**Trabalho Prático 2**

**Análise de Algoritmos de Ordenação**

**Vinícius Pinho Galvão**

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

[viniciuspgalvao@ufmg.br](mailto:viniciuspgalvao@ufmg.br)

**1. Introdução**

O trabalho proposto tem como objetivo implementar uma série de variações de um algoritmo de ordenação conhecido como QuickSort e analisar suas diferenças em desempenho e eficiência. As métricas de comparação foram: número de comparações de chaves, número de copias de registros efetuadas e tempo total de ordenação. Para esse trabalho, foram analisadas as seguintes versões do algoritmo (em módulos):

QuickSort Recursivo: recQuicksort.cpp;

QuickSort mediana: medQuicksort.cpp;

QuickSort com Seleção: selQuicksort.cpp;

QuickSort iterativo: stackQuicksort.cpp;

QuickSort iterativo “inteligente”: pileQuicksort.cpp.

Além disso, após delimitar a versão mais eficiente entre esses, ela foi também comparada com mais dois métodos de ordenação, conhecidos como HeapSort e MergeSort.

**2. Implementação**

O programa foi desenvolvido na linguagem C++, compilada pelo compilador g++ da GNU Compiler Collection. Os testes foram feitos em uma máquina com Windows 10 com WSL2 instalado para a simulação de um ambiente Linux. O computador possui 16GB de memória RAM e um processador Ryzen 5 3600.

**2.1 Estrutura de Dados**

Para implementar os algoritmos necessários, primeiramente temos que criar registros de memorias para serem ordenados. Assim, na biblioteca dados.hpp, são descritas estruturas de dados feitas para armazenarem certa quantidade de memória e manipulá-la.

A mais importante é a struct Item, que representa nosso registro de memória. Ela contém uma chave, que será utilizada na ordenação, 15 cadeias de string com 200 caracteres cada, e 10 números reais. Logo, criamos um array de Itens (registros) de tamanho N e os preenchemos com a utilização da função preencheVetor(). Os últimos dois atributos são gerados aleatoriamente de acordo com uma semente (seed) passada pelo usuário.

Para implementar as funções de ordenação, utilizando a ideia de modularização, os arquivos descritos na introdução foram criados. Cada um deles importa a biblioteca dados.hpp e implementa todas as suas funções necessárias para ordenar o array de Itens pela sua chave, de acordo com sua modificação.

**2.2 Coleta de métricas**

Para analisar o desempenho dos algoritmos, foi necessário coletar as métricas necessárias, que foram:

* Número de comparações de chaves
* Número de copias de registros realizadas.
* Tempo total gasto na ordenação

Para a coleta das duas primeiras, foram criadas duas variáveis globais para servirem de contadores, comparações e copias, e estas foram repassadas a todos os módulos por meio de sua declaração como variável externa no modulo dados.hpp (external int comparações). Elas foram usadas, assim, nos arquivos que implementam as funções de ordenação. Toda vez que uma comparação entre chaves é feita, o seu contador é incrementado. Para as copias, toda vez que uma atribuição envolvendo o registro é realizada, o contador de cópias é incrementado. Assim, por exemplo, uma operação básica de troca tal como {Item C = A; A=B; B=C}, conta como 3 cópias.

Para a medição do tempo gasto na ordenação, foi utilizada a biblioteca <chrono> da STL do C++. Assim, duas marcas são postas no código: uma variável start e uma chamada end. As duas são iguais ao seguinte comando da biblioteca*: chrono::steady\_clock::now()*. As marcas são postas antes e depois da chamada da função de ordenação, de modo a contar apenas o tempo gasto pela função, desconsiderando o tempo de preenchimento do vetor (que inclusive é bem demorado).

Logo, temos dois instantes de tempo, e realizando a diferença entre os dois (end-start), temos o total de tempo gasto exclusivamente pela função.

As três métricas são impressos no arquivo de saída, uma linha por ordenação. A linha ainda expressa a versão do quicksort, se houve algum parâmetro adicional, e o número de registros ordenados.

Exemplo de linha do output:

Versão 2: Registros: 5000 k: 5 Comparações: 48476 Copias: 85628 Tempo: 21ms.

**2.3 Main**

No arquivo main, temos um fluxo de operações que consiste em:

* Ler as flags de comando do usuário informando qual versão do quicksort será analisada, qual o nome dos arquivos de entrada e saída, e a seed escolhida para os testes
* Testar se todas as flags são validas, por meio da função erroAssert da biblioteca msgAssert.h
* Para cada número de registros, devemos rodar o código com 5 seeds diferentes, e pegar a média das métricas avaliadas. Para isso, o loop principal do programa conta com dois for loop aninhados. O primeiro roda o total de numero de testes (int N do arquivo de entrada), e o segundo roda 5 vezes para cada teste, diferenciando a seed no final da execução.

Assim, dentro destes for loops aninhados, temos uma clausula switch que compara a versão do quicksort escolhida pelo usuário e chama a função designada. Por exemplo, se a versão escolhida for igual a 2, o switch ira chamar o case 2, que por sua vez chamará a função medQuickSort().

Ao redor dessa clausula switch, foram colocados os marcadores de tempo da biblioteca <chrono>, que documentam, em microssegundos, qual foi o tempo necessário para realizar a ordenação daqueles registros. Então, todas as métricas medidas são salvas no arquivo de output criado (comparações, copias e tempo), o vetor de registros é deletado, e a seed é alterada para se testar novamente.

Ao final das 5 execuções e de todos os casos de teste, o programa imprime no console o tempo total de execução do programa, fecha os arquivos de input e output, e termina.

**3. Análise de complexidade**

**3.1 Tempo**

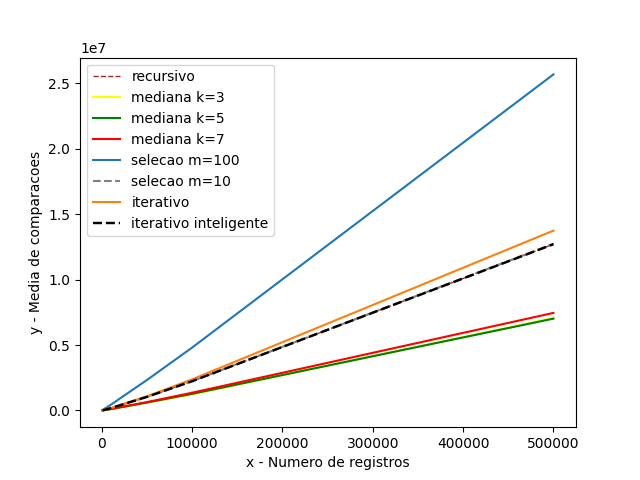
**Awdaw**

**3.2 Espaço**

**4. Análise Experimental**

**4.1. Comparações.**

Segue abaixo um gráfico plotado por um script de python usando a biblioteca matplotlib para análise dos resultados:

****

Analisando o gráfico, percebemos mais superficialmente que o algoritmo menos eficiente em termos de comparação é o quicksort seleção com m=100. Para 500.000 registros, ele faz aproximadamente 25.000.000 comparações, um comportamento bastante custoso.

Logo abaixo dele, na parte de eficiência média do gráfico, encontram se 4 algoritmos: Primeiramente, o menos eficiente se mostra o iterativo comum que não prioriza uma partição para processamento, o fazendo um pouco mais custoso que o iterativo “inteligente”.

Após a linha laranja, temos 3 algoritmos basicamente emaranhados por terem o custo quase idêntico. São eles o de seleção m=10, o iterativo “empilha inteligente” e o recursivo. O motivo para isso é que o iterativo e o recursivo fazem exatamente o mesmo número de comparações por utilizarem o mesmo código de partição. A única mudança entre os dois é a forma com que os índices são ordenados para repartição. O de seleção m=10 entra nesse custo por também utilizar o mesmo código de partição. O fato de utilizar a seleção para ordenar os vetores pequenos o faz levemente mais custoso do que os outros dois, pelo fato do algoritmo selectionSort ser mais comparativo do que o quicksort. Percebe-se também que esse fato é o que torna a seleção m=100 tão mais custosa do que esse e os outros métodos.

No final, os algoritmos mais eficientes são os quicksorts com implementação de pivô mediana. Essa eficiência se dá pela maior chance do pivô se localizar no meio da partição. Entre eles, percebe-se que o custo do k=7 o torna menos eficiente. Os resultados dão a entender que quanto maior o k, menos eficiente ele se torna, com k=3 sendo a medida ideal para todos os números de registros. No entanto, exclusivamente para o maior valor de registros, 500.000, o algoritmo com k=5 se mostrou mais eficiente que o k=3, por ordem de aproximadamente 60.000 comparações a menos.

Concluímos então que, se o intuito da ordenação é minimizar o custo de comparações, o Quicksort com o pivô de mediana deve ser adotado. Para um número de registros pequeno e médio, deve-se usá-lo com k=3. Para registros maiores, o k=5 torna-se levemente mais vantajoso.

**5. Conclusão**

**Referencias**

Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciencia da Computação. Universidade Federal de Minas Gerais. Belo Horizonte

**5. Instruções de compilação e execução**

Para compilar o programa, uma makefile está disponível no diretório raiz e pode ser usada com o comando “make” no terminal.

Para execução, há 4 mensagens de uso possíveis no programa:

    -i  "nome do arquivo de entrada.ppm"

    -o  "nome do arquivo de saída"    (saída em formato .pgm)

    -p  log.out            (registro de desempenho)

    -l                     (padrão de acesso e localidade)

Um código exemplo de execução seria algo como: ./bin/imgconversor -i mineirao.ppm -o mineirao -p log.out -l

É importante salientar que o arquivo de entrada deve ser escrito com sua extensão .ppm, pois se for outro tipo de arquivo um erro será retornado. No entanto, o arquivo de saída pode ser apenas um nome, visto que o próprio programa adicionará a extensão .pgm.