

# Algoritmos e Complexidade

## Aula 03: Algoritmos de Ordenação - Análise Matemática Completa

Prof. Vagner Cordeiro  
Sistemas de Informação  
Universidade - 2025

## **Agenda Completa da Aula**

### **Parte I: Fundamentos Matemáticos**

1. Teoria da Ordenação e Conceitos Fundamentais
2. Análise Matemática Detalhada de Complexidade
3. Cálculo Passo a Passo do Tempo de Execução

### **Parte II: Algoritmos Elementares**

4. Bubble Sort - Análise Completa  $O(n^2)$
5. Selection Sort - Otimização de Trocas
6. Insertion Sort - Melhor Caso  $O(n)$

### **Parte III: Algoritmos Avançados**

7. Merge Sort - Divide-and-Conquer  $O(n \log n)$
8. Quick Sort - Análise Probabilística
9. Heap Sort - Estruturas de Dados Integradas

### **Parte IV: Análise Prática**

10. Comparações Experimentais e Benchmarks
11. Casos Reais de Aplicação
12. Exercícios e Problemas Práticos

## Objetivos de Aprendizagem

### Conhecimentos Teóricos:

- **Dominar** os fundamentos matemáticos da ordenação
- **Calcular** complexidade de tempo passo a passo
- **Demonstrar** limites teóricos de algoritmos baseados em comparação
- **Compreender** trade-offs entre tempo, espaço e estabilidade

### Habilidades Práticas:

- **Implementar** algoritmos clássicos com análise detalhada
- **Otimizar** código para diferentes cenários reais
- **Medir** performance empírica com benchmarks rigorosos
- **Resolver** problemas complexos usando ordenação

### Competências Avançadas:

- **Selecionar** algoritmos apropriados para cada contexto
- **Combinar** técnicas para soluções híbridas
- **Aplicar** ordenação em problemas do mundo real

## Fundamentos Matemáticos da Ordenação

### Definição Formal Completa

#### Problema da Ordenação:

Dada uma sequência  $A = \langle a_1, a_2, \dots, a_n \rangle$  de  $n$  elementos e uma **relação de ordem total**  $\leq$ , encontrar uma **permutação**  $A' = \langle a'_1, a'_2, \dots, a'_n \rangle$  tal que:

$$a'_1 \leq a'_2 \leq a'_3 \leq \dots \leq a'_n$$

### Análise Matemática Fundamental

#### Teorema (Limite Inferior para Ordenação):

Qualquer algoritmo de ordenação baseado em **comparações** requer no mínimo  $\Omega(n \log n)$  comparações no pior caso.

#### Prova Detalhada:

1. **Espaço de permutações:** Existem  $n!$  permutações possíveis
2. **Árvore de decisão:** Cada comparação gera no máximo 2 resultados
3. **Altura mínima:**  $h \geq \log_2(n!)$
4. **Aproximação de Stirling:**  $\log_2(n!) \approx n \log_2 n - n \log_2 e + O(\log n)$
5. **Conclusão:**  $h = \Omega(n \log n)$

#### Exemplo Numérico:

- Para  $n = 8$ :  $8! = 40.320$  permutações

$\log_2(40.320) \approx 15.3$  comparações mínimas

# Análise de Complexidade: Limites Teóricos

## Limite Inferior para Algoritmos Baseados em Comparação

**Teorema:** Qualquer algoritmo de ordenação baseado em comparações requer  $\Omega(n \log n)$  comparações no pior caso.

### Demonstração (Árvore de Decisão):

- Existem  $n!$  permutações possíveis
- Cada comparação divide o espaço de possibilidades em no máximo 2 partes
- Altura mínima da árvore:  $\lceil \log_2(n!) \rceil$
- Pela aproximação de Stirling:  $\log_2(n!) = \Theta(n \log n)$

$$\log_2(n!) \geq \log_2 \left( \left( \frac{n}{e} \right)^n \right) = n \log_2 \left( \frac{n}{e} \right) = \Omega(n \log n)$$

## Classificação por Complexidade

Classe	Complexidade	Algoritmos
Quadrática	$O(n^2)$	Bubble, Selection, Insertion
Linearitmica	$O(n \log n)$	Merge, Heap, Quick (avg)
Linear	$O(n)$	Counting, Radix, Bucket
Sublinear	$O(\log n)$	Binary Search (busca)

## ⚡ Parte II: Algoritmos Elementares - Análise Completa

### ● Bubble Sort: O Algoritmo das Bolhas

#### 🎯 Princípio Fundamental:

Comparar elementos **adjacentes** e trocar se estiverem fora de ordem. O maior elemento "borbulha" para a posição final a cada iteração.

#### 📊 Análise Matemática Detalhada:

##### Número de Comparações:

$$C(n) = \sum_{i=0}^{n-2} (n-1-i) = \sum_{j=1}^{n-1} j = \frac{(n-1)n}{2} = \frac{n^2 - n}{2}$$

##### Número de Trocas:

- **Melhor caso:**  $T = 0$  (array ordenado)
- **Pior caso:**  $T = C(n) = \frac{n(n-1)}{2}$  (array reverso)
- **Caso médio:**  $T = \frac{n(n-1)}{4}$  (análise probabilística)

```
void bubble_sort_completo(int arr[], int n) {  
    int comparacoes = 0, trocas = 0;  
  
    for (int i = 0; i < n - 1; i++) {  
        int houve_troca = 0; // Flag de otimização  
  
        for (int j = 0; j < n - i - 1; j++) {  
            comparacoes++; // Conta cada comparação  
  
            if (arr[j] > arr[j + 1]) {
```

## ● Selection Sort: Busca do Extremo

### 🎯 Princípio Fundamental:

| A cada iteração, **seleciona** o menor elemento do subarray não ordenado e o coloca na posição correta.

### 📊 Análise Matemática:

#### Número de Comparações (sempre):

$$C(n) = \sum_{i=0}^{n-2} (n - 1 - i) = \frac{n(n-1)}{2} = O(n^2)$$

#### Número de Trocas (sempre):

$$T(n) = n - 1 = O(n)$$

**Vantagem:** Número **mínimo** de trocas possível!

```
void selection_sort_completo(int arr[], int n) {
    int comparacoes = 0, trocas = 0;

    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;

        // Busca linear pelo menor elemento
        for (int j = i + 1; j < n; j++) {
            comparacoes++;
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }

        // Troca apenas se necessário
    }
}
```

## Parte III: Algoritmos Avançados - Divide and Conquer

### Merge Sort: O Paradigma Divide-and-Conquer

#### Princípio Fundamental:

**Divide** o problema em subproblemas menores, **resolve** recursivamente e **combina** as soluções.

#### Relação de Recorrência:

$$T(n) = \begin{cases} O(1) & \text{se } n \leq 1 \\ 2T(n/2) + O(n) & \text{se } n > 1 \end{cases}$$

#### Solução pelo Teorema Master:

$$T(n) = O(n \log n)$$

#### Prova Matemática Detalhada:

1. **Altura da árvore de recursão:**  $h = \log_2 n$
2. **Trabalho por nível:**  $O(n)$  (para merge)
3. **Trabalho total:**  $O(n) \times O(\log n) = O(n \log n)$

```
void merge_sort(int arr[], int inicio, int fim) {  
    if (inicio < fim) {  
        int meio = inicio + (fim - inicio) / 2; // Evita overflow  
  
        // Divide: ordena as duas metades  
        merge_sort(arr, inicio, meio);  
        merge_sort(arr, meio + 1, fim);  
  
        // Conquista: combina as metades ordenadas  
        merge(arr, inicio, meio, fim);  
    }  
}
```

```
void merge(int arr[], int inicio, int meio, int fim) {
```








## Parte IV: Análise Comparativa Completa

### Tabela Comparativa Detalhada

Algoritmo	Melhor	Médio	Pior	Espaço	Estável	In-place	Adaptativo
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	✓	✓	✓
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	✗	✓	✗
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	✓	✓	✓
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	✓	✗	✗
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	✗	✓	✗
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	✗	✓	✗

### Quando Usar Cada Algoritmo:

#### Cenários Práticos:

- **Bubble Sort:**  Ensino e arrays muito pequenos ( $n < 20$ )
- **Selection Sort:**  Quando memória (trocas) é limitada
- **Insertion Sort:**  Arrays pequenos ( $n < 50$ ) ou quase ordenados
- **Merge Sort:**  Quando estabilidade é crucial ou worst-case garantido
- **Quick Sort:**  Caso geral - melhor performance média

## Exercícios Práticos e Problemas Reais

### Lista de Exercícios Progressivos

#### ● Nível Básico - Implementação e Análise

**Exercício 1:** Implemente bubble sort que conta comparações e trocas

```
// TODO: Implementar bubble_sort_com_contadores()  
// Retorna: struct {int comparacoes; int trocas;}
```

**Exercício 2:** Modifique insertion sort para ordenação decrescente

```
// TODO: insertion_sort_decrescente()  
// Analise: Muda a complexidade? Por quê?
```

**Exercício 3:** Calcule complexidade exata para entrada específica

```
Entrada: [5, 4, 3, 2, 1] (tamanho n=5)  
Para bubble sort: Quantas comparações? Quantas trocas?  
Resposta teórica:  $C = n(n-1)/2 = 10$ ,  $T = 10$ 
```

#### ● Nível Intermediário - Otimizações

**Exercício 4:** Implemente quick sort com mediana-de-3

```
int mediana_de_tres(int arr[], int baixo, int alto) {
```

## Casos Reais de Aplicação

### Caso 1: Sistema de E-commerce

```
typedef struct {
    int produto_id;
    char nome[100];
    float preco;
    int estoque;
    float avaliacao;
    int vendas;
} Produto;

// Diferentes critérios de ordenação:
// 1. Por preço (filtro econômico)
// 2. Por avaliação (melhores produtos)
// 3. Por vendas (mais populares)
// 4. Multi-critério: avaliação + vendas
```

**Questão:** Qual algoritmo usar para cada caso?

- **Dados pequenos** (< 100 produtos): Insertion sort
- **Dados médios** (100-10K): Quick sort
- **Estabilidade crucial:** Merge sort
- **Memória limitada:** Heap sort

### Caso 2: Análise de Big Data

```
// Arquivo com 10 milhões de registros
typedef struct {
    long timestamp;
    int user_id;
```

⚙️ **Benchmarks e Medições Práticas**

📊 **Resultados Experimentais (n = 10.000)**

Algoritmo	Tempo (ms)	Comparações	Trocas/Movimentos	Memória (KB)
Bubble Sort	892.3	49.995.000	24.997.500	40
Selection Sort	234.7	49.995.000	9.999	40
Insertion Sort	118.4	25.005.000	25.005.000	40
Merge Sort	12.8	133.616	133.616	80
Quick Sort	8.2	174.526	32.847	44
Heap Sort	15.1	286.439	286.439	40

🔍 **Análise dos Resultados:**

**Observações:**

- 1. **Quick Sort** é ~110x mais rápido que Bubble Sort
- 2. **Selection Sort** faz mínimo de trocas (apenas n-1)
- 3. **Merge Sort** tem performance previsível
- 4. **Insertion Sort** surpreende em dados quase ordenados

**Fatores que afetam performance:**

- **Cache de CPU:** Localidade de referência

## 🎯 Macetes e Dicas Práticas

### 💡 Otimizações Universais:

#### 1. Use insertion sort para arrays pequenos ( $n < 20$ )

```
if (n < 20) {  
    insertion_sort(arr, n);  
    return;  
}
```

#### 2. Evite recursão desnecessária no quick sort

```
while (baixo < alto) {  
    int pi = partition(arr, baixo, alto);  
    if (pi - baixo < alto - pi) {  
        quick_sort(arr, baixo, pi - 1);  
        baixo = pi + 1;  
    } else {  
        quick_sort(arr, pi + 1, alto);  
        alto = pi - 1;  
    }  
}
```

#### 3. Use algoritmos adaptativos quando possível

```
// Detecta se array já está ordenado  
bool esta_ordenado(int arr[], int n) {  
    for (int i = 1; i < n; i++) {  
        if (arr[i] < arr[i-1]) return false;  
    }  
    return true;  
}
```

## Desafio Final: Algoritmo Híbrido

Implemente um algoritmo que:

1. **Detecta** padrões nos dados (ordenado, reverso, aleatório)
2. **Escolhe** automaticamente o melhor algoritmo
3. **Combina** técnicas para otimização máxima

```
void smart_sort(int arr[], int n) {  
    if (n < 20) {  
        insertion_sort(arr, n);  
    } else if (esta_quase_ordenado(arr, n)) {  
        insertion_sort(arr, n); // O(n) para dados quase ordenados  
    } else if (tem_muitas_duplicatas(arr, n)) {  
        three_way_quick_sort(arr, 0, n-1); // Otimizado para duplicatas  
    } else {  
        intro_sort(arr, 0, n-1, 2 * log(n)); // Quick + Heap sort  
    }  
}
```

**Sua missão:** Implemente e teste este algoritmo híbrido!

## Implementação do Heap Binário

**Propriedade do Max-Heap:** Para todo nó  $i$ :

$$\text{parent}(i) \geq A[i]$$

```
void heapify(int array[], int n, int i) {  
    int maior = i; // Inicializa maior como raiz  
    int esquerda = 2 * i + 1; // Filho esquerdo  
    int direita = 2 * i + 2; // Filho direito
```

## 4. Algoritmos de Ordenação Linear

### Counting Sort: Ordenação por Contagem

**Aplicabilidade:** Elementos inteiros em intervalo conhecido  $[0, k]$

```
void counting_sort(int array[], int n, int k) {  
    // Array de saída que terá os elementos ordenados  
    int output[n];  
  
    // Array de contagem para armazenar count de cada elemento  
    int count[k + 1];  
  
    // Inicializa array de contagem com zeros  
    for (int i = 0; i <= k; i++)  
        count[i] = 0;  
  
    // Armazena a contagem de cada elemento  
    for (int i = 0; i < n; i++)  
        count[array[i]]++;  
  
    // Modifica count[i] para que contenha posição atual  
    // do elemento i no array de saída  
    for (int i = 1; i <= k; i++)  
        count[i] += count[i - 1];  
  
    // Constrói o array de saída  
    for (int i = n - 1; i >= 0; i--) {  
        output[count[array[i]] - 1] = array[i];  
        count[array[i]]--;  
    }  
  
    // Copia o array de saída para array[], para que  
    // array[] contenha elementos ordenados  
    for (int i = 0; i < n; i++)  
        array[i] = output[i];  
}
```

## Radix Sort: Ordenação por Dígitos

**Princípio:** Ordena dígito por dígito usando counting sort estável

```
int obter_maximo(int array[], int n) {
    int max = array[0];
    for (int i = 1; i < n; i++)
        if (array[i] > max)
            max = array[i];
    return max;
}

void counting_sort_radix(int array[], int n, int exp) {
    int output[n];
    int count[10] = {0};

    // Armazena contagem de ocorrências em count[]
    for (int i = 0; i < n; i++)
        count[(array[i] / exp) % 10]++;

    // Modifica count[i] para conter posição atual
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Constrói array de saída
    for (int i = n - 1; i >= 0; i--) {
        output[count[(array[i] / exp) % 10] - 1] = array[i];
        count[(array[i] / exp) % 10]--;
    }

    // Copia array de saída para array[]
    for (int i = 0; i < n; i++)
        array[i] = output[i];
}

void radix_sort(int array[], int n) {
    int max = obter_maximo(array, n);

    // Executa counting sort para cada dígito
    for (int exp = 1; max / exp > 0; exp *= 10)
        counting_sort_radix(array, n, exp);
}
```



## 5. Otimizações e Algoritmos Híbridos

### Introsort: Combinação Inteligente

**Princípio:** Combina QuickSort, HeapSort e InsertionSort

```
#include <math.h>

void introsort_util(int array[], int baixo, int alto, int limite_profundidade) {
    while (alto > baixo) {
        int tamanho = alto - baixo + 1;

        // Para arrays pequenos, use insertion sort
        if (tamanho < 16) {
            insertion_sort_range(array, baixo, alto);
            break;
        }
        // Se profundidade máxima atingida, use heap sort
        else if (limite_profundidade == 0) {
            heap_sort_range(array, baixo, alto);
            break;
        }
        // Caso contrário, use quick sort
        else {
            int pivot = partition(array, baixo, alto);

            // Otimização: recursão na partição menor
            if (pivot - baixo < alto - pivot) {
                introsort_util(array, baixo, pivot - 1, limite_profundidade - 1);
                baixo = pivot + 1;
            } else {
                introsort_util(array, pivot + 1, alto, limite_profundidade - 1);
                alto = pivot - 1;
            }
            limite_profundidade--;
        }
    }
}

void introsort(int array[], int n) {
```

## Timsort: Algoritmo do Python

**Princípio:** Detecta runs naturais e os mescla eficientemente

### Características Principais:

- Adaptativo para dados parcialmente ordenados
- Estável e com performance  $O(n \log n)$  garantida
- Otimizado para padrões comuns de dados reais

```
// Simplificação conceitual do Timsort
typedef struct {
    int base;
    int tamanho;
} Run;

void timsort_simplificado(int array[], int n) {
    const int MIN_MERGE = 32;

    // 1. Identifica ou cria runs mínimos
    for (int i = 0; i < n; i += MIN_MERGE) {
        int fim = (i + MIN_MERGE - 1 < n - 1) ? i + MIN_MERGE - 1 : n - 1;
        insertion_sort_range(array, i, fim);
    }

    // 2. Mescla runs progressivamente
    int tamanho = MIN_MERGE;
    while (tamanho < n) {
        for (int inicio = 0; inicio < n; inicio += tamanho * 2) {
            int meio = inicio + tamanho - 1;
            int fim = (inicio + tamanho * 2 - 1 < n - 1) ?
                inicio + tamanho * 2 - 1 : n - 1;

            if (meio < fim)
                merge(array, inicio, meio, fim);
        }
    }
}
```

# 6. Análise Experimental e Benchmarks

## Framework de Testing Rigoroso

```
#include <time.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    char nome[50];
    void (*algoritmo)(int[], int);
    double tempo_execucao;
    long comparacoes;
    long trocas;
} ResultadoBenchmark;

typedef enum {
    ALEATORIO,
    ORDENADO,
    REVERSO,
    QUASE_ORDENADO,
    MUITAS_REPETICOES
} TiposDados;

void gerar_dados_teste(int array[], int n, TiposDados tipo) {
    switch (tipo) {
        case ALEATORIO:
            for (int i = 0; i < n; i++)
                array[i] = rand() % (n * 10);
            break;

        case ORDENADO:
            for (int i = 0; i < n; i++)
                array[i] = i;
            break;

        case REVERSO:
            for (int i = 0; i < n; i++)
                array[i] = n - i;
            break;

        case QUASE_ORDENADO:
            for (int i = 0; i < n; i++)
                array[i] = i;
            // faz algumas trocas aleatórias
            for (int i = 0; i < n / 10; i++) {
                int pos1 = rand() % n;
                int pos2 = rand() % n;
                int temp = array[pos1];
                array[pos1] = array[pos2];
                array[pos2] = temp;
            }
            break;

        case MUITAS_REPETICOES:
            for (int i = 0; i < n; i++)
                array[i] = rand() % 10; // Apenas 10 valores distintos
            break;
    }
}

double medir_tempo_algoritmo(void (*algoritmo)(int[], int),
                             int array[], int n) {
    clock_t inicio = clock();
    algoritmo(array, n);
    clock_t fim = clock();
    return ((double)(fim - inicio)) / CLOCKS_PER_SEC;
}

void executar_benchmark_completo(int tamanhos[], int num_tamanhos) {
    const char* nomes_tipos[] = { "Aleatório", "Ordenado", "Reverso",
                                   "Quase Ordenado", "Muitas Repetições" };

    ResultadoBenchmark algoritmos[] = {
        { "Bubble Sort", bubble_sort, 0, 0, 0 },
        { "Selection Sort", selection_sort, 0, 0, 0 },
        { "Insertion Sort", insertion_sort, 0, 0, 0 },
        { "Merge Sort", merge_sort_wrapper, 0, 0, 0 },
        { "Quick Sort", quick_sort_wrapper, 0, 0, 0 },
        { "Heap Sort", heap_sort, 0, 0, 0 }
    };

    printf("Benchmark de Algoritmos de Ordenação\n");
    printf("===== \n");

    for (int t = 0; t < num_tamanhos; t++) {
        int n = tamanhos[t];
        printf("Tamanho da array: %d elementos\n", n);
        printf("... \n");

        for (int tipo = 0; tipo < 5; tipo++) {
            printf("Tipo de Dados: %s\n", nomes_tipos[tipo]);

            for (int alg = 0; alg < 6; alg++) {
                int array_teste = malloc(n * sizeof(int));
                gerar_dados_teste(array_teste, n, (TiposDados)tipo);

                double tempo = medir_tempo_algoritmo(
                    algoritmos[alg].algoritmo, array_teste, n);

                printf("Algo: %s, %d segundos\n",
                    algoritmos[alg].nome, tempo);

                free(array_teste);
            }
        }
        printf("\n");
    }
}
```

# Resultados Experimentais Típicos

Performance para 100.000 elementos

Algoritmo	Aleatório	Ordenado	Reverso	Quase Ord.
Bubble Sort	15.23s	0.03s	30.45s	2.15s
Selection Sort	8.67s	8.66s	8.68s	8.65s
Insertion Sort	4.32s	0.02s	8.64s	0.48s
Merge Sort	0.018s	0.017s	0.018s	0.017s

## Resumo da Aula e Conceitos-Chave

### Principais Aprendizados

#### Fundamentos Matemáticos:

- **Limite inferior** de  $\Omega(n \log n)$  para algoritmos baseados em comparação
- **Relações de recorrência** e Teorema Master para análise
- **Análise de casos:** melhor, médio e pior

#### Algoritmos Implementados:

1. **Bubble Sort:**  $O(n^2)$  - didático, adaptativo
2. **Selection Sort:**  $O(n^2)$  - mínimo de trocas
3. **Insertion Sort:**  $O(n^2)$  - ótimo para dados pequenos/quase ordenados
4. **Merge Sort:**  $O(n \log n)$  - estável, previsível
5. **Quick Sort:**  $O(n \log n)$  avg - rápido na prática

#### Critérios de Escolha:

- **Tamanho dos dados** ( $n < 50$ : insertion,  $n > 10K$ : quick/merge)
- **Estabilidade** (merge sort quando necessária)
- **Memória disponível** (in-place vs. external)
- **Padrão dos dados** (quase ordenado: insertion)

## 💡 Dicas e Macetes Essenciais

### 🚀 Otimizações Práticas:

1. **Híbrido:** Insertion sort para subarrays pequenos
2. **Randomização:** Pivô aleatório no quick sort
3. **Deteção precoce:** Flag para arrays já ordenados
4. **Mediana-de-3:** Melhora escolha do pivô

### ⚠️ Pegadinhas Comuns:

- **Overflow:** Use `meio = baixo + (alto-baixo)/2`
- **Estabilidade:** Cuidado com `<` vs `<=` nas comparações
- **Recursão infinita:** Verificar condições de parada
- **Acesso fora dos limites:** Sempre validar índices

## Lista de Exercícios para Casa

### Exercícios Obrigatórios:

#### 1. Implementação Básica (★)

- Implemente os 5 algoritmos principais
- Adicione contadores de comparações e trocas
- Teste com arrays ordenados, reversos e aleatórios

#### 2. Análise Experimental (★★)

- Meça tempo de execução para diferentes tamanhos
- Compare resultados com análise teórica
- Identifique pontos de transição (quando um supera outro)

#### 3. Otimizações (★★★)

- Quick sort com mediana-de-3
- Merge sort iterativo (bottom-up)
- Insertion sort binário (busca binária para posição)






### Desafios Extras:

#### 4. Algoritmo Híbrido (★★★★)

- Combine técnicas para máxima eficiência
- Adapte automaticamente ao padrão dos dados

## Métricas de Avaliação

### Critérios para Implementações:

-  **Corretude:** Algoritmo ordena corretamente
-  **Eficiência:** Respeita complexidade teórica
-  **Clareza:** Código bem comentado e estruturado
-  **Robustez:** Trata casos extremos ( $n=0$ ,  $n=1$ )
-  **Análise:** Contadores e medições implementados

### Pontuação:

- **Básico (60%):** Implementação correta dos algoritmos
- **Intermediário (80%):** + Otimizações e análise
- **Avançado (100%):** + Híbridos e aplicações reais



## Referências e Material Complementar

### Literatura Fundamental:

1. **Cormen, T. H.** *Introduction to Algorithms*, 4ª ed. (Capítulos 2, 4, 6-8)
2. **Sedgewick, R.** *Algorithms*, 4ª ed. (Parte II: Sorting)
3. **Knuth, D. E.** *The Art of Computer Programming*, Vol. 3 (Sorting and Searching)

### Recursos Online:

- [Visualgo.net](#) - Visualização de algoritmos
- [Big-O Cheat Sheet](#) - Referência rápida
- [Sorting Algorithm Animations](#)

### Ferramentas para Prática:

- [LeetCode](#) - Problemas práticos
- [HackerRank](#) - Desafios de ordenação
- [Codeforces](#) - Competições

## Próxima Aula: Estruturas de Dados Dinâmicas

### Prévia do Conteúdo:

- **Listas Ligadas:** Simples, duplas, circulares
- **Pilhas e Filas:** Implementação e aplicações
- **Árvores Binárias:** Conceitos fundamentais
- **Hash Tables:** Função hash e tratamento de colisões

### Para se Preparar:

1. Revise conceitos de **ponteiros** e **alocação dinâmica**
2. Pratique **manipulação de estruturas** em C
3. Estude **análise amortizada** (opcional)




## ✓ Checklist da Aula

### Conceitos Dominados:

- ☐ Limite inferior teórico para ordenação
- ☐ Análise de complexidade de todos os algoritmos
- ☐ Implementação correta dos 5 algoritmos principais
- ☐ Critérios para escolha de algoritmos
- ☐ Otimizações e técnicas avançadas
- ☐ Aplicações em problemas reais

### Habilidades Desenvolvidas:

- ☐ Análise matemática de algoritmos
- ☐ Implementação eficiente em C
- ☐ Medição e comparação de performance
- ☐ Resolução de problemas práticos
- ☐ Otimização de código

-  **Entrega dos Exercícios:** Próxima aula
-  **Dúvidas:** Monitoria ou fórum online
-  **Contato:** [professor@universidade.edu.br](mailto:professor@universidade.edu.br)

*"A ordenação é a base de quase todos os algoritmos eficientes. Dominá-la é dominar a essência da computação."*

Obrigado pela atenção! 🎓

```
// Critério 3: Número de partidas (mais experiente)
if (j1->partidas_jogadas != j2->partidas_jogadas)
    return j2->partidas_jogadas - j1->partidas_jogadas;

// Critério 4: Atividade recente
return (j2->ultima_partida > j1->ultima_partida) ? 1 : -1;
```

```
}
```

```
void atualizar_ranking(Jogador jogadores[], int num_jogadores) {
```

```
// Usa qsort da biblioteca padrão (tipicamente introsort)
```

```
qsort(jogadores, num_jogadores, sizeof(Jogador), comparar_jogadores_ranking);
```

```
// Atualiza posições no ranking
for (int i = 0; i < num_jogadores; i++) {
    printf("Posição %d: %s (Pontos: %d, Taxa: %.2f%%)\n",
           i + 1, jogadores[i].nome, jogadores[i].pontuacao,
           jogadores[i].taxa_vitoria * 100);
}
```

```
}
```

```
### Sistema de Processamento de Log
```

```
```c
typedef struct {
    time_t timestamp;
    int id; // 15103 // DEBUG INFO WARNING ERROR
```

## 8. Algoritmos de Ordenação Externa

### Ordenação de Arquivos Grandes

**Problema:** Ordenar dados que não cabem na memória principal

**Solução:** External Merge Sort

```
#include <stdio.h>

#define TAMANHO_BUFFER 1000000 // 1 milhão de elementos por chunk

typedef struct {
    FILE *arquivo;
    int buffer[TAMANHO_BUFFER];
    int posicao_buffer;
    int tamanho_buffer;
    int fim_arquivo;
} FluxoArquivo;

void ordenacao_externa(const char *arquivo_entrada,
                      const char *arquivo_saida, long total_elementos) {
    FILE *entrada = fopen(arquivo_entrada, "rb");
    int num_arquivos_temp = 0;

    // Fase 1: Divide em chunks ordenados
    char nome_temp[100];
    while (!feof(entrada)) {
        int buffer[TAMANHO_BUFFER];
        int elementos_lidos = fread(buffer, sizeof(int), TAMANHO_BUFFER, entrada);

        if (elementos_lidos > 0) {
            // Ordena chunk na memória
            qsort(buffer, elementos_lidos, sizeof(int), comparar_inteiros);

            // Salva chunk ordenado
            sprintf(nome_temp, "temp_%d.dat", num_arquivos_temp);
            FILE *temp = fopen(nome_temp, "wb");
            fwrite(buffer, sizeof(int), elementos_lidos, temp);
            fclose(temp);

            num_arquivos_temp++;
        }
    }
    fclose(entrada);

    // Fase 2: Merge dos arquivos temporários
    merge_arquivos_temporarios(arquivo_saida, num_arquivos_temp);

    // Limpeza
    for (int i = 0; i < num_arquivos_temp; i++) {
        sprintf(nome_temp, "temp_%d.dat", i);
        remove(nome_temp);
    }
}

void merge_arquivos_temporarios(const char *arquivo_saida, int num_arquivos) {
    FILE *saida = fopen(arquivo_saida, "wb");
    FluxoArquivo fluxos[num_arquivos];

    // Inicializa fluxos de entrada
    for (int i = 0; i < num_arquivos; i++) {
        char nome_temp[100];
        sprintf(nome_temp, "temp_%d.dat", i);
        fluxos[i].arquivo = fopen(nome_temp, "rb");
        carregar_proximo_elemento(&fluxos[i]);
    }

    // Merge usando heap para eficiência
    while (tem_elementos_restantes(fluxos, num_arquivos)) {
        int indice_menor = encontrar_menor_elemento(fluxos, num_arquivos);

        // Escreve menor elemento no arquivo de saída
        fwrite(&fluxos[indice_menor].buffer[fluxos[indice_menor].posicao_buffer],
```

## 9. Conclusões e Próximos Passos

### Guia de Seleção de Algoritmos

#### Para Arrays Pequenos ( $n < 50$ ):

- **Insertion Sort:** Simples e eficiente
- **Selection Sort:** Mínimo número de trocas

#### Para Arrays Médios/Grandes ( $n > 50$ ):

- **Quick Sort:** Melhor performance média
- **Merge Sort:** Performance garantida e estável
- **Heap Sort:** Quando espaço é limitado

#### Para Dados Especiais:

- **Counting Sort:** Inteiros em range pequeno
- **Radix Sort:** Inteiros ou strings
- **TimSort:** Dados parcialmente ordenados

### Preparação para Próximas Aulas

#### Aula 04: Estruturas de Dados Avançadas

- Árvores Binárias de Busca e AVL
- Hash Tables e Funções de Dispersão
- Grafos: Representação e Algoritmos Básicos

## Bibliografia e Recursos

### Referências Clássicas

- **Cormen, T. H. et al.** *Introduction to Algorithms*, 4ª edição
- **Sedgewick, R.** *Algorithms*, 4ª edição
- **Knuth, D. E.** *The Art of Computer Programming*, Volume 3

### Implementações de Referência

- **GNU libc `qsort()`**: Implementação industrial
- **Java `Arrays.sort()`**: TimSort híbrido
- **C++ `std::sort()`**: Introsort otimizado

### Ferramentas de Análise

- **Complexity Analyzer**: Medição automática de complexidade
- **Profilers**: gprof, Valgrind, Cachegrind
- **Visualizadores**: Algorithm Visualizer, Sorting Algorithms Animations



## Encerramento da Aula

### Algoritmos e Complexidade - Aula 03

*Algoritmos de Ordenação e Análise de Performance*

**Próxima Aula:** Estruturas de Dados Avançadas - Árvores e Hash Tables

**Exercícios:** Implementar e comparar 3 algoritmos de ordenação diferentes

### Material Complementar

**GitHub:** [github.com/cordeirotelecom/algoritimos\\_e\\_complexidade](https://github.com/cordeirotelecom/algoritimos_e_complexidade)

**Simuladores Online:** VisuAlgo, Algorithm-Visualizer

**Prática:** LeetCode Sorting Problems

