Revisão de Análise de Algoritmos

Índice

Introdução à Análise de Algoritmos

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.

Complexidade de Tempo e Espaço

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.

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

---

Notação Big-O

A notação Big-O descreve o comportamento assintótico de algoritmos.

Classes de Complexidade Comuns:

Notação	Nome	Exemplo
$O(1)$	Constante	Acesso a array por índice
$O(\log n)$	Logarítmica	Busca binária
$O(n)$	Linear	Busca linear
$O(n \log n)$	Linearítmica	Merge Sort, Quick Sort
$O(n^2)$	Quadrática	Bubble Sort, Selection Sort
$O(n^3)$	Cúbica	Multiplicação de matrizes ingênua
$O(2^n)$	Exponencial	Torres de Hanói
$O(n!)$	Fatorial	Problema do caixeiro viajante

Regras para Análise:

1. **Constantes são ignoradas**: $O(2n) = O(n)$
2. **Termo dominante**: $O(n^2 + n) = O(n^2)$
3. **Pior caso**: Consideramos sempre o pior cenário

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)

Algoritmos de Ordenação

Bubble Sort

- **Complexidade**: $O(n^2)$
- **Estável**: Sim
- **In-place**: Sim

Selection Sort

- **Complexidade**: $O(n^2)$
- **Estável**: Não
- **In-place**: Sim

Insertion Sort

- **Complexidade**: $O(n^2)$ - pior caso, O(n) - melhor caso
- **Estável**: Sim
- **In-place**: Sim

Merge Sort

- **Complexidade**: O(n log n)
- **Estável**: Sim
- **In-place**: Não

Quick Sort

- **Complexidade**: O(n log n) - médio, $O(n^2)$ - pior caso
- **Estável**: Não
- **In-place**: Sim

Algoritmos de Busca

Busca Linear

```python
def busca_linear(lista, elemento):
    for i in range(len(lista)):
        if lista[i] == elemento:
            return i
    return -1
```

Complexidade: O(n)

Busca Binária

```python
def busca_binaria(lista, elemento):
    esquerda, direita = 0, len(lista) - 1

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

```
        if lista[meio] == elemento:
            return meio
        elif lista[meio] < elemento:
            esquerda = meio + 1
        else:
            direita = meio - 1

    return -1
```

Complexidade: O(log n)

RECURSIVIDADE

Conceitos Fundamentais

O que é Recursividade?

Recursividade é uma técnica de programação onde uma função chama a si mesma para resolver subproblemas menores do mesmo tipo. É uma alternativa elegante à iteração para muitos problemas.

Elementos de uma Função Recursiva:

1. Caso Base (Base Case)

A condição que para a recursão. Sem ele, a função executaria infinitamente.

2. Caso Recursivo (Recursive Case)

A parte onde a função chama a si mesma com um problema menor.

3. Progresso em Direção ao Caso Base

Cada chamada recursiva deve nos aproximar do caso base.

Estrutura Básica:

```
def funcao_recursiva(parametro):
    # Caso base
    if condicao_parada:
        return valor_base

    # Caso recursivo
    return funcao_recursiva(parametro_menor)
```

Exemplos Clássicos de Recursividade

1. Fatorial

O fatorial de n (n!) é o produto de todos os números inteiros positivos de 1 até n.

Definição Matemática:

- n! = n × (n-1) × (n-2) × ... × 1
- 0! = 1 (por definição)

Implementação Recursiva:

```
def fatorial(n):
    # Caso base
    if n == 0 or n == 1:
        return 1

    # Caso recursivo
    return n * fatorial(n - 1)

# Exemplo de uso
print(fatorial(5))  # Output: 120
```

Análise de Complexidade:

- Tempo: O(n)
- Espaço: O(n) - devido à pilha de chamadas

Trace de Execução para fatorial(4):

```
fatorial(4)
├── 4 * fatorial(3)
    ├── 3 * fatorial(2)
        ├── 2 * fatorial(1)
            └── 1 (caso base)
        └── 2 * 1 = 2
    └── 3 * 2 = 6
└── 4 * 6 = 24
```

2. Sequência de Fibonacci

A sequência de Fibonacci é definida como:

- F(0) = 0
- F(1) = 1
- F(n) = F(n-1) + F(n-2) para n > 1

Implementação Recursiva Simples:

```
def fibonacci(n):
    # Casos base
    if n == 0:
        return 0
    if n == 1:
        return 1

    # Caso recursivo
    return fibonacci(n - 1) + fibonacci(n - 2)
```

```python
# Exemplo
print(fibonacci(6))  # Output: 8
```

Análise de Complexidade:

- Tempo: $O(2^n)$ - muito ineficiente!
- Espaço: O(n) - profundidade da recursão

Fibonacci Otimizado (Memoização):

```python
def fibonacci_memo(n, memo={}):
    if n in memo:
        return memo[n]

    if n == 0:
        return 0
    if n == 1:
        return 1

    memo[n] = fibonacci_memo(n - 1, memo) + fibonacci_memo(n - 2, memo)
    return memo[n]
```

Complexidade Otimizada:

- Tempo: O(n)
- Espaço: O(n)

3. Torres de Hanói

Problema clássico que envolve mover discos entre três torres seguindo regras específicas.

Regras:

1. Só pode mover um disco por vez
2. Só pode mover o disco do topo de uma torre
3. Não pode colocar um disco maior sobre um menor

Implementação:

```python
def torres_hanoi(n, origem, destino, auxiliar):
    if n == 1:
        print(f"Mover disco de {origem} para {destino}")
        return

    # Mover n-1 discos para torre auxiliar
    torres_hanoi(n - 1, origem, auxiliar, destino)

    # Mover o disco maior para o destino
    print(f"Mover disco de {origem} para {destino}")

    # Mover n-1 discos da auxiliar para o destino
    torres_hanoi(n - 1, auxiliar, destino, origem)
```

```python
# Exemplo
torres_hanoi(3, 'A', 'C', 'B')
```

Complexidade: $O(2^n)$

Recursividade em Estruturas de Dados

1. Soma de Elementos em Lista

```python
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

```python
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

```python
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:

```python
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.

```python
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.

```python
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.

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

```python
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 Debugging

1. Stack Overflow

Causa: Recursão muito profunda ou sem caso base adequado.

Solução:

```python
import sys
sys.setrecursionlimit(10000)  # Aumentar limite (use com cuidado)
```

2. Casos Base Incorretos

Problema:

```
def conta_regressiva(n):
    print(n)
    return conta_regressiva(n - 1)  # Sem caso base!
```

Solução:

```
def conta_regressiva(n):
    if n <= 0:  # Caso base
        return
    print(n)
    conta_regressiva(n - 1)
```

3. Parâmetros Incorretos

Certifique-se de que cada chamada recursiva progride em direção ao caso base.

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)
```

```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])
```

Programação Dinâmica

Princípios:

1. **Subestrutura Ótima**: A solução ótima contém soluções ótimas de subproblemas
2. **Sobreposição de Subproblemas**: Os mesmos subproblemas são resolvidos múltiplas vezes

Exemplo: Problema da Mochila

```
def mochila(pesos, valores, capacidade):
    n = len(pesos)
    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):
            if pesos[i-1] <= w:
                dp[i][w] = max(
                    valores[i-1] + dp[i-1][w - pesos[i-1]],
                    dp[i-1][w]
                )
            else:
                dp[i][w] = dp[i-1][w]

    return dp[n][capacidade]
```

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)
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 dos Pontos Principais

Complexidade:

- **O(1)**: Constante - ideal
- **O(log n)**: Logarítmica - muito boa
- **O(n)**: Linear - boa
- **O(n log n)**: Linearítmica - aceitável
- **$O(n^2)$**: Quadrática - evitar para grandes entradas
- **$O(2^n)$**: Exponencial - evitar

Estratégias de Algoritmos:

1. **Força Bruta**: Testar todas as possibilidades
2. **Dividir e Conquistar**: Quebrar em subproblemas menores
3. **Programação Dinâmica**: Resolver subproblemas e reutilizar soluções
4. **Algoritmos Gulosos**: Fazer escolhas localmente ótimas
5. **Backtracking**: Explorar todas as possibilidades com retrocesso

Recursividade - Pontos Chave:

- **Sempre defina um caso base claro**
- **Certifique-se de que a recursão progride em direção ao caso base**
- **Considere o uso de memoização para otimizar**
- **Avalie se uma solução iterativa seria mais eficiente**
- **Cuidado com o limite da pilha de recursão**

Dicas para Análise:

1. Identifique as operações dominantes
2. Conte quantas vezes elas são executadas
3. Expresse em função do tamanho da entrada
4. Simplifique usando regras da notação Big-O

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

Este documento serve como um guia completo para revisão de algoritmos e análise de complexidade, com foco especial em recursividade. Continue praticando e explorando novos problemas para aprofundar seu conhecimento!