





# APOSTILA DE ALGORITMOS E ANÁLISE DE COMPLEXIDADE

 Uma Abordagem Prática e Didática



Professor Engenheiro de Computação

**Vagner Cordeiro**

---

 Versão 2.0

 Setembro de 2025

 Material Didático

 Algoritmos & Estruturas



**Material otimizado para aprendizado progressivo com exemplos práticos e teoricamente fundamentados**



## PREFÁCIO

### Missão desta Apostila

Fornecer aos estudantes de Ciência da Computação e Engenharia uma base sólida e prática em análise de algoritmos e complexidade computacional através de uma abordagem didática e progressiva.

## Objetivos de Aprendizagem

Ao final do estudo desta apostila, você será capaz de:

 Competência	 Descrição
 <b>Analisar</b>	Avaliar complexidade temporal e espacial de algoritmos com precisão
 <b>Aplicar</b>	Utilizar notação Big-O em problemas reais e práticos
 <b>Compreender</b>	Dominar algoritmos recursivos e suas otimizações
 <b>Otimizar</b>	Implementar técnicas de programação dinâmica eficientemente
 <b>Resolver</b>	Abordar problemas algorítmicos de forma estruturada
 <b>Identificar</b>	Reconhecer padrões algorítmicos em diferentes contextos

## Metodologia de Ensino

### Estrutura Progressiva

- Conceitos básicos → Tópicos avançados
- Fundamentação teórica sólida
- Aplicações práticas reais

### Recursos por Capítulo

- Exemplos em Python e C
- Exercícios resolvidos passo a passo
- Questões para fixação e macetes

## Sobre o Professor

 **Prof. Vagner Cordeiro** - Professor Universitário especialista em Algoritmos e Estruturas de Dados

### Atuação Acadêmica:

-  Professor de Graduação e Pós-Graduação - Faculdade Estácio Florianópolis
-  Disciplinas: Análise de Algoritmos, Redes, Segurança Cibernética, Big Data, IoT, Pensamento Computacional
-  Instrutor de Informática - Governo do Estado de SC (SEJURI)

 **Formação Acadêmica:**

-  Engenharia de Computação
-  Análise e Desenvolvimento de Sistemas
-  Técnico em Telecomunicações
-  MBA Segurança da Informação
-  Licenciatura em Matemática
-  Especializações em Análise de Dados e Engenharia de Segurança

 **Experiência Profissional:** +15 anos em empresas líderes de tecnologia em SC

-  Intelbras, Embratel, Digitro e startups inovadoras



# ÍNDICE GERAL

## Guia de Navegação

Este índice foi organizado de forma progressiva para facilitar seu aprendizado. Cada capítulo constrói sobre o anterior, criando uma base sólida de conhecimento.

## SEÇÕES PRINCIPAIS

Seção	Página	Foco
 PREFÁCIO	3	Apresentação e objetivos

## CAPÍTULOS FUNDAMENTAIS

### CAPÍTULO 1 - INTRODUÇÃO À ANÁLISE DE ALGORITMOS ..... 5

-  1.1 Conceitos Fundamentais
-  1.2 Importância da Análise Algorítmica
-  1.3 Eficiência vs. Simplicidade
-  **Macete:** Como identificar gargalos algorítmicos rapidamente

### CAPÍTULO 2 - COMPLEXIDADE DE TEMPO E ESPAÇO ..... 12

-  2.1 Definições Básicas
-  2.2 Análise de Caso Médio, Melhor e Pior
-  2.3 Complexidade Espacial
-  **Macete:** Truque da contagem de operações fundamentais

### CAPÍTULO 3 - NOTAÇÃO BIG-O ..... 18

-  3.1 Definição Formal
-  3.2 Propriedades da Notação Big-O
-  3.3 Exemplos Práticos
-  3.4 Outras Notações ( $\Omega$ ,  $\Theta$ )
-  **Macete:** Regra dos "Três Cs" para Big-O

### CAPÍTULO 4 - RECURSIVIDADE ..... 25

-  4.1 Conceitos Fundamentais
-  4.2 Casos Base e Recursivos
-  4.3 Tipos de Recursão
-  4.4 Análise de Complexidade Recursiva

- ⚡ 4.5 Técnicas de Otimização
- ⚡ **Macete:** RBCO - Regra Base, Chamada, Otimização

## 🚀 CAPÍTULOS ALGORITMOS ESSENCIAIS

### ⌚ CAPÍTULO 5 - ALGORITMOS DE ORDENAÇÃO ..... 45

- 📈 5.1 Algoritmos Básicos ( $O(n^2)$ )
- 🚀 5.2 Algoritmos Eficientes ( $O(n \log n)$ )
- ⚖️ 5.3 Análise Comparativa
- ⚡ **Macete:** "BIMM" - Bubble, Insertion, Merge, Quick

### 🌳 CAPÍTULO 6 - ÁRVORES ..... 55

- 🌳 6.1 Conceitos Fundamentais
- 🔎 6.2 Árvores Binárias de Busca
- ⚖️ 6.3 Árvores Balanceadas (AVL)
- ⚡ **Macete:** "EED" - Esquerda, Equal, Direita para BST

## 💻 CAPÍTULOS AVANÇADOS

### CAPÍTULO 7 - GRAFOS ..... 70

- 🌐 7.1 Representações de Grafos
- 🔎 7.2 Algoritmos de Busca (DFS, BFS)
- 🛌 7.3 Caminhos Mínimos
- ⚡ **Macete:** "DLAB" - Dijkstra, Lista, Adjacência, Busca

### ⚡ CAPÍTULO 8 - PROGRAMAÇÃO DINÂMICA ..... 85

- 💡 8.1 Conceitos e Princípios
- 🌸 8.2 Problemas Clássicos
- 📊 8.3 Técnicas de Otimização
- ⚡ **Macete:** "MEMO" - Memoização, Estado, Matriz, Otimização

## 📚 MATERIAL COMPLEMENTAR

### 💡 APÊNDICES & EXERCÍCIOS

- 📄 **APÊNDICE A** - Exercícios Resolvidos ..... 95
- 📄 **APÊNDICE B** - Lista de Exercícios ..... 105
- 📄 **APÊNDICE C** - Glossário de Termos ..... 115

-  **APÊNDICE D** - Referências Bibliográficas ..... 120

 **Total:** ~120 páginas de conteúdo otimizado para seu aprendizado!

- 5.4 Quando Usar Cada Algoritmo

**CAPÍTULO 6 - ALGORITMOS DE BUSCA** ..... 58

- 6.1 Busca Linear
- 6.2 Busca Binária
- 6.3 Busca em Estruturas Complexas

**CAPÍTULO 7 - ANÁLISE AMORTIZADA** ..... 65

- 7.1 Conceitos e Aplicações
- 7.2 Método do Agregado
- 7.3 Método do Contador
- 7.4 Método do Potencial

**CAPÍTULO 8 - INVARIANTES DE LOOP** ..... 72

- 8.1 Definição e Importância
- 8.2 Demonstração de Corretude
- 8.3 Exemplos Práticos

**CAPÍTULO 9 - ESTRATÉGIAS DE RESOLUÇÃO DE PROBLEMAS** ..... 78

- 9.1 Metodologia RICE
- 9.2 Padrões Algorítmicos Comuns
- 9.3 Técnicas de Otimização

**APÊNDICES** ..... 85

- A. Tabela de Complexidades
- B. Glossário de Termos
- C. Bibliografia e Referências
- D. Exercícios Adicionais





# CAPÍTULO 1

## INTRODUÇÃO À ANÁLISE DE ALGORITMOS

⌚ **Objetivo:** Compreender os fundamentos da análise algorítmica e sua importância prática

### ✳️ 1.1 Conceitos Fundamentais

#### 💡 O que é um Algoritmo?

**Definição:** Um algoritmo é uma **receita computacional** - uma sequência finita de instruções bem definidas e não ambíguas para resolver um problema específico.

💡 **Analogia Prática:** Pense em uma receita de bolo! Tem ingredientes (entrada), passos ordenados (instruções) e o bolo pronto (saída).

#### ⭐ Características de um Algoritmo Eficaz

##### ⌚ Finitude

Deve terminar em tempo finito

*Sem loops infinitos!*

##### ⌚ Definição Clara

Cada passo é preciso e inequívoco

*Zero ambiguidade!*

##### 🖨 Entrada

Zero ou mais entradas bem definidas

*Dados de entrada claros!*

##### 📤 Saída

Uma ou mais saídas específicas

*Resultado esperado!*

##### ⚡ Efetividade

Cada operação deve ser básica o suficiente para ser executada por uma máquina

*Computacionalmente possível!*

### 🎯 MACETE FUNDAMENTAL - "FDSEE"

**Para lembrar das características essenciais de um algoritmo:**

- 👉 **Finitude** - Tem que parar! 🎯 **Definição** - Sem ambiguidade!
- 👉 **Saída** - Produz resultado! 🎯 **Entrada** - Recebe dados! ⚡ **Efetividade** - É possível executar!
- 💡 **Dica de Memorização:** "Finito, Definido, com Saída, Entrada e Efetivo = Algoritmo Perfeito!"

## 📊 1.2 Análise de Algoritmos - Por que é Fundamental?

### 🔍 O que é Análise de Algoritmos?

É o processo de determinar a **quantidade de recursos computacionais** (tempo e espaço) que um algoritmo consome em função do tamanho da entrada.

### 💰 Por que Analisar? - Os "4 Rs" da Análise

#### 💎 Redução de Custos

Menos tempo = menos dinheiro gasto em processamento

#### ⚡ Rapidez

Usuários felizes com respostas instantâneas

#### 🏆 Ranking

Destaque profissional com soluções otimizadas

#### ♻️ Reutilização

Códigos eficientes são mais reutilizáveis

### 🎯 MACETE DOS RECURSOS - "TE"

**Os dois recursos fundamentais que sempre analisamos:**

- ⌚ **Tempo** - Quantas operações? 🏃 **Espaço** - Quanta memória?
- 💡 **Lembra assim:** "Tempo Espaço = TEcnologia Eficiente!"

## 1.3 Eficiência vs. Simplicidade - O Dilema do Desenvolvedor

### O Equilíbrio Perfeito

Nem sempre o algoritmo mais eficiente é a melhor escolha. Às vezes, um código simples e legível vale mais que uma otimização complexa.

### Matriz de Decisão - Quando Otimizar?

Situação	 Volume de Dados	 Criticidade Tempo	 Ação Recomendada
 Simples	Pequeno	Baixa	Priorize legibilidade
 Moderado	Médio	Média	Equilibre simplicidade e eficiência
 Crítico	Grande	Alta	Otimize ao máximo

### MACETE DA ESCOLHA - "PEV"

Para decidir entre eficiência e simplicidade:

 Performance necessária?  Equipe consegue manter?  Volume de dados é crítico?

Se 2 ou mais respostas forem "SIM" → Optimize!

Se não → Mantenha simples!

### Exemplo Prático - Busca em Lista

```
# 📦 SIMPLES: Para listas pequenas (< 100 itens)
def busca_simples(lista, item):
    return item in lista # Legível e direto

# 🔥 OTIMIZADA: Para listas grandes (> 1000 itens)
def busca_binaria(lista_ordenada, item):
    inicio, fim = 0, len(lista_ordenada) - 1
    while inicio <= fim:
        meio = (inicio + fim) // 2
        if lista_ordenada[meio] == item:
            return True
        elif lista_ordenada[meio] < item:
            inicio = meio + 1
        else:
```

```
fim = meio - 1  
return False
```

⌚ **Regra de Ouro:** "Premature optimization is the root of all evil" - Donald Knuth

**Tradução prática:** Primeiro faça funcionar, depois otimize se necessário!

---





## CAPÍTULO 2

### COMPLEXIDADE DE TEMPO E ESPAÇO

🎯 **Objetivo:** Dominar a análise de recursos computacionais e otimização de algoritmos

#### ⌚ 2.1 Complexidade de Tempo - A Arte de Contar Operações

##### 💡 O que é Complexidade de Tempo?

**Definição:** Mede o **número de operações fundamentais** que um algoritmo executa em função do tamanho da entrada ( $n$ ).

💡 **Analogia:** É como contar quantos passos você dá para chegar ao trabalho - pode variar com o trânsito (entrada)!

##### 🎯 MACETE PARA CONTAR OPERAÇÕES - "LOOP-IF-CALL"

Como identificar operações que "custam" tempo:

⌚ **LOOP** - Quantas vezes repete? ✎ **IF** - Comparações e decisões 📙 **CALL** - Chamadas de função

💡 **Regra de Ouro:** "Se está dentro de um loop, multiplique. Se é um IF, some. Se é uma chamada, analise recursivamente!"

##### 💻 Exemplo Prático - Análise de Busca Linear

```
# 🔎 Busca Linear Comentada
def busca_linear(lista, item):
    for i in range(len(lista)): # ⌚ LOOP: n iterações
```

```

if lista[i] == item:      # ✎ IF: 1 comparação por iteração
    return i              # ⚡ RETURN: operação constante
return -1                # ⚡ RETURN: operação constante

# 📈 Análise: n * 1 = O(n) operações no pior caso

```

#### 🎯 Contagem Prática:

- Loop de  $n$  elementos =  $n$  operações
- Comparação dentro do loop = 1 operação por iteração
- **Total máximo:  $n$  comparações =  $O(n)$**

## 💻 2.2 Complexidade de Espaço - Gerenciando Memória

#### 📦 O que é Complexidade de Espaço?

Mede a **quantidade de memória adicional** que um algoritmo usa, além da entrada original.

#### 🎯 MACETE PARA ANÁLISE DE ESPAÇO - "VAR-REC-EST"

##### O que conta para espaço:

- ✍ VARáveis locais extras
- ✍ RECursão (pilha de chamadas)
- 📊 ESTruturas auxiliares (arrays, listas)

💡 **Dica:** "Variáveis extras, Recursão e Estruturas = Espaço!"

## 💻 Exemplo Prático - Análise de Espaço

```

# 💻 Exemplo: Soma Recursiva
def soma_recursiva(n):
    if n <= 1:          # 📈 Variável: comparação (espaço constante)
        return n
    return n + soma_recursiva(n - 1) # 📈 Recursão: n chamadas na pilha

# 📈 Espaço: O(n) - cada chamada ocupa espaço na pilha

# 💻 Alternativa Iterativa
def soma_iterativa(n):
    resultado = 0          # 📈 Uma variável extra
    for i in range(1, n + 1): # 📈 Variável i (reutilizada)

```

```
    resultado += i
return resultado

# 📈 Espaço: O(1) - apenas variáveis locais constantes
```

## 📊 2.3 Os Três Casos de Análise - "MCA"

### 🎯 MACETE DOS CASOS - "MCA"

Melhor caso - quando tudo dá certo Caso médio - realidade do dia a dia Alternativa pior - quando tudo dá errado

- 🟢 Melhor Caso
  - 🎯 Cenário: Item na 1<sup>a</sup> posição
  - ⌚ Tempo: O(1)
  - 💡 Exemplo: Buscar 10 em [10,20,30]

- 🟡 Caso Médio
  - 🎯 Cenário: Item no meio
  - ⌚ Tempo:  $O(n/2) = O(n)$
  - 💡 Exemplo: Buscar 20 em [10,20,30]

- 🔴 Pior Caso
  - 🎯 Cenário: Item inexistente
  - ⌚ Tempo: O(n)
  - 💡 Exemplo: Buscar 99 em [10,20,30]

### 🎯 MACETE PARA ANÁLISE RÁPIDA - "PMC"

Ordem de importância na análise:

- 1 Pior caso - O que garante?
- 2 Médio caso - O que espera?
- 3 Caso melhor - Bônus raro

💡 Regra Prática: "Prepare-se para o Pior, Esperando o Médio, Comemorando o Melhor!"

## 📋 2.4 Tabela de Operações Fundamentais

### 🎯 MACETE DA TABELA - "ABIR"

Para lembrar as operações básicas:

- 🎯 Acesso - Ir direto ao elemento
- 🔍 Busca - Procurar um elemento
- ➕ Inserção - Adicionar elemento
- ✖️ Remoção - Excluir elemento

Estrutura	Acesso	Busca	Inserção	Remoção
Array	O(1) ⚡	O(n) 🏃	O(n) 🏃	O(n) 🏃
Lista Ligada	O(n) 🏃	O(n) 🏃	O(1) ⚡	O(1) ⚡
Hash Table	O(1) ⚡	O(1) ⚡	O(1) ⚡	O(1) ⚡

💡 **Legenda:** ⚡ = Rápido (constante/logarítmico) | 🏃 = Lento (linear/quadrático)

⌚ **Dica de Escolha:** Array para acesso, Lista para modificação, Hash para tudo rápido!

| Pilha | O(1) | O(n) | O(1) | O(1) || Fila | O(1) | O(n) | O(1) | O(1) |

### Macete: Contagem de Operações

```
# Como contar operações:
def exemplo(n):
    count = 0                      # 1 operação
    for i in range(n):              # n iterações
        count += 1                  # 1 operação por iteração
    return count                     # 1 operação

# Total: 1 + n + 1 = n + 2 operações = O(n)
```

}

### Macete: Hash table = O(1) para acesso por chave

print(dados["nome"]) # O(1) acesso direto

```
### **Ponteiros e Referências**

```python
# Python usa referências automaticamente
lista_a = [1, 2, 3]
lista_b = lista_a          # lista_b aponta para lista_a
lista_b.append(4)          # Modifica lista_a também!

# Macete: Para copiar, use copy()
import copy
lista_c = copy.copy(lista_a) # Cópia rasa
lista_d = copy.deepcopy(lista_a) # Cópia profunda
```

### Macetes de Estruturas de Dados

ACESSO POR ÍNDICE:

```
Array/Lista → O(1)    # Posição = base + índice × tamanho
Lista Ligada → O(n)  # Precisa percorrer desde o início
```

**BUSCA:**

Array Ordenado →  $O(\log n)$  # Busca binária  
Hash Table →  $O(1)^*$  # Média,  $O(n)$  pior caso  
Lista Ligada →  $O(n)$  # Sempre linear

**INSERÇÃO:**

Array (final) →  $O(1)$  # Amortizada  
Array (meio) →  $O(n)$  # Precisa deslocar elementos  
Lista Ligada →  $O(1)$  # Se tiver a posição  
Hash Table →  $O(1)^*$  # Média

---

# CAPÍTULO 3

## NOTAÇÃO BIG-O



## CAPÍTULO 3

### NOTAÇÃO BIG-O - A LINGUAGEM DA EFICIÊNCIA

🎯 **Objetivo:** Dominar a análise assintótica e escolher os melhores algoritmos

#### 📊 3.1 Big-O Descomplicado - A Arte de Prever o Futuro

##### 💡 O que é Big-O?

**Definição Simples:** Big-O responde à pergunta: "E se eu tivesse MUITO mais dados?"

É como um GPS que te mostra se o caminho fica mais longo conforme você tem mais paradas para fazer.

##### 🎯 MACETE VISUAL - "A ESCADA DO TERROR BIG-O"

###### 📈 VELOCÍMETRO DE COMPLEXIDADE ( $n = 1000$ )

🚀 $O(1)$	= 1	⚡ FLASH!
🛸 $O(\log n)$	= 10	⚡ SUPER RÁPIDO
🚗 $O(n)$	= 1,000	✅ BOM
🏎 $O(n \log n)$	= 10,000	⚠ ACEITÁVEL
🏡 $O(n^2)$	= 1,000,000	🔥 CUIDADO!
👽 $O(2^n)$	= $10^{301}$	✖ IMPOSSÍVEL
💀 $O(n!)$	= $\infty$	🚫 NEM TENTE

 **Regra de Ouro:** "Se passou de  $O(n^2)$ , é hora de repensar sua vida!"

## **3.2 Top 8 das Complexidades - O Ranking da Velocidade**

### **$O(1)$ - CONSTANTE**

 **Exemplo:** Pegar livro marcado

 **Uso:** Acesso direto, Hash

### **$O(\log n)$ - LOGARÍTMICA**

 **Exemplo:** Busca no dicionário

 **Uso:** Busca binária, Árvores

### **$O(n)$ - LINEAR**

 **Exemplo:** Ler lista do início ao fim

 **Uso:** Busca simples, Iteração

### **$O(n \log n)$ - LINEARÍTMICA**

 **Exemplo:** Ordenar cartas eficientemente

 **Uso:** Merge Sort, Quick Sort

### **$O(n^2)$ - QUADRÁTICA**

 **Exemplo:** Comparar todos com todos

 **Uso:** Bubble Sort, força bruta

### **$O(n^3)$ - CÚBICA**

 **Exemplo:** 3 loops aninhados

 **Uso:** ⚠️ Evitar se possível

### **$O(2^n)$ - EXPONENCIAL**

 **Exemplo:** Todas as combinações

 **Uso:** ❌ Só casos muito pequenos

### **$O(n!)$ - FATORIAL**

 **Exemplo:** Todas as permutações

 **Uso:** 💀 Praticamente impossível

## **3.3 Como Calcular Big-O - O Método "LISP"**

### **MACETE "LISP" para Análise**

Loops - Conte os loops aninhados Ignore - Constantes e termos menores

Some - Operações em sequência Pior - Foque no pior caso

### **Passo 1: LOOPS - Contar os Laços**

```
# 📊 UM LOOP =  $O(n)$ 
for i in range(n):          # n iterações
    print(i)                #  $O(1)$  por iteração
# Total:  $n \times O(1) = O(n)$ 

# 📊 📊 DOIS LOOPS ANINHADOS =  $O(n^2)$ 
```

```

for i in range(n):           # n iterações
    for j in range(n):       # n iterações para cada i
        print(i, j)          # O(1) por iteração
# Total: n × n × O(1) = O(n2)

# ❌❌❌ TRÊS LOOPS = O(n3) - CUIDADO!
for i in range(n):
    for j in range(n):
        for k in range(n):
            print(i, j, k)
# Total: O(n3) - Evite isso!

```

## ⓧ Passo 2: IGNORE - Eliminar o Desnecessário

### Regras de Simplificação:

- 🚫 **Corte constantes:**  $O(3n) = O(n)$
- 🚫 **Mantenha o maior:**  $O(n^2 + n) = O(n^2)$
- 🚫 **Ignore bases:**  $O(\log_2 n) = O(\log n)$

# ❌ Análise Incorreta vs ✅ Análise Correta

```

# Código de exemplo
for i in range(n):           # O(n)
    print(i * 2)              # O(1)

for j in range(100):          # O(100) = O(1)
    print("hello")

for k in range(n * n):        # O(n2)
    print(k)

# ❌ Incorreto: O(n) + O(1) + O(n2) = O(n + 1 + n2)
# ✅ Correto: O(n2) - apenas o termo dominante!

```

## ⊕ Passo 3: SOME - Operações em Sequência

### Quando somar vs quando multiplicar:

- ⊕ **SOME:** Operações uma após a outra
- ✗ **MULTIPLIQUE:** Operações aninhadas

```

# ⊕ SOMAR - Operações em sequência
def algoritmo_sequencial(n):
    # Primeira parte: O(n)
    for i in range(n):
        print(i)

```

```

# Segunda parte: O(n2)
for i in range(n):
    for j in range(n):
        print(i, j)

# Total: O(n) + O(n2) = O(n2)

# ✖ MULTIPLICAR - Operações aninhadas
def algoritmo_aninhado(n):
    for i in range(n):           # n vezes
        for j in range(n):       # n vezes para cada i
            for k in range(n):   # n vezes para cada j
                print(i, j, k)

# Total: n × n × n = O(n3)

```

## 🎯 3.4 MACETES PRÁTICOS - "CHEAT CODES" para Big-O

### 🔥 MACETE #1: "A Regra do Olhômetro"

**Contagem rápida visual:**

- ⌚ **Nenhum loop:** O(1)
- ⌚ **1 loop:** O(n)
- ⌚ **2 loops aninhados:** O(n<sup>2</sup>)
- ⌚ **3+ loops aninhados:** O(n<sup>3+</sup>) - Fuga! ⚡ **Dividir pela metade:** O(log n)

### 🔥 MACETE #2: "Padrões que Você Vai Ver Toda Hora"

⌚ Padrão	📊 Complexidade	💻 Exemplo Típico
for i in range(n)	O(n)	Busca linear
while low <= high	O(log n)	Busca binária
for i in range(n): for j in range(n)	O(n <sup>2</sup> )	Bubble sort
dividir_e_conquistar()	O(n log n)	Merge sort
todos_subconjuntos()	O(2 <sup>n</sup> )	Problema da mochila

### 🔥 MACETE #3: "O Teste da Escala"

**Pergunta mágica:** "Se eu multiplicar a entrada por 10, o que acontece com o tempo?"

✓ **Mesmo tempo:**  $O(1)$  ✗ **10x mais tempo:**  $O(n)$   
✗ **100x mais tempo:**  $O(n^2)$  ✗ **Explode exponencialmente:**  $O(2^n)$  ou pior

## 🔥 MACETE #4: "Receitas Algorítmicas Clássicas"

```

# 📊 RECEITA  $O(1)$  - Acesso Direto
def pegar_primeiro(lista):
    return lista[0] # Sempre mesmo tempo

# 📊 RECEITA  $O(\log n)$  - Dividir e Conquistar
def busca_binaria(lista, item):
    # Sempre divide pela metade
    meio = len(lista) // 2
    # ... restante do algoritmo

# 📊 RECEITA  $O(n)$  - Uma Passada
def maximo(lista):
    maior = lista[0]
    for item in lista: # Passa uma vez por cada
        if item > maior:
            maior = item
    return maior

# 📊 RECEITA  $O(n^2)$  - Todos com Todos
def tem_duplicatas(lista):
    for i in range(len(lista)):
        for j in range(i+1, len(lista)): # Compara cada um com os outros
            if lista[i] == lista[j]:
                return True
    return False
  
```

### 2. Termo dominante vence:

- $O(n^2 + n) = O(n^2)$
- $O(n + \log n) = O(n)$
- $O(n^3 + n^2 + n + 1) = O(n^3)$

### 3. Sempre considere o pior caso:

- Mesmo que às vezes seja rápido, Big-O mede o pior cenário

## Exemplos Práticos com Explicação

### Exemplo 1: Busca Linear

```

def encontrar_numero(lista, numero):
    for i in range(len(lista)): # No pior caso, percorre toda a lista
        if lista[i] == numero: #  $O(1)$  para cada comparação
            return i
    return -1
  
```

```
# Análise: No pior caso, o número está no final ou não existe
# Precisa verificar todos os n elementos
# Complexidade: O(n)
```

### Exemplo 2: Busca em Pares

```
def encontrar_par(lista):
    for i in range(len(lista)):           # n iterações
        for j in range(i+1, len(lista)): # n-1, n-2, ..., 1 iterações
            if lista[i] + lista[j] == 10:
                return (i, j)
    return None

# Análise: Dois loops aninhados
# Total de comparações: (n-1) + (n-2) + ... + 1 = n(n-1)/2
# Complexidade: O(n²)
```

### Como Identificar Complexidade Rapidamente

```
# Padrões comuns:

# 1. Um loop simples = O(n)
for item in lista:
    fazer_algo()

# 2. Loop dividindo pela metade = O(log n)
while n > 1:
    n = n // 2

# 3. Dois loops aninhados = O(n²)
for i in range(n):
    for j in range(n):
        fazer_algo()

# 4. Loop dentro de função recursiva = O(n²) ou mais
def recursiva(n):
    if n <= 1: return
    for i in range(n): # O(n)
        fazer_algo()
    recursiva(n-1)      # Chama n vezes

# 5. Dividir e conquistar = O(n log n)
def merge_sort(lista):
    # Divide: O(log n) níveis
    # Conquista: O(n) em cada nível
    # Total: O(n log n)
```

### Dicas para Melhorar Complexidade

## Do Ruim para o Bom:

```
# RUIM: O(n2) - Busca em lista
def buscar_duplicata_ruim(lista):
    for i in range(len(lista)):
        for j in range(i+1, len(lista)):
            if lista[i] == lista[j]:
                return True
    return False

# BOM: O(n) - Usando conjunto
def buscar_duplicata_bom(lista):
    visto = set()
    for item in lista:
        if item in visto:
            return True
        visto.add(item)
    return False
```

## Gráfico Mental de Crescimento

Para entender visualmente como cada complexidade cresce:

n=1	n=10	n=100	n=1000
O(1):			
O(log n):			
O(n):			... (cresce linear)
O(n <sup>2</sup> ):			... (cresce rápido)
O(2 <sup>n</sup> ):	XXX	XXXXXXX	(explode)

## Estruturas de Dados Fundamentais

### Array/Vetor

- **Acesso:** O(1)
- **Busca:** O(n)
- **Inserção:** O(n) - no meio, O(1) - no final
- **Remoção:** O(n) - no meio, O(1) - no final

### Lista Ligada

- **Acesso:** O(n)
- **Busca:** O(n)
- **Inserção:** O(1) - conhecendo a posição
- **Remoção:** O(1) - conhecendo a posição

### Pilha (Stack)

- **Push:** O(1)
- **Pop:** O(1)
- **Top:** O(1)

## Fila (Queue)

- **Enqueue**: O(1)
- **Dequeue**: O(1)
- **Front**: O(1)

## 4.5 Exercícios de Fixação - Capítulo 4

### Exercício 4.1: Implementação Básica

Implemente uma função recursiva que calcule a soma dos dígitos de um número:

```
def soma_digitos(n):
    # Caso base: se n < 10, retorna n
    # Caso recursivo: último dígito + soma_digitos(n // 10)
    pass
```

Solução:

```
def soma_digitos(n):
    if n < 10:
        return n
    return n % 10 + soma_digitos(n // 10)
```

### Exercício 4.2: Análise de Complexidade

Qual a complexidade das seguintes funções recursivas?

```
# Função A
def funcao_a(n):
    if n <= 1:
        return 1
    return funcao_a(n - 1)

# Função B
def funcao_b(n):
    if n <= 1:
        return 1
    return funcao_b(n // 2)

# Função C
def funcao_c(n):
    if n <= 1:
        return 1
    return funcao_c(n - 1) + funcao_c(n - 1)
```

**Respostas:** A = O(n), B = O(log n), C = O( $2^n$ )

### Exercício 4.3: Problema Prático

Implemente o algoritmo das "Torres de Hanói" recursivamente e calcule quantos movimentos são necessários para n=4 discos.

**Resposta:**  $2^4 - 1 = 15$  movimentos

#### **Exercício 4.4: Otimização**

Converta a seguinte função recursiva para iterativa:

```
def potencia_rec(base, exp):
    if exp == 0:
        return 1
    return base * potencia_rec(base, exp - 1)
```

**Solução Iterativa:**

```
def potencia_iter(base, exp):
    resultado = 1
    for i in range(exp):
        resultado *= base
    return resultado
```

# CAPÍTULO 5

## ALGORITMOS DE ORDENAÇÃO

### 5.1 Visão Geral dos Algoritmos

Tabela Comparativa Essencial

Algoritmo	Complexidade	Quando Usar
<b>Bubble Sort</b>	$O(n^2)$	Nunca (só para ensinar)
<b>Selection Sort</b>	$O(n^2)$	Datasets muito pequenos
<b>Insertion Sort</b>	$O(n^2)$	Arrays quase ordenados
<b>Merge Sort</b>	$O(n \log n)$	Quando precisa de estabilidade
<b>Quick Sort</b>	$O(n \log n)$	Uso geral, performance
<b>Heap Sort</b>	$O(n \log n)$	Quando espaço é limitado

### 5.2 Algoritmos Básicos ( $O(n^2)$ )

#### Bubble Sort - "Ordenação da Bolha"

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

# Macete: "Bolhas sobem" - maior elemento "flutua" para o fim
```

#### Selection Sort - "Ordenação por Seleção"

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

# Macete: "Seleciona o menor" e coloca na posição correta
```

## Insertion Sort - "Ordenação por Inserção"

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

# Macete: Como organizar cartas na mão - insere cada carta no lugar certo
```

## 5.3 Algoritmos Eficientes ( $O(n \log n)$ )

### Merge Sort - "Dividir para Conquistar"

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left = arr[:mid]
        right = arr[mid:]

        merge_sort(left)    # Ordena metade esquerda
        merge_sort(right)   # Ordena metade direita

        # Intercala as duas metades
        i = j = k = 0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1

        # Copia elementos restantes
        while i < len(left):
            arr[k] = left[i]
            i += 1
            k += 1
        while j < len(right):
            arr[k] = right[j]
            j += 1
            k += 1

# Macete: Sempre  $O(n \log n)$  - divide até ficar trivial, depois intercala
```

## Quick Sort - "Pivô e Partição"

```
def quick_sort(arr, low=0, high=None):
    if high is None:
        high = len(arr) - 1

    if low < high:
        # Particiona e encontra posição do pivô
        pi = partition(arr, low, high)

        # Ordena elementos antes e depois do pivô
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

def partition(arr, low, high):
    pivot = arr[high] # Escolhe último elemento como pivô
    i = low - 1

    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

# Macete: Escolhe pivô, separa menores/maiores, repete recursivamente
```

## 5.4 Quando Usar Cada Algoritmo

### Escolha Prática

```
# Para arrays pequenos (n < 50)
def ordenar_pequeno(arr):
    return insertion_sort(arr) # Simples e eficiente

# Para arrays médios/grandes (n > 50)
def ordenar_grande(arr):
    return quick_sort(arr) # Rápido na prática

# Quando precisa de garantias (sempre O(n log n))
def ordenar_garantido(arr):
    return merge_sort(arr) # Nunca degrada para O(n^2)

# Quando memória é limitada
def ordenar_economico(arr):
    return heap_sort(arr) # O(1) de espaço extra
```

## Macetes de Complexidade

```
# Como lembrar das complexidades:

# O(n2) - Dois loops aninhados
# Bubble, Selection, Insertion = todos O(n2)

# O(n log n) - Divide e conquista
# Merge, Quick, Heap = todos O(n log n)

# Exceções importantes:
# - Insertion Sort: O(n) para arrays quase ordenados
# - Quick Sort: O(n2) no pior caso (pivô sempre o menor/maior)
```

```
    arr[j], arr[j + 1] = arr[j + 1], arr[j]
    trocou = True

    # Se não houve troca, array já está ordenado
    if not trocou:
        break

return arr
```

## Teste

```
lista = [64, 34, 25, 12, 22, 11, 90] print("Lista original:", lista) print("Lista ordenada:", bubble_sort(lista.copy()))
```

```
**Implementação C:**  
```c  
#include <stdio.h>  
#include <stdbool.h>

void bubble_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        bool trocou = false;

        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Troca elementos
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                trocou = true;
            }
        }
    }

    // Otimização: se não houve troca, array está ordenado
    if (!trocou) {
```

```

        break;
    }
}

void imprimir_array(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Array original: ");
    imprimir_array(arr, n);

    bubble_sort(arr, n);

    printf("Array ordenado: ");
    imprimir_array(arr, n);

    return 0;
}

```

## Selection Sort

**Conceito:** Encontra o menor elemento e o coloca na primeira posição, depois encontra o segundo menor, e assim por diante.

### Implementação Python:

```

def selection_sort(arr):
    n = len(arr)

    for i in range(n):
        # Encontra o índice do menor elemento na parte não ordenada
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j

        # Troca o menor elemento encontrado com o primeiro elemento
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

    return arr

```

### Implementação C:

```

#include <stdio.h>

void selection_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;

        // Encontra o menor elemento na parte não ordenada
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }

        // Troca o menor elemento com o primeiro
        if (min_idx != i) {
            int temp = arr[i];
            arr[i] = arr[min_idx];
            arr[min_idx] = temp;
        }
    }
}

```

## Insertion Sort

**Conceito:** Constrói a lista ordenada um elemento por vez, inserindo cada novo elemento na posição correta.

### Implementação Python:

```

def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        # Move elementos maiores que key uma posição à frente
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        # Insere key na posição correta
        arr[j + 1] = key

    return arr

```

### Implementação C:

```

#include <stdio.h>

void insertion_sort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];

```

```

int j = i - 1;

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

// Insere key na posição correta
arr[j + 1] = key;
}
}

```

## Merge Sort

**Conceito:** Divide o array em duas metades, ordena cada metade recursivamente e depois mescla as duas metades ordenadas.

### Implementação Python:

```

def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    # Divide o array em duas metades
    meio = len(arr) // 2
    esquerda = merge_sort(arr[:meio])
    direita = merge_sort(arr[meio:])

    # Mescla as duas metades ordenadas
    return merge(esquerda, direita)

def merge(esquerda, direita):
    resultado = []
    i = j = 0

    # Mescla elementos enquanto ambas as listas têm elementos
    while i < len(esquerda) and j < len(direita):
        if esquerda[i] <= direita[j]:
            resultado.append(esquerda[i])
            i += 1
        else:
            resultado.append(direita[j])
            j += 1

    # Adiciona elementos restantes
    resultado.extend(esquerda[i:])
    resultado.extend(direita[j:])

    return resultado

```

### Implementação C:

```
#include <stdio.h>
#include <stdlib.h>

void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    // Arrays temporários
    int *L = (int*)malloc(n1 * sizeof(int));
    int *R = (int*)malloc(n2 * sizeof(int));

    // Copia dados para arrays temporários
    for (int i = 0; i < n1; i++) {
        L[i] = arr[l + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = arr[m + 1 + j];
    }

    // Mescla os arrays temporários de volta em arr[l..r]
    int i = 0, j = 0, k = l;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copia elementos restantes de L[], se houver
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copia elementos restantes de R[], se houver
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }

    free(L);
    free(R);
}
```

```

}

void merge_sort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - 1) / 2;

        // Ordena primeira e segunda metades
        merge_sort(arr, l, m);
        merge_sort(arr, m + 1, r);

        // Mescla as metades ordenadas
        merge(arr, l, m, r);
    }
}

```

## Quick Sort

**Conceito:** Escolhe um elemento como pivô e partitiona o array de forma que elementos menores fiquem à esquerda e maiores à direita do pivô.

### Implementação Python:

```

def quick_sort(arr, low=0, high=None):
    if high is None:
        high = len(arr) - 1

    if low < high:
        # pi é o índice de partição
        pi = partition(arr, low, high)

        # Ordena elementos antes e depois da partição
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

    return arr

def partition(arr, low, high):
    # Pivô é o último elemento
    pivot = arr[high]

    # Índice do menor elemento (indica a posição correta do pivô)
    i = low - 1

    for j in range(low, high):
        # Se elemento atual é menor ou igual ao pivô
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    # Coloca pivô na posição correta

```

```
arr[i + 1], arr[high] = arr[high], arr[i + 1]
return i + 1
```

### Implementação C:

```
#include <stdio.h>

void trocar(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Pivô é o último elemento
    int i = (low - 1); // Índice do menor elemento

    for (int j = low; j <= high - 1; j++) {
        // Se elemento atual é menor ou igual ao pivô
        if (arr[j] <= pivot) {
            i++;
            trocar(&arr[i], &arr[j]);
        }
    }

    trocar(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quick_sort(int arr[], int low, int high) {
    if (low < high) {
        // pi é o índice de partição
        int pi = partition(arr, low, high);

        // Ordena elementos antes e depois da partição
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
    }
}
```

# CAPÍTULO 6

## ALGORITMOS DE BUSCA

### 6.1 Algoritmos de Busca Fundamentais

#### Busca Linear

**Conceito:** Percorre o array sequencialmente até encontrar o elemento ou chegar ao final.

**Implementação Python:**

```
def busca_linear(arr, x):
    """
        Busca linear em array não ordenado
        Retorna o índice do elemento ou -1 se não encontrado
    """
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1

# Versão com informações de debug
def busca_linear_debug(arr, x):
    print(f"Buscando {x} em {arr}")
    comparacoes = 0

    for i in range(len(arr)):
        comparacoes += 1
        print(f" Comparação {comparacoes}: arr[{i}] = {arr[i]}")

        if arr[i] == x:
            print(f" Encontrado! Posição {i}")
            print(f" Total de comparações: {comparacoes}")
            return i

    print(f" Não encontrado após {comparacoes} comparações")
    return -1

# Teste
lista = [64, 34, 25, 12, 22, 11, 90]
elemento = 22
resultado = busca_linear_debug(lista, elemento)
```

**Implementação C:**

```
#include <stdio.h>

int busca_linear(int arr[], int n, int x) {
```

```

for (int i = 0; i < n; i++) {
    if (arr[i] == x) {
        return i; // Retorna o índice se encontrado
    }
}
return -1; // Retorna -1 se não encontrado
}

int busca_linear_debug(int arr[], int n, int x) {
    printf("Buscando %d no array\n", x);

    for (int i = 0; i < n; i++) {
        printf(" Comparação %d: arr[%d] = %d\n", i + 1, i, arr[i]);

        if (arr[i] == x) {
            printf(" Encontrado na posição %d!\n", i);
            return i;
        }
    }

    printf(" Elemento não encontrado\n");
    return -1;
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 22;

    int resultado = busca_linear_debug(arr, n, x);

    if (resultado != -1) {
        printf("Elemento %d encontrado no índice %d\n", x, resultado);
    } else {
        printf("Elemento %d não encontrado\n", x);
    }

    return 0;
}

```

## Busca Binária

**Conceito:** Divide repetidamente o array ordenado pela metade, comparando o elemento do meio com o elemento procurado.

### Implementação Python (Iterativa):

```

def busca_binaria_iterativa(arr, x):
    """
    Busca binária iterativa em array ordenado
    Retorna o índice do elemento ou -1 se não encontrado
    """

```

```

esquerda, direita = 0, len(arr) - 1

while esquerda <= direita:
    meio = (esquerda + direita) // 2

    if arr[meio] == x:
        return meio
    elif arr[meio] < x:
        esquerda = meio + 1
    else:
        direita = meio - 1

return -1

# Versão com debug
def busca_binaria_debug(arr, x):
    print(f"Buscando {x} em array ordenado: {arr}")
    esquerda, direita = 0, len(arr) - 1
    comparacoes = 0

    while esquerda <= direita:
        meio = (esquerda + direita) // 2
        comparacoes += 1

        print(f"  Comparaçao {comparacoes}: esq={esquerda}, dir={direita}, meio={meio}")
        print(f"    arr[{meio}] = {arr[meio]}")

        if arr[meio] == x:
            print(f"  Encontrado! Posição {meio}")
            print(f"  Total de comparações: {comparacoes}")

## **6.2 Busca Binária**

### **Conceito: Dividir pela Metade**
Funciona apenas em arrays **ordenados**. Compara com o elemento do meio e elimina metade da busca.

**Implementação Simples:**
```python
def busca_binaria(arr, x):
    esq, dir = 0, len(arr) - 1

    while esq <= dir:
        meio = (esq + dir) // 2

        if arr[meio] == x:
            return meio
        elif arr[meio] < x:
            esq = meio + 1 # Busca na metade direita
        else:
            dir = meio - 1 # Busca na metade esquerda

    return -1 # Não encontrado

```

```
# Macete: Sempre elimina metade das possibilidades
```

### Versão Recursiva:

```
def busca_binaria_rec(arr, x, esq=0, dir=None):
    if dir is None:
        dir = len(arr) - 1

    if esq > dir:
        return -1

    meio = (esq + dir) // 2

    if arr[meio] == x:
        return meio
    elif arr[meio] < x:
        return busca_binaria_rec(arr, x, meio + 1, dir)
    else:
        return busca_binaria_rec(arr, x, esq, meio - 1)
```

### Comparação: Linear vs Binária

Aspecto	Linear	Binária
<b>Complexidade</b>	$O(n)$	$O(\log n)$
<b>Pré-requisito</b>	Nenhum	Array ordenado
<b>Array 1.000</b>	500 comparações	10 comparações
<b>Array 1.000.000</b>	500.000 comparações	20 comparações

### Quando usar cada uma:

- **Linear:** Arrays pequenos ou não ordenados
- **Binária:** Arrays grandes e ordenados

### Macete para Lembrar:

- Busca Linear = "um por um" =  $O(n)$
- Busca Binária = "corta pela metade" =  $O(\log n)$

## 3.4 Exercícios de Fixação - Capítulo 3

### Exercício 3.1: Análise Básica de Complexidade

Determine a complexidade Big-O dos seguintes códigos:

```
# Código A
def codigo_a(n):
    count = 0
    for i in range(n):
```

```

        count += 1
    return count

# Código B
def codigo_b(n):
    count = 0
    for i in range(n):
        for j in range(n):
            count += 1
    return count

# Código C
def codigo_c(n):
    count = 0
    i = 1
    while i < n:
        count += 1
        i *= 2
    return count

```

**Respostas:** A = O(n), B = O( $n^2$ ), C = O(log n)

### Exercício 3.2: Comparaçāo de Algoritmos

Para  $n = 1000$ , calcule aproximadamente quantas operações cada complexidade executaria:

1. O(1): \_\_\_\_ operações
2. O(log n): \_\_\_\_ operações
3. O(n): \_\_\_\_ operações
4. O( $n^2$ ): \_\_\_\_ operações

**Respostas:** 1, 10, 1000, 1.000.000

### Exercício 3.3: Problema Prático

Um algoritmo de busca tem complexidade O(log n) e leva 1ms para processar 1000 elementos. Quanto tempo levará para processar 1.000.000 de elementos?

**Resposta:** Aproximadamente 2ms ( $\log_2(1.000.000) \approx 20$ ,  $\log_2(1000) \approx 10$ , então  $20/10 = 2x$ )

---





## CAPÍTULO 4

### RECURSIVIDADE - A MAGIA DE SE CHAMAR

🎯 **Objetivo:** Dominar a arte de resolver problemas dividindo-os em versões menores

#### 💡 4.1 Recursividade Descomplicada - A Escada Mágica

##### 📘 O que é Recursividade?

**Analogia da Escada Mágica:**

- 🎯 **Objetivo:** Subir N degraus
- 📈 **Regra:** "Para subir N degraus, suba 1 e depois suba os N-1 restantes"
- ⚪ **Parada:** "Se N=0, você chegou no topo!"

**Em programação:** Uma função que chama **ela mesma** para resolver problemas menores do mesmo tipo.

#### 🎯 MACETE FUNDAMENTAL - "BRP" da Recursividade

**Para criar qualquer recursão, você precisa de:**

- **Base** - Quando parar (caso base)
- **Recursão** - Como chamar a si mesmo
- **Progresso** - Problema deve diminuir

💡 **Macete de Memorização:** "Boa Recursão Precisa de base, recursão e progresso!"

#### ✳️ 4.2 Os 3 Ingredientes Sagrados da Recursividade

 CASO BASE	 CASO RECURSIVO	 PROGRESSO
<b>O que é:</b> Condição que PARA a recursão	<b>O que é:</b> Função chama ela mesma	<b>O que é:</b> Caminhando para o caso base
<b>⚠ Sem ele:</b> Loop infinito = Crash!	<b>✓ Regra:</b> Problema deve ser MENOR	<b>✓ Garantia:</b> Cada chamada é mais simples
<b>💡 Exemplo:</b> if n == 0: return 1	<b>💡 Exemplo:</b> return n * factorial(n-1)	<b>💡 Exemplo:</b> n-1, n/2, tamanho/2

### Exemplo Clássico - Fatorial Explicado Passo a Passo

```
# 📈 Fatorial Recursivo - Versão Comentada
def factorial(n):
    # ● CASO BASE - Quando parar
    if n == 0 or n == 1:
        return 1

    # 🔍 CASO RECURSIVO - Chama a si mesmo com problema menor
    # ↗ PROGRESSO - n-1 é menor que n, então progredimos para o caso base
    return n * factorial(n - 1)

# 🎥 RASTREAMENTO: factorial(4)
# factorial(4) → 4 * factorial(3)
#           ↓
#           3 * factorial(2)
#           ↓
#           2 * factorial(1)
#           ↓
#           1 (caso base)
#
# 📐 RESULTADO: 4 * 3 * 2 * 1 = 24
```

 **DICA VISUAL:** Pense na recursão como uma pilha de pratos - você empilha as chamadas (ida) e depois desempilha os resultados (volta)!

## 4.3 Tipos de Recursividade - O Cardápio Recursivo

### MACETE DOS TIPOS - "SLMT"

Simples - Uma chamada recursiva Linear - Chamadas em sequência

Múltipla - Várias chamadas recursivas **Tail** - Recursão no final (otimizável)

### ● RECURSÃO SIMPLES

Uma chamada por execução

```
``python def potencia(base, exp): if exp == 0:  
    return 1 return base * potencia(base, exp-1) ``
```

### ● RECURSÃO MÚLTIPLA

Múltiplas chamadas por execução

```
``python def fibonacci(n): if n <= 1: return n  
    return fibonacci(n-1) + fibonacci(n-2) ``
```

## 🔥 MACETE PARA FIBONACCI - "O Problema Clássico"

Fibonacci sem memoização = LENTO 🚫

**Problema:** fib(5) calcula fib(3) várias vezes!

**Solução:** Memoização (guardar resultados já calculados)

```
# 🚫 LENTO: O(2n) - exponencial  
def fib_lento(n):  
    if n <= 1: return n  
    return fib_lento(n-1) + fib_lento(n-2)  
  
# 🔥 RÁPIDO: O(n) - com memoização  
def fib_rapido(n, memo={}):  
    if n in memo: return memo[n]  
    if n <= 1: return n  
    memo[n] = fib_rapido(n-1, memo) + fib_rapido(n-2, memo)  
    return memo[n]
```

## 💡 4.4 Estratégias de Otimização - Tornando Recursão Eficiente

### 🎯 MACETE DAS OTIMIZAÇÕES - "MRIT"

**Memoização** - Guardar resultados já calculados **Recursão de cauda** - Otimizar última chamada **Iterativo** - Converter para loop quando possível  
**Tabela** - Programação dinâmica (bottom-up)

### 💾 Técnica 1: Memoização - A Memória da Recursão

```

# 💡 MEMOIZAÇÃO: Guardar resultados para evitar recálculos
def fibonacci_memo(n, memo={}):
    if n in memo:           # 🔎 Já calculamos? Use o resultado!
        return memo[n]

    if n <= 1:             # 🟣 Caso base
        return n

    # 🗑 Calcular, guardar e retornar
    memo[n] = fibonacci_memo(n-1, memo) + fibonacci_memo(n-2, memo)
    return memo[n]

# 📈 Performance: O(2n) → O(n) - IMPRESSIONANTE!

```

## 🔗 Técnica 2: Recursão → Iteração

```

# 💡 RECURSIVO: Elegante mas pode dar stack overflow
def fatorial_recursivo(n):
    if n <= 1: return 1
    return n * fatorial_recursivo(n-1)

# 💡 ITERATIVO: Mais eficiente em espaço
def fatorial_iterativo(n):
    resultado = 1
    for i in range(1, n + 1):
        resultado *= i
    return resultado

# 📈 Espaço: O(n) → O(1) - Muito melhor para números grandes!

```

## 🎯 MACETE PARA ESCOLHER - "Quando Usar O Quê?"

🎯 Situação	⌚ Recursivo	⌚ Iterativo
📘 Legibilidade	✓ Mais limpo	✗ Mais verboso
⚡ Performance	✗ Mais lento	✓ Mais rápido
💻 Memória	✗ Usa pilha	✓ Menos memória
🌳 Árvores/Grafos	✓ Natural	✗ Complexo
📊 Fibonacci	⚠ Com memo	✓ Simples

💡 **Regra de Ouro:** Use recursivo para estruturas naturalmente recursivas (árvores), iterativo para sequências simples!

Cada chamada deve nos aproximar da parada `''

## Receita Universal para Recursividade

```
def minha_funcao_recursiva(problema):
    # PRIMEIRO: Verificar caso base
    if problema_muito_simples:
        return solucao_direta

    # SEGUNDO: Quebrar o problema
    problema_menor = reduzir_problema(problema)

    # TERCEIRO: Chamar recursivamente
    resultado_parcial = minha_funcao_recursiva(problema_menor)

    # QUARTO: Combinar resultado
    return combinar(problema_atual, resultado_parcial)
```

## Exemplos Explicados Passo a Passo

### Exemplo 1: Fatorial - O Clássico

#### Como Pensar:

"Para calcular  $5!$ , preciso de  $5 \times 4!$ . Para calcular  $4!$ , preciso de  $4 \times 3!$ ..."

#### Definição Matemática:

```
n! = n × (n-1) × (n-2) × ... × 1
Casos especiais: 0! = 1, 1! = 1
```

#### Implementação Comentada:

```
def fatorial(n):
    # CASO BASE: números pequenos têm resposta direta
    if n == 0 or n == 1:
        print(f"  Caso base: {n}! = 1")
        return 1

    # CASO RECURSIVO: quebrar o problema
    print(f"  Calculando {n}! = {n} × {n-1}!")
    resultado_menor = fatorial(n - 1)  # Problema menor
    resultado_final = n * resultado_menor  # Combinar

    print(f"  Resultado: {n}! = {resultado_final}")
    return resultado_final

# Testando:
print("Calculando 4!:")
```

```
resultado = fatorial(4)
print(f"Resposta final: {resultado}")
```

## Filme da Execução:

```
Calculando 4!:
    Calculando 4! = 4 × 3!
    Calculando 3! = 3 × 2!
    Calculando 2! = 2 × 1!
    Caso base: 1! = 1
    Resultado: 2! = 2
    Resultado: 3! = 6
    Resultado: 4! = 24
Resposta final: 24
```

## **Visualização da Pilha de Chamadas:**

Descendo (Chamadas):	Subindo (Retornos):
fatorial(4)	fatorial(4) ← 24
└ fatorial(3)	└ fatorial(3) ← 6
└ fatorial(2)	└ fatorial(2) ← 2
└ fatorial(1)	└ fatorial(1) ← 1
└ retorna 1	└ 2 × 1 = 2
	└ 3 × 2 = 6
	└ 4 × 6 = 24

## **Exemplo 2: Fibonacci - O Famoso**

## Como Pensar:

"Para saber quantos coelhos tem no mês N, preciso somar os coelhos do mês N-1 com os do mês N-2"

## A Sequência:

$F(0)=0$ ,  $F(1)=1$ ,  $F(2)=1$ ,  $F(3)=2$ ,  $F(4)=3$ ,  $F(5)=5$ ,  $F(6)=8\dots$   
Cada número = soma dos dois anteriores

### **Versão Simples (Ineficiente):**

```
def fibonacci_simples(n):
    print(f"  Calculando F({n})")

    # CASOS BASE
    if n == 0:
        print(f"  Caso base: F(0) = 0")
        return 0

    if n == 1:
        print(f"  Caso base: F(1) = 1")
        return 1

    # CASO RECURSIVO: somar os dois anteriores
```

```

print(f" F({n}) = F({n-1}) + F({n-2})")
esquerda = fibonacci_simples(n - 1)
direita = fibonacci_simples(n - 2)
resultado = esquerda + direita

print(f" F({n}) = {esquerda} + {direita} = {resultado}")
return resultado

# Problema: O(2n) - muito lento!

```

### Versão Otimizada com Memoização:

```

def fibonacci_otimizado(n, memo={}):
    """
    Memo = dicionário que lembra resultados já calculados
    Se já calculamos F(n) antes, só retornamos o valor salvo!
    """

    # Já calculamos antes?
    if n in memo:
        print(f" Cache hit! F({n}) = {memo[n]} (já sabia)")
        return memo[n]

    print(f" Calculando F({n}) pela primeira vez")

    # CASOS BASE
    if n == 0:
        memo[n] = 0
        return 0
    if n == 1:
        memo[n] = 1
        return 1

    # CASO RECURSIVO
    resultado = fibonacci_otimizado(n-1, memo) + fibonacci_otimizado(n-2, memo)
    memo[n] = resultado # Salvar para próxima vez

    print(f" Salvando F({n}) = {resultado}")
    return resultado

# Complexidade melhora de O(2n) para O(n)!

```

### Comparação de Performance:

```

import time

# Teste com n=35
n = 35

# Versão lenta

```

```

inicio = time.time()
resultado1 = fibonacci_simples(35) # Demora ~10 segundos
tempo1 = time.time() - inicio

# Versão rápida
inicio = time.time()
resultado2 = fibonacci_otimizado(35) # Demora ~0.001 segundos
tempo2 = time.time() - inicio

print(f"Simples: {tempo1:.3f}s")
print(f"Otimizado: {tempo2:.6f}s")
print(f"Melhoria: {tempo1/tempo2:.0f}x mais rápido!")

```

### Exemplo 3: Torres de Hanói - O Espetacular

#### O Problema:

- 3 torres: A, B, C
- N discos em A (maior embaixo, menor em cima)
- **Objetivo:** Mover todos para C
- **Regras:**
  - Só move 1 disco por vez
  - Só pega o disco do topo
  - Nunca põe disco maior sobre menor

#### Como Pensar Recursivamente:

"Para mover N discos de A para C:"

1. Mova N-1 discos de A para B (usando C como auxiliar)
2. Mova o disco grande de A para C
3. Mova N-1 discos de B para C (usando A como auxiliar)

#### Implementação Explicada:

```

def torres_hanoi(n, origem, destino, auxiliar, nivel=0):
    """
    n = número de discos
    origem = torre de onde tirar
    destino = torre para onde levar
    auxiliar = torre temporária
    nivel = para identar a saída
    """

    indentacao = "    " * nivel # Para visualizar a recursão

    # CASO BASE: só 1 disco
    if n == 1:
        print(f"{indentacao}Mover disco {n} de {origem} → {destino}")
        return 1 # 1 movimento

    print(f"{indentacao}Para mover {n} discos de {origem} → {destino}:")

```

```

# PASSO 1: Mover n-1 discos para auxiliar
print(f"{indentacao} 1. Primeiro: mover {n-1} discos {origem} → {auxiliar}")
mov1 = torres_hanoi(n-1, origem, auxiliar, destino, nivel+1)

# PASSO 2: Mover o disco grande
print(f"{indentacao} 2. Depois: mover disco {n} de {origem} → {destino}")
mov2 = 1

# PASSO 3: Mover n-1 discos da auxiliar para destino
print(f"{indentacao} 3. Finalmente: mover {n-1} discos {auxiliar} → {destino}")
mov3 = torres_hanoi(n-1, auxiliar, destino, origem, nivel+1)

total = mov1 + mov2 + mov3
print(f"{indentacao}Total para {n} discos: {total} movimentos")
return total

# Testando:
print("Resolvendo Torres de Hanói com 3 discos:")
movimentos = torres_hanoi(3, 'A', 'C', 'B')
print(f"\nResolvido em {movimentos} movimentos!")
print(f"Fórmula: 2^n - 1 = 2^3 - 1 = {2**3 - 1}")

```

## Recursividade vs Iteração - O Duelo

### Comparação Lado a Lado

#### Fatorial Recursivo vs Iterativo:

##### Versão Recursiva:

```

def fatorial_recursivo(n):
    if n <= 1:
        return 1
    return n * fatorial_recursivo(n - 1)

```

##### Versão Iterativa:

```

def fatorial_iterativo(n):
    resultado = 1
    for i in range(1, n + 1):
        resultado *= i
    return resultado

```

##### Versão em C - Recursiva:

```

#include <stdio.h>

int fatorial_recursivo(int n) {
    if (n <= 1) {
        return 1;
    }
    return n * fatorial_recursivo(n - 1);
}

```

```

    }
    return n * fatorial_recursivo(n - 1);
}

int main() {
    int num = 5;
    printf("Fatorial de %d = %d\n", num, fatorial_recursivo(num));
    return 0;
}

```

#### Versão em C - Iterativa:

```

#include <stdio.h>

int fatorial_iterativo(int n) {
    int resultado = 1;
    for (int i = 1; i <= n; i++) {
        resultado *= i;
    }
    return resultado;
}

int main() {
    int num = 5;
    printf("Fatorial de %d = %d\n", num, fatorial_iterativo(num));
    return 0;
}

```

#### Análise Comparativa:

##### Recursivo:

- ✓ Mais elegante e legível
- ✓ Mais próximo da definição matemática
- X Usa mais memória (pilha)
- X Risco de stack overflow

##### Iterativo:

- ✓ Mais eficiente em memória
- ✓ Mais rápido na execução
- X Menos intuitivo
- X Mais código para casos complexos

#### Quando Usar Cada Um

##### Use Recursividade Quando:

- O problema tem **estrutura naturalmente recursiva** (árvores, fractais)
- A solução recursiva é **muito mais clara** que a iterativa
- Você pode **otimizar** com memoização se necessário
- A **profundidade é limitada** (não vai estourar a pilha)

##### Use Iteração Quando:

- **Performance** é crítica
  - A **profundidade** pode ser muito grande
  - A versão iterativa é **simples** de implementar
  - **Memória** é limitada
- 

## Tipos Especiais de Recursividade

### 1. Recursividade Linear

```
# Cada chamada gera APENAS UMA nova chamada
def conta_regressiva(n):
    if n <= 0:
        print("Fogo!")
        return

    print(f"{n}...")
    conta_regressiva(n - 1) # Uma só chamada

# Complexidade: O(n) tempo, O(n) espaço
```

#### Implementação em C:

```
#include <stdio.h>

void conta_regressiva(int n) {
    if (n <= 0) {
        printf("Fogo!\n");
        return;
    }

    printf("%d...\n", n);
    conta_regressiva(n - 1);
}

int main() {
    conta_regressiva(5);
    return 0;
}
```

### 2. Recursividade Binária

```
# Cada chamada gera DUAS novas chamadas
def fibonacci_binario(n):
    if n <= 1:
        return n

    return fibonacci_binario(n-1) + fibonacci_binario(n-2)
        #      ↑ chamada 1      ↑ chamada 2
```

```
# Complexidade: O(2n) tempo - cuidado!
```

### Implementação em C:

```
#include <stdio.h>

int fibonacci_binario(int n) {
    if (n <= 1) {
        return n;
    }

    return fibonacci_binario(n - 1) + fibonacci_binario(n - 2);
}

int main() {
    int num = 10;
    printf("Fibonacci de %d = %d\n", num, fibonacci_binario(num));
    return 0;
}
```

### 3. Recursividade de Cauda (Tail Recursion)

```
# A chamada recursiva é a ÚLTIMA operação
def fatorial_cauda(n, acumulador=1):
    if n <= 1:
        return acumulador

    # Última operação = chamada recursiva
    return fatorial_cauda(n - 1, n * acumulador)

# Vantagem: Pode ser otimizada pelo compilador para O(1) espaço
```

### Implementação em C:

```
#include <stdio.h>

int fatorial_cauda(int n, int acumulador) {
    if (n <= 1) {
        return acumulador;
    }

    return fatorial_cauda(n - 1, n * acumulador);
}

int main() {
    int num = 5;
    printf("Fatorial de %d = %d\n", num, fatorial_cauda(num, 1));
```

```
        return 0;
}
```

## 4. Recursividade Mútua

```
# Duas funções se chamam mutuamente
def eh_par(n):
    if n == 0:
        return True
    return eh_impar(n - 1)

def eh_impar(n):
    if n == 0:
        return False
    return eh_par(n - 1)

# Exemplo: eh_par(4) → eh_impar(3) → eh_par(2) → eh_impar(1) → eh_par(0) → True
```

### Implementação em C:

```
#include <stdio.h>
#include <stdbool.h>

bool eh_impar(int n); // Declaração antecipada

bool eh_par(int n) {
    if (n == 0) {
        return true;
    }
    return eh_impar(n - 1);
}

bool eh_impar(int n) {
    ---
}

## **4.5 Técnicas de Otimização**

### **Memoização - "Cache Inteligente"**
```python
# ❌ Lento: O(2^n)
def fib_lento(n):
    if n <= 1: return n
    return fib_lento(n-1) + fib_lento(n-2)

# ✅ Rápido: O(n)
def fib_rapido(n, cache={}):
    if n in cache:
        return cache[n]
    if n <= 1:
        return n
    cache[n] = fib_rapido(n-1) + fib_rapido(n-2)
    return cache[n]
```

```

cache[n] = fib_rapido(n-1, cache) + fib_rapido(n-2, cache)
return cache[n]

# Macete: Guardar resultados para não recalcular

```

## Programação Dinâmica Bottom-Up

```

def fib_iterativo(n):
    if n <= 1: return n

    a, b = 0, 1
    for i in range(2, n + 1):
        a, b = b, a + b
    return b

# Macete: Construir de baixo para cima, sem recursão

```

## Recursividade vs Iteração

Aspecto	Recursão	Iteração
<b>Legibilidade</b>	✓ Mais clara	X Mais verbosa
<b>Memória</b>	X Usa pilha	✓ Constante
<b>Performance</b>	X Mais lenta	✓ Mais rápida
<b>Stack Overflow</b>	X Risco	✓ Sem risco

### Quando usar recursão:

- Problemas naturalmente recursivos (árvores, fractais)
- Código mais limpo e legível
- Profundidade limitada

### Quando usar iteração:

- Performance crítica
- Grandes volumes de dados
- Memória limitada

**Vantagem:** Pode ser otimizada pelo compilador para usar espaço constante.

## 4. Recursividade Mútua

Duas ou mais funções se chamam mutuamente.

```

def eh_par(n):
    if n == 0:
        return True
    return eh_impar(n - 1)

def eh_impar(n):

```

```
if n == 0:  
    return False  
return eh_par(n - 1)
```

## Técnicas de Otimização

### 1. Memoização

Armazenar resultados de chamadas anteriores para evitar recálculos.

```
# Fibonacci com memoização usando decorador  
from functools import lru_cache  
  
@lru_cache(maxsize=None)  
def fibonacci_otimizado(n):  
    if n <= 1:  
        return n  
    return fibonacci_otimizado(n - 1) + fibonacci_otimizado(n - 2)
```

### 2. Programação Dinâmica Bottom-Up

Construir a solução de baixo para cima.

```
def fibonacci_dp(n):  
    if n <= 1:  
        return n  
  
    dp = [0] * (n + 1)  
    dp[1] = 1  
  
    for i in range(2, n + 1):  
        dp[i] = dp[i - 1] + dp[i - 2]  
  
    return dp[n]
```

## Problemas Comuns e Como Resolver

### 1. Stack Overflow - A Pilha Explodiu!

O que acontece:

```
def conta_infinita(n):  
    print(n)  
    return conta_infinita(n + 1) # ❌ Nunca para!  
  
# RecursionError: maximum recursion depth exceeded
```

Como resolver:

```
# ✅ Sempre tenha um caso base claro:
def conta_segura(n, limite=1000):
    if n >= limite: # Caso base
        print("Parou!")
        return

    print(n)
    conta_segura(n + 1, limite)

# ✅ Ou aumente o limite (use com cuidado):
import sys
sys.setrecursionlimit(10000) # Padrão: ~1000
```

## 2. Casos Base Incorretos

### ✗ Problemas comuns:

```
# Problema 1: Esqueceu caso base
def soma_lista(lista):
    return lista[0] + soma_lista(lista[1:]) # ✗ E se lista vazia?

# Problema 2: Caso base errado
def fatorial_errado(n):
    if n == 1: # ✗ E se n = 0?
        return 1
    return n * fatorial_errado(n - 1)

# Problema 3: Não progride para caso base
def loop_infinito(n):
    if n == 0:
        return 0
    return loop_infinito(n) # ✗ n nunca diminui!
```

### ✓ Versões corretas:

```
# ✅ Sempre trate o caso vazio
def soma_lista_certa(lista):
    if not lista: # Lista vazia
        return 0
    return lista[0] + soma_lista_certa(lista[1:])

# ✅ Cubra todos os casos base
def fatorial_certo(n):
    if n <= 1: # Cobre 0 e 1
        return 1
    return n * fatorial_certo(n - 1)

# ✅ Sempre faça progresso
def contagem_certa(n):
    if n <= 0:
```

```
    return 0
return contagem_certa(n - 1) # n diminui!
```

### 3. Debugging de Recursividade

#### Técnica do Print Investigativo:

```
def debug_fibonacci(n, nivel=0):
    indentacao = " " * nivel
    print(f"{indentacao}→ Entrando: fibonacci({n})")

    if n <= 1:
        print(f"{indentacao}← Saindo: fibonacci({n}) = {n}")
        return n

    esquerda = debug_fibonacci(n-1, nivel+1)
    direita = debug_fibonacci(n-2, nivel+1)
    resultado = esquerda + direita

    print(f"{indentacao}← Saindo: fibonacci({n}) = {resultado}")
    return resultado

# Teste: debug_fibonacci(4)
# Você verá exatamente o que está acontecendo!
```

#### Contando Chamadas:

```
contador_chamadas = 0

def fibonacci_contador(n):
    global contador_chamadas
    contador_chamadas += 1

    if n <= 1:
        return n
    return fibonacci_contador(n-1) + fibonacci_contador(n-2)

# Teste:
contador_chamadas = 0
resultado = fibonacci_contador(10)
print(f"Resultado: {resultado}")
print(f"Chamadas: {contador_chamadas}")
# Fibonacci(10) faz 177 chamadas!
```

## Dicas de Ouro para Recursividade

### 1. Como Projetar uma Função Recursiva:

#### Passo 1: Identifique o padrão

"Para resolver problema de tamanho N,  
posso usar a solução de tamanho N-1?"

### **Passo 2: Encontre o caso mais simples**

"Qual é o menor problema que sei resolver diretamente?"

### **Passo 3: Conecte os dois**

"Como combino a solução menor com o problema atual?"

### **Exemplo Prático: Soma de Lista**

```
# Passo 1: Padrão
# soma([1,2,3,4]) = 1 + soma([2,3,4])

# Passo 2: Caso simples
# soma([]) = 0

# Passo 3: Conectar
def soma_lista(lista):
    if not lista: # Passo 2
        return 0
    return lista[0] + soma_lista(lista[1:]) # Passo 1
```

## **2. Truques Mentais:**

### **"Role-Playing" Mental:**

"Eu sou a função soma\_lista([1,2,3]).  
Meu trabalho é somar essa lista.  
Ei, função soma\_lista([2,3])! Você pode me ajudar?  
Depois eu só preciso somar 1 com sua resposta!"

### **"Princípio da Confiança":**

"Assumo que minha função funciona para problemas menores.  
Só preciso focar em como usar essa resposta."

## **3. Otimizações Práticas:**

### **Memoização Automática:**

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci_turbo(n):
    if n <= 1: return n
    return fibonacci_turbo(n-1) + fibonacci_turbo(n-2)
```

```
# Agora é O(n) automaticamente!
```

### Transformar em Iterativo:

```
# Se a recursividade está lenta, tente iterativo:  
def fibonacci_iterativo(n):  
    if n <= 1: return n  
  
    a, b = 0, 1  
    for _ in range(2, n + 1):  
        a, b = b, a + b  
    return b  
  
# Mesmo resultado, O(n) tempo, O(1) espaço!
```

## Exercícios Práticos - Do Básico ao Ninja

### Nível 1: Primeiro Contato

#### Exercício 1.1: Contagem Regressiva

```
# Implemente uma função que conta de n até 0  
def conta_regressiva(n):  
    # Seu código aqui  
    pass  
  
# Teste: conta_regressiva(5) deve imprimir: 5 4 3 2 1 0
```

#### Exercício 1.2: Soma Simples

```
# Some todos os números de 1 até n  
def soma_ate_n(n):  
    # Seu código aqui  
    pass  
  
# Teste: soma_ate_n(5) deve retornar 15 (1+2+3+4+5)
```

#### Exercício 1.3: Potência

```
# Calcule x^n recursivamente  
def potencia(x, n):  
    # Seu código aqui  
    pass  
  
# Teste: potencia(2, 3) deve retornar 8
```

## Nível 2: Esquentando

### Exercício 2.1: Máximo de Lista

```
# Encontre o maior número em uma lista
def maximo_lista(lista):
    # Seu código aqui
    pass

# Teste: maximo_lista([3, 1, 4, 1, 5]) deve retornar 5
```

### Exercício 2.2: Palíndromo

```
# Verifique se uma string é palíndromo
def eh_palindromo(s):
    # Seu código aqui
    pass

# Teste: eh_palindromo("arara") deve retornar True
```

### Exercício 2.3: Busca Binária

```
# Implemente busca binária recursivamente
def busca_binaria(lista, elemento, inicio=0, fim=None):
    # Seu código aqui
    pass

# Teste: busca_binaria([1,2,3,4,5], 3) deve retornar 2
```

## Nível 3: Desafio

### Exercício 3.1: Permutações

```
# Gere todas as permutações de uma string
def permutacoes(s):
    # Seu código aqui
    pass

# Teste: permutacoes("abc") deve retornar ["abc", "acb", "bac", "bca", "cab", "cba"]
```

### Exercício 3.2: Subconjuntos

```
# Gere todos os subconjuntos de uma lista
def subconjuntos(lista):
    # Seu código aqui
    pass
```

```
# Teste: subconjuntos([1,2]) deve retornar [[], [1], [2], [1,2]]
```

## Soluções Comentadas:

### Solução 1.1:

```
def conta_regressiva(n):
    # Caso base: quando chegar a zero, para
    if n < 0:
        return

    # Ação: imprimir número atual
    print(n)

    # Caso recursivo: chamar com n-1
    conta_regressiva(n - 1)
```

### Solução 2.2:

```
def eh_palindromo(s):
    # Caso base: string vazia ou 1 char é palíndromo
    if len(s) <= 1:
        return True

    # Verificar primeiro e último caracteres
    if s[0] != s[-1]:
        return False

    # Caso recursivo: verificar o meio
    return eh_palindromo(s[1:-1])
```

### Solução 3.1:

```
def permutacoes(s):
    # Caso base: string vazia
    if len(s) <= 1:
        return [s]

    resultado = []

    # Para cada caractere na string
    for i in range(len(s)):
        # Tira o caractere atual
        char = s[i]
        resto = s[:i] + s[i+1:]

        # Gera permutações do resto
        for perm in permutacoes(resto):
```

```

        resultado.append(char + perm)

    return resultado

```

## Exercícios Práticos de Recursividade

### Nível Básico:

1. **Potência:** Calcule  $x^n$  usando recursividade.
2. **Soma de Dígitos:** Some todos os dígitos de um número.
3. **Máximo em Lista:** Encontre o maior elemento de uma lista recursivamente.

### Nível Intermediário:

4. **Palíndromo:** Verifique se uma string é palíndromo.
5. **Busca Binária:** Implemente busca binária recursiva.
6. **GCD/MDC:** Calcule o máximo divisor comum usando algoritmo de Euclides.

### Nível Avançado:

7. **Permutações:** Gere todas as permutações de uma string.
8. **Subconjuntos:** Gere todos os subconjuntos de um conjunto.
9. **N-Queens:** Resolva o problema das N rainhas.

### Soluções dos Exercícios:

```

# 1. Potência
def potencia(x, n):
    if n == 0:
        return 1
    return x * potencia(x, n - 1)

# 2. Soma de Dígitos
def soma_digitos(n):
    if n < 10:
        return n
    return (n % 10) + soma_digitos(n // 10)

# 3. Máximo em Lista
def maximo_lista(lista):
    if len(lista) == 1:
        return lista[0]

    max_resto = maximo_lista(lista[1:])
    return lista[0] if lista[0] > max_resto else max_resto

# 4. Palíndromo
def eh_palindromo(s):
    if len(s) <= 1:
        return True

    if s[0] != s[-1]:
        return False

```

```
return eh_palindromo(s[1:-1])

# 5. Busca Binária Recursiva
def busca_binaria_rec(lista, elemento, inicio=0, fim=None):
    if fim is None:
        fim = len(lista) - 1

    if inicio > fim:
        return -1

    meio = (inicio + fim) // 2

    if lista[meio] == elemento:
        return meio
    elif lista[meio] < elemento:
        return busca_binaria_rec(lista, elemento, meio + 1, fim)
    else:
        return busca_binaria_rec(lista, elemento, inicio, meio - 1)

# 6. GCD (Algoritmo de Euclides)
def gcd(a, b):
    if b == 0:
        return a
    return gcd(b, a % b)
```

# CAPÍTULO 7

## ALGORITMOS DE ORDENAÇÃO AVANÇADOS

### 7.1 Análise dos Algoritmos Elementares

#### Limitações dos Algoritmos $O(n^2)$

```
# Bubble Sort - O(n2) - Só para ensinar!
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

# Macete: n2 operações = LENTO para n > 1000
```

### 7.2 Ordenação por Intercalação (MergeSort)

#### Macete: Divide e Conquista

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    # Divide
    meio = len(arr) // 2
    esq = merge_sort(arr[:meio])
    dir = merge_sort(arr[meio:])

    # Conquista (intercala)
    return merge(esq, dir)

def merge(esq, dir):
    resultado = []
    i = j = 0

    # Intercala ordenado
    while i < len(esq) and j < len(dir):
        if esq[i] <= dir[j]:
            resultado.append(esq[i])
            i += 1
        else:
            resultado.append(dir[j])
            j += 1

    return resultado
```

```

# Adiciona sobras
resultado.extend(esq[i:])
resultado.extend(dir[j:])
return resultado

# Complexidade: O(n log n) SEMPRE!
# Espaço: O(n) - precisa de array auxiliar

```

## 7.3 Ordenação Rápida (QuickSort)

### Macete: Pivô e Partição

```

def quick_sort(arr, inicio=0, fim=None):
    if fim is None:
        fim = len(arr) - 1

    if inicio < fim:
        # Partitiona e encontra pivô
        pivo = particionar(arr, inicio, fim)

        # Recursão nas duas partes
        quick_sort(arr, inicio, pivo - 1)
        quick_sort(arr, pivo + 1, fim)

def particionar(arr, inicio, fim):
    pivo = arr[fim]  # Último elemento como pivô
    i = inicio - 1  # Índice do menor elemento

    for j in range(inicio, fim):
        if arr[j] <= pivo:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[fim] = arr[fim], arr[i + 1]
    return i + 1

# Complexidade:
# Melhor/Médio: O(n log n)
# Pior: O(n2) - se sempre escolher pior pivô
# Espaço: O(log n) - recursão

```

## 7.4 ShellSort

### Macete: Insertion Sort com Gaps

```

def shell_sort(arr):
    n = len(arr)
    gap = n // 2  # Começa com gap = metade

```

```

while gap > 0:
    # Insertion sort com gap
    for i in range(gap, n):
        temp = arr[i]
        j = i

        while j >= gap and arr[j - gap] > temp:
            arr[j] = arr[j - gap]
            j -= gap

        arr[j] = temp

    gap //= 2 # Reduz gap pela metade

# Complexidade: O(n^1.25) a O(n^1.5)
# Melhor que O(n^2), pior que O(n log n)

```

### Comparação de Algoritmos Avançados

Algoritmo	Melhor	Médio	Pior	Espaço	Estável
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Sim
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	Não
ShellSort	$O(n \log n)$	$O(n^{1.25})$	$O(n^2)$	$O(1)$	Não

# CAPÍTULO 8

## ALGORITMOS EM ÁRVORES BINÁRIAS E AVL

### 8.1 Árvore Binária de Busca (BST)

#### Estrutura Básica

```
class No:
    def __init__(self, valor):
        self.valor = valor
        self.esquerda = None
        self.direita = None

class BST:
    def __init__(self):
        self.raiz = None
```

#### Busca - $O(\log n) / O(n)$

```
def buscar(self, valor, no=None):
    if no is None:
        no = self.raiz

    if no is None or no.valor == valor:
        return no

    # Macete: < vai esquerda, > vai direita
    if valor < no.valor:
        return self.buscar(valor, no.esquerda)
    else:
        return self.buscar(valor, no.direita)
```

#### Inserção - $O(\log n) / O(n)$

```
def inserir(self, valor):
    self.raiz = self._inserir_rec(self.raiz, valor)

def _inserir_rec(self, no, valor):
    if no is None:
        return No(valor)

    # Macete: menor esquerda, maior direita
    if valor < no.valor:
        no.esquerda = self._inserir_rec(no.esquerda, valor)
    elif valor > no.valor:
```

```

        no.direita = self._inserir_rec(no.direita, valor)

    return no

```

## Remoção - $O(\log n)$ / $O(n)$

```

def remover(self, valor):
    self.raiz = self._remover_rec(self.raiz, valor)

def _remover_rec(self, no, valor):
    if no is None:
        return no

    if valor < no.valor:
        no.esquerda = self._remover_rec(no.esquerda, valor)
    elif valor > no.valor:
        no.direita = self._remover_rec(no.direita, valor)
    else:
        # Achou o nó para remover
        if no.esquerda is None:
            return no.direita
        elif no.direita is None:
            return no.esquerda

        # Dois filhos: substitui pelo sucessor
        sucessor = self._minimo(no.direita)
        no.valor = sucessor.valor
        no.direita = self._remover_rec(no.direita, sucessor.valor)

    return no

def _minimo(self, no):
    while no.esquerda:
        no = no.esquerda
    return no

```

## 8.2 Percursos em Árvores

### Macetes dos Percursos

```

# In-Order: Esquerda → Raiz → Direita (ordem crescente em BST)
def in_order(self, no):
    if no:
        self.in_order(no.esquerda)
        print(no.valor)           # Processa raiz
        self.in_order(no.direita)

# Pré-Order: Raiz → Esquerda → Direita (cópia da árvore)
def pre_order(self, no):

```

```

if no:
    print(no.valor)          # Processa raiz ANTES
    self.pre_order(no.esquerda)
    self.pre_order(no.direita)

# Pós-Order: Esquerda → Direita → Raiz (deletar árvore)
def pos_order(self, no):
    if no:
        self.pos_order(no.esquerda)
        self.pos_order(no.direita)
        print(no.valor)      # Processa raiz DEPOIS

```

### Complexidade dos Percursos: O(n)

Cada nó é visitado exatamente uma vez.

## 8.3 Balanceamento - Algoritmo DSW

### Problema: BST Degenerada

```

# Inserindo [1,2,3,4,5] sequencialmente vira lista ligada!
# Busca fica O(n) ao invés de O(log n)

```

### Algoritmo DSW (Day-Stout-Warren)

```

def balancear_dsw(self):
    # Fase 1: Criar "espinha dorsal" (vine)
    self._criar_vine()

    # Fase 2: Criar árvore balanceada
    n = self._contar_nos()
    self._vine_para_arvore(n)

def _criar_vine(self):
    # Rotações à direita para criar lista ligada à direita
    pseudo_raiz = No(0)
    pseudo_raiz.direita = self.raiz
    atual = pseudo_raiz

    while atual.direita:
        if atual.direita.esquerda:
            # Rotação à direita
            self._rotacao_direita(atual)
        else:
            atual = atual.direita

    self.raiz = pseudo_raiz.direita

# Complexidade DSW: O(n) - linear!

```

## 8.4 Árvore AVL

### Propriedade AVL

```
# Macete: Diferença de altura entre filhos ≤ 1
def altura(self, no):
    if no is None:
        return 0
    return max(self.altura(no.esquerda), self.altura(no.direita)) + 1

def fator_balanceamento(self, no):
    if no is None:
        return 0
    return self.altura(no.esquerda) - self.altura(no.direita)

def esta_balanceada(self, no):
    return abs(self.fator_balanceamento(no)) <= 1
```

### Rotações AVL

```
# Rotação Simples à Direita
def rotacao_direita(self, y):
    x = y.esquerda
    t2 = x.direita

    # Rotação
    x.direita = y
    y.esquerda = t2

    return x # Nova raiz

# Rotação Simples à Esquerda
def rotacao_esquerda(self, x):
    y = x.direita
    t2 = y.esquerda

    # Rotação
    y.esquerda = x
    x.direita = t2

    return y # Nova raiz

# Macete: 4 casos de rotação
# LL → Rotação direita
# RR → Rotação esquerda
# LR → Esquerda depois direita
# RL → Direita depois esquerda
```

### **Complexidade AVL: SEMPRE O(log n)**

- Busca:  $O(\log n)$
  - Inserção:  $O(\log n)$
  - Remoção:  $O(\log n)$
  - Altura máxima:  $1.44 \times \log_2(n)$
-

# CAPÍTULO 9

## ALGORITMOS EM GRAFOS

### 9.1 Conceitos Básicos

```
# Grafo = G(V, E) onde V = vértices, E = arestas  
# Tipos: Dirigido (setas), Não-dirigido, Ponderado (com pesos)
```

### 9.2 Representação

#### Lista de Adjacência (mais comum)

```
grafo = {  
    'A': ['B', 'C'],  
    'B': ['A', 'D'],  
    'C': ['A', 'D'],  
    'D': ['B', 'C']  
}  
# Espaço: O(V + E) - eficiente para grafos esparsos
```

#### Matriz de Adjacência

```
# matriz[i][j] = 1 se existe aresta entre i e j  
matriz = [  
    [0, 1, 1, 0], # A conecta com B, C  
    [1, 0, 0, 1], # B conecta com A, D  
    [1, 0, 0, 1], # C conecta com A, D  
    [0, 1, 1, 0]  # D conecta com B, C  
]  
# Espaço: O(V2) - melhor para grafos densos
```

### 9.3 Algoritmos de Busca

#### DFS - Busca em Profundidade

```
def dfs(grafo, inicio, visitados=None):  
    if visitados is None:  
        visitados = set()  
  
    visitados.add(inicio)  
    print(inicio)  
  
    for vizinho in grafo[inicio]:
```

```

        if vizinho not in visitados:
            dfs(grafo, vizinho, visitados)

    return visitados

# Macete: "Vai fundo" - usa pilha (recursão)
# Uso: detectar ciclos, componentes conectados

```

## BFS - Busca em Largura

```

from collections import deque

def bfs(grafo, inicio):
    visitados = set([inicio])
    fila = deque([inicio])

    while fila:
        vertice = fila.popleft()
        print(vertice)

        for vizinho in grafo[vertice]:
            if vizinho not in visitados:
                visitados.add(vizinho)
                fila.append(vizinho)

    return visitados

# Macete: "Vai por camadas" - usa fila
# Uso: menor caminho (sem pesos)

```

## 9.4 Caminho Mínimo

### Dijkstra (para pesos positivos)

```

import heapq

def dijkstra(grafo, inicio):
    distancias = {v: float('inf') for v in grafo}
    distancias[inicio] = 0
    heap = [(0, inicio)]

    while heap:
        dist_atual, vertice = heapq.heappop(heap)

        if dist_atual > distancias[vertice]:
            continue

        for vizinho, peso in grafo[vertice]:
            nova_dist = dist_atual + peso

```

```

        if nova_dist < distancias[vizinho]:
            distancias[vizinho] = nova_dist
            heapq.heappush(heap, (nova_dist, vizinho))

    return distancias

# Complexidade: O((V + E) log V)
# Macete: Sempre escolhe o vértice mais próximo

```

### Resumo - Quando Usar Cada Algoritmo

Algoritmo	Uso	Complexidade
<b>DFS</b>	Ciclos, componentes	$O(V + E)$
<b>BFS</b>	Menor caminho (sem peso)	$O(V + E)$
<b>Dijkstra</b>	Menor caminho (peso $\geq 0$ )	$O((V+E) \log V)$
<b>Floyd-Warshall</b>	Todos os pares	$O(V^3)$

```

for k in range(n):          # Vértice intermediário
    for i in range(n):      # Vértice origem
        for j in range(n):  # Vértice destino
            if dist[i][k] + dist[k][j] < dist[i][j]:
                dist[i][j] = dist[i][k] + dist[k][j]

return dist

```

## Macete: Todos os pares de vértices

**Funciona com pesos negativos (sem ciclos negativos)**

```

### **Resumo de Complexidades - Grafos**

| Operação | Lista Adj. | Matriz Adj. |
|-----|-----|-----|
| Espaço |  $O(V + E)$  |  $O(V^2)$  |
| Adicionar vértice |  $O(1)$  |  $O(V^2)$  |
| Adicionar aresta |  $O(1)$  |  $O(1)$  |
| Verificar aresta |  $O(V)$  |  $O(1)$  |
| DFS/BFS |  $O(V + E)$  |  $O(V^2)$  |

### **Quando Usar Cada Algoritmo**

```

BUSCA:

- DFS: Detectar ciclos, componentes, topologia

- BFS: Menor caminho (não ponderado), nível por nível

CAMINHO MÍNIMO:

- Dijkstra: Um para todos, pesos positivos
- Floyd-Warshall: Todos para todos, permite negativos
- Bellman-Ford: Um para todos, detecta ciclo negativo

```
---  
    if vizinho not in visitados:  
        dfs(grafo, vizinho, visitados)
```

**Busca em Largura (BFS):**

```
from collections import deque  
  
def bfs(grafo, inicio):  
    visitados = set()  
    fila = deque([inicio])  
  
    while fila:  
        no = fila.popleft()  
        if no not in visitados:  
            visitados.add(no)  
            print(no)  
            fila.extend(grafo[no])
```

# CAPÍTULO 7

## ANÁLISE AMORTIZADA

### 7.1 Conceitos e Aplicações

#### O que é Análise Amortizada?

A análise amortizada é uma técnica para analisar o tempo de execução de uma sequência de operações, onde algumas operações podem ser custosas, mas o custo médio por operação é baixo quando consideramos uma sequência longa de operações.

#### Diferença entre Análise Amortizada e Caso Médio

- **Caso Médio:** Considera a distribuição probabilística das entradas
- **Análise Amortizada:** Considera uma sequência de operações, garantindo que o custo total é limitado

#### Métodos de Análise Amortizada

##### 1. Método Agregado

**Princípio:** Mostrar que para qualquer sequência de  $n$  operações, o tempo total é  $T(n)$ , então cada operação custa  $T(n)/n$  em média.

##### Exemplo: Array Dinâmico

```
class ArrayDinamico:  
    def __init__(self):  
        self.capacity = 1  
        self.size = 0  
        self.data = [None] * self.capacity  
  
    def append(self, item):  
        if self.size == self.capacity:  
            # Redimensionar: O(n)  
            self._resize()  
  
        self.data[self.size] = item # O(1)  
        self.size += 1  
  
    def _resize(self):  
        old_capacity = self.capacity  
        self.capacity *= 2  
        new_data = [None] * self.capacity  
  
        # Copia todos os elementos: O(n)  
        for i in range(self.size):  
            new_data[i] = self.data[i]  
  
        self.data = new_data
```

```

# Análise:
# - Operação normal: O(1)
# - Redimensionamento: O(n), mas acontece raramente
# - Para n inserções: redimensiona em 1, 2, 4, 8, ..., k onde k ≤ n
# - Custo total de cópias: 1 + 2 + 4 + ... + k ≤ 2n
# - Custo amortizado por inserção: O(1)

```

## 2. Método do Contador

**Princípio:** Atribuir "créditos" para operações baratas que podem ser usados para pagar operações caras futuras.

**Exemplo: Stack com Array Dinâmico**

```

class StackDinamico:
    def __init__(self):
        self.capacity = 1
        self.size = 0
        self.data = [None] * self.capacity

    def push(self, item):
        # Custo real: O(1) normal ou O(n) com redimensionamento
        # Custo amortizado: O(1) + 2 créditos = O(1)

        if self.size == self.capacity:
            self._resize()

        self.data[self.size] = item
        self.size += 1

        # Cada push "paga" 3 unidades:
        # 1 para a inserção atual
        # 2 créditos para futuro redimensionamento

    def _resize(self):
        self.capacity *= 2
        new_data = [None] * self.capacity

        # Usa os créditos acumulados para pagar a cópia
        for i in range(self.size):
            new_data[i] = self.data[i]

        self.data = new_data

```

## 3. Método do Potencial

**Princípio:** Define uma função potencial  $\Phi(D)$  que mede a "energia armazenada" na estrutura de dados.

**Fórmula:** Custo amortizado = Custo real +  $\Phi(D')$  -  $\Phi(D)$

**Exemplo: Array Dinâmico com Potencial**

```

# Função potencial: Φ(D) = 2 * size - capacity
#
# Quando size está próximo de capacity, potencial é alto
# Após redimensionamento, potencial diminui drasticamente

def custo_amortizado_append():
    """
    Análise do custo amortizado usando potencial

    Caso 1: Inserção sem redimensionamento
    - Custo real: 1
    - Δ Potencial: 2 (size aumenta 1, capacity inalterada)
    - Custo amortizado: 1 + 2 = 3

    Caso 2: Inserção com redimensionamento (size = capacity = n)
    - Custo real: n + 1 (n cópias + 1 inserção)
    - Potencial antes: 2n - n = n
    - Potencial depois: 2(n+1) - 2n = 2
    - Δ Potencial: 2 - n = -(n-2)
    - Custo amortizado: (n + 1) + (-(n-2)) = 3

    Em ambos os casos: O(1) amortizado
    """
    pass

```

## Estruturas de Dados com Análise Amortizada

### Union-Find (Disjoint Set Union)

```

class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        # Compressão de caminho
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x]) # Recursão com compressão
        return self.parent[x]

    def union(self, x, y):
        # União por rank
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x != root_y:
            if self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            elif self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x

```

```

        else:
            self.parent[root_y] = root_x
            self.rank[root_x] += 1

# Análise amortizada:
# - Sem otimizações: O(n) por operação
# - Com compressão de caminho + união por rank: O( $\alpha(n)$ ) amortizado
# -  $\alpha(n)$  é a função inversa de Ackermann, praticamente constante

```

## Fibonacci Heap

# CAPÍTULO 10

## PROGRAMAÇÃO DINÂMICA

### 10.1 Conceito: "Dividir + Memorizar"

```

# Problema: Fibonacci ingênuo é muito lento
def fib_lento(n):
    if n <= 1: return n
    return fib_lento(n-1) + fib_lento(n-2) # O(2^n) - LENTO!

# Solução: Guardar resultados calculados
def fib_rapido(n):
    memo = {}
    def fib_aux(x):
        if x in memo: return memo[x]
        if x <= 1: return x
        memo[x] = fib_aux(x-1) + fib_aux(x-2)
        return memo[x]
    return fib_aux(n) # O(n) - RÁPIDO!

# Macete: DP = Recursão + Memoização

```

### 10.2 Problema da Mochila 0/1

```

def mochila(pesos, valores, capacidade):
    n = len(pesos)
    # dp[i][w] = valor máximo com i itens e capacidade w
    dp = [[0] * (capacidade + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, capacidade + 1):
            # Opção 1: não pegar item i-1
            dp[i][w] = dp[i-1][w]

            # Opção 2: pegar item i-1 (se couber)
            if pesos[i-1] <= w:
                dp[i][w] = max(dp[i-1][w], valores[i-1] + dp[i-1][w - pesos[i-1]])

```

```

        if pesos[i-1] <= w:
            pegar = valores[i-1] + dp[i-1][w - pesos[i-1]]
            dp[i][w] = max(dp[i][w], pegar)

    return dp[n][capacidade]

# Macete: Para cada item, decide "pegar ou não pegar"

```

## 10.3 Maior Subsequência Comum (LCS)

```

def lcs(str1, str2):
    m, n = len(str1), len(str2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if str1[i-1] == str2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1 # Caracteres iguais
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]) # Pega o melhor

    return dp[m][n]

# Exemplo: LCS("ABCD", "AEBD") = "ABD" (tamanho 3)

```

## 10.4 Quando Usar DP

- **Subproblemas sobrepostos:** mesmo cálculo repetido
- **Subestrutura ótima:** solução ótima contém soluções ótimas menores
- **Decisões sequenciais:** escolhas em cada passo

### Padrão Bottom-Up (sem recursão)

```

def fib_bottom_up(n):
    if n <= 1: return n
    a, b = 0, 1
    for i in range(2, n + 1):
        a, b = b, a + b
    return b

# Vantagem: sem risco de stack overflow

```

```

* Invariante: max_so_far é o maior elemento em arr[0..i-1]
*/
if (n <= 0) return -1; // Erro

int max_so_far = arr[0]; // Inicialização

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

```

```

// Invariante: max_so_far = max(arr[0..i-1])

if (arr[i] > max_so_far) {
    max_so_far = arr[i];
}

// Invariante mantida: max_so_far = max(arr[0..i])
}

// Terminação: max_so_far = max(arr[0..n-1])
return max_so_far;

}

void insertion_sort_c(int arr[], int n) { /* * Invariante: arr[0..i-1] está ordenado */ for (int i = 1; i < n; i++) { // Invariante: arr[0..i-1] está ordenado

    int key = arr[i];
    int j = i - 1;

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

    arr[j + 1] = key;

    // Invariante mantida: arr[0..i] está ordenado
}
// Terminação: arr[0..n-1] está ordenado

}

```

### ### Benefícios das Invariantes

1. **Prova de Correção**: Garantem que o algoritmo funciona
2. **Debugging**: Ajudam a encontrar bugs lógicos
3. **Otimização**: Identificam propriedades que podem ser exploradas
4. **Documentação**: Explicam como o algoritmo funciona
5. **Mantenção**: Facilitam modificações futuras

### ### Dicas para Criar Boas Invariantes

1. **Seja específico**: "arr está parcialmente ordenado" vs "arr[0..i] está ordenado"
2. **Use quantificadores**: "Para todo  $x$  em  $S$ , propriedade  $P(x)$  é verdadeira"
3. **Relacione com o objetivo**: A invariante deve levar ao resultado desejado
4. **Mantenha simples**: Invariantes complexas são difíceis de verificar
5. **Teste com exemplos**: Verifique a invariante em execuções específicas

---

```

## Exercícios Práticos

### Exercício 1: Análise de Complexidade
Determine a complexidade dos seguintes códigos:

```python
# a)
for i in range(n):
    for j in range(n):
        print(i, j)

# b)
def busca_binaria(lista, x):
    # ... implementação da busca binária

# c)
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

```

### **Exercício 2: Implementação**

Implemente um algoritmo de ordenação merge sort e analise sua complexidade.

### **Exercício 3: Recursividade Avançada**

Implemente uma função recursiva que calcule o número de formas de subir uma escada com  $n$  degraus, onde você pode subir 1 ou 2 degraus por vez.

### **Exercício 4: Programação Dinâmica**

Resolva o problema de encontrar a maior subsequência crescente em um array.

---

## **Resumo Visual dos Pontos Principais**

### **Complexidade - Cheat Sheet:**

COMPLEXIDADES DO MELHOR AO PIOR:

$O(1)$	- Acesso direto	[=====]
$O(\log n)$	- Busca inteligente	[== ]
$O(n)$	- Verificar todos	[=====]
$O(n \log n)$	- Ordenação boa	[=====]
$O(n^2)$	- Comparar todos x todos	[=====]
$O(2^n)$	- Explorar combinações	[XXXXXXXXXXXXXX]
$O(n!)$	- Impossível na prática	[XXXXXXXXXXXXXXXXXX]

### **Recursividade - Checklist:**

#### ANTES DE CODIFICAR:

- Identifiquei o padrão recursivo?
- Defini o caso base claramente?
- Cada chamada progride para o caso base?
- Testei com casos pequenos?

#### SINAIS DE ALERTA:

- Sem caso base → Loop infinito
- Caso base errado → Crash
- Não progride → Stack overflow
- Muito lento → Precisa otimizar

#### TÉCNICAS DE OTIMIZAÇÃO:

- Memoização → Guardar resultados
- Iteração → Quando possível
- Bottom-up → Programação dinâmica

### Kit de Sobrevivência do Programador:

#### Para Análise de Algoritmos:

```
# 1. Conte os loops:  
for i in range(n):      # O(n)  
    for j in range(n):  # x O(n) = O(n2)  
        operacao()      # O(1)  
  
# 2. Identifique o padrão:  
# - Dividir pela metade → O(log n)  
# - Visitar todos → O(n)  
# - Comparar todos x todos → O(n2)  
# - Dividir e conquistar → O(n log n)
```

#### Para Recursividade:

```
# Template universal:  
def resolver_recursivo(problema):  
    # SEMPRE primeiro: caso base  
    if problema_simples:  
        return solucao_direta  
  
    # Quebrar problema  
    subproblema = reduzir(problema)  
  
    # Resolver recursivamente  
    resultado_parcial = resolver_recursivo(subproblema)  
  
    # Combinar resultado  
    return combinar(problema, resultado_parcial)
```

## Estruturas de Dados - Guia Rápido:

Estrutura	Acesso	Busca	Inserção	Remoção	Quando Usar
<b>Array</b>	O(1)	O(n)	O(n)	O(n)	Acesso rápido por índice
<b>Lista Ligada</b>	O(n)	O(n)	O(1)*	O(1)*	Inserções/remoções frequentes
<b>Pilha</b>	O(1) topo	-	O(1)	O(1)	LIFO, desfazer, recursão
<b>Fila</b>	O(1) frente	-	O(1)	O(1)	FIFO, processamento ordem
<b>Hash Table</b>	O(1)*	O(1)*	O(1)*	O(1)*	Busca super rápida
<b>Árvore Binária</b>	O(log n)*	O(log n)*	O(log n)*	O(log n)*	Dados ordenados

\* No caso médio

## Algoritmos Essenciais:

### BUSCA:

Linear  $\rightarrow O(n)$   $\rightarrow$  Simples, qualquer lista  
Binária  $\rightarrow O(\log n)$   $\rightarrow$  Lista ordenada obrigatória

### ORDENAÇÃO:

Bubble/Selection  $\rightarrow O(n^2)$   $\rightarrow$  Só para estudar  
Insertion  $\rightarrow O(n^2)$   $\rightarrow$  Bom para listas pequenas  
Merge  $\rightarrow O(n \log n)$   $\rightarrow$  Estável, sempre eficiente  
Quick  $\rightarrow O(n \log n)^*$   $\rightarrow$  Rápido na prática

### ÁRVORES:

DFS  $\rightarrow$  Profundidade primeiro  $\rightarrow$  Recursivo  
BFS  $\rightarrow$  Largura primeiro  $\rightarrow$  Fila

### OTIMIZAÇÃO:

Programação Dinâmica  $\rightarrow$  Subproblemas sobrepostos  
Guloso  $\rightarrow$  Escolhas localmente ótimas  
Dividir e Conquistar  $\rightarrow$  Quebrar problema

## Estratégias de Resolução de Problemas

### Metodologia RICE:

#### R - Read (Ler)

- Leia o problema 2-3 vezes
- Identifique entrada e saída
- Procure por palavras-chave (ordenado, único, etc.)

#### I - Identify (Identificar)

- Que tipo de problema é? (busca, ordenação, otimização...)
- Há restrições de tempo/espaço?
- Casos especiais ou edge cases?

## C - Code (Codificar)

- Comece com força bruta
- Otimize depois se necessário
- Teste com exemplos pequenos

## E - Evaluate (Avaliar)

- Analise complexidade
- Teste edge cases
- Refatore se possível

## Padrões Comuns de Problemas:

### 1. Problemas de Busca:

```
# Sinais: "encontrar", "buscar", "existe"
# Ferramentas: busca linear, binária, hash

# Exemplo: Buscar elemento em lista ordenada
def buscar(lista, x):
    # O(log n) com busca binária
    esq, dir = 0, len(lista) - 1
    while esq <= dir:
        meio = (esq + dir) // 2
        if lista[meio] == x: return meio
        elif lista[meio] < x: esq = meio + 1
        else: dir = meio - 1
    return -1
```

### 2. Problemas de Contagem:

```
# Sinais: "quantos", "contar", "número de"
# Ferramentas: loops, recursão, DP

# Exemplo: Contar caminhos em grade
def contar_caminhos(m, n):
    # DP: O(mxn)
    dp = [[1]*n for _ in range(m)]
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
    return dp[m-1][n-1]
```

### 3. Problemas de Otimização:

```
# Sinais: "máximo", "mínimo", "melhor", "ótimo"
# Ferramentas: DP, guloso, força bruta

# Exemplo: Maior soma de subarray
def maior_soma_subarray(arr):
    # Algoritmo de Kadane: O(n)
```

```
max_atual = max_global = arr[0]
for i in range(1, len(arr)):
    max_atual = max(arr[i], max_atual + arr[i])
    max_global = max(max_global, max_atual)
return max_global
```

## Dicas para Entrevistas:

### Comunicação:

- Pense em voz alta
- Explique sua abordagem antes de codificar
- Pergunte sobre edge cases
- Discuta trade-offs

### ⌚ Gestão de Tempo:

⌚ 45 minutos típicos:  
5 min → Entender problema  
10 min → Planejar solução  
20 min → Implementar  
5 min → Testar e otimizar  
5 min → Discussão final

### Progressão Típica:

1. Força bruta → Funciona mas é lento
2. Identificar gargalos → O que está lento?
3. Otimizar → Usar estruturas melhores
4. Polir → Edge cases e clareza

## Bibliografia e Recursos Adicionais

### Livros Recomendados:

- "Introduction to Algorithms" - Cormen, Leiserson, Rivest, Stein
- "Algorithms" - Robert Sedgewick
- "Algorithm Design" - Jon Kleinberg, Éva Tardos

### Recursos Online:

- LeetCode: Prática de algoritmos
- HackerRank: Desafios de programação
- Coursera/edX: Cursos de algoritmos

### Visualizadores:

- VisuAlgo: Visualização de algoritmos
- Algorithm Visualizer: Animações interativas

# CAPÍTULO 10

## ALGORITMOS DE PROGRAMAÇÃO DINÂMICA

### 10.1 Conceito de Programação Dinâmica

#### Macete: Subproblemas + Memoização

```
# Fibonacci Ingênuo: O(2^n) - MUITO LENTO!
def fib_ingenuo(n):
    if n <= 1:
        return n
    return fib_ingenuo(n-1) + fib_ingenuo(n-2)

# Fibonacci com DP: O(n) - RÁPIDO!
def fib_dp(n):
    if n <= 1:
        return n

    dp = [0] * (n + 1)
    dp[0], dp[1] = 0, 1

    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]

# Macete: Evita recalcular subproblemas!
```

### 10.2 Problema da Mochila (0/1 Knapsack)

#### Formulação

```
def mochila_01(pesos, valores, capacidade):
    n = len(pesos)
    # dp[i][w] = valor máximo com i itens e capacidade w
    dp = [[0 for _ in range(capacidade + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, capacidade + 1):
            # Não pega o item i-1
            dp[i][w] = dp[i-1][w]

            # Se cabe, tenta pegar o item i-1
            if pesos[i-1] <= w:
                pegar = valores[i-1] + dp[i-1][w - pesos[i-1]]
                dp[i][w] = max(dp[i][w], pegar)
```

```
    return dp[n][capacidade]

# Complexidade: O(n × W) onde W = capacidade
# Espaço: O(n × W)
```

## 10.3 Maior Subsequência Comum (LCS)

### Macete: Comparar Caractere por Caractere

```
def lcs(texto1, texto2):
    m, n = len(texto1), len(texto2)
    # dp[i][j] = LCS de texto1[0:i] e texto2[0:j]
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if texto1[i-1] == texto2[j-1]:
                # Caracteres iguais: soma 1
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                # Caracteres diferentes: pega o melhor
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[m][n]

# Exemplo: LCS("ABCDGH", "AEDFHR") = "ADH" (tamanho 3)
# Complexidade: O(m × n)
```

# CAPÍTULO 11

## ALGORITMOS DE BUSCA E HASHING

### 11.1 Busca Linear vs Binária

#### Busca Linear - $O(n)$

```
def busca_linear(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1

# Macete: Funciona em qualquer array
# Não precisa estar ordenado
```

#### Busca Binária - $O(\log n)$

```
def busca_binaria(arr, x):
    esq, dir = 0, len(arr) - 1

    while esq <= dir:
        meio = (esq + dir) // 2

        if arr[meio] == x:
            return meio
        elif arr[meio] < x:
            esq = meio + 1 # Procura na direita
        else:
            dir = meio - 1 # Procura na esquerda

    return -1

# REQUISITO: Array deve estar ORDENADO!
# Macete: Divide pela metade a cada iteração
```

### 11.2 Tabelas Hash

#### Conceito: Acesso $O(1)$

```
class TabelaHash:
    def __init__(self, tamanho=10):
        self.tamanho = tamanho
        self.tabela = [[] for _ in range(tamanho)] # Lista de listas
```

```
def _hash(self, chave):
    # Função hash simples
    return hash(chave) % self.tamanho

def inserir(self, chave, valor):
    indice = self._hash(chave)
    bucket = self.tabela[indice]

    # Atualiza se já existe
    for i, (k, v) in enumerate(bucket):
        if k == chave:
            bucket[i] = (chave, valor)
            return

    # Adiciona novo
    bucket.append((chave, valor))

def buscar(self, chave):
    indice = self._hash(chave)
    bucket = self.tabela[indice]

    for k, v in bucket:
        if k == chave:
            return v

    return None # Não encontrado

# Complexidade:
# Melhor caso: O(1) - sem colisões
# Pior caso: O(n) - todas as chaves no mesmo bucket
```

# CAPÍTULO 12

## ALGORITMOS GULOSOS E DIVISÃO E CONQUISTA

### 12.1 Algoritmos Gulosos (Greedy)

#### Conceito: Escolha Localmente Ótima

```
# Problema da Troca: Dar troco com menor número de moedas
def troco_guloso(valor, moedas=[100, 50, 25, 10, 5, 1]):
    resultado = []

    for moeda in moedas:
        while valor >= moeda:
            resultado.append(moeda)
            valor -= moeda

    return resultado

# Exemplo: troco_guloso(189) = [100, 50, 25, 10, 1, 1, 1]
# Funciona para sistema monetário brasileiro!
```

#### Problema da Mochila Fracionária

```
def mochila_fracionaria(itens, capacidade):
    # itens = [(peso, valor), ...]
    # Ordena por valor/peso (densidade de valor)
    itens.sort(key=lambda x: x[1]/x[0], reverse=True)

    valor_total = 0
    for peso, valor in itens:
        if capacidade >= peso:
            # Pega item inteiro
            capacidade -= peso
            valor_total += valor
        else:
            # Pega fração do item
            fracao = capacidade / peso
            valor_total += fracao * valor
            break

    return valor_total

# Macete: Sempre pega item com melhor custo-benefício
```

### 12.2 Divisão e Conquista

## Template Geral

```
def divisao_conquista(problema):
    # Caso base
    if problema_pequeno(problema):
        return resolucao_direta(problema)

    # Divisão
    subproblemas = dividir(problema)

    # Conquista (recursão)
    resultados = []
    for sub in subproblemas:
        resultados.append(divisao_conquista(sub))

    # Combinação
    return combinar(resultados)
```

# APÊNDICES

## APÊNDICE A - TABELA MESTRE DE COMPLEXIDADES

### 🎯 SEU GUIA DE REFERÊNCIA RÁPIDA

Cole esta página na parede! Use sempre que precisar escolher algoritmos ou estruturas.

### 🚀 RANKING DE VELOCIDADE - De Flash a Tartaruga

🏆 Ranking	✍️ Complexidade	🏷️ Nome	💡 Exemplo Real	n=1K	n=1M
🥇 FLASH	O(1)	Constante	Abrir gaveta marcada	1	1
🥈 SONIC	O(log n)	Logarítmica	Buscar no dicionário	10	20
🥉 CARRO	O(n)	Linear	Ler livro página por página	1K	1M
🏅 TREM	O(n log n)	Linearítmica	Organizar cartas eficientemente	10K	20M
⚠️ BICICLETA	O(n <sup>2</sup> )	Quadrática	Comparar todos com todos	1M	1T
좆️ LESMA	O(2 <sup>n</sup> )	Exponencial	Testar todas combinações	∞	∞
Ѡ PARADO	O(n!)	Fatorial	Todas as permutações	∞	∞

🎯 **REGRA DE OURO:** Se passar de O(n<sup>2</sup>), pare e repense! Talvez haja uma abordagem melhor.

### Ἑ ESTRUTURAS DE DADOS - Tabela de Performance

Estrutura	Acesso	Busca	Inserção	Remoção	Quando Usar
Array	$O(1)$ ⚡	$O(n)$ 🚫	$O(n)$ 🚫	$O(n)$ 🚫	Acesso frequente por índice
Lista Ligada	$O(n)$ 🚫	$O(n)$ 🚫	$O(1)$ ⚡	$O(1)$ ⚡	Inserção/remoção frequente
Pilha (Stack)	$O(1)$ ⚡	-	$O(1)$ ⚡	$O(1)$ ⚡	LIFO - último entra, primeiro sai
Fila (Queue)	$O(1)$ ⚡	-	$O(1)$ ⚡	$O(1)$ ⚡	FIFO - primeiro entra, primeiro sai
Hash Table	$O(1)^*$ ⚡	$O(1)^*$ ⚡	$O(1)^*$ ⚡	$O(1)^*$ ⚡	Busca super rápida
Árvore BST	$O(\log n)^*$ 🚫	$O(\log n)^*$ 🚫	$O(\log n)^*$ 🚫	$O(\log n)^*$ 🚫	Dados ordenados
Heap	$O(1)$ ⚡	$O(n)$ 🚫	$O(\log n)$ 🚫	$O(\log n)$ 🚫	Prioridades (min/max)

💡 Legenda: ⚡ = Muito Rápido | 🚫 = Rápido | 🚫 = Lento | \* = Caso médio

## 🎯 ALGORITMOS CLÁSSICOS - Performance Guide

### 🔍 ALGORITMOS DE BUSCA

🚫 Busca Linear:  $O(n)$

🚀 Busca Binária:  $O(\log n)$

⚡ Hash Search:  $O(1)$

💡 Escolha: Binária para dados ordenados

### 🔗 ALGORITMOS DE ORDENAÇÃO

🚫 Bubble Sort:  $O(n^2)$

🚀 Merge Sort:  $O(n \log n)$

⚡ Quick Sort:  $O(n \log n)^*$

💡 Escolha: Merge para garantia, Quick para velocidade

## 🎯 MACETE FINAL - "CHEAT SHEET" para Entrevistas

🔥 PERGUNTA CLÁSSICA: "Qual a complexidade do seu algoritmo?"

✅ RESPOSTA NINJA: "No melhor caso  $O(X)$ , caso médio  $O(Y)$ , pior caso  $O(Z)$ . Escolhi essa abordagem porque..."

 **PERGUNTAS QUE IMPRESSIONAM:**

- "Posso otimizar espaço trocando por tempo?"
- "E se os dados já estivessem parcialmente ordenados?"
- "Qual o trade-off entre memória e velocidade aqui?"

 **DECORAR SEMPRE:**

- Busca binária:  $O(\log n)$
- Ordenação eficiente:  $O(n \log n)$
- Hash table operations:  $O(1)$
- Tree operations:  $O(\log n)$

 **MISSÃO CUMPRIDA!**

Agora você tem as ferramentas para escolher sempre o algoritmo mais eficiente!

*Continue praticando e seja um(a) ninja dos algoritmos! *

## **APÊNDICE C - GLOSSÁRIO DE TERMOS**

**Algoritmo:** Sequência finita de instruções bem definidas para resolver um problema.

**Análise Amortizada:** Técnica para analisar o tempo total de uma sequência de operações.

**Big-O:** Notação matemática que descreve o comportamento assintótico de funções.

**Caso Base:** Condição de parada em algoritmos recursivos.

**Complexidade Espacial:** Quantidade de memória necessária para executar um algoritmo.

**Complexidade Temporal:** Tempo necessário para executar um algoritmo em função do tamanho da entrada.

**Divide e Conquista:** Estratégia que divide um problema em subproblemas menores.

**Estrutura de Dados:** Forma de organizar e armazenar dados para acesso e modificação eficientes.

**Heurística:** Técnica para encontrar soluções aproximadas quando métodos exatos são impraticáveis.

**Invariante de Loop:** Propriedade que permanece verdadeira durante todas as iterações de um loop.

**Memoização:** Técnica de otimização que armazena resultados de funções para evitar recálculos.

**Programação Dinâmica:** Método para resolver problemas complexos dividindo-os em subproblemas.

**Recursão:** Técnica onde uma função chama a si mesma para resolver subproblemas.

**Tail Recursion:** Tipo especial de recursão onde a chamada recursiva é a última operação.

---

## APÊNDICE B - TRUQUES E MACETES DE PROGRAMAÇÃO

### B.1 Bitwise Operations (Operações de Bit)

```
# Verificar se número é par
def eh_par(n):
    return (n & 1) == 0 # Mais rápido que n % 2

# Multiplicar/dividir por 2^k
def mult_por_2k(n, k):
    return n << k # n * 2^k

def div_por_2k(n, k):
    return n >> k # n // 2^k

# Trocar dois números sem variável auxiliar
def trocar_xor(a, b):
    a = a ^ b
    b = a ^ b
    a = a ^ b
    return a, b

# Contar bits setados (população de bits)
def contar_bits(n):
    count = 0
    while n:
        count += n & 1
        n >= 1
    return count

# Macete: bin(n).count('1') é mais simples!
```

### B.2 Truques com Strings

```
# Verificar se string é palíndromo
def palindromo(s):
    return s == s[::-1]

# Remover caracteres especiais
def limpar_string(s):
    return ''.join(c for c in s if c.isalnum())

# Converter para title case
def title_case(s):
    return ' '.join(word.capitalize() for word in s.split())

# Encontrar todas as permutações
from itertools import permutations
```

```
def todas_permutacoes(s):
    return [''.join(p) for p in permutations(s)]
```

### B.3 Truques com Listas

```
# Acharar lista aninhada
def acharar(lista):
    resultado = []
    for item in lista:
        if isinstance(item, list):
            resultado.extend(achatar(item))
        else:
            resultado.append(item)
    return resultado

# List comprehension para acharar um nível
def acharar_1nivel(lista):
    return [item for sublista in lista for item in sublista]

# Remover duplicatas mantendo ordem
def remover_duplicatas(lista):
    return list(dict.fromkeys(lista))

# Dividir lista em chunks
def chunks(lista, tamanho):
    return [lista[i:i+tamanho] for i in range(0, len(lista), tamanho)]
```

### B.4 Decoradores Úteis

```
import time
import functools

# Medir tempo de execução
def cronometro(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        inicio = time.time()
        resultado = func(*args, **kwargs)
        fim = time.time()
        print(f"{func.__name__} levou {fim - inicio:.4f}s")
        return resultado
    return wrapper

# Memoização simples
def memoize(func):
    cache = {}
    @functools.wraps(func)
    def wrapper(*args):
        if args in cache:
```

```
        return cache[args]
resultado = func(*args)
cache[args] = resultado
return resultado
return wrapper

# Uso:
@cronometro
@memoize
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

---

## APÊNDICE C - ESTRUTURAS DE DADOS ESPECIAIS

### C.1 Heap (Priority Queue)

```
import heapq

class MinHeap:
    def __init__(self):
        self.heap = []

    def push(self, item):
        heapq.heappush(self.heap, item)

    def pop(self):
        return heapq.heappop(self.heap)

    def peek(self):
        return self.heap[0] if self.heap else None

    def __len__(self):
        return len(self.heap)

# Para MaxHeap, use números negativos
class MaxHeap:
    def __init__(self):
        self.heap = []

    def push(self, item):
        heapq.heappush(self.heap, -item)

    def pop(self):
        return -heapq.heappop(self.heap)
```

### C.2 Trie (Árvore de Prefixos)

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
```

```

        node = node.children[char]
node.is_end_word = True

def search(self, word):
    node = self._find_node(word)
    return node is not None and node.is_end_word

def starts_with(self, prefix):
    return self._find_node(prefix) is not None

def _find_node(self, prefix):
    node = self.root
    for char in prefix:
        if char not in node.children:
            return None
        node = node.children[char]
    return node

# Uso: Autocompletar, corretor ortográfico

```

### C.3 Union-Find (Disjoint Set)

```

class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        # Path compression
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x != root_y:
            # Union by rank
            if self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            elif self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            else:
                self.parent[root_y] = root_x
                self.rank[root_x] += 1

    def connected(self, x, y):
        return self.find(x) == self.find(y)

```

```
# Uso: Detectar ciclos, componentes conectados
```

---

## APÊNDICE D - PADRÕES DE CÓDIGO COMUNS

### D.1 Two Pointers (Dois Ponteiros)

```
# Verificar se array tem soma alvo
def tem_soma_alvo(arr, alvo):
    arr.sort()
    esq, dir = 0, len(arr) - 1

    while esq < dir:
        soma_atual = arr[esq] + arr[dir]
        if soma_atual == alvo:
            return True
        elif soma_atual < alvo:
            esq += 1
        else:
            dir -= 1

    return False

# Remover duplicatas de array ordenado
def remover_duplicatas_ordenado(arr):
    if not arr:
        return 0

    i = 0
    for j in range(1, len(arr)):
        if arr[j] != arr[i]:
            i += 1
            arr[i] = arr[j]

    return i + 1 # Novo comprimento
```

### D.2 Sliding Window (Janela Deslizante)

```
# Maior substring sem caracteres repetidos
def maior_substring_unica(s):
    char_map = {}
    esq = 0
    max_len = 0

    for dir in range(len(s)):
        if s[dir] in char_map and char_map[s[dir]] >= esq:
            esq = char_map[s[dir]] + 1

        char_map[s[dir]] = dir
        max_len = max(max_len, dir - esq + 1)

    return max_len
```

```

# Soma máxima de subarray de tamanho k
def soma_maxima_janela(arr, k):
    if len(arr) < k:
        return -1

    # Primeira janela
    soma_janela = sum(arr[:k])
    soma_maxima = soma_janela

    # Deslizar janela
    for i in range(k, len(arr)):
        soma_janela = soma_janela - arr[i-k] + arr[i]
        soma_maxima = max(soma_maxima, soma_janela)

    return soma_maxima

```

### D.3 Fast & Slow Pointers

```

# Detectar ciclo em lista ligada
def tem_ciclo(head):
    if not head or not head.next:
        return False

    lento = head
    rapido = head.next

    while rapido and rapido.next:
        if lento == rapido:
            return True
        lento = lento.next
        rapido = rapido.next.next

    return False

# Encontrar meio da lista ligada
def encontrar_meio(head):
    lento = rapido = head

    while rapido and rapido.next:
        lento = lento.next
        rapido = rapido.next.next

    return lento

```

## APÊNDICE E - PROBLEMAS CLÁSSICOS DE ENTREVISTA

### E.1 Array e String

```
# 1. Rotacionar array k posições
def rotacionar_array(nums, k):
    n = len(nums)
    k = k % n
    # Reverter todo array, depois reverter partes
    nums.reverse()
    nums[:k] = nums[:k][::-1]
    nums[k:] = nums[k:][::-1]

# 2. Produto de array exceto self
def produto_exceto_self(nums):
    n = len(nums)
    resultado = [1] * n

    # Produto à esquerda
    for i in range(1, n):
        resultado[i] = resultado[i-1] * nums[i-1]

    # Produto à direita
    direita = 1
    for i in range(n-1, -1, -1):
        resultado[i] *= direita
        direita *= nums[i]

    return resultado

# 3. Maior subarray (Kadane's Algorithm)
def maior_subarray(nums):
    max_atual = max_global = nums[0]

    for i in range(1, len(nums)):
        max_atual = max(nums[i], max_atual + nums[i])
        max_global = max(max_global, max_atual)

    return max_global
```

### E.2 Árvores

```
# 1. Validar BST
def validar_bst(root, min_val=float('-inf'), max_val=float('inf')):
    if not root:
        return True

    if root.val <= min_val or root.val >= max_val:
        return False
```

```

    return (validar_bst(root.left, min_val, root.val) and
            validar_bst(root.right, root.val, max_val))

# 2. Árvore balanceada
def eh_balanceada(root):
    def altura(node):
        if not node:
            return 0

        altura_esq = altura(node.left)
        if altura_esq == -1:
            return -1

        altura_dir = altura(node.right)
        if altura_dir == -1:
            return -1

        if abs(altura_esq - altura_dir) > 1:
            return -1

        return max(altura_esq, altura_dir) + 1

    return altura(root) != -1

# 3. Serialize/Deserialize árvore binária
class Codec:
    def serialize(self, root):
        def preorder(node):
            if node:
                vals.append(str(node.val))
                preorder(node.left)
                preorder(node.right)
            else:
                vals.append("#")

        vals = []
        preorder(root)
        return ",".join(vals)

    def deserialize(self, data):
        def build():
            val = next(vals)
            if val == "#":
                return None

            node = TreeNode(int(val))
            node.left = build()
            node.right = build()
            return node

        vals = iter(data.split(","))
        return build()

```

```
vals = iter(data.split(","))
return build()
```

---

## **APÊNDICE F - EXERCÍCIOS RESOLVIDOS (TEÓRICOS)**

### **F.1 Questões sobre Modularização e Funções**

**Questão 1:** A modularização de algoritmos é importante para organizar melhor o código, facilitar a manutenção, entre outras coisas. Sobre funções e procedimentos, assinale a alternativa correta sobre a modularização:

- a) O procedimento sempre retorna um valor ao programa.
- b) A função retorna um valor ao programa.
- c) As variáveis definidas no escopo de cada função são acessíveis em todo o programa.
- d) As variáveis locais são declaradas no escopo do programa inteiro.
- e) Variáveis globais não são acessíveis no corpo de uma função.

**Resposta: b) A função retorna um valor ao programa.**

**Explicação:** Funções são subprogramas que sempre retornam um valor, enquanto procedimentos executam ações mas não retornam valores. Variáveis locais têm escopo limitado à função onde foram declaradas.

---

**Questão 2:** Do ponto de vista de projeção de algoritmos, quais são as questões mais importantes a serem consideradas na escolha de um algoritmo?

- a) Corretude, eficiência, robustez e reusabilidade
- b) Corretude, eficiência, robustez e recursividade
- c) Corretude, eficiência, robustez e versatilidade
- d) Corretude, independência, robustez e autenticidade
- e) Corretude, independência, robustez e recursividade

**Resposta: a) Corretude, eficiência, robustez e reusabilidade**

**Explicação:** Os pilares fundamentais na escolha de algoritmos são:

- **Corretude:** O algoritmo deve resolver o problema corretamente
- **Eficiência:** Deve ter boa performance (tempo e espaço)
- **Robustez:** Capaz de lidar com entradas inesperadas
- **Reusabilidade:** Pode ser aplicado em diferentes contextos

---

### **F.2 Questões sobre Complexidade de Algoritmos**

**Questão 3:** Qual das seguintes afirmações sobre complexidade de algoritmos está correta?

- a) A complexidade de um algoritmo é sempre medida em termos de tempo de execução.
- b) A complexidade de um algoritmo nunca leva em consideração o espaço de memória utilizado.
- c) A complexidade de um algoritmo pode ser representada pela notação "O(n)" para denotar seu pior caso.
- d) A complexidade de um algoritmo no melhor caso é sempre pior do que no pior caso.
- e) A complexidade de um algoritmo não depende da entrada que ele processa.

**Resposta: c) A complexidade de um algoritmo pode ser representada pela notação "O(n)" para denotar seu pior caso.**

**Explicação:** A notação Big O representa o limite superior da complexidade (pior caso). A complexidade pode ser temporal ou espacial, e sempre depende da entrada.

---

**Questão 4:** O que significa uma complexidade O(1) em termos de tempo de execução de um algoritmo?

- a) O tempo de execução do algoritmo é diretamente proporcional ao tamanho da entrada.
- b) O tempo de execução do algoritmo aumenta exponencialmente à medida que o tamanho da entrada aumenta.
- c) O tempo de execução do algoritmo permanece constante, independentemente do tamanho de entrada.
- d) O tempo de execução do algoritmo é impossível de determinar.
- e) O tempo de execução do algoritmo é igual a zero.

**Resposta:** c) O tempo de execução do algoritmo permanece constante, independentemente do tamanho de entrada.

**Explicação:** O(1) significa complexidade constante - o tempo não varia com o tamanho da entrada. Exemplo: acessar um elemento específico de um array.

---

**Questão 5:** Com relação ao algoritmo abaixo, calcule a complexidade Big O, no pior caso:

```
(1) para i de 1 até n faça  
(2)   para j de 0 até n-1 faça  
(3)     a = a*(i+j)
```

- a) O(n)
- b) O( $n^2$ )
- c) O(1)
- d) O( $n^3$ )
- e) O( $n \log n$ )

**Resposta:** b) O( $n^2$ )

**Explicação:** Temos dois loops aninhados, cada um executando n vezes. O loop externo executa n vezes, e para cada execução, o loop interno executa n vezes. Total:  $n \times n = n^2$  operações.

---

**Questão 6:** Qual a complexidade do algoritmo a seguir?

```
bool localizar(int vetor[10], int valor) {  
    int tamanho = 10;  
    for (int i = 0; i < tamanho; i++) {  
        if(vetor[i] == valor)  
            return true;  
    }  
    return false;  
}
```

- a) O(n)
- b) O( $\log n$ )
- c) O( $n^2$ )
- d) O( $n \log n$ )
- e) O( $n^3$ )

**Resposta:** a) O(n)

**Explicação:** O algoritmo realiza uma busca linear. No pior caso, precisa verificar todos os elementos do vetor. Se o tamanho fosse n (ao invés de 10), seriam n operações.

---

**Questão 7:** Suponha que um algoritmo, sendo executado com uma entrada de tamanho n, leve exatos  $5n^2 + 10n + 200$  instruções de máquina. Qual a complexidade de pior caso desse algoritmo, considerando a Notação O (Big Oh)?

- a) O(n)
- b) O( $n^2$ )
- c) O(1)
- d) O( $n^3$ )
- e) O( $n \log n$ )

**Resposta:** b) O( $n^2$ )

**Explicação:** Na notação Big O, consideramos apenas o termo de maior ordem. Em  $5n^2 + 10n + 200$ , o termo dominante é  $5n^2$ , que simplifica para O( $n^2$ ).

---

**Questão 8:** A complexidade de algoritmos considera o tempo de execução que um código usa para solucionar um problema. Selecione a alternativa que mostra a notação da menor complexidade entre as seguintes: Ordem quadrática; Ordem cúbica; Ordem logarítmica; Ordem linear; Ordem exponencial

- a)  $O(n^2)$
- b)  $O(n^3)$
- c)  $O(n)$
- d)  $O(c^n)$
- e)  $O(\log n)$

**Resposta: e)  $O(\log n)$**

**Explicação:** Em ordem crescente de complexidade:  $O(\log n) < O(n) < O(n^2) < O(n^3) < O(c^n)$ . A complexidade logarítmica é a menor entre as listadas.

---

### F.3 Questões sobre Ponteiros

**Questão 9:** Em relação aos ponteiros nas linguagens de programação, selecione a opção que justifica sua aplicação:

- a) Flexibilidade de endereçamento e controle do gerenciamento de armazenamento dinâmico.
- b) Aumento da legibilidade dos programas.
- c) Facilidade de implementação no gerenciamento dinâmico.
- d) Dificuldade na implementação de tipos primitivos.
- e) Código fica mais legível e menos propenso a erros.

**Resposta: a) Flexibilidade de endereçamento e controle do gerenciamento de armazenamento dinâmico.**

**Explicação:** Ponteiros permitem acesso direto à memória, possibilitam alocação dinâmica e oferecem flexibilidade no gerenciamento de dados.

---

**Questão 10:** Marque a alternativa correta sobre ponteiros:

- a) Ponteiro é uma variável cujo conteúdo é um endereço de memória.
- b) Ponteiro é uma variável cujo conteúdo é um valor de variável.
- c) Ponteiros é um tipo de dado do tipo float que consegue armazenar outros tipos de dados.
- d) Ponteiros é um tipo de dado do tipo int que consegue armazenar outros tipos de dados.
- e) Todas as alternativas estão corretas.

**Resposta: a) Ponteiro é uma variável cujo conteúdo é um endereço de memória.**

**Explicação:** Por definição, um ponteiro armazena o endereço de memória onde outro dado está localizado, não o valor em si.

---

### F.4 Questões sobre Recursividade

**Questão 11:** Qual é o conceito fundamental por trás da recursividade em algoritmos?

- a) Repetição de uma operação em um loop.
- b) Dividir um problema em subproblemas semelhantes menores.
- c) Utilizar funções matemáticas.
- d) Organizar dados em uma pilha.
- e) Multiplicação de números inteiros.

**Resposta: b) Dividir um problema em subproblemas semelhantes menores.**

**Explicação:** Recursividade baseia-se no princípio "divide e conquista", onde um problema é decomposto em versões menores do mesmo problema.

---

**Questão 12:** O que é necessário para que uma função recursiva não entre em um loop infinito?

- a) Ela deve sempre conter uma instrução "while".
- b) Ela deve ser chamada com um valor negativo.
- c) Ela deve conter uma condição de parada.
- d) Ela deve chamar outra função recursiva.
- e) Ela deve ser executada apenas uma vez.

**Resposta:** c) Ela deve conter uma condição de parada.

**Explicação:** O caso base (condição de parada) é essencial para interromper as chamadas recursivas e evitar loops infinitos.

---

**Questão 13:** Considere a seguinte função recursiva em Python para calcular o fatorial:

```
def fatorial(n):
    if n == 0:
        return 1
    else:
        return n * fatorial(n - 1)
```

Qual é o valor de fatorial(4)?

- a) 4 b) 6 c) 12 d) 24 e) 120

**Resposta:** d) 24

**Explicação:**

- $fatorial(4) = 4 \times fatorial(3)$
- $fatorial(3) = 3 \times fatorial(2)$
- $fatorial(2) = 2 \times fatorial(1)$
- $fatorial(1) = 1 \times fatorial(0)$
- $fatorial(0) = 1$
- Resultado:  $4 \times 3 \times 2 \times 1 = 24$

---

**Questão 14:** Existem casos em que um procedimento ou função chama a si próprio. Sobre introdução à computação, é correto afirmar que:

- a) quando um procedimento ou função chama a si próprio, denomina-se passagem de parâmetro por referência.
- b) quando um procedimento ou função chama a si próprio, denomina-se passagem de parâmetro por valor.
- c) quando um procedimento ou função chama a si próprio, denomina-se recursividade.
- d) quando um procedimento ou função chama a si próprio, denomina-se passagem de parâmetro por variável.
- e) quando um procedimento ou função chama a si próprio, denomina-se passagem de parâmetro por recursão.

**Resposta:** c) quando um procedimento ou função chama a si próprio, denomina-se recursividade.

**Explicação:** Por definição, recursividade é quando uma função chama a si mesma direta ou indiretamente.

---

**Questão 15:** Sobre funções recursivas, analise as afirmativas:

- I. Toda função recursiva precisa de um caso base para evitar chamadas infinitas.
- II. O uso de recursividade pode levar a consumo elevado de memória devido à pilha de chamadas.
- III. Recursividade é sempre mais eficiente que a versão iterativa.
- IV. Funções recursivas podem ser reescritas como funções iterativas.
- V. Um algoritmo recursivo sempre executa mais rapidamente que um iterativo.

Quais são as alternativas corretas?

**Resposta: I, II e IV estão corretas.**

**Explicação:**

- I: ✓ Caso base é obrigatório
  - II: ✓ Cada chamada usa memória da pilha
  - III: X Nem sempre é mais eficiente
  - IV: ✓ Qualquer recursão pode ser convertida para iteração
  - V: X Frequentemente recursão é mais lenta
- 

## F.5 Questões sobre Estruturas de Dados

**Questão 16:** Estrutura de dados caracterizada por: Ou não ter elemento algum (árvore vazia); Ou tem um elemento distinto denominado raiz, com dois ponteiros para duas estruturas diferentes, denominadas subárvore esquerda e subárvore direita. Essa estrutura é chamada de:

- a) Trevo Binário b) Nô Folha c) Fila d) Árvore Binária e) Folha Binária

**Resposta: d) Árvore Binária**

**Explicação:** A definição descreve perfeitamente uma árvore binária: estrutura hierárquica com no máximo dois filhos por nó.

---

**Questão 17:** Considerando uma árvore de pesquisa binária com  $N$  nós, qual é a complexidade da inserção em uma árvore de pesquisa binária balanceada?

- a)  $O(1)$  b)  $O(\log N)$  c)  $O(N)$  d)  $O(N \log N)$  e)  $O(N^2)$

**Resposta: b)  $O(\log N)$**

**Explicação:** Em uma árvore balanceada, a altura é  $\log N$ , e a inserção requer percorrer da raiz até uma folha, resultando em  $O(\log N)$ .

---

## F.6 Questões sobre Algoritmos de Busca

**Questão 18:** A busca sequencial e a busca binária são dois algoritmos para pesquisa. Diante do cenário, quais alternativas são corretas?

- a) O tempo de execução da busca binária é menor do que o da busca sequencial.
- b) A busca sequencial é uma solução mais eficiente que a busca binária.
- c) A busca sequencial é um algoritmo simples de implementar, mas não é muito eficiente.
- d) A taxa de crescimento de  $\log(n)$  é maior do que  $n$ .

**Resposta: a) e c) estão corretas.**

**Explicação:**

- a) ✓ Busca binária:  $O(\log n)$  vs Sequencial:  $O(n)$
  - b) X Busca binária é mais eficiente
  - c) ✓ Sequencial é simples mas  $O(n)$
  - d) X  $\log(n)$  cresce mais lentamente que  $n$
- 

## F.7 Questões sobre Algoritmos de Ordenação

**Questão 19:** A ordenação é uma operação comum em muitas aplicações. Sobre alguns algoritmos de ordenação, é correto afirmar:

- a) O quicksort particiona os itens em dois segmentos separados por um elemento pivô e ordena-os recursivamente.
- b) O selection sort divide os itens em dois segmentos, ordena-os individualmente e depois mescla-os.
- c) O insertion sort troca dois elementos adjacentes se estiverem fora de ordem.
- d) O bubble sort busca um elemento fora de ordem em elementos sucessivos.
- e) O bubble sort é baseado em passar sempre o menor valor para a primeira posição.

**Resposta:** a) O quicksort particiona os itens em dois segmentos separados por um elemento pivô e ordena-os recursivamente.

**Explicação:** Esta é a descrição correta do QuickSort. As outras alternativas confundem as características dos algoritmos.

---

## F.8 Questões sobre Desenvolvimento de Algoritmos

**Questão 20:** (ENADE) Avalie se, no contexto da lógica de programação, as etapas para o desenvolvimento de um programa estão corretamente descritas:

- I. Estuda-se o enunciado do problema para definir os dados de entrada, o processamento e os dados de saída.
  - II. Usa-se fluxogramas ou português estruturado para descrever o problema com suas soluções.
  - III. O algoritmo é transformado em códigos da linguagem de programação escolhida.
- a) I, II e III b) I e III, apenas c) II e III, apenas d) I e II, apenas e) I, apenas

**Resposta:** a) I, II e III

**Explicação:** Todas as etapas estão corretas e representam o processo completo de desenvolvimento: análise → projeto → implementação.

---

## F.9 Questões Adicionais (Nível Fácil)

**Questão 21:** Quantas vezes posso chamar a mesma função?

- a) 1 b) 4 c) nenhuma d) quantas quiser e) 3

**Resposta:** d) quantas quiser

**Explicação:** Não há limite para o número de chamadas de uma função, exceto limitações de memória do sistema.

**Questão 22:** Em programação, qual é a principal diferença entre recursividade e iteração?

- a) A recursividade usa um contador para executar repetições.
- b) A recursividade utiliza estruturas de laço como for e while.
- c) A recursividade é uma técnica onde uma função chama a si mesma até atingir um caso base, enquanto a iteração usa estruturas de laço.
- d) A iteração ocorre apenas em linguagens que suportam estruturas de laço.
- e) A recursividade é sempre mais eficiente que a iteração.

**Resposta:** c) A recursividade é uma técnica onde uma função chama a si mesma até atingir um caso base, enquanto a iteração usa estruturas de laço.

**Explicação:** Esta é a diferença fundamental: recursão usa chamadas de função, iteração usa loops.

**Questão 23:** Qual das seguintes afirmações sobre recursividade está correta?

- a) Funções recursivas são mais rápidas que funções iterativas em qualquer caso.
- b) Toda função recursiva deve ter pelo menos dois casos base.
- c) Uma função recursiva chama a si mesma até atingir um caso base.
- d) Funções recursivas não podem usar estruturas de dados como pilhas.
- e) Recursividade sempre leva a um aumento de consumo de memória.

**Resposta: c) Uma função recursiva chama a si mesma até atingir um caso base.**

**Explicação:** Esta é a definição correta de recursividade. O caso base é o que interrompe as chamadas recursivas.

---

## F.10 Resumo dos Conceitos-Chave

**Principais tópicos abordados nos exercícios:**

1. **Modularização:** Funções vs Procedimentos, escopo de variáveis
2. **Complexidade:** Big O, análise de loops, casos de complexidade
3. **Ponteiros:** Definição, uso, vantagens
4. **Recursividade:** Caso base, pilha de chamadas, comparação com iteração
5. **Estruturas de Dados:** Árvores binárias, operações básicas
6. **Algoritmos de Busca:** Linear vs Binária, complexidades
7. **Algoritmos de Ordenação:** Características dos principais algoritmos
8. **Desenvolvimento:** Etapas de criação de programas

**Dicas para resolver questões similares:**

- Sempre identifique o conceito principal sendo testado
  - Para complexidade, conte os loops aninhados
  - Para recursividade, trace a execução passo a passo
  - Para estruturas de dados, visualize a organização dos elementos
-

## APÊNDICE G - BIBLIOGRAFIA E REFERÊNCIAS

### Bibliografia Básica

1. **CORMEN, Thomas H. et al.** *Introduction to Algorithms*, 4th Edition. MIT Press, 2022.
2. **SEGEWICK, Robert; WAYNE, Kevin.** *Algorithms*, 4th Edition. Addison-Wesley, 2011.
3. **KLEINBERG, Jon; TARDOS, Éva.** *Algorithm Design*. Pearson, 2005.

### Bibliografia Complementar

4. **AHO, Alfred V. et al.** *Data Structures and Algorithms*. Addison-Wesley, 1983.
5. **SKIENA, Steven S.** *The Algorithm Design Manual*, 3rd Edition. Springer, 2020.
6. **DASGUPTA, Sanjoy et al.** *Algorithms*. McGraw-Hill, 2008.

### Recursos Online

- **LeetCode:** <https://leetcode.com/> - Prática de algoritmos
- **HackerRank:** <https://www.hackerrank.com/> - Desafios de programação
- **GeeksforGeeks:** <https://www.geeksforgeeks.org/> - Tutoriais e exemplos
- **VisuAlgo:** <https://visualgo.net/> - Visualização de algoritmos
- **Algorithm Visualizer:** <https://algorithm-visualizer.org/> - Animações interativas

### Artigos Científicos Relevantes

- Knuth, D. E. (1976). "Big Omicron and big Omega and big Theta". *SIAGCT News*, 8(2), 18-24.
  - Tarjan, R. E. (1985). "Amortized computational complexity". *SIAM Journal on Algebraic Discrete Methods*, 6(2), 306-318.
-

## APÊNDICE D - EXERCÍCIOS ADICIONAIS

### Seção 1: Análise de Complexidade

**Exercício D.1:** Determine a complexidade dos seguintes algoritmos:

```
# Algoritmo A
def algoritmo_a(n):
    soma = 0
    for i in range(n):
        for j in range(i):
            soma += i * j
    return soma

# Algoritmo B
def algoritmo_b(n):
    if n <= 1:
        return 1
    return algoritmo_b(n//2) + algoritmo_b(n//2) + n
```

**Exercício D.2:** Calcule quantas operações básicas são executadas para  $n=16$ :

- Busca linear em array desordenado
- Busca binária em array ordenado
- Insertion sort

### Seção 2: Recursividade Avançada

**Exercício D.3:** Implemente a função `ackermann(m, n)` e analise sua complexidade.

**Exercício D.4:** Converta o seguinte algoritmo recursivo para iterativo:

```
def fibonacci_rec(n):
    if n <= 1:
        return n
    return fibonacci_rec(n-1) + fibonacci_rec(n-2)
```

### Seção 3: Problemas Práticos

**Exercício D.5: Problema da Moeda:** Dado um valor  $V$  e moedas de denominações  $[1, 5, 10, 25]$ , encontre o número mínimo de moedas necessárias.

**Exercício D.6: Torres de Hanói:** Implemente a solução recursiva e calcule o número de movimentos para  $n$  discos.

### Gabarito Resumido

- **D.1:** Algoritmo A:  $O(n^2)$ , Algoritmo B:  $O(n \log n)$
- **D.2:** Linear: 16 ops (pior caso), Binária: 4 ops, Insertion: 136 ops (pior caso)
- **D.3:** Ackermann tem crescimento mais que exponencial
- **D.4:** Usar loop com duas variáveis para  $O(n)$
- **D.5:** Usar programação dinâmica para  $O(V \times n)$
- **D.6:**  $2^n - 1$  movimentos,  $O(2^n)$  complexidade

**FIM DA APOSTILA**

---

© 2025 - Prof. Vagner Cordeiro  
*Material Didático - Algoritmos e Análise de Complexidade*