AED2 - Algoritmos e Estr. de Dados II

Aula 2: Análise de Complexidade de Algoritmos

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Fevereiro de 2026

Sumário

1. Introdução: Por que analisar algoritmos?

No desenvolvimento de software de alto desempenho e competições:

- Escrever código correto é apenas metade da batalha.
- Garantir que execute dentro dos limites de **tempo** e **memória** é a outra metade.

Exemplo: Processamento de transações bancárias.

- 10 transações em 1 seg $\rightarrow$ OK.
- 1×10^5 transações em 10^5 seg (27 horas) $\rightarrow$ **Inviável**.

Essa relação entre **Tamanho da Entrada (N)** vs **Custo de Execução** é a Análise de Complexidade.

Para analisar algoritmos de forma independente de máquina, utilizamos uma aproximação teórica:

O Modelo RAM

- As instruções executam sequencialmente (uma por vez).
- Operações simples (matemática, atribuição, ifs) custam **1 unidade de tempo**.
- Acesso à memória gasta tempo constante $O(1)$.

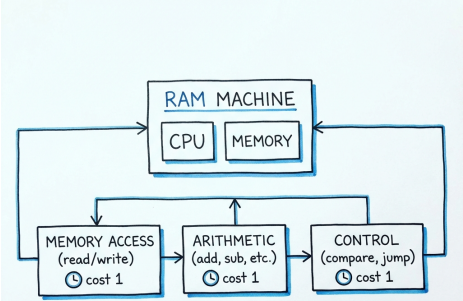


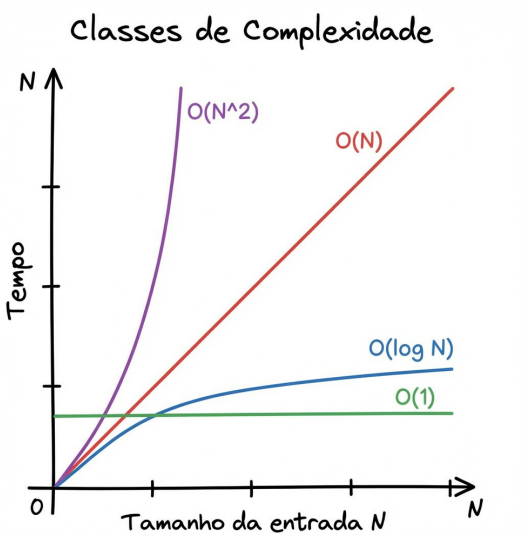
Figure: Modelo RAM: custo unitário.

2. Notação Assintótica (Big O)

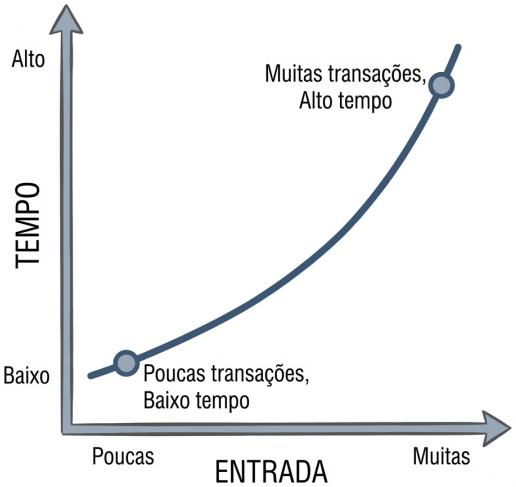
A notação Big O (O) descreve o **limite superior** quando $N \to \infty$.

- Foco no **Pior Caso**.
- Ignoramos constantes.
- Se N dobra, como o tempo cresce?

$O(N^2) \Rightarrow$ tempo **quadrático**.



2b. Tempo vs tamanho da entrada



3.1. Complexidade Linear $O(N)$

O exemplo mais clássico é a **Busca Linear**. Precisamos verificar cada elemento até encontrar o alvo ou chegar ao fim.

```java
public int buscaLinear(int[] vetor, int alvo) {
    // N é o tamanho do vetor
    for (int i = 0; i < vetor.length; i++) { // Executa N vezes no
        pior caso
        if (vetor[i] == alvo) {
            return i; // O(1)
        }
    }
    return -1;
}
```

Análise: No pior caso (elemento não existe ou está no final), o loop roda N vezes. Logo, $O(N)$.

4. Analisando Recursão: Teorema Mestre

Para algoritmos recursivos, nem sempre é óbvio olhar os loops. Usamos frequentemente o **Teorema Mestre** para dividir e conquistar. Uma recorrência comum é: $T(N) = aT(N/b) + f(N)$.

- a: número de chamadas recursivas.
- N/b: tamanho de cada subproblema.
- $f(N)$: custo para dividir/combinar.

Exemplo: Merge Sort O Merge Sort divide o vetor em 2 metades ($a = 2, b = 2$) e depois intercala as metades em tempo linear $O(N)$.

$$T(N) = 2T(N/2) + O(N)$$

Pelo Teorema Mestre, isso resulta em $O(N \log N)$.

Códigos Recursivos "Ingênuos": O cálculo de Fibonacci recursivo sem memorização:

```java
public int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2);
}
```

Isso gera uma árvore de recursão onde cada nó (estado) empilha 2 novas chamadas.

- Nível 0: 1 chamada
- Nível 1: 2 chamadas
- Nível 2: 4 chamadas ... Nível N: 2^N chamadas.

Total de Operações: $1 + 2 + 4 + \cdots + 2^{N-1} = 2^N - 1 \approx O(2^N)$.
Extremamente ineficiente para $N > 40$.

5. Dicas para Maratonas e Juízes Online

Ao submeter um problema no Maratona ou no Beecrowd, Codeforces ou LeetCode, você verá um "Time Limit" (geralmente 1 ou 2 segundos).
Embora processadores modernos atinjam bilhões de ciclos, devido ao ambiente restrito (sandbox) e ao overhead das instruções, a regra de bolso prática é: Podemos usar essa regra de bolso para estimar se nossa solução vai passar (TLE) ou não:

N	**Complexidade**	**Sugestão**
≤ 10	$O(N!)$	Permutação
≤ 20	$O(2^N)$	Backtracking, Bitmask
≤ 500	$O(N^3)$	Floyd-Warshall
$\leq 2\,000$	$O(N^2)$	DP 2D
$\leq 10^5$	$O(N \log N)$	Sort, Hash, Árvores
$\leq 10^9$	$O(\sqrt{N})$	Fatoração
$\leq 10^{18}$	$O(\log N)$	Exp. Rápida

Por que ≤ 500 aceita $O(N^3)$?
Se $N = 500$, então $500^3 = 125.000.000 \approx 1.25 \times 10^8$.
Esse valor está muito próximo do limite de 10^8 operações. Qualquer constante pesada causará **TLE**.

Exemplo de Análise Prévia

Problema: Dado um vetor de $N = 10^5$ números, encontre dois números que somam K.

1. **Abordagem Força Bruta:** Testar todos os pares.
   - Complexidade: $O(N^2)$.
   - Operações estimadas: $(10^5)^2 = 10^{10}$.
   - **Veredito:** $10^{10} > 10^8$. Muito lento. **TLE** (Time Limit Exceeded).
2. **Abordagem Otimizada:** Testar depois de ordenar ou usar HashSet.
   - Usando Sort + Two Pointers: $O(N \log N)$.
   - Operações: $10^5 \times 17 \approx 1.7 \times 10^6$.
   - **Veredito:** $1.7 \times 10^6 < 10^8$. **Passa com folga (Accepted).**

5. Dicas para Juízes Online: Ordenação

E quando os algoritmos de ordenação estouram o tempo limite de 10^8 operações?

Bubble Sort ($O(N^2)$):

- Para $N = 10.000$ (10^4): $(10^4)^2 = 10^8$ operações. **(Passa raspando)**.
- Para $N = 100.000$ (10^5): $(10^5)^2 = 10^{10}$ operações. **(TLE - Demora ≈ 100 segundos)**.

Merge/Quick Sort ($O(N \log_2 N)$):

- Para $N = 100.000$ (10^5): $10^5 \times 17 \approx 1.7 \times 10^6$ operações. **(Gasta ≈ 0.01 segundos, Passa!)**.
- Para $N = 1.000.000$ (10^6): $10^6 \times 20 \approx 2 \times 10^7$ operações. **(Gasta ≈ 0.2 segundos, Passa!)**.
- Quando o QuickSort falha? Somente se $N > 4.000.000$ (4×10^6) em juízes muito rígidos (limite de 1s).

6. Complexidade de Espaço: Conceitos

Não esqueça da **memória**! "Memory Limit Exceeded" (MLE) também é um erro comum nas plataformas.

- $O(1)$: Usa algumas variáveis auxiliares. (int, double).
- $O(N)$: Usa um vetor/lista auxiliar do tamanho da entrada.
- $O(N^2)$: Cria uma matriz/tabela $N \times N$.

Na prática, sempre tente reduzir a complexidade de tempo primeiro, mas fique atento se você não está alocando memória demais (ex: recursão muito profunda estoura a Stack, tabelas de programação dinâmica gigantes estouram a Heap).

6. Complexidade de Espaço: Limites no Java

Qual o limite para um `int[]` em Java de 512MB?

Em Java, `int` pesa 4 bytes.

- 1 MB $\approx 10^6$ bytes $\approx$ 250.000 inteiros.
- 512 MB $\approx 5.12 \times 10^8$ bytes $\approx 1.28 \times 10^8$ inteiros (128 milhões).

Atenção: A JVM usa overhead, objetos (como `Integer`) e Garbage Collector gastam mais memória. Na prática, alocar um `int[100_000_000]` pode estourar os limites dependendo do Juiz.

Cuidado com matrizes: Uma matriz $N = 10.000$ (`int[10000][10000]`) ocupa quase **400 MB**, facilmente estourando limites comuns de 256MB.

7. Gargalos de Entrada e Saída (I/O)

Mesmo que sua complexidade de tempo seja excelente ($O(N)$), seu código pode tomar **TLE** simplesmente pela lentidão na **leitura/escrita** de dados massivos na tela.

O Problema no Java:

- A classe `Scanner` é muito lenta porque faz *parsing* robusto usando Expressões Regulares por baixo dos panos.
- `System.out.println` descarrega o buffer na tela a cada chamada, o que é custoso.

Quando me preocupar?

- Se o problema exige ler/imprimir 10^5 **ou mais** números/linhas.
- Substitua `Scanner` por `BufferedReader` e `StringTokenizer` (também chamado de "Fast I/O").
- Substitua `println()` em loops por `StringBuilder` ou `PrintWriter`.

7.1. Tópico Extra: `Scanner` como objeto

O `Scanner` é um **objeto** que lê de uma **fonte** (teclado, arquivo, string). Usar só `next()` é limitado e pode gerar erros.

Fontes de entrada

```
1 Scanner sc = new Scanner(System.in);       // Teclado
2 Scanner arq = new Scanner(new File("in")); // Arquivo
3 Scanner str = new Scanner("10 20 30");     // String
```

Problema: `next()` vs `nextLine()`

`next()` lê até o próximo **delimitador** (espaço, tab) e **não consome** a quebra de linha.
`nextLine()` lê até o fim da linha e consome a quebra.

```
1 int n = sc.nextInt();
2 sc.nextLine(); // Consome a quebra deixada pelo nextInt()
3 String linha = sc.nextLine(); // Agora le a linha inteira
```

7.2. `Scanner`: EOF (End Of File)

EOF — fim da entrada

EOF é o sinal de que não há mais dados para ler.

- Em juízes online e em redirecionamento de arquivo (`java Main < entrada.txt`), a entrada termina no fim do arquivo.
- No `Scanner`, **não existe** um método `isEOF()`.
- O fim da entrada é detectado quando os métodos `hasNext*` retornam **false**: não há próximo token/linha.

7.3. Scanner: Métodos hasNext e uso com EOF

Métodos hasNext

- `hasNext()`, `hasNextInt()`, `hasNextDouble()` — há próximo token? (`false` = EOF)
- `hasNextLine()` — há próxima linha? (`false` = EOF)

Exemplos: ler até EOF

```
// Ler inteiros ate EOF (tipico em juizes: "le ate o fim")
while (sc.hasNextInt()) {
    int x = sc.nextInt();
    soma += x;
}

// Ler todas as linhas ate EOF
while (sc.hasNextLine()) {
    String linha = sc.nextLine();
}
```

7.4. `String.split()` e mais exemplos

`split(regex)`: divide uma `String` em um array de substrings conforme o delimitador (regex).

Exemplo: ler linha e quebrar em tokens

```
String linha = sc.nextLine();
String[] partes = linha.split(" ");  // Divide por espaco

for (String s : partes) {
    int x = Integer.parseInt(s);
}
```

Delimitador customizado no `Scanner`

```
sc.useDelimiter(",");  // Agora next() le ate a virgula
sc.useDelimiter("\\s+"); // Um ou mais espacos (padrao)
```

7.5. `split()` com regex e exemplos práticos

Regex comuns em `split`

- `split(" ")` — por espaço.
- `split("\s+")` — um ou mais espaços/tabs.
- `split(",")` — por vírgula (CSV).
- `split(";\s*")` — ponto e vírgula, com espaços opcionais.

Exemplo: ler CSV ou múltiplos valores por linha

```
// Entrada: "nome; 10; 20.5"
String[] tokens = linha.split(";\\s*");
String nome = tokens[0];
int a = Integer.parseInt(tokens[1]);
double b = Double.parseDouble(tokens[2]);
```

8. Complexidade Amortizada

Às vezes, uma operação pode ser lenta no pior caso, mas, quando avaliada ao longo de **MUITAS execuções**, o custo médio *(amortizado)* dilui-se e fica pequeno.

O Clássico `ArrayList.add()`

Quando você insere num `ArrayList` em Java, a complexidade geralmente é $O(1)$.

Porém, quando o array interno **enche**, o Java precisa:

- Criar um novo array interno maior (geralmente $+50\%$ em Java).
- Copiar todos os N elementos para o novo vetor (custo $O(N)$).

Logo, no pior caso o `add()` é $O(N)$. No entanto, como o aumento de tamanho acontece exponencialmente espaçado, o custo "caro" das cópias se dilui ao longo das muitas inserções $O(1)$ recentes.

Conclusão: Dizemos que a inserção sistemática no `ArrayList` tem custo $O(1)$ **amortizado**.