ALGORITMOS DE ORDENAÇÃO

Roses are red
Violets are grand
Hugo's the best—
Just ask Hugo, he'll help you understand.
Modest? Not quite. Humble? No way.
But according to Hugo, he saves the day.

ALGORITMOS DE ORDENAÇÃO

Os algoritmos de ordenação são essenciais em várias áreas da optimização combinatória e da investigação operacional, sendo utilizados para organizar dados de forma eficiente, melhorar o desempenho de algoritmos de procura e reduzir a complexidade computacional de problemas em larga escala [1].

Em contextos de otimização combinatória, onde para vários problemas se torna fundamental a ordenação prévia de dados, como nas heurísticas para o problema Knapsack, no algoritmo de Kruskal para construção de árvores geradoras mínimas, ou em algoritmos gulosos para cobertura de conjuntos, a escolha do método de ordenação pode impactar significativamente a eficiência global da solução [2].

Este trabalho visa explorar os principais algoritmos de ordenação, destacando as suas características, o seu funcionamento, a complexidade computacional associada e uma análise comparativa entre os mesmos.

Serão abordados algoritmos de ordenação representativos de diferentes estratégias fundamentais, nomeadamente os métodos baseados em troca (*exchange sort*), como o Bubble Sort; em seleção (*selection sort*), como o Selection Sort; em inserção (*insertion sort*), como o Insertion Sort; e em divisão e conquista, como o Merge Sort e o Quick Sort. Para além destes, serão também analisados algoritmos baseados em distribuição, como o Counting Sort e o Radix Sort, os quais não se baseiam em comparações diretas entre elementos.

O objetivo é compreender como a ordenação pode ser utilizada de forma estratégica para melhorar a eficiência de algoritmos em diversos cenários, e como otimizações na própria ordenação podem levar a avanços significativos em problemas computacionalmente intensivos.

II. Algoritmos de Ordenação

Nesta secção apresentam-se nove algoritmos de ordenação selecionados, com uma descrição individual do seu princípio de funcionamento, análise da complexidade temporal, identificação das principais vantagens e limitações, bem como a caracterização dos cenários de melhor caso, pior caso e caso médio.

A. BUBBLE SORT

Tlvz para cada alg:

- Como funciona por tópicos a explicar em texto tp
- ~~Pseudocogisos~~ tlvz são demasiado grandes e técnicos para apresentação
- Justificacao da complexidade?
- Pior melhor e médio caso
- Gif a funcionar

B. SELECTION SORT

C. INSERTION SORT

D. COUNTING SORT

E. RADIX SORT

F. QUICK SORT

G. MERGE SORT

H. HEAP SORT

1. TIMSORT

RESULTAD

que n é o número de elementos a ordenar, k o maior valor presente e d o número máximo de algarismos de um elemento.

Algoritmo	Complexidade
Bubble Sort	$\mathcal{O}(n^2)$
Selection Sort	$\mathcal{O}(n^2)$
Insertion Sort	$\mathcal{O}(n^2)$
Counting Sort	$\mathcal{O}(n+k)$
Radix Sort	$\mathcal{O}(n \times d)$
Quick Sort	$\mathcal{O}(n \log n)$

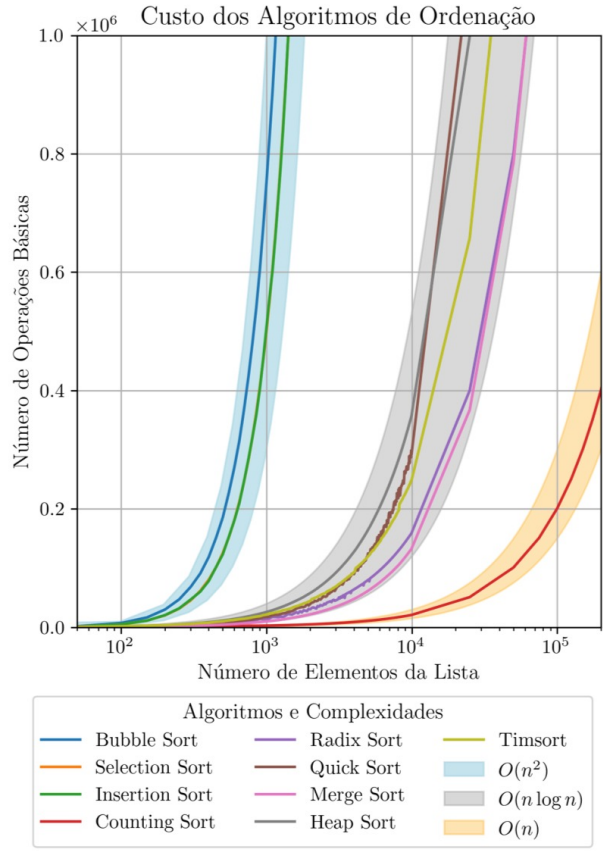

Figura 1: Análise do número de operações básicas dos diferentes algoritmos face a diferentes tamanhos de listas.

No caso do Counting Sort, é possível verificar que a sua complexidade se comporta como $\mathcal{O}(n)$, uma vez que o valor máximo presente na lista ($k = 1000$) é significativamente menor do que os tamanhos das listas utilizadas nesta simulação. Assim, o termo adicional k não afeta substancialmente o crescimento do tempo de execução.

Relativamente aos algoritmos Bubble Sort, Selection Sort e Insertion Sort, observa-se um crescimento quadrático consistente com a complexidade teórica $\mathcal{O}(n^2)$. Nota-se que o desempenho do Selection Sort não se destaca visualmente no gráfico, por se sobrepor ao do Insertion Sort, evidenciando o mesmo comportamento assintótico.

Quanto aos restantes algoritmos, com exceção do Radix Sort, todos apresentam um crescimento compatível com a complexidade $\mathcal{O}(n \log n)$, conforme previsto teoricamente. Já o Radix Sort, cuja complexidade é dada por $\mathcal{O}(n \cdot b)$, onde b representa o número de dígitos necessários para representar

CONCLUS

IV. Conclusão

Neste trabalho, foram analisados e comparados nove algoritmos de ordenação, abrangendo abordagens clássicas (como Bubble Sort, Selection Sort e Insertion Sort), algoritmos mais eficientes baseados em paradigmas de dividir-para-conquistar (como Quick Sort, Merge Sort e Heap Sort), e algoritmos especializados como Counting Sort, Radix Sort e Timsort. Para cada algoritmo, foram descritos o funcionamento, a complexidade teórica, as vantagens e limitações, e o comportamento prático observado através de simulações em *Python*.

A análise empírica permitiu confirmar, de forma geral, as previsões teóricas. Algoritmos quadráticos apresentaram um crescimento acentuado com o aumento do tamanho da lista, tornando-se inviáveis para grandes volumes de dados. Por outro lado, algoritmos com complexidade $\mathcal{O}(n \log n)$ revelaram-se muito mais eficientes e escaláveis. O caso do Radix Sort destacou a importância de considerar não apenas a complexidade assintótica, mas também os fatores constantes ocultos e as condições específicas dos dados, uma vez que, apesar da sua complexidade quase linear, o seu desempenho prático aproxima-se frequentemente dos algoritmos $\mathcal{O}(n \log n)$ devido ao custo das múltiplas passagens.

Conclui-se, assim, que a escolha do algoritmo de ordenação mais adequado não deve ser feita apenas com base na complexidade teórica, mas também considerando as características concretas dos dados, o contexto de utilização e os requisitos práticos de desempenho e memória.