AED2 - Algoritmos e Estr. de Dados II

Aula 5: Ordenação Linear em $O(N)$

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

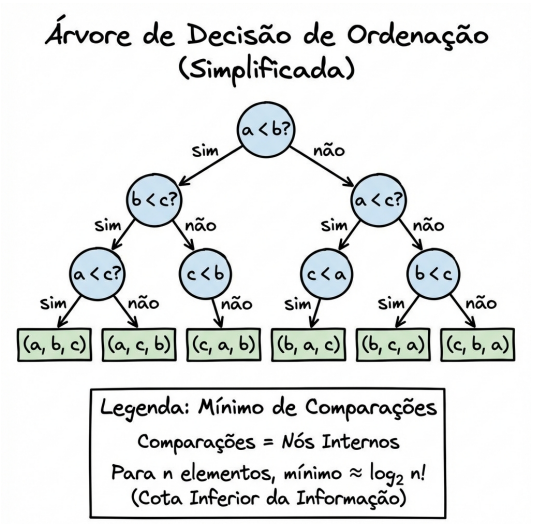
CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Fevereiro de 2026

Sumário

1. O Limite Inferior da Ordenação

Algoritmos clássicos (Quick, Merge, Heap Sort) são baseados em **comparação**: mínimo $\Omega(N \log N)$ comparações. Como ordenar em $O(N)$? **Não comparando** diretamente; explorando propriedades numéricas (ex.: inteiros em intervalo limitado).



2. Counting Sort (Ordenação por Contagem)

Este algoritmo é extremamente rápido quando os números a serem ordenados estão em um intervalo pequeno $[0, K]$.

Algoritmo

1. Crie um vetor de contagem `count` de tamanho $K + 1$. 2. Percorra a entrada e conte a frequência de cada elemento. 3. Transforme `count` em um vetor de prefixos acumulados (para determinar posições finais). 4. Posicione os elementos no vetor de saída (de trás para frente, para manter a **estabilidade**).

Implementação em Java

```java
public int[] countingSort(int[] arr) {
    if (arr.length == 0) return arr;

    // 1. Achar o maior valor (K)
    int max = arr[0];
    for(int x : arr) max = Math.max(max, x);
```

3. Radix Sort

O Radix Sort resolve o problema do K grande ordenando os números **dígito a dígito**. Geralmente usamos o **LSD (Least Significant Digit)**: ordenamos pelas unidades, depois dezenas, centenas... usando uma variante estável do Counting Sort para cada dígito.

Exemplo

Vetor: [170, 45, 75, 90, 802, 24, 2, 66] 1. Ordenar por **Unidades**: [170, 90, 802, 2, 24, 45, 75, 66] 2. Ordenar por **Dezenas**: [802, 2, 24, 45, 66, 170, 75, 90] (Note que 802 vem antes de 24 porque 0 ¡ 2 nas dezenas) 3. Ordenar por **Centenas**: [2, 24, 45, 66, 75, 90, 170, 802] -¿ **Ordenado! Complexidade:**

$$O(D \cdot (N + B))$$

- D: Número de dígitos (para inteiros de 32 bits, até 10 dígitos decimais).
- B: Base (10 para decimal).
- Na prática, é linear $O(N)$ para inteiros de tamanho fixo.

4. Bucket Sort

Ideal quando os dados são distribuídos uniformemente em um intervalo (ex: números reais entre [0, 1)).

Algoritmo

1. Crie N baldes (listas vazias). 2. Para cada elemento `val`, coloque no balde $\lfloor N \cdot val \rfloor$. 3. Ordene cada balde individualmente (ex: Insertion Sort ou Collections.sort). 4. Concatene os baldes. **Complexidade:**

- **Melhor/Médio:** $O(N)$. Se a distribuição for uniforme, cada balde terá ≈ 1 elemento.
- **Pior:** $O(N^2)$. Se todos caírem no mesmo balde e usarmos insertion sort.

5. Resumo Comparativo

Algoritmo	Médio	Pior	Espaço	Estável	Uso
Quick Sort	$O(N \log N)$	$O(N^2)$	$O(\log N)$	Não	Geral
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N)$	Sim	Estabilidade
Counting	$O(N+K)$	$O(N+K)$	$O(K)$	Sim	Inteiros
Radix	$O(N)$	$O(N)$	$O(N)$	Sim	Inteiros/strings
Bucket	$O(N)$	$O(N^2)$	$O(N)$	Sim	Floats

Dica de Java

O `Arrays.sort(int[])` do Java usa **Dual-Pivot Quicksort**. O `Collections.sort(List)` usa **Timsort** (híbrido de Merge + Insertion), que é estável e $O(N \log N)$. **Não existe Radix Sort nativo no Java Standard Library.** Se precisar ordenar 10^7 inteiros em 1 segundo no contest, você terá que implementar seu próprio Radix ou Counting Sort.