Algoritmos e Estruturas de Dados II

Capítulo da Aula 2: Análise de Complexidade de Algoritmos

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

Fevereiro de 2026

0.1 1. Introdução: Por que analisar algoritmos?

No desenvolvimento de software de alto desempenho e, especialmente, em competições de programação, escrever um código que produza a saída correta é apenas metade da batalha. A outra metade é garantir que o código execute dentro dos limites de **tempo** e **memória** disponíveis.

Imagine um sistema que precisa processar transações bancárias. Se um algoritmo demora 1 segundo para processar 10 transações, ele pode parecer rápido. Mas se ele demorar 100 segundos para 100 transações e 10^5 segundos para 1000 transações, ele é inviável para um banco real (veja a Figura 1). Essa relação entre o **tamanho da entrada** (N) e o **custo de execução** é o que estudamos na Análise de Complexidade.

Figure 1: Relação entre tamanho da entrada e tempo de execução: exemplo transações bancárias.

0.1.1 O Modelo RAM

Para analisar algoritmos independentemente do hardware (seja um supercomputador ou um celular), usamos um modelo teórico simples chamado **Random Access Machine (RAM)**, ilusrado na Figura 2:

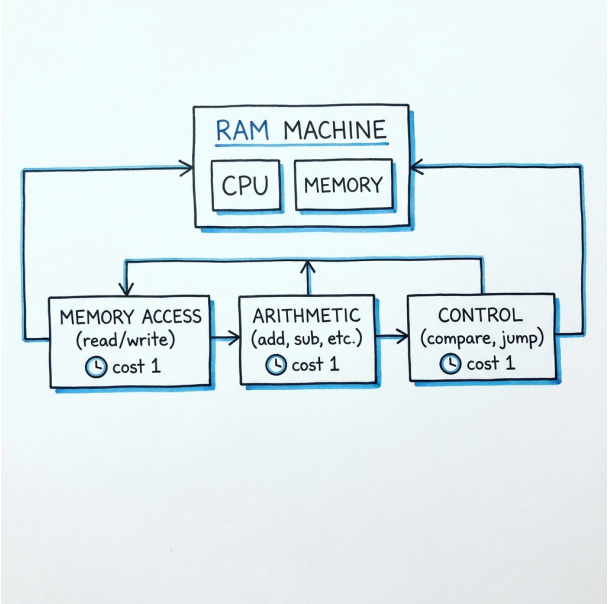

Figure 2: Modelo RAM: cada operação simples tem custo unitário; acesso à memória é constante.

- Cada operação simples (+, -, *, /, if, chamada de função) leva 1 unidade de tempo.
- Loops e subrotinas não são simples; são compostos por várias operações simples.
- Acesso à memória é constante $O(1)$.

0.2 2. Notação Assintótica (Big O, Ω e Θ)

A notação Big O (O) descreve o **limite superior assintótico** do tempo de execução de um algoritmo à medida que a entrada N cresce para o infinito. Ela nos diz, no "pior cenário de ordem de grandeza", como o algoritmo escala.

Se dizemos que um algoritmo é $O(N^2)$, não estamos dizendo que ele leva exatamente N^2 segundos, mas sim que o tempo de execução cresce **quadraticamente** em relação à entrada (ignorando constantes e termos de menor ordem). O crescimento comparativo dessas famílias de função pode ser observado na Figura 3.

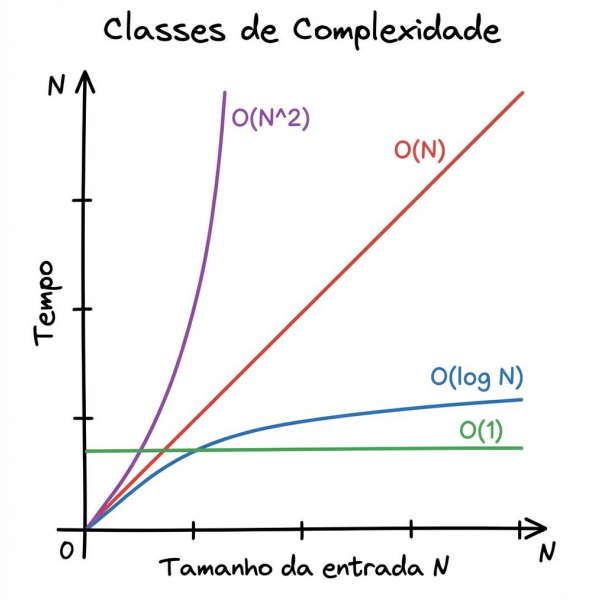

Figure 3: Crescimento assintótico: comparação entre O(1), O(log N), O(N) e O(N²).

• Teoria

Formalmente, dizemos que $T(N) \in O(f(N))$ se existem constantes $c > 0$ e N_0 tais que, para todo $N \geq N_0$, vale:

$$T(N) \leq c \cdot f(N).$$

De forma análoga, $T(N) \in \Omega(f(N))$ se, para N suficientemente grande, $T(N)$ é *no mínimo* proporcional a $f(N)$, e $T(N) \in \Theta(f(N))$ quando vale simultaneamente $T(N) \in O(f(N))$ e $T(N) \in \Omega(f(N))$.

0.2.1 Classes de Complexidade Comuns

0.3 3. Exemplos Práticos em Java

0.3.1 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.

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

Notação	Nome	Descrição	Exemplo Típico
$O(1)$	Constante	O tempo não muda com N.	Acessar `arr[i]`, operações aritméticas.
$O(\log N)$	Logarítmica	O problema é dividido em pedaços menores.	Busca Binária, operações em `TreeSet`.
$O(N)$	Linear	Percorre a entrada uma vez.	Loop simples, Busca Linear.
$O(N \log N)$	Linearítmica	Combinação de linear e log.	Merge Sort, `Arrays.sort()`.
$O(N^2)$	Quadrática	Loop aninhado (todos com todos).	Bubble Sort, Selection Sort.
$O(N^3)$	Cúbica	Três loops aninhados.	Multiplicação de Matrizes ingênua.
$O(2^N)$	Exponencial	Tenta todas as subcombinações.	Caixeiro Viajante (Força Bruta).
$O(N!)$	Fatorial	Tenta todas as permutações.	Gerar anagramas.

Table 1: Classes de Complexidade Assintótica Comuns.

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

0.3.2 3.2. Complexidade Quadrática $O(N^2)$

Geralmente ocorre quando temos loops aninhados dependentes de N. Um exemplo é verificar se há **duplicatas** comparando todos com todos.

```
public boolean temDuplicata(int[] vetor) {
    int n = vetor.length;
    // Loop Externo: Roda N vezes
    for (int i = 0; i < n; i++) {
        // Loop Interno: Roda cerca de ~N vezes
        for (int j = i + 1; j < n; j++) {
            if (vetor[i] == vetor[j]) {
                return true;
            }
        }
    }
    return false;
}
```

Análise: O loop interno roda $(N-1)+(N-2)+\cdots+1$, que é a soma aritmética $\frac{N(N-1)}{2}$. Ignorando constantes e termos menores, temos $\frac{N^2}{2} \approx O(N^2)$.

0.3.3 3.3. Complexidade Logarítmica $O(\log N)$

A **Busca Binária** é o exemplo fundamental. A cada passo, cortamos o espaço de busca pela metade. Isso só funciona se o vetor estiver **ordenado**.

```java
public int buscaBinaria(int[] vetor, int alvo) {
    int inicio = 0;
    int fim = vetor.length - 1;

    while (inicio <= fim) {
        // Evita overflow de (inicio+fim)
        int meio = inicio + (fim - inicio) / 2;

        if (vetor[meio] == alvo) return meio;

        if (vetor[meio] < alvo) {
            inicio = meio + 1; // Descarta metade esquerda
        } else {
            fim = meio - 1;    // Descarta metade direita
        }
    }
    return -1;
}
```

Análise: Quantas vezes podemos dividir N por 2 até chegar a 1? A resposta é $\log_2 N$. Para $N = 1.000.000$, o loop executa apenas cerca de 20 iterações!

0.4 4. Analisando Recursão

Para algoritmos recursivos, nem sempre é óbvio olhar os loops. Em muitos casos, escrevemos uma **recorrência** para o tempo de execução $T(N)$ e a resolvemos. Para algoritmos de "divisão e conquista", usamos frequentemente o **Teorema Mestre**.

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 1: 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 (caso 2), isso resulta em $O(N \log N)$.

Exemplo 2: Busca Binária Recursiva

$$T(N) = T(N/2) + O(1) \Rightarrow T(N) \in O(\log N).$$

Exemplo 3: Divisão em a subproblemas de tamanho N/b com custo linear:

$$T(N) = aT(N/b) + O(N).$$

Se $a < b$, o custo é dominado pela parte "de fora" e tende a $O(N)$; se $a = b$, temos algo como Merge Sort ($O(N \log N)$); se $a > b$, o custo explode e obtemos $O(N^{\log_b a})$.

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

```
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 gera 2 chamadas filhas.

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

O total de operações soma a progressão geométrica $1 + 2 + 4 + \cdots + 2^{N-1} = 2^N - 1$. A complexidade, portanto, é $O(2^N)$. Isso se torna **extremamente ineficiente** para $N > 40$.

0.5 5. Dicas para Maratonas e Juízes Online

Ao submeter um problema no Beecrowd, Codeforces ou LeetCode, você verá um "Time Limit" (geralmente 1 ou 2 segundos). Um computador moderno processa aproximadamente 10^8 operações simples por segundo.

Podemos usar essa regra de bolso para estimar se nossa solução vai passar (Time Limit Exceeded - TLE) ou não:

Tamanho (N)	Complexidade Esperada	Algoritmos Sugeridos
$N \leq 10$	$O(N!)$	Permutação Completa
$N \leq 20$	$O(2^N)$	Backtracking, Bitmask DP
$N \leq 500$	$O(N^3)$	Floyd-Warshall, Multiplicação Matrizes
$N \leq 2.000$	$O(N^2)$	Bubble Sort, DP 2D, Dijkstra (denso)
$N \leq 10^5$ ou 10^6	$O(N \log N)$ ou $O(N)$	Sort padrão, Busca Binária, Hash Map
$N \leq 10^9$	$O(\sqrt{N})$	Fatoração de Primos, Mo's Algorithm
$N \leq 10^{18}$	$O(\log N)$ ou $O(1)$	Exponenciação Rápida, Algor. Matemáticos

Table 2: Limites de Tempo e Complexidade Recomendada.

Por que ≤ 500 aceita $O(N^3)$? Se a entrada $N = 500$, então $500^3 = 125.000.000 \approx 1.25 \times 10^8$. Esse valor fica exatamente ali próximo ao limiar tolerado de 10^8 operações em 1 segundo. Ou seja, qualquer algoritmo cúbico de constante alta falharia para $N \geq 500$ tomando TLE.

0.5.1 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: $(10^5)^2 = 10^{10}$.

- **Veredito:** $10^{10} > 10^8$. Muito lento. **TLE**.

2. **Abordagem Otimizada:** Usar um HashSet ou ordenar.

- 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).**

0.5.2 Limites práticos para Ordenações Comuns

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 - Demoraria cerca de 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. **(Passa com folga)**.
  - Para $N = 1.000.000$ (10^6): $10^6 \times 20 \approx 2 \times 10^7$ operações. **(Passa com folga)**.

0.6 6. Complexidade de Espaço

Não esqueça da **memória**! O veredito "Memory Limit Exceeded" (MLE) também é um erro comum em problemas competitivos.

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

Qual o limite prático para alocação no Java?
O tipo `int` ocupa 4 bytes em Java. Portanto:

- 1 MB é capaz de armazenar cerca de ≈ 250.000 inteiros.
- 512 MB (um limite comum elevado na maratona) serviria idealmente para alocar até 128 milhões de inteiros ($\approx 1.28 \times 10^8$).

Atenção: A JVM usa muito overhead arquitetural, objetos como instâncias de `Integer` gastam bem mais memória e o *Garbage Collector* precisa de margem térmica. Na prática, alocar `int[100_000_000]` pode acionar rapidamente o limite dependendo do juiz online. Uma matriz 10.000×10.000 de inteiros suga rapidamente **400 MB**, falhando fatalmente nas configurações de juiz de 256MB.

Sempre tente focar na redução da complexidade de *tempo* primeiro, mas atente-se a matrizes de DP gigantes na *Heap* ou chamadas recursivas super profundas no arremato da *Stack*.

0.7 7. Gargalos de Entrada e Saída (I/O Limit)

Mesmo que a sua análise assintótica indique uma complexidade de tempo excelente (como um algoritmo $O(N)$), o seu código pode ainda assim ser punido com o veredito de **Time Limit Exceeded (TLE)** simplesmente pela ineficiência intrínseca na **leitura ou escrita** de dados massivos no terminal.

O Problema no Java: A leitura de dados padrão do Java, por meio da classe `Scanner`, é reconhecida por ser extremamente lenta. Isso acontece porque o `Scanner` provê um *parsing* robusto que internamente utiliza Expressões Regulares de altíssimo custo computacional apenas para entender onde termina e começa um simples tipo `int`. Além disso, imprimir linhas na tela utilizando `System.out.println` para cada resposta em um laço força a máquina virtual a descarregar o *buffer* de saída (fazer *flush*) sistematicamente para o terminal, bloqueando a execução.

Quando me preocupar (A Regra dos 10^5):

- Se o problema computacional exigir a leitura ou a impressão de um volume de dados da ordem de 10^5 **elementos** ou mais.
- A solução mais rápida é substituir o uso do `Scanner` por classes mais primitivas de buffer como a amálgama do `BufferedReader` em conjunto com `StringTokenizer` (também chamado em fóruns de programação de "Fast I/O" ou "FastReader").
- Em vez de chamar o `println()` iterativamente em um loop grande de respostas independentes, concatene todas em uma classe `StringBuilder` e depois imprima todas elas agrupadas no final, de uma só vez. Alternativamente, considere o uso seguro de um `PrintWriter`.

0.8 8. Análise Amortizada

Em certas estruturas de dados e algoritmos, você se deparará com operações que podem até exibir um comportamento flagrante de lentidão em um cenário isolado de Pior Caso. Porém, quando avaliamos essa mesma estrutura ao longo de **uma grande sequência de execuções**, esse elevado custo momentâneo acaba se "diluindo", tornando o tempo global de uso excelente. A avaliação matemática focada em enxergar esse custo médio e espalhado na linha do tempo da aplicação é a **Complexidade Amortizada**.

O Clássico ArrayList.add()

O exemplo prático essencial de análise amortizada é a inserção repetitiva (`add`) na classe `ArrayList` do Java. Por ser um redimensionamento dinâmico baseado em um vetor por debaixo dos panos, o custo comum de adicionar um número no final do array é constante: $O(1)$.

Porém, quando o array de retaguarda interno enche, o Java é obrigado a entrar em ação de maneira agressiva:

1. Instanciar um novo array interno maior na memória (geralmente expandido de 50% da capacidade em Java e até 100% em linguagens baseadas em C++).
2. Copiar um a um todos os N elementos velhos para a nova região (custo imediato e brutal de $O(N)$).

Em uma análise individual e míope de pior caso, um desenvolvedor poderia assumir que o ato de usar `add()` é $O(N)$ e portando descartá-lo de laços de repetição. Contudo, perceba que a dobra ou o aumento de 1.5x no tamanho de alocação da retaguarda acontece de maneira *exponencialmente espaçada* e não constantemente. Para que aquele evento de lentidão de cópia pesada ($O(N)$) venha a ocorrer novamente, garantimos, pela matemática, que tivemos a certeza de faturar inúmeras operações super baratas em $O(1)$ antes dessa fadiga.

Logo esse tal *stress computacional* dilui-se pelos elementos recém e facilmente inseridos, de tal forma que formalizamos a sua descrição no jargão dos algoritmos cravando que a inserção sequencial em um arranjo dinâmico possui custo sistemático $O(1)$ **amortizado**.

0.9 9. Exercícios

9.1. Exercícios Conceituais

1 Seja $T_1(N) = 3N^2+5N+10$ e $T_2(N) = 0{,}5N^2$. Mostre formalmente que $T_1(N) \in \Theta(N^2)$ e que $T_2(N) \in O(N^2)$ e $\Omega(N^2)$.

2 Ordene, do menor para o maior crescimento assintótico, as seguintes funções: N^3, 2^N, $N \log N$, $N!$, $\log N$, N^2. Justifique brevemente com argumentos matemáticos (por exemplo, limites ou comparação de razões).

3 Dê um exemplo de algoritmo com complexidade de **tempo** $O(N)$ mas que usa **espaço** adicional $O(N^2)$. Explique em que contexto isso pode ser aceitável ou não.

9.2. Exercícios Analíticos

1 Para cada um dos trechos de código a seguir, determine a complexidade de tempo no pior caso em função de N e justifique:

- Um laço simples de $i = 0$ até $N - 1$ com corpo $O(1)$.
- Dois laços aninhados, ambos de 0 até $N - 1$.
- Um laço externo de $i = 1$ até N e um laço interno de $j = 1$ até i.

2 Escreva a recorrência que modela o tempo de execução de:

- Um algoritmo de divisão e conquista que divide o problema em 4 subproblemas de tamanho $N/2$ e faz trabalho linear para combinar as respostas.
- Um algoritmo recursivo que, a cada chamada, reduz o tamanho da entrada em 3 (isto é, chama recursão com $N - 3$) e faz trabalho constante fora da recursão.

Em cada caso, discuta qualitativamente (sem resolver completamente) se a solução tende a ser polinomial, logarítmica, exponencial, etc.

9.3. Exercícios de Programação

1 Dado um conjunto de algoritmos candidatos para resolver um problema, cada um com uma expressão de custo $T(N)$ diferente (por exemplo, $N^2 + 100N$, $50N \log N$, 2^N), escreva um programa que, para um determinado N, estime qual deles terá o menor tempo de execução. Use essa ferramenta para comparar alternativas em instâncias reais.

2 Escolha três problemas de um juiz online (Beecrowd, Codeforces, LeetCode) e, antes de codificar, escreva:

- A complexidade esperada da solução pretendida.
- O limite de N informado no enunciado.
- Uma estimativa de número de operações e se isso cabe em 1 segundo.

Depois, implemente a solução e compare o tempo obtido com sua estimativa.

3 Implemente versões iterativa e recursiva de um mesmo algoritmo (por exemplo, cálculo de fatorial, Fibonacci com memorização ou busca binária). Meça, para diferentes valores de N, o tempo de execução e o uso de memória (se possível), e comente as diferenças observadas.