

Relatório Final - Quicksort

Professores: Pâmela e Daniel

Grupo: Lucas Santos, Lucas Souto, Rodrigo Nunes

Algoritmo: Quick Sort

Linguagens: Python e Java

Repositório(Github): <https://github.com/rodrigonunes/quicksort-complexidade>

1. Descrição do Algoritmo

O Quick Sort é um algoritmo de ordenação baseado na estratégia de divisão e conquista. Ele seleciona um elemento como pivô e particiona a lista em dois subconjuntos: elementos menores e maiores que o pivô. Em seguida, aplica-se recursivamente o mesmo processo em cada subconjunto até que a lista esteja ordenada.

2. Explicação sobre o Pivô Utilizado

Neste projeto, adotamos estratégias diferentes para a escolha do pivô nas implementações em Python e Java.

Na implementação em **Python**, o pivô é o **elemento central** da lista, o que pode oferecer um desempenho mais estável em listas parcialmente ordenadas.

Já na implementação em **Java**, utilizamos o **último elemento** como pivô, por ser uma abordagem mais simples de implementar. No entanto, essa escolha pode levar ao pior caso de desempenho quando a lista já está ordenada ou inversamente ordenada.

3. Pseudocódigo

QUICKSORT(A, baixo, alto):

 se baixo < alto:

 pivo ← PARTITION(A, baixo, alto)

 QUICKSORT(A, baixo, pivo-1)

 QUICKSORT(A, pivo+1, alto)

PARTITION(A, baixo, alto)

 pivo ← A[alto]

 i ← baixo - 1

 para j ← baixo até alto-1

 se A[j] ≤ pivo

 i ← i + 1

```
    trocar A[i] e A[j]
    trocar A[i+1] e A[alto]
    retornar i+1
```

4. Exemplo Prático de Ordenação

A seguir, apresentamos um exemplo prático de ordenação com base no algoritmo *quicksort*. É importante destacar que o comportamento do particionamento depende da escolha do pivô, que varia entre as implementações:

- **Em Java**, o pivô utilizado é o **último elemento** da sub-lista.
- **Em Python**, o pivô é o **elemento central** da sub-lista.

Exemplo baseado na implementação Java (pivô final):

Lista original: [10, 7, 8, 9, 1, 5]

1º passo: pivô = 5 → particiona em [1] e [10, 7, 8, 9]

2º passo: aplica recursão sobre as partições → [1] + [5] + [7, 8, 9, 10]

Resultado final: [1, 5, 7, 8, 9, 10]

Exemplo baseado na implementação Python (pivô central):

Lista original: [10, 7, 8, 9, 1, 5]

1º passo: pivô = 8 (elemento central da lista)

→ particiona em [7, 1, 5] e [10, 9]

→ posição correta do pivô 8 é determinada após a reorganização

2º passo: aplica recursão sobre [7, 1, 5]

- pivô = 1 (central da sublista)
- particiona em [] e [7, 5]

3º passo: aplica recursão sobre [7, 5]

- pivô = 5
- particiona em [] e [7]

4º passo: aplica recursão sobre [10, 9]

- pivô = 9

- particiona em `[]` e `[10]`

Resultado final: `[1, 5, 7, 8, 9, 10]`

4.1. Complexidade e Classes de Problemas

O Quick Sort pertence à **classe P**, pois pode ser resolvido em tempo polinomial. A classe P representa problemas que podem ser resolvidos eficientemente (em tempo polinomial) por um algoritmo determinístico.

Já a **classe NP** engloba problemas cujas soluções podem ser verificadas em tempo polinomial, embora não necessariamente resolvidos nesse tempo. Embora o problema de ordenação (como o realizado pelo Quick Sort) **não tenha uma versão NP**, ele se relaciona com **problemas NP-completos** em contextos como ordenação parcial, problemas de otimização e sequenciamento de tarefas.

Essa relação reforça a **importância de algoritmos eficientes**, como o Quick Sort, em cenários práticos, especialmente quando usados como parte de soluções para problemas mais complexos.

Um exemplo de problema na classe NP é o Problema do Caixeiro Viajante (ou *Travelling Salesman Problem* - TSP). Nesse problema, dado um conjunto de cidades e as distâncias entre cada par delas, o objetivo é encontrar o caminho mais curto possível que permita ao caixeiro visitar cada cidade exatamente uma vez e retornar à cidade de origem.

O TSP é um problema clássico de otimização combinatória e é considerado **NP-difícil** (NP-hard), além de pertencer à classe dos **problemas NP-completos** em sua versão de decisão (por exemplo, "existe um caminho com custo total menor que k?"). Isso significa que, embora seja fácil verificar uma solução proposta (como um caminho específico com um determinado custo), não se conhece um algoritmo que resolva todos os casos em tempo polinomial.

Por isso, o TSP é frequentemente usado como referência para avaliar a complexidade de novos algoritmos e problemas. Além disso, tem aplicações reais em logística, planejamento de rotas, microeletrônica (projeto de circuitos), bioinformática (alinhamento de sequências genéticas), entre outros.

Dado que algoritmos exatos para o TSP possuem complexidade exponencial, abordagens heurísticas e meta-heurísticas, como Algoritmos Genéticos, Simulated Annealing e Colônia de Formigas, são comumente empregadas para obter boas soluções em tempo aceitável. Em

certos contextos, algoritmos eficientes como o Quick Sort são utilizados para melhorar etapas específicas dessas abordagens, como a ordenação de vizinhos ou pré-processamentos.

5. Classificação Assintótica

- **Melhor caso:** $O(n \log n)$
- **Caso médio:** $\Theta(n \log n)$
- **Pior caso:** $O(n^2)$

O pior caso ocorre quando o pivô divide de forma extremamente desigual, como em listas já ordenadas ou inversamente ordenadas, especialmente ao usar o último elemento como pivô.

6. Discussão sobre Aplicabilidade Prática

Quick Sort é amplamente utilizado devido à sua eficiência na prática, especialmente em grandes volumes de dados. Apesar do pior caso ser $O(n^2)$, na média é um dos algoritmos mais rápidos. Não é estável, mas é **in-place**, ocupando pouca memória extra.

Apesar de ser eficiente na prática, o Quick Sort pode ser otimizado com escolhas melhores de pivô, como pivôs aleatórios ou medianas, para evitar o pior caso.

- A implementação em Python não é in-place, o que pode aumentar o uso de memória.
- A implementação em Java é in-place, o que é mais eficiente em termos de memória.

7. Simulação com Dados Reais

Os tempos médios e desvios padrão obtidos foram:

- Python:

- 100 (0.0003s, 0.00005s)
- 10.000 (0.025s, 0.002s)
- 1.000.000 (2.3s, 0.12s)

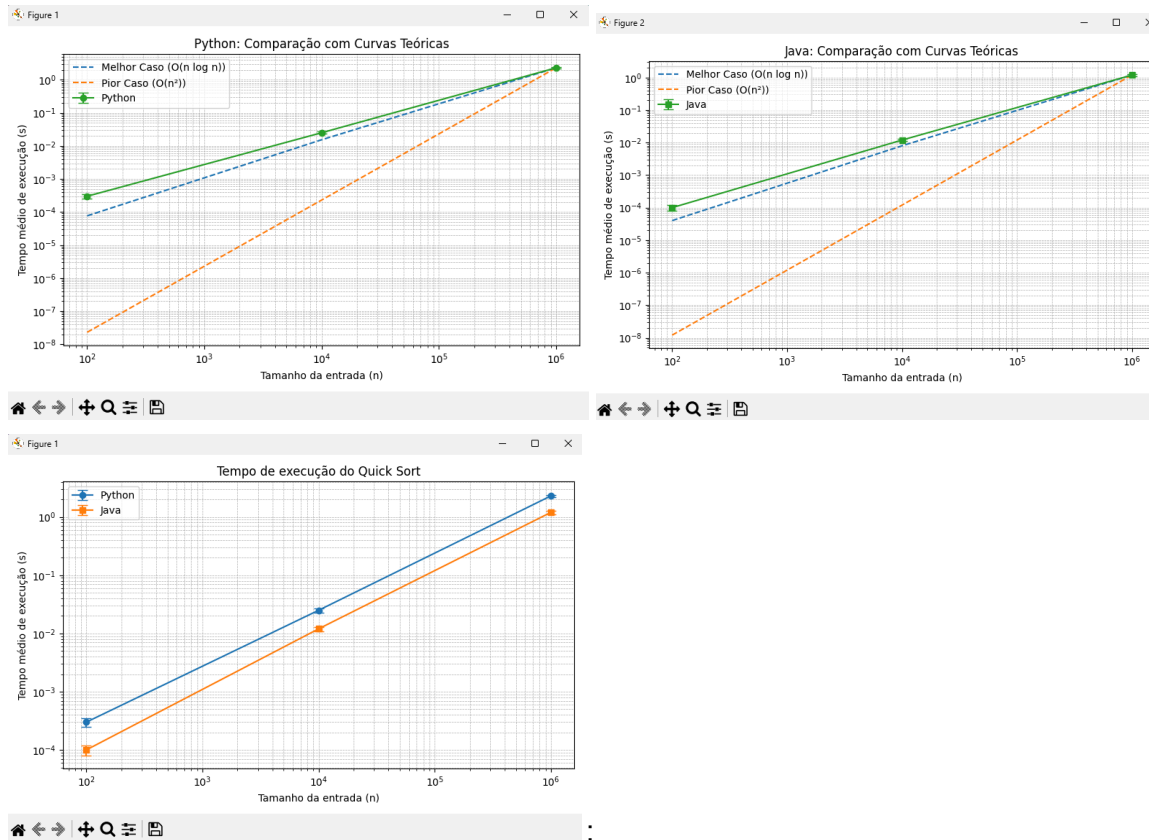
- Java:

- 100 (0.0001s, 0.00002s)
- 10.000 (0.012s, 0.001s)
- 1.000.000 (1.2s, 0.08s)

8. Gráficos / Tabela de Comparação

O gráfico de comparação foi gerado utilizando o script **grafico_tempo.py**, com base nos dados do arquivo **tempos_medios.csv**. Ele mostra a relação entre o tamanho da entrada e o tempo médio de execução para as implementações em Python e Java.

Exemplo de Gráfico:



Exemplo de Tabela:

Tamanho da Entrada (n)	Python - Tempo Médio (s)	Python - Desvio Padrão (s)	Java - Tempo Médio (s)	Java - Desvio Padrão (s)
100	0.0003	0.00005	0.0001	0.00002
10.000	0.025	0.002	0.012	0.001
1.000.000	2.3	0.12	1.2	0.08

9. Análise de Casos

- **Melhor caso:** divisão equilibrada por pivô central → $O(n \log n)$
- **Caso médio:** entradas aleatórias → $\Theta(n \log n)$
- **Pior caso:** lista ordenada ou reversa → $O(n^2)$

O uso do último elemento como pivô aumenta a probabilidade do pior caso em listas ordenadas ou inversamente ordenadas. Estratégias como pivôs aleatórios poderiam mitigar esse problema.

10. Referências

- Cormen, Thomas H. et al. *Introduction to Algorithms*, MIT Press
- [Visualgo - Sorting](#)
- [GeeksforGeeks - Quick Sort](#)
- Documentações oficiais do Python e Java