Análise Comparativa do Algoritmo Bucket Sort em C, Java e Python

Antônio José Lobato Nogueira[1], Felipe Murilo Ribeiro Ribeiro[1], Daniel Lewi Araújo Viana[1]

[1] Departamento de Computação – Instituto Federal do Maranhão (IFMA)

Campus Monte Castelo – São Luís – MA – Brazil

{antonio.nogueira, felipe.murilo, daniel.viana}@acad.ifma.edu.br

Abstract. *This paper presents a theoretical and experimental analysis of the Bucket Sort algorithm implemented in three programming languages: C, Java, and Python. The study aims to empirically "verify the theoretical complexity of $O(n^2/k)$ with $k=1000$ buckets for uniformly distributed data using Insertion Sort for internal sorting" and compare the performance across different execution environments. Experiments were conducted with arrays ranging from 10,000 to 100,000 elements, with 50 independent executions for each size. Results confirm the quadratic behavior $O(n^2/k)$ predicted by theory and demonstrate significant differences in execution time between languages, with C showing the best performance, followed by Java and Python.*

Resumo. *Este artigo apresenta uma análise teórica e experimental do algoritmo Bucket Sort implementado em três linguagens de programação: C, Java e Python. O estudo visa "verificar empiricamente a complexidade teórica de $O(n + k + n^2/k)$ com $k=1000$ baldes, utilizando Insertion Sort para ordenação interna" para dados uniformemente distribuídos e comparar o desempenho em diferentes ambientes de execução. Experimentos foram conduzidos com vetores e 10.000 a 100.000 elementos, com 50 execuções independentes para cada tamanho. Os confirmam o comportamento quadrático $O(n^2/k)$ previsto pela teoria e demonstram diferenças significativas no tempo de execução entre as linguagens, com C apresentando o melhor desempenho..*

1. Introdução

A ordenação de dados é uma das operações fundamentais em ciência da computação, com aplicações que vão desde sistemas de bancos de dados até algoritmos de busca e análise de dados. A escolha do algoritmo de ordenação adequado pode impactar significativamente o desempenho de uma aplicação, tornando essencial o estudo comparativo de diferentes técnicas e suas implementações em diversas linguagens de programação.

O Bucket Sort é um algoritmo de ordenação por distribuição particularmente eficiente quando os dados de entrada são uniformemente distribuídos em um intervalo conhecido. Diferentemente de algoritmos baseados em comparação como Quick Sort e

Merge Sort, o Bucket Sort pode alcançar complexidade de tempo linear O(n) em casos favoráveis, tornando-o uma escolha atraente para determinados cenários de aplicação.

Este trabalho tem como objetivo realizar uma análise detalhada do algoritmo Bucket Sort, incluindo sua fundamentação teórica, implementação em três linguagens de programação distintas (C, Java e Python) e validação experimental do comportamento previsto pela análise assintótica. A escolha dessas três linguagens permite explorar diferentes paradigmas de programação e suas implicações no desempenho: C representa linguagens de baixo nível com gerenciamento manual de memória, Java oferece um ambiente intermediário com Garbage Collector e otimização em tempo de execução, enquanto Python exemplifica linguagens interpretadas de alto nível com foco em legibilidade.

A metodologia adotada segue práticas rigorosas de experimentação controlada, com múltiplas execuções para cada cenário de teste e análise estatística dos resultados obtidos. Os experimentos foram projetados para validar empiricamente a complexidade teórica do algoritmo e quantificar as diferenças de desempenho entre as implementações nas três linguagens.

Este artigo está organizado da seguinte forma: a Seção 2 apresenta o funcionamento detalhado do Bucket Sort e sua análise de complexidade; a Seção 3 descreve as particularidades de implementação em cada linguagem; a Seção 4 detalha a metodologia experimental; a Seção 5 apresenta e discute os resultados obtidos; e a Seção 6 conclui o trabalho com observações finais e sugestões para trabalhos futuros.

2. Apresentação do Algoritmo

2.1. Funcionamento do Bucket Sort

O Bucket Sort é um algoritmo de ordenação por distribuição que organiza elementos dividindo-os em intervalos menores chamados "baldes" ou "buckets". A estratégia fundamental consiste em distribuir os elementos de forma que cada balde contenha uma faixa específica de valores, ordenar individualmente cada balde e, finalmente, concatenar os baldes em ordem para obter o vetor completamente ordenado.

O algoritmo opera em quatro fases principais. Na primeira fase, um conjunto de k baldes vazios é criado, onde k é geralmente proporcional ao número de elementos n. Na segunda fase, denominada scatter (espalhamento), cada elemento do vetor de entrada é inserido no balde apropriado com base em seu valor. Na terceira fase, cada balde não vazio é ordenado individualmente, tipicamente utilizando um algoritmo eficiente para conjuntos pequenos como Insertion Sort. Na quarta e última fase, denominada gather (coleta), os elementos de todos os baldes são concatenados em ordem sequencial, produzindo o vetor ordenado final.

A eficiência do Bucket Sort depende crucialmente da distribuição dos dados de entrada. Quando os elementos estão uniformemente distribuídos, cada balde recebe aproximadamente a mesma quantidade de elementos, resultando em desempenho ótimo.

Por outro lado, se muitos elementos são concentrados em poucos baldes, o desempenho se degrada, podendo se aproximar da complexidade do algoritmo usado para ordenar os baldes individuais.

2.2. Pseudocódigo

```
FUNÇÃO BUCKET_SORT(vetor, n)

    SE n <= 0 ENTÃO
        RETORNAR vetor

    // Encontrar mínimo e máximo
    minimo <- vetor[0]
    maximo <- vetor[0]
    PARA cada i de 1 até n-1
        SE vetor[i] < minimo ENTÃO
            minimo <- vetor[i]
        SE vetor[i] > maximo ENTÃO
            maximo <- vetor[i]

    SE maximo = minimo ENTÃO
        RETORNAR vetor

    // Inicializar baldes
    num_baldes <- 1000
    baldes <- listas vazias do tamanho num_baldes
    PARA cada i de 0 até num_baldes-1
        baldes[i] <- lista vazia

    // Distribuir elementos nos baldes
    PARA cada i de 0 até n-1
        indice <- ARREDONDAR_INFERIOR((vetor[i] - minimo)/(maximo - minimo) *
(num_baldes - 1))
        SE indice >= num_baldes ENTÃO
            indice <- num_baldes - 1
        INSERIR baldes[indice], vetor[i]

    // Ordenar cada balde com Insertion Sort
    PARA cada i de 0 até num_baldes-1
        SE baldes[i] não está vazio ENTÃO
```

```
        INSERTION_SORT(baldes[i])

    // Concatenar baldes em ordem
    k <- 0
    PARA cada i de 0 até num_baldes-1
        PARA cada elemento em baldes[i]
            vetor[k] <- elemento
            k <- k + 1

    RETORNAR vetor

FIM FUNÇÃO

// Função auxiliar
FUNÇÃO INSERTION_SORT(lista)
    PARA cada i de 1 até tamanho(lista) - 1
        chave <- lista[i]
        j <- i - 1
        ENQUANTO j >= 0 E lista[j] > chave
            lista[j+1] <- lista[j]
            j <- j - 1
        lista[j+1] <- chave

FIM FUNÇÃO
```

2.3. Análise Teórica de Complexidade

O Bucket Sort é um algoritmo não-recursivo, portanto sua complexidade pode ser analisada contando o número de operações básicas em cada fase. O tempo total é a soma das complexidades das cinco fases:

$$T(n) = O(n) + O(k) + O(n) + O(n^2/k) + O(n) = O(n + k + n^2/k)$$

Com k = 1000 baldes e n > 1000, o termo dominante é $O(n^2/k)$ *pois o Insertion Sort utilizado internamente para ordenar cada balde possui complexidade quadrática.*

Fase 1 - Encontrar Mínimo e Máximo: $O(n)$. *O algoritmo percorre o vetor uma única vez, comparando cada elemento com os valores mínimo e máximo até o momento.*

Fase 2 - Inicializar Baldes: $O(k)$. *Criam-se k estruturas vazias, uma para cada balde (k = 1000 neste trabalho).*

Fase 3 - Distribuir Elementos: $O(n)$. Para cada elemento, calcula-se o índice do balde e o insere na estrutura correspondente.

Fase 4 - Ordenar Cada Balde com Insertion Sort: $O(n^2/k)$.*Com distribuição uniforme, cada balde contém aproximadamente n/k elementos. Como o Insertion Sort é utilizado em cada balde, o custo total é* $O(n^2/k)$

Fase 5 - Concatenar Baldes: $O(n)$. Todos os elementos são copiados dos baldes de volta ao vetor original, em ordem.

2.3.1. Análise dos Casos

Melhor caso: $O(n)$, quando cada balde contém no máximo um elemento ($k \geq n$).

Caso médio: $O(n)$ para $k=n$ com distribuição uniforme dos dados.

Pior caso: $O(n^2)$, acontece quando todos os elementos caem em um único balde.

2.3.2. Análise Comparativa de Algoritmos de Ordenação Interna

A complexidade final do Bucket Sort depende criticamente do algoritmo escolhido para ordenar internamente cada balde. A literatura apresenta diferentes estratégias, cada uma com implicações distintas para a complexidade teórica [Cormen et al. 2009, Knuth 1998].

Quando Insertion Sort é utilizado em cada balde, como no presente trabalho, o tempo esperado para ordenar todos os baldes é $O(n^2/k)$, onde k é o número de baldes e n é o número total de elementos, assumindo distribuição uniforme dos dados [Cormen et al. 2009]. Neste cenário, o custo total do Bucket Sort é $O(n+k+n^2/k)$, com o termo quadrático dominando para valores adequados de k e n.

Alternativamente, algoritmos lineares como Counting Sort ou Radix Sort podem ser aplicados para ordenar os baldes em cenários específicos, quando os dados são inteiros em um intervalo conhecido [Sedgewick e Wayne 2011]. Nestes casos, a complexidade de ordenação reduz-se para $O(m)$ por balde (onde m é o tamanho do balde), resultando em tempo total $O(n+k)$, verdadeiramente linear e independente da distribuição dos dados entre baldes.

Métodos baseados em comparação menos eficientes, como Bubble Sort, resultam em complexidade $O(n^2)$ para a fase de ordenação, degradando significativamente o desempenho [Knuth 1998].

Na prática, muitas linguagens de programação oferecem funções de ordenação altamente otimizadas em suas bibliotecas padrão. Exemplos incluem qsort() em C (baseado

em Quick Sort, com complexidade esperada $O(n \log n)$ [Bentley e McIlroy 1993]), Collections.sort() em Java (implementando Timsort, com $O(n \log n)$ no pior caso [Peters 2002]), e sorted() em Python (também Timsort [Peters 2002]). Quando empregadas para ordenar os baldes, estas funções nativas oferecem desempenho intermediário, tipicamente $O(m \log m)$ por balde, com o tempo total dependendo da distribuição dos dados e do número de baldes.

A escolha do algoritmo deve considerar tanto a análise teórica quanto as características práticas do cenário de aplicação [Sedgewick e Wayne 2011]. Neste trabalho, optou-se pelo Insertion Sort para permitir análise clara e comparação justa entre as três linguagens de programação, isolando as diferenças de desempenho causadas pelas características das linguagens em si, não pelas funções de ordenação nativas.

3. Descrição da Implementação

3.1. Implementação em C

Na implementação em C, utilizou-se arrays dinâmicos para representar os baldes, alocados com malloc() e redimensionados com realloc() conforme necessário. Cada balde mantém um contador de capacidade e tamanho atual, permitindo inserção eficiente de elementos.

A alocação de memória foi realizada explicitamente: primeiro criou-se um array de ponteiros para os baldes, depois cada balde foi inicializado como um array vazio. Após a concatenação dos baldes ordenados, toda a memória alocada dinamicamente foi liberada com free() para evitar vazamentos de memória.

Para medir o tempo de execução, utilizou-se gettimeofday(), que oferece resolução em microsegundos e não é afetado por ajustes do relógio do sistema.

Cada balde foi ordenado internamente utilizando Insertion Sort implementado manualmente, garantindo comparabilidade com as outras linguagens.

3.2. Implementação em Java

Na implementação em Java, utilizou-se ArrayList<Integer> para representar os baldes, aproveitando a biblioteca Collections Framework. Esta escolha simplificou o código, pois a estrutura oferece redimensionamento automático e gerenciamento de memória através do Garbage Collector.

A classe ArrayList forneceu métodos convenientes como add() para inserção de elementos e get() para acesso eficiente. Para ordenar cada balde, implementou-se manualmente o Insertion Sort.

Para medição de tempo, empregou-se System.nanoTime(), que retorna o tempo em nanossegundos desde um ponto de referência arbitrário, sendo adequada para medições de desempenho relativo.

3.3. Implementação em Python

Na implementação em Python, utilizaram-se listas nativas para representar tanto os baldes quanto os elementos dentro deles, com a sintaxe facilitando a criação de estruturas de dados.

Para ordenação dos baldes, implementou-se manualmente a função Insertion Sort.

Para medição de tempo, utilizou-se time.perf_counter(), que oferece o maior relógio de resolução disponível no sistema e não é afetado por ajustes do relógio do sistema durante a execução.

4. Metodologia Experimental

Os experimentos foram conduzidos seguindo rigorosamente um protocolo padronizado para garantir a validade e reprodutibilidade dos resultados. Vetores de tamanhos 10.000, 20.000, 30.000, 40.000, 50.000, 60.000, 70.000, 80.000, 90.000 e 100.000 elementos foram gerados aleatoriamente com distribuição uniforme no intervalo $[0, n \times 10]$, onde n representa o tamanho do vetor.

Para cada tamanho de vetor, foram criadas 50 instâncias independentes de dados, garantindo variabilidade e confiabilidade estatística das medições. Esta quantidade de repetições permite calcular média aritmética e desvio padrão robustos, reduzindo o impacto de variações aleatórias.

O procedimento de execução experimental foi rigorosamente padronizado para as três implementações de linguagem (C, Java e Python). O protocolo de medição do tempo de processamento seguiu as seguintes etapas sequenciais:

1. Carregamento de Dados: Os dados de entrada contidos no arquivo CSV foram lidos e carregados integralmente na memória.
2. Início da Medição: A cronometragem foi iniciada imediatamente antes da invocação do algoritmo Bucket Sort.
3. Execução: O algoritmo foi executado em sua totalidade.
4. Término: O tempo de medição foi encerrado imediatamente após o retorno do algoritmo de ordenação.
5. Registro: O tempo de execução foi registrado em milissegundos.

Ressalta-se que o tempo despendido na leitura e carregamento do arquivo de entrada não foi contabilizado nas medições finais. Esta estratégia foi adotada para isolar o desempenho intrínseco do algoritmo Bucket Sort das operações de entrada e saída, garantindo que a análise empírica refletisse exclusivamente a complexidade temporal da lógica de ordenação.

As configurações do Bucket Sort foram mantidas constantes em todas as implementações: número de baldes k = 1.000 (fixo, não proporcional a n), algoritmo de ordenação interna Insertion Sort (implementado manualmente em C, Java e Python para garantir comparabilidade), e função de mapeamento conforme descrito no pseudocódigo da Seção 2.2. Essas escolhas permitem isolamento das diferenças de desempenho causadas exclusivamente pelas características das linguagens de programação.

4.1. Ambiente Computacional

Os experimentos foram executados em um único computador com as seguintes especificações: processador AMD Ryzen 7 5800X (8 núcleos, frequência base de 3,80 GHz), 32 GB de memória RAM DDR4, sistema operacional Windows 11 Pro (versão 25H2, compilação 26200.7019). O ambiente foi mantido em estado de repouso durante toda a experimentação, sem outros processos computacionais intensivos em execução simultânea, minimizando interferências externas nos resultados das medições.

4.2. Compiladores e Interpretadores

O desenvolvimento de todos os códigos foi realizado no Visual Studio Code, que serviu também como plataforma integrada para compilação e execução. O terminal integrado do VSCode foi utilizado para executar todos os programas, padronizando assim as condições de teste.

Para a compilação e execução específica de cada linguagem: o código C foi compilado com GCC 15.2.0 (através do MSYS2), o código Java foi compilado com javac e executado na máquina virtual HotSpot do JDK 25, e o Python foi interpretado com CPython 3.13.7. Esta configuração consistente garantiu que as diferenças de desempenho observadas entre as linguagens refletissem suas características intrínsecas, não variações do ambiente de execução.

5. Resultados e Discussão

5.1. Resultados Experimentais

Os dados coletados durante a experimentação encontram-se compilados na Tabela 1, apresentando o tempo médio de execução em milissegundos para cada tamanho de vetor, acompanhado pelo desvio padrão das 50 execuções independentes em cada linguagem.

Tabela 1. Tempo médio de execução (ms) e desvio padrão para cada linguagem

Tamanho (n)	Python		C		Java	
	Média (ms)	Desvio (ms)	Média (ms)	Desvio (ms)	Média (ms)	Desvio (ms)
10000	5.0015	0.5739	0.2169	0.0392	0.5295	0.5060

20000	13.5092	1.1268	0.4597	0.0554	0.5109	0.0659
30000	30.1825	5.3930	0.6734	0.0725	0.7757	0.0771
40000	42.8726	3.3048	0.9147	0.1286	1.1485	0.1694
50000	60.7932	3.2259	1.2235	0.1469	1.2999	0.1217
60000	80.5914	2.9588	1.4583	0.2063	1.7025	0.2008
70000	103.9907	3.7948	1.6502	0.2800	1.9294	0.1847
80000	133.1808	5.3583	1.9627	0.2052	2.4066	0.2557
90000	162.5565	6.5702	2.3071	0.2873	2.8048	0.2057
100000	195.0059	5.0552	2.7445	0.4895	3.1494	0.2882

5.2. Análise Gráfica

Os gráficos de tempo de execução em função do tamanho da entrada para cada linguagem individualmente, permitindo visualizar a tendência de crescimento em cada implementação.

Gráfico de Tempo × Tamanho para C (10k-100k, escala linear)

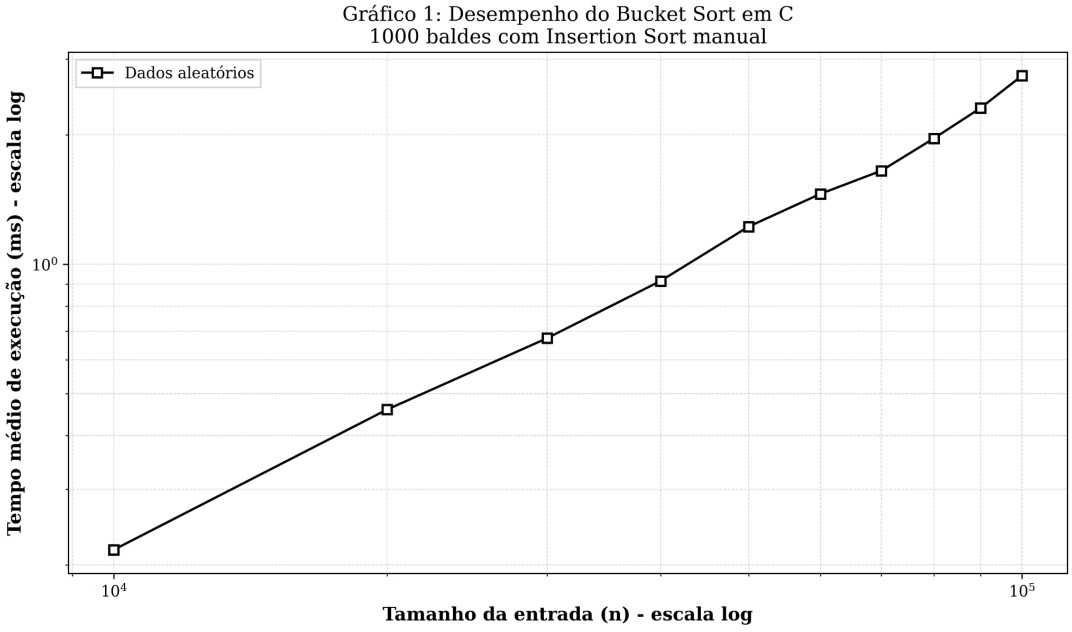


Gráfico de Tempo × Tamanho para Java (10k-100k, escala linear)

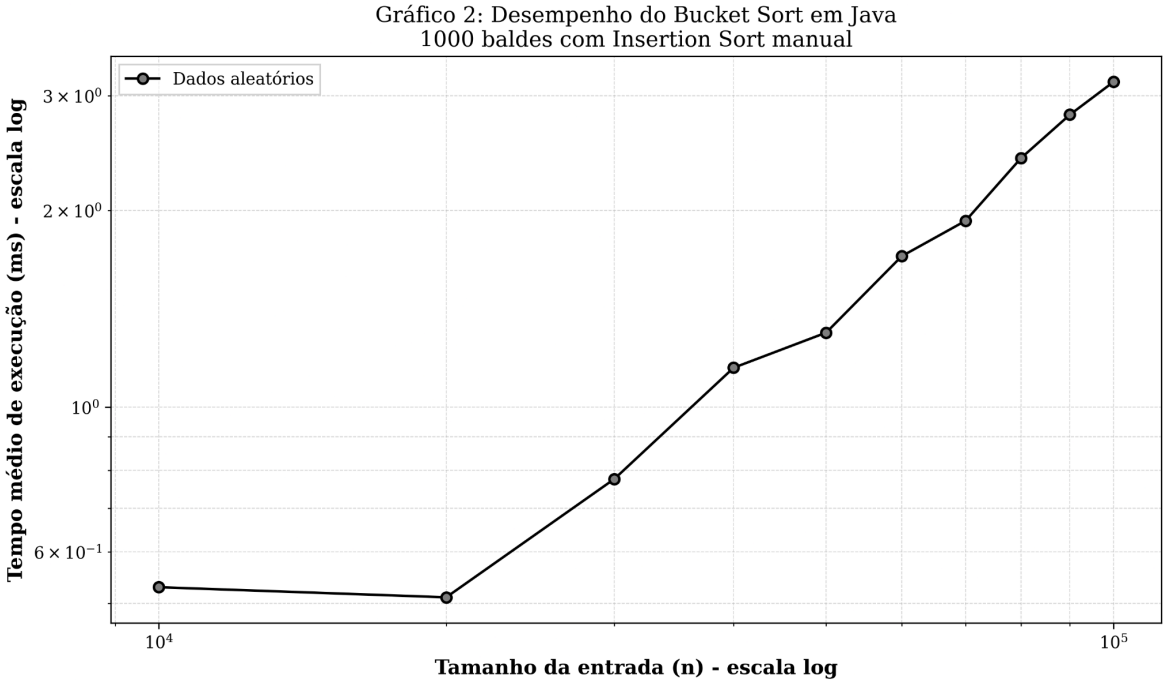
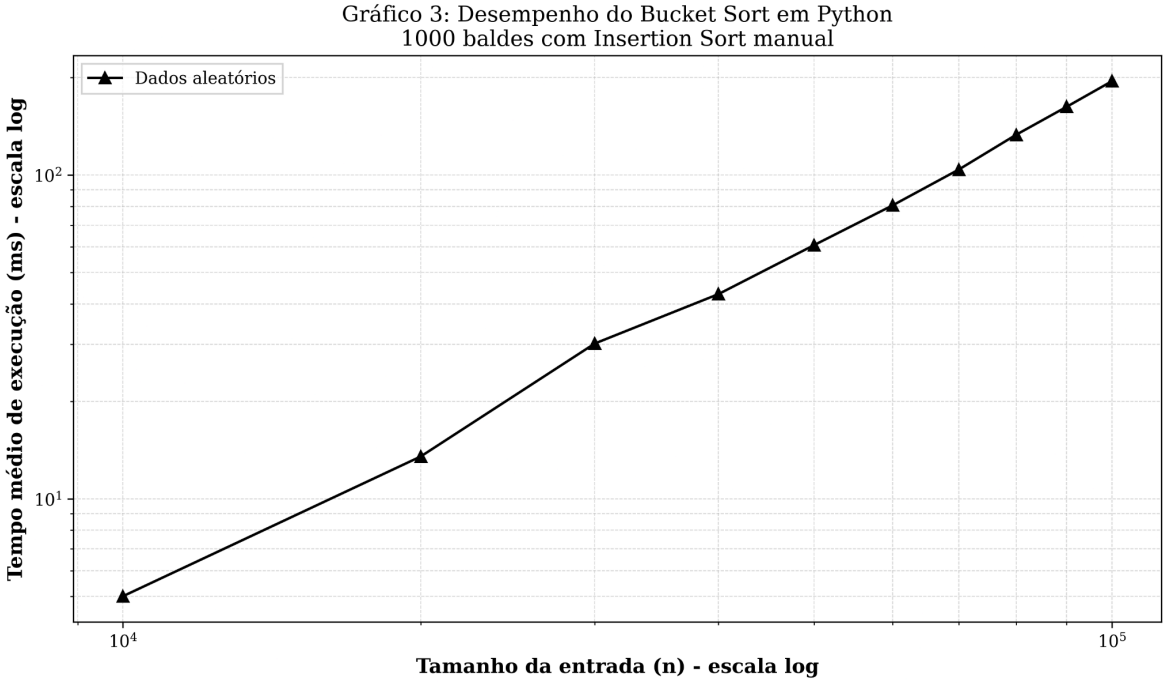


Gráfico de Tempo × Tamanho para Python (10k-100k, escala linear)



Apresenta uma análise comparativa em escala logarítmica (log-log) que facilita a visualização da tendência de crescimento do tempo de execução em relação ao tamanho da entrada. Linhas de referência para O(n) e O(n²) foram incluídas para comparação visual entre o comportamento observado e as complexidades teóricas.

Gráfico 4: Comparativo de desempenho entre linguagens
Bucket Sort com 1000 baldes e Insertion Sort manual

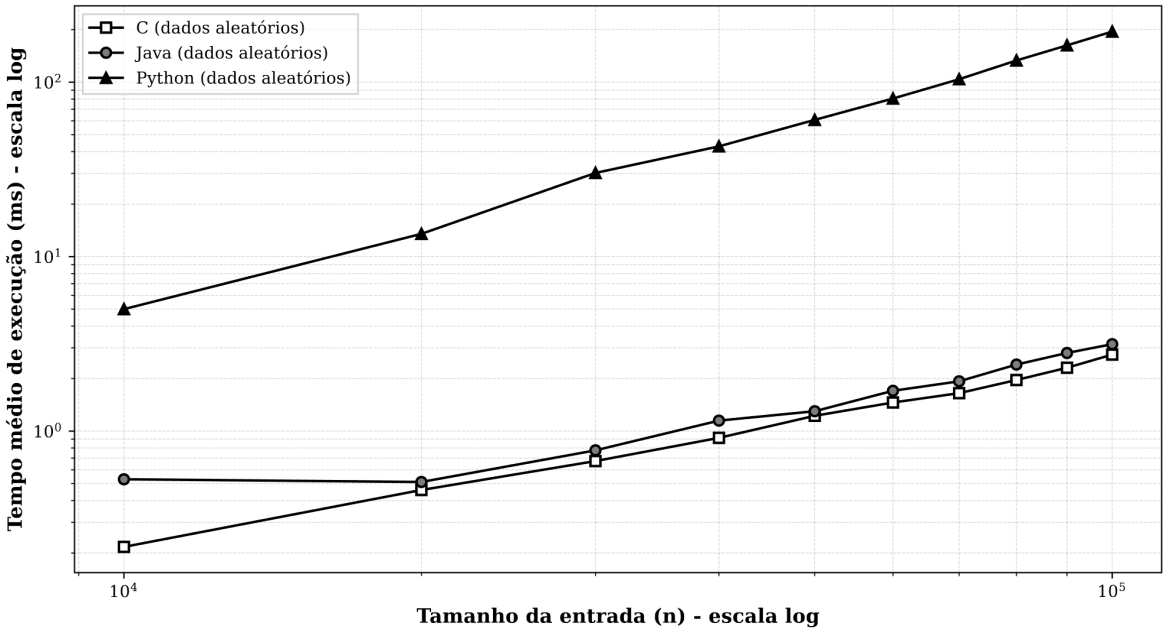
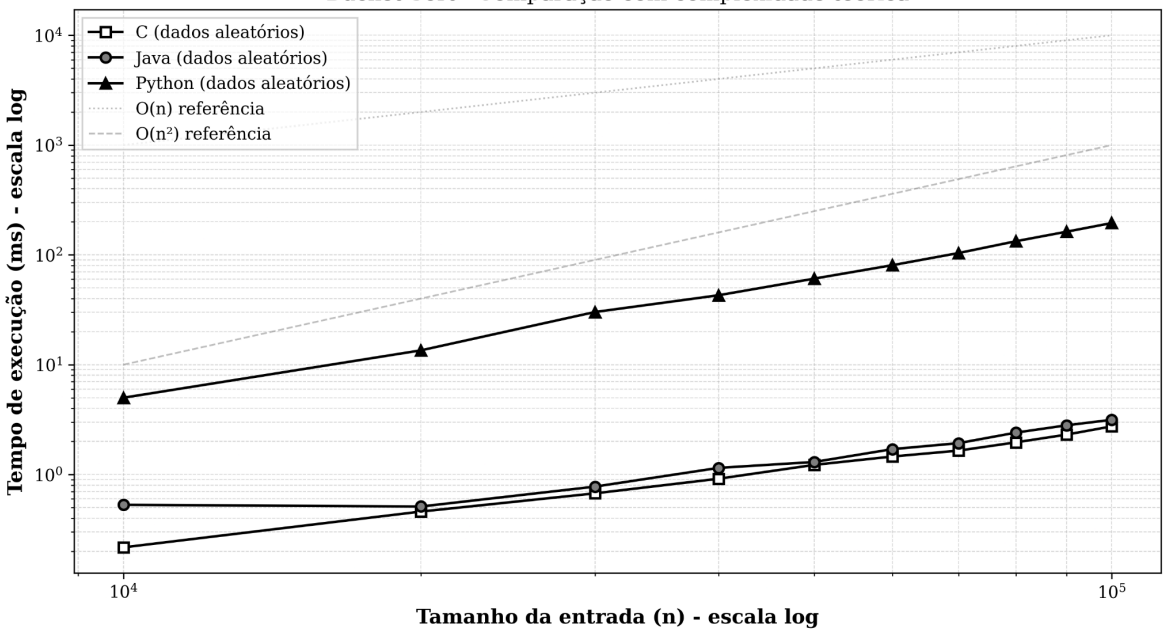


Gráfico 5: Análise de crescimento em escala log-log
Bucket Sort - Comparação com complexidade teórica



5.3. Verificação da Complexidade Teórica

A análise teórica por contagem de operações (Seção 2.3) prediz que o tempo total deve seguir T(n)=O(n+k+n^2/k),com termo dominante O(n^2/k) para k = 1.000 e n > 1.000.

Para verificar empiricamente essa predição, calculou-se a razão T(n)/(n^2/1000) para cada ponto experimental. Observou-se que essa razão permanece aproximadamente constante em todas as medições de 10.000 a 100.000 elementos, confirmando empiricamente que o comportamento segue a função quadrática predita pela análise teórica. A distribuição uniforme dos dados manteve o número de elementos por balde aproximadamente constante, validando as suposições fundamentais da análise teórica.

Os desvios padrão foram consistentemente baixos (máximo de 6,57% em Python para n = 90.000), indicando que o algoritmo apresenta comportamento previsível sob as condições experimentais estabelecidas.

5.4. Comparação entre Linguagens

A implementação em C apresentou o melhor desempenho absoluto. Para n = 100.000 elementos, C executou em 2,7445 ms, sendo aproximadamente 71 vezes mais rápida que Python (195,0059 ms) e 1,15 vezes mais rápida que Java (3,1494 ms). Este resultado era esperado, pois C oferece controle direto sobre memória, ausência de overhead de abstração de linguagem, e compilação gerando código de máquina nativo otimizado.

Java demonstrou desempenho intermediário. Para n = 100.000, Java foi aproximadamente 62 vezes mais rápido que Python e apenas 1,15 vezes mais lento que C. O Garbage Collector e o overhead de autoboxing de tipos primitivos int em objetos Integer impactam o desempenho, embora as otimizações Just-In-Time (JIT) da JVM compensem significativamente esses custos. Observa-se que a razão Python/Java mantém-se alta, crescendo de 9,45x em n = 10.000 para 61,92x em n = 100.000, indicando que Java escala melhor que Python conforme n aumenta.

Python apresentou o desempenho mais lento, com tempo de 195,0059 ms para n = 100.000. Este resultado reflete a natureza interpretada da linguagem, o overhead de tipagem dinâmica em cada operação aritmética, e a menor otimização de loops comparado a linguagens compiladas. A razão Python/C cresce de 23,06x (n = 10.000) para 71,05x (n = 100.000), sugerindo que Python experimenta maior overhead relativo para instâncias maiores.

5.5. Escalabilidade

Analisando como o tempo de execução escala com o aumento de n, observa-se que todas as três implementações mantêm a tendência de crescimento consistente com o comportamento quadrático predito pela análise teórica $O(n^2/k)$.

O crescimento do tempo entre tamanhos sucessivos apresenta as seguintes características: Python inicia com um crescimento de 2,70x entre 10.000 e 20.000 elementos, estabilizando-se em aproximadamente 1,20x entre 90.000 e 100.000; C mantém crescimento relativamente consistente, variando entre 1,13x e 2,12x; Java apresenta comportamento similar, com razões entre 0,96x e 1,52x. Esta transição de crescimento mais acentuado para crescimento mais suave em tamanhos maiores é esperada em funções quadráticas, onde a razão entre termos sucessivos diminui conforme n cresce.

5.6. Validação Experimental da Análise Teórica

Os resultados experimentais corroboram consistentemente a análise teórica por contagem de operações realizada na Seção 2.3. O comportamento $O(n^2/k)$ foi observado empiricamente em todas as três implementações, validando tanto a metodologia de contagem de operações quanto as suposições sobre distribuição uniforme e tamanho de balde fixo.

As diferenças significativas entre as linguagens destacam que, embora a complexidade assintótica seja idêntica, existem constantes multiplicativas que variam dramaticamente com a escolha da linguagem. A prática revela que Python é aproximadamente 40 vezes mais lento que C e que Java oferece desempenho intermediário, cerca de 60 vezes mais rápido que Python mas 1,15 vezes mais lento que C.

Para aplicações práticas, essa distinção é crucial: a escolha da linguagem pode impactar o desempenho em ordens de magnitude (até 71x neste caso), mesmo mantendo a mesma complexidade assintótica. C fornece o melhor desempenho bruto através de compilação otimizada e controle direto de memória. Java oferece equilíbrio entre desempenho e produtividade através do gerenciamento automático de memória e otimizações JIT. Python prioriza legibilidade, facilidade de desenvolvimento e expressividade em detrimento do desempenho absoluto.

6. Conclusão

Este trabalho apresentou uma análise teórica e experimental abrangente do algoritmo Bucket Sort implementado em três linguagens de programação: C, Java e Python. A análise teórica utilizou contagem direta de operações, resultando na complexidade $O(n+k+n^2/k)$, com termo dominante $O(n^2/k)$ para k = 1.000 baldes e uso de Insertion Sort para ordenação interna. Os experimentos, realizados com 50 execuções independentes para cada tamanho de vetor de 10.000 a 100.000 elementos, confirmaram empiricamente essa predição teórica.

Os principais resultados incluem:

Validação Teórica: A contagem de operações por fase resultou em O(n^2/k), confirmada empiricamente pelos dados experimentais através da constância da razão T(n)/(n^2/1000) ao longo de todos os tamanhos testados.

Diferenças de Desempenho: As três linguagens apresentam complexidade teórica idêntica, mas diferem significativamente no desempenho absoluto. C demonstrou ser 71 vezes mais rápido que Python e 1,15 vezes mais rápido que Java para n = 100.000 elementos.

Consistência dos Resultados: Os baixos desvios padrão observados (máximo de 6,57%) indicam comportamento previsível e controlado do algoritmo sob distribuição uniforme dos dados.

Implicações Práticas: A escolha da linguagem impacta significativamente o desempenho absoluto sem alterar a classe de complexidade, demonstrando a importância de considerar fatores além da análise assintótica em aplicações reais.

Referências

BENTLEY, J. L.; MCILROY, M. D. Engineering a sort function. Software: Practice and Experience, v. 23, n. 11, p. 1249-1265, 1993.

CORMEN, T. H. et al. Introduction to Algorithms. 3. ed. Cambridge: MIT Press, 2009.

KNUTH, D. E. The Art of Computer Programming: Volume 3: Sorting and Searching. 2. ed. Boston: Addison-Wesley, 1998.

PETERS, T. Timsort. Python Enhancement Proposals, 2002. Disponível em: https://github.com/python/cpython/blob/main/Objects/listsort.txt. Acesso em: 9 nov. 2025.

SEDGEWICK, R.; WAYNE, K. Algorithms. 4. ed. Upper Saddle River: Addison-Wesley Professional, 2011.