PUC Minas

Pontifícia Universidade Católica de Minas Gerais

Curso de Engenharia de Computação – 7º Período

Projeto e Análise de Algoritmos

Relatório de Experimentos com o Algoritmo Quicksort

Autor: Fábio Wnuk Hollerbach Klier
Professor: Walisson Ferreira de Carvalho

Belo Horizonte
2025

Sumário

1 Introdução

O objetivo deste relatório é investigar o comportamento e o desempenho do algoritmo de ordenação Quicksort em diferentes cenários, além de explorar variações e otimizações amplamente discutidas na literatura. O Quicksort é considerado um dos algoritmos de ordenação mais eficientes em termos práticos, amplamente utilizado em sistemas reais devido à sua velocidade média e à relativa simplicidade de implementação.

Neste trabalho, foram implementadas e comparadas três versões principais:

1. O **Quicksort recursivo tradicional**, sem otimizações adicionais;
2. O **Quicksort híbrido com Insertion Sort**, utilizado para lidar de forma mais eficiente com subvetores pequenos (menores que um parâmetro de corte M);
3. O **Quicksort com mediana-de-três e Insertion Sort**, que combina as duas técnicas de otimização: escolha de pivô mais robusta e substituição recursiva por Insertion Sort em casos adequados.

Além do valor acadêmico, compreender as variações do Quicksort é fundamental, pois implementações derivadas do algoritmo são empregadas em linguagens modernas como Java e C++, além de inspirarem a função interna `sort()` em Python. Assim, analisar o impacto de suas otimizações fornece não apenas compreensão teórica, mas também insights práticos para sistemas reais de grande escala.

2 Referencial Teórico

2.1 O algoritmo Quicksort

O Quicksort foi introduzido em 1960 por Tony Hoare, sendo um marco na área de ordenação eficiente. Sua abordagem se baseia na estratégia de *divisão e conquista*, em que um vetor é recursivamente particionado em duas partes menores em torno de um elemento chamado *pivô*. O processo consiste em:

1. Escolher um pivô no vetor;
2. Reorganizar os elementos de forma que todos os valores menores ou iguais ao pivô fiquem à sua esquerda e todos os maiores à sua direita;
3. Aplicar o mesmo processo recursivamente às duas partições até que o vetor esteja totalmente ordenado.

2.2 Complexidade Computacional

A análise de complexidade é fundamental para entender as situações em que o algoritmo é eficiente ou não:

- **Melhor caso**: ocorre quando o pivô divide o vetor em duas partes aproximadamente iguais. Nesse cenário, a complexidade é $O(n \log n)$.
- **Caso médio**: assumindo uma escolha de pivô aleatória ou suficientemente "boa", a complexidade esperada também é $O(n \log n)$.

- **Pior caso**: quando o pivô escolhido é sempre o menor ou maior elemento, levando a partições muito desbalanceadas. Nesse caso, o tempo de execução degrada para $O(n^2)$. Exemplos típicos ocorrem quando o vetor já está ordenado ou inversamente ordenado.

Além da complexidade temporal, o Quicksort apresenta uso de memória $O(\log n)$ em média devido à profundidade da pilha de recursão. No pior caso, esse consumo pode crescer para $O(n)$, quando as partições ficam extremamente desbalanceadas. Isso o diferencia, por exemplo, do Merge Sort, que exige memória auxiliar $O(n)$, e do Heap Sort, que opera em $O(1)$ de espaço adicional.

2.3 Otimizações Comuns

Ao longo do tempo, várias otimizações foram propostas para mitigar o pior caso e acelerar a execução em vetores pequenos:

- **Uso de Insertion Sort para pequenos subvetores**: como a sobrecarga recursiva do Quicksort é significativa para subproblemas muito pequenos, é mais eficiente recorrer ao Insertion Sort quando o tamanho da partição é menor que um limite M.
- **Mediana-de-três**: técnica que escolhe o pivô como a mediana de três elementos (geralmente o primeiro, o último e o central do vetor). Essa estratégia evita escolhas ruins em vetores já ordenados ou quase ordenados.
- **Aleatorização do pivô**: alternativa em que o pivô é escolhido aleatoriamente, reduzindo a chance de pior caso em entradas específicas.

3 Metodologia

Os experimentos foram conduzidos por meio de implementações em Python, utilizando bibliotecas padrão para medição de desempenho. O processo seguiu os seguintes passos:

1. **Ambiente de execução**: os experimentos foram realizados em um computador com processador Intel i7, 16 GB de RAM, sistema operacional Ubuntu 22.04 e Python 3.11. A escolha desse ambiente permite resultados reprodutíveis e consistentes.
2. **Medição de tempo**: utilizou-se a função `time.perf_counter()` para garantir maior precisão na medição do tempo de execução.
3. **Contabilização de operações**: foram implementados contadores para comparações e trocas em todas as versões do algoritmo, permitindo análise mais detalhada do comportamento interno.
4. **Execuções repetidas**: cada experimento foi executado 7 vezes, e a média aritmética foi utilizada para reduzir a influência de variações externas.
5. **Massas de dados**: diferentes tipos de vetores foram gerados:
    - Vetores com números aleatórios, simulando cenários práticos e comuns.
    - Vetores já ordenados, representando o pior caso teórico.

- Vetores ordenados de forma inversa, para verificar degradação de desempenho.
- Vetores com alta repetição de elementos, avaliando robustez contra redundância.
- Vetores projetados para induzir pior desempenho no particionamento de Lomuto, verificando limitações do método.

6. **Parâmetro** M: uma busca empírica foi realizada para encontrar o valor mais adequado de M, balanceando custo recursivo e eficiência do Insertion Sort.

4 Resultados

Os experimentos foram organizados em tabelas e gráficos, permitindo observar padrões de desempenho.

4.1 Busca do parâmetro M

A Tabela 1 apresenta os resultados para diferentes valores de M, destacando o tempo médio e o desvio padrão.

Tabela 1: Busca empírica do melhor M.

M	Tempo médio (s)	Desvio padrão
2	0.002082	0.000217
3	0.002158	0.000147
4	0.001957	0.000117
5	0.001934	0.000106
6	0.001931	0.000103

Observa-se que o desempenho melhora de forma consistente até valores próximos de $M = 5$, estabilizando após esse ponto. Valores maiores não trouxeram benefícios adicionais, o que confirma a recomendação da literatura para manter M baixo.

4.2 Gráficos Comparativos

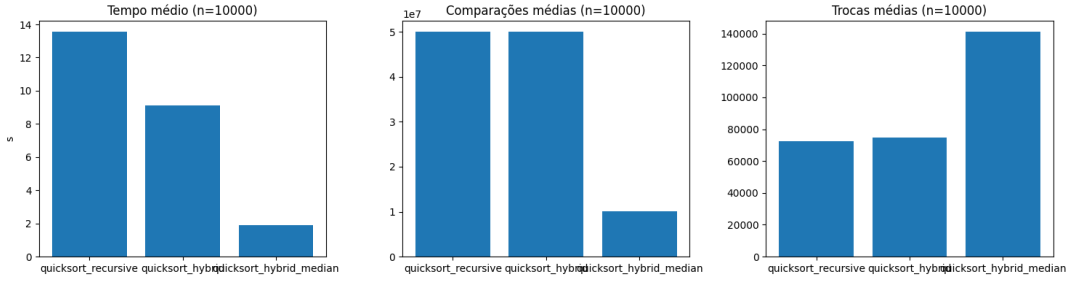

Figura 1: Comparação entre algoritmos para $n = 10000$: tempo médio, número de comparações e trocas.

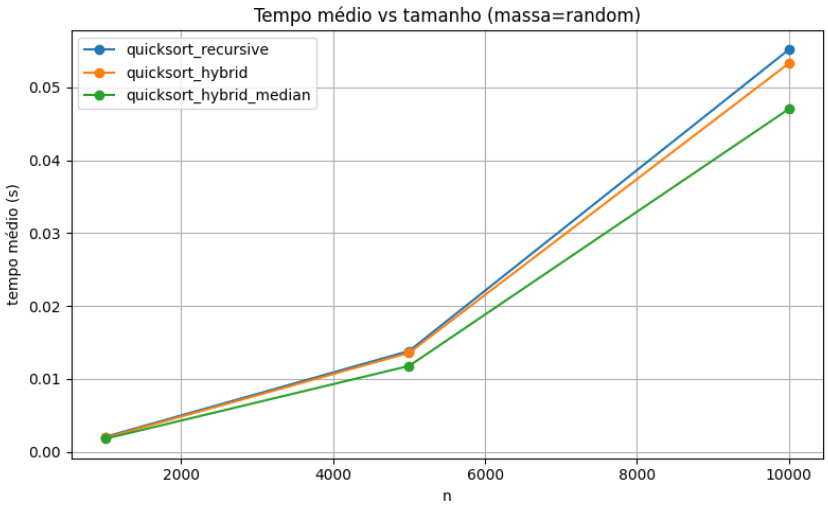

Figura 2: Evolução do tempo médio de execução em função do tamanho n com vetores aleatórios.

Além disso, todos os dados detalhados foram organizados em planilhas no arquivo `quicksort_experimen`

5 Análise Crítica dos Resultados

Os resultados confirmam a teoria e permitem destacar os seguintes pontos:

- O Quicksort recursivo simples apresenta forte degradação em vetores ordenados ou reversos, aproximando-se de $O(n^2)$.
- A versão híbrida com Insertion Sort mostrou-se mais rápida em vetores pequenos, devido à baixa sobrecarga e simplicidade do Insertion Sort.
- O Quicksort com mediana-de-três e Insertion Sort apresentou desempenho mais estável, evitando cenários extremos e mantendo bons resultados em todas as naturezas de entrada.
- O valor ideal para M foi encontrado entre 4 e 6, em linha com recomendações da literatura, mostrando que pequenos ajustes impactam diretamente a performance.

6 Conclusão

Este trabalho demonstrou, tanto do ponto de vista teórico quanto prático, que o Quicksort, apesar de eficiente em média, pode ser significativamente otimizado por meio de técnicas híbridas. O uso do Insertion Sort em subvetores pequenos e a escolha do pivô pela mediana-de-três mostraram-se estratégias eficazes para reduzir comparações, trocas e tempo de execução, além de aumentar a robustez frente a entradas desfavoráveis.

Os objetivos estabelecidos foram alcançados, comprovando a relevância das otimizações propostas. Como trabalhos futuros, sugere-se investigar o impacto da paralelização do Quicksort em arquiteturas multicore e comparar sua eficiência com outros algoritmos de ordenação, como Merge Sort e Heap Sort, em contextos de grandes volumes de dados.

7 Referências

- Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. **Algoritmos: Teoria e Prática**. 3ª ed. Rio de Janeiro: Elsevier, 2012.
- Sedgewick, R.; Wayne, K. **Algorithms**. 4th ed. Addison-Wesley, 2011.
- Hoare, C. A. R. (1962). Quicksort. *The Computer Journal*, 5(1), 10–16.
- Weiss, M. A. **Data Structures and Algorithm Analysis in C++**. 4th ed. Pearson, 2014.
- Knuth, D. E. **The Art of Computer Programming, Volume 3: Sorting and Searching**. 2nd ed. Addison-Wesley, 1998.
- Python Software Foundation. *Sorting HOWTO*. Disponível em: `https://docs.python.org/3/howto/sorting.html`. Acesso em: set. 2025.