *chave* em  $A[j+1]$ .
3. Repita até que todos os elementos tenham sido percorridos; a lista estará ordenada no final da última iteração.

### 2.1.6 Pseudocódigo

```

for i ← 1 ... n-1
  chave ← A[i]
  j ← i-1
  while j ≥ 0 and A[j] > chave
    A[j+1] ← A[j]
    j ← j-1
  A[j+1] ← chave

```

### 2.1.7 Diagrama ASCII

```

[ 5 | 3 | 8 | 2 ]
^
passo 1 → [ 3 5 | 8 2 ]
passo 2 → [ 3 5 8 | 2 ]
passo 3 → [ 2 3 5 8 ]

```

## 2.2 Selection Sort

### 2.2.1 Breve descrição

Seleciona o menor elemento e troca com a posição corrente.

### 2.2.2 Análise

- Tempo:  $O(n^2)$  sem caso melhor
- Memória:  $O(1)$

### 2.2.3 Pros/Cons

✓ Prós ✗ Contras

---

Simples, mínima troca ( $n-1$ )    Sempre lento, instável

### 2.2.4 Cenários

- Embedded systems com trocas custosas em EEPROM

### 2.2.5 Passo-a-passo

#### 1. Divida o vetor em duas partes

1. Sublista ordenada (inicialmente vazia, à esquerda).
2. Sublista não-ordenada (o restante do vetor).

#### 2. Percorra posições de 0 até $n - 2$ (índice $i$ ):

1. Assuma  $i$  como posição do menor elemento ( $\text{min} = i$ ).
2. Varra a sublista não-ordenada da posição  $i + 1$  até  $n - 1$ :
  - Se encontrar um valor menor que  $A[\text{min}]$ , atualize  $\text{min}$  para esse índice.
3. Troque  $A[i]$  com  $A[\text{min}]$ .
  - Isso "seleciona" e coloca o menor elemento restante na posição correta da sublista ordenada.

3. Repita até que  $i$  alcance  $n - 2$ . Ao final, todos os elementos estarão na ordem crescente.

### 2.2.6 Pseudocódigo

```
for i ← 0 ... n-2
  min ← i
  for j ← i+1 ... n-1
    if A[j] < A[min] then min ← j
  swap A[i] ↔ A[min]
```

### 2.2.7 Diagrama ASCII

```
[ 4 1 3 2 ]
^min
→ [ 1 | 4 3 2 ]
   [ 1 | 4 3 2 ]
       ^min
→ [ 1 2 | 3 4 ]
       ^min
   [ 1 2 3 | 4 ]
```

## 2.3 Merge Sort

### 2.3.1 Breve descrição

Divide a lista até unidades, depois mescla ordenadamente.

### 2.3.2 Análise

- Tempo:  $O(n \log n)$  todos os casos
- Memória:  $O(n)$  extra (arrays auxiliares)

### 2.3.3 Pros/Cons

✓ Prós

✗ Contas

---

Estável, garante  $O(n \log n)$     Custo de memória, não in-place em arrays

### 2.3.4 Cenários

- Ordenação externa em arquivos grandes

### 2.3.5 Passo-a-passo

#### 1. Dividir

- Se o vetor tiver mais de um elemento, divida-o ao meio (índices `low ... mid` e `mid+1 ... high`).

#### 2. Recursão nas metades

1. Chame Merge Sort recursivamente para ordenar a metade esquerda.
2. Chame Merge Sort recursivamente para ordenar a metade direita.

#### 3. Mesclar (merge)

1. Crie dois ponteiros, um para cada metade já ordenada.
2. Compare os elementos apontados: copie o menor para um vetor auxiliar e avance o ponteiro correspondente.
3. Continue até que uma das metades seja totalmente copiada.
4. Copie os elementos restantes da outra metade.

#### 4. Copiar de volta

- Substitua a fatia original pelos valores do vetor auxiliar, agora em ordem crescente.

#### 5. Resultado

- Ao voltar das chamadas recursivas, cada nível entrega um segmento maior já ordenado; no topo da recursão, o vetor inteiro estará ordenado.

### 2.3.6 Pseudocódigo

```
mergeSort(A):  
  if |A| ≤ 1 return A  
  L,R ← split(A)  
  return merge(mergeSort(L), mergeSort(R))
```

### 2.3.7 Diagrama ASCII

```
  [8 3 5 2]  
→ [8 3] [5 2]  
→ [3 8] [2 5]  
→ [2 3 5 8]
```

## 2.4 Quick Sort

### 2.4.1 Breve descrição

Escolhe pivô, particiona menores ↔ maiores, recursão.

### 2.4.2 Análise

- Tempo:  $O(n \log n)$  avg,  $O(n^2)$  worst (lista ordenada, pivô extremo)
- Memória:  $O(\log n)$  chamada recursiva

### 2.4.3 Pros/Cons

✓ Prós

✗ Contras

---

In-place, rápido em prática    Pior caso ruim, instável

### 2.4.4 Cenários

- Bases de dados em memória
- Ordenar por score

### 2.4.5 Passo-a-passo

1. Escolha do pivô
  - Selecione um elemento do vetor (início, fim, meio ou estratégia “mediana de três”). Esse elemento é o pivô.
2. Particionamento
  1. Reorganize o vetor de modo que:
    - todos os valores menores que o pivô fiquem à esquerda;
    - o pivô fique em sua posição definitiva;
    - todos os valores maiores que o pivô fiquem à direita.
  - Existem várias técnicas; a mais comum usa dois índices que varrem o vetor de fora para dentro.
3. Chamadas recursivas
  1. Aplique Quick Sort recursivamente à sub-lista esquerda (elementos menores que o pivô).
  2. Aplique Quick Sort recursivamente à sub-lista direita (elementos maiores que o pivô).
4. Condição de parada
  - Se a sub-lista possuir zero ou um elemento, ela já está ordenada, e a recursão retorna.
5. Conclusão
  - Quando todas as chamadas recursivas retornam, o vetor inteiro está ordenado.

### 2.4.6 Pseudocódigo

```
quick(A, lo, hi):  
  if lo ≥ hi return  
  p ← partition(A, lo, hi)  
  quick(A, lo, p-1)  
  quick(A, p+1, hi)
```

### 2.4.7 Diagrama ASCII

```

inicial      [ 4 3 1 2 5 6 9 8 7 ]      pivot=6
particiona → [ 4 3 1 2 5 | 6 | 9 8 7 ]
-----
sub-array esq [ 4 3 1 2 5 ]      pivot=5
particiona → [ 4 3 1 2 | 5 | ]
↳ [ 4 3 1 2 ]      pivot=2
particiona → [ 1 | 2 | 4 3 ]
↳ [ 4 3 ]      pivot=3
particiona → [ | 3 | 4 ]
resultado esq [ 1 2 3 4 ]      (ordenado)
-----
sub-array dir [ 9 8 7 ]      pivot=7
particiona → [ | 7 | 9 8 ]
↳ [ 9 8 ]      pivot=8
particiona → [ | 8 | 9 ]
resultado dir [ 7 8 9 ]      (ordenado)
-----
array final [ 1 2 3 4 5 6 7 8 9 ]      (ordenado)

```

## 2.5 Heap Sort

### 2.5.1 Breve descrição

Descrição: build-heap + remove root & heapify.

### 2.5.2 Análise

- Tempo:  $O(n \log n)$  worst/avg
- Memória:  $O(1)$  extra (in-place no array)

### 2.5.3 Pros/Cons

✓ Prós

✗ Contras

Pior caso garantido, prior-queue embutida    Não estável, lento em cache

### 2.5.4 Cenários

- Sistemas de tempo real (pior caso previsível)

### 2.5.5 Passo-a-passo

#### 1. Construção do heap

1. Transforme o vetor completo em um heap binário máximo (max-heap), no qual o maior elemento fica na raiz.
  - Isso pode ser feito percorrendo o vetor de trás para frente e aplicando a operação **heapify** (ajustar o sub-heap) em cada índice.

#### 2. Ordenação propriamente dita Repita até que restem apenas um ou zero elementos no heap:

1. Troque o primeiro elemento (máximo) com o último elemento ainda não ordenado. O maior valor passa para sua posição final no fim do vetor.
  2. Reduza o tamanho lógico do heap em 1 (ignore a posição já ordenada).
  3. Aplique **heapify** na raiz para restaurar a propriedade de max-heap.
3. Conclusão
- Quando o heap tiver tamanho 1, todos os elementos estarão em ordem crescente no vetor.

### 2.5.6 Pseudocódigo

```

heapSort(A):
  buildMaxHeap(A)
  for i ← n-1 ... 1
    swap A[0] ↔ A[i]
    heapify(A, 0, i)

```

### 2.5.7 Diagrama ASCII

```

inicial          [ 4 10 3 5 1 8 ]

constrói heap → [ 10 5 8 4 1 3 ]          (max-heap)

-----

extração 1
swap raiz ↔ 3   [ 3 5 8 4 1 | 10 ]
heapify        → [ 8 5 3 4 1 | 10 ]

extração 2
swap raiz ↔ 1   [ 1 5 3 4 | 8 10 ]
heapify        → [ 5 4 3 1 | 8 10 ]

extração 3
swap raiz ↔ 1   [ 1 4 3 | 5 8 10 ]
heapify        → [ 4 1 3 | 5 8 10 ]

extração 4
swap raiz ↔ 3   [ 3 1 | 4 5 8 10 ]
heapify        → [ 3 1 | 4 5 8 10 ]   (já heap)

extração 5
swap raiz ↔ 1   [ 1 | 3 4 5 8 10 ]   (resta tamanho 1)

-----

array final      [ 1 3 4 5 8 10 ]          (ordenado)

```

## 3. Priority Queue

### 3.1 Breve descrição

ADT que retorna sempre o elemento com maior (ou menor) prioridade.

### 3.2 Análise (Heap implementation)

- Insert:  $O(\log n)$
- Extract-max/min:  $O(\log n)$
- Peek:  $O(1)$

### 3.3 Pros/Cons



Ótima para agendamentos, Dijkstra, filas de impressão



Heap não é ordenado globalmente; busca arbitrária é  $O(n)$

### 3.4 Cenários de aplicação

- Scheduler de processos OS
- Feed classificado por relevância
- Autocomplete por frequência

### 3.5 Passo-a-passo

1. Receba o vetor original  $A[0 \dots n-1]$ .
2. Crie uma Priority Queue vazia (implementada como min-heap).
3. Insira todos os elementos de  $A$  na fila, um por vez, mantendo a propriedade de heap.
4. Crie uma lista (ou reutilize  $A$ ) para armazenar o resultado ordenado.
5. Enquanto a fila não estiver vazia:
  1. Remova (**poll**) o elemento de menor valor do heap.
  2. Grave esse elemento na próxima posição da lista de saída.
6. Quando a fila esvaziar, todos os elementos terão sido extraídos em ordem crescente; a lista está ordenada.

### 3.6 Pseudocódigo

```
# Premissas
# - Vetor A indexado a partir de 1
# - n representa o tamanho atual do heap
# - capacity é o tamanho máximo permitido
# - A[1] contém sempre o menor elemento

PQ.insert(x):
    if n == capacity:
        erro "fila cheia"
    A[++n] = x
    swim(n)

PQ.extractMin():
    if n == 0:
        erro "fila vazia"
```



```

min = A[1]
swap A[1] ↔ A[n]      # coloca a folha na raiz
n = n - 1
if n >= 1:             # ainda há elementos para reequilibrar
    sink(1)
A[n+1] = null          # limpeza opcional
return min

```

### 3.7 Diagrama ASCII (Heap)

```

# Inserção (1-indexado)
n=0

insert(6) → A=[6]
    6

insert(4) → A=[6,4] → swim → A=[4,6]
    4
   /
  6

insert(8) → A=[4,6,8]
    4
   / \
  6   8

insert(1) → A=[4,6,8,1] → swim → A=[1,4,8,6]
    1
   / \
  4   8
 /
6

insert(7) → A=[1,4,8,6,7]
    1
   / \
  4   8
 / \
6   7

insert(3) → A=[1,4,8,6,7,3] → swim → A=[1,4,3,6,7,8]
    1
   / \
  4   3
 / \ /
6  7 8

insert(2) → A=[1,4,3,6,7,8,2] → swim → A=[1,4,2,6,7,8,3]
    1
   / \
  4   2

```

```

    / \   / \
   6   7 8   3

# Extração (1-indexado)
out=[]

extractMin()=1 → swap→ [3,4,2,6,7,8], n=6 → sink→ [2,4,3,6,7,8]
      2           out=[1]
    / \
   4   3
  / \ /
 6   7 8

extractMin()=2 → swap→ [8,4,3,6,7], n=5 → sink→ [3,4,8,6,7]
      3           out=[1,2]
    / \
   4   8
  / \
 6   7

extractMin()=3 → swap→ [7,4,8,6], n=4 → sink→ [4,6,8,7]
      4           out=[1,2,3]
    / \
   6   8
  /
 7

extractMin()=4 → swap→ [7,6,8], n=3 → sink→ [6,7,8]
      6           out=[1,2,3,4]
    / \
   7   8

extractMin()=6 → swap→ [8,7], n=2 → sink→ [7,8]
      7           out=[1,2,3,4,6]
    \
     8

extractMin()=7 → swap→ [8], n=1 → sink→ [8]
      8           out=[1,2,3,4,6,7]

extractMin()=8 → heap vazio
                out=[1,2,3,4,6,7,8]

```

## 4. Caso prático para discussão

Desafio: você recebe 10.000 pedidos, cada um com priorityScore (0-100) e dispatchWindow (min). Qual estratégia você usaria para manter a lista em ordem "melhor-pedido-primeiro" enquanto novos pedidos chegam a cada segundo?

- Passo 1: discuta em dupla pros/cons de: Insertion na cauda, Heap de duplas chaves, QuickSort periódico.

- Passo 2: em grupos de 4, elaborem diagrama rápido (quadro branco) demonstrando o fluxo de chegada e extração.
- Passo 3: círculos de feedback – cada grupo explica e recebe 2 perguntas de outro grupo.

Quiz (5 questões). Cobre: identificador de pivô em Quick Sort, estabilidade, O-notation de Heapify, etc.

## Acessibilidade

- Todos os diagramas descritos verbalmente e disponíveis em texto alternativo.
- Slides em contraste alto (#000 / #FFF + #FFD700 para destaque).