Algoritmos de ordenação

Ícaro T. D. Rocha - 156.307,
Nicolas Pereira Novaes - 156.253
Victor Augusto - 156.620
Algoritmos e estrutura de dados 2 - Turma – I

Introdução

Este relatório tem como foco a análise e comparação de três algoritmos de ordenação: Bubble Sort, QuickSort e Counting Sort. O objetivo principal é avaliar o desempenho dos algoritmos escolhidos em diferentes cenários implementados em diferentes linguagens de programação para delimitação do melhor algoritmo em cada caso.

A investigação foi realizada para aprofundar e observar na prática as diferenças dos algoritmos de ordenação escolhidos previamente estudados em sala de aula em diferentes linguagens de programação. Além disso, a investigação possibilitou a solidificação do entendimento dos algoritmos abordados neste relatório.

No início da investigação já era conhecida a estrutura geral dos algoritmos e algumas características gerais, como vantagens, desvantagens e complexidade de tempo de cada algoritmo.

No caso do Bubble Sort, era conhecido que ele é adequado para pequenos conjuntos de dados, ou quando a lista já está parcialmente ordenada. Possuindo uma complexidade de tempo de O(n) para melhor caso, quando a lista já está ordenada, e $O(n^2)$ para quando a lista está na ordem inversa (CORMEN, 2002).

Para o QuickSort, o princípio de ordenação é por divisão e conquista, tendo como complexidade O(n log n) para o melhor caso, quando o pivô escolhido divide a lista aproximadamente pela metade. E $O(n^2)$ para o pior caso, quando a lista está desbalanceada, não ocorrendo a divisão pela metade pelo pivô. O Quicksort é amplamente utilizado em listas grandes e não pré-ordenadas, ordenação In-Place e quando a estabilidade não é algo necessário (CORMEN, 2002).

O Counting Sort é amplamente utilizado quando o intervalo das chaves usadas para ordenação for pequeno em comparação com o tamanho da lista, tendo com melhor caso O(n). No seu pior caso, quando os elementos têm um grande intervalo entre a menor e a maior chave de comparação, a complexidade pode chegar a O(n + k), sendo ‘n’ a quantidade de elementos e ‘k’ a diferença entre a maior e a menor chave de comparação (CORMEN, 2002).

Em relação às linguagens escolhidas para implementação dos códigos, é amplamente conhecido que o C++ é uma

linguagem de alto desempenho. Além disso, há uma tendência que os algoritmos de ordenação sejam mais rápidos nessa linguagem em comparação às outras linguagens escolhidas, Java e Python, devido a natureza de baixo nível do C++, resultando em um tempo de processamento mais rápido.

No tocante a linguagem Java deve apresentar um desempenho pior em relação ao C++, uma vez que é uma linguagem executada em uma máquina virtual, porém apesar dessa diferença, não deve ser tão significativa os resultados.

Por último, Python não é uma boa opção para ordenação de um grande volume de dados devido a diversos fatores. O primeiro desses fatores é por se tratar de uma linguagem interpretada, implicando em um desempenho inferior a linguagem compiladas como C++. Além disso, Python apresenta *overhead* de memória, em relação às outras linguagem, e algumas limitações de implementação, tornando, Python, uma opção ruim para ordenação de grande volume de dados.

Entretanto, não havia uma análise profunda que comparasse o desempenho entre esses algoritmos em cenários diferentes, motivando esta pesquisa.

Metodologia

Para a criação do ambiente de teste foram selecionados três tipos de dados: inteiros aleatórios, inteiros 95% ordenados e inteiros em ordem decrescente. Para cada um desses tipos foram criados 8 tipos de arquivos diferentes, sendo que a diferença entre eles está na quantidade de números no arquivo e no número máximo permitido em cada um. As quantidades são: 1000, 10000, 100000 e 500000, enquanto os números máximos em cada arquivo podem ser 10000 ou 10000000.

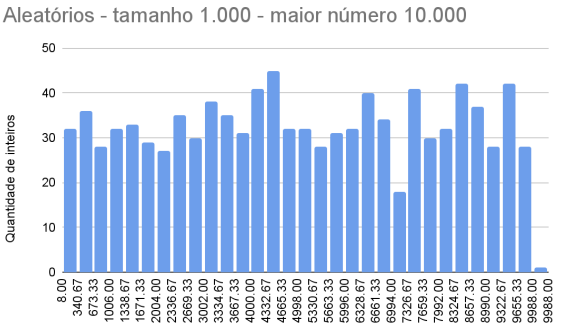

Figura 1 - Histograma para números aleatórios (AUTORES, 2023)

Os números aleatórios não apresentam uma ordem definida, como pode ser observado no histograma da figura 1. A média deste conjunto de números foi igual a 5051.974, que corresponde aproximadamente à média do maior número.

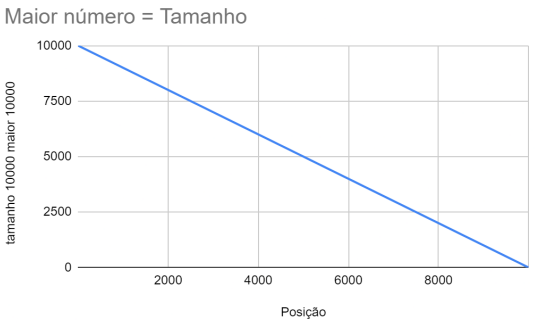

Figura 2 - Números decrescentes x posição quando o maior número é igual a quantidade de elementos da lista (AUTORES, 2023)

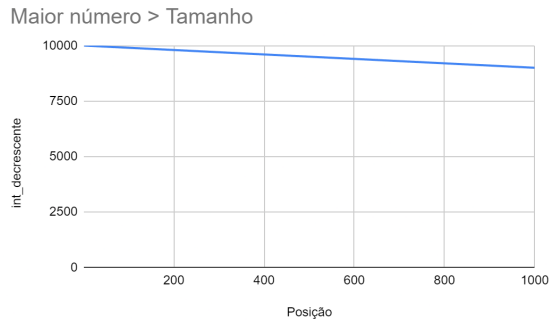

Figura 3 - Números decrescentes x posição quando o maior número é maior que a quantidade de elementos da lista (AUTORES, 2023)

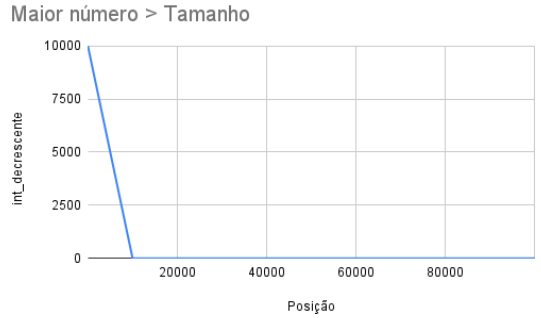

Figura 4 - Números decrescentes x posição quando o maior número é menor que a quantidade de elementos da lista (AUTORES, 2023)

Diferente dos números aleatórios, os decrescentes apresentam três comportamentos diferentes.

No primeiro deles, o maior número da lista é igual a quantidade elementos da lista, nele os números vão do maior elemento até 0 sem se repetirem.

No segundo caso, o maior número da lista é maior do que a quantidade total de elementos, deste modo, os números vão do maior número até o maior - quantidade de elementos.

No último caso, a quantidade de elementos excede o tamanho da lista, então a partir do elemento quantidade de elementos - maior número, os elementos foram preenchidos com 0.

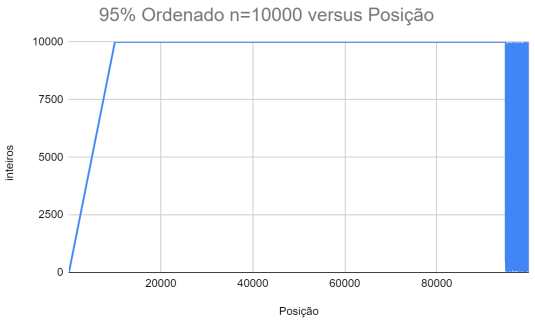

Figura 5 - Gráfico para números 95% ordenados x Posição para maior número < quantidade de elementos(AUTORES, 2023)

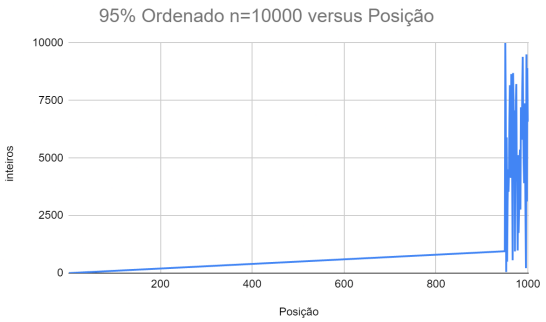

Figura 6 - Gráfico para números 95% ordenados x Posição para maior número > quantidade de elementos(AUTORES, 2023)

No último tipo de lista, os números apresentam dois comportamentos. Caso o maior número seja menor que a quantidade de alimentos da lista, ao alcançar o maior número possível, esse número irá se repetir até alcançar os 95% e ficar aleatório.

Já no segundo caso, os números apresentam ordem crescente a partir do número 0 e, ao alcançar os 95% o número impresso é o maior número possível, após esta porcentagem os números apresentam comportamento aleatório.

Os arquivos .txt citados anteriormente foram descritos anteriormente foram produzidos por um programa escrito em C que, ao se passar as informações necessárias, cria esses arquivos contendo um cabeçalho com o nome do arquivo na primeira linha e os dados nas linhas posteriores.

Já para ordenar os números desses arquivos foram escolhidos três métodos diferentes de ordenação: bubbleSort, quickSort e countingSort. Sendo que esses foram implementados em três linguagens de programação diferentes: Python, Java e C + +. Cada caso foi executado 3 vezes em todas as linguagens e o resultado utilizado foi a média destes valores.

A execução de cada linguagem ficou a cargo de cada integrante do grupo, sendo que, em testes realizados, o tempo de execução em diferentes computadores foi similar.

Ao executar os programas, os tempos de execução foram salvos em outros arquivos .txt chamados outputs e posteriormente tratados em planilhas.

Resultados e discussão

CountingSort:

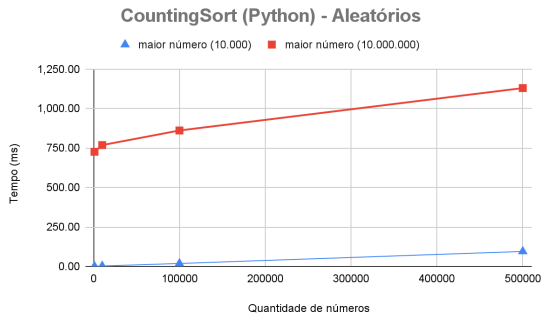

Figura 7 - Gráfico de linha para countingSort em python para números aleatórios (AUTORES, 2023)

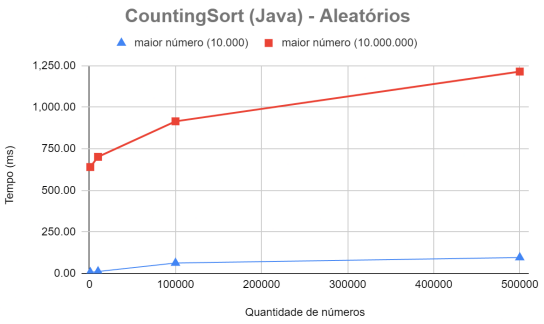

Figura 8 - Gráfico de linha para countingSort em java para números aleatórios (AUTORES, 2023)

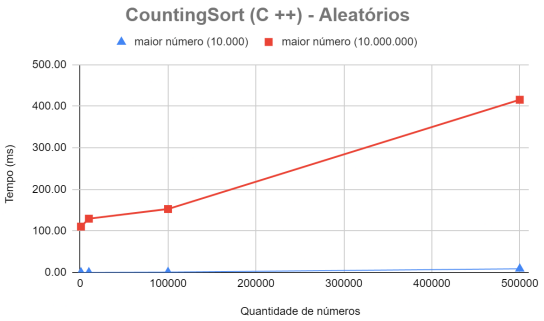

Figura 9 - Gráfico de linha para countingSort em c++ para números aleatórios (AUTORES, 2023)

Como pode ser observado na figuras 7, 8 e 9, o método countingSort apresenta um comportamento linear, também presente nos tipos decrescentes e quase ordenados.

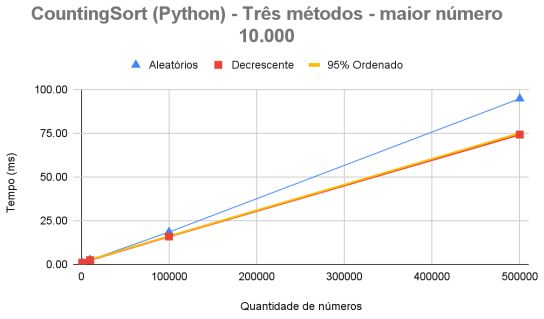

Figura 10 - Gráfico de linha para 3 tipos de arquivos utilizando countingSort em Python (AUTORES, 2023)

Ao comparar os três tipos de arquivos, percebemos que os aleatórios possuem um tempo de execução um pouco maior. Este comportamento também é notado em java e c++

BubbleSort:

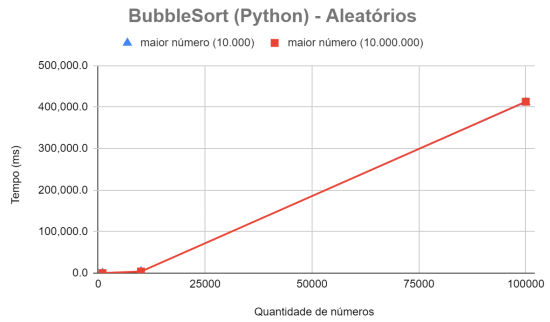

Figura 11 - Gráfico de linha para countingSort em python para números decrescentes (AUTORES, 2023)

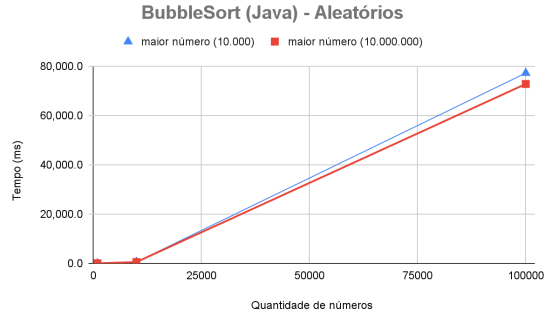

Figura 12 - Gráfico de linha para countingSort em java para números decrescentes (AUTORES, 2023)

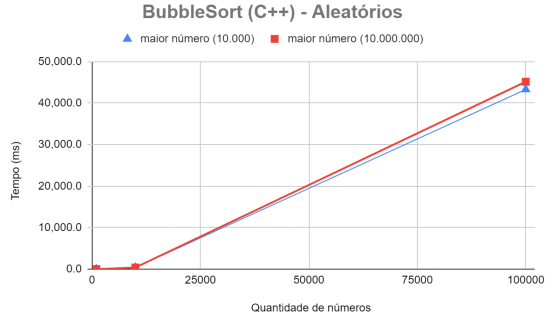

Figura 13 - Gráfico de linha para countingSort em python para números decrescentes (AUTORES, 2023)

Já na figura 11, 12 e 13 fica claro que o bubbleSort leva um intervalo de tempo consideravelmente maior que o countingSort para ser executado, sendo que, agora, ao aumentar a quantidade de tempo elementos, o tempo de execução aumenta ao quadrado.

No bubbleSort, o tamanho do maior número não possui grande influência sobre o tempo de execução.

As figuras não estão mostrando o valor de 500000 elementos, pois o tempo de execução impossibilitou que fosse realizado em todas as listas.

Neste algoritmo de ordenação o python leva uma desvantagem muito grande em relação aos demais, sendo que o tempo de execução foi quase 5 vezes maior.

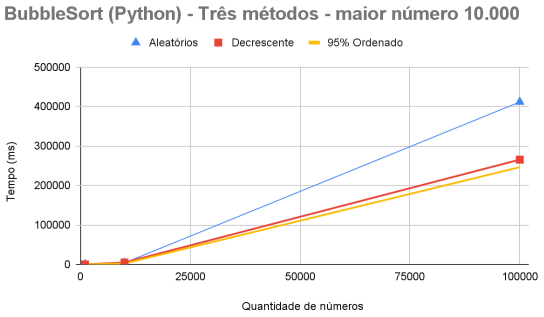

Figura 14 - Gráfico de linha para 3 tipos de arquivos utilizando countingSort em Python (AUTORES, 2023)

Ao se comparar os três tipos de arquivos fica claro que ao se ordenar números aleatórios, o tempo de execução foi quase 50% maior. O motivo desse fato está na natureza do bubbleSort, pois são necessárias uma quantidade maior de trocas entre elementos

adjacentes quando se tem uma lista não ordenada.

QuickSort :

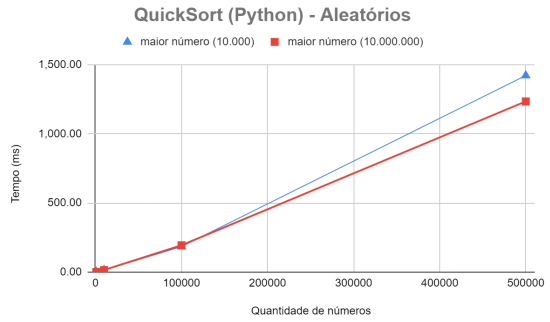

Figura 15 - Gráfico de linha para quickSort em python para números aleatórios (AUTORES, 2023)

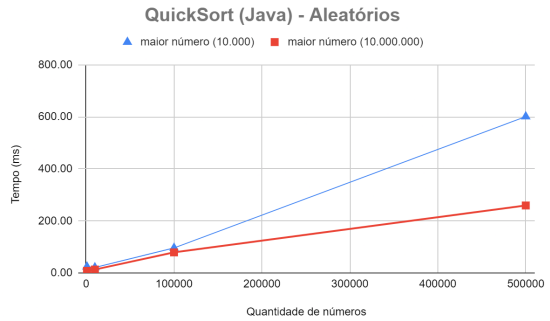

Figura 16 - Gráfico de linha para quickSort em java para números aleatórios (AUTORES, 2023)

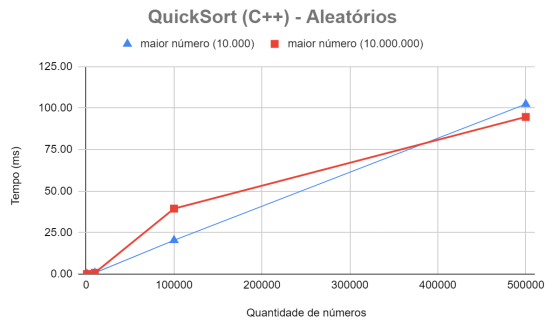

Figura 17 - Gráfico de linha para quickSort em C++ para números aleatórios (AUTORES, 2023)

Quando as figuras 15, 16 e 17 são comparadas, nota-se que elas apresentam algumas semelhanças entre si, porém com comportamentos diferentes quando o número era igual a 10.000.000.

quickSort (Python)		
	Decrescente	
Tamanho da lista	tempo(ms) n= 10^4	tempo(ms) n= 10^6
1000	39.42	40.24
10000	3,937.91	Seg. F
100000	Seg. F	Seg. F
500000	Seg. F	Seg. F
	95% ordenado	
Tamanho da lista	tempo(ms) n= 10^4	tempo(ms) n= 10^6
1000	39.4	40.2
10000	3,937.9	Seg. F
100000	Seg. F	Seg. F
500000	Seg. F	Seg. F

Tabela 1 - Tempos de execução do quickSort em python

O QuickSort foi o algoritmo que mais apresentou erros na hora da execução. Em python, o número de recursões é limitada a 1000 por padrão, podendo ser alterada com a biblioteca sys, mas ainda com esta ferramenta, nos piores casos de pivô (decrescente e quase ordenado), ao executar o programa ele exibe um erro chamado “segmentation fault”, o que pode ser visto na tabela 1.

quickSort (java)		
	Decrescente	
Tamanho da lista	tempo(ms) n= 10^4	tempo(ms) n= 10^6
1000	36.0	27.3
10000	534.0	377.3
100000	Stack Over.	Stack Over.
500000	Stack Over.	Stack Over.
	95% ordenado	
Tamanho da lista	tempo(ms) n= 10^4	tempo(ms) n= 10^6

1000	18.00	39.33
10000	56.00	477.33
100000	3,153.00	446.00
500000	Stack Over.	Stack Over.

Tabela 2 - Tempos de execução do quickSort em python

Quando rodados em java observou-se o erro de “Stack overflow”.

Os achados apresentados neste estudo significam que a eficiência e o tempo de execução dos algoritmos de ordenação variam significativamente dependendo da linguagem de programação em que são implementados. Além disso, essas diferenças podem ser influenciadas pela implementação específica de cada algoritmo na linguagem em questão. Aqui estão algumas conclusões específicas:

- Bubble Sort: É um algoritmo de ordenação simples, porém ineficiente, que apresenta um desempenho pior à medida que o tamanho da entrada aumenta. Ele é mais adequado para pequenas quantidades de dados. Os resultados podem ter mostrado que o Bubble Sort é o mais lento dos três algoritmos em todas as linguagens.
- Quick Sort: É um algoritmo de ordenação eficiente que geralmente supera o Bubble Sort em termos de tempo de execução, especialmente em grandes conjuntos de dados. No entanto, a implementação específica do Quick Sort e otimizações na linguagem podem influenciar seu desempenho. Pode ter sido observado que o Quick Sort tende a ser mais rápido que o Bubble Sort e o Counting Sort na maioria dos cenários.
- Counting Sort: É um algoritmo de ordenação linear que é altamente eficiente para classificar números inteiros dentro de um intervalo limitado. Sua eficácia depende da natureza dos dados. Se os dados estiverem dentro de um intervalo pequeno e conhecido, o Counting Sort pode superar tanto o Bubble Sort quanto o Quick Sort em termos de velocidade.

C++:

- Bubble Sort: Em C++, o Bubble Sort pode ser implementado de maneira mais eficiente do que em Java devido à ausência de sobrecarga de objeto e chamadas de método. No entanto, ainda é um algoritmo ineficiente para grandes conjuntos de dados em comparação com o Quick Sort e o Counting Sort.
- Quick Sort: C++ é conhecido por ser uma linguagem de baixo nível que permite uma implementação altamente eficiente do Quick Sort. O uso de ponteiros e manipulação direta de memória contribui para um desempenho rápido e eficaz.
- Counting Sort: A implementação do Counting Sort em C++ também pode ser muito eficiente devido à capacidade de manipular arrays

diretamente e ao controle preciso da memória. Ele é particularmente eficaz para ordenação de inteiros.

Python:

- Bubble Sort: Em Python, o Bubble Sort pode ser implementado de forma simples, mas tende a ser consideravelmente mais lento do que em C++ ou Java devido à interpretação em tempo de execução e à natureza dinâmica da linguagem. Portanto, o Bubble Sort em Python é geralmente o mais lento dos três.
- QuickSort: O QuickSort em Python pode ser menos eficiente do que em C++ ou Java devido à sobrecarga de objeto e à natureza interpretada da linguagem. No entanto, o Python possui bibliotecas otimizadas, que oferecem implementações eficientes do Quick Sort para arrays de números mas mesmo essas podem acabar se tornando ineficientes.
- Counting Sort: A implementação do Counting Sort em Python pode ser eficiente, especialmente para ordenação de inteiros em um intervalo limitado. A linguagem Python permite a criação fácil de dicionários ou listas para contar elementos, tornando o Counting Sort uma opção viável.

Java:

- Bubble Sort: Em Java, o Bubble Sort pode ser implementado de forma relativamente simples, mas tende a ser lento, especialmente para grandes conjuntos de dados. Isso ocorre devido ao custo de objetos e chamadas de método em Java, o que o torna menos eficiente em comparação com linguagens mais próximas do hardware.
- Quick Sort: Java é uma linguagem que permite a implementação eficiente do Quick Sort devido ao suporte a arrays. Portanto, o Quick Sort em Java geralmente apresenta um desempenho sólido em relação ao Bubble Sort, especialmente para conjuntos de dados maiores.
- Counting Sort: A implementação do Counting Sort em Java pode ser eficiente, especialmente para ordenação de inteiros em um intervalo limitado. No entanto, a necessidade de criar um array auxiliar grande o suficiente para acomodar todo o intervalo pode ser uma desvantagem em termos de consumo de memória.

Em resumo, a diferença na execução dos algoritmos nessas três linguagens está principalmente relacionada à eficiência das operações de manipulação de memória, sobrecarga de objeto e interpretação em tempo de execução. C++ geralmente oferece a melhor eficiência, enquanto Python tende a ser mais lento devido à sua natureza interpretada, mas ainda pode ser eficaz para tarefas específicas, especialmente com bibliotecas otimizadas.

Em geral, os achados são consistentes com o que se espera de cada algoritmo e como eles são afetados pelas linguagens de programação, Bubble Sort é conhecido por ser ineficiente e não é amplamente utilizado em cenários reais devido à sua baixa eficiência em grandes conjuntos de dados. Quick Sort é

amplamente reconhecido como um algoritmo eficiente de ordenação, embora a implementação e otimizações específicas possam variar. Counting Sort é conhecido por ser extremamente eficiente em certas situações, especialmente quando os dados estão dentro de um intervalo pequeno.

Dadas essas características de cada algoritmo, nossos testes confirmaram essa ideia. O Bubble Sort teve um tempo de execução consideravelmente maior com 500 mil números, devido à sua ineficiência.

O QuickSort mostrou-se muito eficiente na maioria dos casos, exceto quando os números estavam em ordem decrescente, devido à escolha do pivô estar no início.O Counting Sort foi, de longe, o mais eficiente para a quantidade de elementos que tínhamos. Não tivemos conjuntos de dados grandes o suficiente para que ele começasse a ficar atrás em relação ao Quick Sort."

O estudo adiciona informações sobre como o desempenho dos algoritmos de ordenação pode variar em diferentes linguagens de programação. Ele demonstra a importância de escolher o algoritmo de ordenação apropriado para o contexto específico, levando em consideração não apenas as características do algoritmo, mas também a linguagem em que ele será implementado.

Além disso, o estudo destaca a influência das otimizações de linguagem e técnicas de programação na eficiência dos algoritmos. Isso pode ser útil para desenvolvedores que desejam escrever código mais eficiente em uma linguagem específica.

No geral, o estudo contribui para uma compreensão mais profunda de como os algoritmos de ordenação se comportam em diferentes contextos e como as escolhas de implementação e linguagem podem afetar o desempenho.

Conclusão

Em síntese, o Counting Sort foi o que apresentou o melhor resultado nos dados usados para o *input*. O QuickSort poderia ter sido otimizado em cenários específicos, como na ordenação decrescente através da implementação de uma função para randomizar o pivô, evitando que ele permaneça fixa na primeira posição.

Em relação a linguagem de programação, a linguagem C++ foi a que apresentou melhor desempenho, algo já esperado por ser uma linguagem de alto desempenho e não mostrar problemas com sobrecarga de pilha, algo que prejudicou o desempenho em Python e Java. Python enfrentou desafios com grande número de recursões, resultando em falhas (segmentation fault), o que revelou ineficiência na ordenação de grande volume de dados. Para a linguagem Java, o algoritmo QuickSort atingiu a capacidade máxima de recursões, gerando falhas (StackOverflowError), demonstrando também uma ineficiência em lidar com um grande volume de dados.

Dessa forma, os resultados obtidos estão em conformidade com o conhecimento previamente documentado na literatura, reforçando a validade e a consistência das conclusões alcançadas neste estudo.

Referências

CORMEN, Thomas H; SOUZA, Vanderberg D; STEIN, Clifford; RIVEST, Ronald L; LEISERSON, Charles E. **Algoritmos: teoria e prática**. Rio de Janeiro: Campus, 2002.

FreeCodeCamp. **Algoritmos de Ordenação Explicados com Exemplos em Python, Java e C**. FreeCodeCamp. Disponível em: https://www.freecodecamp.org/portuguese/news/algoritmos-de-ordenacao-explicados-com-exemplos-em-python-java-e-c/. Acesso em: 24/09/2023.

ZIVIANI, Nivio. **Projeto de algoritmos: com implementações em PASCAL e C**. 3 ed. rev. e ampl. São Paulo: Heinle Cengage Learning, 2011. 639 p. ISBN 9788522110506.