Algoritmos e Estruturas de Dados II

Capítulo da Aula 5: Ordenação Linear em $O(N)$

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br
CEFET-MG - Campus Timóteo

Fevereiro de 2026

0.1 1. O Limite Inferior da Ordenação

Algoritmos clássicos como **Quick Sort**, **Merge Sort** e **Heap Sort** são baseados em **comparação**. Existe uma prova matemática que diz que qualquer algoritmo que ordena comparando elementos dois a dois precisa fazer, no mínimo, $\Omega(N \log N)$ comparações no pior caso.

Então, como é possível ordenar em $O(N)$? **Resposta:** Não comparando elementos diretamente, mas explorando propriedades numéricas das chaves (como o fato de serem inteiros em um intervalo limitado).

0.2 2. Counting Sort (Ordenação por Contagem)

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

0.2.1 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**).

0.2.2 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);

    // 2. Contar Frequências
    int[] count = new int[max + 1];
```

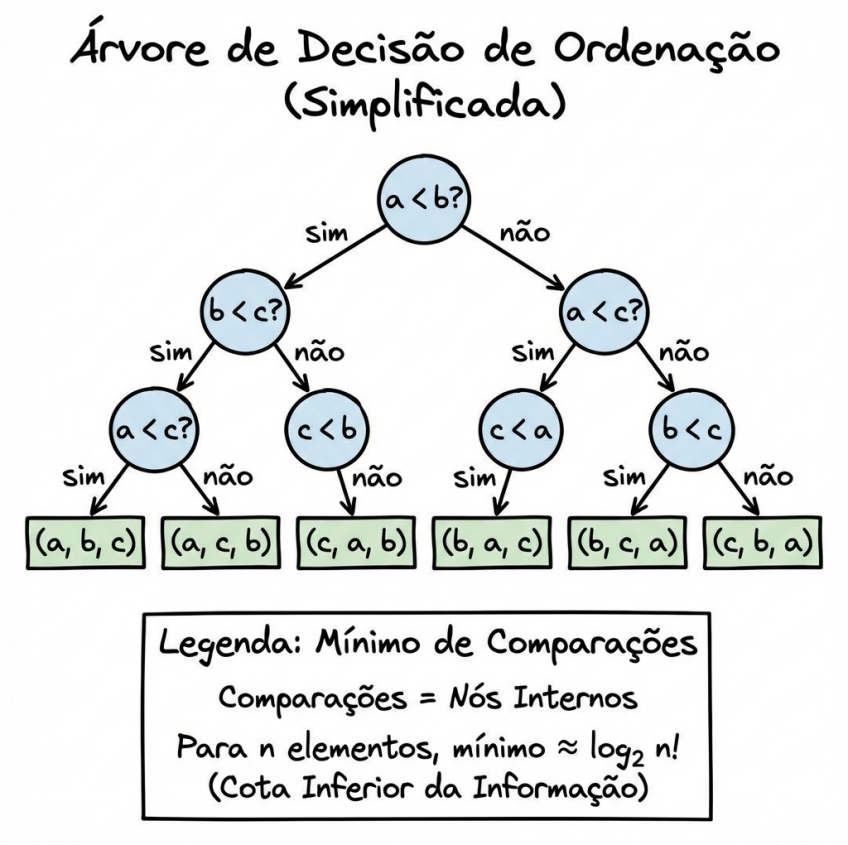

Figure 1: Limite inferior: ordenação por comparação exige pelo menos $\Omega(N \log N)$ comparações.

```
10      for(int x : arr) count[x]++;
11
12      // 3. Acumular (Prefix Sum)
13      for(int i = 1; i <= max; i++) {
14          count[i] += count[i-1];
15      }
16
17      // 4. Construir saída (invertido para estabilidade)
18      int[] output = new int[arr.length];
19      for(int i = arr.length - 1; i >= 0; i--) {
20          int valor = arr[i];
21          int posicao = count[valor] - 1;
22          output[posicao] = valor;
23          count[valor]--;
24      }
25      return output;
26  }
```

Análise:

- **Tempo:** $O(N + K)$. Se $K \approx N$, é $O(N)$.
- **Espaço:** $O(K)$.

- **Limitação:** Se tivermos que ordenar apenas 10 números, mas um deles for 1.000.000.000, precisaremos de um array de 1GB! **Counting Sort não serve para intervalos gigantes.**

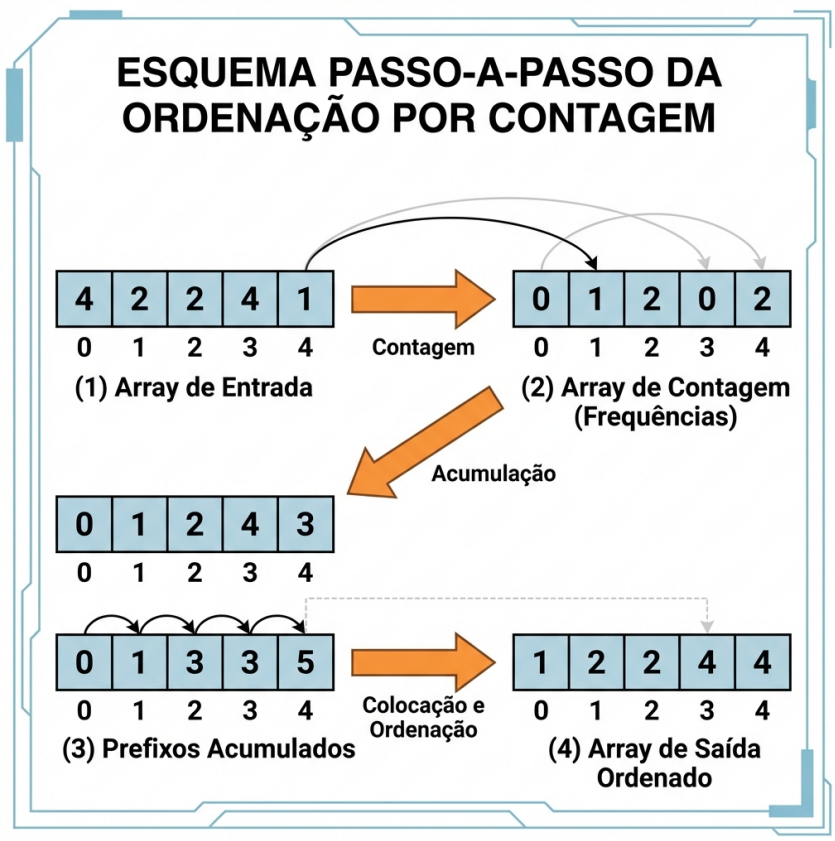

Figure 2: Counting Sort: vetor de contagem e depois prefixos acumulados para posicionar elementos.

0.3 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.

0.3.1 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.

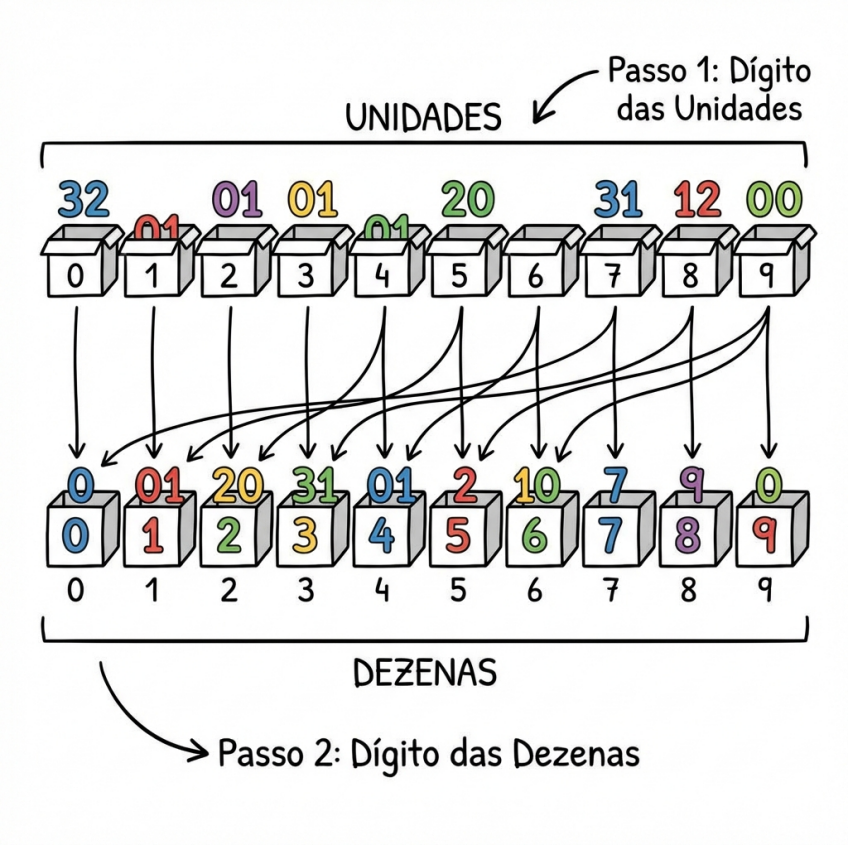

Figure 3: Radix Sort: ordenação dígito a dígito (unidades, dezenas, centenas) usando ordenação estável.

0.4 4. Bucket Sort

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

0.4.1 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.

0.5 5. Resumo Comparativo

Algoritmo	Tempo (Médio)	Tempo (Pior)	Espaço	Estável?	Quando usar?
Quick Sort	$O(N \log N)$	$O(N^2)$	$O(\log N)$	Não	Uso geral (padrão C-
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N)$	Sim	Quando estabilidade
Counting Sort	$O(N + K)$	$O(N + K)$	$O(K)$	Sim	Inteiros com $K < 10$
Radix Sort	$O(N)$	$O(N)$	$O(N)$	Sim	Inteiros grandes ou S
Bucket Sort	$O(N)$	$O(N^2)$	$O(N)$	Sim	Floats uniformes.

Table 1: Resumo comparativo dos algoritmos de ordenação.

0.5.1 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.

0.6 6. Exercícios

6.1. Exercícios Conceituais

1 Explique por que algoritmos de ordenação baseados em comparação não conseguem, em geral, quebrar a barreira de $\Omega(N \log N)$. Qual é a ideia central da prova desse limite inferior?

2 Descreva os cenários em que Counting Sort, Radix Sort e Bucket Sort são apropriados. Para cada algoritmo, dê exemplos de formatos de dados reais em Engenharia de Computação onde ele seria uma boa escolha.

3 Discuta as vantagens e desvantagens de usar algoritmos de ordenação linear em comparação com Quick Sort/Merge Sort em termos de:

- Complexidade de tempo.
- Uso de memória.
- Generalidade (tipos de dados suportados).

6.2. Exercícios Analíticos

1 Para Counting Sort, suponha que $N = 10^6$ e $K = 10^4$. Estime o tempo e a memória usados em relação a um Quick Sort padrão. O que muda se K passa a ser 10^9?

2 Para Radix Sort, analise o impacto da base escolhida (por exemplo, base 10, base 2^8, base 2^{16}) no número de passadas (D) e no custo de cada passada. Quais são os trade-offs práticos?

3 Mostre como Bucket Sort pode degradar para $O(N^2)$ em um caso de pior distribuição e construa um exemplo concreto de entrada que cause esse comportamento.

6.3. Exercícios de Programação

1 Implemente Counting Sort e Radix Sort para inteiros de 32 bits e compare o desempenho com `Arrays.sort(int[])` em diferentes cenários:

- Números uniformemente distribuídos em um intervalo pequeno (por exemplo, $[0, 10^5]$).
- Números concentrados em uma faixa estreita.
- Números espalhados em um intervalo muito grande.

2 Implemente Bucket Sort para números reais em $[0, 1)$ gerados por uma distribuição uniforme e por uma distribuição altamente concentrada em torno de um valor. Compare o tempo de execução e discuta os resultados.

3 Escolha um problema de juiz online em que a solução natural dependa de ordenar grandes volumes de inteiros (por exemplo, processamento de logs, classificação de chaves numéricas). Experimente substituir o sort padrão por uma implementação de Radix Sort e meça o impacto no tempo de execução.