Relatório de Trabalho - Ordenação

CI1056 - Algoritmos e Estrutura de Dados 2

Vinicius Evair da Silva
Graduando do Curso de Bacharelado em Ciência da Computação
Departamento de Informática
Universidade Federal do Paraná – UFPR
Curitiba, Brasil
vinicius.evair@ufpr.br

I. Introdução

O presente relatório documenta diversos testes de algoritmos de ordenação e de busca para a disciplina de Algoritmos e Estrutura de Dados 2 (CI1056).

O objetivo do trabalho é a comparação de desempenho entre cinco algoritmos de ordenação (*Insertion Sort*, *Selection Sort*, *Merge Sort*, *Quick Sort* e *Heap Sort*) e também entre dois algoritmos de busca (*Sequential Search* e *Binary Search*).

II. Da Construção

Os algoritmos deste trabalho foram inteiramente implementados utilizando a máquina virtual **CPU1** do Departamento de Informática, que utiliza o Sistema Operacional `LMDE 5 (elsie) x86_64`. O programa em C que testa os algoritmos foi compilado sem erros ou avisos, utilizando a versão `10.2.1-6` do **GCC** (*GNU Compiler Collection*).

A. Parâmetros Utilizados:

Foram construídos vetores com tamanhos variando entre 5000 e 65000, em incrementos de 5000, com cada tamanho tendo três variações: um vetor de valores aleatórios, um vetor ordenado de maneira crescente e um vetor ordenado de maneira decrescente.

O vetor aleatório utilizado foi gerado no início do programa com 65000 espaços e para cada teste, foi utilizado um *memcpy*, garantindo assim uma padronização dos testes.

B. Especificações dos Algoritmos de Ordenação:

1) Insertion Sort: Para a implementação do *Insertion Sort*, foi utilizada a busca binária, garantindo um melhor desempenho ao algoritmo, que depende completamente de seu esquema de busca interno.

2) Quick Sort: Foi definido como escolha de pivô para a implementação do *Quick Sort* sempre o último índice do vetor. Apesar de afetar negativamente o desempenho do algoritmo, a decisão foi tomada para manter um paralelo com os cálculos e construções realizadas durante as aulas.

III. Das Análises de Ordenação

A. Com vetores aleatórios:

A tabela a seguir mostra a quantidade de comparações em relação ao tamanho para um vetor de valores aleatórios:

Comparações / Tamanho	*Insertion Sort*	*Selection Sort*	*Merge Sort*	*Quick Sort*	*Heap Sort*
5.000	54.526	12.497.500	61.808	69.900	107.683
10.000	118.989	49.995.000	133.616	151.340	235.267
15.000	187.331	112.492.500	208.616	244.104	370.192
20.000	257.978	199.990.000	287.232	338.497	510.738
25.000	330.707	312.487.500	367.232	415.995	654.984
30.000	404.724	449.985.000	447.232	533.783	800.704
35.000	479.750	612.482.500	529.464	612.007	949.283
40.000	556.085	799.980.000	614.464	705.869	1.101.771
45.000	633.397	1.012.477.500	699.464	835.452	1.255.288
50.000	711.426	1.249.975.000	784.464	898.375	1.409.885
55.000	790.145	1.512.472.500	869.464	1.040.183	1.565.669
60.000	869.476	1.799.970.000	954.464	1.154.555	1.721.309
65.000	949.211	2.112.467.500	1.039.464	1.237.724	1.877.904

Logo de início é possível perceber que o *Selection Sort* realiza uma quantidade colossal de comparações em contraste com os demais algoritmos, o que indica uma provável qualidade inferior.

Com exceção do algoritmo acima explicitado, os resultados não apresentam grande disparidade entre si mesmos, e consequentemente, não mostram muitas informações. Porém, ao olharmos para o gráfico adiante, que compara o tempo dos algoritmos, alguns esclarecimentos já podem começar a serem feitos:

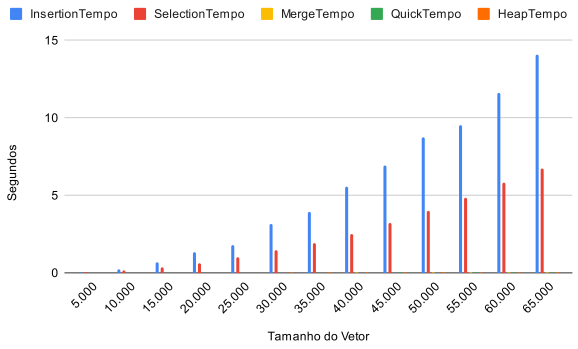

Figura 1. Tamanho x Tempo de execução com vetor aleatório.

B. Com vetores crescentemente ordenados:

Para maiores conclusões, olhemos agora para os resultados de comparações dos vetores ordenados de maneira crescente:

Comparações / Tamanho	*Insertion Sort*	*Selection Sort*	*Merge Sort*	*Quick Sort*	*Heap Sort*
5.000	56.821	12.497.500	61.808	12.497.500	112.126
10.000	123.630	49.995.000	133.616	49.995.000	244.460
15.000	193.630	112.492.500	208.616	112.492.500	383.177
20.000	267.247	199.990.000	287.232	199.990.000	529.074
25.000	342.247	312.487.500	367.232	312.487.500	677.688
30.000	417.247	449.985.000	447.232	449.985.000	826.347
35.000	494.480	612.482.500	529.464	612.482.500	979.446
40.000	574.480	799.980.000	614.464	799.980.000	1.138.114
45.000	654.480	1.012.477.500	699.464	1.012.477.500	1.296.774
50.000	734.480	1.249.975.000	784.464	1.249.975.000	1.455.438
55.000	814.480	1.512.472.500	869.464	1.512.472.500	1.614.090
60.000	894.480	1.799.970.000	954.464	1.799.970.000	1.772.744
65.000	974.480	2.112.467.500	1.039.464	2.112.467.500	1.931.415

Como discutido em sala, o *Quick Sort* não lida muito bem com vetores ordenados quando o pivô escolhido é o primeiro ou o último elemento. E na tabela acima é possível ver que suas comparações saltaram para a mesma quantidade do *Selection Sort*, que vale lembrar, não tem melhor ou pior caso e portanto continua com a mesma quantidade de comparações.

O problema do *Quick Sort* fica ainda mais evidente ao olharmos para o gráfico de comparação do tempo de execução:

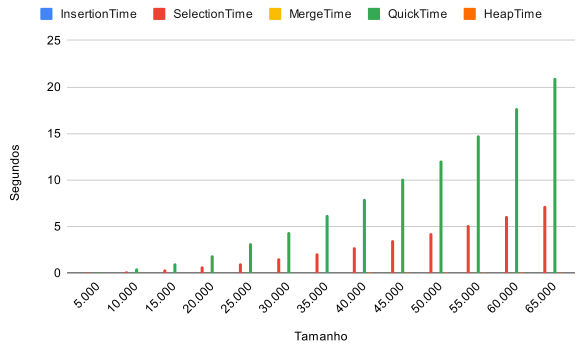

Figura 2. Tamanho x Tempo de execução com vetor crescente.

O tempo de execução do *Quick Sort* se mostra ainda maior que o tempo do *Insertion Sort* com vetor aleatório, e além disso, fazendo muito mais comparações.

Por outro lado, o tempo de execução do *Insertion Sort* foi de 0,012 segundos para um vetor de tamanho 65000, um desempenho excelente em comparação com seu resultado anterior. A queda drástica no tempo se dá pelo fato do *Insertion Sort* ser um algoritmo de ordenação adaptativo, que tem uma eficiência diretamente proporcional a quão ordenado o vetor previamente estava.

Os demais algoritmos não foram afetados — pelo menos não de maneira relevante — pela mudança do tipo vetor e portanto não serão comentados.

C. Com vetores decrescentemente ordenados

Os resultados para vetores ordenados de maneira decrescente apenas reforçam o que já foi previamente discutido. Para um vetor de tamanho 65000:

- Em questão de comparações, os algoritmos *Insertion Sort* e *Heap Sort* tiveram uma variação pequena em seus resultados, com uma quantidade de 64985 e 106238 comparações a menos respectivamente. Os demais obtiveram a exata mesma quantidade de comparações.
- Em questão de tempo, dois algoritmos tiveram uma variação significativa em seus resultados, sendo um acréscimo de 27,78 segundos para o *Insertion Sort* e decréscimo de 7,71 para o *Quick Sort*. Os demais algoritmos tiveram uma variação insignificante no tempo.

Após todas as análises, o *Merge Sort* aparenta ter o melhor dos desempenhos, porém outras variáveis, como o custo de memória dos algoritmos, não foram contempladas nesta pesquisa.

IV. Das Análises de Busca

Para a análise dos algoritmos de busca, apenas um teste simples foi feito: para cada tamanho, um valor aleatório foi selecionado e buscado no vetor com cada um dos algoritmos. Os resultados se encontram na tabela a seguir:

Comparações / Tamanho	Busca Sequencial	Busca Binária	Comparações / Índice
5.000	1.100	10	3900
10.000	5.973	13	4027
15.000	6.439	13	8561
20.000	9.649	14	10351
25.000	23.867	11	1133
30.000	5.605	12	24395
35.000	16.913	12	18087
40.000	30.444	14	9556
45.000	27.485	16	17515
50.000	20.814	16	29186
55.000	6.733	16	48267
60.000	10.267	16	49733
65.000	45.523	13	19477

O tempo não foi explicitado na tabela acima, uma vez que todos os testes retornaram 0 ou um valor próximo de 0 como tempo de execução (utilizando oito dígitos significativos).

Buscando índices aleatórios, podemos perceber que o desempenho da Busca Sequencial é completamente dependente do índice sendo buscado: quanto mais longe do ponto inicial, mais comparações são necessárias.

Mostrando grande disparidade, a Busca Binária, um algoritmo extremamente eficiente, faz uma quantidade infinitamente menor de comparações, que inclusive não dependem da posição buscada, mas por outro lado, a Busca Binária depende do vetor já estar ordenado, enquanto a Busca Sequencial está livre desse problema.

V. Conclusão

Mesmo neste pequeno recorte é possível observar o comportamento dos algoritmos estudados durante a disciplina. Os testes explicitam a conclusão já discutida em sala de aula, de que não existe simplesmente um algoritmo melhor que os demais, pois cada um tem as suas próprias nuances e casos de uso.

Apesar de limitada, esta pesquisa aqui relatada instiga uma investigação continuada, com pesquisas futuras mais robustas, com mais recursos e, consequentemente, com resultados mais esclarecedores, numa constante busca pelo aperfeiçoamento.

Referências

[1] P. R. L. de Almeida, "**Algoritmos e Estrutura de Dados 2**," [Notas e Slides de Aula], Dept. de Informática, Universidade Federal do Paraná (UFPR), Curitiba, PR, 2022.