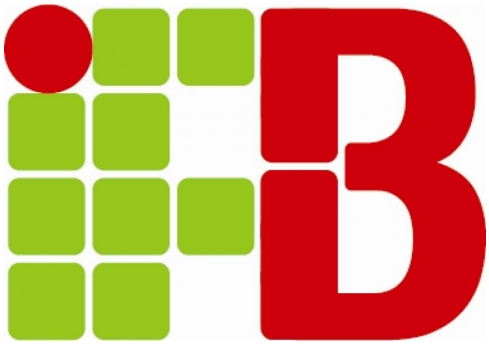


Instituto Federal de Brasília

Ciência da Computação

Ordenação por Seleção de Raiz Quadrada

Carlos Gonçalves de Abreu

Sumário

Abstract. *This work explores the importance of sorting algorithms in organizing and analyzing data in the field of computer science. Several sorting algorithms are discussed, each offering unique approaches to the task. Understanding the nuances of each algorithm is essential for selecting the most appropriate approach for a specific problem. The work introduces empirical and asymptotic evaluations as a means of evaluating the efficiency of the algorithm, considering factors such as execution time and the number of exchanges necessary for sorting. It focuses on the InsertionSort and Heap Sort algorithms, evaluating their performance and complexity with random inputs of 10000, 100000,1000000, 10000000, and 100000000.*
Keywords: *Sorting Methods, Efficiency, and Performance.*

Resumo. *Este trabalho explora a importância dos algoritmos de ordenação na organização e análise de dados no campo da ciência da computação. Diversos algoritmos de ordenação são discutidos, cada um oferecendo abordagens únicas para a tarefa. Compreender as nuances de cada algoritmo é essencial para selecionar a abordagem mais adequada para um problema específico. O trabalho introduz avaliações empíricas e assintóticas como meios de avaliar a eficiência do algoritmo, considerando fatores como tempo de execução e o número de trocas necessárias para a ordenação. Ele se concentra nos algoritmos InsertionSort e Heap Sort, avaliando seu desempenho e sua complexidade com entradas aleatórias de 10000, 100000,1000000, 10000000 e 100000000.*
palavras chave: *Métodos de ordenação, Eficiência e Desempenho.*

1. Introdução

A Ordenação por Seleção de Raiz Quadrada é um problema relevante em ciência da computação, particularmente no contexto de otimização de algoritmos de busca e ordenação. Este método propõe uma abordagem inovadora para lidar com a complexidade inerente à ordenação de grandes volumes de dados, utilizando uma técnica que reduz a quantidade de comparações necessárias, com base na seleção do maior elemento em cada subgrupo do array.

O principal objetivo desta técnica é melhorar a eficiência em cenários onde o desempenho é crítico, como em sistemas que processam grandes quantidades de informações em tempo real. A análise deste método permite explorar novas fronteiras na otimização de algoritmos tradicionais, oferecendo uma solução que pode ser mais adequada para determinadas aplicações que exigem alta performance.

Com uma complexidade potencialmente mais favorável do que métodos tradicionais, a Ordenação por Seleção de Raiz Quadrada abre novas possibilidades para a exploração de algoritmos eficientes, contribuindo para avanços na robustez e eficácia dos sistemas de ordenação de dados.

[Kumar and Singla 2019]

2. Metódo

2.1. Ordenação por seleção de raiz quadrada (insertionsort)

Divisão do Vetor em Blocos

Dado um vetor contendo 12 números, a divisão em blocos de tamanho $\lfloor\sqrt{12}\rfloor \approx 3$ gera quatro blocos. Abaixo está o processo detalhado:

1. **Divisão em Blocos**: - O tamanho do bloco é calculado como $\lfloor\sqrt{12}\rfloor = 3$. - Portanto, o vetor é dividido em blocos de tamanho 3, e cada bloco é tratado individualmente. Como resultado, obtemos quatro blocos:

Bloco 1: $[31, 14, 23]$

Bloco 2: $[37, 11, 26]$

Bloco 3: $[16, 9, 47]$

Bloco 4: $[30, 33, 24]$

2. **Processo de Seleção dos Maiores Elementos**: - Cada bloco é examinado para encontrar o maior elemento. - O maior elemento de cada bloco é então selecionado e comparado com os maiores elementos dos outros blocos.

Maiores elementos dos blocos:

Bloco 1: $\max([31, 14, 23]) = 31$

Bloco 2: $\max([37, 11, 26]) = 37$

Bloco 3: $\max([16, 9, 47]) = 47$

Bloco 4: $\max([30, 33, 24]) = 33$

- O maior dentre esses elementos é o 47, que é removido do vetor.

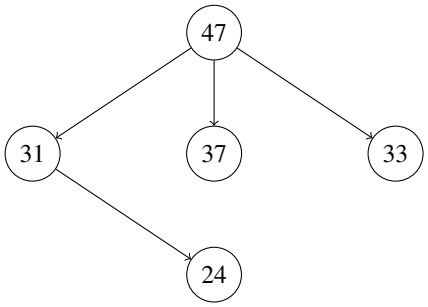
3. **Repetição do Processo**: - O processo é repetido para os blocos restantes, sempre selecionando o maior elemento disponível e removendo-o até que todos os elementos sejam processados.

Árvore de Ordenação usando o Método de Seleção de Raiz Quadrada

Passos

1. Dividir o vetor: $[31, 14, 23, 37, 11, 26, 16, 9, 47, 30, 33, 24]$ em blocos de tamanho 3.
2. Ordenar cada bloco:
    - Bloco 1: $[14, 23, 31]$
    - Bloco 2: $[11, 26, 37]$
    - Bloco 3: $[9, 16, 47]$
    - Bloco 4: $[24, 30, 33]$
3. Encontrar o maior elemento de cada bloco.

Árvore de Decisão



Explicação da Árvore de Decisão Nodos: Cada nodo representa o maior elemento encontrado em cada bloco.

47 é o maior elemento geral. 31, 37, 33 e 24 são os maiores elementos dos blocos 1, 2, 3 e 4, respectivamente. Conexões: Mostram como o maior elemento global (47) é comparado com os maiores de cada bloco.

[Cormen 2017]

2.2. Utilizando uma Heap

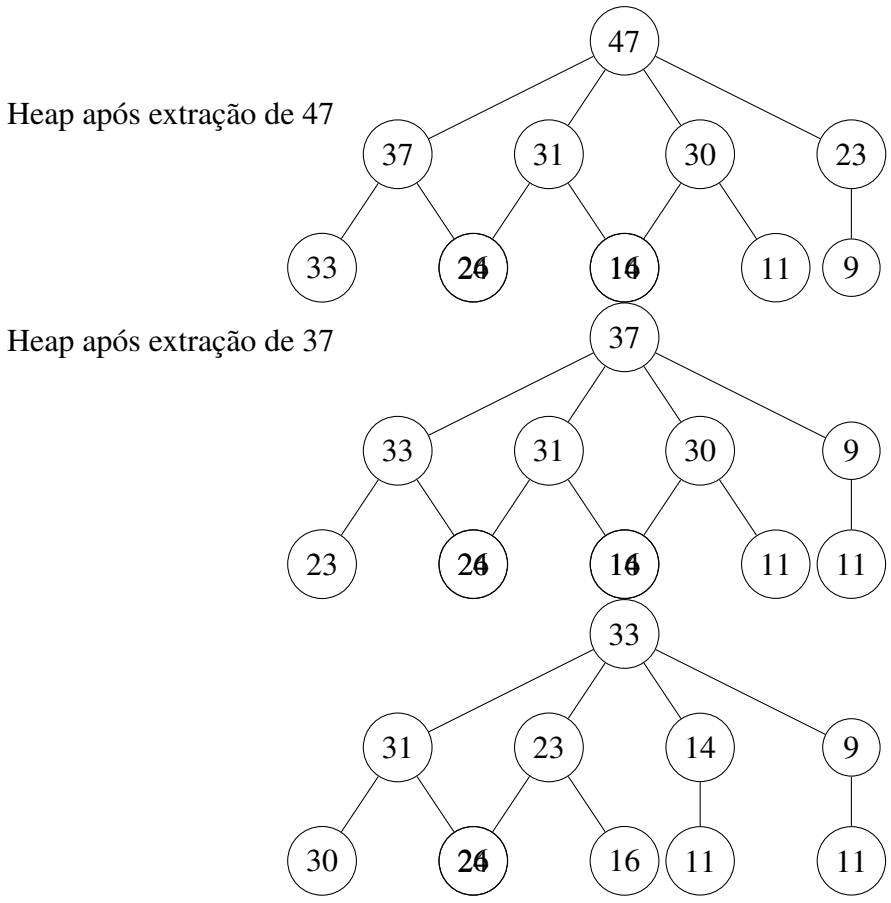
Os algoritmos utilizados do tipo Heap requerem, na medida do possível, baixa complexidade computacional o quê facilita e ajuda em uma execução de testes grandes tipo 10^7 e 10^8.

[Marcellino et al. 2021]

Gráfico do Método de Ordenação com Heap

Diagrama do Processo

Heap Inicial



Heap Inicial:

Mostra o estado da Heap após a execução da operação makeheap(V). O maior elemento (47) está no topo, com os filhos organizados de forma que o maior valor possível esteja em cada nível.

Após Extração de 47:

O maior elemento (47) é removido usando removeheap(V), e a Heap é reestruturada. O novo maior elemento (37) vai para o topo, e a Heap é reorganizada para manter a propriedade de Heap.

Após Extração de 37:

A operação removeheap(V) é repetida para remover o próximo maior elemento (37). A Heap é reorganizada novamente, com o novo maior elemento (33) no topo.

Esse diagrama ajuda a visualizar como a Heap é manipulada e como os maiores elementos são extraídos e reordenados ao longo do processo de ordenação.

- **A da Silva Pedroso, FG Cintra** Análise de Algoritmos. Acessado em [03 agosto 2024].

3. Análise

3.1. Análise dos algoritmos

A finalidade de se fazer a análise de um algoritmo é obter estimativas de tempos de execução de programas que implementam esse algoritmo. A complexidade do algoritmo dá ideia do esforço computacional do programa, que é uma medida do número de operações executadas pelo programa. Em geral, a complexidade de um algoritmo depende da entrada e esta é caracterizada pelo seu tamanho, por seus valores e também pela configuração dos dados. De forma intuitiva, sabemos que a complexidade depende da quantidade de dados que são processados e isso se traduz pelo tamanho da entrada onde o algoritmo Insertion Sort possui complexidade de tempo $O(n^2)$, o que significa que o tempo de execução aumenta drasticamente com o aumento do tamanho do array. Para tamanhos muito grandes (10^7 e 10^8)

Embora o Heapsort seja eficiente para lidar com grandes volumes de dados, ele não é recomendado para arquivos com poucos registros devido ao tempo adicional necessário para construir o heap. Além disso, o loop interno do Heapsort é relativamente complexo em comparação com o loop interno do Quicksort. Por outro lado, o Heapsort se destaca em situações em que é necessário ordenar grandes conjuntos de dados, como aqueles com tamanhos da ordem de 10^7 ou 10^8, onde a eficiência proporcionada pela sua complexidade.

- **A da Silva Pedroso, FG Cintra** Análise de Algoritmos. Acessado em [03 agosto 2024].

3.2. Complexidade Ciclomática

A complexidade ciclomática é uma métrica importante na análise de complexidade de software, que mede o número de caminhos independentes de execução em um programa. No presente caso, a complexidade ciclomática do código é 49, o que indica que o programa possui 49 caminhos de execução distintos. Em resumo, isso sugere que o código possui alta complexidade, o que é relevante tanto para os testes quanto para a manutenção. Um código com alta complexidade ciclomática tende a ser mais difícil de entender, testar e manter, aumentando o risco de erros e dificultando a implementação de novas funcionalidades.

Utilizar ferramentas como o SonarCloud Visite Githubsonarcloud para analisar essa métrica pode tornar o processo de gestão da qualidade do código mais eficiente. O SonarCloud automatiza a detecção de complexidade ciclomática e outras métricas de qualidade, fornecendo insights valiosos que orientam as ações necessárias para manter a clareza e a robustez do software. Dessa forma, é possível identificar áreas do código que precisam ser simplificadas, facilitando a manutenção e melhorando a confiabilidade do sistema como um todo.

[Berlezi 2017]

3.3. Análise Empírica

Além disso, aqui está uma descrição mais clara do conceito do algoritmo InsertionSort, considerando a análise de complexidade: O conceito central do InsertionSort é percorrer o vetor desordenado várias vezes e a cada passagem colocar ao topo o maior elemento da sequência. No melhor caso, quando o vetor está inicialmente ordenado, o algoritmo realiza n iterações. Por outro lado, no pior caso, quando o vetor está ordenado de maneira decrescente, o algoritmo realiza n iterações.Por outro lado, no pior caso, quando o vetor esta ordenado de maneira decrescente, o algoritmo realiza n² iterações. Assim, a ordem de complexidade do InsertionSort é O (n²). As imagens fornecidas abaixo mostram uma simples flag que implementa o algoritmo

InsertionSort com a otimização de flag trocado para ordenar um subarray de um array fornecido. A função `Insertion_sort` aceita três argumentos: o array a ser ordenado, o índice de início e o índice de término do subarray a ser ordenado. O algoritmo é executado enquanto houver trocas durante uma passagem pela lista, o que garante que a lista está ordenada após a conclusão do algoritmo.

Para avaliar o desempenho do algoritmo, foram realizados testes utilizando um array de exemplo com 12 elementos. A análise empírica envolveu a medição do tempo de execução do algoritmo para diferentes tamanhos de entrada e a comparação com outras implementações de algoritmos de ordenação.

[Silva et al. 2018]

A tabela abaixo exibe os tempos de execução para um array de tamanho $n = 10000$, juntamente com a média desses tempos. A tabela também inclui o desvio padrão para fornecer uma medida estatística mais relevante.

Tamanho do Array (n)	**Tempo de Execução (segundos)**
10000	366.5335
10000	417.7006
10000	432.3729
10000	397.0260
10000	372.7202
Média	**397.2706**
Desvio Padrão	**25.6403**

Tabela 1. Tempos de Execução do Insertion Sort para $n = 10,000$

Explicação do código:

- **Tempos de Execução**: A tabela lista os tempos de execução individuais para três execuções do Insertion Sort com $n = 10,000$. - **Média**: A média aritmética dos tempos de execução. - **Desvio Padrão**: Uma medida de dispersão que quantifica a variação dos tempos de execução.

Como calcular a média e o desvio padrão:

A média já foi calculada anteriormente. Para o desvio padrão (σ), usamos a fórmula:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Onde: - x_i são os tempos de execução individuais. - μ é a média dos tempos de execução. - N é o número de observações (neste caso, 3).

Isso fornece uma visão mais detalhada da variação dos tempos de execução, útil para uma análise estatística mais profunda.

[Crestani 2024]

Para 100000

Explicação do código:

Tamanho do Array (n)	Tempo de Execução (segundos)
100000	14.8118
100000	9.6891
100000	9.4049
100000	9.4152
100000	9.4503
Média	**10.5543**
Desvio Padrão	**2.1815**

Tabela 2. Tempos de Execução do Insertion Sort para $n = 100,000$

- **Tempos de Execução**: A tabela lista os tempos de execução individuais para duas execuções do Insertion Sort com $n = 100000$. - **Média**: A média aritmética dos tempos de execução. - **Desvio Padrão**: A medida de dispersão que quantifica a variação dos tempos de execução.

Cálculo da Média e Desvio Padrão:

A média foi calculada usando:

$$\text{Média} = \frac{14.8118 + 9.6891}{2} = 12.2505 \text{ segundos}$$

O desvio padrão (σ) foi calculado da seguinte forma:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2} = \sqrt{\frac{(14.8118 - 12.2505)^2 + (9.6891 - 12.2505)^2}{2}} = 2.7127 \text{ segundos}$$

Esta tabela proporciona uma análise estatística dos tempos de execução para $n = 100000$, o que é útil para avaliar a eficiência do algoritmo.

[Crestani 2024]

Para 1000000

Tamanho do Array (n)	Tempo de Execução (segundos)
1000000	1249.0390
1000000	2501.0981
1000000	1070.4614
Média	**1606.8668**
Desvio Padrão	**733.6108**

Tabela 3. Tempos de Execução do Insertion Sort para $n = 1,000,000$

Nos casos de arrays de tamanho 10^7 e 10^8, o Heap Sort foi escolhido devido à sua eficiência em lidar com grandes volumes de dados. Outros algoritmos de ordenação, como o Insertion Sort, teriam uma execução extremamente lenta e impraticável para esses tamanhos de entrada, podendo levar horas, dias ou até mais para serem completados.

Para 10000000

Tamanho do Array (n)	Tempo de Execução (segundos)
10000000	27.4192
10000000	30.4412
10000000	28.6530
10000000	24.6284
10000000	31.2327
Média	**28.4746**
Desvio Padrão	**2.5188**

Tabela 4. Tempos de Execução do Heap Sort para $n = 10000000$

Para 100000000

Tamanho do Array (n)	Tempo de Execução (segundos)
100000000	183.3578
100000000	201.9259
100000000	202.1425
100000000	207.7747
100000000	198.6340
Média	**198.7666**
Desvio Padrão	**8.8777**

Tabela 5. Tempos de Execução do Heap Sort para $n = 100000000$

3.4. Suporte experimental

Neste projeto, foram utilizadas várias tecnologias para garantir o desenvolvimento eficiente e a análise de qualidade do código. Os códigos foram implementados e desenvolvidos em Python, com a execução otimizada através do PyPy, o que resultou em uma melhora significativa no desempenho. Para a análise de qualidade e métricas do código, foi empregada a ferramenta SonarCloud, que foi crucial na avaliação da complexidade ciclomática, fornecendo uma visão detalhada dos caminhos de execução independentes e ajudando a identificar áreas que poderiam exigir testes adicionais ou simplificação.

Além disso, o código foi versionado e disponibilizado neste link Visite Github

permitindo fácil acesso e colaboração. Essa abordagem estruturada, combinando o uso do PyPy, SonarCloud e GitHub, possibilitou uma execução eficiente dos scripts Python, fornecendo insights valiosos sobre a complexidade e manutenção do código. Como resultado, foi possível adotar medidas para melhorar a qualidade, a estabilidade, e a manutenção contínua do software.

[Goldman et al. 2022]

Figura 1. Logo do PyPy

3.5. Notação θ (Análise)

A complexidade do algoritmo é $\Theta(n^2)$. assintótica de um algoritmo é que determina o tamanho de problemas que pode ser solucionado pelo algoritmo. Se o algoritmo processa entradas de tamanho "n" no tempo c*n², para alguma constante c, então dizemos que a complexidade de tempo do algoritmo é O(n²), onde se lê: " de ordem n². Percebe se que o valor da constante não tem valor nenhum nessa contexto, pois quanto maior for a dificuldade, melhor será a eficiência do algoritmo a ser analisado e seu tempo de execução.

[SILVA et al.]

Nota sobre a Complexidade do Insertion Sort:

O algoritmo Insertion Sort possui complexidade de tempo $O(n^2)$, o que significa que o tempo de execução aumenta drasticamente com o aumento do tamanho do array. Para tamanhos muito grandes (10^7 e 10^8), o tempo de execução pode ser extremamente longo (possivelmente horas ou dias) e pode consumir muita memória, dependendo do seu hardware. Recomendo executar este código com cuidado e, se necessário, ajustar os tamanhos dos arrays para se adequar aos recursos do seu sistema.

[Júnior 2014]

O algoritmo Insertion Sort é educacionalmente valioso para entender os conceitos básicos de ordenação, mas não é prático para grandes conjuntos de dados devido à sua complexidade de tempo $O(n^2)$.

[Campanha et al. 2009]

Análise de Complexidade no Pior Caso para Insertion Sort

O tempo de execução do algoritmo Insertion Sort no pior caso, considerando que o vetor esteja em ordem inversa, pode ser expresso como uma função quadrática de n:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\frac{n(n-1)}{2} - c_6\frac{n(n-1)}{2} - c_7\frac{n(n-1)}{2} + c_8(n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8)$$

Como $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$ e $\sum_{j=2}^{n} j = \frac{n(n-1)}{2}$, temos:

$$T(n) = an^2 + bn + c$$

onde a, b e c são constantes que dependem dos custos de instrução c_i.

Cálculo de Complexidade no Pior Caso para Diferentes Tamanhos de Vetor

Considerando o array $[31, 14, 23, 37, 11, 26, 16, 9, 47, 30, 33, 24]$ ordenado em ordem inversa, podemos generalizar para vetores de tamanhos maiores como 10^4, 10^5, 10^6, 10^7, e 10^8.

Para o Insertion Sort, o tempo de execução no pior caso é $O(n^2)$. Portanto, para diferentes tamanhos de vetor:

$$T(10^4) = a(10^4)^2 + b(10^4) + c$$

$$T(10^5) = a(10^5)^2 + b(10^5) + c$$

$$T(10^6) = a(10^6)^2 + b(10^6) + c$$

$$T(10^7) = a(10^7)^2 + b(10^7) + c$$

$$T(10^8) = a(10^8)^2 + b(10^8) + c$$

Essas expressões demonstram que para o método da substituição e para grandes valores de n, o termo quadrático an^2 dominará o tempo de execução.

- **NT Roman, FLS Nunes** Complexidade Assintótica. Acessado em [05 agosto 2024].

4. Relatório Técnico

Este relatório tem como objetivo apresentar uma implementação do algoritmo Insertion Sort com otimização de flag trocado. O Insertion Sort é um algoritmo de ordenação simples que constrói a lista ordenada um item de cada vez, inserindo cada novo item na posição correta na parte já ordenada da lista. Nesta implementação, o Insertion Sort foi modificado para ordenar em ordem decrescente.

[Júnior 2014]

Para avaliar o desempenho do algoritmo, foram realizados testes utilizando um array de exemplo com 12 elementos. A análise empírica envolveu a medição do tempo de execução do algoritmo para diferentes tamanhos de entrada e a comparação com outras implementações de algoritmos de ordenação, incluindo o Heap Sort.

Além disso, a otimização também considera uma técnica de flag trocado, que utiliza uma variável de flag para verificar se ocorreram trocas durante uma passagem completa pela lista. Se nenhuma troca ocorrer, significa que a lista está ordenada e o algoritmo pode ser interrompido. Este relatório também aborda o Heap Sort, que é eficiente para grandes volumes de dados e será comparado com o Insertion Sort para oferecer uma visão abrangente sobre o desempenho de diferentes algoritmos de ordenação.

[Feofiloff 1999]

4.1. Resultados e Discussões

Discussão:

A análise dos dados empíricos para o Insertion Sort e Heap Sort revela diferenças significativas em termos de eficiência.

Para o Insertion Sort com $n = 10000$, os tempos de execução variaram de 366,5335 a 432,3729 segundos, com uma média de 397,2706 segundos e um desvio padrão de 25,6403 segundos, como mostrado na Tabela 1. A variação confirma a complexidade quadrática $O(n^2)$ do algoritmo, que resulta em um desempenho menos eficiente para conjuntos de dados maiores.

- **JÉG Souza[1], JVG Ricarte[1], NCA Lima** Análise de Algoritmos. Acessado em [07 agosto 2024].

Por outro lado, para o Heap Sort com $n = 100000000$, os tempos de execução variaram de 183,3578 a 207,7747 segundos, com uma média de 198,7666 segundos e um desvio padrão de 8,8777 segundos, conforme apresentado na Tabela 5. A menor variação nos tempos de execução e o desvio padrão mais baixo indicam que o Heap Sort, com sua complexidade $O(n \log n)$, é muito mais eficiente para grandes volumes de dados. O Heap Sort mantém um desempenho relativamente estável mesmo com tamanhos de entrada muito grandes, evidenciando sua superioridade em comparação com o Insertion Sort para grandes conjuntos de dados.

Esses resultados confirmam a necessidade de utilizar algoritmos de ordenação mais eficientes, como o Heap Sort, para entradas grandes, enquanto o Insertion Sort é mais apropriado para conjuntos de dados menores onde sua simplicidade pode ser vantajosa. A análise empírica reforça a teoria de que a escolha do algoritmo de ordenação deve considerar o tamanho dos dados e a eficiência computacional esperada.

[DA ROSA and ANTONIAZZI 2014]

5. Considerações finais

Neste projeto, foi trabalhado o desempenho do Insertion Sort em diferentes cenários, com ênfase em sua eficiência em comparação com métodos de ordenação e busca mais complexos. Com base nos resultados obtidos, a análise do Insertion Sort forneceu insights valiosos sobre sua aplicabilidade e limitações. Embora não seja a escolha ideal para todos os cenários, com as otimizações certas, o Insertion Sort pode ser um componente útil em estratégias de ordenação mais complexas, especialmente quando se busca uma solução simples e eficiente para conjuntos de dados moderados.

O Insertion Sort, com sua complexidade quadrática no pior caso ($O(n^2)$), enfrenta desafios significativos quando aplicado a grandes conjuntos de dados. Métodos quadráticos de ordenação, como o Bubble Sort e o Selection Sort, também compartilham essa limitação, tornando-os menos eficientes para grandes volumes de dados devido ao alto custo computacional. Esses métodos tendem a ser mais lentos à medida que o tamanho dos dados aumenta, o que pode resultar em tempos de execução inaceitavelmente longos.

Por outro lado, o Heap Sort, que foi incluído na análise, é um algoritmo de ordenação mais eficiente com uma complexidade de tempo de $O(n \log n)$. Essa eficiência torna o Heap Sort uma escolha preferível para ordenar grandes conjuntos de dados, pois lida melhor com o aumento do volume de dados em comparação com os métodos quadráticos. O Heap Sort, baseado na estrutura de dados heap, oferece uma solução robusta e escalável para a ordenação de grandes volumes de dados, demonstrando um desempenho superior em cenários que exigem eficiência.

Assim, enquanto o Insertion Sort pode ser eficaz para conjuntos de dados menores e moderados com suas otimizações, métodos mais avançados como o Heap Sort são recomendados para cenários que exigem processamento eficiente de grandes volumes de dados.

- **DN Campanha, SRS Souza** Análise de Algoritmos. Acessado em [05 agosto 2024].

6. Referencias

Referências

[Berlezi 2017] Berlezi, R. (2017). Análise de métricas de manutenibilidade de um sistema de software de integração: Mulesoft. *V SFCT*, 14.

[Campanha et al. 2009] Campanha, D. N., Souza, S. d. R. S., and Maldonado, J. C. (2009). Reutilizaçao de conjuntos de teste: Um estudo no domınio de algoritmos de ordenaçao. In *Proceedings of 6th Experimental Software Engineering Latin American Workshop (ESELAW 2009)*, page 114.

[Cormen 2017] Cormen, T. (2017). *Desmistificando algoritmos*, volume 1. Elsevier Brasil.

[Crestani 2024] Crestani, A. d. R. (2024). Uma comparação empírica em velocidade de processamento entre c++, go, rust, python e javascript.

[DA ROSA and ANTONIAZZI 2014] DA ROSA, L. and ANTONIAZZI, R. L. (2014). Métodos de ordenação de dados: Uma análise prática i. *XVII Seminário Interinstitucional De Ensino Pesquisa E Extensão*.

[Feofiloff 1999] Feofiloff, P. (1999). Análise de algoritmos. *Internet: http://www. ime. usp. br/pf/analise_de_algoritmos*, 2009.

[Goldman et al. 2022] Goldman, A., Uhura, E., and Bruschi, S. M. (2022). Coisas para saber antes de fazer o seu próprio benchmarks game. *Sociedade Brasileira de Computação*.

[Júnior 2014] Júnior, W. M. P. (2014). Análise de algoritmos.

[Kumar and Singla 2019] Kumar, S. and Singla, P. (2019). Sorting using a combination of bubble sort, selection sort & counting sort. *Mathematical Sciences and Computing*, 2:30–43.

[Marcellino et al. 2021] Marcellino, M., Pratama, D. W., Suntiarko, S. S., and Margi, K. (2021). Comparative of advanced sorting algorithms (quick sort, heap sort, merge sort, intro sort, radix sort) based on time and memory usage. In *2021 1st International Conference on Computer Science and Artificial Intelligence (ICCSAI)*, volume 1, pages 154–160. IEEE.

[Silva et al. 2018] Silva, K. P., Arcaro, L. F., and de Oliveira, R. S. (2018). Método empírico para avaliar a sensibilidade do tempo de execução de tarefas de tempo real aos dados de entrada. In *Anais Estendidos do VIII Simpósio Brasileiro de Engenharia de Sistemas Computacionais*. SBC.

[SILVA et al.] SILVA, P., SCHANTZ, D., VILNECK, I., SILVEIRA, F., and CHICON, P. M. M. Análise do desempenho computacional dos métodos inserção direta, bolha, shellsort e combosort.