# Algoritmos e Complexidade

## Aula 01: Algoritmos - Funções e Passagem de Parâmetros

**Prof. Vagner Cordeiro**
**Sistemas de Informação**
**Universidade - 2024**

## Agenda da Aula

1. **Conceitos Fundamentais de Algoritmos**
2. **Definições Matemáticas e Formais**
3. **Linguagens de Programação: C vs Python**
4. **Funções e Modularização**
5. **Passagem de Parâmetros**
6. **Escopo e Contexto**
7. **Exemplos Práticos e Implementações**
8. **Análise de Performance**

## Objetivos de Aprendizagem

Ao final desta aula, o aluno será capaz de:

- **Definir** algoritmos de forma matemática e computacional
- **Implementar** funções em C e Python
- **Compreender** mecanismos de passagem de parâmetros
- **Aplicar** conceitos de escopo e modularização
- **Analisar** complexidade de funções simples
- **Comparar** abordagens entre linguagens

# 1. Conceitos Fundamentais

## Definição Matemática de Algoritmo

Um **algoritmo** é uma sequência finita de instruções bem definidas que:

$$A : D \to C$$

Onde:

- $D$ = Domínio (conjunto de entradas válidas)
- $C$ = Contradomínio (conjunto de saídas possíveis)
- $A$ = Função algorítmica

## Propriedades Fundamentais

**1.** Finitude

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

**2.** Determinismo

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

**3.** Efetividade

$$\text{Cada instruç ã o deve ser executá vel em tempo finito}$$

## Notação Big-O para Complexidade

### Definição Formal

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

### Hierarquia de Complexidade

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$$

## 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

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

**2.** Sobrejetividade

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

**3.** Bijetividade

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

## 4. Implementação de Funções em C

### Estrutura Básica

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

### Exemplo: Função Potência

```
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

```c
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

```c
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

```c
// 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

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

### Macros Funcionais

```
#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*