Algoritmos e Complexidade

Aula 03: Algoritmos de Ordenação e Análise de Performance

Prof. Vagner Cordeiro
Sistemas de Informação
Universidade - 2024

Agenda da Aula

1. **Fundamentos Matemáticos da Ordenação**
2. **Algoritmos de Ordenação Elementares**
3. **Algoritmos Avançados: Divide-and-Conquer**
4. **Análise Comparativa de Performance**
5. **Algoritmos de Ordenação Especializados**
6. **Otimizações e Técnicas Avançadas**
7. **Implementações Práticas e Benchmarks**
8. **Aplicações Reais e Casos de Uso**

Objetivos de Aprendizagem

Ao final desta aula, o estudante será capaz de:

Fundamentos Teóricos:

- **Definir** matematicamente o problema de ordenação e suas variantes
- **Analisar** complexidade de tempo e espaço de diferentes algoritmos
- **Demonstrar** limites teóricos inferiores para comparação-based sorting

Implementação Prática:

- **Implementar** algoritmos de ordenação clássicos em C
- **Otimizar** algoritmos para diferentes cenários e tipos de dados
- **Avaliar** performance empírica através de benchmarks rigorosos

Aplicações Avançadas:

- **Selecionar** algoritmos apropriados para contextos específicos
- **Projetar** soluções híbridas combinando múltiplas técnicas
- **Resolver** problemas complexos usando ordenação como subrotina

1. Fundamentos Matemáticos da Ordenação

Definição Formal do Problema

Entrada: Sequência $A = \langle a_1, a_2, \ldots, a_n \rangle$ de n elementos
Saída: Permutação $A' = \langle a'_1, a'_2, \ldots, a'_n \rangle$ tal que:

$$a'_1 \leq a'_2 \leq \ldots \leq a'_n$$

Propriedades Matemáticas Essenciais

Relação de Ordem Total:
Para qualquer conjunto S com relação $\leq$:

- **Reflexividade:** $a \leq a$
- **Antissimetria:** $a \leq b \land b \leq a \Rightarrow a = b$
- **Transitividade:** $a \leq b \land b \leq c \Rightarrow a \leq c$
- **Totalidade:** $\forall a, b \in S : a \leq b \lor b \leq a$

Invariantes de Ordenação:

- Preservação de elementos (sem perda ou adição)
- Manutenção da relação de ordem estabelecida
- Estabilidade (quando aplicável)

Análise de Complexidade: Limites Teóricos

Limite Inferior para Algoritmos Baseados em Comparação

Teorema: Qualquer algoritmo de ordenação baseado em comparações requer $\Omega(n \log n)$ comparações no pior caso.

Demonstração (Árvore de Decisão):

- Existem $n!$ permutações possíveis
- Cada comparação divide o espaço de possibilidades em no máximo 2 partes
- Altura mínima da árvore: $\lceil \log_2(n!) \rceil$
- Pela aproximação de Stirling: $\log_2(n!) = \Theta(n \log n)$

$$\log_2(n!) \geq \log_2\left(\left(\frac{n}{e}\right)^n\right) = n \log_2\left(\frac{n}{e}\right) = \Omega(n \log n)$$

Classificação por Complexidade

Classe	Complexidade	Algoritmos
Quadrática	$O(n^2)$	Bubble, Selection, Insertion
Linearítmica	$O(n \log n)$	Merge, Heap, Quick (avg)
Linear	$O(n)$	Counting, Radix, Bucket
Sublinear	$O(\log n)$	Binary Search (busca)

2. Algoritmos de Ordenação Elementares

Bubble Sort: Análise Matemática Completa

Princípio: Comparações adjacentes com "borbulhamento" do maior elemento

```c
void bubble_sort(int array[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int houve_troca = 0;  // Otimização para detecção precoce

        for (int j = 0; j < n - i - 1; j++) {
            if (array[j] > array[j + 1]) {
                // Troca elementos adjacentes
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
                houve_troca = 1;
            }
        }

        // Se não houve trocas, array já está ordenado
        if (!houve_troca) break;
    }
}
```

Análise de Complexidade:

- **Melhor caso:** $T(n) = O(n)$ - array já ordenado
- **Caso médio:** $T(n) = O(n^2)$ - ordem aleatória
- **Pior caso:** $T(n) = O(n^2)$ - ordem reversa
- **Espaço:** $S(n) = O(1)$ - in-place

Selection Sort: Busca do Mínimo Iterativa

Princípio: Seleciona o menor elemento e coloca na posição correta

```c
void selection_sort(int array[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int indice_minimo = i;

        // Encontra o menor elemento no subarray não ordenado
        for (int j = i + 1; j < n; j++) {
            if (array[j] < array[indice_minimo]) {
                indice_minimo = j;
            }
        }

        // Troca apenas se necessário (otimização)
        if (indice_minimo != i) {
            int temp = array[i];
            array[i] = array[indice_minimo];
            array[indice_minimo] = temp;
        }
    }
}
```

Vantagens:

- Número mínimo de trocas: exatamente $n-1$ trocas
- Performance constante independente da entrada
- Simples de implementar e entender

Análise: Sempre $O(n^2)$ comparações, mas apenas $O(n)$ trocas

Insertion Sort: Construção Incremental

Princípio: Constrói a solução inserindo elementos na posição correta

```c
void insertion_sort(int array[], int n) {
    for (int i = 1; i < n; i++) {
        int chave = array[i];
        int j = i - 1;

        // Move elementos maiores que a chave uma posição à frente
        while (j >= 0 && array[j] > chave) {
            array[j + 1] = array[j];
            j--;
        }

        // Insere a chave na posição correta
        array[j + 1] = chave;
    }
}
```

Características Especiais:

- **Adaptativo:** Eficiente para dados quase ordenados
- **Estável:** Mantém ordem relativa de elementos iguais
- **Online:** Pode ordenar dados conforme chegam
- **Eficiente para pequenos arrays:** Melhor que $O(n \log n)$ para $n < 50$

Análise Detalhada:

- **Melhor caso:** $T(n) = O(n)$ - já ordenado
- **Pior caso:** $T(n) = O(n^2)$ - ordem reversa
- **Número médio de movimentos:** n^2

3. Algoritmos Avançados: Divide-and-Conquer

Merge Sort: Paradigma Fundamental

Princípio: Divide o problema, resolve recursivamente e combina soluções

```
void merge(int array[], int esquerda, int meio, int direita) {
    int n1 = meio - esquerda + 1;
    int n2 = direita - meio;

    // Arrays temporários
    int L[n1], R[n2];

    // Copia dados para arrays temporários
    for (int i = 0; i < n1; i++)
        L[i] = array[esquerda + i];
    for (int j = 0; j < n2; j++)
        R[j] = array[meio + 1 + j];

    // Merge dos arrays temporários de volta no array original
    int i = 0, j = 0, k = esquerda;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            array[k] = L[i];
            i++;
        } else {
            array[k] = R[j];
            j++;
        }
        k++;
    }

    // Copia elementos restantes
    while (i < n1) {
        array[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        array[k] = R[j];
        j++;
        k++;
    }
}

void merge_sort(int array[], int esquerda, int direita) {
    if (esquerda < direita) {
        int meio = esquerda + (direita - esquerda) / 2;

        // Recursão nas metades
        merge_sort(array, esquerda, meio);
        merge_sort(array, meio + 1, direita);
```

Quick Sort: Algoritmo de Particionamento

Princípio: Particiona o array em torno de um pivô

```
int partition(int array[], int baixo, int alto) {
    int pivot = array[alto];  // Último elemento como pivô
    int i = (baixo - 1);      // Índice do menor elemento

    for (int j = baixo; j <= alto - 1; j++) {
        // Se elemento atual é menor ou igual ao pivô
        if (array[j] <= pivot) {
            i++;
            // Troca array[i] e array[j]
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }

    // Troca array[i+1] e array[alto] (ou pivô)
    int temp = array[i + 1];
    array[i + 1] = array[alto];
    array[alto] = temp;

    return (i + 1);
}

void quick_sort(int array[], int baixo, int alto) {
    if (baixo < alto) {
        // Índice de particionamento
        int pi = partition(array, baixo, alto);

        // Recursão nas metades
        quick_sort(array, baixo, pi - 1);
        quick_sort(array, pi + 1, alto);
    }
}
```

Heap Sort: Estrutura de Dados Avançada

Implementação do Heap Binário

Propriedade do Max-Heap: Para todo nó i:

$$\text{parent}(i) \geq A[i]$$

```
void heapify(int array[], int n, int i) {
    int maior = i;          // Inicializa maior como raiz
    int esquerda = 2 * i + 1;  // Filho esquerdo
    int direita = 2 * i + 2;   // Filho direito

    // Se filho esquerdo é maior que raiz
    if (esquerda < n && array[esquerda] > array[maior])
        maior = esquerda;

    // Se filho direito é maior que maior até agora
    if (direita < n && array[direita] > array[maior])
        maior = direita;

    // Se maior não é raiz
    if (maior != i) {
        int temp = array[i];
        array[i] = array[maior];
        array[maior] = temp;

        // Recursivamente heapify a subárvore afetada
        heapify(array, n, maior);
    }
}

void heap_sort(int array[], int n) {
    // Constrói heap (rearranja array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(array, n, i);

    // Extrai elementos do heap um por um
    for (int i = n - 1; i > 0; i--) {
        // Move raiz atual para o final
        int temp = array[0];
        array[0] = array[i];
```

4. Algoritmos de Ordenação Linear

Counting Sort: Ordenação por Contagem

Aplicabilidade: Elementos inteiros em intervalo conhecido $[0, k]$

```c
void counting_sort(int array[], int n, int k) {
    // Array de saída que terá os elementos ordenados
    int output[n];

    // Array de contagem para armazenar count de cada elemento
    int count[k + 1];

    // Inicializa array de contagem com zeros
    for (int i = 0; i <= k; i++)
        count[i] = 0;

    // Armazena a contagem de cada elemento
    for (int i = 0; i < n; i++)
        count[array[i]]++;

    // Modifica count[i] para que contenha posição atual
    // do elemento i no array de saída
    for (int i = 1; i <= k; i++)
        count[i] += count[i - 1];

    // Constrói o array de saída
    for (int i = n - 1; i >= 0; i--) {
        output[count[array[i]] - 1] = array[i];
        count[array[i]]--;
    }

    // Copia o array de saída para array[], para que
    // array[] contenha elementos ordenados
    for (int i = 0; i < n; i++)
        array[i] = output[i];
}
```

Radix Sort: Ordenação por Dígitos

Princípio: Ordena dígito por dígito usando counting sort estável

```c
int obter_maximo(int array[], int n) {
    int max = array[0];
    for (int i = 1; i < n; i++)
        if (array[i] > max)
            max = array[i];
    return max;
}

void counting_sort_radix(int array[], int n, int exp) {
    int output[n];
    int count[10] = {0};

    // Armazena contagem de ocorrências em count[]
    for (int i = 0; i < n; i++)
        count[(array[i] / exp) % 10]++;

    // Modifica count[i] para conter posição atual
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Constrói array de saída
    for (int i = n - 1; i >= 0; i--) {
        output[count[(array[i] / exp) % 10] - 1] = array[i];
        count[(array[i] / exp) % 10]--;
    }

    // Copia array de saída para array[]
    for (int i = 0; i < n; i++)
        array[i] = output[i];
}

void radix_sort(int array[], int n) {
    int max = obter_maximo(array, n);

    // Executa counting sort para cada dígito
    for (int exp = 1; max / exp > 0; exp *= 10)
        counting_sort_radix(array, n, exp);
}
```

5. Otimizações e Algoritmos Híbridos

Introsort: Combinação Inteligente

Princípio: Combina QuickSort, HeapSort e InsertionSort

```
#include <math.h>

void introsort_util(int array[], int baixo, int alto, int limite_profundidade) {
    while (alto > baixo) {
        int tamanho = alto - baixo + 1;

        // Para arrays pequenos, use insertion sort
        if (tamanho < 16) {
            insertion_sort_range(array, baixo, alto);
            break;
        }
        // Se profundidade máxima atingida, use heap sort
        else if (limite_profundidade == 0) {
            heap_sort_range(array, baixo, alto);
            break;
        }
        // Caso contrário, use quick sort
        else {
            int pivot = partition(array, baixo, alto);

            // Otimização: recursão na partição menor
            if (pivot - baixo < alto - pivot) {
                introsort_util(array, baixo, pivot - 1, limite_profundidade - 1);
                baixo = pivot + 1;
            } else {
                introsort_util(array, pivot + 1, alto, limite_profundidade - 1);
                alto = pivot - 1;
            }
            limite_profundidade--;
        }
    }
}

void introsort(int array[], int n) {
```

Timsort: Algoritmo do Python

Princípio: Detecta runs naturais e os mescla eficientemente

Características Principais:

- Adaptativo para dados parcialmente ordenados
- Estável e com performance $O(n \log n)$ garantida
- Otimizado para padrões comuns de dados reais

```c
// Simplificação conceitual do Timsort
typedef struct {
    int base;
    int tamanho;
} Run;

void timsort_simplificado(int array[], int n) {
    const int MIN_MERGE = 32;

    // 1. Identifica ou cria runs mínimos
    for (int i = 0; i < n; i += MIN_MERGE) {
        int fim = (i + MIN_MERGE - 1 < n - 1) ? i + MIN_MERGE - 1 : n - 1;
        insertion_sort_range(array, i, fim);
    }

    // 2. Mescla runs progressivamente
    int tamanho = MIN_MERGE;
    while (tamanho < n) {
        for (int inicio = 0; inicio < n; inicio += tamanho * 2) {
            int meio = inicio + tamanho - 1;
            int fim = (inicio + tamanho * 2 - 1 < n - 1) ?
                      inicio + tamanho * 2 - 1 : n - 1;

            if (meio < fim)
                merge(array, inicio, meio, fim);
        }
```

6. Análise Experimental e Benchmarks

Framework de Testing Rigoroso

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

typedef struct {
    char nome[50];
    void (*algoritmo)(int[], int);
    double tempo_execucao;
    long comparacoes;
    long trocas;
} ResultadoBenchmark;

typedef enum {
    ALEATORIO,
    ORDENADO,
    REVERSO,
    QUASE_ORDENADO,
    MUITAS_REPETICOES
} TipoDados;

void gerar_dados_teste(int array[], int n, TipoDados tipo) {
    switch (tipo) {
        case ALEATORIO:
            for (int i = 0; i < n; i++)
                array[i] = rand() % (n * 10);
            break;

        case ORDENADO:
            for (int i = 0; i < n; i++)
                array[i] = i;
            break;

        case REVERSO:
            for (int i = 0; i < n; i++)
                array[i] = n - i;
            break;

        case QUASE_ORDENADO:
            for (int i = 0; i < n; i++)
                array[i] = i;
            // Faz algumas trocas aleatórias
            for (int i = 0; i < n / 10; i++) {
                int pos1 = rand() % n;
                int pos2 = rand() % n;
                int temp = array[pos1];
                array[pos1] = array[pos2];
                array[pos2] = temp;
            }
            break;

        case MUITAS_REPETICOES:
            for (int i = 0; i < n; i++)
                array[i] = rand() % 10;  // Apenas 10 valores distintos
            break;
    }
}

double medir_tempo_algoritmo(void (*algoritmo)(int[], int),
                            int array[], int n) {
    clock_t inicio = clock();
    algoritmo(array, n);
    clock_t fim = clock();

    return ((double)(fim - inicio)) / CLOCKS_PER_SEC;
}

void executar_benchmark_completo(int tamanhos[], int num_tamanhos) {
    const char* nomes_tipos[] = {"Aleatório", "Ordenado", "Reverso",
                                "Quase Ordenado", "Muitas Repetições"};

    ResultadoBenchmark algoritmos[] = {
        {"Bubble Sort", bubble_sort, 0, 0, 0},
        {"Selection Sort", selection_sort, 0, 0, 0},
        {"Insertion Sort", insertion_sort, 0, 0, 0},
        {"Merge Sort", merge_sort_wrapper, 0, 0, 0},
        {"Quick Sort", quick_sort_wrapper, 0, 0, 0},
        {"Heap Sort", heap_sort, 0, 0, 0}
    };

    printf("Benchmark de Algoritmos de Ordenação\n");
    printf("=====================================\n\n");

    for (int t = 0; t < num_tamanhos; t++) {
        int n = tamanhos[t];
        printf("Tamanho do Array: %d elementos\n", n);
        printf("-" * 40);

        for (int tipo = 0; tipo < 5; tipo++) {
            printf("\nTipo de Dados: %s\n", nomes_tipos[tipo]);

            for (int alg = 0; alg < 6; alg++) {
                int *array_teste = malloc(n * sizeof(int));
                gerar_dados_teste(array_teste, n, (TipoDados)tipo);

                double tempo = medir_tempo_algoritmo(
                    algoritmos[alg].algoritmo, array_teste, n);

                printf("%-15s: %.6f segundos\n",
                       algoritmos[alg].nome, tempo);

                free(array_teste);
            }
        }
        printf("\n");
    }
```

Resultados Experimentais Típicos

Performance para 100.000 elementos

Algoritmo	Aleatório	Ordenado	Reverso	Quase Ord.
Bubble Sort	15.23s	0.03s	30.45s	2.15s
Selection Sort	8.67s	8.66s	8.68s	8.65s
Insertion Sort	4.32s	0.02s	8.64s	0.48s
Merge Sort	0.018s	0.017s	0.018s	0.017s
Quick Sort	0.014s	0.012s	0.013s	0.013s
Heap Sort	0.022s	0.021s	0.023s	0.022s

Análise dos Resultados

Observações Importantes:

1. **Algoritmos** $O(n^2)$ degradam drasticamente com tamanho
2. **Insertion Sort** é surpreendentemente eficiente para dados quase ordenados
3. **Merge Sort** tem performance mais consistente
4. **Quick Sort** é geralmente o mais rápido na prática

7. Aplicações Reais e Casos de Uso

Sistema de Ranking de Jogadores

```c
typedef struct {
    char nome[100];
    int pontuacao;
    int partidas_jogadas;
    double taxa_vitoria;
    time_t ultima_partida;
} Jogador;

int comparar_jogadores_ranking(const void *a, const void *b) {
    Jogador *j1 = (Jogador *)a;
    Jogador *j2 = (Jogador *)b;

    // Critério 1: Pontuação (mais importante)
    if (j1->pontuacao != j2->pontuacao)
        return j2->pontuacao - j1->pontuacao;  // Decrescente

    // Critério 2: Taxa de vitória
    if (j1->taxa_vitoria != j2->taxa_vitoria)
        return (j2->taxa_vitoria > j1->taxa_vitoria) ? 1 : -1;

    // Critério 3: Número de partidas (mais experiente)
    if (j1->partidas_jogadas != j2->partidas_jogadas)
        return j2->partidas_jogadas - j1->partidas_jogadas;

    // Critério 4: Atividade recente
    return (j2->ultima_partida > j1->ultima_partida) ? 1 : -1;
}

void atualizar_ranking(Jogador jogadores[], int num_jogadores) {
    // Usa qsort da biblioteca padrão (tipicamente introsort)
    qsort(jogadores, num_jogadores, sizeof(Jogador), comparar_jogadores_ranking);

    // Atualiza posições no ranking
    for (int i = 0; i < num_jogadores; i++) {
        printf("Posição %d: %s (Pontos: %d, Taxa: %.2f%%)\n",
               i + 1, jogadores[i].nome, jogadores[i].pontuacao,
               jogadores[i].taxa_vitoria * 100);
    }
}
```

8. Algoritmos de Ordenação Externa

Ordenação de Arquivos Grandes

Problema: Ordenar dados que não cabem na memória principal

Solução: External Merge Sort

```c
#include <stdio.h>

#define TAMANHO_BUFFER 1000000  // 1 milhão de elementos por chunk

typedef struct {
    FILE *arquivo;
    int buffer[TAMANHO_BUFFER];
    int posicao_buffer;
    int tamanho_buffer;
    int fim_arquivo;
} FluxoArquivo;

void ordenacao_externa(const char *arquivo_entrada,
                      const char *arquivo_saida, long total_elementos) {
    FILE *entrada = fopen(arquivo_entrada, "rb");
    int num_arquivos_temp = 0;

    // Fase 1: Divide em chunks ordenados
    char nome_temp[100];
    while (!feof(entrada)) {
        int buffer[TAMANHO_BUFFER];
        int elementos_lidos = fread(buffer, sizeof(int), TAMANHO_BUFFER, entrada);

        if (elementos_lidos > 0) {
            // Ordena chunk na memória
            qsort(buffer, elementos_lidos, sizeof(int), comparar_inteiros);

            // Salva chunk ordenado
            sprintf(nome_temp, "temp_%d.dat", num_arquivos_temp);
            FILE *temp = fopen(nome_temp, "wb");
            fwrite(buffer, sizeof(int), elementos_lidos, temp);
            fclose(temp);

            num_arquivos_temp++;
        }
    }
    fclose(entrada);

    // Fase 2: Merge dos arquivos temporários
    merge_arquivos_temporarios(arquivo_saida, num_arquivos_temp);

    // Limpeza
    for (int i = 0; i < num_arquivos_temp; i++) {
        sprintf(nome_temp, "temp_%d.dat", i);
        remove(nome_temp);
    }
}

void merge_arquivos_temporarios(const char *arquivo_saida, int num_arquivos) {
    FILE *saida = fopen(arquivo_saida, "wb");
    FluxoArquivo fluxos[num_arquivos];

    // Inicializa fluxos de entrada
    for (int i = 0; i < num_arquivos; i++) {
        char nome_temp[100];
        sprintf(nome_temp, "temp_%d.dat", i);
        fluxos[i].arquivo = fopen(nome_temp, "rb");
        carregar_proximo_elemento(&fluxos[i]);
    }

    // Merge usando heap para eficiência
    while (tem_elementos_restantes(fluxos, num_arquivos)) {
        int indice_menor = encontrar_menor_elemento(fluxos, num_arquivos);

        // Escreve menor elemento no arquivo de saída
        fwrite(&fluxos[indice_menor].buffer[fluxos[indice_menor].posicao_buffer]
```

9. Conclusões e Próximos Passos

Guia de Seleção de Algoritmos

Para Arrays Pequenos (n < 50):

- **Insertion Sort:** Simples e eficiente
- **Selection Sort:** Mínimo número de trocas

Para Arrays Médios/Grandes (n > 50):

- **Quick Sort:** Melhor performance média
- **Merge Sort:** Performance garantida e estável
- **Heap Sort:** Quando espaço é limitado

Para Dados Especiais:

- **Counting Sort:** Inteiros em range pequeno
- **Radix Sort:** Inteiros ou strings
- **TimSort:** Dados parcialmente ordenados

Preparação para Próximas Aulas

Aula 04: Estruturas de Dados Avançadas

- Árvores Binárias de Busca e AVL
- Hash Tables e Funções de Dispersão
- Grafos: Representação e Algoritmos Básicos

Bibliografia e Recursos

Referências Clássicas

- **Cormen, T. H. et al.** *Introduction to Algorithms*, 4ª edição
- **Sedgewick, R.** *Algorithms*, 4ª edição
- **Knuth, D. E.** *The Art of Computer Programming*, Volume 3

Implementações de Referência

- **GNU libc qsort():** Implementação industrial
- **Java Arrays.sort():** TimSort híbrido
- **C++ std::sort():** Introsort otimizado

Ferramentas de Análise

- **Complexity Analyzer:** Medição automática de complexidade
- **Profilers:** gprof, Valgrind Cachegrind
- **Visualizadores:** Algorithm Visualizer, Sorting Algorithms Animations

Encerramento da Aula

Algoritmos e Complexidade - Aula 03

Algoritmos de Ordenação e Análise de Performance

Próxima Aula: Estruturas de Dados Avançadas - Árvores e Hash Tables
Exercícios: Implementar e comparar 3 algoritmos de ordenação diferentes

Material Complementar

GitHub: github.com/cordeirotelecom/algoritimos_e_complexidade
Simuladores Online: VisuAlgo, Algorithm-Visualizer
Prática: LeetCode Sorting Problems