

# Algoritmos de Ordenação

## Selection Sort e Insertion Sort

---

Aulas 3 e 4

Estrutura de Dados - 3º Semestre ADS


# Revisão da Aula Anterior

## Conceitos Fundamentais das Aulas 1 e 2

Antes de avançarmos para os novos algoritmos, vamos revisar brevemente os conceitos principais que estudamos:



### O Que é Ordenação?

Processo de reorganizar uma coleção de elementos em uma determinada ordem (crescente, decrescente ou outro critério).

 Aplicações: busca eficiente, processamento de dados, visualização clara, detecção de anomalias.


### Classificação dos Algoritmos

Os algoritmos de ordenação podem ser classificados por diferentes critérios:

 **In-place** vs **Not-in-place** (uso de memória)  
 **Estável** vs **Instável** (ordem relativa de chaves iguais)




### Notação Big O

Forma de descrever o comportamento assintótico de funções, focando no termo dominante e ignorando constantes.

 Principais classes:  $O(1)$ ,  $O(\log n)$ ,  $O(n)$ ,  $O(n \log n)$ ,  $O(n^2)$ ,  $O(n^3)$ ,  $O(2^n)$ ,  $O(n!)$

### Análise de Casos

Diferentes cenários para análise de desempenho de algoritmos:

 **Melhor Caso:** Menor número de operações  
 **Pior Caso:** Maior número de operações  
 **Caso Médio:** Desempenho esperado para entrada típica

## Bubble Sort: Nosso Primeiro Algoritmo

Revisamos o **Bubble Sort**, um algoritmo simples que compara pares adjacentes e os troca se estiverem na ordem errada.

- ✓ Complexidade:  $O(n^2)$  no pior e caso médio,  $O(n)$  no melhor caso (otimizado)
- ✓ Espaço:  $O(1)$  - In-place
- ✓ Estabilidade: Estável



Comparação e troca de elementos adjacentes

# Selection Sort: Conceito

## A Ideia Fundamental

O **Selection Sort** é baseado na ideia de **selecionar** repetidamente o menor (ou maior) elemento da parte não ordenada e colocá-lo na posição correta na parte ordenada.

## Como Funciona

1. Divide o array em duas partes: ordenada (inicialmente vazia) e não ordenada.
2. Encontra o menor elemento na parte não ordenada.
3. Troca esse elemento com o primeiro elemento da parte não ordenada.
4. Expande a parte ordenada para incluir esse elemento.
5. Repete até que toda a lista esteja ordenada.

## Características

- ✓ **Simple**s de entender e implementar
- ✓ Número de **troc**as é **minimizado** (máximo  $n-1$  trocas)
- ✓ Desempenho **previsível** (sempre  $O(n^2)$ )
- ✗ **Não é adaptativo** (não se beneficia de ordem parcial)
- ✗ **Não é estável** (pode alterar a ordem relativa de elementos iguais)

## Visualização Conceitual

Ordenando o array: [64, 25, 12, 22, 11]

Passo 1: Encontrar o menor elemento (11)

64	25	12	22	11
----	----	----	----	----

Trocar 11 com o primeiro elemento (64)

Passo 2: Encontrar o próximo menor (12)

11	25	12	22	64
----	----	----	----	----

Trocar 12 com o segundo elemento (25)

Passo 3: Encontrar o próximo menor (22)

11	12	25	22	64
----	----	----	----	----

Trocar 22 com o terceiro elemento (25)

Resultado Final

11	12	22	25	64
----	----	----	----	----

💡 O Selection Sort é eficiente em termos de **número de trocas**, realizando no máximo  $n-1$  trocas (uma por iteração). Isso pode ser vantajoso quando o custo de troca é alto.

# Selection Sort: Algoritmo

## Algoritmo Passo a Passo

Vamos detalhar o funcionamento do Selection Sort com um exemplo completo:

### Pseudocódigo

```
function selectionSort(arr[], n):  
    // Percorre o array  
    for i = 0 to n-2:  
        // Encontra o índice do menor elemento  
        // na parte não ordenada  
        min_idx = i  
        for j = i+1 to n-1:  
            if arr[j] < arr[min_idx]:  
                min_idx = j  
  
        // Troca o elemento mínimo encontrado  
        // com o primeiro elemento não ordenado  
        if min_idx != i:  
            swap(arr[min_idx], arr[i])
```

### Explicação

#### Loop Externo (i)

Controla a posição atual na parte ordenada. A cada iteração, um novo elemento é adicionado à parte ordenada.

#### Loop Interno (j)

Percorre a parte não ordenada para encontrar o elemento mínimo.

#### Troca

Coloca o elemento mínimo encontrado na posição correta da parte ordenada.

#### Otimização

A verificação `if min_idx != i` evita trocas desnecessárias quando o elemento já está na posição correta.

### Exemplo Detalhado

Array: [29, 10, 14, 37, 13]

Iteração 1 (i=0)

29	10	14	37	13
----	----	----	----	----

min\_idx = 1, trocar arr[0] e arr[1]

10	29	14	37	13
----	----	----	----	----

Iteração 2 (i=1)

10	29	14	37	13
----	----	----	----	----

Comparando com arr[4]=13

10	29	14	37	13
----	----	----	----	----

min\_idx = 4, trocar arr[1] e arr[4]

10	13	14	37	29
----	----	----	----	----

Iteração 3 (i=2)

10	13	14	37	29
----	----	----	----	----

min\_idx = 2, nenhuma troca necessária

10	13	14	37	29
----	----	----	----	----

Iteração 4 (i=3)

10	13	14	37	29
----	----	----	----	----

min\_idx = 4, trocar arr[3] e arr[4]

10	13	14	29	37
----	----	----	----	----

# Selection Sort: Implementação

## Implementação em C++

Vamos implementar o Selection Sort em C++ básico, sem usar recursos avançados como STL ou POO:

```
// Função para trocar dois elementos de lugar
void trocar(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Função para imprimir um array
void imprimirArray(int arr[], int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}

// Função que implementa o Selection Sort
void selectionSort(int arr[], int n) {
    int i, j, indice_minimo;

    // Um por um, move o limite do subarray não ordenado
    for (i = 0; i < n - 1; i++) {
        // Encontra o elemento mínimo no array não ordenado
        indice_minimo = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[indice_minimo]) {
                indice_minimo = j;
            }
        }

        // Troca o elemento mínimo encontrado com o primeiro
        // elemento da parte não ordenada (se não for o mesmo)
        if (indice_minimo != i) {
            trocar(&arr[indice_minimo], &arr[i]);
        }

        // Opcional: Imprimir o array após cada iteração
        // std::cout << "Após iteração " << i + 1 << ": ";
        // imprimirArray(arr, n);
    }
}
```

```
// Função principal para testar o Selection Sort
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    std::cout << "Array original: \n";
    imprimirArray(arr, n);

    selectionSort(arr, n);

    std::cout << "Array ordenado: \n";
    imprimirArray(arr, n);

    return 0;
}
```

### Explicação do Código

Função trocar()

Função auxiliar que troca dois elementos usando ponteiros. Esta é uma implementação genérica de troca que pode ser reutilizada em outros algoritmos.

Função selectionSort()

> **Loop externo (i):** Controla a posição atual na parte ordenada, indo de 0 a n-2.

> **Loop interno (j):** Percorre a parte não ordenada (de i+1 até n-1) para encontrar o elemento mínimo.

> **Verificação:** Compara cada elemento com o mínimo atual e atualiza o índice se necessário.

> **Troca condicional:** Troca apenas se o mínimo encontrado não for o elemento atual (otimização).

Detalhes de Implementação

- ✔ Uso de **ponteiros** para realizar a troca eficientemente.
- ✔ **Comentários** explicativos para facilitar a compreensão.
- ✔ Código **modular** com funções separadas para cada responsabilidade.
- ✔ **Otimização** para evitar trocas desnecessárias (verificação if (indice\_minimo != i)).

Dica

💡 Para visualizar o progresso do algoritmo, você pode descomentar as linhas que imprimem o array após cada iteração.

# Selection Sort: Análise

## Análise de Complexidade

Vamos analisar o Selection Sort em termos de complexidade de tempo, espaço e outras características:

### Complexidade de Tempo

Análise do Algoritmo:

- > **Loop externo:** Executa (n-1) vezes
- > **Loop interno:** Na primeira iteração executa (n-1) vezes, na segunda (n-2) vezes, e assim por diante
- > **Total de comparações:**  $(n-1) + (n-2) + \dots + 1 = n(n-1)/2 \approx n^2/2$

Complexidade:

- ✓ **Melhor caso:**  $O(n^2)$  - Mesmo número de comparações
- ✓ **Caso médio:**  $O(n^2)$
- ✓ **Pior caso:**  $O(n^2)$

### Outras Características

Complexidade de Espaço:

- >  **$O(1)$**  - Ordenação in-place
- > Usa apenas algumas variáveis auxiliares (índices e temporária para troca)

Estabilidade:

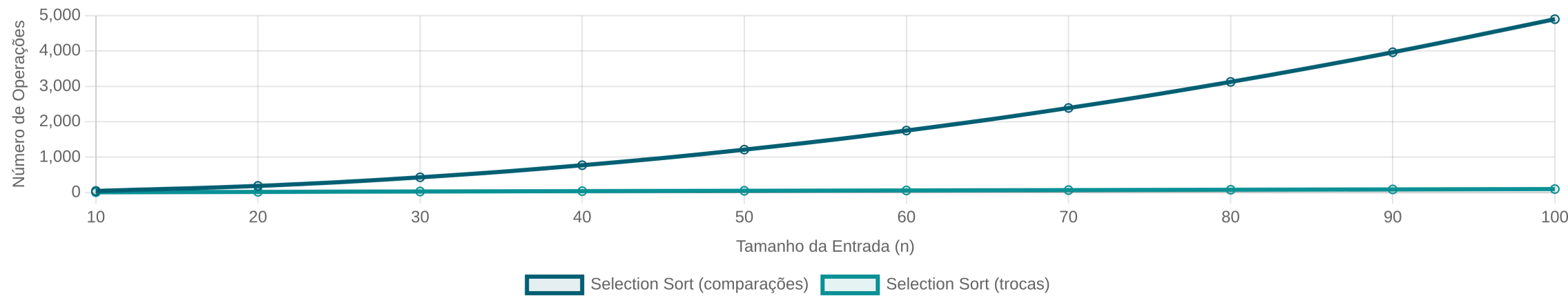
- ✗ **Não é estável** - Pode alterar a ordem relativa de elementos com valores iguais
- > Exemplo:  $[4, 2, 3, 4^*, 1] \rightarrow [1, 2, 3, 4^*, 4]$  (a ordem dos 4s pode ser alterada)

Número de Trocas:

- > **No máximo (n-1) trocas** - Uma por iteração do loop externo
- > Significativamente menor que o Bubble Sort (que pode fazer até  $n(n-1)/2$  trocas)

## Desempenho Comparativo

Comparações vs. Trocas no Selection Sort



### Vantagens

- + Simples de entender e implementar
- + Número mínimo de trocas ( $O(n)$ )
- + Desempenho previsível (sempre  $O(n^2)$ )
- + Funciona bem para arrays pequenos

### Desvantagens

- Ineficiente para grandes conjuntos de dados
- Sempre  $O(n^2)$ , mesmo quando o array já está ordenado
- Não é estável
- Não é adaptativo (não se beneficia de ordem parcial)

# Insertion Sort: Conceito

## A Ideia Fundamental

O **Insertion Sort** é baseado na ideia de construir a sequência ordenada um elemento por vez, **inserindo** cada elemento na posição correta.

## Como Funciona

- 1. Divide o array em duas partes: ordenada (inicialmente apenas o primeiro elemento) e não ordenada.
- 2. Pega o primeiro elemento da parte não ordenada.
- 3. Insere este elemento na posição correta dentro da parte ordenada.
- 4. Expande a parte ordenada para incluir esse elemento.
- 5. Repete até que toda a lista esteja ordenada.

## Características

- ✓ **Simple**s de entender e implementar
- ✓ **Efficiente** para conjuntos pequenos ou quase ordenados
- ✓ **Adaptativo** (se beneficia de ordem parcial)
- ✓ **Estável** (mantém a ordem relativa de elementos iguais)
- ✓ **In-place** (usa memória constante)
- ✗ **Ineficiente** para grandes conjuntos desordenados

## Visualização Conceitual

Ordenando o array: [5, 2, 4, 6, 1, 3]

Estado inicial: Primeiro elemento já está "ordenado"

5	2	4	6	1	3
---	---	---	---	---	---

Parte ordenada: [5], Parte não ordenada: [2, 4, 6, 1, 3]

Passo 1: Inserir 2 na posição correta

2	5	4	6	1	3
---	---	---	---	---	---

Parte ordenada: [2, 5], Parte não ordenada: [4, 6, 1, 3]

Passo 2: Inserir 4 na posição correta

2	4	5	6	1	3
---	---	---	---	---	---

Parte ordenada: [2, 4, 5], Parte não ordenada: [6, 1, 3]

Passo 3: Inserir 6 na posição correta

2	4	5	6	1	3
---	---	---	---	---	---

Parte ordenada: [2, 4, 5, 6], Parte não ordenada: [1, 3]

Resultado Final

1	2	3	4	5	6
---	---	---	---	---	---

Array completamente ordenado

💡 O Insertion Sort é particularmente eficiente quando o array já está **parcialmente ordenado**. Nesse caso, muitos elementos já estarão em suas posições corretas ou próximas delas.



# Insertion Sort: Analogia

## Ordenando Cartas de Baralho

Uma das melhores analogias para entender o Insertion Sort é a forma como muitas pessoas organizam cartas de baralho em suas mãos:



### Como Organizamos Cartas

1. Começamos com uma carta na mão (parte ordenada).
2. Pegamos uma nova carta do monte (parte não ordenada).
3. Comparamos com as cartas que já temos na mão.
4. Inserimos a nova carta na posição correta.
5. Repetimos até organizar todas as cartas.



### Por Que Esta Analogia Funciona

- ✓ Mantemos uma **seção ordenada** que cresce gradualmente.
- ✓ Cada nova carta é **inserida** na posição correta.
- ✓ Precisamos **deslocar** cartas para abrir espaço.
- ✓ O processo é **intuitivo** e natural para humanos.
- ✓ É **eficiente** quando já temos algumas cartas ordenadas.

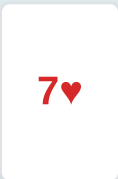
### Visualização da Analogia

Passo 1: Começamos com uma carta



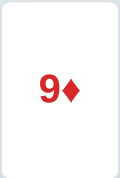
Parte ordenada: [7♥]  
Próxima carta: 3♠

Passo 2: Inserimos 3♠ antes do 7♥



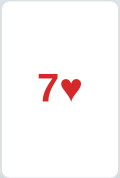
Parte ordenada: [3♠, 7♥]  
Próxima carta: 9♦

Passo 3: Inserimos 9♦ após o 7♥



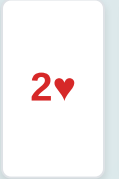
Parte ordenada: [3♠, 7♥, 9♦]  
Próxima carta: 5♣

Passo 4: Inserimos 5♣ entre 3♠ e 7♥



Parte ordenada: [3♠, 5♣, 7♥, 9♦]  
Próxima carta: 2♥

Passo 5: Inserimos 2♥ antes do 3♠



Parte ordenada: [2♥, 3♠, 5♣, 7♥, 9♦]  
Mão completamente ordenada!

**i** Esta analogia ilustra perfeitamente por que o Insertion Sort é **adaptativo** e **eficiente para pequenas entradas**. Quando temos poucas cartas ou quando muitas já estão em ordem, o processo é rápido e intuitivo.



# Insertion Sort: Algoritmo

## Algoritmo Passo a Passo

Vamos detalhar o funcionamento do Insertion Sort com um exemplo completo:

### Pseudocódigo

```
function insertionSort(arr[], n):
    // Percorre o array a partir do segundo elemento
    for i = 1 to n-1:
        // Guarda o elemento atual como chave
        key = arr[i]

        // Move elementos maiores que a chave
        // para uma posição à frente
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j+1] = arr[j]
            j = j - 1

        // Insere a chave na posição correta
        arr[j+1] = key
```

### Explicação

#### Loop Principal (i)

Percorre o array a partir do segundo elemento (índice 1). Cada elemento será inserido na parte ordenada à esquerda.

#### Elemento Chave

O elemento atual (arr[i]) é armazenado como "chave" para ser inserido na posição correta.

#### Loop Interno (j)

Desloca os elementos maiores que a chave para a direita, abrindo espaço para inserir a chave.

#### Inserção

Insera a chave na posição correta, após todos os elementos menores e antes de todos os maiores.

### Exemplo Detalhado

Array: [31, 41, 59, 26, 41, 58]

Estado inicial: Primeiro elemento já está "ordenado"

31	41	59	26	41	58
----	----	----	----	----	----

Iteração 1 (i=1): key = 41

31	41	59	26	41	58
----	----	----	----	----	----

41 > 31, então permanece na posição

31	41	59	26	41	58
----	----	----	----	----	----

Iteração 2 (i=2): key = 59

31	41	59	26	41	58
----	----	----	----	----	----

59 > 41, então permanece na posição

31	41	59	26	41	58
----	----	----	----	----	----

Iteração 3 (i=3): key = 26

31	41	59	26	41	58
----	----	----	----	----	----

Deslocando elementos maiores que 26

26	31	41	59	41	58
----	----	----	----	----	----

Resultado após todas as iterações

26	31	41	41	58	59
----	----	----	----	----	----

# Insertion Sort: Implementação

## Implementação em C++

Vamos implementar o Insertion Sort em C++ básico, sem usar recursos avançados como STL ou POO:

```
// Função para imprimir um array
void imprimirArray(int arr[], int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}

// Função que implementa o Insertion Sort
void insertionSort(int arr[], int n) {
    int i, chave, j;

    // Percorre o array a partir do segundo elemento
    for (i = 1; i < n; i++) {
        // Guarda o elemento atual como chave
        chave = arr[i];

        // Move elementos maiores que a chave
        // para uma posição à frente
        j = i - 1;
        while (j >= 0 && arr[j] > chave) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }

        // Insere a chave na posição correta
        arr[j + 1] = chave;

        // Opcional: Imprimir o array após cada iteração
        // std::cout << "Após iteração " << i << ": ";
        // imprimirArray(arr, n);
    }
}
```

```
// Função principal para testar o Insertion Sort
int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    std::cout << "Array original: \n";
    imprimirArray(arr, n);

    insertionSort(arr, n);

    std::cout << "Array ordenado: \n";
    imprimirArray(arr, n);

    return 0;
}
```

### Explicação do Código

#### Função insertionSort()

- > **Loop principal (i):** Percorre o array a partir do segundo elemento (índice 1).
- > **Variável chave:** Armazena o elemento atual que será inserido na posição correta.
- > **Loop interno (while):** Desloca elementos maiores que a chave para a direita.
- > **Inserção:** Coloca a chave na posição correta após o deslocamento.

#### Detalhes de Implementação

- ✔ **Simplicidade:** Implementação direta e fácil de entender.
- ✔ **In-place:** Ordenação feita no próprio array, sem arrays auxiliares.
- ✔ **Estabilidade:** Mantém a ordem relativa de elementos iguais.
- ✔ **Adaptabilidade:** Eficiente para arrays quase ordenados.

#### Visualização do Processo

Para o array [12, 11, 13, 5, 6]:

Após iteração 1: [11, 12, 13, 5, 6]

Após iteração 2: [11, 12, 13, 5, 6]

Após iteração 3: [5, 11, 12, 13, 6]

Após iteração 4: [5, 6, 11, 12, 13]

💡 Para visualizar o progresso do algoritmo, você pode descomentar as linhas que imprimem o array após cada iteração.

# Insertion Sort: Análise

## Análise de Complexidade

Vamos analisar o Insertion Sort em termos de complexidade de tempo, espaço e outras características:

### Complexidade de Tempo

Análise do Algoritmo:

- > **Loop externo:** Executa (n-1) vezes
- > **Loop interno:** No pior caso, executa i vezes para cada i
- > **Total de comparações (pior caso):**  $1 + 2 + \dots + (n-1) = n(n-1)/2 \approx n^2/2$

Complexidade:

- ✓ **Melhor caso:**  $O(n)$  - Array já ordenado
- ✓ **Caso médio:**  $O(n^2)$
- ✓ **Pior caso:**  $O(n^2)$  - Array em ordem inversa

### Outras Características

Complexidade de Espaço:

- >  **$O(1)$**  - Ordenação in-place
- > Usa apenas algumas variáveis auxiliares (chave, índices)

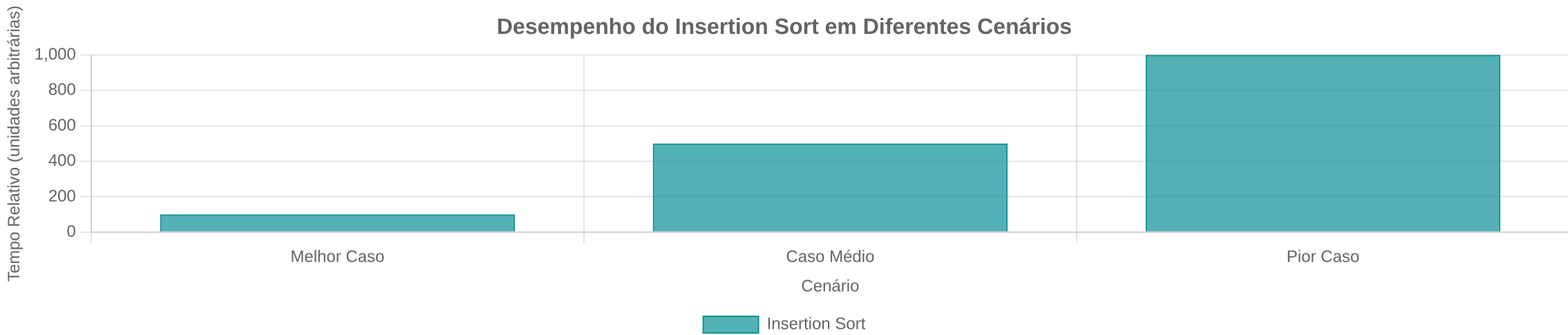
Estabilidade:

- ✓ **Estável** - Mantém a ordem relativa de elementos com valores iguais
- > Exemplo:  $[4, 2, 3, 4^*, 1] \rightarrow [1, 2, 3, 4, 4^*]$  (a ordem dos 4s é preservada)

Adaptabilidade:

- ✓ **Adaptativo** - Eficiente para arrays quase ordenados
- > Quanto mais ordenado o array, menos comparações e movimentações são necessárias

## Desempenho em Diferentes Cenários



### Vantagens

- + Simples de entender e implementar
- + Eficiente para conjuntos pequenos (< 20 elementos)
- + Adaptativo - eficiente para arrays quase ordenados
- + Estável - mantém a ordem relativa de elementos iguais
- + In-place - requer espaço auxiliar constante
- + Online - pode ordenar os dados à medida que chegam

### Desvantagens

- Ineficiente para grandes conjuntos de dados
- $O(n^2)$  no pior caso e caso médio
- Significativamente mais lento que algoritmos  $O(n \log n)$
- Muitas operações de deslocamento para arrays grandes

# Comparação: Selection vs Insertion

## Análise Comparativa

Vamos comparar os dois algoritmos estudados em diferentes aspectos:

Característica	Selection Sort	Insertion Sort	Vantagem
Melhor Caso	$O(n^2)$	$O(n)$	Insertion Sort
Caso Médio	$O(n^2)$	$O(n^2)$	Empate
Pior Caso	$O(n^2)$	$O(n^2)$	Empate
Espaço	$O(1)$	$O(1)$	Empate
Estabilidade	Não	Sim	Insertion Sort
Adaptabilidade	Não	Sim	Insertion Sort
Número de Trocas	$O(n)$ no máximo	$O(n^2)$ no pior caso	Selection Sort
Número de Comparações	Sempre $n(n-1)/2$	Depende da ordem	Insertion Sort

## Quando Usar Cada Algoritmo

### Selection Sort

Melhor escolha quando:

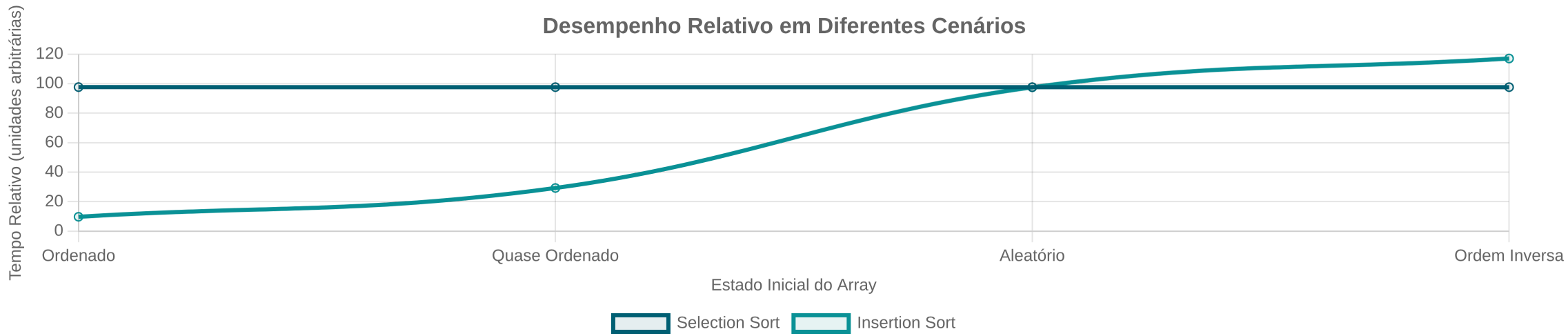
- ✔ O custo de **troca de elementos é alto** (objetos grandes)
- ✔ Você precisa de um **comportamento previsível** (sempre  $O(n^2)$ )
- ✔ A **memória é extremamente limitada** (usa menos variáveis auxiliares)
- ✔ A **simplicidade de implementação** é mais importante que o desempenho

### Insertion Sort

Melhor escolha quando:

- ✔ Os dados estão **parcialmente ordenados**
- ✔ Você precisa de um algoritmo **estável** (mantém ordem de elementos iguais)
- ✔ Os dados chegam **online** (um elemento por vez)
- ✔ Você está trabalhando com **conjuntos pequenos** (< 20 elementos)

## Comparação de Desempenho



# Exercícios Propostos

## Atividades Práticas

Para consolidar o aprendizado sobre Selection Sort e Insertion Sort, propomos os seguintes exercícios:

### 1 Implementação e Teste

Implemente o Selection Sort e o Insertion Sort em C++ e compare o desempenho deles com diferentes tipos de entrada:

- > Array já ordenado
- > Array em ordem inversa
- > Array com elementos aleatórios

💡 Dica: Use a função `clock()` da biblioteca `ctime` para medir o tempo de execução.

### 2 Modificação de Algoritmos

Modifique os algoritmos para ordenar em ordem decrescente em vez de crescente.

```
// Exemplo de modificação para o Selection Sort
void selectionSortDecrescente(int arr[], int n) {
    int i, j, indice_maximo;

    for (i = 0; i < n - 1; i++) {
        indice_maximo = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] > arr[indice_maximo]) {
                indice_maximo = j;
            }
        }

        if (indice_maximo != i) {
            trocar(&arr[indice_maximo], &arr[i]);
        }
    }
}
```

💡 Dica: Basta alterar o operador de comparação nos algoritmos.

### 3 Ordenação de Estruturas

Crie uma estrutura `Aluno` com campos para nome, matrícula e nota. Implemente o Insertion Sort para ordenar um array de alunos por nota (decrescente) e, em caso de empate, por nome (alfabética).

```
struct Aluno {
    char nome[50];
    int matricula;
    float nota;
};

// Implemente a função:
void insertionSortAlunos(Aluno arr[], int n);
```

💡 Dica: Use a função `strcmp()` da biblioteca `cstring` para comparar strings.

### 4 Análise de Complexidade

Analise e compare a complexidade de tempo e espaço dos seguintes algoritmos:

```
// Algoritmo A
void algoritmoA(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Algoritmo B
void algoritmoB(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int chave = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > chave) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = chave;
    }
}
```

💡 Dica: Identifique qual algoritmo é qual e compare suas características.

### Desafio Extra



Implemente uma versão do Insertion Sort que utilize busca binária para encontrar a posição correta de inserção de cada elemento, em vez de busca linear.

Esta variação é chamada de **Binary Insertion Sort** e pode reduzir o número de comparações, embora o número de movimentações continue o mesmo.

❓ Pergunta: Qual é a complexidade de tempo desta variação? O Binary Insertion Sort ainda é  $O(n^2)$  ou melhora para  $O(n \log n)$ ?

### Entrega dos Exercícios

- 📁 Arquivos `.cpp` com as implementações solicitadas
- 📄 Relatório com análises de complexidade e comparações de desempenho
- 📅 Data de entrega: Próxima aula

# Próximas Aulas

## O Que Vem Por Aí

Agora que compreendemos os algoritmos de ordenação por troca e inserção, nas próximas aulas vamos explorar algoritmos mais eficientes baseados em outras estratégias:



### Aulas 5-6: Merge Sort

- ✓ Paradigma de Divisão e Conquista
- ✓ Implementação recursiva
- ✓ Análise de complexidade  $O(n \log n)$
- ✓ Estabilidade e aplicações práticas



### Aulas 7-10: Ordenação por Distribuição

- ✓ Counting Sort - ordenação por contagem
- ✓ Radix Sort - ordenação dígito a dígito
- ✓ Complexidade linear em casos específicos
- ✓ Limitações e casos de uso ideais

## Visão Geral do Módulo

### Semana 1

- Introdução e Complexidade
- Bubble Sort
- **Selection Sort**
- **Insertion Sort**

### Semana 2

- Divisão e Conquista
- Merge Sort
- Counting Sort
- Radix Sort (início)

### Semana 3

- Radix Sort (conclusão)
- Comparação de Algoritmos
- Casos de Uso Práticos
- Revisão e Avaliação

## Preparação para a Próxima Aula



Para aproveitar melhor a próxima aula sobre Merge Sort, recomendamos:

- > Revisar o conceito de recursão em C++
- > Completar os exercícios propostos sobre Selection e Insertion Sort
- > Pesquisar sobre o paradigma "Dividir para Conquistar" (opcional)