🎓 Algoritmos e Complexidade

Aula 01: Algoritmos - Funções e Passagem de Parâmetros

Prof. Vagner Cordeiro
Sistemas de Informação
Universidade - 2024

💡 Objetivo: **Dominar conceitos fundamentais de algoritmos, funções e análise matemática**

📋 Agenda da Aula

1. 🔍 Conceitos Fundamentais de Algoritmos

Definições matemáticas e propriedades essenciais

2. 📊 Análise de Complexidade e Big-O

Notações assintóticas e hierarquia de complexidade

3. 💻 Linguagens: C vs Python

Comparações práticas e escolhas técnicas

4. ⚙️ Funções e Modularização

Implementação, escopo e boas práticas

🎯 Objetivos de Aprendizagem

Ao final desta aula, você será capaz de:

- ✅ **Definir** algoritmos de forma matemática rigorosa
- ✅ **Implementar** funções eficientes em C e Python
- ✅ **Analisar** complexidade usando notação Big-O
- ✅ **Aplicar** técnicas de passagem de parâmetros
- ✅ **Otimizar** código usando memoização e recursão
- ✅ **Comparar** performance entre diferentes abordagens

🔍 1. Conceitos Fundamentais

📐 Definição Matemática de Algoritmo

Um **algoritmo** é uma função matemática:

$$A: D \rightarrow C$$

Onde:

• D = Domínio (entradas válidas)

• C = Contradomínio (saídas possíveis)

• A = Transformação algorítmica

🔄 Visualização Conceitual

ENTRADA → [ALGORITMO] → **SAÍDA**

$x \in D$ → [Processamento] → $A(x) \in C$

⚖️ Propriedades Fundamentais dos Algoritmos

🏁 1. Finitude

$$\forall x \in D, \text{ o algoritmo } A(x) \text{ termina em tempo finito}$$

🎯 2. Determinismo

$$\forall x \in D, \text{ } A(x) \text{ produz sempre o mesmo resultado}$$

⚡ 3. Efetividade

$$\text{Cada instrução deve ser executável em tempo finito}$$

⚠️ Importante: **Algoritmos que não satisfazem essas propriedades não são considerados válidos!**

📊 Análise de Complexidade: Big-O

📚 Definição Formal

$$f(n) = O(g(n)) \text{ se e somente se}$$
$$\exists c > 0, n_0 \geq 0 \text{ tal que } 0 \leq f(n) \leq c \cdot g(n)$$
$$\forall n \geq n_0$$

📈 Hierarquia de Complexidade (do melhor ao pior)

$$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

📊 Gráfico Visual de Crescimento

Crescimento das Funções de Complexidade

Para n = 1000:

- O(1): 1 operação ⚡
- O(log n): ~10 operações 🚀
- O(n): 1.000 operações ✅
- O(n^2): 1.000.000 operações ⚠️
- O(2^n): 10^{300} operações ❌ (impossível!)

💡 Regra de Ouro: **Prefira sempre complexidades menores!**

💻 2. Linguagens de Programação

⚔️ Comparação: C vs Python

🔍 Aspecto	🔧 C	🐍 Python
🎯 **Paradigma**	Procedural	Multi-paradigma
⚡ **Execução**	Compilada (rápida)	Interpretada (flexível)
🏷️ **Tipagem**	Estática (segura)	Dinâmica (flexível)
🧠 **Memória**	Manual (controle total)	Automática (GC)
🚀 **Performance**	Alta (10-100x)	Moderada
⏰ **Desenvolvimento**	Lento	Rápido

🧮 Exemplo Prático: Função Factorial

🔧 Implementação em C:

```c
#include <stdio.h>

// Versão recursiva limpa
long long factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

int main() {
    printf("5! = %lld\n", factorial(5));
    return 0;
}
```

Performance: ~0.001ms para n=10

🐍 Implementação em Python:

```python
def factorial(n):
    """Calcula o fatorial de n recursivamente"""
    if n <= 1:
        return 1
    return n * factorial(n - 1)

def main():
    print(f"5! = {factorial(5)}")

if __name__ == "__main__":
    main()
```

Performance: ~0.01ms para n=10 (10x mais lenta)

🎯 Conclusão: **C é mais rápido, Python é mais legível!**

📐 3. Funções: Base Matemática

🎯 Definição Formal de Função

Uma função $f: A \rightarrow B$ associa:
• Cada elemento $a \in A$ (domínio)
• A exatamente um elemento $b \in B$ (contradomínio)

$$f(a) = b$$

🔍 Propriedades Matemáticas Importantes

🎯 **Injetividade:** $f(x_1) = f(x_2) \Rightarrow x_1 = x_2$ (um-para-um)

🎯 **Sobrejetividade:** $\forall b \in B, \exists a \in A: f(a) = b$ (sobre)

🎯 **Bijetividade:** Injetiva E Sobrejetiva (correspondência perfeita)

⚙️ 4. Implementação de Funções em C

🏗️ Estrutura Básica

```c
tipo_retorno nome_funcao(lista_parametros) {
    // Documentação interna
    // Validação de entrada
    // Processamento
    return valor;
}
```

⚡ Exemplo: Função Potência Simples

```c
double potencia(double base, int expoente) {
    double resultado = 1.0;

    for (int i = 0; i < expoente; i++) {
        resultado *= base;
    }

    return resultado;
}
```

Complexidade: $T(n) = O(n)$ onde n é o expoente

🚀 Otimização: Exponenciação Rápida

💡 Algoritmo Inteligente

$$a^n = \begin{cases} 1 & \text{se } n = 0 \\ (a^{n/2})^2 & \text{se } n \text{ é par} \\ a \cdot a^{n-1} & \text{se } n \text{ é ímpar} \end{cases}$$

```
double potencia_rapida(double base, int exp) {
    if (exp == 0) return 1.0;

    if (exp % 2 == 0) {
        double temp = potencia_rapida(base, exp/2);
        return temp * temp;  // Reutiliza cálculo!
    }

    return base * potencia_rapida(base, exp-1);
}
```

🚀 **Melhoria:** De O(n) para O(log n) - ganho exponencial!

📊 Comparação de Performance

Potência de 2^{10} = 1024

🐌 **Versão Simples:** 10 multiplicações

🚀 **Versão Rápida:** 4 multiplicações

Para 2^{30}:

🐌 Simples: 30 operações

🚀 Rápida: 5 operações (6x mais rápido!)

🔄 5. Passagem de Parâmetros

📋 Tipos de Passagem

1. 📄 Por Valor (Call by Value)
Cópia segura, sem efeitos colaterais

2. 🔗 Por Referência (Call by Reference)
Acesso direto, modificações persistem

3. 📍 Por Ponteiro (Call by Pointer)
Flexibilidade máxima, controle total

📄 Passagem por Valor - Segura

🛡️ Características

- ✅ Segura (não modifica original)
- ❌ Custosa para dados grandes
- ✅ Sem efeitos colaterais

```c
void incrementa_valor(int x) {
    x++;  // Modifica apenas a cópia local
    printf("Dentro da função: %d\n", x);
}

int main() {
    int num = 5;
    incrementa_valor(num);
    printf("Fora da função: %d\n", num);  // Ainda é 5!
    return 0;
}
```

🔍 Saída: **Dentro: 6, Fora: 5**

🔗 Passagem por Ponteiro - Poderosa

⚡ Características

- ✅ Eficiente (apenas endereço)
- ⚠️ Pode modificar original
- ✅ Flexível para estruturas grandes

```c
void incrementa_ponteiro(int *x) {
    (*x)++;  // Modifica o valor original!
    printf("Dentro da função: %d\n", *x);
}

int main() {
    int num = 5;
    incrementa_ponteiro(&num);  // Passa endereço
    printf("Fora da função: %d\n", num);  // Agora é 6!
    return 0;
}
```

🔍 **Saída:** Dentro: 6, Fora: 6

💰 Análise de Custo: Passagem de Parâmetros

📊 Custo Computacional

Por Valor: $\text{Custo} = O(\text{sizeof}(\text{tipo}))$

Por Ponteiro: $\text{Custo} = O(1)$ sempre

📈 Exemplo Prático

Tipo de Dado	Por Valor	Por Ponteiro	Economia
`int`	4 bytes	8 bytes	❌ Pior
`struct (100 bytes)`	100 bytes	8 bytes	✅ 92% menor
`array[1000]`	4000 bytes	8 bytes	✅ 99.8% menor

💡 Regra: **Use ponteiros para estruturas grandes!**

📚 6. Funções com Arrays

🎯 Comportamento Especial

Arrays em C são **sempre** passados por referência!

```
void processa_array(int arr[], int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        arr[i] *= 2;  // Modifica array original!
    }
}

int main() {
    int numeros[5] = {1, 2, 3, 4, 5};

    printf("Antes: ");
    imprimir_array(numeros, 5);  // 1 2 3 4 5

    processa_array(numeros, 5);

    printf("Depois: ");
    imprimir_array(numeros, 5);  // 2 4 6 8 10

    return 0;
}
```

🧮 Função Matemática: Soma de Array

📐 Definição Matemática

$$\text{soma}(A) = \sum_{i=0}^{n-1} A[i]$$

💻 Implementação Eficiente

```
int soma_array(int arr[], int n) {
    int soma = 0;

    // Loop otimizado
    for (int i = 0; i < n; i++) {
        soma += arr[i];
    }

    return soma;
}
```

🚀 **Complexidade:** $T(n) = \Theta(n)$ - Linear e ótima!

🔄 7. Recursão: Poder Matemático

📐 Definição Formal

Uma função f é recursiva se:

$$f(n) = \begin{cases} \text{caso base} & \text{se } n \leq k \\ g(n, f(h(n))) & \text{se } n > k \end{cases}$$

Onde $h(n) < n$ (garante convergência)

🔍 Componentes Essenciais

1. 🛑 **Caso Base:** Condição de parada
2. 🔄 **Caso Recursivo:** Chamada a si mesmo
3. ⬇️ **Convergência:** Aproximação do caso base

🔢 Exemplo Clássico: Fibonacci

📐 Definição Matemática

$$F(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ F(n-1) + F(n-2) & \text{se } n > 1 \end{cases}$$

💻 Implementação Recursiva Simples

```
long long fibonacci(int n) {
    // Casos base claros
    if (n <= 1) return n;

    // Caso recursivo
    return fibonacci(n-1) + fibonacci(n-2);
}
```

⚠️ Problema: **Complexidade exponencial $O(\phi^n)$!**

⚡ Otimização: Fibonacci com Memoização

🧠 Técnica Inteligente

```c
#define MAX_N 100
long long memo[MAX_N];
int inicializado = 0;

long long fibonacci_memo(int n) {
    // Inicialização única
    if (!inicializado) {
        for (int i = 0; i < MAX_N; i++) memo[i] = -1;
        inicializado = 1;
    }

    // Casos base
    if (n <= 1) return n;

    // Verifica cache
    if (memo[n] != -1) return memo[n];

    // Calcula e armazena
    memo[n] = fibonacci_memo(n-1) + fibonacci_memo(n-2);
    return memo[n];
}
```

🚀 **Melhoria Dramática:** De $O(\phi^n)$ para $O(n)$!

📊 Comparação de Performance: Fibonacci

Fibonacci(40):

🐌 **Recursivo Simples:** ~1.5 segundos
🧠 **Com Memoização:** ~0.001 segundos
⚡ **Iterativo:** ~0.0001 segundos

Fibonacci(100):
🐌 Recursivo: Impossível (anos)
🧠 Memoização: Instantâneo
⚡ Iterativo: Instantâneo

🎯 8. Ponteiros para Funções

🔧 Conceito Avançado

Ponteiros podem apontar para funções!

```c
// Declaração de ponteiro para função
int (*operacao)(int, int);

// Funções matemáticas
int soma(int a, int b) { return a + b; }
int mult(int a, int b) { return a * b; }
int potencia(int a, int b) {
    int resultado = 1;
    for (int i = 0; i < b; i++) resultado *= a;
    return resultado;
}

// Uso dinâmico
operacao = soma;
int resultado = operacao(5, 3);  // 8
```

🧮 Calculadora Inteligente

```c
typedef int (*Operacao)(int, int);

void calculadora(int a, int b, Operacao op, const char* nome) {
    printf("%s(%d, %d) = %d\n", nome, a, b, op(a, b));
}

int main() {
    int x = 10, y = 3;

    calculadora(x, y, soma, "Soma");         // Soma(10, 3) = 13
    calculadora(x, y, mult, "Produto");      // Produto(10, 3) = 30
    calculadora(x, y, potencia, "Potência"); // Potência(10, 3) = 1000

    return 0;
}
```

💡 **Vantagem:** Código flexível e reutilizável!

🏗 9. Funções de Ordem Superior

📐 Conceito Matemático

Função que opera sobre outras funções:

$$H: (A \rightarrow B) \times A \rightarrow B$$

🗺 Exemplo: Função Map

```c
void map(int arr[], int n, int (*func)(int)) {
    for (int i = 0; i < n; i++) {
        arr[i] = func(arr[i]);
    }
}

// Funções de transformação
int quadrado(int x) { return x * x; }
int cubo(int x) { return x * x * x; }
int dobro(int x) { return x * 2; }
```

🎯 Aplicação Prática

```c
int main() {
    int nums[5] = {1, 2, 3, 4, 5};

    printf("Original: ");
    imprimir_array(nums, 5);  // 1 2 3 4 5

    map(nums, 5, quadrado);
    printf("Quadrados: ");
    imprimir_array(nums, 5);  // 1 4 9 16 25

    return 0;
}
```

🎯 Benefício: **Código mais limpo e funcional!**

⏱ 10. Medição de Performance

🔬 Ferramentas de Análise

```c
#include <time.h>

double medir_tempo(void (*funcao)(), int repeticoes) {
    clock_t inicio = clock();

    for (int i = 0; i < repeticoes; i++) {
        funcao();
    }

    clock_t fim = clock();
    return ((double)(fim - inicio)) / CLOCKS_PER_SEC;
}

void benchmark_algoritmos() {
    printf("Fibonacci(30):\n");
    printf("Recursivo: %.6f s\n", medir_tempo_fibonacci(30, fibonacci));
    printf("Memoização: %.6f s\n", medir_tempo_fibonacci(30, fibonacci_memo));
}
```

📊 Resultados de Benchmark

Comparação Real (Fibonacci 35):

🐌 **Recursivo Puro:** 1.234 segundos
🧠 **Com Memoização:** 0.001 segundos
⚡ **Iterativo DP:** 0.0003 segundos
🚀 **Fórmula Binet:** 0.00001 segundos

Ganho de Performance: 123.400x mais rápido!

🔍 11. Algoritmos de Busca

🔄 Busca Linear: Força Bruta

```c
int busca_linear(int arr[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) {
            return i;  // Encontrado na posição i
        }
    }
    return -1;  // Não encontrado
}
```

Complexidade:

• Melhor caso: $O(1)$ (primeiro elemento)

• Caso médio: $O(n/2) = O(n)$

• Pior caso: $O(n)$ (último elemento)

🎯 Busca Binária: Estratégia Inteligente

⚠️ Pré-requisito: **Array deve estar ordenado!**

```c
int busca_binaria(int arr[], int l, int r, int x) {
    while (l <= r) {
        int m = l + (r - l) / 2;  // Evita overflow!

        if (arr[m] == x) return m;        // Encontrado!

        if (arr[m] < x)
            l = m + 1;                    // Busca na metade direita
        else
            r = m - 1;                    // Busca na metade esquerda
    }
    return -1;                            // Não encontrado
}
```

🚀 **Complexidade:** $T(n) = O(\log n)$ - Logarítmica!

📈 Comparação Visual: Busca Linear vs Binária

Para um array de 1.000.000 elementos:

🔄 **Busca Linear:**

• Máximo: 1.000.000 comparações

• Média: 500.000 comparações

🎯 **Busca Binária:**

• Máximo: 20 comparações

• Sempre: ~20 comparações

Eficiência: 50.000x mais rápida!

🛡️ 12. Tratamento de Erros

📋 Sistema de Códigos de Erro

```
typedef enum {
    SUCCESS = 0,
    ERROR_NULL_POINTER = -1,
    ERROR_INVALID_INPUT = -2,
    ERROR_OUT_OF_BOUNDS = -3,
    ERROR_DIVISION_BY_ZERO = -4
} ErrorCode;

ErrorCode divisao_segura(double a, double b, double *resultado) {
    // Validação de ponteiro
    if (resultado == NULL) return ERROR_NULL_POINTER;

    // Validação matemática
    if (b == 0.0) return ERROR_DIVISION_BY_ZERO;

    // Operação segura
    *resultado = a / b;
    return SUCCESS;
}
```

🎯 Uso Prático do Sistema

```c
int main() {
    double resultado;
    ErrorCode status = divisao_segura(10.0, 3.0, &resultado);

    switch (status) {
        case SUCCESS:
            printf("Resultado: %.2f\n", resultado);
            break;
        case ERROR_DIVISION_BY_ZERO:
            printf("Erro: Divisão por zero!\n");
            break;
        default:
            printf("Erro inesperado: %d\n", status);
    }

    return 0;
}
```

💡 Benefício: **Código robusto e profissional!**

⚡ 13. Técnicas de Otimização

🧠 1. Memoização (Já vimos)

- Cache de resultados computados
- Troca espaço por tempo

🏃 2. Tail Recursion

```
// Recursão tradicional (pilha cresce)
int factorial_normal(int n) {
    if (n <= 1) return 1;
    return n * factorial_normal(n - 1);  // Operação após recursão
}

// Tail recursion (otimizável)
int factorial_tail(int n, int acc) {
    if (n <= 1) return acc;
    return factorial_tail(n - 1, n * acc);  // Recursão é última operação
}
```

🔄 3. Loop Unrolling

```c
// Loop normal
int soma_normal(int arr[], int n) {
    int soma = 0;
    for (int i = 0; i < n; i++) {
        soma += arr[i];
    }
    return soma;
}

// Loop desenrolado (mais rápido)
int soma_unrolled(int arr[], int n) {
    int soma = 0;
    int i;

    // Processa 4 elementos por vez
    for (i = 0; i < n - 3; i += 4) {
        soma += arr[i] + arr[i+1] + arr[i+2] + arr[i+3];
    }

    // Processa elementos restantes
    for (; i < n; i++) {
        soma += arr[i];
    }

    return soma;
}
```

💻 14. Comparação Final: C vs Python

🔧 Exemplo Completo em C

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

void quicksort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}

int main() {
    int arr[10000];
    // ... preencher array ...

    clock_t inicio = clock();
    quicksort(arr, 0, 9999);
    clock_t fim = clock();

    printf("Tempo C: %.6f s\n",
           ((double)(fim - inicio)) / CLOCKS_PER_SEC);
    return 0;
}
```

Equivalente em Python

```python
import time

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

    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return quicksort(left) + middle + quicksort(right)

# Teste de performance
arr = list(range(10000, 0, -1))  # Array reverso

inicio = time.time()
arr_ordenado = quicksort(arr)
fim = time.time()

print(f"Tempo Python: {fim - inicio:.6f} s")
```

📊 Resultado da Comparação

QuickSort - 10.000 elementos:

🔧 **C (otimizado):** 0.002 segundos
🐍 **Python (puro):** 0.150 segundos
🐍 **Python (sorted):** 0.001 segundos

Conclusão:
• C: Sempre rápido
• Python: Use bibliotecas otimizadas!

🎯 15. Boas Práticas Essenciais

📝 1. Nomenclatura Clara

```c
// ❌ Ruim
int calc(int x, int y) { return x + y; }
int f(int n) { /* factorial */ }

// ✅ Bom
int calcular_soma(int primeiro, int segundo) { return primeiro + segundo; }
int calcular_factorial(int numero) { /* implementação */ }
```

📚 2. Documentação Profissional

```c
/**
 * Calcula o fatorial de um número usando recursão otimizada
 * @param n: número inteiro não negativo (0 <= n <= 20)
 * @return: fatorial de n, ou -1 se entrada inválida
 * @complexity: O(n) tempo, O(n) espaço (pilha de recursão)
 * @example: factorial(5) retorna 120
 */
long long factorial(int n);
```

🧪 16. Debugging e Testes

🔍 Uso Estratégico de Assertions

```
#include <assert.h>

int divisao_inteira(int dividendo, int divisor) {
    // Pré-condições
    assert(divisor != 0);
    assert(dividendo >= 0);

    int resultado = dividendo / divisor;

    // Pós-condições
    assert(resultado * divisor <= dividendo);
    assert((resultado + 1) * divisor > dividendo);

    return resultado;
}
```

🎯 Suite de Testes Abrangente

```
void executar_todos_os_testes() {
    printf("🧪 Executando testes...\n");

    // Testes de funções básicas
    assert(factorial(0) == 1);
    assert(factorial(5) == 120);
    assert(fibonacci(10) == 55);
```

🏆 17. Conclusões e Próximos Passos

🎯 O que Dominamos Hoje:

- ✅ **Fundamentos Matemáticos:** Definições formais e rigorosas
- ✅ **Análise de Complexidade:** Big-O e otimizações
- ✅ **Implementação Prática:** Funções eficientes em C
- ✅ **Técnicas Avançadas:** Recursão, memoização, ponteiros
- ✅ **Comparações:** C vs Python em cenários reais
- ✅ **Boas Práticas:** Código profissional e robusto

🚀 Próxima Aula: Estruturas de Dados

- **Arrays multidimensionais** e matrizes
- **Ponteiros avançados** e aritmética
- **Structs e Unions** para dados complexos
- **Alocação dinâmica** e gerenciamento de memória

📝 Exercícios Desafiadores

1. 🚀 Implementação Avançada:
Crie uma função genérica de exponenciação modular: $a^b \bmod m$

2. 🧠 Otimização Inteligente:
Implemente memoização para função de Ackermann

3. 📊 Análise Empírica:
Compare performance: recursão vs iteração vs memoização

4. 🔧 Sistema Robusto:
Desenvolva calculadora com tratamento completo de erros

5. 🎯 Projeto Integrador:
Crie biblioteca de funções matemáticas otimizadas

📚 Bibliografia Essencial

📖 Livros Fundamentais

- **Cormen, T. H.** et al. *Introduction to Algorithms*, 4ª ed.
- **Kernighan, B. W.; Ritchie, D. M.** *The C Programming Language*, 2ª ed.
- **Sedgewick, R.** *Algorithms in C*, 3ª ed.
- **Knuth, D. E.** *The Art of Computer Programming*, Vol. 1

🌐 Recursos Online

- **MIT OpenCourseWare:** Estruturas de Dados
- **Coursera:** Algoritmos Especializados
- **LeetCode:** Prática de Implementação

📞 Contato e Suporte

Prof. Vagner Cordeiro

📧 Email: **[email do professor]**

🕐 Atendimento: **[horários de atendimento]**

💻 Material Completo: **github.com/cordeirotelecom/algoritimos_e_complexidade**

🎯 Próxima Aula: **Estruturas de Dados - Arrays, Ponteiros e Structs**

🎉 Obrigado!

💬 Dúvidas e Discussões?

🎓 Algoritmos e Complexidade - Aula 01

Funções e Passagem de Parâmetros

🎯 **Objetivo Alcançado:** Base sólida para estruturas de dados avançadas!

2. Linguagens de Programação

Comparação: C vs Python

Aspecto	C	Python
Paradigma	Procedural	Multi-paradigma
Compilação	Compilada	Interpretada
Tipagem	Estática	Dinâmica
Gerência Memória	Manual	Automática
Performance	Alta	Moderada

Exemplo Comparativo: Função Factorial

Em C:

```c
#include <stdio.h>

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

int main() {
    printf("%lld\n", factorial(5));
    return 0;
}
```

Em Python:

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

def main():
    print(factorial(5))

if __name__ == "__main__":
    main()
```

3. Funções: Conceitos Matemáticos

Definição Formal de Função

Uma função $f : A \to B$ é uma relação que associa:

- Cada elemento $a \in A$ (domínio)
- A exatamente um elemento $b \in B$ (contradomínio)

$$f(a) = b \text{ onde } a \text{ é argumento e } b \text{ é valor de retorno}$$

Propriedades Matemáticas

1. **Injetividade**

$$f \text{ é } \text{injetiva se } f(x_1) = f(x_2) \Rightarrow x_1 = x_2$$

2. **Sobrejetividade**

$$f \text{ é } \text{sobrejetiva se } \forall b \in B, \exists a \in A : f(a) = b$$

3. **Bijetividade**

$$f \text{ é } \text{bijetiva se é } \text{injetiva e sobrejetiva}$$

4. Implementação de Funções em C

Estrutura Básica

```c
tipo_retorno nome_funcao(lista_parametros) {
    // Corpo da função
    return valor;
}
```

Exemplo: Função Potência

```c
double potencia(double base, int expoente) {
    double resultado = 1.0;
    for (int i = 0; i < expoente; i++) {
        resultado *= base;
    }
    return resultado;
}
```

Análise Matemática da Função Potência

Complexidade Temporal

$$T(n) = \Theta(n) \text{ onde } n \text{ é o expoente}$$

Versão Otimizada (Exponenciação Rápida)

```
double potencia_rapida(double base, int exp) {
    if (exp == 0) return 1.0;
    if (exp % 2 == 0) {
        double temp = potencia_rapida(base, exp/2);
        return temp * temp;
    }
    return base * potencia_rapida(base, exp-1);
}
```

Complexidade: $T(n) = O(\log n)$

5. Passagem de Parâmetros

Tipos de Passagem

1. Por Valor (Call by Value)
2. Por Referência (Call by Reference)
3. Por Ponteiro (Call by Pointer)

Passagem por Valor

Conceito

- Cópia do valor é enviada para a função
- Modificações não afetam a variável original

```
void incrementa_valor(int x) {
    x++;  // Não modifica a variável original
}

int main() {
    int num = 5;
    incrementa_valor(num);
    printf("%d\n", num);  // Saída: 5
    return 0;
}
```

Passagem por Ponteiro

Conceito

- Endereço da variável é passado
- Permite modificação da variável original

```c
void incrementa_ponteiro(int *x) {
    (*x)++;  // Modifica a variável original
}

int main() {
    int num = 5;
    incrementa_ponteiro(&num);
    printf("%d\n", num);  // Saída: 6
    return 0;
}
```

Análise Matemática: Custo de Passagem

Por Valor

$$\text{Custo} = O(\text{tamanho\_tipo})$$

Por Ponteiro

$$\text{Custo} = O(1)$$

Para estruturas grandes:

$$\text{sizeof(struct)} >> \text{sizeof(ponteiro)}$$

6. Funções com Arrays

Passagem de Arrays em C

```c
// Array sempre passado por referência
void processa_array(int arr[], int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        arr[i] *= 2;  // Modifica array original
    }
}

int main() {
    int numeros[5] = {1, 2, 3, 4, 5};
    processa_array(numeros, 5);
    // Array foi modificado
    return 0;
}
```

Função para Soma de Array

Implementação Matemática

$$\text{soma}(A) = \sum_{i=0}^{n-1} A[i]$$

```c
int soma_array(int arr[], int n) {
    int soma = 0;
    for (int i = 0; i < n; i++) {
        soma += arr[i];
    }
    return soma;
}
```

Complexidade: $T(n) = \Theta(n)$

7. Recursão: Definição Matemática

Função Recursiva

Uma função f é recursiva se:

$$f(n) = \begin{cases} \text{caso base} & \text{se } n \leq k \\ g(n, f(h(n))) & \text{se } n > k \end{cases}$$

Onde $h(n) < n$ (convergência garantida)

Exemplo: Sequência de Fibonacci

Definição Matemática

$$F(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ F(n-1) + F(n-2) & \text{se } n > 1 \end{cases}$$

Implementação Recursiva

```
long long fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Análise de Complexidade do Fibonacci

Complexidade Recursiva Simples

$$T(n) = T(n-1) + T(n-2) + O(1)$$

$$T(n) = O(\phi^n) \text{ onde } \phi = \frac{1+\sqrt{5}}{2}$$

Versão Otimizada (Programação Dinâmica)

```
long long fibonacci_dp(int n) {
    if (n <= 1) return n;
    long long a = 0, b = 1, temp;
    for (int i = 2; i <= n; i++) {
        temp = a + b;
        a = b;
        b = temp;
    }
    return b;
}
```

Complexidade: $T(n) = O(n)$

8. Escopo de Variáveis

Tipos de Escopo em C

1. **Global** - Visível em todo o programa
2. **Local** - Visível apenas na função
3. **Bloco** - Visível apenas no bloco `{}`
4. **Estático** - Persiste entre chamadas

Exemplo de Escopo

```c
int global = 10;  // Escopo global

void funcao() {
    static int contador = 0;  // Estático
    int local = 5;            // Local

    contador++;
    printf("Contador: %d\n", contador);

    {
        int bloco = 3;  // Escopo de bloco
        printf("Bloco: %d\n", bloco);
    }
    // bloco não existe aqui
}
```

9. Ponteiros para Funções

Conceito

Ponteiros podem apontar para funções, permitindo:

- Passagem de funções como parâmetros
- Arrays de funções
- Implementação de callbacks

```c
// Declaração de ponteiro para função
int (*operacao)(int, int);

int soma(int a, int b) { return a + b; }
int mult(int a, int b) { return a * b; }

operacao = soma;
int resultado = operacao(5, 3);  // 8
```

Exemplo: Calculadora com Ponteiros

```c
typedef int (*Operacao)(int, int);

int soma(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int mult(int a, int b) { return a * b; }

void calculadora(int a, int b, Operacao op) {
    printf("Resultado: %d\n", op(a, b));
}

int main() {
    calculadora(10, 5, soma);  // 15
    calculadora(10, 5, sub);   // 5
    calculadora(10, 5, mult);  // 50
    return 0;
}
```

10. Funções de Ordem Superior

Conceito Matemático

Função que recebe outras funções como parâmetros:

$$H : (A \to B) \times A \to B$$

Exemplo: Map Function

```
void map(int arr[], int n, int (*func)(int)) {
    for (int i = 0; i < n; i++) {
        arr[i] = func(arr[i]);
    }
}

int quadrado(int x) { return x * x; }

int main() {
    int nums[5] = {1, 2, 3, 4, 5};
    map(nums, 5, quadrado);
    // nums agora é {1, 4, 9, 16, 25}
    return 0;
}
```

11. Análise de Performance

Medição de Tempo em C

```c
#include <time.h>

clock_t inicio = clock();
// Código a ser medido
clock_t fim = clock();

double tempo = ((double)(fim - inicio)) / CLOCKS_PER_SEC;
printf("Tempo: %f segundos\n", tempo);
```

Comparação de Algoritmos

Exemplo: Busca Linear vs Binária

```
// Busca Linear: O(n)
int busca_linear(int arr[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) return i;
    }
    return -1;
}

// Busca Binária: O(log n)
int busca_binaria(int arr[], int l, int r, int x) {
    if (r >= l) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x) return mid;
        if (arr[mid] > x) return busca_binaria(arr, l, mid-1, x);
        return busca_binaria(arr, mid+1, r, x);
    }
    return -1;
}
```

12. Tratamento de Erros

Códigos de Retorno

```c
typedef enum {
    SUCCESS = 0,
    ERROR_NULL_POINTER = -1,
    ERROR_INVALID_INPUT = -2,
    ERROR_OUT_OF_BOUNDS = -3
} ErrorCode;

ErrorCode divisao_segura(double a, double b, double *resultado) {
    if (resultado == NULL) return ERROR_NULL_POINTER;
    if (b == 0.0) return ERROR_INVALID_INPUT;

    *resultado = a / b;
    return SUCCESS;
}
```

13. Otimização de Funções

Técnicas de Otimização

1. **Memoização** - Cache de resultados
2. **Tail Recursion** - Recursão de cauda
3. **Loop Unrolling** - Desenrolamento de loops
4. **Inline Functions** - Funções inline

Exemplo: Memoização em Fibonacci

```c
#define MAX_N 100
long long memo[MAX_N];
int inicializado = 0;

long long fibonacci_memo(int n) {
    if (!inicializado) {
        for (int i = 0; i < MAX_N; i++) memo[i] = -1;
        inicializado = 1;
    }

    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];

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

14. Comparação com Python

Vantagens do C:

- **Performance:** 10-100x mais rápido
- **Controle de memória:** Gestão manual
- **Previsibilidade:** Comportamento determinístico

Vantagens do Python:

- **Produtividade:** Desenvolvimento mais rápido
- **Expressividade:** Código mais conciso
- **Bibliotecas:** Ecossistema rico

Exemplo Comparativo: Quick Sort

Python (Simplicidade)

```python
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)
```

C (Performance)

```c
void quicksort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            trocar(&arr[i], &arr[j]);
        }
    }
    trocar(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

15. Boas Práticas

Nomenclatura de Funções

- **Verbos** para ações: `calcular()`, `processar()`
- **Nomes descritivos:** `calcular_media()` vs `calc()`
- **Consistência:** `get_` e `set_` para acessores

Documentação

```
/**
 * Calcula o fatorial de um número
 * @param n: número inteiro não negativo
 * @return: fatorial de n, ou -1 se n < 0
 * Complexidade: O(n)
 */
long long fatorial(int n);
```

16. Debugging e Testes

Uso de Assertions

```c
#include <assert.h>

int divisao(int a, int b) {
    assert(b != 0);  // Garante que b não é zero
    return a / b;
}
```

Função de Teste

```c
void testar_funcoes() {
    assert(fatorial(5) == 120);
    assert(fibonacci(10) == 55);
    assert(potencia(2, 3) == 8);
    printf("Todos os testes passaram!\n");
}
```

17. Considerações de Memória

Stack vs Heap

Stack (Pilha):

- Variáveis locais
- Parâmetros de função
- Endereços de retorno
- Limitado em tamanho

Heap (Monte):

- Alocação dinâmica
- `malloc()`, `free()`
- Maior flexibilidade
- Gerenciamento manual

Exemplo: Função com Alocação Dinâmica

```c
int* criar_array(int tamanho) {
    int *arr = malloc(tamanho * sizeof(int));
    if (arr == NULL) {
        printf("Erro de alocação!\n");
        return NULL;
    }

    for (int i = 0; i < tamanho; i++) {
        arr[i] = i * i;  // Inicializa com quadrados
    }

    return arr;
}

void liberar_array(int *arr) {
    free(arr);
}
```

18. Preprocessador e Macros

Definindo Constantes

```c
#define PI 3.14159265359
#define MAX_SIZE 1000
```

Macros Funcionais

```c
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define SQUARE(x) ((x) * (x))

int maior = MAX(10, 20);  // 20
int quadrado = SQUARE(5); // 25
```

19. Estruturas de Controle Avançadas

Switch com Funções

```c
typedef enum { SOMA, SUB, MULT, DIV } Operador;

double calcular(double a, double b, Operador op) {
    switch (op) {
        case SOMA: return a + b;
        case SUB:  return a - b;
        case MULT: return a * b;
        case DIV:  return (b != 0) ? a / b : 0;
        default:   return 0;
    }
}
```

20. Conclusões e Próximos Passos

O que Aprendemos:

- **Conceitos matemáticos** de algoritmos e funções
- **Implementação** de funções em C
- **Mecanismos** de passagem de parâmetros
- **Análise** de complexidade e performance
- **Boas práticas** de programação

Próxima Aula:

- **Estruturas de Dados** homogêneas e heterogêneas
- **Arrays multidimensionais**
- **Ponteiros avançados**
- **Structs e Unions**

Exercícios Propostos

1. Implemente uma função que calcule x^n em $O(\log n)$
2. Crie uma função genérica de ordenação usando ponteiros
3. Implemente memoização para a sequência de Fibonacci
4. Compare performance entre recursão e iteração
5. Desenvolva um sistema de tratamento de erros robusto

Bibliografia

- **Cormen, T. H.** et al. *Introduction to Algorithms*, 4ª ed.
- **Kernighan, B. W.; Ritchie, D. M.** *The C Programming Language*, 2ª ed.
- **Sedgewick, R.** *Algorithms in C*, 3ª ed.
- **Knuth, D. E.** *The Art of Computer Programming*, Vol. 1

Contato e Dúvidas

Prof. Vagner Cordeiro

📧 **Email:** [email do professor]

🕐 **Atendimento:** [horários de atendimento]

📚 **Material:** github.com/cordeirotelecom/algoritimos_e_complexidade

Próxima aula: Estruturas de Dados - Arrays, Ponteiros e Structs

Obrigado!

Perguntas?

Algoritmos e Complexidade - Aula 01

Funções e Passagem de Parâmetros