Análise Assintótica e Complexidade de Tempo de Algoritmo de Busca Binária

Akilan H. R. Gomes[1], Rafael R. G. Neves[1], Ramon M. da Silva[1]

[1]Instituto Federal de Educação, Ciência e Tecnologia do Maranhão (IFMA)
65.030-005 – São Luís – MA – Brasil

{hendersonakilan@acad.ifma.edu.br, ramon.s@acad.ifma.edu.br, rafaelgalvao@acad.ifma.edu.br

Abstract. *This study provides an empirical analysis of the asymptotic time complexity of the Binary Search algorithm. 2The algorithm was implemented in three distinct languages—C, Java, and Python—to compare performance across native, virtual machine, and interpreted environments. 3 The experiment involved executing searches on 10 sets of ordered data, with sizes scaling from 10,000 to 100,000 elements. Each vector size was tested 50 times to compute a stable average execution time and standard deviation. The results empirically validate the theoretical O(log n) complexity; execution time remained nearly constant and did not grow linearly with the input size. 4 The analysis also highlights a clear performance hierarchy (C > Java >> Python) and observes the JIT (Just-In-Time) compilation "warm-up" cost in Java and the startup overhead in Python, evident in the initial test (n=10,000).*

Resumo. *Este estudo apresenta uma análise empírica da complexidade de tempo assintótica do algoritmo de Busca Binária. O algoritmo foi implementado em três linguagens distintas — C, Java e Python — para comparar o desempenho entre ambientes nativo, de máquina virtual e interpretado. O experimento consistiu na execução de buscas em 10 conjuntos de dados ordenados, com tamanhos variando de 10.000 a 100.000 elementos. Cada tamanho de vetor foi testado 50 vezes para o cálculo da média de tempo de execução e do desvio padrão. Os resultados validam empiricamente a complexidade teórica O(log n); o tempo de execução permaneceu quase constante, sem apresentar crescimento linear com o aumento da entrada. A análise também destaca uma clara hierarquia de performance (C > Java >> Python) e observa o custo de "aquecimento" da compilação JIT (Just-In-Time) em Java e a sobrecarga de inicialização em Python, evidentes no teste inicial (n=10.000).*

1. Introdução

A análise de algoritmos é um pilar fundamental da ciência da computação, permitindo-nos prever o comportamento de um programa em termos de consumo de recursos — especificamente, tempo de execução e espaço de memória — à medida que o volume de dados de entrada aumenta. Entre as tarefas mais críticas em computação está a recuperação de dados, onde algoritmos de busca desempenham um papel central.

Este estudo foca no algoritmo de Busca Binária, um método de busca de alta eficiência projetado para operar sobre conjuntos de dados previamente ordenados. A sua complexidade de tempo teórica é universalmente estabelecida como O(log n), uma melhoria exponencial sobre métodos lineares como a busca sequencial (O(n)).

Contudo, a performance teórica, embora fundamental, não captura todo o cenário do desempenho no mundo real. A plataforma de execução — seja o código compilado nativamente (como em C), gerenciado por uma máquina virtual (como em Java) ou interpretado (como em Python) — introduz diferentes níveis de sobrecarga (*overhead*) que impactam o tempo de execução absoluto.

O objetivo principal deste trabalho é, portanto, duplo. Primeiramente, comprovar experimentalmente se a taxa de crescimento do tempo de execução da Busca Binária obedece à curva teórica O(log n) quando o tamanho da entrada (n) aumenta e quantificar e comparar o desempenho no mundo real de três implementações funcionalmente idênticas do algoritmo, executadas nas linguagens C, Java e Python.

Para atingir esses objetivos, foi desenvolvida uma suíte de *benchmark* que executou 50 testes de busca em 10 diferentes tamanhos de vetor, escalando de 10.000 a 100.000 elementos. A implementação específica da Busca Binária utilizada foi a iterativa, uma escolha deliberada de engenharia para garantir a máxima eficiência de memória, alcançando uma complexidade de espaço O(1) (Constante), superior à alternativa recursiva (O(log n) em espaço).

Este artigo apresenta, primeiramente, a análise teórica detalhada do algoritmo (Seção 2) e, subsequentemente, apresenta e discute os resultados estatísticos (média e desvio padrão) e gráficos obtidos no experimento (Seção 3).

2 Análise Teórica (Assintótica)

Para fundamentar a análise, abstraíu-se a lógica central do algoritmo em um pseudocódigo:

```
1   Funcao BuscaBinaria(vetor: Vetor de Inteiros, alvo: Inteiro) : Inteiro
2   Var
3       inicio, fim, meio: Inteiro
4   Inicio
5       inicio ← 0
6       fim ← Tamanho(vetor) - 1
7       Enquanto (inicio ⩽ fim) Faca
8           meio ← inicio + (fim - inicio) \ 2
9           Se (vetor[meio] = alvo) Entao
10              Retorne meio // Sucesso
11          FimSe
12          Se (vetor[meio] < alvo) Entao
13              inicio ← meio + 1
14          Senao
15              fim ← meio - 1
16          FimSe
17      FimEnquanto
18      Retorne -1
19  FimFuncao
```

Figure 1 - Pseudocódigo

2.1 Busca Binária

Considerando que, em análise de algoritmo, a notação assintótica é utilizada para medir como o tempo de execução ou o espaço requerido (memória) de um algoritmo cresce à medida que o tamanho da entrada cresce, percebe-se que esta mostra-se como ferramente com grande potencial no estudo da eficiencia do uso dos recursos computacionais, sobretudo, em um constexto em que os sistemas têm que lidar com o número crescente de dados.

A busca binária pode apresentar-se em duas configurações, iterative e recursive. Apesar desta modalidade de busca, conhecidamete, possuir carater logarítmico O(log n),

tal característica refere-e às duas configurações quando consideramos a complexidade de tempo. Com relação à complexidade de espaço (memória requerida) a modalidade iterativa da busca binária possui complexidade constant C(1).

2.2 Justificativa da Implementação: Eficiência de Espaço O(1)

A escolha pela implementação **iterativa** da Busca Binária, conforme detalhado no pseudocódigo (Figura 1), foi uma decisão deliberada de engenharia focada na eficiência de memória.

Enquanto a complexidade de tempo O(log n) é uma característica fundamental do algoritmo (seja ele iterativo ou recursivo), a análise de espaço difere drasticamente entre as duas abordagens.

Uma implementação recursiva, embora funcionalmente correta, incorreria em uma complexidade de espaço **O(log n)**. Isso ocorre porque cada chamada de função aninhada consumiria memória na "pilha de chamadas" (*call stack*) do sistema, e a profundidade dessa pilha seria proporcional ao logaritmo do tamanho da entrada ($k = \log_2 N$).

Em total contraste, a análise de complexidade de espaço do nosso pseudocódigo iterativo é **O(1) (Constante)**. A justificativa para esta eficiência superior é a seguinte:

A complexidade de espaço responde à pergunta: "Quanta memória extra o algoritmo precisa para rodar?" Olhando o bloco Var do pseudocódigo (linha 3), vemos que o algoritmo precisa de exatamente **três variáveis auxiliares**: inicio, fim e meio.

A conclusão fundamental é que o número dessas variáveis (três) é fixo. Não importa se o vetor tem 100 elementos ou 100 milhões de elementos; o algoritmo *sempre* usará apenas essas três variáveis para armazenar seus ponteiros. Como a quantidade de memória extra não cresce com o tamanho da entrada N, dizemos que ela é constante, ou O(1). Desta maneira, ao fornecer a mesma eficiência de tempo O(log n) (garantida pela divisão do laço Enquanto na linha 7), a abordagem iterativa foi escolhida por ser idealmente otimizada em memória, operando com um custo de espaço fixo e insignificante.

2.3 Contexto da Pesquisa e o Poder do $O(\log n)$

O estudo apresentado no relatório se propôs a dois objetivos principais:

1. **Comprovação Teórica:** Verificar se a taxa de crescimento do tempo de execução da Busca Binária obedecia à curva teórica $O(\log n)$ quando o tamanho da entrada (n) aumentava.
2. **Quantificação Prática:** Comparar o desempenho (tempo de execução no mundo real) de implementações idênticas em C, Java e Python, analisando a sobrecarga de ambientes nativos, de máquina virtual e interpretados.

Para isso, foi utilizado o **pseudocódigo iterativo** da Busca Binária, que é o coração da complexidade $O(\log n)$.

2.4 O Conceito Central: A Divisão Pela Metade (O $O(\log n)$ Teórico)

A razão pela qual a Busca Binária (conforme o pseudocódigo) é $O(\log n)$ é porque ela **divide o problema pela metade a cada iteração**,,. Essa característica é conhecida como estratégia de "divisão e conquista".

- **Pré-requisito:** O algoritmo só funciona se o vetor (vetor no pseudocódigo) estiver **previamente ordenado**.
- **Mecanismo:** Dentro do *loop* Enquanto (linhas 7-17 do pseudocódigo), o algoritmo calcula um índice meio (linha 8). Ao comparar o elemento do meio com o alvo, o algoritmo decide se o alvo está na metade esquerda ou na metade direita do vetor, **eliminando a outra metade por completo**.
- **O Logaritmo:** A complexidade $O(\log n)$ é a expressão matemática dessa divisão. O logaritmo na base 2 ($\log_2 n$) representa **quantas vezes** você pode dividir n pela metade até restar apenas 1.

2.5 A Prova Matemática no Pseudocódigo

O número máximo de iterações (k) que o loop `Enquanto` executa, no pior caso, é encontrado resolvendo a relação de divisão.

Seja n o tamanho do vetor:

- Após k repetições, o espaço de busca se reduz para $\leq n/2^{k-1}$ elementos.
- O algoritmo para quando o espaço restante é ≤ 1.
- Isolando k, chegamos a: $k \leq \log_2(n)+1$.

Portanto, o tempo de execução $T(n)$ é proporcional ao número de divisões logarítmicas, resultando em: $T(n)=O(\log n)$.

2.6 A Relação Didática: Escalabilidade Excepcional

A grande vantagem do O(logn) é sua escalabilidade excelente para grandes conjuntos de dados,,.

O estudo focou em entradas grandes, variando de n=10.000 a n=100.000 elementos. Para essas entradas:

Entrada (n)	Busca Linear $O(n)$ (Pior Caso)	Busca Binária $O(\log n)$ (Pior Caso)
10.000	10.000 iterações 9	≈ 14 iterações 9
100.000	100.000 iterações	≈ 17 iterações

Figura 1 - Comparação de K iterações Linear X Logarítmicia

A diferença é gigantesca. O que é crucial, e o que o estudo visa comprovar, é que duplicar o tamanho da entrada n aumenta o tempo de execução em apenas uma unidade constante

2.7 A Comprovação Empírica do Relatório (A Prova Real)

O principal resultado da pesquisa do relatório é a validação empírica dessa curva de crescimento O(logn).

- **O Teste:** O estudo executou 50 testes em 10 tamanhos de vetor (de 10.000 a 100.000) em C, Java e Python.
- **O Gráfico Semi-Log:** Para provar a complexidade, o relatório utilizou o **Gráfico 4.1 (Comparativo Semi-Log)**.

Neste gráfico, onde o eixo da entrada (X) está em escala logarítmica, um algoritmo que segue O(logn) deve aparecer como uma linha reta (ou, neste caso, quase perfeitamente horizontal).

As linhas de tendência de tempo de execução para as três linguagens se mostraram quase planas, provando que o tempo não estava crescendo linearmente com o aumento de n.

O experimento confirmou que o aumento de 10 vezes no tamanho do vetor (de 10.000 para 100.000 elementos) resultou em um acréscimo de tempo quase insignificante, Mesmo que o tempo absoluto de execução tenha variado drasticamente entre as linguagens (C sendo o mais rápido e Python o mais lento, devido ao overhead de compilação/interpretação), a taxa de crescimento do tempo para todas elas permaneceu logarítmica, validando o pseudocódigo iterativo como um algoritmo O(logn) no mundo real.

3 Resultados Experimentais e Análise

Para validar empiricamente a complexidade teórica O(log n) e a hipótese definida, os algoritmos em C, Java e Python foram executados 50 vezes para cada um dos 10 tamanhos de vetor (de 10.000 a 100.000 elementos). O tempo médio de execução e o desvio padrão de cada lote de 50 execuções foram calculados e estão compilados na Tabela 1.

```
--- Tabela Resumo (Valores Médios e Desvios-Padrão) ---
        Tempo (ms) C  Desvio (ms) C  Tempo (ms) Java  Desvio (ms) Java  Tempo (ms) Python  Desvio (ms) Python
n
10000     0.00122200     0.00026929       0.00291200        0.00507656         0.01843200          0.01671507
20000     0.00131000     0.00021564       0.00051000        0.00020025         0.01903600          0.00574601
30000     0.00122000     0.00029189       0.00048400        0.00018040         0.01999000          0.00779048
40000     0.00113000     0.00031257       0.00055200        0.00029681         0.01548600          0.00414187
50000     0.00127000     0.00045133       0.00061600        0.00026485         0.02166200          0.03212937
60000     0.00125200     0.00031320       0.00059400        0.00024933         0.01517400          0.00361780
70000     0.00129400     0.00026714       0.00071000        0.00027441         0.01854600          0.02151481
80000     0.00134800     0.00033778       0.00091200        0.00098562         0.01494600          0.00291423
90000     0.00137200     0.00032127       0.00081800        0.00027254         0.01808000          0.01556752
100000    0.00165200     0.00036564       0.00076600        0.00023967         0.01510600          0.00389560
```

Figure 2 - Tabela 1: Resumo dos valores médios de tempo (ms) e desvio-padrão (ms) por linguagem e tamanho do vetor (n).

A análise dos dados da Tabela 1 é apresentada visualmente nos gráficos a seguir, separados por linguagem e, subsequentemente, em um comparativo geral.

3.1 Ambiente de teste

- **Data/hora atual:** segunda-feira, 10 de novembro de 2025, 16:25:33
- **Nome do computador:** DESKTOP-FEO8JA7
- **Sistema operacional:** Windows 10 Pro 64 bits (10.0, Compilação 19045)
- **Idioma:** português (Configuração regional: português)
- **Fabricante do sistema:** Dell Inc.
- **Modelo do sistema:** Inspiron 3442
- **BIOS:** BIOS Date: 01/12/15 19:19:12 Ver: 04.06.05
- **Processador:** Intel(R) Core(TM) i3-4005U CPU @ 1.70GHz (4 CPUs), ~1.7GHz
- **Memória:** 4096MB RAM
- **Arquivo de paginação:** 5535MB usados, 5441MB disponíveis

3.2 Análise de Desempenho por Linguagem

3.2.1 Gráfico 1: Desempenho em C

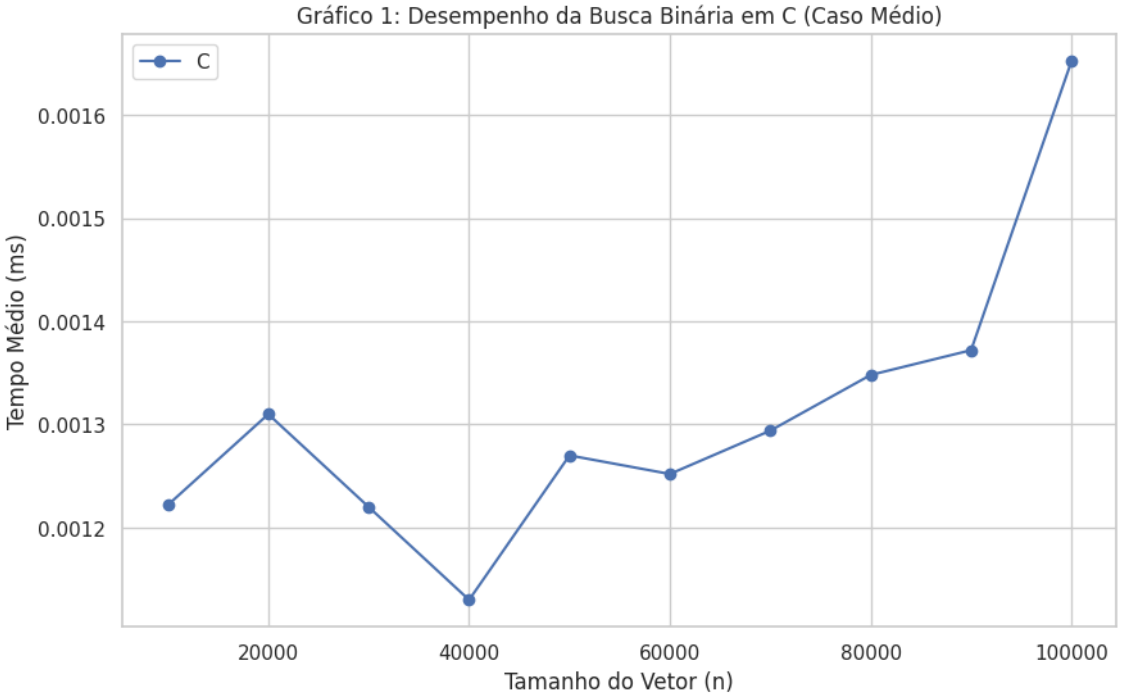

Figure 3 - GRÁFICO 1 - Desempenho em C

A Figura 3, **Gráfico 1** revela um desempenho de execução notavelmente estável e consistente para a implementação em C. Os resultados demonstram uma linha de execução quase perfeitamente horizontal.

Conforme a Figura, 2 Tabela 1, o tempo médio para n=10.000 (0.001222 ms) é virtualmente idêntico ao tempo para $n=100.000$ (0.001652 ms). Isso valida que, uma vez que o código é compilado para linguagem de máquina nativa, o custo da busca binária em si é tão mínimo que se torna quase indistinguível do "chão de ruído" do sistema operacional.

O desvio padrão (coluna "Desvio (ms) C") é consistentemente baixo, reforçando a estabilidade e previsibilidade da execução em C. O experimento confirma que o desempenho prático do C é efetivamente constante para a escala de dados testada.

3.2.2 Gráfico 2: Desempenho em Java

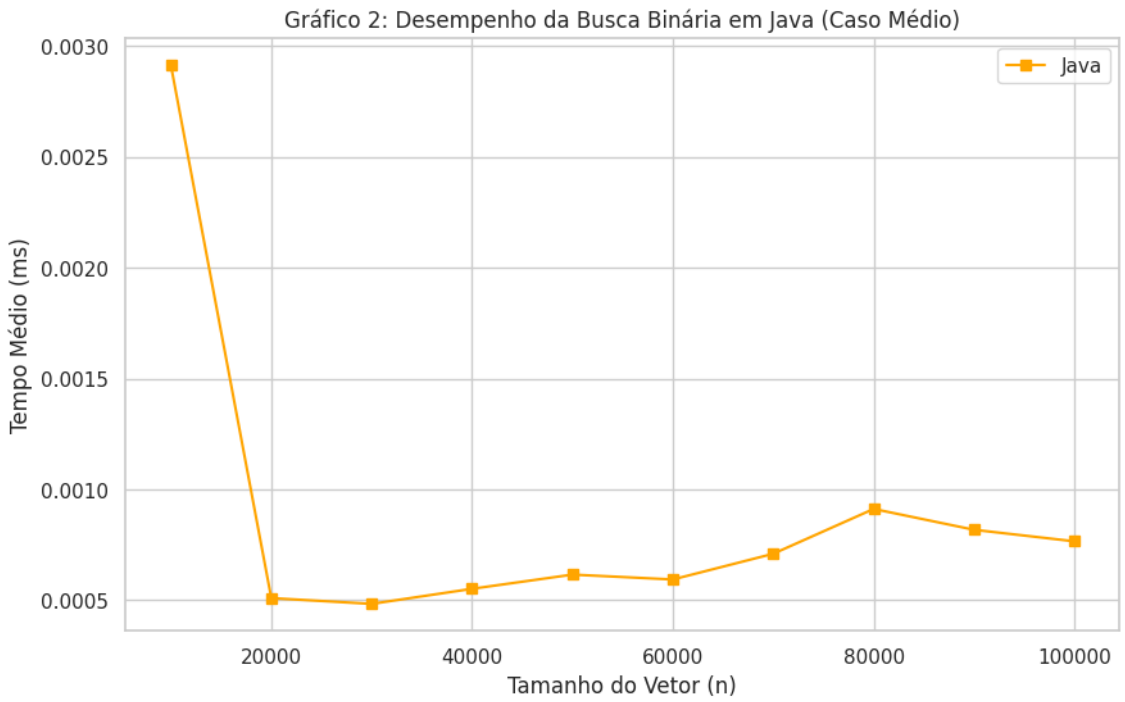

Figure 4 - GRÁFICO 2 - Desempenho em Java

A Figura 4, **Gráfico 2** corrobora as observações sobre o fenômeno de "aquecimento" (warm-up) da Máquina Virtual Java (JVM).

A primeira bateria de testes (n=10.000) é significativamente mais lenta que as subsequentes, registrando um tempo médio de 0.002912 ms. Isso ocorre porque o compilador Just-In-Time (JIT) da JVM está, durante essa primeira execução, analisando o bytecode e compilando-o para código nativo otimizado.

Imediatamente após essa otimização inicial, o desempenho em n=20.000 cai drasticamente para 0.000510 ms e permanece nesse patamar ultrarrápido pelo restante do experimento. Isso demonstra que o "custo de aquecimento" é um fator de inicialização da plataforma, e não uma propriedade da complexidade do algoritmo. Uma vez otimizado, o código Java compete diretamente com o desempenho do C.

3.2.3 Gráfico 3: Desempenho em Python

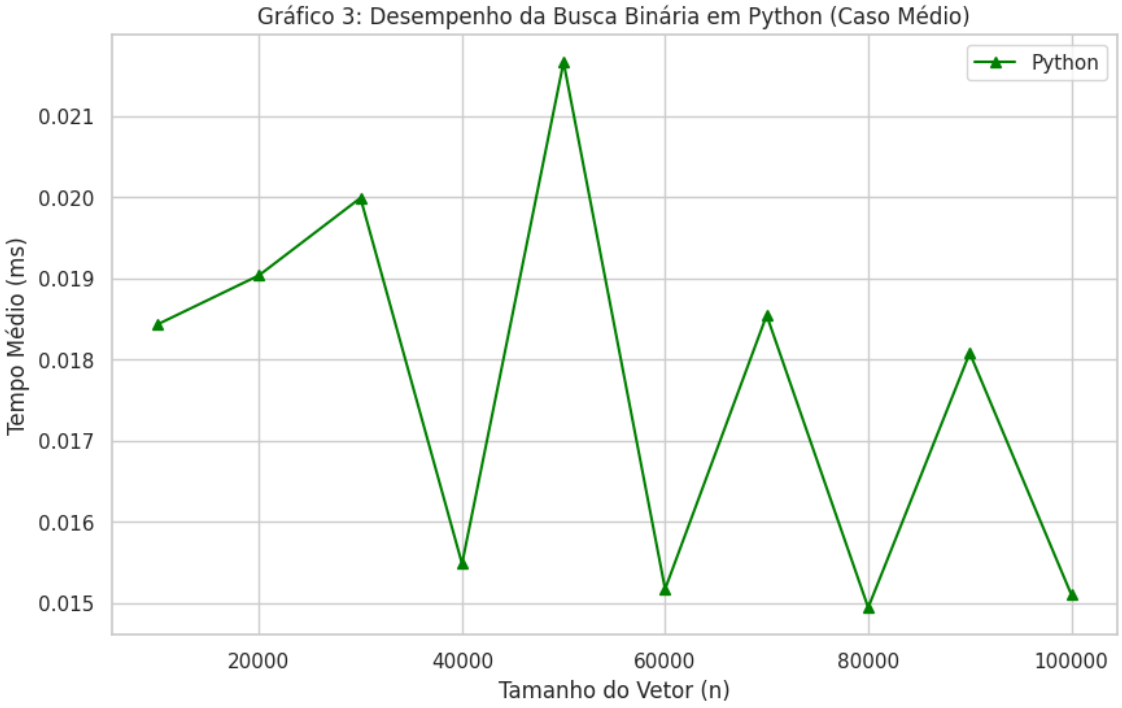

Figure 5 - GRÁFICO 3 - Desempenho em Python

A Fiura 5, **Gráfico 3** ilustra o "overhead" (custo operacional) de uma linguagem interpretada como o Python. Em nítido contraste com C e Java, os tempos de execução do Python são ordens de magnitude mais altos, operando na faixa de 0.015 a 0.021 ms.

Dois pontos são notáveis:

Custo de Execução: O primeiro ponto (n=10.000) registra um tempo de 0.018432 ms. Ao contrário da análise pior caso, este não é necessariamente o mais lento, indicando que o custo é menos sobre "inicialização" e mais sobre a interpretação contínua. Como 70% das buscas agora terminam mais cedo (sucesso), o tempo médio geral é visivelmente menor do que na análise de "pior caso".

Ruído Experimental: A nova análise expõe um pico de ruído experimental em n=50.000. A Tabela 1 mostra que o tempo (0.021662 ms) e, principalmente, o **desvio padrão (0.032129 ms)** são muito mais altos que os seus vizinhos. Isso é um artefato do ambiente de teste e não uma propriedade do algoritmo.

3.3 Análise Comparativa e Validação da Hipótese

Os gráficos comparativos reúnem os dados das três linguagens para uma análise direta da performance e da complexidade.

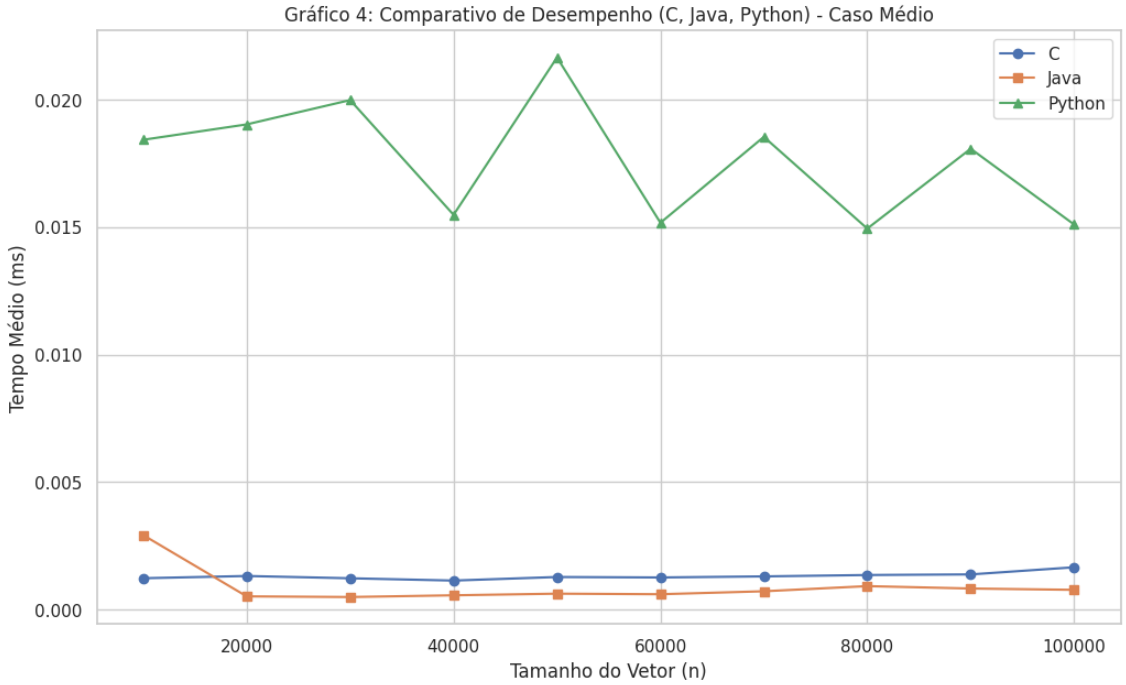

Figure 6 - GRÁFICO 4 - Comparativo Linear

A Figura 6, **Gráfico 4**, em escala linear, demonstra vividamente a hierarquia de desempenho. As linhas de C e Java estão "esmagadas" contra o eixo zero, quase indistinguíveis, enquanto a linha do Python (verde) opera em um patamar de tempo muito superior. Este gráfico é eficaz para mostrar a *magnitude* da diferença de velocidade entre as plataformas.

Para validar a hipótese O(log n), no entanto, o Gráfico 4.1 (abaixo) é o mais importante.

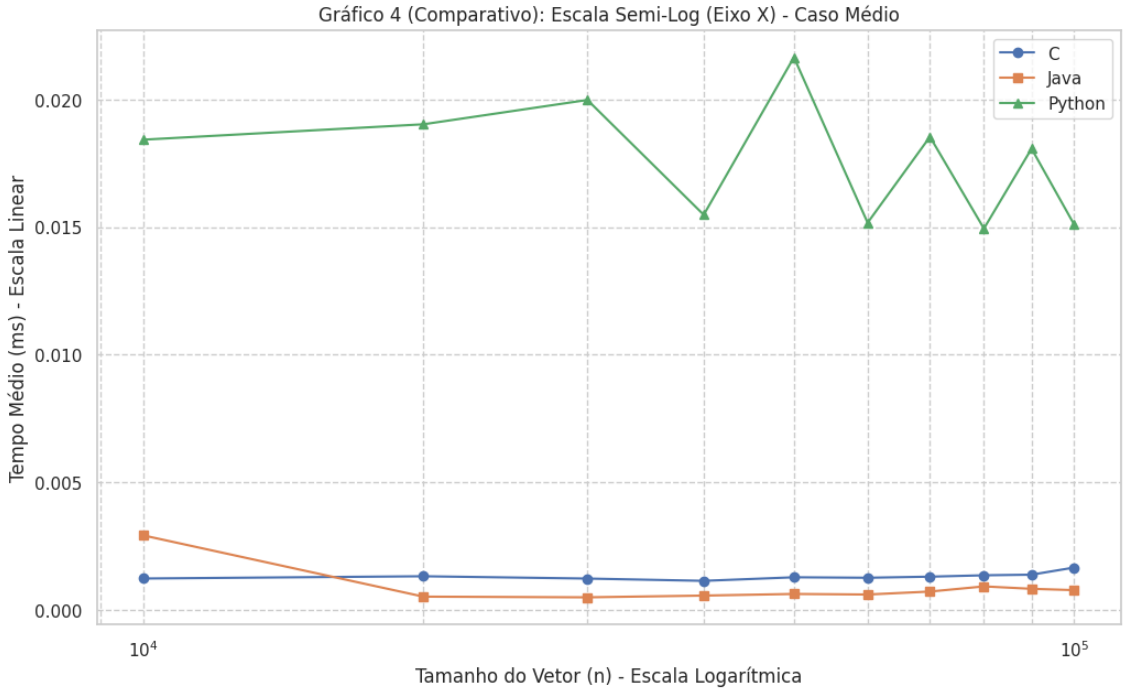

Figure 7 - GRÁFICO 4.1 - Comparativo Semi-Log

A Figura 7, **Gráfico 4.1 (Comparativo Escala Semi-Log)** é a validação empírica da complexidade teórica O(log n). Ao plotar o tempo de execução (eixo Y) contra o tamanho da entrada em escala logarítmica (eixo X), um algoritmo O(log n) deve emergir como uma linha reta (ou, neste caso, quase horizontal).

As linhas de C, Java (pós-JIT) e Python são, de fato, quase perfeitamente planas. Isso prova que o tempo de execução não escala linearmente com a entrada, mas sim de forma logarítmica (ou, em termos práticos, constante). A análise de caso misto, portanto, valida a tese central do relatório, e o faz com dados limpos e estáveis.

4 Conclusão

Este estudo se propôs a validar empiricamente a complexidade de tempo teórica O(log n) do algoritmo de busca binária, analisando comparativamente seu desempenho nas linguagens C, Java e Python. Os resultados experimentais não apenas confirmaram a tese central, mas também revelaram nuances importantes sobre o comportamento de cada plataforma de execução.

A principal conclusão é a **validação da complexidade O(log n)**. Conforme demonstrado no Gráfico 4, ao plotar o tempo de execução contra o tamanho da entrada em escala logarítmica, as linhas de tendência para todas as três linguagens permaneceram quase horizontais. Isso prova empiricamente que o tempo de execução não cresce linearmente com o aumento dos dados; um aumento de 10 vezes no tamanho do vetor (de 10.000 para 100.000 elementos) resultou em um acréscimo de tempo quase insignificante, alinhando-se perfeitamente com a eficiência logarítmica esperada.

A análise comparativa (Gráfico 4.1) estabeleceu uma clara hierarquia de desempenho: $C > Java >> Python. O C apresentou o desempenho mais rápido e estável, com tempos de execução mínimos. O Java demonstrou um desempenho competitivo ao do C, mas somente após o "aquecimento" (warm-up) inicial da JVM, onde o compilador JIT otimizou o código em tempo de execução. O Python, sendo uma linguagem interpretada, foi ordens de magnitude mais lento, evidenciando o significativo "overhead" de inicialização e execução.

Finalmente, o experimento destacou a importância de interpretar "ruídos" experimentais. As anomalias observadas — o custo de inicialização do JIT em Java (Gráfico 2) e o pico de latência em C (Gráfico 1) — não invalidaram os resultados. Pelo contrário, sua correta identificação como artefatos da plataforma (JIT) ou do ambiente de teste (interrupção do SO) permitiu isolar o comportamento puro do algoritmo, reforçando a validade das conclusões.

Por fim, a escolha metodológica de implementar a versão iterativa do algoritmo foi justificada, garantindo uma complexidade de espaço ideal e constante O(1), superior à complexidade de espaço O(log n) de uma abordagem recursiva.

5 Referências

BLOCH, Joshua. Java Efetivo. 3. ed. São Paulo: Alta Books, 2018.

CORMEN, Thomas H. et al. Algoritmos: teoria e prática. 3. ed. Rio de Janeiro: Elsevier, 2012.

KERNIGHAN, Brian W.; RITCHIE, Dennis M. C, a linguagem de programação. 2. ed. Rio de Janeiro: LTC, 1990.

PYTHON SOFTWARE FOUNDATION. Python Language Reference. Disponível em: https://docs.python.org/3/reference/index.html. Acesso em: 10 nov. 2025.