Universidade Federal do Tocantins

Análise Empírica de Algoritmos de Ordenação

Raphael Sales de Souza, Thais Carvalho de Freitas e Sophia Ribeiro Prado

1 Introdução

A ordenação de dados é uma operação fundamental em Ciência da Computação, crucial para a otimização de diversos processos computacionais. Ela facilita a busca eficiente, a fusão de conjuntos de dados e a realização de análises complexas, servindo como base para algoritmos mais avançados. Knuth (1998) destaca que a ordenação é um dos problemas mais estudados na área de algoritmos, dado seu impacto direto na eficiência de sistemas de informação.

O objetivo deste trabalho é realizar uma análise empírica de diferentes algoritmos de ordenação, abordando tanto aspectos teóricos quanto práticos. A investigação inclui os algoritmos Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort, onde será feita uma comparação detalhada entre esses algoritmos, considerando critérios como complexidade temporal e espacial, estabilidade, facilidade de implementação e desempenho em cenários variados. A análise empírica será conduzida por meio de experimentos práticos que demonstram o comportamento dos algoritmos com diferentes tamanhos e distribuições de dados, permitindo uma avaliação abrangente de suas eficiências relativas.

Além de sua relevância teórica, a ordenação de dados possui um impacto prático significativo em diversas aplicações reais. Em bancos de dados, por exemplo, a ordenação é essencial para a realização de consultas eficientes e para a manutenção da integridade e rapidez dos sistemas. No processamento de grandes volumes de dados, como em análises de big data, a ordenação eficiente pode significar a diferença entre um processamento viável e um impraticável. Cormen et al. (2009) argumentam que a escolha do algoritmo de ordenação apropriado pode reduzir drasticamente o tempo de processamento, tornando as operações computacionais mais rápidas e econômicas.

Através de implementações reais dos algoritmos Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort, serão examinados aspectos como tempo de execução e uso de memória em diferentes cenários. Serão considerados tanto conjuntos de dados ordenados aleatoriamente quanto aqueles com características específicas, como ordenação prévia ou presença de duplicatas. Esta abordagem permite uma avaliação holística, combinando a análise de com-

plexidade teórica com observações práticas, para oferecer uma visão completa sobre as vantagens e limitações de cada algoritmo.

2 Revisão teórica

Para compreender plenamente as nuances e eficiências dos diferentes algoritmos de ordenação, é essencial examinar suas características teóricas e práticas.

O algoritmo Bubble Sort é um algoritmo de ordenação simples, caracterizado pela repetida troca de elementos adjacentes que estão fora de ordem. Possui complexidade temporal $O(n^2)$, tanto no pior quanto no melhor caso, tornando-o ineficiente para grandes conjuntos de dados (Cormen et al., 2009). É considerado um algoritmo estável, pois não altera a ordem relativa de elementos iguais durante a ordenação. É um algoritmo in-place, o que significa que não requer memória adicional significativa além da necessária para a lista original.

O algoritmo Selection Sort funciona selecionando repetidamente o menor (ou maior) elemento da lista não ordenada e movendo-o para a posição correta. Sua complexidade temporal é $O(n^2)$ em todos os casos, devido à necessidade de buscar o elemento mínimo em cada iteração. Assim como o Bubble Sort, é um algoritmo in-place, entretanto, o Selection Sort não é estável.

O Insertion Sort constrói a lista ordenada de forma incremental, inserindo cada novo elemento na posição correta. Possui complexidade temporal $O(n^2)$ no pior caso, mas $O(n)$ no melhor caso, quando a lista já está ordenada ou quase ordenada. É um algoritmo in-place e estável, pois a inserção de elementos não altera a ordem relativa dos elementos iguais.

Merge Sort é um algoritmo de ordenação baseado na técnica de divisão e conquista. Divide a lista em sublistas menores, ordena-as recursivamente e, em seguida, mescla as sublistas ordenadas. Sua complexidade temporal é $O(n \log n)$ em todos os casos, tornando-o mais eficiente para grandes conjuntos de dados. É estável, preservando a ordem relativa de elementos iguais, mas não é in-place, pois requer memória adicional proporcional à lista original para armazenar as sublistas temporárias.

Quick Sort também utiliza a técnica de divisão e conquista, escolhendo um pivô e particionando a lista em sublistas de elementos menores e maiores que o pivô. Sua complexidade temporal é $O(n \log n)$ no caso médio, mas $O(n^2)$ no pior caso, que pode ser mitigado com a escolha adequada do pivô. É um algoritmo in-place, não requerendo memória adicional significativa, e pode ser instável, dependendo da implementação, pois a troca de elementos pode alterar a ordem relativa de elementos iguais.

Por último, o algoritmo Heap Sort transforma a lista em uma estrutura de heap (max-heap ou min-heap) e então extrai o elemento máximo, ou mínimo, repetidamente para construir a lista ordenada. Apresenta complexidade temporal $O(n \log n)$ em todos os casos. É um algoritmo in-place, utilizando apenas uma quantidade constante de espaço adicional além da lista original, porém, não é estável, pois a operação de "heapificação" pode alterar a ordem relativa de elementos iguais.

3 Metodologia

Os testes foram realizados numa máquina com as seguintes especificações técnicas:

- Hardware: AMD RYZEN 7 5700G Cache 20MB, 8 Núcleos, 16 Threads Ram Kingston 24 Gb 3600 Mhz SSD Kingston NV2 1TB NVMe M.2 2280 (Leitura até 3500MB/s e Gravação até 2100MB/s).
- Sistema operacional: Windows 11 23H2.
- Software: Visual Studio Code, Python 3.12.

A metodologia adotada inclui a instrumentação dos algoritmos para medir o tempo de execução e o número de comparações e trocas realizadas, permitindo uma comparação detalhada. Conjuntos de dados variados, incluindo sequências aleatórias, ordenadas e inversamente ordenadas, são utilizados para testar o comportamento dos algoritmos em diferentes condições. Esta abordagem prática permite uma avaliação abrangente dos algoritmos através das ferramentas desenvolvidas pelos alunos. Para medição do tempo dos algoritmos, foi utilizada a biblioteca do Python "import time". Todos os algoritmos elaborados pelos alunos estão num bloco principal, que é sucedido pelo bloco de cálculo de tempo.

4 Resultados

4.1 Bubble Sort

Primeiro Input
Input.txt
Tempo de execução: 0.046229 segundos Número de comparações: 499035 Número de trocas: 251488
Segundo array: randomly_shuffled_array_1000
Tempo de execução: 0.046049 segundos Número de comparações: 499035 Número de trocas: 251488
Terceiro array: randomly_shuffled_array_5000
Tempo de execução: 1.173299 segundos Número de comparações: 12496510 Número de trocas: 6226258
Quarto Array: randomly_shuffled_array_10000
Tempo de execução: 4.871483 segundos Número de comparações: 49977980 Número de trocas: 25190187
Quinto Array: randomly_shuffled_array_25000
Tempo de execução: 30.601796 segundos Número de comparações: 312475565 Número de trocas: 155982964
Sexto Array: randomly_shuffled_array_50000
empo de execução: 122.870896 segundos Número de comparações: 1249902990 Número de trocas: 623011403
Setimo Array: randomly_shuffled_array_75000

Tempo de execução: 294.347848 segundos Número de comparações: 2812387034 Número de trocas: 1408752501

Oitavo Array: randomly_shuffled_array_100000

Tempo de execução: 504.871291 segundos Número de comparações: 4999791797 Número de trocas: 2502357267

Nono Array: reverse_sorted_array_1000

Tempo de execução: 0.061565 segundos Número de comparações: 499500 Número de trocas: 499500

Decimo array: reverse_sorted_array_5000

Tempo de execução: 1.601466 segundos Número de comparações: 12497500 Número de trocas: 12497500

Decimo primeiro Array: reverse_sorted_array_10000

Tempo de execução: 6.554653 segundos Número de comparações: 49995000 Número de trocas: 49995000

Decimo segundo array: reverse_sorted_array_25000

Tempo de execução: 39.401005 segundos Número de comparações: 312487500 Número de trocas: 312487500

Decimo terceiro array: reverse_sorted_array_50000

Tempo de execução: 318.203249 segundos Número de comparações: 1249975000 Número de trocas: 1249975000

Decimo quarto array: reverse_sorted_array_75000

Tempo de execução: 362.287393 segundos Número de comparações: 2812462500 Número de trocas: 2812462500

Decimo quinto array: reverse_sorted_array_100000

Tempo de execução: 643.738521 segundos Número de comparações: 4999950000 Número de trocas: 4999950000

Decimo sexto Array: sorted_array_1000

Tempo de execução: 0.001002 segundos Número de comparações: 999 Número de trocas: 0

Decimo setimo array: sorted_array_5000

Tempo de execução: 0.004001 segundos Número de comparações: 4999 Número de trocas: 0

Decimo oitavo array: sorted_array_10000

Tempo de execução: 0.007002 segundos Número de comparações: 9999 Número de trocas: 0

Decimo nono array: sorted_array_25000

Tempo de execução: 0.01652 segundos Número de comparações: 24999 Número de trocas: 0

Vigesimo array: sorted_array_50000

Tempo de execução: 0.033874 segundos Número de comparações: 49999 Número de trocas: 0

Vigesimo primeiro array: sorted_array_75000

Tempo de execução: 0.050231 segundos Número de comparações: 74999 Número de trocas: 0

Vigesimo segundo array: sorted_array_100000

Tempo de execução: 0.066738 segundos Número de comparações: 99999 Número de trocas: 0

4.2 Heap Sort

Primeiro Input

Input.txt Tempo de execução: 0.003006 segundos Número de comparações: 16876 Número de trocas: 9114

Segundo array: randomly_shuffled_array_1000

Tempo de execução: 0.00403 segundos Número de comparações: 16876 Número de trocas: 9114

Terceiro array: randomly_shuffled_array_5000

Tempo de execução: 0.01871 segundos Número de comparações: 107678 Número de trocas: 57144

Quarto Array: randomly_shuffled_array_10000 Tempo de execução: 0.038686 segundos Número de comparações: 235346 Número de trocas: 124199

Quinto Array: randomly_shuffled_array_25000

Tempo de execução: 0.109452 segundos Número de comparações: 654868 Número de trocas: 343876

Sexto Array: randomly_shuffled_array_50000

Tempo de execução: 0.233054 segundos Número de comparações: 1409958 Número de trocas: 737787

Setimo Array: randomly_shuffled_array_75000

Tempo de execução: 0.388952 segundos Número de comparações: 2200294 Número de trocas: 1148931

Oitavo Array: randomly_shuffled_array_100000

Tempo de execução: 0.51347 segundos Número de comparações: 3019448 Número de trocas: 1574757

Nono Array: reverse_sorted_array_1000

Tempo de execução: 0.003 segundos Número de comparações: 15965 Número de trocas: 8316

Decimo array: reverse_sorted_array_5000

Tempo de execução: 0.016209 segundos Número de comparações: 103227 Número de trocas: 53436

Decimo primeiro Array: reverse_sorted_array_10000

Tempo de execução: 0.036985 segundos Número de comparações: 226682 Número de trocas: 116696

Decimo segundo array: reverse_sorted_array_25000

Tempo de execução: 0.100085 segundos Número de comparações: 633719 Número de trocas: 326586

Decimo terceiro array: reverse_sorted_array_50000

Tempo de execução: 0.212812 segundos Número de comparações: 1366047 Número de trocas: 698892

Decimo quarto array: reverse_sorted_array_75000

Tempo de execução: 0.365465 segundos Número de comparações: 2138650 Número de trocas: 1095164

Decimo quinto array: reverse_sorted_array_100000

Tempo de execução: 0.453005 segundos Número de comparações: 2926640 Número de trocas: 1497434

Decimo sexto Array: sorted_array_1000

Tempo de execução: 0.003508 segundos Número de comparações: 17583 Número de trocas: 9708

Decimo setimo array: sorted_array_5000

Tempo de execução: 0.019552 segundos Número de comparações: 112126 Número de trocas: 60932

Decimo oitavo array: sorted_array_10000

Tempo de execução: 0.039175 segundos Número de comparações: 244460 Número de trocas: 131956

Decimo nono array: sorted$_a rray_2$5000

Tempo de execução: 0.108692 segundos Número de comparações: 677688 Número de trocas: 361454

Vigesimo array: sorted$_a rray_5$0000

Tempo de execução: 0.227708 segundos Número de comparações: 1455438 Número de trocas: 773304

Vigesimo primeiro array: sorted$_a rray_7$5000

Tempo de execução: 0.366183 segundos Número de comparações: 2268087 Número de trocas: 1203722

Vigesimo segundo array: sorted$_a rray_1$00000

Tempo de execução: 0.496127 segundos Número de comparações: 3112517 Número de trocas: 1650854

4.3 Insertion Sort

Primeiro Input

Input.txt

Tempo de execução: 0.023127 segundos Número de comparações: 252480 Número de trocas: 251488

Segundo array: randomly$_s huffled_a rray_1$000

Tempo de execução: 0.01928 segundos Número de comparações: 252480 Número de trocas: 251488

Terceiro array: randomly$_s huffled_a rray_5$000

Tempo de execução: 0.573234 segundos Número de comparações: 6231251 Número de trocas: 6226258

Quarto Array: randomly$_s huffled_a rray_1$0000

Tempo de execução: 2.327012 segundos Número de comparações: 25200177 Número de trocas: 25190187

Quinto Array: randomly$_s huffled_a rray_2$5000

Tempo de execução: 14.513901 segundos Número de comparações: 156007953 Número de trocas: 155982964

Sexto Array: randomly$_s huffled_a rray_5$0000

Tempo de execução: 62.837826 segundos Número de comparações: 623061390 Número de trocas: 623011403

Setimo Array: randomly$_s huffled_a rray_7$5000
Tempo de execução: 143.918671 segundos Número de comparações: 1408827493 Número de trocas: 1408752501
Oitavo Array: randomly$_s huffled_a rray_1$00000
Tempo de execução: 271.0181 segundos Número de comparações: 2502457259 Número de trocas: 2502357267
Nono Array: reverse$_s orted_a rray_1$000
Tempo de execução: 0.052575 segundos Número de comparações: 499500 Número de trocas: 499500
Decimo array: reverse$_s orted_a rray_5$000
Tempo de execução: 1.175837 segundos Número de comparações: 12497500 Número de trocas: 12497500
Decimo primeiro Array: reverse$_s orted_a rray_1$0000
Tempo de execução: 4.597773 segundos Número de comparações: 49995000 Número de trocas: 49995000
Decimo segundo array: reverse$_s orted_a rray_2$5000
Tempo de execução: 29.271974 segundos Número de comparações: 312487500 Número de trocas: 312487500
Decimo terceiro array: reverse$_s orted_a rray_5$0000
Tempo de execução: 113.519543 segundos Número de comparações: 1249975000 Número de trocas: 1249975000
Decimo quarto array: reverse$_s orted_a rray_7$5000
Tempo de execução: 272.950963 segundos Número de comparações: 2812462500 Número de trocas: 2812462500
Decimo quinto array: reverse$_s orted_a rray_1$00000
Tempo de execução: 479.113865 segundos Número de comparações: 4999950000 Número de trocas: 4999950000
Decimo sexto Array: sorted$_a rray_1$000
Tempo de execução: 0.001002 segundos Número de comparações: 999 Número de trocas: 0
Decimo setimo array: sorted$_a rray_5$000
Tempo de execução: 0.004001 segundos Número de comparações: 4999 Número de trocas: 0
Decimo oitavo array: sorted$_a rray_1$0000
Tempo de execução: 0.008003 segundos Número de comparações: 9999 Número de trocas: 0
Decimo nono array: sorted$_a rray_2$5000
Tempo de execução: 0.013003 segundos Número de comparações: 24999 Número de trocas: 0
Vigesimo array: sorted$_a rray_5$0000
Tempo de execução: 0.035786 segundos Número de comparações: 49999 Número de trocas: 0
Vigesimo primeiro array: sorted$_a rray_7$5000
Tempo de execução: 0.053027 segundos Número de comparações: 74999 Número de trocas: 0
Vigesimo segundo array: sorted$_a rray_1$00000

Tempo de execução: 0.074097 segundos Número de comparações: 99999 Número de trocas: 0

4.4 Merge Sort

Primeiro Input

Input.txt

Tempo de execução: 0.004007 segundos Número de comparações: 8721 Número de atribuições: 9976

Segundo array: randomly$_s huffled_a rray_1$000

Tempo de execução: 0.003006 segundos Número de comparações: 8721 Número de atribuições: 9976

Terceiro array: randomly$_s huffled_a rray_5$000

Tempo de execução: 0.016948 segundos Número de comparações: 55241 Número de atribuições: 61808

Quarto Array: randomly$_s huffled_a rray_1$0000

Tempo de execução: 0.033596 segundos Número de comparações: 120448 Número de atribuições: 133616

Quinto Array: randomly$_s huffled_a rray_2$5000

Tempo de execução: 0.093672 segundos Número de comparações: 334033 Número de atribuições: 367232

Sexto Array: randomly$_s huffled_a rray_5$0000

Tempo de execução: 0.191561 segundos Número de comparações: 718086 Número de atribuições: 784464

Setimo Array: randomly$_s huffled_a rray_7$5000

Tempo de execução: 0.320766 segundos Número de comparações: 1121120 Número de atribuições: 1218928

Oitavo Array: randomly$_s huffled_a rray_1$00000

Tempo de execução: 0.437936 segundos Número de comparações: 1536403 Número de atribuições: 1668928

Nono Array: reverse$_s orted_a rray_1$000

Tempo de execução: 0.003 segundos Número de comparações: 4932 Número de atribuições: 9976

Decimo array: reverse$_s orted_a rray_5$000

Tempo de execução: 0.014045 segundos Número de comparações: 29804 Número de atribuições: 61808

Decimo primeiro Array: reverse$_s orted_a rray_1$0000

Tempo de execução: 0.030649 segundos Número de comparações: 64608 Número de atribuições: 133616

Decimo segundo array: reverse$_s orted_a rray_2$5000

Tempo de execução: 0.079315 segundos Número de comparações: 178756 Número de atribuições: 367232

Decimo terceiro array: reverse$_s orted_a rray_5$0000

Tempo de execução: 0.170266 segundos Número de comparações: 382512 Número de atribuições: 784464

Decimo quarto array: reverse$_s orted_a rray_7$5000

Tempo de execução: 0.263662 segundos Número de comparações: 594612 Número de atribuições: 1218928

Decimo quinto array: reverse$_s orted_a rray_1$00000

Tempo de execução: 0.400006 segundos Número de comparações: 815024 Número de atribuições: 1668928

Decimo sexto Array: sorted$_a rray_1$000

Tempo de execução: 0.003001 segundos Número de comparações: 5044 Número de atribuições: 9976

Decimo setimo array: sorted$_a rray_5$000

Tempo de execução: 0.015588 segundos Número de comparações: 32004 Número de atribuições: 61808

Decimo oitavo array: sorted$_a rray_1$0000

Tempo de execução: 0.030288 segundos Número de comparações: 69008 Número de atribuições: 133616

Decimo nono array: sorted$_a rray_2$5000

Tempo de execução: 0.079329 segundos Número de comparações: 188476 Número de atribuições: 367232

Vigesimo array: sorted$_a rray_5$0000

Tempo de execução: 0.168777 segundos Número de comparações: 401952 Número de atribuições: 784464

Vigesimo primeiro array: sorted$_a rray_7$5000

Tempo de execução: 0.264639 segundos Número de comparações: 624316 Número de atribuições: 1218928

Vigesimo segundo array: sorted$_a rray_1$00000

Tempo de execução: 0.351124 segundos Número de comparações: 853904 Número de atribuições: 1668928

4.5 Quick Sort

Primeiro Input

Input.txt

Tempo de execução: 0.001992 segundos Número de comparações: 11286 Número de trocas: 6546

Segundo array: randomly$_s huffled_a rray_1$000

Tempo de execução: 0.002 segundos Número de comparações: 11286 Número de trocas: 6546

Terceiro array: randomly$_s huffled_a rray_5$000

Tempo de execução: 0.011553 segundos Número de comparações: 71058 Número de trocas: 43202

$QuartoArray : randomly_s huffled_a rray_1$0000

Tempo de execução: 0.024005 segundos Número de comparações: 157793 Número de trocas: 85752

Quinto Array: randomly$_s huffled_a rray_2$5000

Tempo de execução: 0.061252 segundos Número de comparações: 417959 Número de trocas: 223534

Sexto Array: randomly$_s huffled_a rray_5$0000

Tempo de execução: 0.130269 segundos Número de comparações: 920733 Número de trocas: 463236

Setimo Array: randomly$_s huffled_a rray_7$5000

Tempo de execução: 0.213394 segundos Número de comparações: 1494908 Número de trocas: 770653

Oitavo Array: randomly$_s huffled_a rray_1$00000

Tempo de execução: 0.271243 segundos Número de comparações: 2066493 Número de trocas: 1091692

Nono Array: reverse$_s orted_a rray_1$000

Tempo de execução: 0.044667 segundos Número de comparações: 499500 Número de trocas: 250499

Decimo array: reverse$_s orted_a rray_5$000

Tempo de execução: 1.154645 segundos Número de comparações: 12497500 Número de trocas: 6252499

Decimo primeiro Array: reverse$_s orted_a rray_1$0000

Tempo de execução: 4.580122 segundos Número de comparações: 49995000 Número de trocas: 25004999

Decimo segundo array: reverse$_s orted_a rray_2$5000

Tempo de execução: 28.1353 segundos Número de comparações: 312487500 Número de trocas: 156262499

Decimo terceiro array: reverse$_s orted_a rray_5$0000

Tempo de execução: 110.077596 segundos Número de comparações: 1249975000 Número de trocas: 625024999

Decimo quarto array: reverse$_s orted_a rray_7$5000

Tempo de execução: 253.308849 segundos Número de comparações: 2812462500 Número de trocas: 1406287499

Decimo quinto array: reverse$_s orted_a rray_1$00000

Tempo de execução: 461.833923 segundos Número de comparações: 4999950000 Número de trocas: 2500049999

Decimo sexto Array: sorted$_a rray_1$000

Tempo de execução: 0.06045 segundosA Número de comparações: 499500 Número de trocas: 500499

Decimo setimo array: sorted$_a rray_5$000

Tempo de execução: 1.583868 segundos Número de comparações: 12497500 Número de trocas: 12502499

Decimo oitavo array: sorted$_a rray_1$0000

Tempo de execução: 6.173214 segundos Número de comparações: 49995000 Número de trocas: 50004999

Decimo nono array: sorted$_a rray_2$5000

Tempo de execução: 38.997139 segundos Número de comparações: 312487500 Número de trocas: 312512499

Vigesimo array: sorted$_a rray_5$0000

Tempo de execução: 153.87767 segundos Número de comparações: 1249975000 Número de trocas: 1250024999

Vigesimo primeiro array: sorted$_a rray_7$5000

Tempo de execução: 362.491921 segundos Número de comparações: 2812462500 Número de trocas: 2812537499

Vigesimo segundo array: sorted$_array_1$00000

Tempo de execução: 647.299226 segundos Número de comparações: 4999950000 Número de trocas: 5000049999

5 Discussão

O estudo comparativo dos algoritmos de ordenação oferece uma análise detalhada de suas características teóricas e práticas. Através da medição do tempo de execução, número de comparações e trocas, foi possível identificar as vantagens e limitações de cada algoritmo em diferentes cenários. Verificou-se que algoritmos simples como Bubble Sort e Selection Sort são inadequados para grandes conjuntos de dados devido à sua ineficiência, enquanto algoritmos mais avançados como Merge Sort e Quick Sort oferecem melhor desempenho. A escolha do algoritmo de ordenação ideal deve considerar o tamanho e a natureza dos dados, bem como as restrições de memória e a necessidade de estabilidade. Os resultados obtidos destacam a importância de selecionar o algoritmo de ordenação apropriado para cada situação específica. Algoritmos como Insertion Sort mostraram-se eficientes para listas pequenas ou quase ordenadas, enquanto Merge Sort e Quick Sort são preferíveis para grandes volumes de dados. Além disso, a estabilidade de Merge Sort e a eficiência in-place de Quick Sort e Heap Sort fornecem opções valiosas para diferentes necessidades de processamento.

Os algoritmos demonstraram características de comportamento diferentes conforme foram testados com os diversos Arrays: O uso de memória permaneceu estável, considerando os 32 gigas de RAM e com o VS Code rodando sem outros aplicativos lado a lado, assim como o uso da CPU permaneceu em pico máximo de 15%.

6 Conclusão

A análise empírica dos algoritmos de ordenação revelou diferenças significativas em termos de complexidade temporal, uso de memória e estabilidade. Os algoritmos Bubble Sort e Selection Sort, ambos com complexidade temporal $O(n^2)$, demonstraram desempenho insatisfatório em grandes conjuntos de dados, confirmando a inadequação para aplicações práticas de grande escala (Cormen et al., 2009). O Insertion Sort, embora também possua complexidade $O(n^2)$ no pior caso, mostrou-se eficiente para listas pequenas ou quase ordenadas, devido à sua complexidade no melhor caso. Em contraste, Merge Sort e Heap Sort apresentaram complexidade $O(n \log n)$, tornando-os adequados para grandes volumes de dados, apesar de Merge Sort exigir memória adicional para sublistas temporárias.

Quick Sort, com complexidade média, provou ser altamente eficiente na maioria dos casos, embora sua performance possa degradar para $O(n^2)$ sem

uma escolha adequada do pivô.

Considerando a análise detalhada, cada algoritmo de ordenação tem sua aplicabilidade ideal: os algoritmos Bubble Sort e Selection Sort, que devido à sua simplicidade, em detrimento de sua eficiência, são preferíveis para fins educacionais ou listas muito pequenas, onde a compreensão dos conceitos básicos de ordenação é mais valorizada que a eficiência prática. O algoritmo Insertion Sort é recomendado para listas pequenas ou quase ordenadas, sendo uma excelente escolha para integração em algoritmos híbridos que lidam com sublistas menores. Para grandes conjuntos de dados, Merge Sort é altamente recomendado devido à sua estabilidade e complexidade $O(n \log n)$, sendo especialmente útil quando o uso de memória adicional é viável. Quick Sort, com sua eficiência média, é ideal para a maioria das aplicações práticas, desde que técnicas adequadas de escolha de pivô sejam implementadas para evitar o pior caso. Heap Sort, com sua natureza in-place, é adequado para ambientes com restrições de memória onde a estabilidade dos dados não é crucial.

A análise efetuada permite uma escolha informada do algoritmo de ordenação mais adequado para cada contexto específico.

7 Referências

- GOODRICH, M. T.; TAMASSIA, R.; GOLDWASSER, M. H. *Data Structures and Algorithms in Python.* Irvine, United States: Wiley, 2013.
- SEDGEWICK, R.; WAYNE, K. *Algorithms* (4th edition). Boston, United States: Pearson, 2011.
- KNUTH, D. E. *The Art of Computer Programming, The Sorting and Searching*, Volume 3 (2nd edition). Boston, United States: Addison-Wesley Professional, 1998.
- CORMEN, T. H. et al. *Introduction to Algorithms* (3rd edition). Cambridge, United States: MIT Press, 2009.

8 Tabelas

Bubble Sort - Arrays Ordenados

Quantidade de elementos	1000	5000	10000	25000	50000	75000	100000
Tempo de execução (s)	0,001002	0,004001	0,007002	0,01652	0,033874	0,050231	0,066738
Número de comparações	999	4999	9999	24999	49999	74999	99999
Número de trocas	0	0	0	0	0	0	0

Bubble Sort - Arrays Desordenados

Quantidade de Elementos	Tempo de execução (s)	Número de comparações	Número de trocas
1000	0,046	499.035	251.488
5000	1,173	12.496.510	6.226.258
10.000	4,871	49.977.980	25.190.187
25.000	30,602	312.475.565	155.982.964
50.000	122,871	1.249.902.990	623.011.403
75.000	294,348	2.812.387.034	1.408.752.501
100.000	504,871	4.999.791.797	2.502.357.267

Bubble Sort - Arrays Ordenados Reversamente

Quantidade de Elementos	Tempo de execução (s)	Número de comparações	Número de trocas
1000	0,061	499.500	499.500
5000	1,601	12.497.500	12.497.500
10.000	6,555	49.995.000	49.995.000
25.000	39,401	312.487.500	312.487.500
50.000	318,203	1.249.975.000	1.249.975.000
75.000	362,287	2.812.462.500	2.812.462.500
100.000	643,739	4.999.950.000	4.999.950.000

Bubble Sort - Arrays Ordenados

Quantidade de Elementos	Tempo de execução (s)	Número de comparações	Número de trocas
1000	0,001	999	0
5000	0,004	4.999	0
10.000	0,007	9.999	0
25.000	0,017	24.999	0
50.000	0,034	49.999	0
75.000	0,050	74.999	0
100.000	0,067	99.999	0

Heap Sort - Arrays Desordenados

Quantidade de Elementos	Tempo de execução (s)	Número de comparações	Número de trocas
1000	0,004	16.876	9.114
5000	0,019	107.678	57.144
10.000	0,039	235.346	124.199
25.000	0,109	654.868	343.876
50.000	0,233	1.409.958	737.787
75.000	0,389	2.200.294	1.148.931
100.000	0,513	3.019.448	1.574.757

Heap Sort - Arrays Ordenados Reversamente

Quantidade de Elementos	Tempo de execução (s)	Número de comparações	Número de trocas
1000	0,003	15.965	8.316
5000	0,016	103.227	53.436
10.000	0,037	226.682	116.696
25.000	0,100	633.719	326.586
50.000	0,213	1.366.047	698.892
75.000	0,365	2.138.650	1.095.164
100.000	0,453	2.926.640	1.497.434

Heap Sort - Arrays Ordenados

Quantidade de Elementos	Tempo de execução (s)	Número de comparações	Número de trocas
1000	0,004	17.583	9.708
5000	0,020	112.126	60.932
10.000	0,039	244.460	131.956
25.000	0,109	677.688	361.454
50.000	0,228	1.455.438	773.304
75.000	0,366	2.268.087	1.203.722
100.000	0,496	3.112.517	1.650.854

Quick Sort - Arrays Desordenados

Quantidade de elementos	Tempo de execução (s)	Número de comparações	Número de trocas
1000	0,002	11.286	6.546
5000	0,011	71.058	43.202
10.000	0,024	157.793	85.752
25.000	0,061	417.959	223.534
50.000	0,130	920.733	463.236
75.000	0,213	1.494.908	770.653
100.000	0,271	2.066.493	1.091.692

Quick Sort - Arrays Ordenados Reversamente

Quantidade de elementos	Tempo de execução (s)	Número de comparações	Número de trocas
1000	0,045	499.500	250.499
5000	1,155	12.497.500	6.252.499
10.000	4,580	49.995.000	25.004.999
25.000	28,135	312.487.500	156.262.499
50.000	110,078	1.249.975.000	625.024.999
75.000	253,309	2.812.462.500	1.406.287.499
100.000	461,834	4.999.950.000	2.500.049.999

Quick Sort - Arrays Ordenados

Quantidade de elementos	Tempo de execução (s)	Número de comparações	Número de trocas
1000	0,060	499.500	500.499
5000	1,584	12.497.500	12.502.499
10.000	6,173	49.995.000	50.004.999
25.000	38,997	312.487.500	312.512.499
50.000	153,878	1.249.975.000	1.250.024.999
75.000	362,492	2.812.462.500	2.812.537.499
100.000	647,299	4.999.950.000	5.000.049.999

Merge Sort - Arrays Desordenados

Quantidade de Elementos	Tempo de execução (s)	Número de comparações	Número de atribuições
1000	0,003	8.721	9.976
5000	0,016	55.241	61.808
10.000	0,033	120.448	133.616
25.000	0,093	334.033	367.232
50.000	0,192	718.086	784.464
75.000	0,321	1.121.120	1.218.928
100.000	0,438	1.536.403	1.668.928

Merge Sort - Arrays Reversamente Ordenados

Quantidade de Elementos	Tempo de execução (s)	Número de comparações	Número de atribuições
1000	0,003	4.932	9.976
5000	0,014	29.804	61.808
10.000	0,031	64.608	133.616
25.000	0,079	178.756	367.232
50.000	0,170	382.512	784.464
75.000	0,264	594.612	1.218.928
100.000	0,400	815.024	1.668.928

Merge Sort - Arrays Ordenados

Quantidade de Elementos	Tempo de execução (s)	Número de comparações	Número de atribuições
1000	0,003	5.044	9.976
5000	0,016	32.004	61.808
10.000	0,030	69.008	133.616
25.000	0,079	188.476	367.232
50.000	0,169	401.952	784.464
75.000	0,265	624.316	1.218.928
100.000	0,351	853.904	1.668.928

Insertion Sort - Arrays Desordenados

Quantidade de Elementos	Tempo de execução (s)	Número de comparações	Número de trocas
1000	0,01928	252.480	251.488
5000	0,573234	6.231.251	6.226.258
10.000	2,327012	25.200.177	25.190.187
25.000	14,513901	156.007.953	155.982.964
50.000	62,837826	623.061.390	623.011.403
75.000	143,918671	1.408.827.493	1.408.752.501
100.000	271,0181	2.502.457.259	2.502.357.267

Insertion Sort - Arrays Ordenados Reversamente

Quantidade de Elementos	Tempo de execução (s)	Número de comparações	Número de trocas
1000	0,052575	499.500	499.500
5000	1,175837	12.497.500	12.497.500
10.000	4,597773	49.995.000	49.995.000
25.000	29,271974	312.487.500	312.487.500
50.000	113,519543	1.249.975.000	1.249.975.000
75.000	272,950963	2.812.462.500	2.812.462.500
100.000	479,113865	4.999.950.000	4.999.950.000

Insertion Sort - Arrays Ordenados

Quantidade de Elementos	Tempo de execução (s)	Número de comparações	Número de trocas
1000	0,001002	999	0
5000	0,004001	4.999	0
10.000	0,008003	9.999	0
25.000	0,013003	24.999	0
50.000	0,035786	49.999	0
75.000	0,053027	74.999	0
100.000	0,074097	99.999	0