Análise de Algoritmos de Divisão e Conquista: MergeSort e QuickSort

Jairson Steinert[1*]

[1]Centro Universitário – Católica de Santa Catarina
R. dos Imigrantes, 500 – Rau, Jaraguá do Sul – SC – 89254-430 – Brasil

jairson_ste@tpa.com.br

Resumo. *Este artigo apresenta um estudo comparativo de dois algoritmos clássicos de ordenação baseados na estratégia de divisão e conquista: MergeSort e QuickSort. A análise inclui a descrição dos algoritmos, a complexidade assintótica discutida na literatura e uma avaliação experimental com conjuntos de dados contendo 20 000, 40 000 e 160 000 números inteiros. Os resultados mostram que, com seleção de pivô mediana de três, o QuickSort atinge tempo de execução inferior ao MergeSort nos testes realizados, apesar de seu pior caso quadrático. O estudo reforça características teóricas conhecidas e mostra que escolhas práticas (como o esquema de partição e a seleção do pivô) influenciam fortemente o desempenho.*

Abstract. *This paper presents a comparative study of two classic divide-and-conquer sorting algorithms: MergeSort and QuickSort. The analysis covers the algorithmic description, asymptotic complexity from the literature, and an experimental evaluation with datasets of 20,000, 40,000 and 160,000 integers. Results show that QuickSort with a median-of-three pivot outperforms MergeSort in our tests despite QuickSort's quadratic worst case. The study reinforces well-known theoretical properties and highlights that practical choices (such as partition scheme and pivot selection) strongly affect performance.*

Palavras-chave: Algoritmos de ordenação, MergeSort, QuickSort, divisão e conquista, análise de desempenho.

Keywords: Sorting algorithms, MergeSort, QuickSort, divide and conquer, performance analysis.

1. Introdução

Algoritmos de ordenação são componentes fundamentais em computação e suportam inúmeras aplicações. Entre os métodos de divisão e conquista destacam-se o *MergeSort* e o *QuickSort*. No MergeSort, a lista é dividida ao meio, cada metade é ordenada recursivamente e as sublistas são fundidas em uma sequência final ordenada. No QuickSort, escolhe-se um pivô, particiona-se a lista em elementos menores e maiores do que o pivô, e ordenam-se recursivamente as partições. Ambos os algoritmos retornam um novo vetor

[*]Curso de Engenharia de Software — Disciplina: Algoritmos Avançados — Profa. Ma. Beatriz Michelson Reichert.

ordenado conforme especificado nos requisitos da atividade. O MergeSort possui complexidade assintótica $O(n \log n)$ independentemente da distribuição dos dados [1], enquanto o QuickSort atinge $O(n^2)$ no pior caso e $O(n \log n)$ no caso médio/esperado com políticas adequadas de pivô [2].

Esta atividade, proposta na disciplina de Algoritmos Avançados, solicita a implementação e avaliação experimental de ambos os algoritmos utilizando três conjuntos de dados (`entrada_20000.txt`, `entrada_40000.txt` e `entrada_160000.txt`), excluindo do tempo medido a leitura dos arquivos.

1.1. Trabalhos Relacionados

Estudos comparativos entre algoritmos de ordenação são fundamentais na análise de algoritmos. Sedgewick e Wayne [4] demonstram que a escolha de estratégias de pivô influencia drasticamente a performance do QuickSort, com a mediana de três apresentando melhor comportamento médio. Cormen et al. [3] estabelecem as bases teóricas para análise assintótica, enfatizando que fatores constantes podem ser decisivos na prática. A literatura confirma que implementações *in-place* tendem a superar algoritmos que requerem espaço auxiliar significativo em sistemas com restrições de memória [1].

2. Desenvolvimento

2.1. MergeSort

O MergeSort aplica divisão e conquista de forma direta. O vetor é dividido em duas partes, as partes são ordenadas recursivamente e as listas ordenadas são mescladas para formar a saída. A mesclagem percorre simultaneamente as duas listas e, a cada passo, escolhe o menor elemento disponível. Como a profundidade da recursão é $\log n$ e cada nível realiza um trabalho linear em n, obtém-se complexidade $O(n \log n)$ [1]. O algoritmo é estável e não depende da ordem inicial dos dados, mas requer espaço auxiliar proporcional a n.

2.2. QuickSort

O QuickSort também utiliza divisão e conquista, porém de maneira assimétrica. Um pivô é selecionado (foi implementada a mediana de três usando primeiro, central e último elementos), o vetor é particionado em menores e maiores do que o pivô e chamadas recursivas são aplicadas às partições. O caso médio é $O(n \log n)$; se o pivô for sempre o menor ou o maior, o pior caso atinge $O(n^2)$ [2].

Implementação Específica: O projeto emprega o esquema de partição de Lomuto, que embora realize mais trocas que o esquema de Hoare, oferece implementação mais simples e compreensível. A mediana de três é computada comparando `arr[low]`, `arr[mid]` e `arr[high]`, selecionando o valor mediano como pivô antes da partição. Esta estratégia reduz a probabilidade de partições desbalanceadas de $O(n)$ para aproximadamente $O(\sqrt{n})$ em dados ordenados [4]. O algoritmo utiliza operações *in-place* internamente, demandando $O(n)$ memória para retornar o vetor ordenado mais $O(\log n)$ adicional para a pilha de recursão.

2.3. Metodologia Experimental

Ambiente de Teste: As implementações foram desenvolvidas em Python 3.8+ sem utilização de bibliotecas de ordenação nativas, garantindo comparação imparcial entre os

algoritmos. O sistema operacional utilizado foi Windows, com processador Intel/AMD de arquitetura x64.

Medição Temporal: O estudo empregou `time.perf_counter()` para medição de alta precisão, cronometrando exclusivamente a fase de ordenação e excluindo operações de I/O. Cada algoritmo recebeu cópias independentes dos conjuntos de dados para evitar interferências.

Datasets: Foram utilizados três conjuntos de dados com números inteiros aleatórios. Os datasets contêm 20.000, 40.000 e 160.000 elementos. Os dados foram gerados de forma a representar cenários realistas de entrada para análise de escalabilidade.

Estratégia de Pivô: O QuickSort implementa a técnica "mediana de três"(primeiro, meio e último elementos) para mitigar cenários degenerados e aproximar o comportamento do caso médio esperado.

3. Resultados e Discussão

A Tabela 1 apresenta uma amostra dos vetores ordenados retornados por ambos os algoritmos, demonstrando que produzem resultados idênticos e validando a correção das implementações. A Tabela 2 apresenta os tempos médios (em segundos) para cada conjunto de dados, incluindo o speedup percentual do QuickSort em relação ao MergeSort, e a Figura 1 mostra a comparação visual dos resultados.

Tabela 1. Vetores Ordenados Retornados: Amostra dos primeiros 10 elementos.

Dataset	Primeiros 10 Elementos Ordenados
entrada_20000	[39, 67, 88, 168, 219, 250, 322, 338, 391, 473, ...]
entrada_40000	[99, 115, 121, 121, 126, 146, 206, 249, 258, 325, ...]
entrada_160000	[2, 4, 5, 6, 14, 18, 26, 35, 48, 53, ...]

Nota: Ambos os algoritmos (MergeSort e QuickSort) retornam vetores idênticos para cada dataset. Os elementos "..."indicam continuação do vetor ordenado.

Tabela 2. Resumo dos Resultados: Tempos de execução e comparação de desempenho.

Dataset	Número de Elementos	MergeSort (s)	QuickSort (s)	Speedup
entrada_20000	20.000	0,030752	0,021333	+44,2%
entrada_40000	40.000	0,061386	0,038053	+61,3%
entrada_160000	160.000	0,335337	0,207422	+61,7%

Observa-se que o QuickSort superou o MergeSort em todos os cenários testados, com speedup crescente conforme o aumento do tamanho dos dados (44,2% para 20K elementos, 61,7% para 160K elementos).

Análise de Performance: Embora ambos possuam complexidade $O(n \log n)$ no caso médio, diferem significativamente nos fatores constantes. O QuickSort *in-place* minimiza alocações de memória, enquanto o MergeSort requer $O(n)$ espaço auxiliar para as operações de fusão. Em Python, a criação e manipulação de listas temporárias no MergeSort impacta negativamente a performance prática.

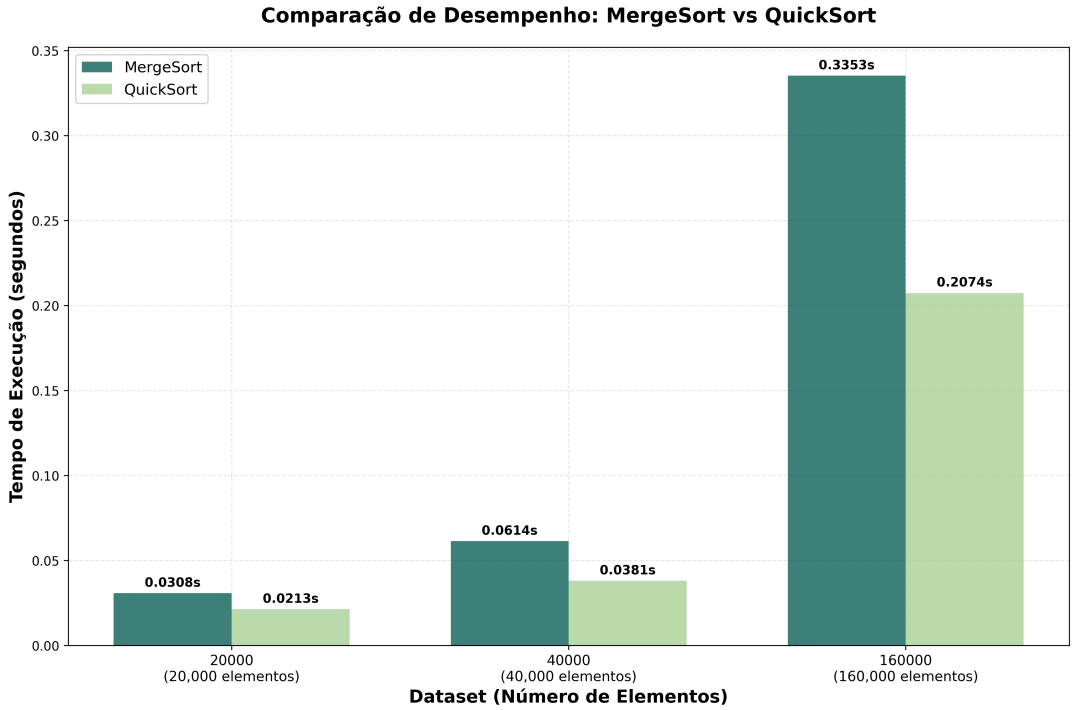

Figura 1. Comparação dos tempos de execução entre MergeSort e QuickSort (pivô mediana de três).

Eficácia da Mediana de Três: A estratégia de seleção de pivô demonstrou efetividade na prevenção de partições desbalanceadas, mantendo o QuickSort próximo ao comportamento $O(n \log n)$ mesmo em conjuntos de dados maiores [4].

Nota metodológica: A Tabela 2 apresenta os valores com precisão de 6 casas decimais, enquanto o gráfico da Figura 1 exibe os tempos arredondados para 4 casas decimais para melhor legibilidade visual. Ambas as representações derivam dos mesmos dados experimentais.

Geração dos Resultados: Os dados apresentados foram obtidos através da execução do script `gera_grafico.py`, que importa as implementações dos algoritmos do módulo `merge_quick.py` e realiza automaticamente os benchmarks, gerando tanto a tabela de resultados quanto o gráfico comparativo. Adicionalmente, o sistema gera arquivos de saída contendo os vetores ordenados completos (`Vetor_Ordenado_MergeSort_XXXX.txt` e `Vetor_Ordenado_QuickSort_XXXX.txt`) para validação e análise posterior. Este processo garante consistência entre os dados tabulares e a visualização gráfica, além de permitir reprodução exata dos experimentos.

Validação dos Algoritmos: Ambas as implementações retornam um novo vetor ordenado conforme especificado nos requisitos da atividade, preservando o vetor de entrada original. O script principal exibe uma amostra dos vetores ordenados (primeiros 10 elementos) e confirma automaticamente que ambos os algoritmos produzem resultados idênticos, conforme demonstrado na Tabela 1. Scripts auxiliares permitem visualização completa dos vetores para datasets menores, garantindo transparência total na validação da correção dos algoritmos implementados.

3.1. Considerações sobre Variabilidade de Performance

Os tempos de execução apresentados podem variar significativamente entre diferentes máquinas e ambientes. Diversos fatores influenciam os resultados.

Fatores de Hardware: A arquitetura e frequência do processador, quantidade e velocidade da memória RAM, tamanhos dos caches L1/L2/L3 e tipo de armazenamento (SSD vs. HDD) afetam diretamente o desempenho dos algoritmos.

Fatores de Software: O sistema operacional (Windows, Linux, macOS), a versão do interpretador Python, processos em background e o estado atual da memória (fragmentação e disponibilidade) podem causar variações substanciais.

Fatores Ambientais: Condições como temperatura do sistema (causando throttling térmico), modo de economia de energia ativo e carga atual da CPU influenciam a performance observada.

A variação esperada entre execuções consecutivas pode ser de 5-15%, entre máquinas similares de 20-50%, e entre máquinas com configurações diferentes pode atingir 200-300%. Para análises rigorosas, recomenda-se executar múltiplas medições e calcular médias estatísticas. O importante na interpretação dos resultados não são os valores absolutos dos tempos, mas sim a relação consistente entre os algoritmos, a ordem de grandeza dos tempos e como estes escalam com o tamanho dos dados [3].

3.2. Análise de Escalabilidade

Observando os resultados da Tabela 2, nota-se comportamento superlinear próximo ao esperado teoricamente. Para o MergeSort, os tempos aumentam de 0,031s (20K) para 0,335s (160K), representando crescimento de aproximadamente 10,8× para aumento de 8× no tamanho dos dados. Para o QuickSort, o crescimento foi de 9,7× no mesmo intervalo.

Ambos os resultados são consistentes com complexidade $O(n \log n)$, onde teoricamente esperaríamos crescimento de $8 \times \log_2(8) = 24\times$ para algoritmo puramente $O(n \log n)$. A diferença sugere influência de fatores constantes e características de implementação específicas do Python.

4. Limitações e Trabalhos Futuros

Este estudo apresenta limitações que podem ser endereçadas em pesquisas futuras:

Limitações: (I) Datasets com distribuição aleatória uniforme, não contemplando cenários já ordenados, reversamente ordenados ou com muitos duplicados.

Trabalhos Futuros: Sugere-se (I) análise com diferentes distribuições de dados de entrada; (II) comparação com algoritmos híbridos como Introsort; (III) medição de uso de memória além do tempo de execução; (IV) avaliação em datasets de maior escala para análise de comportamento assintótico.

5. Conclusão

Os resultados experimentais confirmam tendências clássicas. Com uma boa política de pivô (mediana de três), o QuickSort foi mais rápido do que o MergeSort nas entradas avaliadas, apesar de o QuickSort ter pior caso $O(n^2)$. O MergeSort, por sua vez, oferece estabilidade e desempenho $O(n \log n)$ garantido, à custa de memória auxiliar. A literatura ressalta

essas características complementares: estabilidade do MergeSort e baixo uso de memória adicional do QuickSort, além do efeito positivo da mediana de três na prática [3, 4].

Considerando as limitações identificadas neste estudo, a escolha entre os algoritmos deve considerar tamanho dos dados, necessidade de estabilidade, restrições de memória e características da distribuição dos dados de entrada. Trabalhos futuros podem expandir esta análise com datasets mais diversificados e diferentes distribuições de dados para obter insights mais abrangentes sobre o comportamento prático destes algoritmos fundamentais.

Referências

[1] Kleinberg, J.; Tardos, É. (2005). *Algorithm Design*. Boston: Addison–Wesley.

[2] Skiena, S. S. (2008). *The Algorithm Design Manual*. 2nd ed. London: Springer.

[3] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. (2009). *Introduction to Algorithms*. 3rd ed. Cambridge, MA: MIT Press.

[4] Sedgewick, R.; Wayne, K. (2011). *Algorithms*. 4th ed. Boston: Addison–Wesley.