Algoritmos de Ordenação

Introdução e Análise de Complexidade

Aulas 1 e 2

Estrutura de Dados - 3º Semestre ADS

Boas-vindas e Apresentação

Sobre o Módulo

Este módulo de **Algoritmos de Ordenação** faz parte da disciplina de Estrutura de Dados e abordará:

- Conceitos fundamentais de ordenação
- Análise de complexidade algorítmica
- Implementação em C++ básico
- Comparação entre diferentes algoritmos
- Aplicações práticas e exercícios

Objetivos de Aprendizagem

Ao final deste módulo, você será capaz de:

- Compreender o funcionamento dos principais algoritmos de ordenação
- Analisar a complexidade de tempo e espaço
- Implementar algoritmos de ordenação em C++
- Escolher o algoritmo mais adequado para cada cenário

Agenda das Aulas 1 e 2

Aula 1

Introdução aos Algoritmos de Ordenação

- O que é ordenação e por que é importante
- Aplicações práticas
- Classificação dos algoritmos

Aula 2

Análise de Complexidade

- Revisão da Notação Big O
- Classes de complexidade
- Análise de casos (melhor, médio, pior)
- Análise de código simples

Nas próximas aulas, começaremos a implementar nosso primeiro algoritmo de ordenação: o Bubble Sort.

O Que é Ordenação?

Definição

Ordenação é o processo de reorganizar uma coleção de elementos em uma determinada ordem - crescente, decrescente ou segundo algum outro critério.

Conceito Fundamental

A ordenação é uma das operações mais básicas e importantes na ciência da computação, servindo como base para muitos outros algoritmos.

Podemos ordenar praticamente qualquer tipo de dado que possa ser comparado: números, strings, datas, objetos complexos (usando uma chave de comparação).

Exemplo Intuitivo

Imagine organizar cartas de baralho na sua mão:

Características da Ordenação

Propriedades

- **Determinística:** Produz sempre o mesmo resultado para a mesma entrada
- **Comparativa:** Baseada em comparações entre elementos
- **Transformativa:** Reorganiza os dados sem alterar seu conteúdo

Critérios de Ordenação

- **Crescente (Ascendente):** Do menor para o maior valor
- **Decrescente (Descendente):** Do maior para o menor valor
- **Lexicográfica:** Ordem alfabética para strings
- **Personalizada:** Baseada em critérios específicos

Embora pareça simples conceitualmente, a ordenação eficiente de grandes conjuntos de dados é um desafio computacional significativo.

Por Que Ordenar Dados?

Aplicações Práticas da Ordenação

A ordenação é uma operação fundamental que melhora significativamente a eficiência de muitos processos computacionais:

Busca Eficiente

Dados ordenados permitem o uso de algoritmos de busca muito mais eficientes, como a **Busca Binária** (O(log n)).

- Buscar em dados não ordenados: **O(n)**
- Buscar em dados ordenados: **O(log n)**

Processamento de Dados

Facilita operações como encontrar duplicatas, medianas, valores mínimos/máximos e intervalos específicos.

- Bancos de dados usam índices ordenados para acelerar consultas e junções.

Visualização

Dados ordenados são mais fáceis de visualizar e interpretar em relatórios, gráficos e interfaces de usuário.

- Listas ordenadas melhoram significativamente a experiência do usuário.

Detecção de Anomalias

Facilita a identificação de valores atípicos (outliers) e padrões incomuns nos dados.

- Útil em sistemas de segurança e monitoramento de qualidade.

Compressão

Alguns algoritmos de compressão funcionam melhor com dados ordenados ou pré-processados.

- Ordenar pode revelar padrões que aumentam a taxa de compressão.

Algoritmos Complexos

Muitos algoritmos avançados dependem de dados ordenados como pré-requisito.

- Exemplos: algoritmos de grafos, geometria computacional, etc.

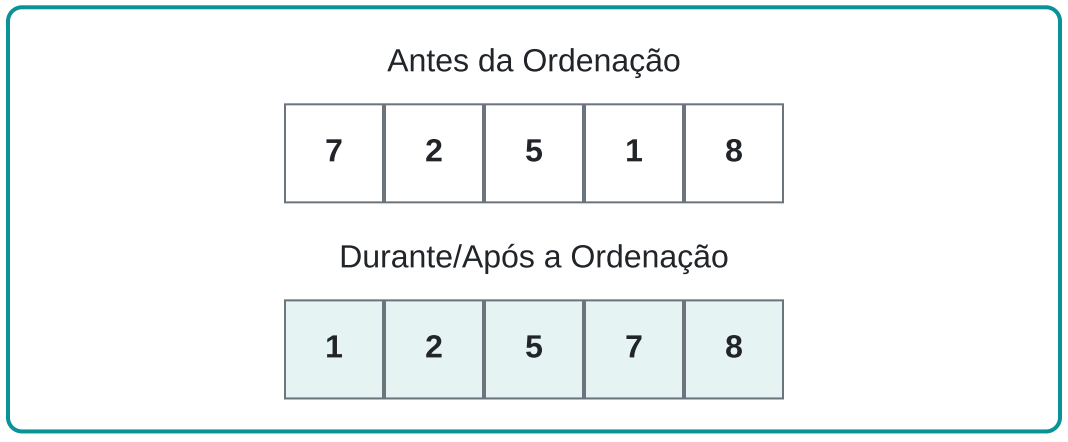
Classificação: Uso de Memória

Algoritmos de Ordenação e Uso de Memória

Uma forma importante de classificar algoritmos de ordenação é pelo seu uso de memória auxiliar durante o processo de ordenação:

In-place

Algoritmos **in-place** realizam a ordenação diretamente no array original, usando apenas uma quantidade constante (O(1)) ou muito pequena de memória auxiliar.

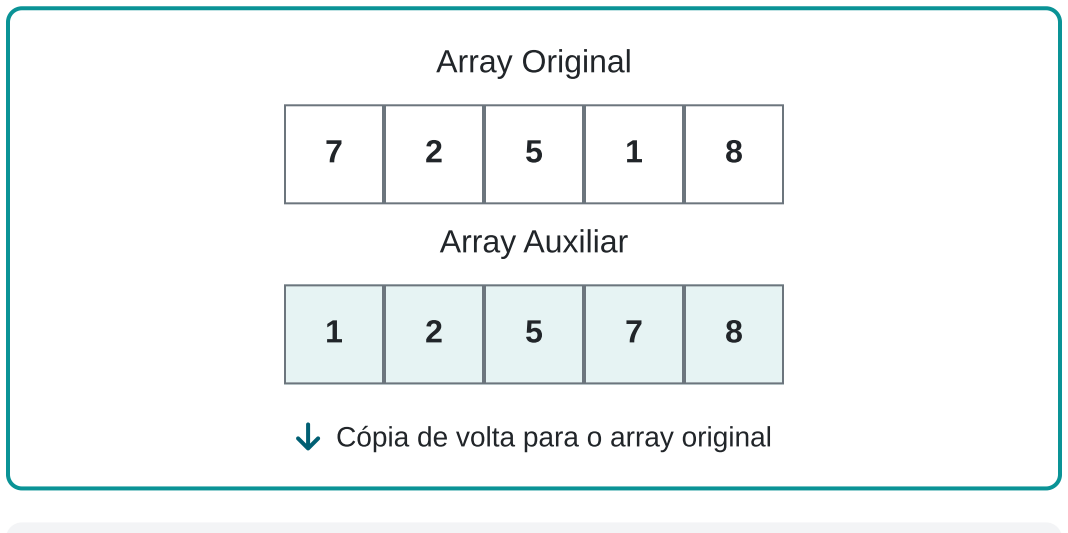


Exemplos:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Heap Sort

Not-in-place

Algoritmos **not-in-place** requerem espaço adicional proporcional ao tamanho da entrada (geralmente O(n)) para realizar a ordenação.



Exemplos:

- Merge Sort
- Counting Sort
- Radix Sort

Comparação

Característica	In-place	Not-in-place
Uso de memória	O(1) ou muito pequeno	O(n) ou mais
Vantagem principal	Economia de memória	Geralmente mais rápidos
Desvantagem	Pode ser mais lento	Maior consumo de memória
Ideal para	Dispositivos com memória limitada	Quando a velocidade é prioritária

Classificação: Estabilidade

O Que é Estabilidade em Algoritmos de Ordenação?

A **estabilidade** é uma propriedade que determina se elementos com chaves iguais mantêm sua ordem relativa original após a ordenação.

Algoritmo Estável

Considere cartas com valores e naipes:

5♥ vem antes de 5♠ na entrada e na saída

Algoritmo Instável

Mesmo exemplo, mas com algoritmo instável:

5♥ vem antes de 5♠ na entrada, mas **5♠ vem antes de 5♥** na saída

Algoritmos Estáveis

Mantêm a ordem relativa de registros com chaves iguais.

Exemplos:

- Bubble Sort
- Insertion Sort
- Merge Sort
- Counting Sort (se implementado corretamente)

Algoritmos Instáveis

Não garantem a preservação da ordem relativa de registros com chaves iguais.

Exemplos:

- Selection Sort
- Heap Sort
- Quick Sort (implementação padrão)

Quando a Estabilidade Importa?

- Ordenações múltiplas sequenciais
- Ordenação por múltiplos critérios
- Preservação da ordem natural dos dados

Exemplo Prático

Ordenar uma lista de alunos:

1. Primeiro por turma (A, B, C)
2. Depois por nota (decrescente)

Se o algoritmo usado na segunda ordenação não for estável, a ordenação por turma será perdida!

A Importância da Eficiência

Por Que Medir a Eficiência dos Algoritmos?

Não basta apenas que um algoritmo resolva um problema; é crucial entender **como** ele o faz em termos de recursos computacionais.

Eficiência de Tempo

Mede quanto tempo um algoritmo leva para executar em função do tamanho da entrada.

Perguntas-chave:

- Quanto tempo leva para ordenar 1.000 elementos?
- E se aumentarmos para 1.000.000?
- Como o tempo cresce em relação ao tamanho da entrada?

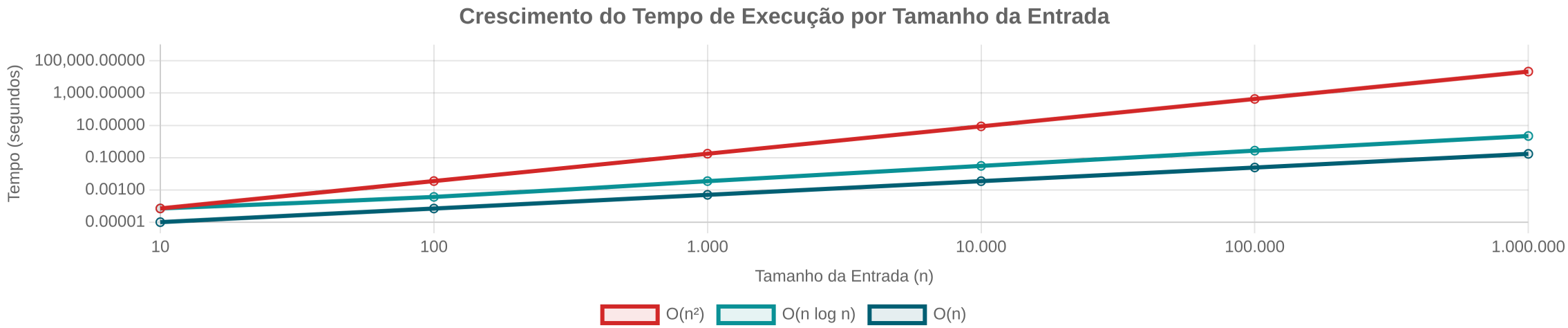
Eficiência de Espaço

Mede quanta memória adicional um algoritmo requer para executar em função do tamanho da entrada.

Perguntas-chave:

- Quanta memória extra é necessária?
- O algoritmo trabalha in-place ou precisa de arrays auxiliares?
- Como o uso de memória escala com o tamanho da entrada?

Impacto da Eficiência na Prática



Escala

Um algoritmo que funciona bem para 100 elementos pode ser impraticável para 1 milhão.

Trade-offs

Frequentemente há um equilíbrio entre eficiência de tempo e espaço.

Contexto

A escolha do algoritmo deve considerar o cenário específico e as restrições.

Revisão: Notação Big O

O Que é Big O?

A **Notação Big O** é uma forma matemática de descrever o comportamento assintótico de funções, especialmente útil para analisar a eficiência de algoritmos.

Ela nos dá um **limite superior** para a taxa de crescimento de uma função, focando no termo dominante e ignorando constantes e termos de menor ordem.

O Que Big O Representa?

- **Não** representa o tempo exato de execução
- **Sim** representa como o algoritmo escala com o tamanho da entrada
- Descreve o **pior cenário** de desempenho
- Permite **comparar** algoritmos independentemente de hardware ou linguagem

Notação Formal

Dizemos que f(n) = O(g(n)) se existem constantes positivas c e n_0 tais que:

$$0 \leq f(n) \leq c \cdot g(n) \text{ para todo } n \geq n_0$$

Ou seja, g(n) é um limite superior para f(n) a partir de um certo ponto n_0.

Simplificando Big O

Regras para Simplificar:

1. Remover constantes multiplicativas:

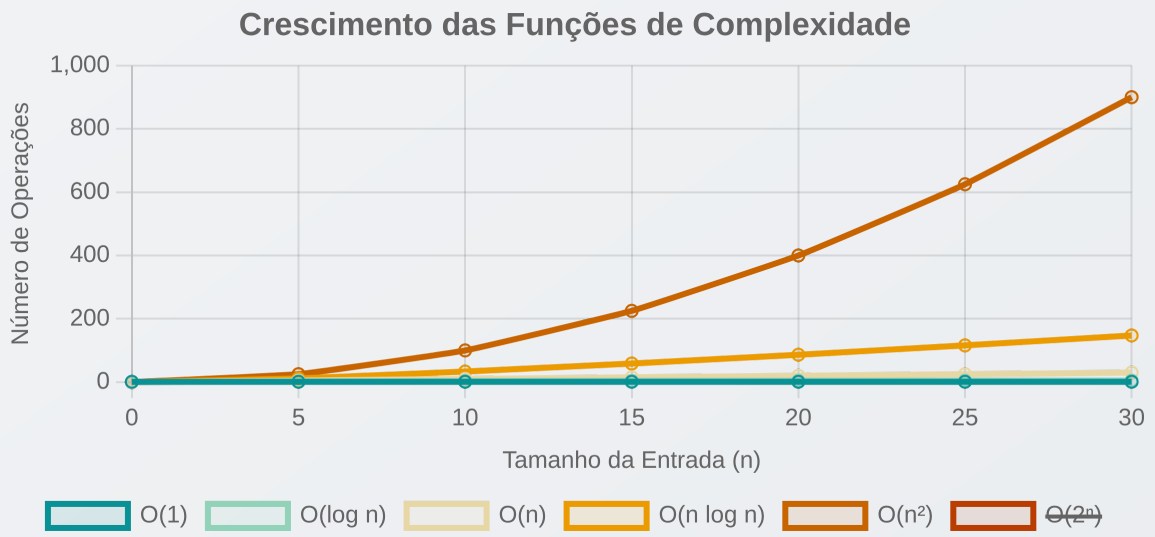
$$O(5n) = O(n)$$

2. Manter apenas o termo de maior crescimento:

$$O(n^2 + 3n + 2) = O(n^2)$$

3. Simplificar funções compostas:

$$O(2^n + n^2) = O(2^n)$$



A diferença entre $O(n^2)$ e $O(n \log n)$ pode parecer pequena para entradas pequenas, mas torna-se enorme para grandes volumes de dados.

Classes de Complexidade (Parte 1)

Complexidades Comuns em Algoritmos

Vamos explorar as classes de complexidade mais eficientes:

O(1)

Constante

O tempo de execução não depende do tamanho da entrada.

```
int primeiro = array[0];
```

Exemplos: acesso a elemento de array por índice, operações aritméticas simples.

O(log n)

Logarítmica

O tempo cresce logaritmicamente com o tamanho da entrada.

```
while (n > 1) {
  n = n / 2;
}
```

Exemplos: busca binária, alguns algoritmos de divisão e conquista.

O(n)

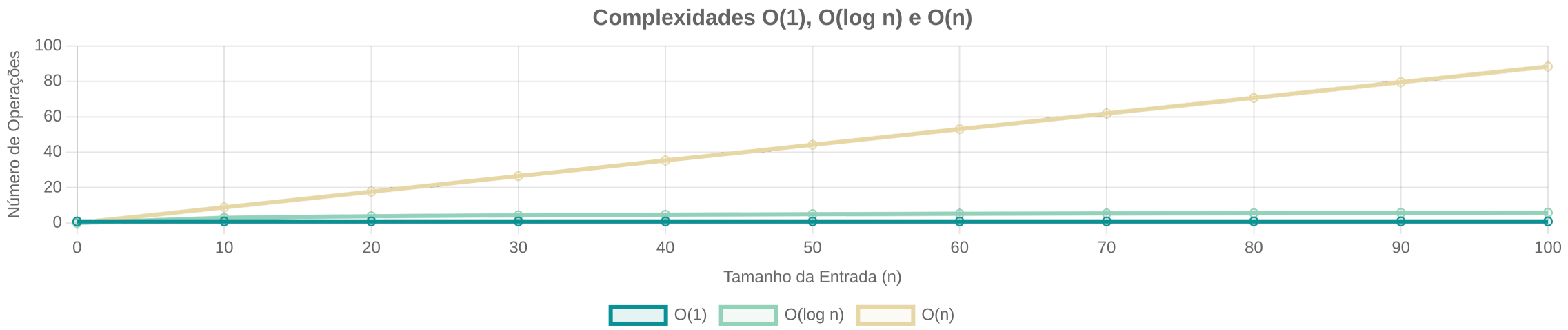
Linear

O tempo cresce linearmente com o tamanho da entrada.

```
for (int i = 0; i < n; i++) {
  // operação O(1)
}
```

Exemplos: busca linear, percorrer um array uma vez.

Visualização Comparativa



Classes de Complexidade (Parte 2)

Complexidades Intermediárias

Estas classes são especialmente relevantes para algoritmos de ordenação:

O(n log n)

Linearítmica

Cresce mais rápido que linear, mas significativamente mais lento que quadrático.

```
// Exemplo conceitual de Merge Sort
void mergeSort(int arr[], int l, int r) {
  if (l < r) {
    int m = l + (r - l) / 2;
    mergeSort(arr, l, m);      // T(n/2)
    mergeSort(arr, m + 1, r);  // T(n/2)
    merge(arr, l, m, r);       // O(n)
  }
}
```

Algoritmos eficientes de ordenação como Merge Sort, Heap Sort e Quick Sort (caso médio) têm esta complexidade.

O(n²)

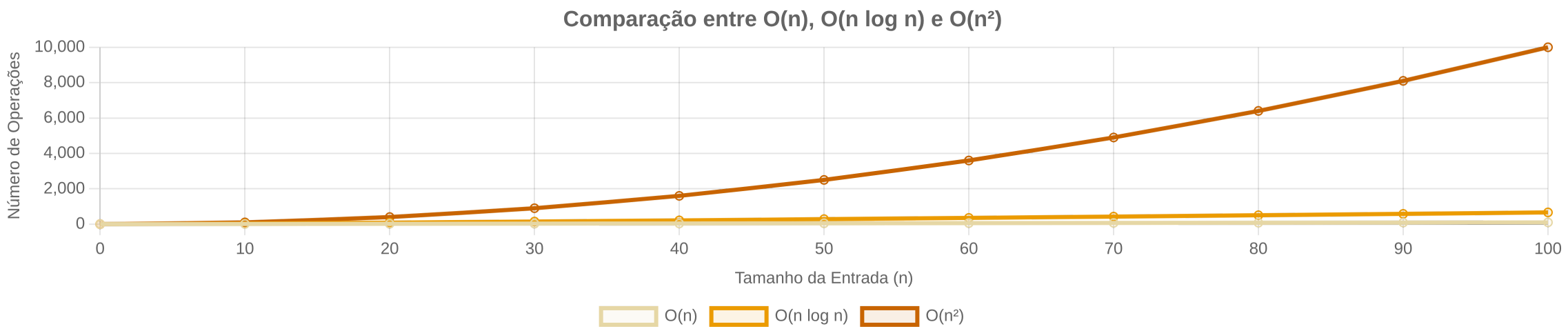
Quadrática

O tempo cresce com o quadrado do tamanho da entrada.

```
// Exemplo de Bubble Sort
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n - i - 1; j++) {
    if (arr[j] > arr[j + 1]) {
      // troca elementos
    }
  }
}
```

Algoritmos simples de ordenação como Bubble Sort, Selection Sort e Insertion Sort têm esta complexidade.

Comparação Visual



Para conjuntos de dados pequenos, a diferença entre O(n log n) e O(n²) pode ser imperceptível. No entanto, à medida que o tamanho dos dados aumenta, a diferença torna-se dramática.

Classes de Complexidade (Parte 3)

Complexidades Maiores

Estas classes representam algoritmos com crescimento muito rápido, geralmente inviáveis para grandes entradas:

$O(n^3)$

Cúbica

O tempo cresce com o cubo do tamanho da entrada.

```
// Três loops aninhados
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
      // operação O(1)
    }
  }
}
```

Exemplos: alguns algoritmos de multiplicação de matrizes, certos algoritmos de grafos.

$O(2^n)$

Exponencial

O tempo dobra (ou mais) a cada elemento adicional na entrada.

```
// Fibonacci recursivo ingênuo
int fib(int n) {
  if (n <= 1) return n;
  return fib(n-1) + fib(n-2);
}
```

Exemplos: algoritmos de força bruta, alguns algoritmos recursivos sem memoização.

$O(n!)$

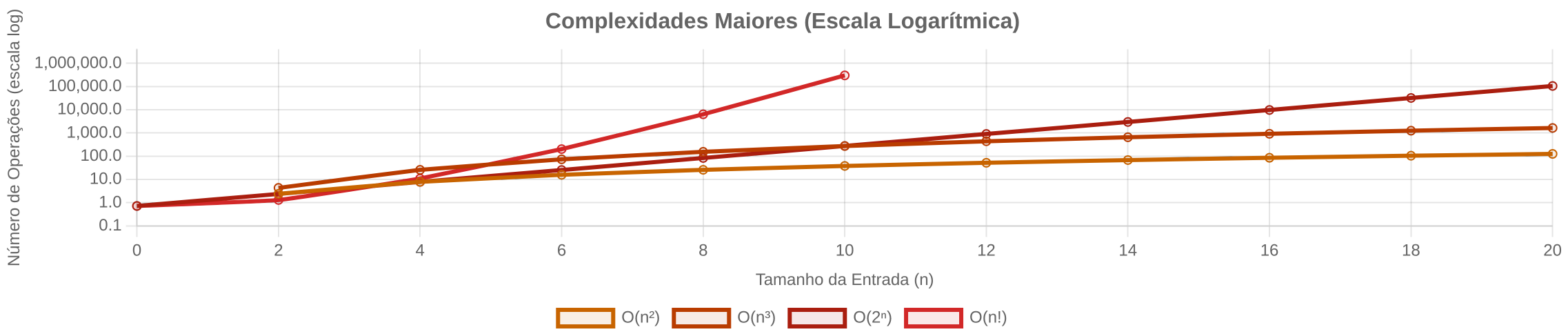
Fatorial

O tempo cresce fatorialmente com o tamanho da entrada.

```
// Gerar todas as permutações
void permute(char* a, int l, int r) {
  if (l == r) {
    printf("%s\n", a);
  } else {
    for (int i = l; i <= r; i++) {
      swap(a[l], a[i]);
      permute(a, l+1, r);
      swap(a[l], a[i]); // backtrack
    }
  }
}
```

Exemplos: problema do caixeiro viajante por força bruta, geração de permutações.

Crescimento Explosivo



Algoritmos com complexidade $O(2^n)$ ou $O(n!)$ são geralmente impraticáveis para entradas maiores que 20-30 elementos. Para problemas que naturalmente levam a essas complexidades, buscamos aproximações ou heurísticas.

Análise de Casos

Cenários de Análise de Algoritmos

Ao analisar a complexidade de um algoritmo, consideramos diferentes cenários de entrada:

Melhor Caso

O cenário em que o algoritmo executa o **menor número de operações** possível.

Exemplo (Bubble Sort):

Array já ordenado: O(n)
(com otimização de flag)

Pior Caso

O cenário em que o algoritmo executa o **maior número de operações** possível.

Exemplo (Bubble Sort):

Array em ordem inversa: $O(n^2)$
(máximo de comparações e trocas)

Caso Médio

O desempenho **esperado** para uma entrada aleatória ou típica.

Exemplo (Bubble Sort):

Array com elementos aleatórios: $O(n^2)$
(análise probabilística)

Comparação de Casos para Algoritmos de Ordenação

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Espaço
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)
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(n+k)

* k representa o intervalo de valores no Counting Sort

*Na prática, frequentemente nos concentramos na análise de **pior caso**, pois ela fornece uma garantia de desempenho. No entanto, entender o melhor caso e o caso médio é crucial para escolher o algoritmo mais adequado para cenários específicos.*

Analisando Código Simples

Como Analisar a Complexidade de Trechos de Código

Vamos aplicar os conceitos de Big O para analisar a complexidade de algoritmos simples:

Loop Simples

```
// Soma de elementos em um array
int soma = 0;
for (int i = 0; i < n; i++) {
    soma += arr[i];  // O(1)
}
```

Complexidade: O(n)

- O loop executa exatamente n vezes
- Cada operação dentro do loop é O(1)
- Total: n × O(1) = O(n)

Loops Aninhados

```
// Encontrar pares com soma específica
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (arr[i] + arr[j] == alvo) {  // O(1)
            // encontrou par
        }
    }
}
```

Complexidade: O(n²)

- Loop externo executa n vezes
- Para cada iteração externa, o loop interno executa n vezes
- Total: n × n × O(1) = O(n²)

Loops Consecutivos

```
// Dois processamentos separados
for (int i = 0; i < n; i++) {
    // processo 1 - O(1)
}

for (int j = 0; j < n; j++) {
    // processo 2 - O(1)
}
```

Complexidade: O(n)

- Primeiro loop: O(n)
- Segundo loop: O(n)
- Total: O(n) + O(n) = O(2n) = O(n)

Constantes são removidas na notação Big O

Loop com Divisão

```
// Dividindo o problema pela metade
int i = n;
while (i > 0) {
    // operação O(1)
    i = i / 2;
}
```

Complexidade: O(log n)

- Cada iteração divide i por 2
- Número de iterações: $\log_2(n)$
- Total: $\log_2(n)$ × O(1) = O(log n)

Dicas para Análise de Complexidade

Regras Básicas:

- Operações simples (atribuição, comparação, aritmética): O(1)
- Loop único de 0 a n: O(n)
- Loops aninhados: multiplicar complexidades
- Loops consecutivos: somar complexidades
- Divisão repetida por constante: O(log n)

Lembre-se:

- Considere sempre o termo de maior crescimento
- Ignore constantes multiplicativas
- Analise o pior caso quando não especificado
- Recursão: T(n) = a·T(n/b) + f(n)
- Considere o tamanho da entrada, não o valor

Próximas Aulas

O Que Vem Por Aí

Agora que compreendemos os conceitos fundamentais de ordenação e análise de complexidade, nas próximas aulas vamos mergulhar nos algoritmos específicos:

Aula 3: Bubble Sort

- Conceito da "bolha" subindo
- Implementação passo a passo
- Análise de complexidade detalhada
- Visualização do algoritmo em ação

Aula 4: Bubble Sort Otimizado

- Otimização com flag de troca
- Comparação com versão padrão
- Exercícios práticos de implementação
- Discussão sobre vantagens e limitações

Visão Geral do Módulo

Semana 1

- Introdução e Complexidade
- **Bubble Sort**

Semana 2

- Selection Sort
- Insertion Sort
- Merge Sort

Semana 3

- Counting Sort
- Radix Sort
- Comparação e Revisão

Preparação para a Próxima Aula

Para aproveitar melhor a próxima aula, recomendamos:

- Revisar os conceitos de complexidade vistos hoje
- Praticar a análise de complexidade em trechos simples de código
- Pensar intuitivamente em como você ordenaria uma sequência de números