Trabalho 2 - Ordenação - CIX56

Vitor Faria Medeiros da Silveira / GRR20232342

Bacharelado em Ciência da Computação / UFPR

1. Resumo

Esse relatório tem como objetivo demonstrar e analisar os testes feitos com os algoritmos Merge Sort, Quick Sort, Heap Sort e Counting Sort, desenvolvidos nas aulas de Algoritmos e Estruturas de Dados 2. Além também do Tim Sort, que foi elaborado através de pesquisas de minha parte.

2. Algoritmos de Ordenação

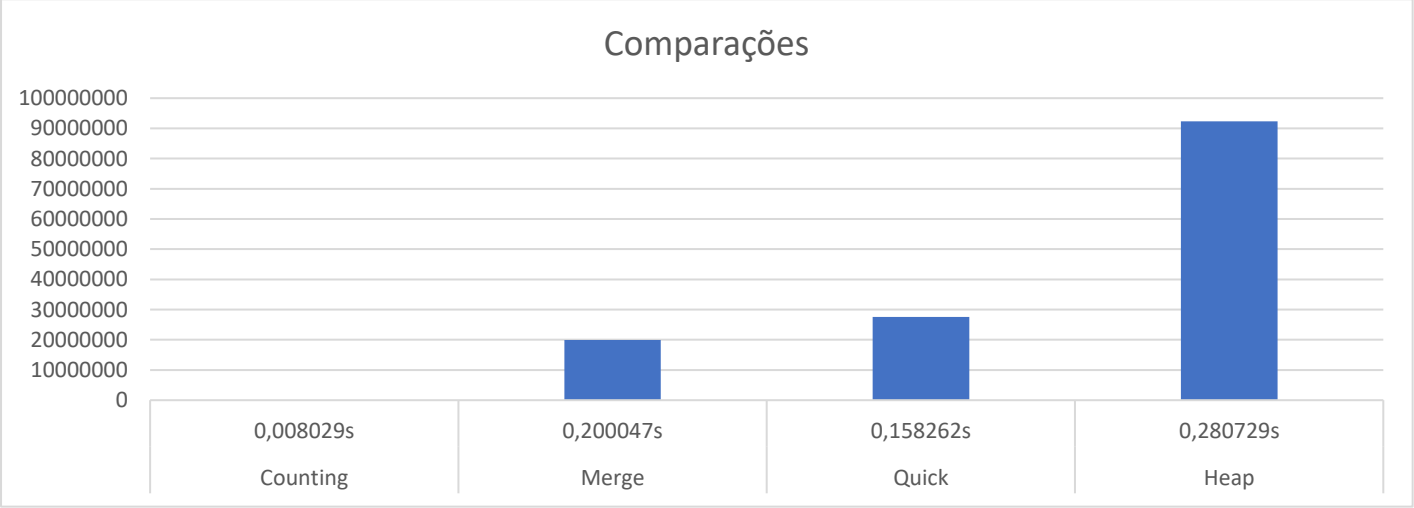
1º Teste: Vetor com 1.000.000 posições preenchidas com elementos aleatórios de range (0-100.000)

Foi feito um teste com os parâmetros acima utilizando os 5 algoritmos, como via de obter uma visão e estatística geral sobre esses algoritmos, e os resultados foram:

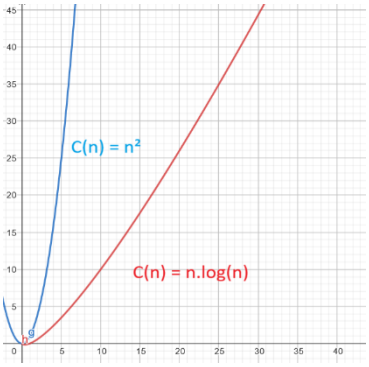
Counting Sort: 0,008s; **Quick Sort:** 0,16s e 27.517.986 comparações;

Heap Sort: 0,28s e 92.333.068 comparações; **Merge Sort:** 0,20s e 19.951.424 comparações.

Tim Sort: 23,74s e ~16.000.000.000 comparações.



(Gráfico comparação tempo x comparações, excluindo Tim Sort pelos seus dados serem discrepantes)



	Tempo	Comparações
Counting	0,008029s	0
Merge	0,200047s	19.951.424
Quick	0,158262s	27.517.986
Heap	0,280729s	92.333.068
Tim	23,749362s	15.649.361.180

(Comparação do custo n² (Tim Sort) e n.log(n) no gráfico) // (Tabelas de tempos e comparações obtidos no teste)

Comentário: No 1° teste realizado, foi possível notar a tamanha eficiência que os algoritmos analisados trazem. Quase todos (com exceção do Tim Sort, que será comentado depois) demonstraram desempenhos absurdos, ordenando o vetor de 1.000.000 de posições em menos de 1 segundo. Entretanto, esse teste apenas não foi o suficiente para concluir as peculiaridades de cada algoritmo, já que apresentaram resultados similares e o teste foi ordinário.

2° Teste: A partir do 2° teste, busquei encontrar o ponto fraco de cada algoritmo mencionado anteriormente. Como via de comparação, decidi usar vetores do mesmo tamanho de antes (1 milhão de posições). Começando com o Counting Sort, depois de algumas pesquisas e testes, percebi que seu ponto fraco está localizado no universo em que o seu range máximo de elementos do vetor é consideravelmente maior que o tamanho do vetor. Por exemplo, a baixo, vetor de 1.000.000 de posições com elementos de (0-10.000.000.000):

```
Tempo mergeSort: 0.151291s / num comparacoes: 25525776
Tempo quickSort: 0.204101s / num comparacoes: 19951424
Tempo heapSort: 0.287704s / num comparacoes: 92339048
Tempo countingSort : 6.060400s
Tempo TimSort: 26.762661s / num comparacoes: 15649348594
```

Comentário: Percebe-se então que quando aumentado o range dos elementos do vetor, o Counting Sort vai perdendo sua eficiência gradativamente. No primeiro teste, ele era de longe o mais eficiente dos 5 algoritmos. Nesse, os outros se mantiveram com um desempenho similar e o Counting Sort piorou drasticamente. Isso se leva ao fato de que o Counting Sort cria um vetor auxiliar com tamanho = ao número do seu maior elemento contido.

3° Teste: Para o terceiro teste, decidi analisar o ponto fraco do Quick Sort. Como visto no cálculo de recursões em sala, o Quick Sort entra em seu pior caso quando o vetor a ser ordenado já está previamente ordenado, de ordem crescente ou decrescente. Durante o teste, essa característica foi comprovada, resultando nas seguintes estatísticas com um vetor de 130.000 de posições ordena das:

```
Tempo quickSort: 31.372675s / num comparacoes: 8449935000
Tempo mergeSort: 0.012819s / num comparacoes: 2208928
Tempo heapSort: 0.032912s / num comparacoes: 10656834
Tempo countingSort : 0.001355s
Tempo TimSort: 0.479314s / num comparacoes: 264964625
```

Comentário: Foi possível analisar então que de fato o Quick Sort se torna inviável em seu pior caso, já que adquire um desempenho muito a baixo da média levando muito mais tempo para realizar a tarefa de ordenação.

Observação: Para os algoritmos Merge Sort e Heap Sort, não consegui encontrar um caso de ponto baixo, já que eles apresentaram um desempenho constante ao longo de todos os testes realizados.

3. Tim Sort

Após as pesquisas feitas, existem alguns comentários pertinentes sobre o algoritmo extra escolhido para residir no trabalho, sendo esse o Tim Sort. Dentre as curiosidades, estão:

- O Tim Sort foi criado por Tim Peters, um dos maiores contribuidores do desenvolvimento da linguagem Python, e é utilizado como algoritmo de ordenação padrão desta linguagem.
- O Tim Sort tem como principal característica interessante o fato de que foi desenvolvido para ser eficiente em dados comuns na vida real. Ou seja, apesar de não ser tão eficaz em vetores incomuns como aqueles utilizados nos testes deste trabalho, em teoria, ele apresenta um desempenho acima da média para vetores que contém valores comuns ao dia a dia.
- O Tim Sort é um algoritmo de ordenação hibrido, isso se deve ao fato de que usa tanto o Merge Sort quanto o Insertion Sort em suas computações.

4. Conclusão

A partir dos variados testes realizados, foi possível tirar algumas conclusões, dentre elas:

- Foi possível notar que apesar de EXTREMAMENTE EFICIENTE, o Counting Sort tem muitas desvantagens que reduzem sua capacidade de uso, sendo elas: A interferência que os items do vetor causam; A capacidade de ordenar apenas valores inteiros; A capacidade de ordenar apenas valores positivos.
- Foi possível ver na prática o impacto que o melhor / pior caso de um algoritmo proporciona para um programa. Como exemplo, ao realizar os testes em um vetor ordenado, o QuickSort não conseguiu ordenar vetores maiores do que 130.000 posições, mostrando que seu pior caso tem uma eficiência menor do que de todos os outros algoritmos testados.
- A eficiência de um algoritmo está intrinsicamente relacionada com a forma que é implementado e para que é implementado, como no caso do Tim Sort. Apesar de em meus testes este algoritmo não ter se mostrado muito eficiente, em sua versão elaborada ele é relatado como tendo uma utilidade bem acima da média, tanto que é utilizado como algoritmo padrão de uma das linguagens de programação mais utilizadas no mundo, o Python.
- Nem todo algoritmo possui um "pior caso", como visto nos exemplos do Merge Sort e Heap Sort.

6. Referências

1. https://deinfo.uepg.br/~alunoso/2019/AEP/TIMSORT/REA-TimSort.htm