APOSTILA DE ALGORITMOS E ANÁLISE DE COMPLEXIDADE

Uma Abordagem Prática e Didática

Professor Engenheiro de Computação

Vagner Cordeiro

VERSÃO 1.0
Setembro de 2025

Material didático para estudo de Análise de Algoritmos e Estruturas de Dados

PREFÁCIO

Esta apostila foi desenvolvida com o objetivo de fornecer aos estudantes de Ciência da Computação e Engenharia de Software uma base sólida em análise de algoritmos e complexidade computacional. O material apresenta de forma didática e progressiva os conceitos fundamentais, desde a notação Big-O até técnicas avançadas de otimização.

Objetivos de Aprendizagem

Ao final do estudo desta apostila, o aluno será capaz de:

- **Analisar** a complexidade temporal e espacial de algoritmos
- **Aplicar** a notação Big-O em problemas reais
- **Compreender** e implementar algoritmos recursivos
- **Otimizar** soluções utilizando técnicas de programação dinâmica
- **Resolver** problemas de algoritmos de forma estruturada
- **Identificar** padrões algorítmicos em diferentes contextos

Metodologia

O material está estruturado de forma progressiva, começando com conceitos básicos e evoluindo para tópicos avançados. Cada capítulo inclui:

- Fundamentação teórica
- Exemplos práticos em Python e C
- Exercícios resolvidos
- Questões para fixação
- Aplicações reais

Sobre o Autor

Prof. Vagner Cordeiro é Professor Universitário do Curso de Graduação e Pós-Graduação em Sistemas de Informação na Faculdade Estácio de Florianópolis. Leciona diversas disciplinas como Análise de Algoritmos, Redes de Computadores, Segurança Cibernética, Tópicos de Big Data em Python, IoT e Indústria 4.0 em Python, e Pensamento Computacional. Atua também como Instrutor de Informática no Governo do Estado de SC pela SEJURI.

Possui formação em Tecnólogo em Análise e Desenvolvimento de Sistemas, Técnico em Telecomunicações, Engenharia de Computação, especializações em Análise de Dados, MBA em Segurança da Informação e Engenharia e Segurança do Trabalho. Também possui Licenciatura em Matemática.

Com mais de 15 anos de experiência em empresas de destaque no setor de tecnologia de Santa Catarina como Intelbras, Embratel, Digitro e startups, traz para o ensino uma perspectiva prática e atual do mercado de trabalho em tecnologia.

ÍNDICE

CAPÍTULO 1

INTRODUÇÃO À ANÁLISE DE ALGORITMOS

1.1 Conceitos Fundamentais

O que é um Algoritmo?

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

Características de um Bom Algoritmo:

- **Finitude**: Deve terminar após um número finito de passos
- **Definição**: Cada passo deve ser precisamente definido
- **Entrada**: Zero ou mais entradas
- **Saída**: Uma ou mais saídas
- **Efetividade**: Cada operação deve ser básica o suficiente para ser executada

Análise de Algoritmos

A análise de algoritmos é o processo de determinar a quantidade de recursos computacionais (tempo e espaço) que um algoritmo consome.

CAPÍTULO 2

COMPLEXIDADE DE TEMPO E ESPAÇO

2.1 Definições Básicas

Complexidade de Tempo

Mede o tempo de execução de um algoritmo em função do tamanho da entrada.

Complexidade de Espaço

Mede a quantidade de memória necessária para executar um algoritmo.

2.2 Casos de Análise

- **Melhor Caso**: Menor tempo possível para qualquer entrada de tamanho n
- **Caso Médio**: Tempo médio para todas as entradas possíveis de tamanho n
- **Pior Caso**: Maior tempo possível para qualquer entrada de tamanho n

---

CAPÍTULO 3

NOTAÇÃO BIG-O

3.1 Definição Formal

A notação Big-O descreve o comportamento assintótico de algoritmos, ou seja, **como o tempo de execução cresce em relação ao tamanho da entrada**.

Como Entender Big-O de Forma Simples

Imagine que você tem uma tarefa para fazer e precisa saber quanto tempo vai demorar:

- **O(1)**: Não importa quantos dados você tem, sempre demora o mesmo tempo
- **O(n)**: Se você tem 10 itens, demora X tempo. Se tem 100 itens, demora 10X tempo
- **O(n^2)**: Se você tem 10 itens, demora X tempo. Se tem 100 itens, demora 100X tempo!

Visualização do Crescimento

```
Para n = 10:
O(1)     = 1          | Excelente
O(log n) = 3          | Muito bom
O(n)     = 10         | Bom
O(n log n) = 33       | Aceitável
O(n²)    = 100        | Cuidado
O(2ⁿ)    = 1024       | Evitar
O(n!)    = 3,628,800  | Impraticável

Para n = 1000:
O(1)     = 1              | Ainda excelente
O(log n) = 10             | Ainda muito bom
O(n)     = 1,000          | Ainda bom
O(n log n) = 10,000       | Ainda aceitável
O(n²)    = 1,000,000      | Já problemático
O(2ⁿ)    = 10^301         | Impossível
```

Classes de Complexidade - Do Melhor ao Pior

Ranking	Notação	Nome	Exemplo Prático	Quando usar
1º	O(1)	**Constante**	Pegar item da geladeira	Acesso direto
2º	O(log n)	**Logarítmica**	Buscar palavra no dicionário	Busca inteligente
3º	O(n)	**Linear**	Ler um livro página por página	Verificar todos
4º	O(n log n)	**Linearítmica**	Organizar cartas de forma eficiente	Ordenação boa
5º	O(n^2)	**Quadrática**	Comparar todos com todos	Pequenas entradas

6º	$O(n^3)$	**Cúbica**	Três loops aninhados	Evitar
7º	$O(2^n)$	**Exponencial**	Testar todas combinações	Só para problemas pequenos
8º	O(n!)	**Fatorial**	Testar todas permutações	Praticamente impossível

Como Calcular Big-O - Passo a Passo

Passo 1: Identifique os loops

```
# Um loop = O(n)
for i in range(n):
    print(i)  # O(1)
# Total: O(n)

# Dois loops aninhados = O(n²)
for i in range(n):      # n vezes
    for j in range(n):  # n vezes para cada i
        print(i, j)     # O(1)
# Total: O(n²)
```

Passo 2: Some as complexidades

```
# Operações em sequência se somam
for i in range(n):      # O(n)
    print(i)

for j in range(n):      # O(n)
    print(j)

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

Passo 3: Aplique as regras de simplificação

Regras de Ouro para Big-O

1. **Constantes são ignoradas**:
    - O(2n) = O(n)
    - O(100) = O(1)
    - O(n/2) = O(n)
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):     |      |      |      |     (sempre igual)
O(log n): |     ||     |||    ||||   (cresce devagar)
O(n):     |   ||||||||||||  ||||...  (cresce linear)
O(n²):    |     ||||    ||||||||...  (cresce rápido)
O(2ⁿ):    |      XXX      XXXXXXXX   (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:

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

Solução:

```python
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?

```python
# 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):
```

```python
    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:

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

Solução Iterativa:

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

CAPÍTULO 5

ALGORITMOS DE ORDENAÇÃO

5.1 Algoritmos Básicos de Ordenação

Visão Geral dos Algoritmos de Ordenação

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Espaço	Estável	In-place
Bubble Sort	O(n)	O(n²)	O(n²)	O(1)	Sim	Sim
Selection Sort	O(n²)	O(n²)	O(n²)	O(1)	Não	Sim
Insertion Sort	O(n)	O(n²)	O(n²)	O(1)	Sim	Sim
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Sim	Não
Quick Sort	O(n log n)	O(n log n)	O(n²)	O(log n)	Não	Sim
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	Não	Sim

Bubble Sort

Conceito: Compara elementos adjacentes e os troca se estiverem na ordem errada.

Implementação Python:

```python
def bubble_sort(arr):
    n = len(arr)

    for i in range(n):
        # Flag para otimização: se não houve troca, array está ordenado
        trocou = False

        # Últimos i elementos já estão ordenados
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                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:

```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:

```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:

```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:

```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:

```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:])
```

```python
    # 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:

```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 - l) / 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 particiona o array de forma que elementos menores fiquem à esquerda e maiores à direita do pivô.

Implementação Python:

```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)
```

```python
        # 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:

```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:

```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:

```c
#include <stdio.h>

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

```c
    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):

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

```python
    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"  Comparação {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}")
            return meio
        elif arr[meio] < x:
            print(f"    {arr[meio]} < {x}, buscar à direita")
            esquerda = meio + 1
        else:
            print(f"    {arr[meio]} > {x}, buscar à esquerda")
            direita = meio - 1

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

Implementação Python (Recursiva):

```python
def busca_binaria_recursiva(arr, x, esquerda=0, direita=None):
    if direita is None:
        direita = len(arr) - 1

    # Caso base: elemento não encontrado
    if esquerda > direita:
        return -1
```

```python
    meio = (esquerda + direita) // 2

    # Caso base: elemento encontrado
    if arr[meio] == x:
        return meio

    # Busca recursiva
    if arr[meio] < x:
        return busca_binaria_recursiva(arr, x, meio + 1, direita)
    else:
        return busca_binaria_recursiva(arr, x, esquerda, meio - 1)
```

Implementação C (Iterativa):

```c
#include <stdio.h>

int busca_binaria_iterativa(int arr[], int n, int x) {
    int esquerda = 0, direita = n - 1;

    while (esquerda <= direita) {
        int meio = esquerda + (direita - esquerda) / 2;

        if (arr[meio] == x) {
            return meio;
        }

        if (arr[meio] < x) {
            esquerda = meio + 1;
        } else {
            direita = meio - 1;
        }
    }

    return -1;  // Não encontrado
}

int busca_binaria_debug(int arr[], int n, int x) {
    printf("Buscando %d em array ordenado\n", x);
    int esquerda = 0, direita = n - 1;
    int comparacoes = 0;

    while (esquerda <= direita) {
        int meio = esquerda + (direita - esquerda) / 2;
        comparacoes++;

        printf("  Comparação %d: esq=%d, dir=%d, meio=%d\n",
               comparacoes, esquerda, direita, meio);
        printf("    arr[%d] = %d\n", meio, arr[meio]);

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

```c
            printf("  Total de comparações: %d\n", comparacoes);
            return meio;
        }

        if (arr[meio] < x) {
            printf("    %d < %d, buscar à direita\n", arr[meio], x);
            esquerda = meio + 1;
        } else {
            printf("    %d > %d, buscar à esquerda\n", arr[meio], x);
            direita = meio - 1;
        }
    }

    printf("  Não encontrado após %d comparações\n", comparacoes);
    return -1;
}
```

Implementação C (Recursiva):

```c
#include <stdio.h>

int busca_binaria_recursiva(int arr[], int esquerda, int direita, int x) {
    if (direita >= esquerda) {
        int meio = esquerda + (direita - esquerda) / 2;

        // Elemento encontrado
        if (arr[meio] == x) {
            return meio;
        }

        // Se elemento é menor que meio, está na metade esquerda
        if (arr[meio] > x) {
            return busca_binaria_recursiva(arr, esquerda, meio - 1, x);
        }

        // Caso contrário, está na metade direita
        return busca_binaria_recursiva(arr, meio + 1, direita, x);
    }

    return -1;  // Elemento não encontrado
}

int main() {
    int arr[] = {2, 3, 4, 10, 40, 50, 60, 70};
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;

    // Teste busca binária iterativa com debug
    printf("=== Busca Binária Iterativa ===\n");
    int resultado1 = busca_binaria_debug(arr, n, x);
```

```c
    // Teste busca binária recursiva
    printf("\n=== Busca Binária Recursiva ===\n");
    int resultado2 = busca_binaria_recursiva(arr, 0, n - 1, x);

    if (resultado2 != -1) {
        printf("Elemento %d encontrado no índice %d (recursiva)\n", x, resultado2);
    } else {
        printf("Elemento %d não encontrado (recursiva)\n", x);
    }

    return 0;
}
```

Comparação entre Busca Linear e Binária

Análise de Complexidade:

Aspecto	Busca Linear	Busca Binária
Complexidade Tempo	O(n)	O(log n)
Complexidade Espaço	O(1)	O(1) iterativa, O(log n) recursiva
Pré-requisito	Nenhum	Array deve estar ordenado
Melhor para	Arrays pequenos ou não ordenados	Arrays grandes e ordenados

Exemplo de Performance:

```
Para um array de 1.000.000 elementos:

Busca Linear:
- Pior caso: 1.000.000 comparações
- Caso médio: 500.000 comparações

Busca Binária:
- Pior caso: 20 comparações (log₂ 1.000.000 ≈ 20)
- Caso médio: ~18 comparações

Diferença: 50.000x mais rápida no pior caso!
```

3.4 Exercícios de Fixação - Capítulo 3

Exercício 3.1: Análise Básica de Complexidade

Determine a complexidade Big-O dos seguintes códigos:

```python
# Código A
def codigo_a(n):
    count = 0
    for i in range(n):
        count += 1
```

```python
    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: Comparação 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) ≈ 20, $\log_2$(1000) ≈ 10, então 20/10 = 2x)

CAPÍTULO 4

RECURSIVIDADE

4.1 Conceitos Fundamentais

O que é Recursividade?

Recursividade é como ensinar alguém a subir escadas:

- **Regra simples**: "Para subir N degraus, suba 1 degrau e depois suba os N-1 restantes"
- **Regra de parada**: "Se não há mais degraus (N=0), você chegou!"

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

Os 3 Ingredientes Mágicos da Recursividade

1. Caso Base (Base Case)

```
A condição que PARA a recursão
Sem ele = Loop infinito = Crash!
```

2. Caso Recursivo (Recursive Case)

```
A função chama ela mesma com um problema MENOR
```

3. Progresso em Direção ao Caso Base

```
Cada chamada deve nos aproximar da parada
```

Receita Universal para Recursividade

```python
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 × 4!. Para calcular 4!, preciso de 4 × 3!..."

Definição Matemática:

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

Implementação Comentada:

```python
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...
Cada número = soma dos dois anteriores
```

Versão Simples (Ineficiente):

```python
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:

```python
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]
```

```python
    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(2^n) para O(n)!
```

Comparação de Performance:

```python
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:

```python
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
    """

    identacao = "  " * nivel  # Para visualizar a recursão

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

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

    # PASSO 1: Mover n-1 discos para auxiliar
    print(f"{identacao}  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"{identacao}  2. Depois: mover disco {n} de {origem} → {destino}")
    mov2 = 1

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

    total = mov1 + mov2 + mov3
    print(f"{identacao}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:

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

Versão Iterativa:

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

Versão em C - Recursiva:

```c
#include <stdio.h>

int fatorial_recursivo(int n) {
    if (n <= 1) {
        return 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:

```c
#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
- ✗ Usa mais memória (pilha)
- ✗ Risco de stack overflow

Iterativo:

- ✓ Mais eficiente em memória
- ✓ Mais rápido na execução
- ✗ Menos intuitivo
- ✗ 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:

```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

```python
# 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:

```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)

```python
# 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:

```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

```python
# 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:

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

bool eh_impar(int n);  // Declaração antecipada
```

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

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

int main() {
    int num = 7;
    printf("%d é %s\n", num, eh_par(num) ? "par" : "ímpar");
    return 0;
}
```

Técnicas de Otimização

1. Memoização - O Cache Inteligente

```python
# ❌ SEM memoização: O(2ⁿ)
def fib_lento(n):
    if n <= 1: return n
    return fib_lento(n-1) + fib_lento(n-2)

# ✅ COM memoização: O(n)
def fib_rapido(n, cache={}):
    if n in cache:
        return cache[n]

    if n <= 1:
        cache[n] = n
        return n

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

# Usando decorador do Python (ainda mais fácil):
from functools import lru_cache

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

2. Programação Dinâmica Bottom-Up

```
# Em vez de recursão, construa de baixo para cima:
def fib_bottom_up(n):
    if n <= 1: return n

    # Tabela para guardar resultados
    dp = [0] * (n + 1)
    dp[0], dp[1] = 0, 1

    # Construir do menor para o maior
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]

# Complexidade: O(n) tempo, O(n) espaço
# Vantagem: Sem risco de stack overflow
```

Recursividade em Estruturas de Dados

1. Soma de Elementos em Lista

```
def soma_lista(lista):
    # Caso base: lista vazia
    if not lista:
        return 0

    # Caso recursivo
    return lista[0] + soma_lista(lista[1:])

# Exemplo
print(soma_lista([1, 2, 3, 4, 5]))  # Output: 15
```

2. Busca em Lista

```
def busca_recursiva(lista, elemento, indice=0):
    # Caso base: elemento não encontrado
    if indice >= len(lista):
        return -1

    # Caso base: elemento encontrado
    if lista[indice] == elemento:
        return indice

    # Caso recursivo
    return busca_recursiva(lista, elemento, indice + 1)
```

3. Inversão de String

```
def inverter_string(s):
    # Caso base
    if len(s) <= 1:
        return s

    # Caso recursivo
    return s[-1] + inverter_string(s[:-1])

# Exemplo
print(inverter_string("hello"))  # Output: "olleh"
```

4. Contagem de Dígitos

```
def contar_digitos(n):
    # Caso base
    if n < 10:
        return 1

    # Caso recursivo
    return 1 + contar_digitos(n // 10)

# Exemplo
print(contar_digitos(12345))  # Output: 5
```

Recursividade vs Iteração

Quando Usar Recursividade:

✅ **Problemas que têm estrutura recursiva natural**

- Árvores e grafos
- Fractais
- Dividir e conquistar

✅ **Problemas que podem ser quebrados em subproblemas menores**

- Torres de Hanói
- Busca em profundidade

✅ **Quando a solução recursiva é mais clara e elegante**

Quando Evitar Recursividade:

❌ **Problemas com alta sobreposição de subproblemas** (sem memoização)

- Fibonacci ingênuo

❌ **Quando a profundidade pode ser muito grande**

- Risco de stack overflow

❌ **Problemas simples onde iteração é mais eficiente**

Comparação: Fatorial Recursivo vs Iterativo

Recursivo:

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

Iterativo:

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

Análise:

- **Recursivo**: Mais legível, mas usa mais memória
- **Iterativo**: Mais eficiente em memória, mas menos intuitivo

Tipos Especiais de Recursividade

1. Recursividade Linear

Cada chamada recursiva gera apenas uma nova chamada.

```
def fatorial(n):  # Exemplo já visto
    if n <= 1:
        return 1
    return n * fatorial(n - 1)
```

2. Recursividade Binária

Cada chamada recursiva gera duas novas chamadas.

```
def fibonacci(n):  # Exemplo já visto
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

3. Recursividade de Cauda (Tail Recursion)

A chamada recursiva é a última operação da função.

```
def fatorial_cauda(n, acumulador=1):
    if n <= 1:
```

```python
        return acumulador
    return fatorial_cauda(n - 1, n * acumulador)
```

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

4. Recursividade Mútua

Duas ou mais funções se chamam mutuamente.

```python
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.

```python
# 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.

```python
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:

```python
# ✅ 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:

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

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

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

Contando Chamadas:

```python
contador_chamadas = 0

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

    if n <= 1:
        return n
```

```python
    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

```python
# 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!"
```

"Principio da Confiança":

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

3. Otimizações Práticas:

Memoização Automática:

```python
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:

```python
# 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

```python
# 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

```python
# Some todos os números de 1 até n
def soma_ate_n(n):
```

```python
    # Seu código aqui
    pass

# Teste: soma_ate_n(5) deve retornar 15 (1+2+3+4+5)
```

Exercício 1.3: Potência

```python
# 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

```python
# 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

```python
# 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

```python
# 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

```python
# 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

```python
# 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:

```python
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:

```python
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:

```python
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:

```python
# 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)
```

```python
# 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)
```

Algoritmos em Árvores

Árvore Binária

Uma árvore onde cada nó tem no máximo dois filhos.

Traversal de Árvore:

- **Inorder**: Esquerda → Raiz → Direita
- **Preorder**: Raiz → Esquerda → Direita

- **Postorder**: Esquerda → Direita → Raiz

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

def inorder(raiz):
    if raiz:
        inorder(raiz.esquerda)
        print(raiz.valor)
        inorder(raiz.direita)
```

Algoritmos de Grafos

Representação:

- **Lista de Adjacência**: Mais eficiente em espaço
- **Matriz de Adjacência**: Mais eficiente para consultas

Busca em Profundidade (DFS):

```
def dfs(grafo, inicio, visitados=set()):
    visitados.add(inicio)
    print(inicio)

    for vizinho in grafo[inicio]:
        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

```python
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 Φ(D) que mede a "energia armazenada" na estrutura de dados.

Fórmula: Custo amortizado = Custo real + Φ(D') - Φ(D)

Exemplo: Array Dinâmico com Potencial

```python
# 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)

```python
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
```

```python
            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(α(n)) amortizado
# - α(n) é a função inversa de Ackermann, praticamente constante
```

Fibonacci Heap

```python
# Operações em Fibonacci Heap (conceitual):
# - Insert: O(1) amortizado
# - Extract-Min: O(log n) amortizado
# - Decrease-Key: O(1) amortizado
# - Delete: O(log n) amortizado
# - Union: O(1) real

# A análise amortizada é crucial aqui porque:
# - Extract-Min pode ser O(n) no pior caso
# - Mas o potencial acumulado pelas inserções "paga" por isso
```

CAPÍTULO 8

INVARIANTES DE LOOP

8.1 Definição e Importância

O que são Invariantes de Loop?

Uma **invariante de loop** é uma propriedade que:

1. É verdadeira antes da primeira iteração do loop
2. Se é verdadeira antes de uma iteração, permanece verdadeira após a iteração
3. Quando o loop termina, a invariante + condição de parada implica na correção do algoritmo

Como Usar Invariantes para Provar Correção

Exemplo 1: Busca Linear

```python
def busca_linear(arr, x):
    """
    Invariante: arr[0..i-1] não contém x
    """
    for i in range(len(arr)):
        # Invariante: x não está em arr[0..i-1]

        if arr[i] == x:
            return i  # Encontrado!

        # Invariante se mantém: x não está em arr[0..i]

    # Loop terminou: x não está em arr[0..n-1] = arr completo
    return -1
```

Prova da Invariante:

- **Inicialização**: Antes da primeira iteração (i=0), arr[0..-1] é vazio, então não contém x ✓
- **Manutenção**: Se arr[0..i-1] não contém x e arr[i] ≠ x, então arr[0..i] não contém x ✓
- **Terminação**: Se loop termina, então arr[0..n-1] não contém x ✓

Exemplo 2: Insertion Sort

```python
def insertion_sort(arr):
    """
    Invariante: arr[0..i-1] está ordenado
    """
    for i in range(1, len(arr)):
        # Invariante: arr[0..i-1] está ordenado

        key = arr[i]
        j = i - 1
```

```python
        # Invariante do loop interno: arr[j+2..i] > key e arr[0..j] ∪ {key} ∪ arr[j+2..i]
        # é uma permutação de arr[0..i] original
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        arr[j + 1] = key

        # Invariante se mantém: arr[0..i] agora está ordenado

    # Loop terminou: arr[0..n-1] está ordenado
    return arr
```

Prova da Invariante:

- **Inicialização**: arr[0..0] tem um elemento, logo está ordenado ✓
- **Manutenção**: Se arr[0..i-1] está ordenado, após inserir arr[i] na posição correta, arr[0..i] fica ordenado ✓
- **Terminação**: arr[0..n-1] = array completo está ordenado ✓

Exemplo 3: Busca Binária

```python
def busca_binaria(arr, x):
    """
    Invariante: se x está no array, então x está em arr[left..right]
    """
    left, right = 0, len(arr) - 1

    # Invariante inicial: se x existe, está em arr[0..n-1]

    while left <= right:
        # Invariante: se x existe no array original, então x está em arr[left..right]

        mid = (left + right) // 2

        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            left = mid + 1  # x só pode estar em arr[mid+1..right]
        else:
            right = mid - 1  # x só pode estar em arr[left..mid-1]

        # Invariante se mantém com novo intervalo [left, right]

    # Loop terminou com left > right: intervalo vazio, x não existe
    return -1
```

Invariantes em Algoritmos Mais Complexos

Exemplo: Algoritmo de Dijkstra

```python
import heapq

def dijkstra(graph, start):
    """
    Invariante: Para todo vértice v em S (conjunto de vértices processados),
    dist[v] é a distância mínima real de start até v
    """
    dist = {v: float('inf') for v in graph}
    dist[start] = 0
    pq = [(0, start)]
    S = set()  # Conjunto de vértices processados

    # Invariante inicial: S = {}, dist[start] = 0, dist[outros] = ∞

    while pq:
        # Invariante: Para todo v em S, dist[v] é ótimo

        current_dist, u = heapq.heappop(pq)

        if u in S:
            continue

        S.add(u)
        # u agora tem distância ótima (propriedade do algoritmo guloso)

        for v, weight in graph[u]:
            if v not in S and dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight
                heapq.heappush(pq, (dist[v], v))

        # Invariante se mantém: todos os vértices em S têm distância ótima

    return dist
```

Como Criar Invariantes

Passo 1: Identifique o objetivo

"O que o algoritmo deve conseguir ao final?"

Passo 2: Generalize para o meio do loop

"Que progresso parcial o algoritmo fez até agora?"

Passo 3: Verifique as três propriedades

1. **Inicialização**: Verdadeira antes do primeiro loop
2. **Manutenção**: Se verdadeira antes, continua após iteração
3. **Terminação**: Invariante + condição de parada = correção

Exemplo Prático: Encontrar Máximo

```python
def encontrar_maximo(arr):
    """
    Objetivo: Retornar o maior elemento do array
    Invariante: max_so_far é o maior elemento em arr[0..i-1]
    """
    if not arr:
        return None

    max_so_far = arr[0]  # Inicialização: maior em arr[0..0]

    for i in range(1, len(arr)):
        # Invariante: max_so_far = max(arr[0..i-1])

        if arr[i] > max_so_far:
            max_so_far = arr[i]

        # Invariante se mantém: max_so_far = max(arr[0..i])

    # Terminação: max_so_far = max(arr[0..n-1]) = máximo do array
    return max_so_far
```

Invariantes em C

```c
#include <stdio.h>

int encontrar_maximo(int arr[], int n) {
    /*
     * 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. **Manutenção**: 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:

```
# 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)
```

```python
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(n²)     - Comparar todos x todos [====================]
O(2ⁿ)     - Explorar combinações  [XXXXXXXXXXXXXXXX]
O(n!)     - Impossível na prática [XXXXXXXXXXXXXXXXXXXXX]
```

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:

```python
# 1. Conte os loops:
for i in range(n):      # O(n)
    for j in range(n):  # × 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:

```python
# 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
Array	O(1)	O(n)	O(n)	O(n)	Acesso rápido por índice
Lista Ligada	O(n)	O(n)	O(1)*	O(1)*	Inserções/remoções frequentes
Pilha	O(1) topo	-	O(1)	O(1)	LIFO, desfazer, recursão
Fila	O(1) frente	-	O(1)	O(1)	FIFO, processamento ordem
Hash Table	O(1)*	O(1)*	O(1)*	O(1)*	Busca super rápida
Árvore Binária	O(log n)*	O(log n)*	O(log n)*	O(log n)*	Dados ordenados

** No caso médio*

Algoritmos Essenciais:

```
BUSCA:
   Linear → O(n) → Simples, qualquer lista
   Binária → O(log n) → Lista ordenada obrigatória

ORDENAÇÃO:
   Bubble/Selection → O(n²) → Só para estudar
   Insertion → O(n²) → Bom para listas pequenas
   Merge → O(n log n) → Estável, sempre eficiente
   Quick → O(n log n)* → Rápido na prática

ÁRVORES:
   DFS → Profundidade primeiro → Recursivo
   BFS → Largura primeiro → Fila

OTIMIZAÇÃO:
   Programação Dinâmica → Subproblemas sobrepostos
   Guloso → Escolhas localmente ótimas
   Dividir e Conquistar → 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
```

```python
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:

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

# Exemplo: Contar caminhos em grade
def contar_caminhos(m, n):
    # DP: O(m×n)
    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:

```python
# 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

APÊNDICES

APÊNDICE A - TABELA DE COMPLEXIDADES

Tabela Resumo de Complexidades Comuns

Complexidade	Nome	Exemplo	n=10	n=100	n=1000
O(1)	Constante	Acesso a array[i]	1	1	1
O(log n)	Logarítmica	Busca binária	3	7	10
O(n)	Linear	Busca linear	10	100	1000
O(n log n)	Linearítmica	Merge Sort	30	700	10000
$O(n^2)$	Quadrática	Bubble Sort	100	10000	1000000
$O(2^n)$	Exponencial	Subconjuntos	1024	2^{100}	2^{1000}
O(n!)	Fatorial	Permutações	3628800	100!	1000!

Complexidades por Estrutura de Dados

Estrutura	Acesso	Busca	Inserção	Remoção
Array	O(1)	O(n)	O(n)	O(n)
Lista Ligada	O(n)	O(n)	O(1)	O(1)
Pilha	O(1)	-	O(1)	O(1)
Fila	O(1)	-	O(1)	O(1)
Hash Table	O(1)*	O(1)*	O(1)*	O(1)*
Árvore Binária	O(log n)*	O(log n)*	O(log n)*	O(log n)*
Heap	O(1)	O(n)	O(log n)	O(log n)

*Caso médio

APÊNDICE B - 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 C - BIBLIOGRAFIA E REFERÊNCIAS

Bibliografia Básica

1. **CORMEN, Thomas H. et al.** *Introduction to Algorithms*, 4th Edition. MIT Press, 2022.

2. **SEDGEWICK, 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". *SIGACT 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:

```python
# 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:

```python
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
