Disciplina: Implementação Algorítmica
Curso: Engenharia de Software
Professor: Higa
Estudantes:
Maycon Felipe Mota - 2021.1906-069-7
Breno Ferreira Rodrigues - 2021.190.605-9

Trabalho 1

1. Introdução

A eficiência dos algoritmos de ordenação é um tema de pesquisa contínua. Neste estudo, conduzimos um conjunto de experimentos para comparar o desempenho de seis algoritmos de ordenação clássicos: *selection sor*t, *insertion sort*, *merge sort*, *heap sort*, *quick sort* e *counting sort.*

Para cada algoritmo, foram gerados diversos conjuntos de dados com diferentes características, variando o tamanho, a distribuição e a ordem dos elementos. O tempo de execução de cada algoritmo foi medido em cada conjunto de dados, permitindo uma análise estatística dos resultados.

Os resultados obtidos neste estudo fornecem insights valiosos sobre as vantagens e desvantagens de cada algoritmo em diferentes cenários. Essa informação é fundamental para a seleção do algoritmo de ordenação mais adequado em diversas aplicações.

1. Metodologia

Para avaliar o desempenho dos algoritmos de ordenação *selection sor*t, *insertion sort*, *merge sort*, *heap sort*, *quick sort* e *counting sort*, realizamos experimentos em diversos cenários. Foram gerados vetores de tamanhos variados com diferentes graus de ordenação: aleatória, crescente, decrescente e semi-ordenada (algoritmos que estão predominantemente em ordem crescente, mas com algumas pequenas perturbações para simular dados reais que raramente são perfeitamente ordenados.). Para cada configuração, o tempo de execução foi medido repetidamente, exceto para os vetores já ordenados, onde uma única execução foi suficiente, pois o tempo de execução é constante nesse caso.

A implementação em Python proporcionou flexibilidade para a configuração dos experimentos e a coleta dos dados. Um script automatizado, experiment.py, foi desenvolvido para gerar os vetores, aplicar os algoritmos de ordenação e medir o tempo de execução utilizando a função time.time() da biblioteca padrão do Python. O script também gera relatórios detalhados, facilitando a análise dos resultados.

O script experiment.py possui a seguinte estrutura: inicialmente o algoritmo questiona ao usuário quais parâmetros serão utilizados na análise, como: valor inicial *inc*, valor final *fim* e incremento *stp*. Em seguida, o algoritmo realiza a geração de vetores utilizado cada uma das estratégias citadas acima e aplica as estratégias de buscas mencionadas anteriormente.

Essa metodologia permitiu comparar os algoritmos em condições controladas e identificar o algoritmo mais eficiente para cada tipo de entrada

2. Resultados encontrados

Conforme a metodologia, para realizar as experimentações, foram utilizados os seguintes parâmetros:

- Valor inicial como 1000;
- Valor final como 50000;
- Incremento como 1000;

3.1 Vetores aleatórios (RANDOM)

Para a geração de valores aleatórios, foi desenvolvido um método na classe *GenerateVectors* que recebe três parâmetros, sendo: *inc* valor inicial, *fim* valor final do intervalo e *stp* que é o tamanho do vetor. Utilizando uma *list comprehension*, o método chama a função *randint(inc, fim)* para, repetidamente, gerar *stp* números aleatórios dentro do intervalo especificado. Ao final, ele gera um vetor com números aleatórios para ser ordenado.

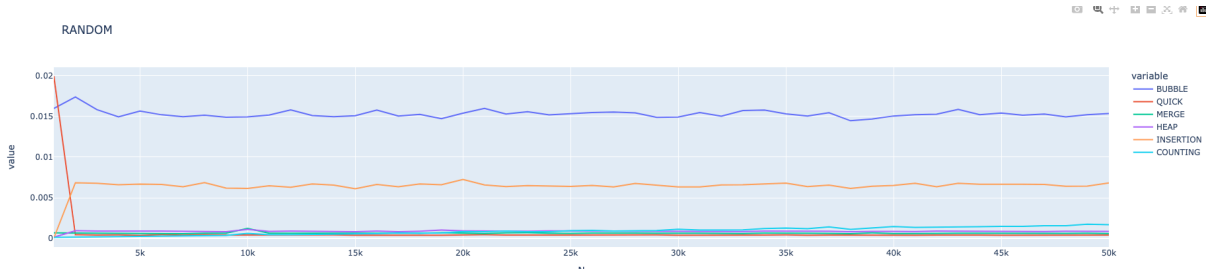

Figura 1. Comparativo do desempenho de gráficos durante a ordenação de um vetor aleatório. (Fonte: Autores, 2024).

O gráfico apresentado compara o desempenho de diversos algoritmos de ordenação em relação ao tamanho crescente do conjunto de dados. Observa-se que o **Bubble Sort demonstra ser o menos eficiente**, enquanto algoritmos como **Merge, Heap, Counting** e **Insertion apresentam tempos de execução significativamente menores.**

Contudo, utilizar vetores aleatórios, ou seja, gerar um vetor com 1000 números aleatórios, e ir replicando isso até chegar um vetor com 50000 números aleatórios não é suficiente. Há diversas justificativas, sendo elas: cenário não representativos: alguns algoritmos podem se comportar muito bem em vetores aleatórios, mas mal em outros cenários, como é o caso do Quick Sort: funciona bem com uma sequência aleatória, mas tende a ter seu desempenho prejudicado em vetores ordenado. Outra justificativa é a estabilidade: não é possível entender a capacidade do algoritmo de manter uma ordem de tempo, uma vez que dificilmente as entradas vão se repetir.

3.2 Vetores ordenados de maneira não crescente (REVERSE)

Similar ao método desenvolvido no 3.1, para gerar vetores ordenados decrescentemente, foi necessário adequar o método, utilizando *list comprehension* para executar a função *randint* com o objetivo de gerar um vetor ordenado crescente, no intervalo de 1000 a 50000, e ao final, interverter a posição, fazendo ficar decrescentemente.

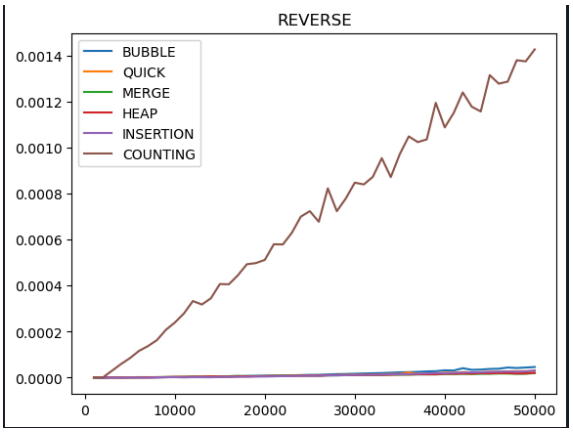

Figura 1. Comparativo do desempenho de gráficos durante a ordenação de um vetor ordenado de maneira não crescente (Fonte: Autores, 2024).

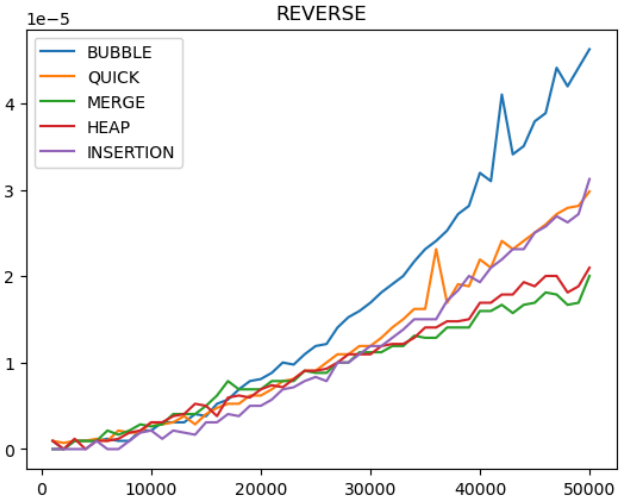

Figura 2. Comparativo do desempenho de gráficos durante a ordenação de um vetor ordenado de maneira não crescente sem utilizar o Counting Sort. (Fonte: Autores, 2024).

O comportamento do Counting Sort. O Counting Sort é um algoritmo de ordenação projetado especificamente para ordenar elementos inteiros dentro de um intervalo conhecido. Sua eficiência reside na criação de um array auxiliar para contar a frequência de cada elemento no conjunto de dados original. Essa contagem é então utilizada para posicionar os elementos diretamente em suas posições finais no array ordenado. Contudo, o Counting Sort se destaca em alguns casos, como por exemplo:

Melhor caso: Quando os elementos a serem ordenados estão distribuídos uniformemente dentro de um intervalo pequeno, o Counting Sort atinge sua complexidade de tempo linear O(n+k), onde n é o tamanho do conjunto de dados e k é o intervalo dos valores. Isso o torna extremamente eficiente nesses casos.
Pior caso: O desempenho do Counting Sort se degrada quando o intervalo dos valores é muito grande em relação ao tamanho do conjunto de dados. Isso ocorre porque o array auxiliar precisa ser

dimensionado para acomodar todos os possíveis valores, mesmo que muitos deles não estejam presentes no conjunto de dados.

Nesse caso, o intervalo dos valores utilizados nos dados de teste pode não ser ideal para o Counting Sort. Um intervalo muito grande pode ter impactado negativamente o desempenho do algoritmo.

Isolando o Counting Sort. Ao isolarmos o Counting Sort, podemos identificar o rápido desempenho do Merge Sort, enquanto tivemos o pior desempenho do Bubble Sort. Para o Bubble Sort, um conjunto de dados totalmente ordenado de forma inversa representa o pior caso possível. Isso ocorre porque o algoritmo precisa realizar o número máximo de comparações e trocas para colocar os elementos na ordem correta. A cada iteração, ele compara apenas elementos adjacentes, movendo o maior elemento para o final da lista lentamente. Enquanto que Merge Sort funciona dividindo o problema em subproblemas menores, ordenando cada subproblema recursivamente e, em seguida, mesclando os subproblemas ordenados. Outros algoritmos, como Quick Sort e Heap Sort, podem apresentar um bom desempenho em dados reversamente ordenados, mas suas eficiências podem variar dependendo de fatores como a escolha do pivô e a implementação.

3.3 Vetores ordenados de maneira crescente (ORDERED)

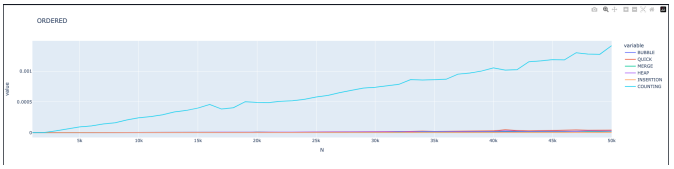

Figura 1. Comparativo do desempenho de gráficos durante a ordenação de um vetor ordenado de maneira crescente. (Fonte: Autores, 2024).

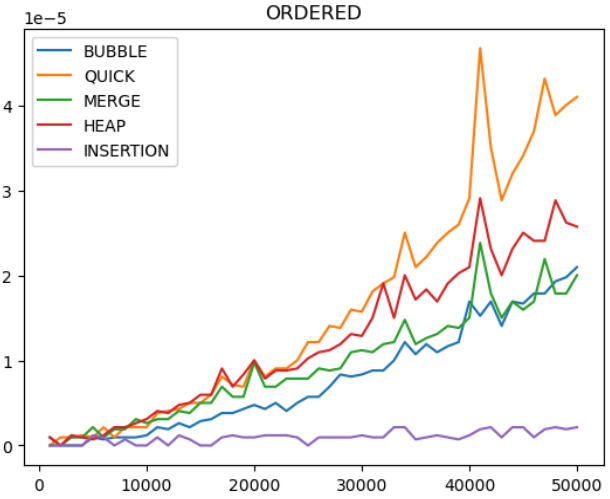

Figura 2. Comparativo do desempenho de gráficos durante a ordenação de um vetor ordenado de maneira crescente. (Fonte: Autores, 2024).

No caso de vetores ordenados de maneira crescente, o Counting Sort obteve o mesmo comportamento, devido a característica do conjunto de dados mencionado em 3.2. Contudo, ao

olharmos os outros algoritmos, identificamos o **Bubble Sort** tendo o melhor desempenho, uma vez que o algoritmo realiza apenas uma passagem pelos dados, comparando elementos adjacentes e constatando que não há necessidade de trocas, atingindo a complexidade de tempo linear O(n), o que torna bastante eficiente para os dados já ordenados.

O algoritmo **Insertion Sort** também consegue ser extremamente eficiente, uma vez que cada elemento é comparado com o elemento anterior e inserido em sua posição correta, sendo necessário fazer poucas comparações ou trocas, ficando com sua complexidade linear O(n).

Já no caso do Merge Sort, o desempenho permanece constante, uma vez que a ordem dos dados não importa para a execução do algoritmo.

Quick Sort e Heap Sort são afetados negativamente dado a ordenação do vetor. Heap Sort, requer que a construção do heap máximo (ou mínimo) tenha um número significativo de comparações e trocas. Isso porque cada elemento precisa ser "subido" na árvore do heap até encontrar sua posição correta. Já o Quick Sort, é prejudicado devido a escolha do pivô em dados ordenados que leva a partições desbalanceadas, resultando em um grande número de comparações e trocas desnecessárias.

3.4 Vetores com 10% dos elementos embaralhados (ALMOST ORDERED)

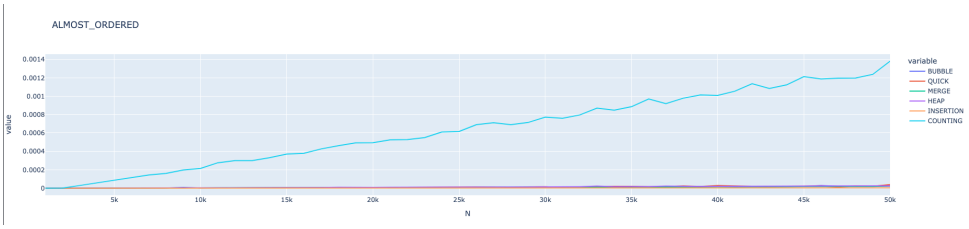

Figura 1. Comparativo do desempenho de gráficos durante a ordenação de um vetor ordenado de maneira embaralhada. (Fonte: Autores, 2024).

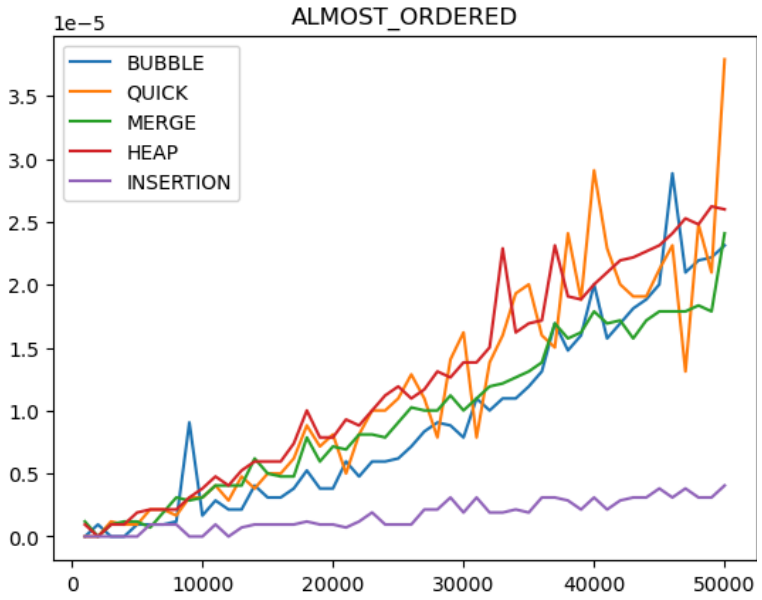

Figura 2. Comparativo do desempenho de gráficos durante a ordenação de um vetor oordenado de maneira embaralhada. (Fonte: Autores, 2024).

Semelhante ao 3.1, 3.2 e 3.3, o Counting Sort também tem comportamento similar devido a característica dos dados. Isolando ele, podemos perceber o excelente comportamento do Bubble Sort e Insertion Sort, ambos os algoritmos demonstram um desempenho relativamente bom em vetores quase ordenados. Isso ocorre porque eles são eficientes em lidar com pequenas inversões na ordem dos elementos.

Algoritmos como Insertion Sort e Bubble Sort tendem a se sair melhor, enquanto o desempenho do Quick Sort pode variar dependendo da escolha do pivô e da distribuição dos elementos. O Merge Sort mantém um desempenho consistente em diferentes tipos de dados, incluindo vetores quase ordenados.