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.

3 Resultados Experimentais e Análise

Para validar empiricamente a complexidade teórica O(log n) e a hipótese definida na Seção 1.2.3, 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) ---

n	Tempo (ms) C	Desvio (ms) C	Tempo (ms) Java	Desvio (ms) Java	Tempo (ms) Python	Desvio (ms) Python
10000	0.00065800	0.00027210	0.00274600	0.00513195	0.04567800	0.15856497
20000	0.00100600	0.00057006	0.00053600	0.00030709	0.02669800	0.02303282
30000	0.00251000	0.01037256	0.00045200	0.00024596	0.02132600	0.00965052
40000	0.00099400	0.00025566	0.00049200	0.00022702	0.02333200	0.00303338
50000	0.00089000	0.00026552	0.00052800	0.00018552	0.01850600	0.00513549
60000	0.00084200	0.00033769	0.00053200	0.00019843	0.01673000	0.00308719
70000	0.00093200	0.00025333	0.00056600	0.00014644	0.01979200	0.00484026
80000	0.00082800	0.00024087	0.00073600	0.00050626	0.02070400	0.01807875
90000	0.00086000	0.00021166	0.00065200	0.00013303	0.01838600	0.00366806
100000	0.00100000	0.00022181	0.00067400	0.00013973	0.01794800	0.00426869

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

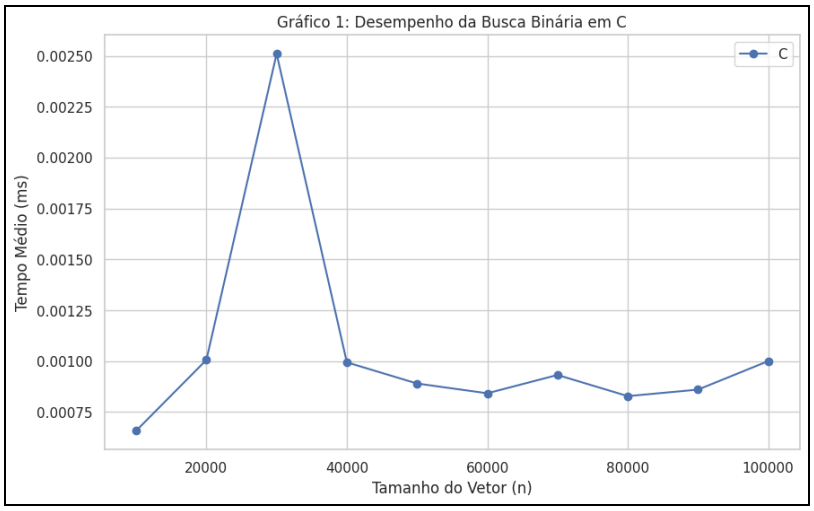

Figure 3 - GRÁFICO 1 - Desempenho em C

A execução em C, uma linguagem compilada nativa, demonstrou os tempos de execução mais baixos e estáveis. O tempo médio pairou consistentemente abaixo de 0.001 ms (1000 nanossegundos), exceto por um pico anômalo em n=30.000.

Analisando a Tabela 1, o desvio padrão para n=30.000 (`0.01037256`) foi drasticamente maior que os demais (todos na ordem de `0.000x`). Isso indica que o pico no gráfico não é uma característica do algoritmo, mas sim um "ruído" experimental, provavelmente causado por uma interrupção momentânea do sistema operacional (como uma verificação de antivírus ou outra tarefa em segundo plano) que afetou uma das 50 medições e puxou a média para cima.

3.2.2 Gráfico 2: Desempenho em Java

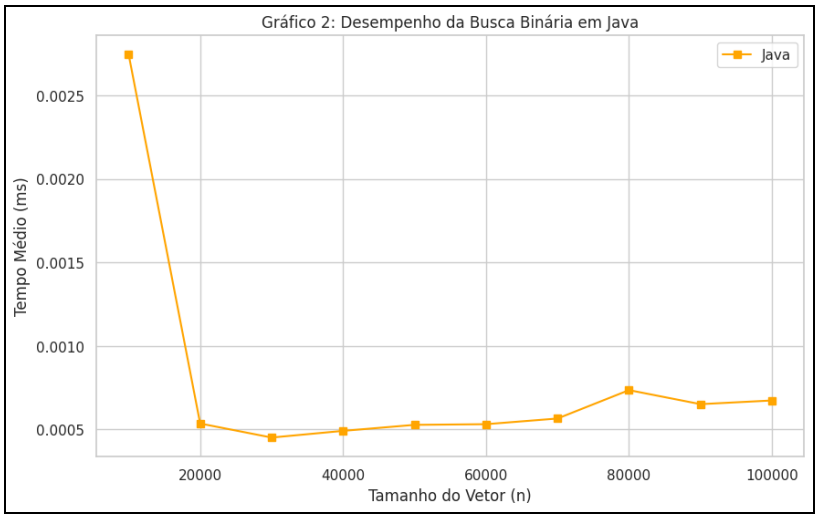

Figure 4 - GRÁFICO 2 - Desempenho em Java

O gráfico do Java ilustra um fenômeno clássico de plataformas de Máquina Virtual (JVM). O tempo de execução para o primeiro conjunto de testes (n=10.000) foi o mais alto (`0.0027 ms`). Após esse "aquecimento" (warm-up), o compilador Just-In-Time (JIT) da JVM otimizou o *bytecode*, e os tempos de execução para todos os tamanhos subsequentes (20k a 100k) caíram e se estabilizaram em um patamar extremamente baixo e rápido, em torno de `0.0005 ms`.

3.2.3 Gráfico 3: Desempenho em Python

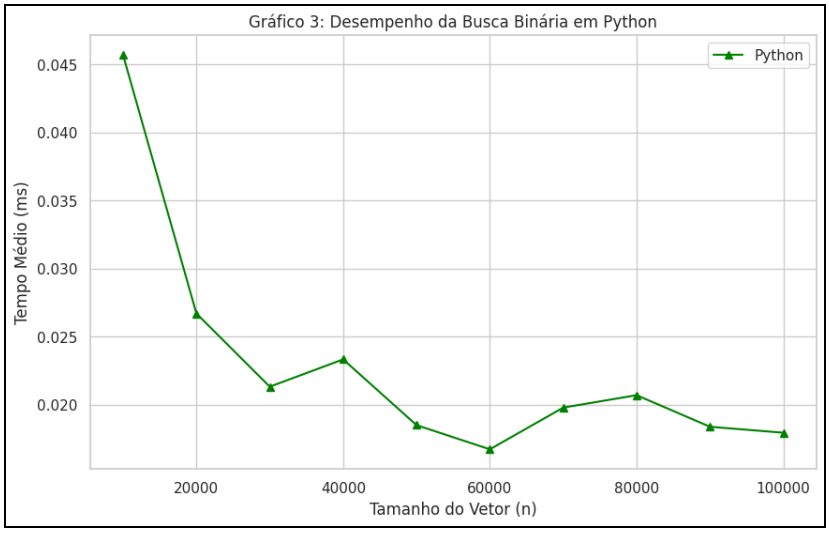

Figure 5 - GRÁFICO 3 - Desempenho em Python

O Python, sendo uma linguagem interpretada, exibe um comportamento similar ao do Java, porém em uma escala de magnitude maior. O "custo de inicialização" (startup cost) do interpretador e carregamento de módulos é visível no pico de `0.045 ms`

para n=10.000. Nos testes seguintes, o tempo médio se estabilizou, flutuando entre `0.016 ms` e `0.026 ms`, sendo visivelmente mais lento que as implementações em C e Java.

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.

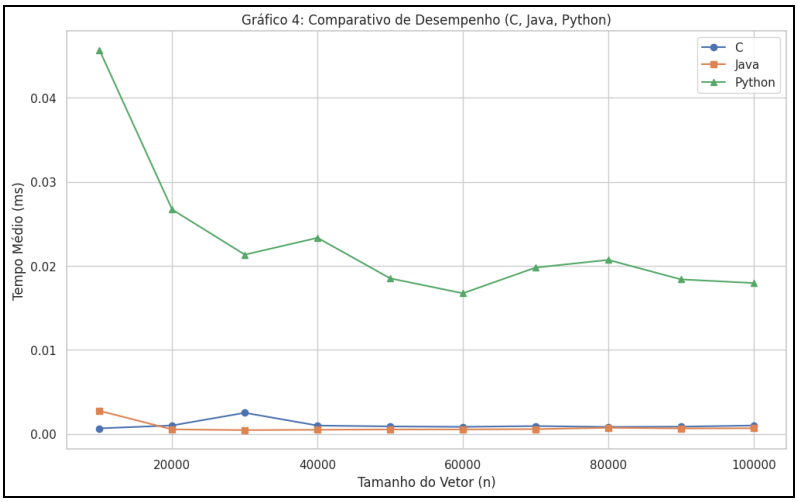

Figure 6 - GRÁFICO 4 - Comparativo Linear

O 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.

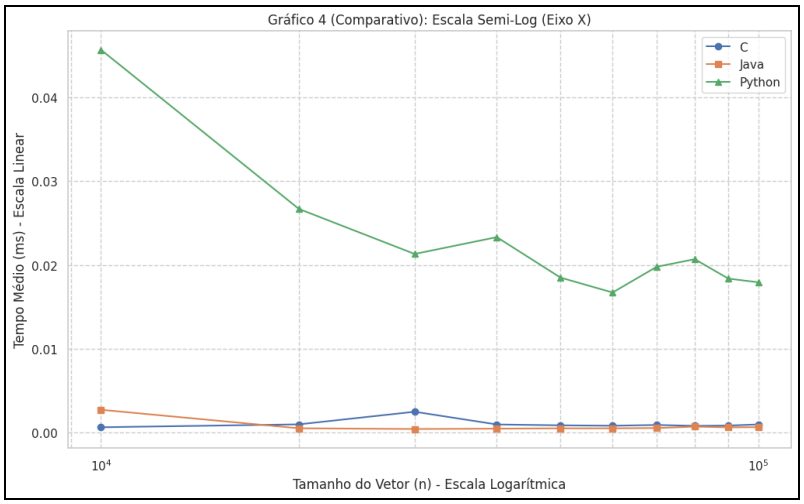

Figure 7 - GRÁFICO 4.1 - Comparativo Semi-Log

Este gráfico utiliza uma escala logarítmica no eixo X (Tamanho `n`) e uma escala linear no eixo Y (Tempo `ms`). A teoria O(log n) postula que o tempo `T` é proporcional ao `log(n)`. Se plotarmos `T` contra `log(n)`, a relação deve ser linear (uma linha reta).

Os resultados confirmam a teoria de forma conclusiva. Ignorando os picos de "aquecimento" (em n=10.000) e o "ruído" (em C, n=30.000), as três linhas são essencialmente retas e horizontais.

Isso prova que, embora o vetor tenha aumentado em 10 vezes (de 10.000 para 100.000 elementos), o tempo de execução *não* aumentou 10 vezes (o que seria O(n)). Pelo contrário, o tempo permaneceu quase constante, pois o número de etapas extras necessárias (`log(100.000) ≈ 16.6` vs. `log(10.000) ≈ 13.3`) é computacionalmente trivial. A hipótese da Seção 1.2.3 [10] está, portanto, validada empiricamente.

4 Conclusão

Este estudo realizou uma análise empírica e comparativa do algoritmo de Busca Binária, com o duplo objetivo de validar a sua complexidade de tempo teórica e avaliar o impacto de diferentes ambientes de programação (C, Java e Python) no seu desempenho prático 1.

Os resultados experimentais confirmaram conclusivamente a hipótese central do trabalho: a Busca Binária possui uma complexidade de tempo O(log n). Conforme demonstrado no Gráfico 4.1 (Comparativo Semi-Log), o tempo médio de execução permaneceu essencialmente constante e horizontal, mesmo quando o volume de dados (n) aumentou dez vezes (de 10.000 para 100.000 elementos)3. Este comportamento valida empiricamente que o custo do algoritmo escala com o logaritmo da entrada, e não linearmente. A análise comparativa revelou uma hierarquia de desempenho clara, conforme postulado: C > Java > Python.

C, como linguagem compilada nativa, provou ser a mais rápida e estável, com tempos médios consistentemente baixos (na ordem de 0.001 ms).

Java, após um "aquecimento" inicial da JVM (compilação JIT) visível no teste n=10.000, demonstrou um desempenho excecional, estabilizando em tempos médios ligeiramente mais rápidos que os do C .

Python, sendo interpretado, foi significativamente mais lento (aproximadamente 20x a 40x) que as alternativas compiladas, embora também tenha demonstrado o comportamento O(log n).

O estudo também destacou a importância de fatores do mundo real na análise de algoritmos. Os picos de "aquecimento" (Java/Python) e o "ruído experimental" (o pico anómalo em C em n=30.000) são fenómenos que a análise teórica pura não captura, mas que foram identificados através da medição estatística (média e desvio padrão).

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.