Projeto e Análise de Algoritmos
Implementação e Comparação de Algoritmos de Ordenação

Thiago O. da Silva[1], **Rhuan M. O. M. de Carvalho**[2], **Vicente C. G. de S. Carvalho**[3]

[1]*Departamento de Ciência da Computação – Universidade Federal do Piauí (UFPI)*
Teresina – PI – Brazil

[2]*Departamento de Ciência da Computação – Universidade Federal do Piauí (UFPI)*
Teresina – PI – Brazil

[3]*Departamento de Ciência da Computação – Universidade Federal do Piauí (UFPI)*
Teresina – PI – Brazil

{*thiago_silva*,*rhuanmourao*}@ufpi.edu.br, vicentecleyton@gmail.com

Abstract. *This report aims to discuss the implementation and comparison of sorting algorithms in order to evaluate their efficiency and performance.*

Resumo. *Este relatório tem como objetivo discutir a implementação e a comparação de algoritmos de ordenação com o objetivo de avaliar sua eficiência e desempenho.*

1. Metodologia

A implementação dos algoritmos de ordenação envolve traduzir as descrições dos algoritmos em código executável. Existem diferentes técnicas e abordagens para implementar algoritmos de ordenação, com destaque ao BubbleSort, InsertionSort, MergeSort, HeapSort, QuickSort. Além disso, cada algoritmo possui características específicas em relação à complexidade de tempo, consumo de recursos e facilidade de implementação.

Neste estudo, com o objetivo de comparar o desempenho, será realizado a implementação e análise desses algoritmos utilizando a linguagem de programação Python, considerando diferentes tamanhos de entradas para realizar as ordenações, variando de 100 a 200.000 elementos.

2. Implementação

2.1. BubbleSort:

Estratégia: O Bubble Sort compara repetidamente pares de elementos adjacentes e os troca de posição se estiverem na ordem errada. Essa estratégia é repetida até que toda a lista esteja ordenada.

Complexidade: Possui uma complexidade de tempo de $O(n^2)$, onde ”n” é o número de elementos na lista. Isso significa que o tempo de execução do algoritmo aumenta de forma quadrática à medida que o tamanho da entrada aumenta.

2.2. InsertionSort:

O Insertion Sort é um algoritmo de ordenação que segue uma estratégia de inserção. Ele percorre uma lista de elementos e os insere em sua posição correta na parte já ordenada do vetor, construindo gradualmente uma lista ordenada.

Estratégia: Comparar o elemento atual com os elementos anteriores na parte ordenada do vetor, deslocando os elementos maiores uma posição para a direita até encontrar a posição correta para inserir o elemento atual.

Complexidade: A complexidade do Insertion Sort é de $O(n^2)$ no pior caso, onde n é o número de elementos no vetor. No melhor caso, quando o vetor já está ordenado, a complexidade é de O(n).

2.3. MergeSort:

O Merge Sort é um algoritmo de ordenação que segue uma abordagem de divisão e conquista. Ele divide a lista em pequenas partes, ordena cada parte separadamente e, em seguida, combina as partes ordenadas para obter a lista finalmente ordenada.

Estratégia: Consiste em dividir recursivamente a lista ao meio até que cada parte tenha apenas um elemento. Em seguida, combina as partes de forma ordenada, comparando os elementos das partes e mesclando-os em uma nova lista. Esse processo de divisão e combinação é repetido até que a lista completa esteja ordenada.

Complexidade: A complexidade do Merge Sort é de O(n log n), onde n é o número de elementos na lista. Com isso, torna-se eficiente mesmo para listas grandes, pois mantém a mesma complexidade independentemente do grau de ordenação da lista.

2.4. HeapSort:

O Heap Sort utiliza a propriedade de um heap binário, que é uma árvore binária completa em que o valor de cada nó é maior ou igual aos valores de seus filhos, explorando essa propriedade para ordenar os elementos da lista.

Estratégia: Envolve a construção de um heap máximo a partir da lista de elementos. Isso é feito percorrendo a lista de elementos e refazendo o heap a cada iteração. Em seguida, o algoritmo realiza a etapa de ordenação, que consiste em extrair o maior elemento do heap e colocá-lo na posição correta no final da lista, repetindo até que todos os elementos estejam ordenados.

Complexidade: A complexidade do Heap Sort é de O(n log n), onde n é o número de elementos na lista.

2.5. QuickSort:

O Quick Sorte seleciona um elemento da lista, chamado de pivô, e reorganiza os elementos de forma que todos os elementos menores que o pivô fiquem à sua esquerda e os elementos maiores fiquem à sua direita. Em seguida, o algoritmo é aplicado recursivamente nas sublistas à esquerda e à direita do pivô até que a lista esteja totalmente ordenada.

Estratégia: Envolve a escolha de um pivô e a partição da lista com base nesse pivô, que pode ser selecionado de diversas maneiras. Em seguida, o algoritmo percorre a lista e compara cada elemento com o pivô, movendo-os para a posição correta.

Complexidade: Varia dependendo da escolha do pivô. No pior caso, em que o pivô divide a lista de forma desbalanceada, a complexidade é de $O(n^2)$. No entanto, em média, o Quick Sort possui uma complexidade de O(n log n)

3. Comparação dos resultados obtidos

Após a implementação dos algoritmos de ordenação, foi realizado uma comparação entre eles. Essa comparação é feita com base em métricas como tempo de execução e número de comparações. Além disso, os resultados foram subdivididos de acordo com os diferentes tipos de dados de entrada, como conjuntos ordenados, inversamente ordenados e aleatórios. Por conseguinte, essa comparação permitiu identificar o desempenho relativo de cada algoritmo em diferentes cenários, ajudando a selecionar o algoritmo mais adequado para uma determinada aplicação.

É importante destacar que, como o Insertion e o Bubble são algoritmos que crescem quadraticamente em complexidade com o tamanho de entrada, não foi possível determinar o tempo de execução quando as entradas possuem um tamanho de 150000 e 200000.

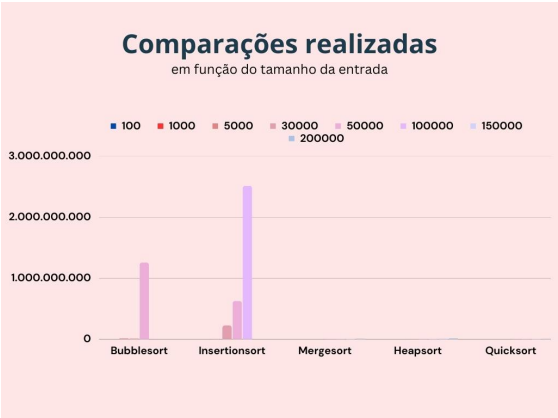
Tempo de execução dos algoritmos (em segundos)

Vetor aleatório								
	100	1000	5000	30000	50000	100000	150000	200000
Bubble	0.001s	0.104s	2.720s	99.277s	279.065s	1192.681s	indeterminado	indeterminado
Insertion	0.0003s	0.057s	1.398s	51.710s	143.891s	619.926s	indeterminado	indeterminado
Merge	0.0s	0.0045s	0.026s	0.180s	0.318s	0.672s	1.047s	1.432s
Heap	0.001s	0.006s	0.039s	0.284s	0.514s	1.112s	1.719s	2.400s
Quick	0.0s	0.0026s	0.015s	0.112s	0.171s	0.395s	0.580s	0.858s
Vetor crescente								
	100	1000	5000	30000	50000	100000	150000	200000
Bubble	0.001s	0.055s	1.441s	52.030s	143.469s	600.556s	1358.423s	indeterminado
Insertion	0.0s	0.0s	0.002s	0.007s	0.010s	0.022s	0.032s	0.042s
Merge	0.0s	0.003s	0.019s	0.137s	0.242s	0.510s	0.803s	1.076s
Heap	0.0s	0.006s	0.040s	0.289s	0.512s	1.103s	1.759s	2.349s
Quick	0.0s	0.003s	0.018s	0.130s	0.214s	0.481s	0.717s	0.926s
Vetor decrescente								
	100	1000	5000	30000	50000	100000	150000	200000
Bubble	0.002s	0.155s	3.925s	3.953s	398.302s	indeterminado	indeterminado	indeterminado
Insertion	0.001s	0.102s	2.740s	2.738s	285.180s	indeterminado	indeterminado	indeterminado
Merge	0.0s	0.003s	0.021s	0.021s	0.248s	0.513s	0.814s	1.082s
Heap	0.0s	0.006s	0.033s	0.035s	0.468s	1.006s	1.612s	2.174s
Quick	0.001s	0.003s	0.017s	0.135s	0.219s	0.4611s	0.698s	0.973s

3.1. Condições de processamento:

- Python dentro do Pycharm;
- Windows 10;
- AMD Ryzen 7 1700;
- 2x8 GB de RAM 2800 MHz DDR4.

3.2. Conjuntos ordenados:



3.3. Conjuntos inversamente ordenados:

3.4. Conjuntos aleatórios:

4. Proposta de Algoritmo de Ordenação Híbrido (AOH)

Comparando os gráficos, conclui-se que o algorítmo merge sort tem os resultados mais rápidos que o heap sort. Em paralelo, o quick sort apresentou problemas de recursividade nos vetores crescentes e decrescentes. Dessa forma, foram escolhidos o Merge Sort, que tem um desempenho constantemente ótimo em todos os casos, e o Insertion Sort, por ser o algorítmo mais eficiente em vetores crescentes.

A principal vantagem do algorítmo híbrido é que ele explora a ordem existente nos dados para minimizar o número de comparações e trocas. Ele faz isso dividindo o array em pequenos subarrays, que já estão ordenados, e então mesclando esses subarrays usando um algoritmo de merge sort modificado. A vantagem de mesclar os subarrays ordenados em vez de mesclar sub-listas de tamanho fixo (como é feito pelo merge sort tradicional) é que isso diminui o número total de comparações necessárias para ordenar toda a lista.

Em resumo, o algoritmo híbrido combina as vantagens do Insertion Sort para arrays pequenos e do Merge Sort para arrays maiores, visando otimizar o desempenho da ordenação.

4.1. Resultados obtidos do AOH

Conjunto aleatório

Conjunto ordenado

Conjunto inversamente ordenado

4.2. Complexidade do AOH

Verifica-se que a complexidade do algorítmo híbrido é equivalente ao merge sort para o caso médio e pior caso O(n*log(n)). Além disso, nota-se que a complexidade para o melhor caso é O(n), devido a implementação do insertion sort.

5. Conclusão

A implementação e a comparação de algoritmos de ordenação são importantes para entender o desempenho e a eficiência de diferentes abordagens. Com base nos resultados da comparação, é possível escolher o algoritmo mais adequado para uma determinada situação, levando em consideração as características dos dados de entrada e os requisitos de desempenho.

6. Bibliografia

Referência:
Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2002). Algoritmos: Teoria e Prática. Editora Campus. Rio de Janeiro, Brasil.