AVALIAÇAO DE PERFORMANCE DE ALGORÍTMOS DE ORDENAÇÃO - C++ 💻

Mestrado em Modelagem Computacional - UFF/ EEIMVR - 20/04/2024

Nomes: Cleber Barros / Italo Galon github: https://github.com/cleberbarros1/mcct-perf-analysis-sort-algorithms

Objetivo deste projeto é executar e cronometrar algoritmos para ordenação de vetores nas linguagens C/C++, obter os tempos de ordenação para as amostragem e comparar as complexidades de cada algoritmo. 🕒

Motivação do Estudo:
--> Muitos problemas computacionais do mundo atual consistem em avaliar, ordenar e/ou classificar grandes massas de informações.
Problemas como pré-processar, segmentar, classificar, comuns do segmento de **Visão Computacional**, são exemplos de aplicações que necessitam essencialmente de algoritmos poderosos e que consigam produzir respostas em espaço de tempo relativamente pequenos. Naturalmente, para problemas de **Visão Computacional** utiliza-se matrizes M[m][n].
Neste estudo, trataremos de objetos de dimensão menor, os Vetores V[x], mas cujo o propósito é similar ao de matrizes: guardar informações.

Análise Teórica de complexidade O(n), juntamente com a análise de tempos de execução de algoritmos.

palavras-chave:
EFICIÊNCIA / COMPLEXIDADE / PERFORMANCE / C / C++

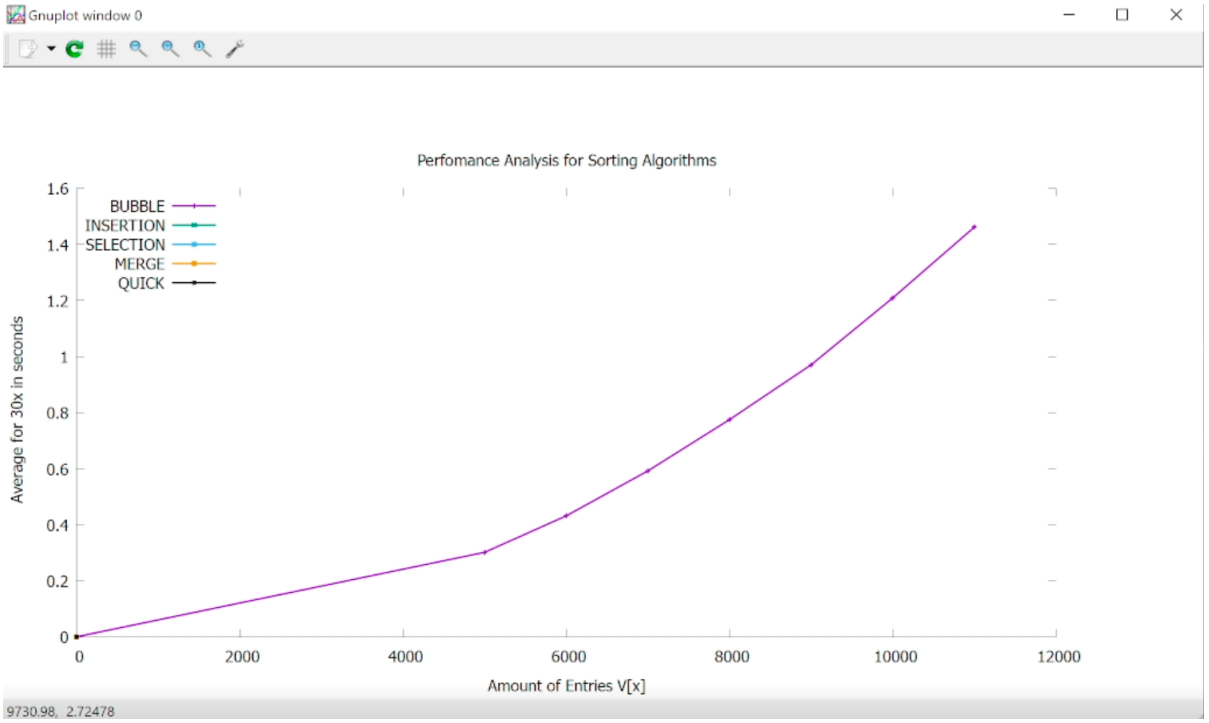

Figura 1 - Figura animada mostrando em tempo real a obtenção dos resultados para cada método

Parâmetros:

---> Vetores com V[5000] ~ V[20000];

---> Cada vetor é criado com entradas aleatórias, utilizando como apoio a biblioteca padrão da biblioteca C/C++.

---> x30 vezes a execução de cada ordenação para obter medias de tempo

---> Os tempos são obtidos com o apoio da Biblioteca , sendo iniciado uma marcação de tempo antes do processo de ordenação, após o processo e então calculada a diferença entre os tempos para obter o tempo final de execução do processo.

---> Execução de algoritmos de ordenação:
• Bubble Sort
• Insertion Sort
• Selection Sort
• Merge Sort
• Quick Sort

---> Cada método de ordenação é validado pelo programa adicional de testes **TesteUnitarioDosMetodos.cpp** criado para este fim.

Instruções para futuros usuários do software:

---> Utilizado software gratuito para plotar gráficos GNUPLOT

---> Os gráficos são gerados e atualizados em tempo real de forma automática pelo programa, através de um Pipe criado para utilizar os recursos disponibilizados pelo GNUPLOT.

---> É necessário possuir o GNUPLOT instalado e adicionado como "variável de ambiente" para permitir acesso aos recursos do software via Pipe dentro do programa principal.

---> Os resultados das médias para cada método em cada configuração de teste é armazenado e identificado pelos arquivos no diretório "relatórios-de-testes".

---> Ao finalizar uma bateria de testes, crie uma cópia dos relatórios gerados para garantir que os mesmos não sejam perdidos ao se executar uma nova análise.

---> O programa principal a ser compilado se encontra no arquivo **CronometroPerformance.cpp**

---> Os parâmetros iniciais como devem ser configurados diretamente no código fonte principal **CronometroPerformance.cpp** através das seguintes variáveis declaradas:

```
37    ///AQUI É A INVOCACAO DOs METODOs PRINCIPAIS. ALTERE NA ORDEM ABAIXO
38    /// -> valor inicial (1000)
39    /// -> Valor Final (2000)
40    /// -> Passo (100)
41    /// -> Repeticoes (30)
42    /// -> Qual Metodo deseja executar - "bubble" / "Insertion"
43
44    int inicio = 5000;
45    int final = 20000;
46    int passo = 1000;
47    int qtdTestes = 30;
```

Figura 2 - Figura ilustrativa para configuração dos parâmetros de avaliação das baterias de testes.

---> Os testes foram rodados em um computador com processador de 4GHz e 32GB de memória.

RESULTADOS

A seguir os resultados do experimento:

Médias de tempo obtidas ao executar x30 a operação de ordenação para tamanhos de vetores:

Vetor V[X]	Bubble (s)	Insertion (s)	Selection (s)	Merge (s)	Quick (s)
5000	0.301503	0.076705	0.139803	0.0254888	0.00855007
6000	0.431632	0.1109	0.204562	0.0284656	0.0114516
7000	0.59201	0.153208	0.276488	0.0328481	0.0151287
8000	0.77493	0.19752	0.360183	0.0360075	0.0176446
9000	0.970366	0.247786	0.451369	0.0422448	0.0234544
10000	1.20921	0.307304	0.567066	0.0472601	0.0266327
11000	1.46172	0.37517	0.693931	0.0516884	0.0306901
12000	1.73793	0.441935	0.8079	0.0577216	0.0356233
13000	2.03334	0.52244	0.947553	0.0611123	0.0398861
14000	2.3498	0.608143	1.10294	0.0670111	0.0502336
15000	2.6992	0.691989	1.26146	0.0695877	0.0562454
16000	3.04141	0.789506	1.4391	0.0783118	0.0650946
17000	3.41981	0.887748	1.62199	0.0811971	0.0631295
18000	3.83621	1.00276	1.81682	0.0863501	0.0687258
19000	4.25198	1.11069	2.05156	0.090426	0.0899011
20000	4.7058	1.23645	2.25016	0.0968273	0.0889181

A seguir gráfico interpolado dos resultados:

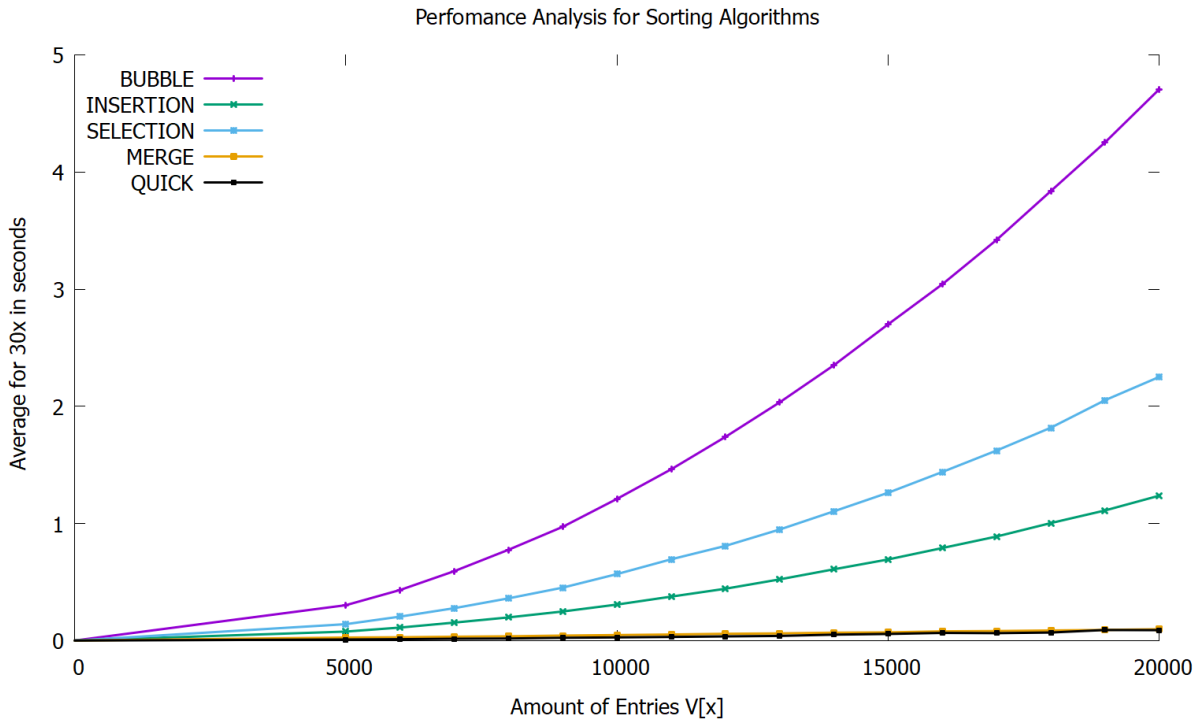

Figura 3 - Gráfico comparativo dos resultados obtidos, médias dos tempos de execução dos algoritmos de ordenação para vetores com tamanhos progressivos.

Comentários:

Dos 5 métodos avaliados, o **Bubble Sort** demonstrou significativo aumento de tempo necessário, caracterizando uma curva do tipo $O(n^2)$, o que concorda com sua análise de complexidade.

Tanto **Insertion sort** quanto **Selection sort** apresentaram resultados significativamente melhores se comparados ao **Bubble Sort** apesar de também possuírem complexidade de tempo **$O(n^2)$**, sendo **Insertion sort** o algoritmo com melhor desempenho (aproximadamente x4 menor do que o **Bubble sort!**) dos 3.

Tanto o **Merge** quanto o **Quick Sort** apresentaram resultados extremamente satisfatórios para os tempos estimados (aproximadamente x50 vezes menores do que o pior resultado - **Bubble Sort**) e isso se explica pela análise de complexidade, que para o **Merge Sort**, que utiliza de técnicas recursivas de divisão do vetor para realizar a ordenação, equivale a uma curva do tipo **O(n Log n)**. Já o **Quick Sort** apesar de em maioria apresentar comportamento similar ao do **Merge Sort**, ou seja **O(n log n)** para casos mais favoráveis, é dependente assim da escolha do **elemento pivot** podendo no seu pior caso, apresentar comportamento do tipo **$O(n^2)$**.

Uma observação importante é que ambos os métodos (**Merge e Quick**) fazem uso de **consumos elevados de memória** para realizar operações eficientes, e isso acontece **devido a natureza de seus algoritmos recursivos**, podendo neste caso não serem adequados à sistemas que possuem escassez deste tipo de recurso.

CONCLUSÃO

O resultado deste experimento permite concluir que se o que é desejado no processo de ordenação é rapidez, independente do consumo de mais recurso de memória disponível, **Merge** e **Quick Sort** são opções interessantes, visto que apresentam resultados melhores de tempo, sendo o **Quick Sort** uma opção menos instável, a depender da escolha do pivot e também do nível de desordem do Vetor. Todavia, caso memória e outros recursos sejam limitantes no processo, o **Insert Sort** pode ser mostrar uma alternativa mais interessante, pois apesar de apresentar uma curva de consumo de tempo do tipo **$O(n^2)$**, ainda sim entrega resultados de tempos bem menores do que as outras opções, não tendo seu algoritmo focando em recursividade, mas em iteração , o que nos leva há um consumo de memoria inferior se comparado aos métodos recursivos.

MATERIAL DE REFERÊNCIA

1 - 2010, ESTRUTURA DE DADOS - Ana Fernanda Gomes Ascencio & Graziela Santos de Araujo