UNIVERSIDADE DE SÃO PAULO

INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

JOÃO VICTOR BRECHES ALVES

13677142

FRANCYÉLIO DE JESUS CAMPOS LIMA

13676537

FILIPE SANTOS LOPES

13734409

Ordenação e complexidade de algoritmos

São Carlos

2023

INTRODUÇÃO

A partir das implementações dos diversos algoritmos de ordenação, é possível realizar diferentes tipos de análises a respeito dos códigos, sejam de caráter técnico ou não. Ambos os tipos de análise serão abordados neste relatório, apresentando desde opiniões pessoais dos alunos sobre a funcionalidade dos códigos, até observações relacionadas à complexidade técnica do algoritmos.

Diferentes tipos de dados serão apresentados para reforçar as as conclusões obtidas ao longo do relatório.

OBSERVAÇÕES GERAIS

A fim de ampliar a abrangência do código, no que diz respeito ao entendimento por parte de outros leitores, boa parte dos códigos se encontra em inglês, especialmente os comentários.

Foi utilizada a ferramenta de alocação dinâmica para para toda criação de TAD, sendo priorizada a criação de listas pela sua maior facilidade de manuseio.

Algumas bibliotecas menos convencionais, como *boollib.h,* foram utilizadas para facilitar a implementação do código ou para deixá-lo mais intuitivo.

Detalhes mais específicos de cada implementação serão abordados na análise específica de cada tipo de algoritmo.

1 - BUBBLE SORT

Bubble sort é, possivelmente, o mais simples dentre os códigos de ordenação. Isso é consequência do seu caráter bastante intuitivo, uma vez que constitui um método de ordenação extremamente direto e com pouca sofisticação. Em contrapartida, demanda tempos de execução bem maiores e, a depender do caso, bastante demorados.

O método utiliza a troca de posição entre valores em sequência para fazer a ordenação, troca que é feita baseada em passar os menores valores para a esquerda e os maiores para a direita (ou inverso, caso esteja ordenando de forma decrescente).

Por percorrer todo o array utilizando dois loops, o Bubble Sort tem uma complexidade de $O(n^2)$ no pior caso e no caso médio. No pior caso, o array está completamente desordenado e o algoritmo precisará fazer n-1 comparações no primeiro loop e n-2 no segundo, totalizando (n-1) * (n-2) comparações, ou seja, $O(n^2)$.

Entretanto é possível otimizar esse código, de maneira que, conferindo se houve trocas na iteração anterior podemos mandar ele continuar ou parar, fazendo com que no melhor caso sua complexidade seja linear (O(n)).

As tabelas a seguir nos mostra os dados obtidos para o Bubble Sort comum, sem otimização, para os 3 casos (lista aleatória, lista crescente e lista decrescente, respectivamente):

ENTRADAS	**1.000**	**10.000**	**100.000**
TEMPO	0.0000669000	0.0065020001	1.3911503553

lista aleatória (não otimizado)

ENTRADAS	**1.000**	**10.000**	**100.000**
TEMPO	0.0000419000	0.0039419001	0.3864744902

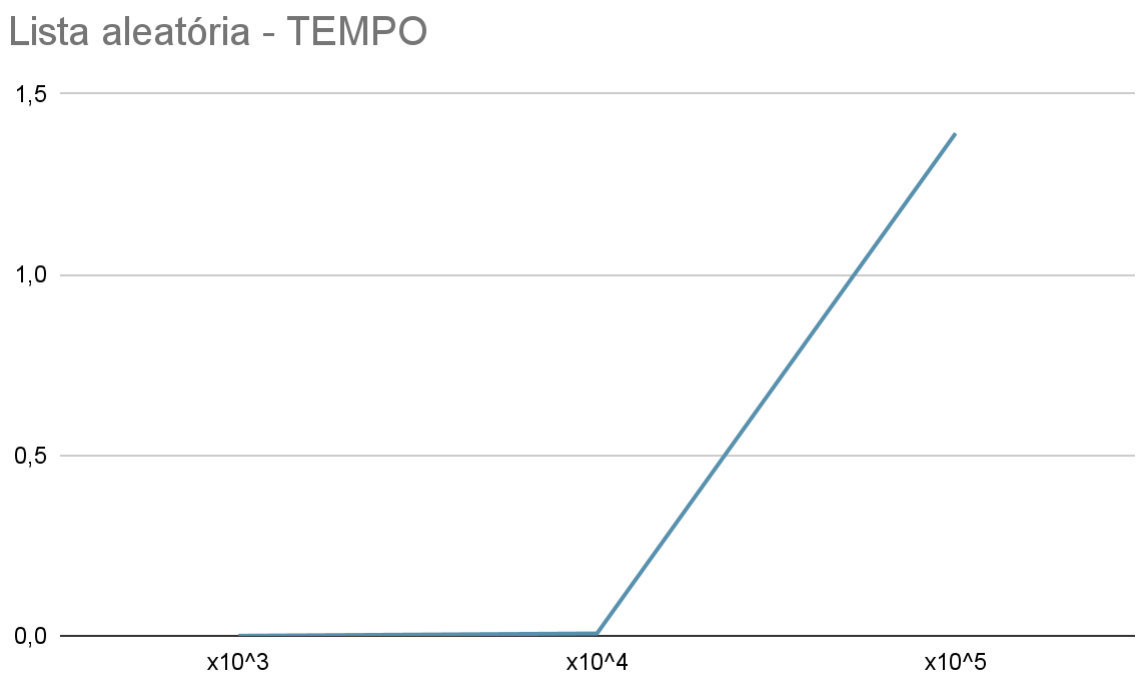
lista crescente (não otimizado)

ENTRADAS	**1.000**	**10.000**	**100.000**
TEMPO	0.0000528000	0.0052725999	0.4458710849

lista decrescente (não otimizado)

A partir dos gráficos acima, é notável um grande crescimento do tempo, dado pela complexidade já citada antes de $O(n^2)$.

Tal crescimento pode ser melhor notado no gráfico abaixo, que plota os valores da primeira tabela:



Apesar desse grande crescimento, bubble sort se mantém viável em caso de pequenas entradas. Sua facilidade de aplicação pode ser muito útil numa situação onde não irá utilizar números muito grandes de entradas.

Óbvio que essa viabilidade se dá mais para o caso do uso do bubble sort otimizado, que apesar de ainda ter uma complexidade quadrática, podemos dar a "sorte" de cair no melhor caso, onde se torna linear.

ENTRADAS	**1.000**	**10.000**	**100.000**
TEMPO	0.0000443000	0.0055239000	1.2554682493

lista aleatória

ENTRADAS	**1.000**	**10.000**	**100.000**
TEMPO	0.0000241000	0.0030532000	0.3138917983

lista crescente

ENTRADAS	1.000	10.000	100.000
TEMPO	0.0000274000	0.0058005000	0.4291703105

lista decrescente

Os dados acima nos mostram o tempo gasto no Bubble Sort otimizado.

2 - QUICKSORT

QuickSort é um algoritmo de ordenação mais complexo, entretanto, consideravelmente mais eficiente que o Bubble Sort citado anteriormente.

Nós chamamos o método utilizado pelo Quicksort de “método de divisão e conquista”, ou seja, ele divide o vetor original quantas vezes forem necessárias para se encontrar vetores menores e mais fáceis de se trabalhar

A complexidade do Quicksort é dada por O(n log n) no melhor caso e no caso médio, que ocorrem quando o vetor a ser ordenado é perfeitamente dividido em duas partes iguais (melhor caso) ou quando ele é dividido em duas partes não iguais, entretanto, equilibradas entre si.

Porém, mesmo com esse bom desempenho no caso médio e melhor caso, se considerarmos o pior teremos uma complexidade de $O(n^2)$, que ocorre quando o pivô escolhido é o primeiro ou o último elemento do vetor.

Mas apesar do pior caso, Quicksort ainda é o algoritmo de ordenação mais viável na maioria das situações por causa do seu pequeníssimo uso de memória e resultados extremamente satisfatórios na maioria dos casos.

Abaixo temos três tabelas que mostram o tempo gasto com o uso de Quicksort para ordenação de vetores aleatórios, crescentes e decrescentes, respectivamente.

ENTRADAS	1.000	10.000	100.000
TEMPO	0.0001006000	0.0104216998	1.0183081627

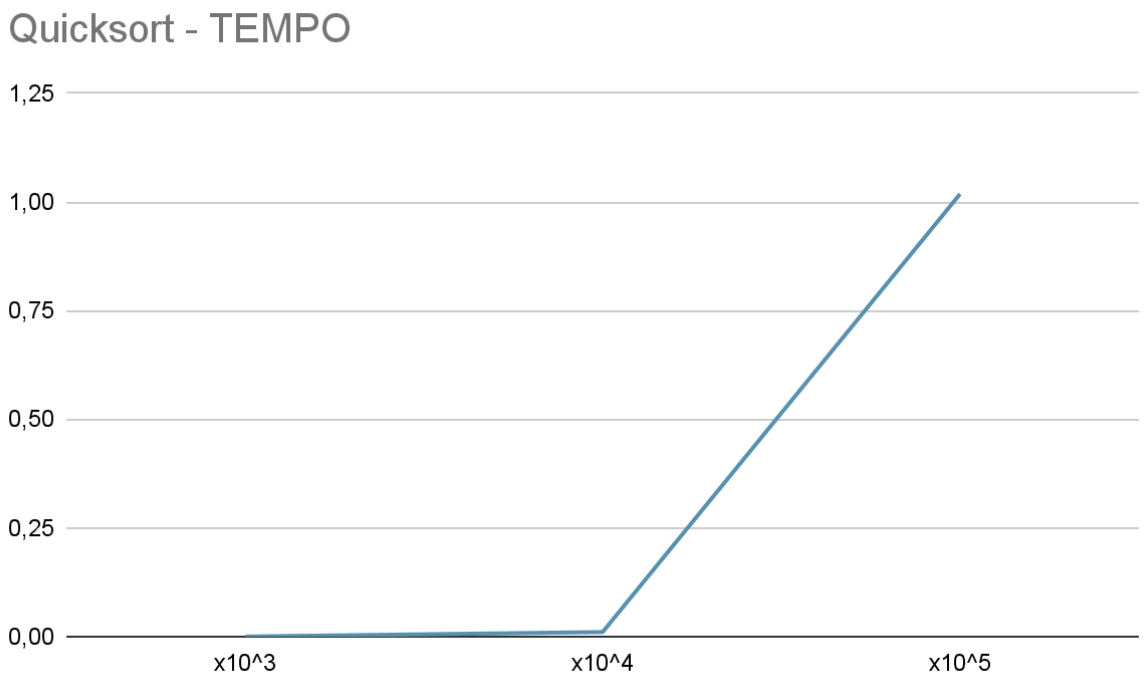
lista aleatória

ENTRADAS	1.000	10.000	100.000
TEMPO	0.0000000000	0.0000001000	0.0000001000

lista crescente

ENTRADAS	1.000	10.000	100.000
TEMPO	0.0000947000	0.0102977995	1.0256780386

lista decrescente



Acima podemos ver o gráfico que representa o tempo gasto pelo Quicksort na ordenação do vetor de números aleatórios.

3 - RADIX SORT

4 - HEAPSORT

CONCLUSÃO

Os quatro algoritmos apresentados tem seus pontos fortes e fracos. Por conta disso é impossível fazer uma comparação direta e simplesmente afirmar que algoritmo *"x"* é melhor que *"y"* e *"z"*, isso apenas a situação na qual será aplicada o algoritmo de ordenação pode nos responder.

Em âmbito geral, Quicksort é o mais comum por ser extremamente eficiente na maioria dos casos. O Heapsort, por conta usar ainda menos memória que o Quicksort, se torna mais viável quando a memória é uma limitação do sistema.

Radix é ainda mais situacional, muito usado para classificação de strings, por conta da sua complexidade (O(d*(n+b))), que considera o números de dígitos no elemento a ser classificado, o que faz que o mesmo seja usado em tarefas, como já dito, de classificação de strings de diferentes comprimentos.

Sendo assim, deve-se analisar a situação do seu problema para escolher o melhor algoritmo de ordenação.