

Notação Big O

Entenda o que é, como e onde usar na programação para otimizar seus algoritmos

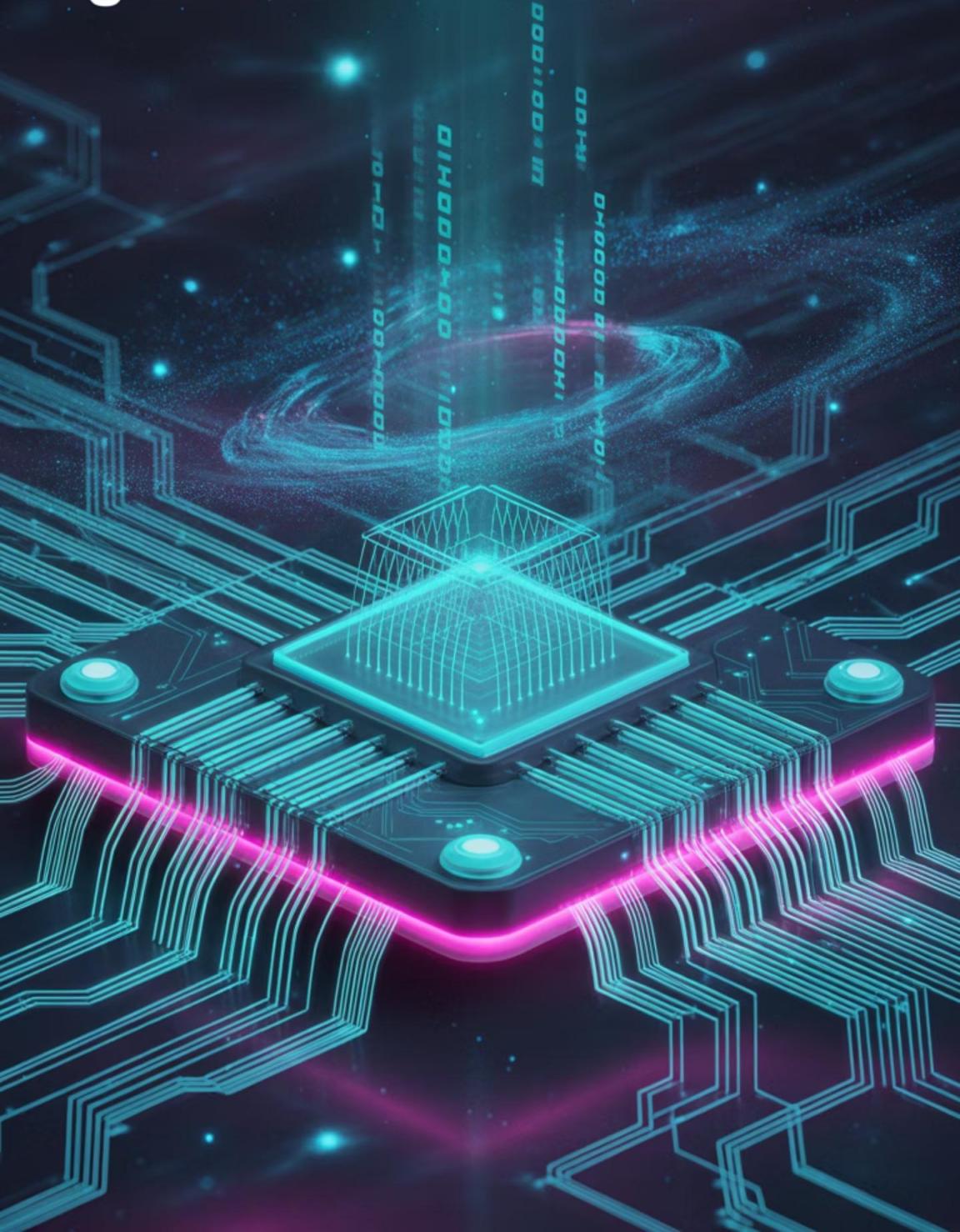


Algorithms and Data Structures

Capítulo 1

Introdução à Notação Big O

Compreenda os fundamentos da análise de complexidade algorítmica e sua importância no desenvolvimento de software



O que é Notação Big O?

Comportamento Assintótico

Representa como o tempo ou espaço de um algoritmo cresce conforme o tamanho da entrada aumenta

Análise do Pior Caso

Foca na situação mais desfavorável para medir a eficiência em função do tamanho da entrada (n)

Por que Big O importa?

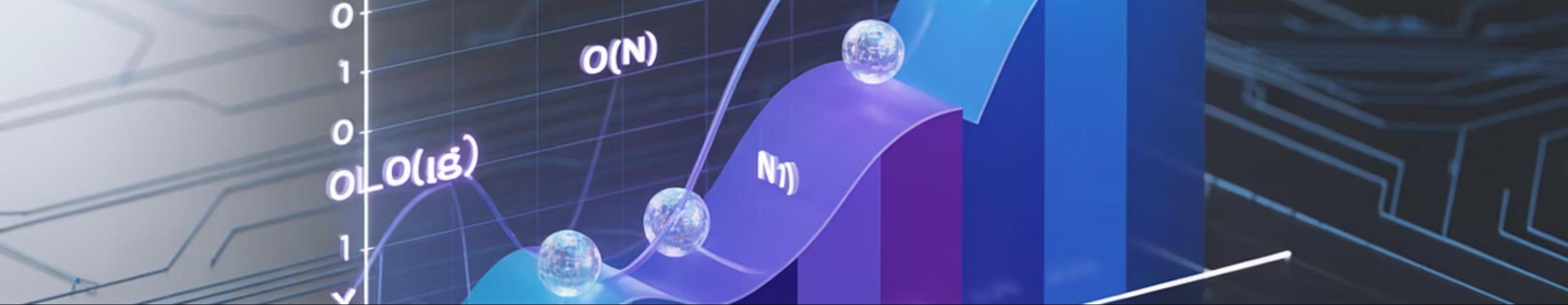


Comparação Universal

Permite comparar algoritmos independentemente da máquina ou linguagem utilizada

Predição de Escalabilidade

Ajuda a prever como o desempenho se comportará com grandes volumes de dados



Crescimento das Funções

Visualização do crescimento das principais complexidades algorítmicas: desde constante até exponencial

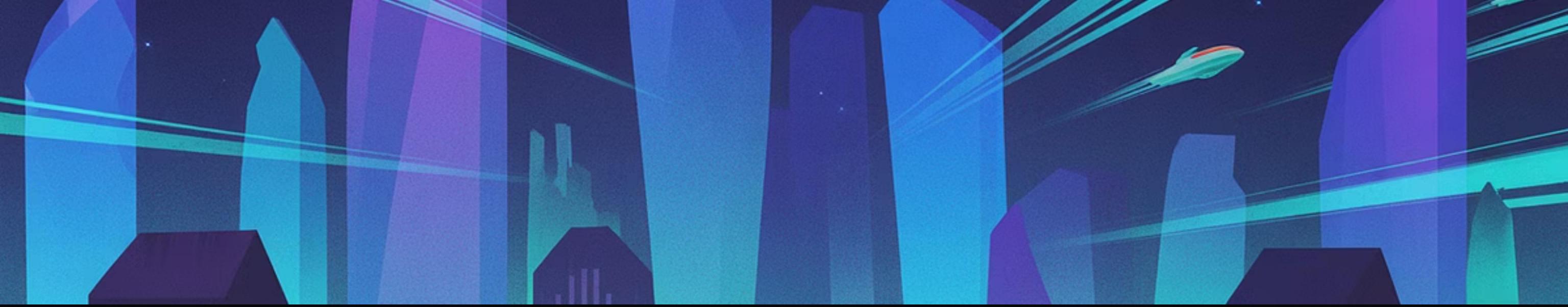
Como ler Big O?

Notação $O(f(n))$

Significa que o tempo ou complexidade cresce no máximo como $f(n)$ quando n tende ao infinito

Simplificação

Ignora constantes e termos de menor ordem para focar apenas no fator dominante



Capítulo 2

Exemplos Práticos de Big O

Explore casos reais de diferentes complexidades algorítmicas

$O(1)$ - Tempo Constante



Características

- Tempo de execução independe do tamanho da entrada
- Operação sempre rápida e previsível

✓ Exemplo: acessar um elemento pelo índice em um array

`lista[0]` sempre leva o mesmo tempo

$O(n)$ - Tempo Linear

01

Definição

Tempo de execução cresce
proporcionalmente ao tamanho da
entrada

02

Exemplo Prático

Percorrer todos os elementos de uma
lista para calcular a soma

03

Implementação

Loop que visita cada elemento
exatamente uma vez

$O(n^2)$ - Tempo Quadrático

Características

Tempo cresce como o quadrado do tamanho da entrada. Comum em algoritmos com dois loops aninhados.

- | Para cada elemento, verifica todos os outros elementos da lista

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        System.out.println("Comparando elemento " + i + " com " + j);  
    }  
}
```



$O(\log n)$ - Tempo Logarítmico

Busca Binária

Algoritmo clássico que divide o problema pela metade a cada passo

Eficiência

Extremamente eficiente para grandes conjuntos de dados ordenados

Aplicações

Árvores平衡adas, estruturas de dados hierárquicas

$O(n \log n)$ - Tempo Linearítmico

Merge Sort

Divide a lista em sublistas menores, ordena cada uma e depois combina os resultados

Heap Sort

Constrói uma heap e extrai elementos em ordem, mantendo a propriedade da heap

Quick Sort (caso médio)

Particiona a lista em torno de um pivô e ordena recursivamente



Busca Binária em Ação

Implementação clássica da busca binária demonstrando como o espaço de busca é reduzido pela metade a cada iteração

Capítulo 3

Como Calcular o Big O?

Metodologia passo a passo para determinar a complexidade de qualquer algoritmo



Passos para Determinar Big O



1. Identificar Estruturas

Localizar loops, recursões e aninhamentos no código

2. Contar Operações

Identificar as operações que dominam o tempo de execução

3. Simplificar

Descartar constantes e manter apenas o termo de maior ordem

Exemplo: Loops Aninhados

```
for i in range(n):      # Loop externo: n vezes    for j  
    in range(n): # Loop interno: n vezes        print(i, j)  
# Operação O(1)
```

n

Iterações Externas

Análise: O loop externo executa n vezes, e para cada iteração, o loop interno também executa n vezes.

n^2

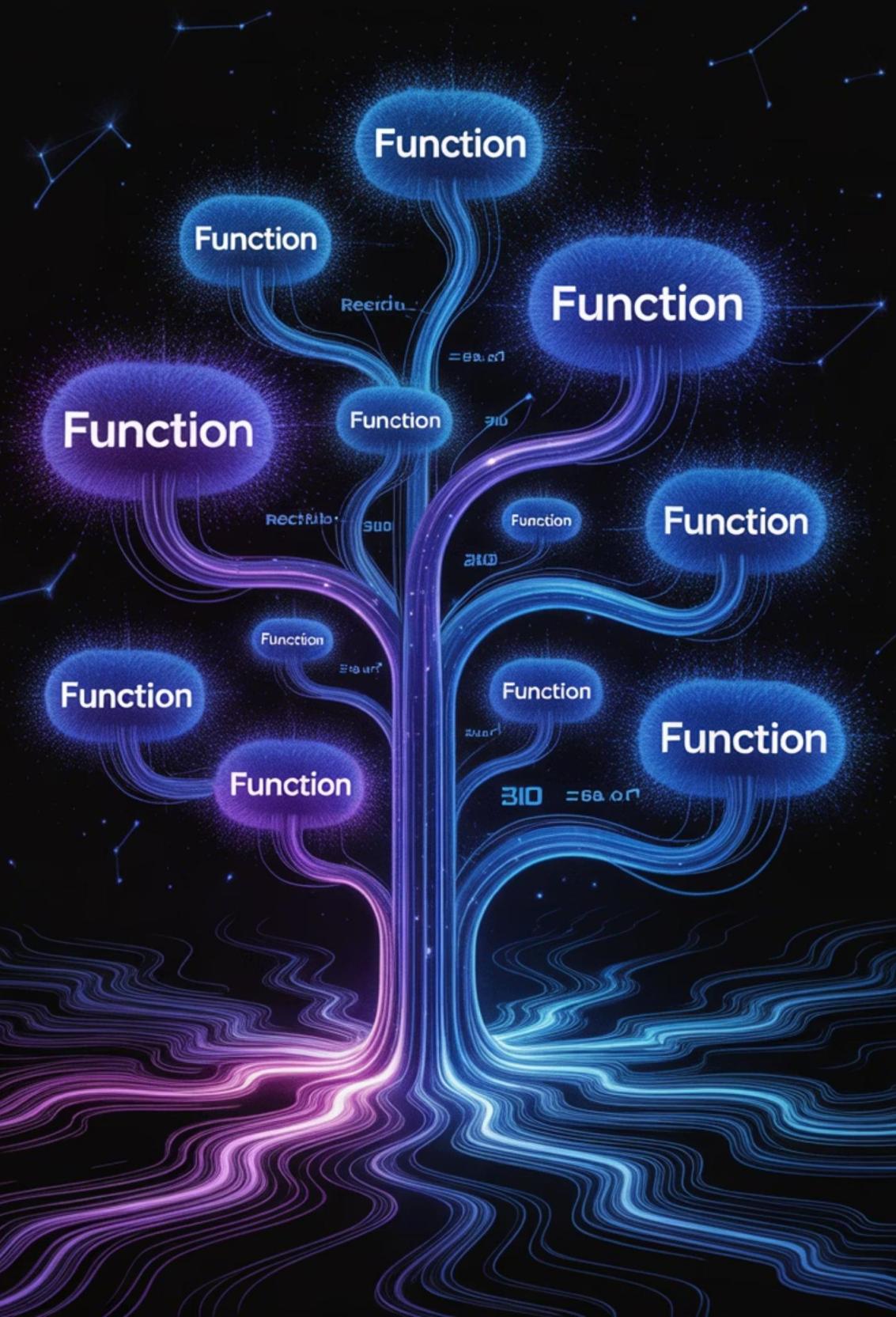
Total de Operações



Resultado: $O(n^2)$

Exemplo: Recursão

Para analisar recursão, observe o número de chamadas e a profundidade da árvore de recursão. Multiplique estes fatores para obter a complexidade total.



Capítulo 4

Propriedades da Notação Big O

Fundamentos matemáticos que regem o comportamento da análise assintótica



Propriedade 1: Reflexividade

Definição

Toda função é limitada por ela mesma

$$f(n) = O(f(n))$$

Esta propriedade estabelece que qualquer função tem pelo menos ela mesma como limite superior assintótico.

Propriedade 2: Transitividade

1

Se

$$f(n) = O(g(n))$$

2

E

$$g(n) = O(h(n))$$

3

Então

$$f(n) = O(h(n))$$

Permite encadear relações de complexidade para estabelecer limites transitivos.

Propriedade 3: Fator Constante

Regra Fundamental

Constantes multiplicativas não alteram a ordem de crescimento assintótico

$$c \cdot f(n) = O(f(n))$$

onde c é uma constante positiva



Propriedade 4: Regra da Soma



Adição de Complexidades

$$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$



Termo Dominante

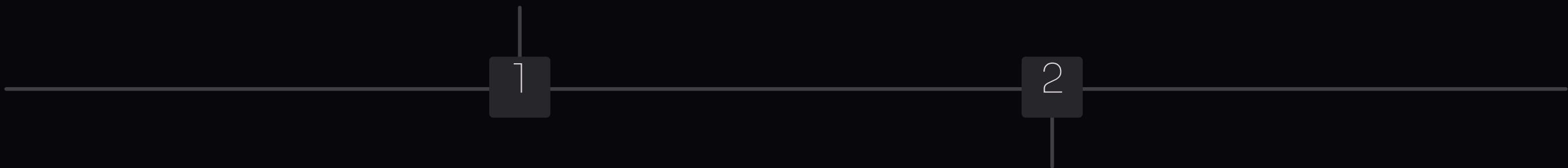
O termo de maior ordem domina a complexidade total

Exemplo prático: $O(n) + O(n^2) = O(n^2)$, pois n^2 cresce mais rapidamente que n

Propriedade 5: Regra do Produto

Multiplicação

$$O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$$



Loops Aninhados

Útil para analisar estruturas aninhadas

- Aplicação: Um loop $O(n)$ dentro de outro loop $O(n)$ resulta em $O(n^2)$

Propriedade 6: Composição

Composição de Funções

Se $f(n) = O(g(n))$, então $f(h(n)) = O(g(h(n)))$

Esta propriedade é especialmente útil para analisar algoritmos com estruturas recursivas complexas.





Capítulo 5

Classes de Complexidade

Explorando as principais categorias de complexidade algorítmica com exemplos do mundo real

$O(1)$ - Tempo Constante



Acesso a Array

Buscar um elemento pelo índice em array ou lista. A posição é calculada diretamente.



Hash Table

Inserção e busca em tabelas hash bem dimensionadas com boa função de dispersão.



Stack Operations

Push e pop em pilhas - sempre operam no topo da estrutura.

$O(\log n)$ - Tempo Logarítmico

- Busca Binária

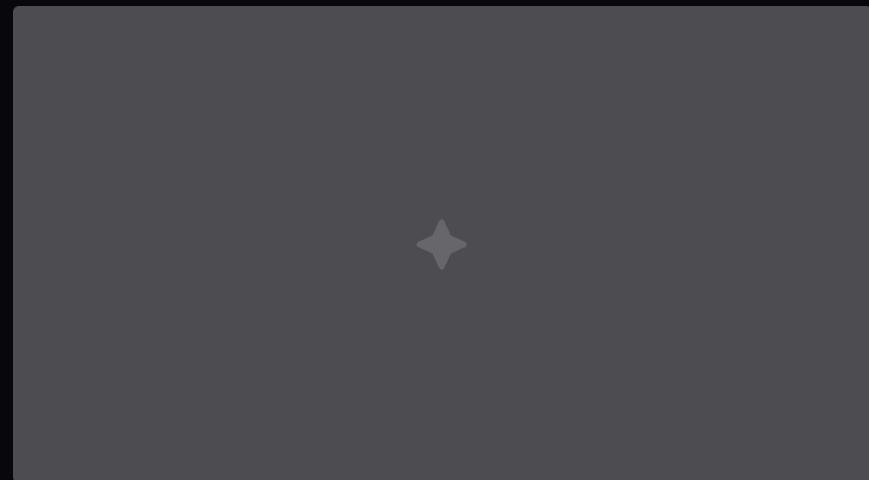
Pesquisa em arrays ordenados dividindo o espaço pela metade

- Árvores Balanceadas

Operações em BST, AVL e Red-Black trees

- Heap Operations

Inserção e remoção em heaps mantendo propriedades



Eficiência excepcional mesmo com milhões de elementos

$O(n)$ - Tempo Linear

Busca Linear

Percorrer lista não ordenada para encontrar um elemento específico

Soma de Elementos

Calcular soma, média ou encontrar máximo/mínimo em coleções

Cópia de Arrays

Duplicar ou transferir todos os elementos de uma estrutura

$O(n \log n)$ - Tempo Linearítmico

Merge Sort

Algoritmo de ordenação estável que divide e conquista, garantindo desempenho consistente

Heap Sort

Ordenação in-place usando estrutura de heap, excelente para memória limitada

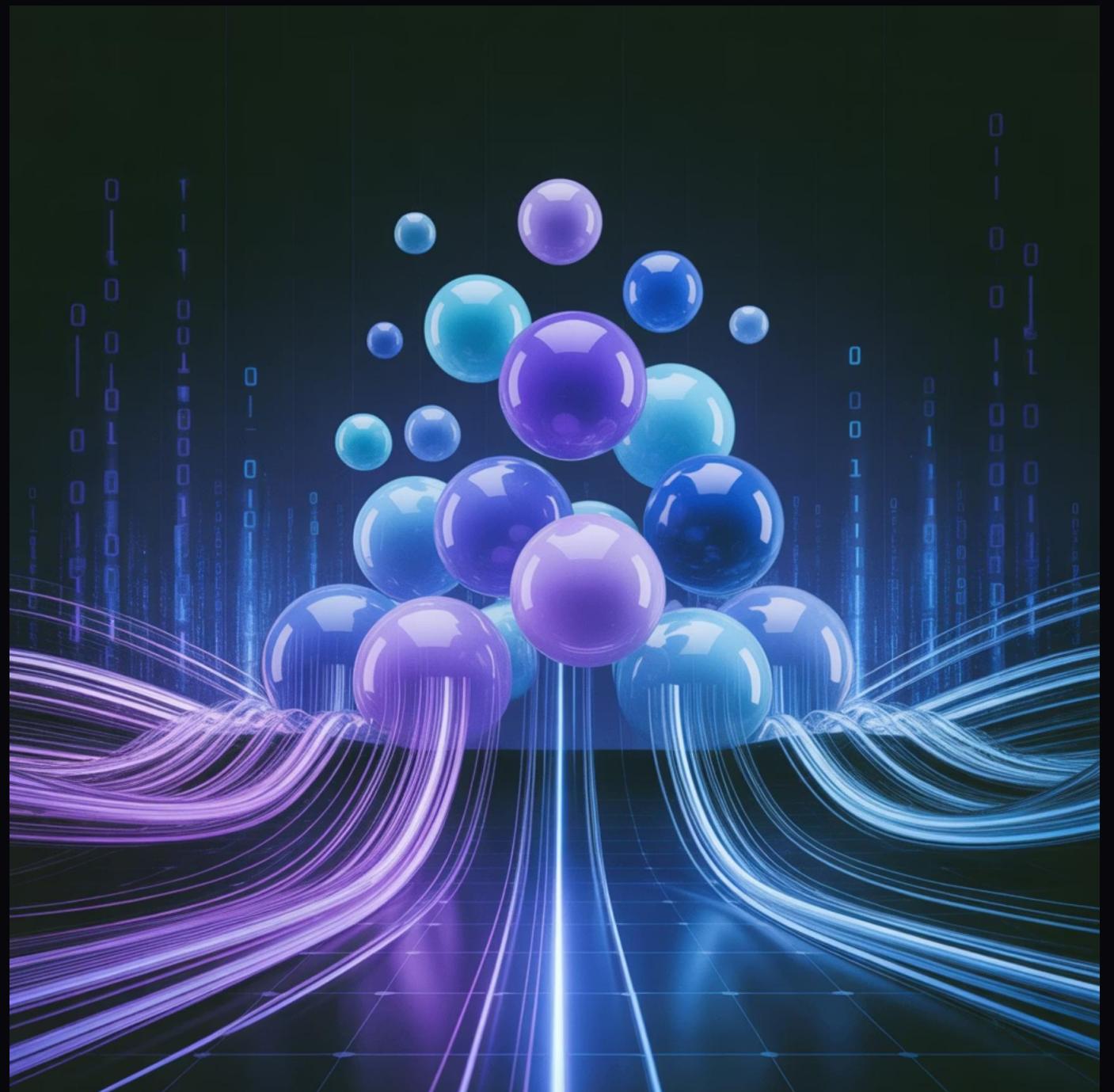
Quick Sort (médio)

Algoritmo recursivo com particionamento, muito eficiente na prática

$O(n^2)$ - Tempo Quadrático

Algoritmos Simples

- **Bubble Sort:** Compara pares adjacentes repetidamente
- **Selection Sort:** Encontra mínimo e troca posições
- **Insertion Sort:** Insere elementos na posição correta



$O(2^n)$ e $O(n!)$ - Crescimento Explosivo

2^n

Exponencial

Problemas de subconjuntos, sequência de Fibonacci
recursiva

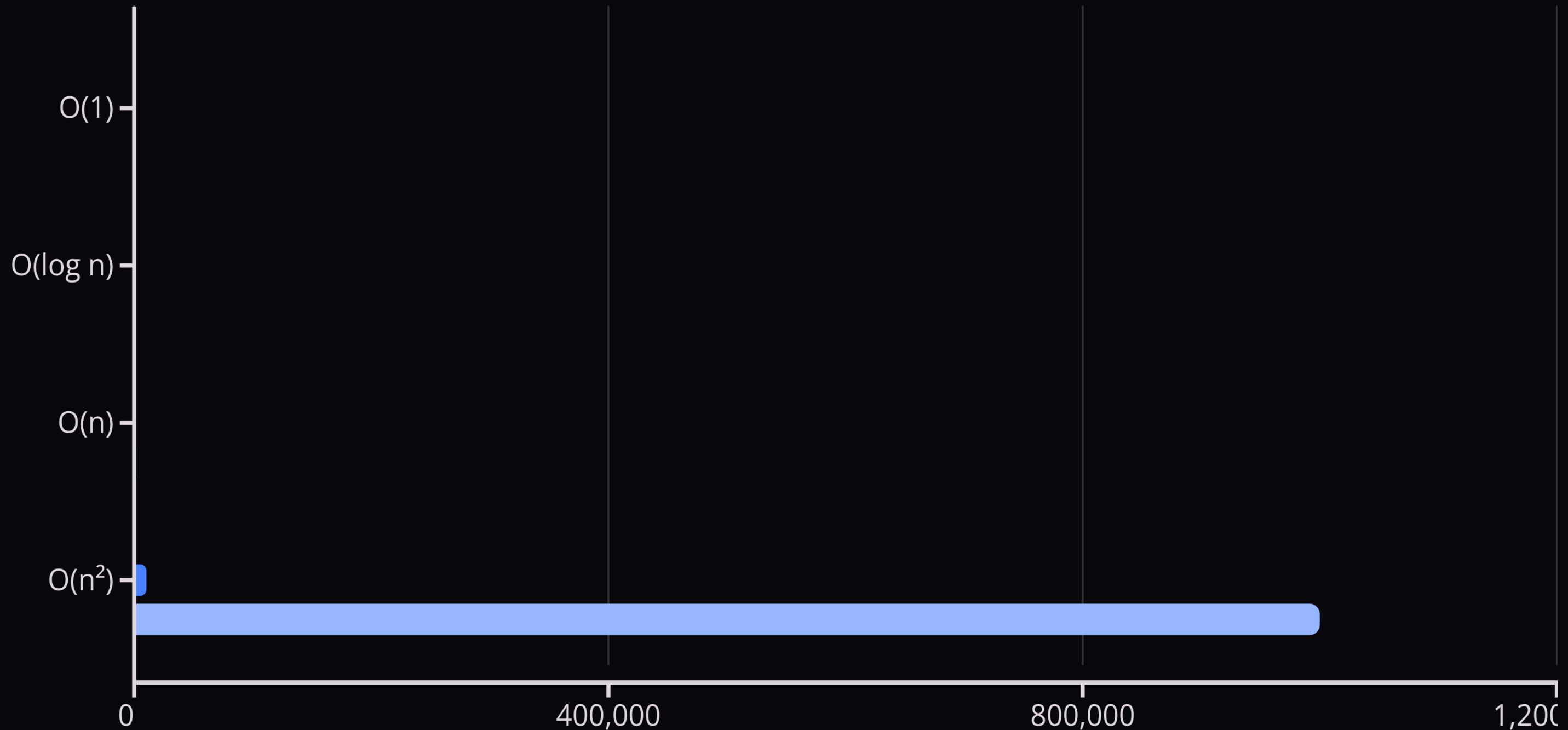
$n!$

Fatorial

Problema do caixeiro viajante, permutações completas

⚠ Impraticável para $n > 20-25$. Requer otimizações como programação dinâmica ou heurísticas

Comparação de Tempos de Execução





Capítulo 6

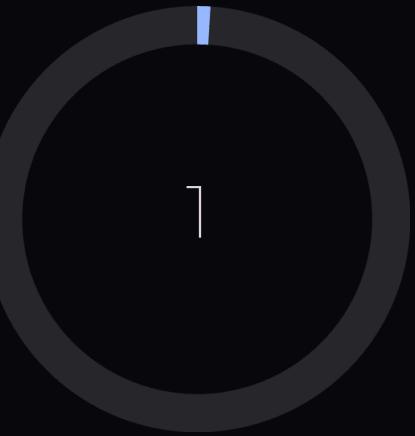
Análise de Casos Práticos

Aplicando a teoria em exemplos reais de código para solidificar o aprendizado

Exemplo 1: Soma de Elementos $O(n)$

```
def soma_elementos(lista):    total = 0          # O(1)      for elemento in lista:  # O(n)  
    total += elemento   # O(1)      return total          # O(1)
```

Análise completa: O loop percorre n elementos, executando uma operação constante para cada um.



Loop Principal

Único loop que define a complexidade



Elementos Visitados

Cada elemento é processado exatamente uma vez

✓ Complexidade Final: $O(n)$