Relatório - Algoritmos de Ordenação - Seus custos e parâmetros de eficiência

Bruno Peres[1]**,Eric Bernardes**[1]

[1]Departamento de Computação – Universidade Federal do Espírito Santo (UFES)
Alegre – ES – Brasil

[2]Departamento de Sistemas e Computação

bruno.peres@edu.ufes.br, eric.dordenuni@edu.ufes.br

Resumo. *A questão do acúmulo massivo de dados assume uma relevância crescente e, como resposta a essa problemática, torna-se essencial a utilização de algoritmos de ordenação. Esses algoritmos, que se manifestam como processos lógicos, desempenham um papel crucial na organização de estruturas lineares, independentemente de sua natureza física ou virtual. O escopo deste estudo é realizar uma avaliação por meio de experimentação dos dados gerados ao aplicar os algoritmos de ordenação Bubble Sort, Insertion Sort (Binary and Direct), Selection Sort, Merge Sort, Heap Sort, Quick Sort, Shell Sort e Bucket Sort, analisando suas utilidades e eficiências. Para tal propósito, foram conduzidos três conjuntos de testes, cada um composto por três vetores de diferentes tamanhos, para cada algoritmo, e os resultados obtidos foram submetidos a uma análise comparativa.*

Abstract. *The issue of massive data accumulation is gaining increasing relevance, and in response to this problem, the use of sorting algorithms becomes essential. These algorithms, acting as logical processes, play a crucial role in organizing linear structures, regardless of their physical or virtual nature. The scope of this study is to conduct an evaluation through experimentation of the data generated by applying sorting algorithms such as Bubble Sort, Insertion Sort (Binary and Direct), Selection Sort, Merge Sort, Heap Sort, Quick Sort, Shell Sort and Bucket Sort analyzing their utilities and efficiencies. For this purpose, three sets of tests were conducted, each consisting of three vectors of different sizes for each algorithm, and the obtained results were subjected to a comparative analysis.*

1. Indrodução

Este estudo tem como objetivo conduzir uma avaliação experimental dos dados gerados pelos algoritmos de ordenação, tais como Bubble Sort, Insertion Sort (Binary and Direct), Selection Sort, Merge Sort, Heap Sort, Quick Sort, Shell Sort e Bucket Sort. É relevante observar que o algoritmo Quick Sort será explorado em três variações distintas, todas relacionadas à escolha do pivô. As opções disponíveis incluem o elemento inicial, o elemento central e um elemento mediano, onde os elementos escolhidos para a mediana serão o primeiro, o central e o último. A abordagem metodológica em relação aos algoritmos será mais superficial, focando particularmente nas comparações e custos associados a cada um.

Para atingir esse propósito, foram conduzidos três conjuntos de testes, cada um composto por três vetores de tamanhos distintos para cada algoritmo. Os dados coletados foram submetidos a uma análise comparativa, visando proporcionar uma compreensão mais aprofundada das vantagens e desvantagens de cada algoritmo em termos de eficiência e utilidade.

Apesar de ser possível implementar os algoritmos em diversas estruturas lineares, optou-se por realizar os testes exclusivamente com algarismos numéricos inteiros para obter dados quantitativos mais claros. Inicialmente, utilizando um vetor com capacidade para 100 valores distintos, avaliamos os tempos de execução, o número de comparações e o número de movimentações para organizar listas ordenadas de forma ascendente, descendente e desordenada. Esse procedimento foi replicado com vetores de tamanhos 1.000 e 10.000. A ênfase recairá na análise técnica das comparações e custos, proporcionando insights valiosos sobre a escolha adequada de algoritmos em diferentes cenários.

2. Resultados e Testes

A Linguagem C foi deliberadamente selecionada como uma das linguagens para a codificação dos algoritmos, devido à sua otimização significativa no tempo de execução. Nesse contexto, a IDE Visual Studio Code foi adotada como o ambiente de desenvolvimento. Quanto ao hardware, a máquina na qual os testes foram conduzidos possui as seguintes configurações:

- Sistema Operacional: Linux Mint 21.2 Cinnamon(5.8.4);
- Memória RAM: 16,00 GB (DDR4 3200MHz);
- Processador: AMD - 5500 ryzen 5 - Hexa Core - 3.6GHz até Até 4.2GHz;

3. Instruções de Execução

Visite o repositório no GitHub por meio do link a seguir:

- https://github.com/EricBernardes/Trabalho-Ordenacao-C

Com o Git instalado em seu sistema, utilize o comando em seu terminal para obter uma cópia do repositório, outra opção é baixar o arquivo e descompactá-lo. Após isso, acesse o diretório correspondente e execute o comando make.

Na linha de comando, insira ./gera -a 1000 para utilizar o programa. As opções de flags disponíveis são: -a, para criar um conjunto de dados aleatório; -c, para dados em ordem crescente; e -d, para dados em ordem decrescente. Certifique-se de substituir ”1000”pelo valor desejado, representando a quantidade de dados que você pretende organizar. Este é um exemplo de como o comando pode ser estruturado.

./gera -a 1000

Imediatamente após, insira o comando ./ordena bolha 1000 entrada.txt. Após o ./gera, especifique o nome do algoritmo desejado. As opções disponíveis são:

- **bolha**
- **insercaodireta**
- **insercaobinaria**
- **shellsort**

- **selecaodireta**
- **heapsort**
- **quicksortini**
- **quicksortcentro**
- **quicksortmediana**
- **mergesort**
- **radixsort**
- **bucketsort**

O valor que segue imediatamente após o nome do algoritmo deve ser o mesmo utilizado com o comando ./gera. O último parâmetro é o nome do arquivo que será gerado com os dados, obrigatoriamente entrada.txt. Abaixo está um exemplo dos dois comandos necessários para utilizar o programa:

- **./gera -a 1000**
- **./ordena bolha 1000 entrada.txt**

Os resultados irão aparecer na pasta Saídas, com os respectivos nome dos algoritmos utilizados.

4. Teste - Vetor de 100 Elementos

Lista	**Ordem Crescente**			**Ordem Decrescente**			**Ordem Aleatória**		
Algoritmo	**Tempo(s)**	**Comp.**	**Trocas**	**Tempo(s)**	**Comp.**	**Trocas**	**Tempo(s)**	**Comp.**	**Trocas**
Bolha	1.2×10^{-5}	4950	0	2.1×10^{-5}	4950	4950	3.4×10^{-5}	4950	2642
Inserção direta	2×10^{-6}	99	0	1.3×10^{-5}	99	99	1.5×10^{-5}	99	97
Inserção binária	3×10^{-6}	573	0	1.8×10^{-5}	480	4950	1.2×10^{-5}	527	2642
Shellsort	6×10^{-6}	342	0	1.1×10^{-5}	572	230	2×10^{-5}	784	442
Seleção direta	1.7×10^{-5}	4950	0	1.7×10^{-5}	4950	50	1.4×10^{-5}	4950	96
Heapsort	1.1×10^{-5}	690	640	9×10^{-6}	566	516	8×10^{-6}	622	572
Quicksort(ini)	2×10^{-5}	4950	99	1.7×10^{-5}	7400	2599	9×10^{-6}	918	562
Quicksort(centro)	6×10^{-6}	480	63	8×10^{-6}	386	112	9×10^{-6}	468	190
Quicksort(mediana)	1.7×10^{-5}	4950	99	1.3×10^{-5}	2548	99	1.1×10^{-5}	512	188
Mergesort	1×10^{-5}	672	0	1.1×10^{-5}	672	573	1.3×10^{-5}	672	334
Radixsort	1.5×10^{-5}	300	600	1.5×10^{-5}	300	600	1.3×10^{-5}	300	600
Bucketsort	8×10^{-6}	480	63	8×10^{-6}	386	112	1.9×10^{-5}	468	190

5. Teste - Vetor de 1000 Elementos

Vetor[1000]									
Lista	**Ordem Crescente**			**Ordem Decrescente**			**Ordem Aleatória**		
Algoritmo	**Tempo(s)**	**Comp.**	**Trocas**	**Tempo(s)**	**Comp.**	**Trocas**	**Tempo(s)**	**Comp.**	**Trocas**
Bolha	1.164×10^{-3}	499500	0	2.325×10^{-3}	499500	499500	1.470×10^{-3}	499500	255039
Inserção direta	6.0×10^{-6}	999	0	1.213×10^{-3}	999	999	4.52×10^{-4}	999	993
Inserção binária	4.7×10^{-5}	8977	0	1.8×10^{-5}	480	4950	4.58×10^{-4}	8587	255039
Shellsort	4.7×10^{-5}	5457	0	7.2×10^{-5}	9377	3920	2.2×10^{-4}	14516	9059
Seleção direta	7.41×10^{-4}	499500	0	1.223×10^{-3}	499500	500	1.932×10^{-3}	499500	987
Heapsort	8.0×10^{-5}	10208	9708	7.6×10^{-5}	8816	8316	9.5×10^{-5}	9543	9043
Quicksort(ini)	1.071×10^{-3}	499500	999	1.095×10^{-3}	749000	250999	9.9×10^{-5}	12952	7854
Quicksort(centro)	2.8×10^{-5}	7987	511	3.8×10^{-5}	6996	1010	9.3×10^{-5}	7059	2605
Quicksort(mediana)	1.515×10^{-3}	499500	999	5.63×10^{-4}	250498	999	9.9×10^{-5}	8962	2617
Mergesort	6.9×10^{-5}	9976	0	1.11×10^{-4}	9976	8977	1.45×10^{-4}	9976	4826
Radixsort	1.33×10^{-4}	4000	8000	1.5×10^{-4}	4000	8000	1.29×10^{-4}	4000	8000
Bucketsort	8.0×10^{-6}	480	63	6.6×10^{-5}	6996	1010	6.2×10^{-5}	7059	2605

6. Teste - Vetor de 10000 Elementos

Vetor[10000]			
Lista	**Ordem Crescente**		
Algoritmo	**Tempo(s)**	**Comp.**	**Trocas**
Bolha	6.7953×10^{-2}	49995000	0
Inserção direta	2.8×10^{-5}	9999	0
Inserção binária	4.01×10^{-4}	123617	0
Shellsort	4.9×10^{-4}	75243	0
Seleção direta	8.5357×10^{-2}	49995000	0
Heapsort	8.9×10^{-4}	136956	131956
Quicksort(ini)	9.6742×10^{-2}	49995000	9999
Quicksort(centro)	3.59×10^{-4}	113631	5904
Quicksort(mediana)	9.7035×10^{-2}	49995000	9999
Mergesort	9.16×10^{-4}	133616	0
Radixsort	1.664×10^{-3}	50000	100000
Bucketsort	6.44×10^{-4}	113631	5904

Vetor[10000]			
Lista	**Ordem Decrescente**		
Algoritmo	**Tempo(s)**	**Comp.**	**Trocas**
Bolha	1.41669×10^{-1}	49995000	49995000
Inserção direta	8.9835×10^{-2}	9999	9999
Inserção binária	7.7718×10^{-2}	113631	49995000
Shellsort	9.64×10^{-4}	128947	53704
Seleção direta	1.00805×10^{-1}	49995000	5000
Heapsort	8.96×10^{-4}	121696	116696
Quicksort(ini)	1.00797×10^{-1}	74990000	25009999
Quicksort(centro)	3.64×10^{-4}	103644	10904
Quicksort(mediana)	5.4543×10^{-2}	25004998	9999
Mergesort	1.155×10^{-3}	133616	123617
Radixsort	1.649×10^{-3}	50000	100000
Bucketsort	5.74×10^{-4}	103644	10904

Vetor[10000]			
Lista	**Ordem Aleatória**		
Algoritmo	**Tempo(s)**	**Comp.**	**Trocas**
Bolha	1.21124×10^{-1}	49995000	25062990
Inserção direta	5.472×10^{-2}	9999	9988
Inserção binária	3.0653×10^{-2}	118946	25062990
Shellsort	3.929×10^{-3}	243013	167770
Seleção direta	9.3298×10^{-2}	49995000	9992
Heapsort	1.217×10^{-3}	129206	124206
Quicksort(ini)	1.184×10^{-3}	197310	107126
Quicksort(centro)	8.79×10^{-4}	111264	33438
Quicksort(mediana)	1.173×10^{-3}	114947	34155
Mergesort	1.345×10^{-3}	133616	65148
Radixsort	1.648×10^{-3}	50000	100000
Bucketsort	1.278×10^{-3}	111264	33438

7. Resultados

QuickSort Início (Pivô no Início):

Melhor Desempenho em Listas Aleatórias: O QuickSort Início (Tempo: 1.184×10^{-3} segundos) apresentou um desempenho notavelmente rápido em listas aleatórias, evidenciando sua eficiência quando a lista está desordenada. Desempenho em Listas Crescentes ou Decrescentes: Mostrou um desempenho um pouco inferior em listas quase ordenadas (Tempo Médio: 9.87695×10^{-2} segundos), como esperado. O QuickSort Início pode sofrer em cenários em que a lista já está parcialmente ordenada.

QuickSort Centro (Pivô no Centro):

Desempenho em Listas Aleatórias: O QuickSort Centro (Tempo: 8.79×10^{-4} segundos) teve um desempenho intermediário em listas aleatórias, escolhendo o pivô no centro. Desempenho em Listas Crescentes ou Decrescentes: Apresentou um desempenho moderado em listas quase ordenadas (Tempo Médio: 1.238×10^{-3} segundos). A escolha do pivô no centro ajuda a evitar alguns problemas de partições desequilibradas.

QuickSort Mediana de Três (Pivô da Mediana de Três):

Melhor Desempenho em Listas Decrescentes: O QuickSort Mediana de Três (Tempo: 5.4543×10^{-2} segundos) destacou-se em listas decrescentes, sugerindo uma escolha eficaz de pivô para esses cenários específicos.

Vetor Aleatório

Melhor Desempenho: Bucket Sort

Tempo decorrido: 6.44×10^{-4} segundos
Comparações: 113631
Trocas: 5904

Segundo Melhor: Radix Sort

Tempo decorrido: 1.649×10^{-3} segundos
Comparações: 50000
Trocas: 100000

Pior Desempenho: Bubble Sort

Tempo decorrido: 1.41669×10^{-1} segundos
Comparações: 49995000
Trocas: 49995000

Vetor Crescente

Melhor Desempenho: Merge Sort

Tempo decorrido: 9.16×10^{-4} segundos
Comparações: 133616
Trocas: 0

Segundo Melhor: Shell Sort

Tempo decorrido: 4.9×10^{-4} segundos
Comparações: 75243
Trocas: 0

Pior Desempenho: Bubble Sort

Tempo decorrido: 6.7953×10^{-2} segundos
Comparações: 49995000
Trocas: 0

Vetor Decrescente

Melhor Desempenho: Heap Sort

Tempo decorrido: 8.96×10^{-4} segundos
Comparações: 121696
Trocas: 116696

Segundo Melhor: Shell Sort

Tempo decorrido: 4.9×10^{-4} segundos
Comparações: 75243
Trocas: 0

Pior Desempenho: Bubble Sort

Tempo decorrido: 1.41669×10^{-1} segundos
Comparações: 49995000
Trocas: 49995000

Métodos como Bucket Sort, Merge Sort e Radix Sort mostraram-se consistentemente eficientes em diferentes cenários, destacando-se em listas aleatórias e crescentes. Bubble Sort demonstrou desempenho inferior em todos os cenários, sendo especialmente ineficiente em listas aleatórias e decrescentes. Heap Sort e Shell Sort apresentaram resultados competitivos, destacando-se em listas decrescentes. A escolha do método deve levar em consideração não apenas o tempo de execução, mas também outros fatores como requisitos de memória e complexidade do algoritmo.

8. Observações

As informações referentes aos vetores de tamanho 10000 foram prioritariamente consideradas, uma vez que proporcionam resultados mais precisos. No que tange aos tempos médios, procedeu-se à média dos tempos obtidos nos vetores ordenados de forma crescente e decrescente. Tal abordagem visa garantir uma análise robusta e abrangente do desempenho dos métodos de ordenação em diferentes contextos, reforçando a confiabilidade e a representatividade dos resultados.

É saliente destacar que o algoritmo de ordenação QuickSort foi avaliado em três distintas variações, variando a escolha do pivô. Nesse contexto, observou-se que o desempenho do algoritmo apresenta variações significativas em diferentes cenários, em consonância com a escolha do pivô. A análise dessas variações oferece insights valiosos sobre a sensibilidade do algoritmo às diferentes estratégias de escolha do pivô, contribuindo

para uma compreensão mais aprofundada de seu comportamento em diversos contextos de aplicação.

9. Referências

Tabelas [Souza et al. 2017] Heurística dos algoritmos [Beder]

Referências

[Beder] Beder, D. M. Complexidade de algoritmos.

[Souza et al. 2017] Souza, J. E., Ricarte, J. V. G., and de Almeida Lima, N. C. (2017). Algoritmos de ordenação: Um estudo comparativo. *Anais do Encontro de Computação do Oeste Potiguar ECOP/UFERSA (ISSN 2526-7574)*, (1).