Universidade Católica de Brasília

Pró-Reitoria Acadêmica
Curso de Ciência da Computação

ALGORITMOS DE ORDENAÇÃO
Estrutura de Dados

Autor: Arthur Lemos Bendini
Orientador: Weldes Lima Oliveira

Brasília – DF
2024

Arthur Lemos Bendini

ALGORITMOS DE ORDENAÇÃO

Trabalho desenvolvido no curso de graduação em Ciência da Computação da Universidade Católica de Brasília, como requisito parcial para obtenção do título de (Bacharel/Licenciado) em Ciência da Computação.

Orientador: Weldes Lima Oliveira

Brasília

2024

Resumo

Este trabalho refere-se à análise dos algoritmos de ordenação e seus funcionamentos, abordando os principais métodos, como *Merge Sort*, *Quick Sort, Shell Sort* e *Heap Sort*. São discutidas suas complexidades, eficiência e aplicações práticas, bem como suas vantagens e desvantagens em diferentes contextos. O objetivo é oferecer uma visão comparativa dos algoritmos, destacando situações ideais para a utilização de cada um, com base em critérios como desempenho e consumo de recursos.

Palavras-chave: algoritmos de ordenação, complexidade, eficiência, desempenho.

Abstract

This paper analyzes sorting algorithms and their mechanisms, addressing the main methods such as Merge Sort, *Quick Sort, Shell Sort and Heap Sort. It discusses their complexities, efficiency, and practical applications, as well as the advantages and disadvantages within different contexts. The objective is to provide a comparative overview of the algorithms, highlighting the ideal situations for the use of each one, based on criteria such as performance and resource consumption.*

Keywords*: sorting algorithms, complexity, efficiency, performance.*

SUMÁRIO

1. INTRODUÇÃO

Os algoritmos de ordenação desempenham um papel fundamental em diversas áreas da computação, sendo uma das operações mais comuns e essenciais em sistemas de processamento de dados. Sua importância reside no fato de que muitos problemas computacionais exigem que dados estejam ordenados para serem manipulados de maneira eficiente. Aplicações como busca, organização de registros em bancos de dados e otimização de estruturas de dados dependem de algoritmos de ordenação para alcançar alto desempenho.

2. OBJETIVO

analisar alguns algoritmos de ordenação, explorando suas características, complexidades computacionais e casos de uso. Serão abordados métodos como:

- Shell Sort
- Merge Sort
- Quick Sort
- Heap Sort

Com comparações baseadas em fatores como eficiência, consumo de recursos e aplicabilidade em diferentes situações.

2.1 OBJETIVO GERAL

Fornecer uma compreensão sólida sobre qual algoritmo é mais adequado em cenários específicos, considerando o tamanho dos dados, o nível de desorganização inicial e a necessidade de otimização de tempo e espaço.

2.2 OBJETIVOS ESPECÍFICOS

Contribuir para o entendimento dos principais aspectos teóricos e práticos dos algoritmos de ordenação, auxiliando o leitor a escolher a melhor abordagem para seus próprios projetos e necessidades computacionais.

3. ALGORITMOS DE ORDENAÇÃO

Um algoritmo de ordenação é usado para reordenar um *array* (vetor) ou lista de elementos de acordo com um operador de comparação entre os elementos. Um operador de comparador é usado para decidir a nova ordem dos elementos na respectiva estrutura de dados.

Por exemplo: Observe que a lista de números abaixo é ordenada em uma ordem crescente dos seus valores. Isto é, o número com menor valor será colocado antes dos números de valores maiores.

Array desordenado

6	49	12	7	67	50

Array ordenado (ordem crescente)

6	7	12	49	50	67

3.1 O QUE É SORTING?

Se refere à reordenação de elementos em uma lista utilizando um padrão específico, seja de forma crescente ou decrescente, em pares, ímpares etc. Um operador de comparação é usado para decidir qual será a nova ordem dos elementos de acordo com a condição atribuída.

3.2 CARACTERÍSTICAS

- Complexidade de tempo
  - Quanto tempo leva para o algoritmo concluir seu processo. Dependendo da quantidade de elementos e a forma de como eles estão distribuídos inicialmente, este fator poderá ser afetado, podendo consumir mais tempo para finalizar o algoritmo.

- Espaço auxiliar
  - A quantidade de espaço extra (além do tamanho das listas) necessária para a ordenação. Por exemplo, Insertion Sort necessita de 1 espaço auxiliar para realizar a operação.

- Estabilidade
  - Um algoritmo de ordenação é considerado estável se a ordem relativa de elementos iguais é preservada após a ordenação. Isso é importante em certas aplicações onde a ordem original de elementos iguais deve ser mantida.

- Ordenação In-Place
    - Um algoritmo de ordenação que não precisa de memória adicional para ordenar os dados. Isso é importante quando a memória disponível é limitada, ou quando os dados não podem ser movidos.

- Adaptabilidade
    - Um algoritmo de ordenação adaptável é aquele que toma vantagem de uma ordem pré-existente dos dados para melhorar a performance.

3.3 APLICABILIDADE

Os algoritmos de ordenação têm ampla aplicação em diversas áreas do desenvolvimento, desempenhando um papel crucial em operações como algoritmos de busca e gerenciamento de dados. Sua utilidade é notória em áreas como bancos de dados, machine learning e sistemas operacionais, onde a eficiência no processamento e organização de informações é essencial para o desempenho geral dos sistemas.

Além disso, são fundamentais para otimizar algoritmos que lidam com grandes volumes de dados, como em buscas binárias, que requerem listas previamente ordenadas para garantir eficiência. Em bancos de dados, a ordenação facilita a criação de índices, melhorando o tempo de resposta em consultas.
No contexto de machine learning, algoritmos de ordenação auxiliam na preparação e limpeza de dados, além de serem usados em técnicas de classificação e agrupamento. Já em sistemas operacionais, a ordenação é aplicada na organização de filas de processos e na alocação de recursos, garantindo uma gestão eficiente do tempo e do espaço de execução.

4. NOTAÇÃO *BIG O*

A notação *Big O* é uma forma matemática de descrever o comportamento assintótico de um algoritmo em relação ao tempo ou espaço de execução em função do tamanho da entrada. Ela fornece uma maneira de expressar a complexidade do algoritmo, permitindo a comparação entre diferentes algoritmos independentemente da implementação específica.

4.1 CONCEITOS BÁSICOS

- **Complexidade de Tempo**: Refere-se ao tempo que um algoritmo leva para ser executado em relação ao tamanho da entrada. Por exemplo, um algoritmo que leva tempo proporcional a *nnn* é descrito como *O(n)*, onde *nnn* é o tamanho da entrada.

- **Complexidade de Espaço**: Refere-se à quantidade de memória que um algoritmo utiliza em relação ao tamanho da entrada. Assim como na

complexidade de tempo, se um algoritmo requer espaço proporcional a *nnn*, sua complexidade de espaço é *O(n)*.

4.2 CLASSES COMUNS DE COMPLEXIDADE

- *O(1)*: Complexidade constante. O tempo de execução não depende do tamanho da entrada.
- *O(log n)*: Complexidade logarítmica. O tempo de execução cresce de forma logarítmica em relação ao tamanho da entrada.
- *O(n)*: Complexidade linear. O tempo de execução cresce linearmente com o aumento do tamanho da entrada.
- *O(n log n)*: Complexidade linear-logarítmica. Comum em algoritmos eficientes de ordenação, como *Merge Sort* e *Quick Sort*.
- *O(n²)*: Complexidade quadrática. O tempo de execução cresce de forma quadrática com o aumento do tamanho da entrada, comum em algoritmos de ordenação simples, como *Bubble Sort* e *Insertion Sort*.

4.3 IMPORTÂNCIA

A notação *Big O* é fundamental para entender a eficiência de algoritmos e ajuda os desenvolvedores a tomar decisões informadas ao escolher o algoritmo adequado para resolver um problema específico. Ao analisar algoritmos, a ênfase deve ser colocada no comportamento assintótico, uma vez que isso fornece uma melhor visão do desempenho do algoritmo em grandes entradas.

5. MERGE SORT

O *Merge Sort* é um algoritmo de ordenação que segue o paradigma de divisão e conquista. A lista é continuamente dividida em duas partes até que cada sublista contenha apenas um elemento. Em seguida, os pares de sublistas são mesclados em ordem crescente (ou decrescente, dependendo da preferência) até que toda a lista esteja combinada e ordenada.
A vantagem do *Merge Sort* está na sua eficiência garantida de O(n log n) em todos os casos, incluindo o pior cenário. Isso ocorre porque, independentemente da ordem inicial dos dados, o processo de divisão e fusão mantém estrutura de tempo. No entanto, o *Merge Sort* não é um algoritmo in-place, pois requer espaço adicional para armazenar os sub-arrays durante o processo de fusão. Por ser estável, ele mantém a ordem relativa dos elementos iguais, o que pode ser desejável em certas aplicações.

6. QUICK SORT

O *Quick Sort* também segue o paradigma de divisão e conquista, mas de forma diferente do *Merge Sort*. Em vez de dividir a lista ao meio, ele seleciona um elemento como pivô e organiza os elementos ao redor desse pivô, colocando os menores à esquerda e os maiores à direita. O algoritmo então é aplicado recursivamente às duas sublistas.
A eficiência do *Quick Sort* depende da escolha do pivô. No melhor e caso médio, ele tem complexidade de O(n log n), o que o torna mais rápido que o *Merge Sort* em média, especialmente porque é um algoritmo *In-Place* e não necessita de espaço extra significativo. No entanto, no pior cenário (quando o pivô escolhido é o menor ou maior elemento), sua complexidade é $O(n^2)$. É importante notar que o *Quick Sort* não é estável, ou seja, ele pode alterar a ordem relativa de elementos iguais.

7. SHELL SORT

O *Shell Sort* é uma versão generalizada do *Insertion Sort* que melhora a eficiência ao ordenar elementos que estão a uma certa distância entre si. Ele usa uma sequência de distâncias (ou *gaps*) que diminuem ao longo do tempo. A cada passo, o algoritmo aplica o *Insertion Sort* nos elementos que estão separados pela distância atual do *gap*, até que o *gap* seja 1, quando o *Shell Sort* se comporta como um *Insertion Sort* tradicional.
A eficiência do *Shell Sort* depende da escolha da sequência de gaps. No caso médio, sua complexidade é geralmente O(n log n), mas pode variar de acordo com a implementação. No pior caso, ele pode atingir $O(n^2)$. Embora seja mais eficiente do que o *Insertion Sort* em listas grandes e parcialmente ordenadas, ele não é estável.

8. HEAP SORT

O *Heap Sort* utiliza uma estrutura de dados chamada *heap*, que é uma árvore binária completa onde o valor de cada nó pai é maior (ou menor) que o de seus filhos. O *Heap Sort* primeiro transforma o *array* em um *heap* máximo (ou mínimo), e então remove o maior (ou menor) elemento do *heap* e o coloca no final da lista. Esse processo é repetido até que todos os elementos estejam ordenados.
O *Heap Sort* tem complexidade O(n log n) em todos os casos, sendo eficiente e sem o pior cenário de $O(n^2$ que o *Quick Sort* pode enfrentar. No entanto, ele não é estável e, embora seja in-place, o desempenho prático do *Heap Sort* pode ser inferior ao *Quick Sort* devido ao maior número de comparações feitas durante a reorganização da estrutura de heap.

9. COMPARATIVO DE EFICIÊNCIA

Cada algoritmo de ordenação apresentado tem suas próprias vantagens e desvantagens. O *Merge Sort* oferece uma solução robusta com tempo garantido de O(n log n), mas com um consumo de espaço adicional. O *Quick Sort* é geralmente o mais rápido em prática, mas pode ter desempenho ruim no pior caso sem otimizações. O *Shell Sort* é uma melhoria sobre o *Insertion Sort*, útil para listas parcialmente ordenadas, enquanto o *Heap Sort* é confiável em todos os casos, mas pode ser mais lento que o *Quick Sort* em média.

A escolha do algoritmo ideal depende do contexto. Em cenários onde a estabilidade e o consumo de memória são críticos, o *Merge Sort* pode ser preferível. *O Quick Sort* é uma excelente opção para listas grandes quando se pode otimizar a escolha do pivô. Já o *Heap Sort* se destaca quando é importante garantir um desempenho consistente, e o Shell Sort é útil quando as listas são pequenas ou parcialmente ordenadas.

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Estabilidade	In-Place
Merge Sort	O(n log n)	O(n log n)	O(n log n)	Sim	Não
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	Não	Sim
Shell Sort	O(n log n)	$O(n \log^2 n)$	$O(n^2)$	Não	Sim
Heap Sort	O(n log n)	O(n log n)	O(n log n)	Não	Sim

10. CONCLUSÃO

A análise dos algoritmos de ordenação apresentados revela um panorama rico em técnicas e estratégias que atendem a diferentes necessidades. Cada algoritmo, ao longo de suas implementações, destaca-se por características que podem ser mais ou menos favoráveis dependendo do contexto em que é aplicado.

O *Merge Sort*, com sua estrutura robusta, é ideal para situações em que a estabilidade é crucial e o consumo de memória adicional é aceitável. Por outro lado, o *Quick Sort* se destaca em cenários práticos, onde um bom desempenho em médias é desejado, embora tenha um potencial de degradação em casos adversos sem as devidas otimizações.

Além disso, o *Heap Sort* oferece uma solução confiável com desempenho consistente, enquanto o *Shell Sort* é particularmente vantajoso para listas que já possuem certa ordem.

Essa diversidade de opções ilustra a importância de selecionar o algoritmo mais apropriado, levando em consideração não apenas a complexidade computacional, mas também fatores como a estrutura dos dados e os requisitos específicos da aplicação.

A escolha do algoritmo de ordenação pode impactar significativamente a eficiência de um sistema, e entender as nuances de cada técnica é fundamental para o desenvolvimento de software de alta performance. Com isso, a habilidade de analisar e comparar algoritmos é uma competência essencial para programadores e engenheiros de software, permitindo a entrega de soluções mais eficazes e adequadas às demandas do mercado.

11. REFERÊNCIAS BIBLIOGRÁFICAS

1 GEEKSFORGEEKS. ***Sorting Algorithms*.** Disponível em: https://www.geeksforgeeks.org/sorting-algorithms/. Acesso em: 26 set. 2024.

2 VISUALGO. ***Sorting Algorithms Visualization.*** Disponível em: https://visualgo.net/en/sorting. Acesso em: 26 set. 2024.

3 NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. ***Dictionary of Algorithms and Data Structures***. Disponível em: https://xlinux.nist.gov/dads/. Acesso em: 26 set. 2024.

4 BBC BYTESIZE. **Searching and sorting algorithms – OCR Merge Sort.** Disponível em: https://www.bbc.co.uk/bitesize/guides/zjdkw6f/revision/5. Acesso em: 27 set. 2024.