

EXERCÍCIOS RESOLVIDOS - Algoritmos e Complexidade

Lista Completa com Soluções Detalhadas

SEÇÃO 1: ANÁLISE DE COMPLEXIDADE

Exercício 1.1: Análise Básica

Questão: Determine a complexidade dos seguintes códigos:

a) Código A:

```
int somar_pares(int arr[], int n) {  
    int soma = 0;                // 1 operação  
    for (int i = 0; i < n; i++) { // n iterações  
        if (arr[i] % 2 == 0) {    // 1 operação por iteração  
            soma += arr[i];       // 1 operação (quando executa)  
        }  
    }  
    return soma;                // 1 operação  
}
```

Solução A:

a. Inicialização: 1 operação

Exercício 1.2: Análise Avançada

Questão: Analise a complexidade do algoritmo de busca ternária:

```
int busca_ternaria(int arr[], int esq, int dir, int valor) {
    if (dir >= esq) {
        int meio1 = esq + (dir - esq) / 3;
        int meio2 = dir - (dir - esq) / 3;

        if (arr[meio1] == valor)
            return meio1;
        if (arr[meio2] == valor)
            return meio2;

        if (valor < arr[meio1])
            return busca_ternaria(arr, esq, meio1 - 1, valor);
        else if (valor > arr[meio2])
            return busca_ternaria(arr, meio2 + 1, dir, valor);
        else
            return busca_ternaria(arr, meio1 + 1, meio2 - 1, valor);
    }
    return -1;
}
```

1234 SEÇÃO 2: ALGORITMOS DE ORDENAÇÃO

Exercício 2.1: Implementação e Análise

Questão: Implemente o Selection Sort e analise suas operações:

```
void selection_sort_detalhado(int arr[], int n) {
    int comparacoes = 0, trocas = 0;

    for (int i = 0; i < n-1; i++) {
        int indice_minimo = i;

        // Encontrar o menor elemento no restante do array
        for (int j = i+1; j < n; j++) {
            comparacoes++;
            if (arr[j] < arr[indice_minimo]) {
                indice_minimo = j;
            }
        }

        // Trocar se necessário
        if (indice_minimo != i) {
            int temp = arr[i];
            arr[i] = arr[indice_minimo];
            arr[indice_minimo] = temp;
            trocas++;
        }
    }
}
```

Exercício 2.2: Comparação Prática

Questão: Compare Bubble Sort otimizado vs Insertion Sort:

```
// Bubble Sort com otimização early stop
void bubble_sort_otimizado(int arr[], int n) {
    int comparacoes = 0, trocas = 0;

    for (int i = 0; i < n-1; i++) {
        int houve_troca = 0;

        for (int j = 0; j < n-i-1; j++) {
            comparacoes++;
            if (arr[j] > arr[j+1]) {
                // Trocar
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                trocas++;
                houve_troca = 1;
            }
        }

        // Se não houve troca, array está ordenado
        if (!houve_troca) {
            printf("Array ordenado na iteração %d\n", i+1);
            break;
        }
    }

    printf("Bubble Sort - Comparações: %d, Trocas: %d\n", comparacoes, trocas);
}

// Insertion Sort detalhado
void insertion_sort_detalhado(int arr[], int n) {
    int comparacoes = 0, trocas = 0;

    for (int i = 1; i < n; i++) {
        int chave = arr[i];
        int j = i - 1;

        // Mover elementos maiores que chave uma posição à frente
        while (j >= 0) {
            comparacoes++;
            if (arr[j] > chave) {
                arr[j+1] = arr[j];
                j--;
                trocas++;
            } else {
                break;
            }
        }
        arr[j+1] = chave;
    }
}
```



SEÇÃO 3: ESTRUTURAS DE DADOS

Exercício 3.1: Implementação de Pilha

Questão: Implemente uma pilha com histórico de operações e análise de uso:

```
#define MAX_SIZE 100
#define MAX_HISTORICO 1000

typedef struct {
    char operacao[20];
    int valor;
    time_t timestamp;
} OperacaoHistorico;

typedef struct {
    int dados[MAX_SIZE];
    int topo;
    OperacaoHistorico historico[MAX_HISTORICO];
    int total_operacoes;
    int max_tamanho_attingido;
} PilhaComHistorico;

// Inicializar pilha
void inicializar_pilha(PilhaComHistorico* p) {
    p->topo = -1;
    p->total_operacoes = 0;
    p->max_tamanho_attingido = 0;
}

// Push com logging
int push_com_log(PilhaComHistorico* p, int valor) {
    if (p->topo >= MAX_SIZE - 1) {
        printf("Erro: Pilha cheia!\n");
        return 0;
    }

    p->dados[++p->topo] = valor;

    // Atualizar estatísticas
    if (p->topo + 1 > p->max_tamanho_attingido) {
        p->max_tamanho_attingido = p->topo + 1;
    }

    // Adicionar ao histórico
    if (p->total_operacoes < MAX_HISTORICO) {
        strcpy(p->historico[p->total_operacoes].operacao, "PUSH");
        p->historico[p->total_operacoes].valor = valor;
        p->historico[p->total_operacoes].timestamp = time(NULL);
        p->total_operacoes++;
    }

    return 1; // Sucesso
}

// Pop com logging
int pop_com_log(PilhaComHistorico* p) {
    if (p->topo < 0) {
        printf("Erro: Pilha vazia!\n");
        return INT_MIN; // Valor especial para erro
    }

    int valor = p->dados[p->topo--];

    // Adicionar ao histórico
    if (p->total_operacoes < MAX_HISTORICO) {
        strcpy(p->historico[p->total_operacoes].operacao, "POP");
        p->historico[p->total_operacoes].valor = valor;
        p->historico[p->total_operacoes].timestamp = time(NULL);
        p->total_operacoes++;
    }

    return valor;
}

// Análise de uso da pilha
void analisar_pilha(PilhaComHistorico* p) {
    printf("=== ANÁLISE DE USO DA PILHA ===\n");
    printf("Total de operações: %d\n", p->total_operacoes);
    printf("Tamanho atual: %d\n", p->topo + 1);
    printf("Máximo tamanho atingido: %d\n", p->max_tamanho_attingido);
    printf("Utilização máxima: %.2f%%\n",
           (p->max_tamanho_attingido * 100.0) / MAX_SIZE);
}

// Contar operações
int pushes = 0, pops = 0;
for (int i = 0; i < p->total_operacoes; i++) {
    if (strcmp(p->historico[i].operacao, "PUSH") == 0) {
        pushes++;
    }
}
```

Exercício 3.2: Lista Ligada vs Array - Comparação Prática

Questão: Compare inserção de 1000 elementos no início da estrutura:

```
#include <time.h>
#include <stdlib.h>

// Estrutura para lista ligada
typedef struct No {
    int dados;
    struct No* proximo;
} No;

typedef struct {
    No* cabeca;
    int tamanho;
} ListaLigada;

// Array dinâmico
typedef struct {
    int* dados;
    int tamanho;
    int capacidade;
} ArrayDinamico;

// Implementações da Lista Ligada
void init_lista(ListaLigada* lista) {
    lista->cabeca = NULL;
    lista->tamanho = 0;
}

void inserir_inicio_lista(ListaLigada* lista, int valor) {
    No* novo = malloc(sizeof(No));
    novo->dados = valor;
    novo->proximo = lista->cabeca;
    lista->cabeca = novo;
    lista->tamanho++;
}

// Implementações do Array Dinâmico
void init_array(ArrayDinamico* arr) {
    arr->dados = malloc(10 * sizeof(int));
    arr->tamanho = 0;
    arr->capacidade = 10;
}

void inserir_inicio_array(ArrayDinamico* arr, int valor) {
    // Verificar se precisa expandir
    if (arr->tamanho >= arr->capacidade) {
        arr->capacidade *= 2;
        arr->dados = realloc(arr->dados, arr->capacidade * sizeof(int));
    }

    // Mover todos os elementos uma posição à frente
    for (int i = arr->tamanho; i > 0; i--) {
        arr->dados[i] = arr->dados[i-1];
    }

    arr->dados[0] = valor;
    arr->tamanho++;
}

// Teste de performance
void teste_performance() {
    const int N = 1000;

    // Teste Lista Ligada
    ListaLigada lista;
    init_lista(&lista);

    clock_t inicio = clock();
    for (int i = 0; i < N; i++) {
        inserir_inicio_lista(&lista, i);
    }
    clock_t fim = clock();
    double tempo_lista = ((double)(fim - inicio)) / CLOCKS_PER_SEC;

    // Teste Array Dinâmico
    ArrayDinamico arr;
    init_array(&arr);

    inicio = clock();
    for (int i = 0; i < N; i++) {
        inserir_inicio_array(&arr, i);
    }
    fim = clock();
    double tempo_array = ((double)(fim - inicio)) / CLOCKS_PER_SEC;

    // Resultados
    printf("=== TESTE DE PERFORMANCE ===\n");
    printf("Inserções no início: %d elementos\n", N);
    printf("Lista Ligada: %.6f segundos\n", tempo_lista);
    printf("Array Dinâmico: %.6f segundos\n", tempo_array);
    printf("Lista é %.2fx mais rápida\n", tempo_array / tempo_lista);

    // Análise de memória
    int memoria_lista = lista.tamanho * sizeof(No);
    int memoria_array = arr.capacidade * sizeof(int);

    printf("\n=== USO DE MEMORIA ===\n");
    printf("Lista Ligada: %d bytes (%d nós)\n", memoria_lista, lista.tamanho);
    printf("Array Dinâmico: %d bytes (%d capacidade)\n", memoria_array, arr.capacidade);
}
```

SEÇÃO 4: ALGORITMOS DE BUSCA

Exercício 4.1: Busca Linear vs Binária

Questão: Implemente e compare ambos algoritmos com análise estatística:

```
typedef struct {
    int comparacoes;
    double tempo_execucao;
    int encontrado;
    int posicao;
} ResultadoBusca;

// Busca Linear com estatísticas
ResultadoBusca busca_linear_stats(int arr[], int n, int valor) {
    ResultadoBusca resultado = {0, 0.0, 0, -1};

    clock_t inicio = clock();

    for (int i = 0; i < n; i++) {
        resultado.comparacoes++;
        if (arr[i] == valor) {
            resultado.encontrado = 1;
            resultado.posicao = i;
            break;
        }
    }

    clock_t fim = clock();
    resultado.tempo_execucao = ((double)(fim - inicio) / CLOCKS_PER_SEC);

    return resultado;
}

// Busca Binária com estatísticas
ResultadoBusca busca_binaria_stats(int arr[], int n, int valor) {
    ResultadoBusca resultado = {0, 0.0, 0, -1};

    clock_t inicio = clock();

    int esq = 0, dir = n - 1;

    while (esq <= dir) {
        resultado.comparacoes++;
        int meio = esq + (dir - esq) / 2;

        if (arr[meio] == valor) {
            resultado.encontrado = 1;
            resultado.posicao = meio;
            break;
        }

        resultado.comparacoes++;
        if (arr[meio] < valor) {
            esq = meio + 1;
        } else {
            dir = meio - 1;
        }
    }

    clock_t fim = clock();
    resultado.tempo_execucao = ((double)(fim - inicio) / CLOCKS_PER_SEC);

    return resultado;
}

// Teste comparativo
void teste_comparativo_busca() {
    // Cria array ordenado de 10000 elementos
    const int N = 10000;
    int arr = malloc(N * sizeof(int));
    for (int i = 0; i < N; i++) {
        arr[i] = i + 1; // Números pares: 0, 2, 4, 6, ...
    }

    // Teste 1: Elemento no início
    printf("=== TESTE 1: Buscar 3d (inicio) ===\n", arr[0]);
    ResultadoBusca linear1 = busca_linear_stats(arr, N, arr[0]);
    ResultadoBusca binaria1 = busca_binaria_stats(arr, N, arr[0]);

    printf("Linear: 3d comparacoes, posicao 3d\n",
           linear1.comparacoes, linear1.posicao);
    printf("Binaria: 3d comparacoes, posicao 3d\n",
           binaria1.comparacoes, binaria1.posicao);

    // Teste 2: Elemento no meio
    int meio = N / 2;
    printf("=== TESTE 2: Buscar 3d (meio) ===\n", arr[meio]);
    ResultadoBusca linear2 = busca_linear_stats(arr, N, arr[meio]);
    ResultadoBusca binaria2 = busca_binaria_stats(arr, N, arr[meio]);

    printf("Linear: 3d comparacoes, posicao 3d\n",
           linear2.comparacoes, linear2.posicao);
    printf("Binaria: 3d comparacoes, posicao 3d\n",
           binaria2.comparacoes, binaria2.posicao);

    // Teste 3: Elemento no final
    printf("=== TESTE 3: Buscar 3d (final) ===\n", arr[N-1]);
    ResultadoBusca linear3 = busca_linear_stats(arr, N, arr[N-1]);
    ResultadoBusca binaria3 = busca_binaria_stats(arr, N, arr[N-1]);

    printf("Linear: 3d comparacoes, posicao 3d\n",
           linear3.comparacoes, linear3.posicao);
    printf("Binaria: 3d comparacoes, posicao 3d\n",
           binaria3.comparacoes, binaria3.posicao);

    // Teste 4: Elemento não existe
```




Exercício 5.1: Sistema de Cache LRU (Least Recently Used)

Questão: Implemente um cache LRU para simular cache de páginas web:

[illegible]

RESUMO DE COMPLEXIDADES DOS EXERCÍCIOS

Exercício	Algoritmo/Estrutura	Complexidade	Aplicação Prática
1.1	Análise básica	$O(n)$, $O(n^2)$	Fundamentos
1.2	Busca ternária	$O(\log n)$	Busca otimizada
2.1	Selection Sort	$O(n^2)$	Ordenação simples
2.2	Bubble vs Insertion	$O(n^2)$	Comparação de algoritmos
3.1	Pilha com histórico	$O(1)$ por operação	Calculadora RPN
3.2	Lista vs Array	$O(1)$ vs $O(n)$	Estruturas dinâmicas
4.1	Busca linear vs binária	$O(n)$ vs $O(\log n)$	Busca eficiente
5.1	Cache LRU	$O(1)$ amortizado	Sistema de cache


DICAS PARA RESOLUÇÃO

Estratégias Gerais

1. **Identifique o padrão:** Linear, quadrático, logarítmico?
2. **Conte operações básicas:** Comparações, atribuições, acessos
3. **Analise loops:** Simples = $O(n)$, aninhados = $O(n^2)$
4. **Recursão:** Monte a equação de recorrência
5. **Teste com valores pequenos** antes de generalizar

Debugging de Complexidade

1. Use contadores para operações
2. Meça tempo real para validar teoria
3. Teste diferentes tamanhos de entrada
4. Compare com complexidades conhecidas

 Esta lista de exercícios cobre os principais conceitos de algoritmos e complexidade com soluções detalhadas e análises práticas.

Última atualização: 27 de agosto de 2025