Complexidade de Algoritmos

Análise de Eficiência Computacional

Prof. Fernando Figueira

UFRN

7 de Novembro de 2025

Material compilado em 7 de novembro de 2025.

O que é Complexidade de Algoritmos?

Definição

Medida teórica do consumo de recursos computacionais (tempo e memória) de um algoritmo em função do tamanho da entrada.

- **Independente de hardware**: Analisa o algoritmo, não a máquina
- **Foco no crescimento**: Como o consumo escala com o tamanho da entrada
- **Análise assintótica**: Comportamento para entradas muito grandes

Por que estudar complexidade?

Aplicações práticas:

- Escolha de algoritmos
- Previsão de desempenho
- Otimização de recursos
- Projeto de sistemas

Exemplo real:

- Algoritmo $O(n^2)$: 1s para 1000 elementos
- Algoritmo $O(n \log n)$: 10s para 10000 elementos
- Algoritmo $O(n^2)$: 100s para 10000 elementos

Complexidade Temporal

Complexidade Temporal
Mede o número de operações em função do tamanho da entrada n.

O(1)	Tempo constante (acesso a array)
O(log n)	Tempo logarítmico (busca binária)
O(n)	Tempo linear (busca sequencial)
O(n log n)	(Merge sort, Quick sort)
O(n^2)	Tempo quadrático (Bubble sort)
O(2^n)	Tempo exponencial (Fibonacci recursivo)

Complexidade Espacial

Complexidade Espacial
Mede a quantidade de memória utilizada.

O(1)	Uso constante de memória (algoritmos in-place)
O(n)	Memória proporcional à entrada
$O(n^2)$	Matriz $n \times n$
O(log n)	Pilha de recursão

Hierarquia de Memória

Hierarquia de Memória e Custos:

1. **Registradores**: Mais rápido, menor capacidade
2. **Cache L1/L2/L3**: Muito rápido, acesso em nanosegundos
3. **RAM**: Rápido, acesso em microssegundos
4. **Disco/SSD**: Lento, acesso em milissegundos

Custo das Operações:

- Operação na CPU: 1 unidade de tempo
- Acesso à RAM: 100-200 unidades
- Acesso ao disco: 1.000.000 unidades

Exemplos Práticos

Exemplo 1: Busca Linear - O(n)

```
int busca_linear(int arr[], int n, int alvo) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == alvo) {
            return i;
        }
    }
    return -1;
}
```

Exemplo 2: Bubble Sort - $O(n^2)$

```
void bubble_sort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

Exemplo 3: Busca Binária - O(log n)

```
int busca_binaria(int arr[], int esq, int dir, int alvo)
    while (esq <= dir) {
        int meio = esq + (dir - esq) / 2;
        if (arr[meio] == alvo) return meio;
        if (arr[meio] < alvo) esq = meio + 1;
        else dir = meio - 1;
    }
    return -1;
}
```

Análise do Problema 2

Problema: Encontrar trios de primos (x, x+2, x+6) para x <= 50000

Solução Não Otimizada:

```
for (int x = 2; x <= 50000; x++) {
    if (primo(x) && primo(x+2) && primo(x+6)) {
        printf("(%d, %d, %d)\n", x, x+2, x+6);
    }
}
```

Problemas da Solução Não Otimizada:

- Verificações redundantes
- Cada número verificado múltiplas vezes
- Complexidade: $O(n \times \sqrt{n})$
- $\approx$ 150.000 chamadas à função primo()

Solução Otimizada com Vetor:

```
int janela[7] = {0};
for (int i = 0; i < 7; i++) {
    janela[i] = primo(inicio + i);
}

for (int x = inicio; x <= 50000; x++) {
    if (janela[0] && janela[2] && janela[6]) {
        printf("(%d, %d, %d)\n", x, x+2, x+6);
    }
    // Atualiza janela
    for (int i = 0; i < 6; i++) {
        janela[i] = janela[i+1];
    }
    janela[6] = primo(x + 7);
}
```

Vantagens da Solução Otimizada:

- Reduz chamadas à função primo()
- Reutiliza verificações anteriores
- Complexidade: O(n)
- 50.000 chamadas à função primo()
- Redução de 66% nas operações

Solução com Lista Ligada:

```
// Implementacao com lista ligada para reduzir operacoes
typedef struct Node {
    int value;
    bool is_prime;
    struct Node* next;
} Node;

// Funcao para criar um novo no
Node* create_node(int value, bool is_prime) {
    Node* new_node = (Node*)malloc(sizeof(Node));
    new_node->value = value;
    new_node->is_prime = is_prime;
    new_node->next = NULL;
    return new_node;
}

// Inicializa lista com primeiros 7 numeros
for (int i = 2; i <= 8; i++) {
    bool prime = is_prime(i);
    Node* new_node = create_node(i, prime);
    // Adiciona a lista...
```

Comparação das Soluções:

Métrica	Não Otimizada	Vetor	Lista
Chamadas primo()	150.000	50.000	50.000
Operações escrita	0	6/iteração	2/iteração
Uso memória	O(1)	O(1)	O(k)
Simplicidade	Alta	Média	Baixa

Ferramentas de Análise

Ferramentas Open-Source:

- **Valgrind**: Análise de memória e performance
- **gprof**: Profiling de código
- **time**: Medição de tempo de execução
- **perf**: Análise de performance do Linux

Instalação:

```
sudo apt-get install valgrind gcc  # Linux
brew install valgrind gcc          # macOS
# Windows: usar WSL ou ferramentas alternativas
```

Uso do Valgrind - Callgrind:

```
valgrind --tool=callgrind ./primos_nao_otimizado
valgrind --tool=callgrind ./primos_otimizado_vetor
valgrind --tool=callgrind ./primos_otimizado_lista

kcachegrind callgrind.out.*
```

Uso do Valgrind - Massif:

```
valgrind --tool=massif ./primos_nao_otimizado
valgrind --tool=massif ./primos_otimizado_vetor
valgrind --tool=massif ./primos_otimizado_lista

ms_print massif.out.*
```

Script de Comparação Automática:

```
#!/bin/bash
echo "=== COMPARACAO DE DESEMPENHO ==="

echo -e "\n1. Solucao Nao Otimizada:"
time ./primos_nao_otimizado > /dev/null

echo -e "\n2. Solucao Otimizada com Vetor:"
time ./primos_otimizado_vetor > /dev/null

echo -e "\n3. Solucao Otimizada com Lista:"
time ./primos_otimizado_lista > /dev/null
```

Métricas a Observar:

- Tempo real de execução
- Número de instruções executadas
- Cache misses
- Uso de memória heap
- Número de chamadas da função primo()

Resumo e Próximos Passos

Pontos Principais:

- Complexidade ajuda na escolha de algoritmos adequados
- Trade-off entre tempo e espaço
- Otimizações devem considerar o contexto de uso
- Ferramentas de profiling são essenciais para análise prática

Próximos Tópicos:

- Notações assintóticas (Big O, Theta, Omega)
- Análise de casos (melhor, médio, pior)
- Complexidade amortizada
- Estruturas de dados avançadas

Perguntas?

Obrigado!