**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. 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 memoria. Ela contem uma chave, que será utilizada na ordenação, 15 cadeias de string com 200 caracteres cada, e 10 números reais. Os últimos dois atributos são gerados aleatoriamente de acordo com uma semente (seed) passada pelo usuário por meio da função preenche Vetor().

**2.2. Classes**

A implementação do projeto conta com três classes diferentes e interligadas, são elas: imagem, imagemPGM e imagemPPM. Todos os arquivos fontes podem ser encontrados na pasta src e seus cabeçalhos na pasta include.

A classe imagem define um template para as especificações PGM e PPM. Ela possui métodos do tipo getSize() e setSize() e possui como atributos básicos:

Unsigned int comprimento e altura, referentes a imagem;

Int maxcolor, encontrado no header das imagens ppm e pgm;

String Id, também encontrado no header, esperado “P3” ou “P2”.

As outras classes herdam da imagem e possuem métodos específicos para seu desempenho no programa.

A classe imagemPPM possui o método leitorPPM(), que recebe como parâmetro uma ifstream que contêm o arquivo de entrada. Essa função lê a imagem e armazena os valores de vermelho, verde e azul em cada um dos pixels alocados na matriz.

Essa classe também tem a função getPixel(), que recebe como parâmetro uma linha e uma coluna e retorna o pixel referente a essa posição, que será usado nas próximas funções.

A imagemPGM, por outro lado, possui dois métodos que são usados em conjunto: o writePGM() e o createImageFromPPM(). A primeira é chamada na main após a leitura do arquivo ppm ter sido retornada com sucesso. Ao ser chamada, recebe como parâmetro o tamanho da imagem ppm recebida, assim como uma própria instancia da classe imagemPPM por referência, além do nome do arquivo de saída.

Logo, ela cria o arquivo com a extensão .pgm, e se não houve erro, chama a função createImageFromPPM, que recebe a mesma referência de imagem, assim como o seu tamanho, e aloca a matriz necessária para o armazenamento da imagem, além de fazer a conversão de cada pixel RGB para seu tom de cinza correspondente.

Assim, após o encerramento dessa função, se tudo retornou status validos, a writePGM insere no arquivo o cabeçalho da imagem, com seu maxcolor, tamanho e a assinatura “P2”, e escreve o conteúdo da matriz no arquivo.

**2.3 Main**

No arquivo main, primeiramente tentamos abrir o arquivo de entrada recebido. Se tudo ocorrer corretamente, criamos uma variável imagemPPM chamada img\_colorida e lemos e gravamos seu tamanho, maxcolor, e checamos se a assinatura da foto bate com a esperada (P3). Após isso, chamamos a função leitorPPM() para ler e armazenar corretamente a imagem de entrada. Se nenhum erro for retornado nessa etapa, criamos então uma imagemPGM img\_cinza e chamamos o método writePGM() com os valores requeridos, o que cria o output da imagem no diretório padrão com o nome escolhido pela flag de inicialização.

**3. Análise de complexidade**

**3.1 Tempo**

Temos como funções relevantes para a análise da complexidade do algoritmo as funções de alocação e preenchimento das matrizes de pixels.

A primeira alocação, da matriz de pixels RGB, possui um custo dependente da resolução da imagem, assim como seu preenchimento, que por ser feito dinamicamente sempre vai utilizar 100% da matriz. Assim, o custo dessas duas operações e da ordem de O(n²).

A função de conversão também se da como O(n²), pois ela acessa toda a matriz de pixels RGB da imagem ppm, e converte cada pixel em preto e branco. Mesmo sendo três operações de uma vez, essas constantes são desconsideradas na hora da soma total e continuam na mesma ordem de crescimento.

No final, temos a função de escrita do arquivo, writePGM(), que possui funcionamento similar as de cima. Por contar como um acesso por toda a matriz, possui ordem O(n²).

Logo, a soma dessas ordens da uma complexidade total de O(n²).

**3.2 Espaço**

Na análise de complexidade de espaço, veremos como as funções de alocação de matrizes se comportam.

A alocação da matriz PPM por definição consome o triplo do espaço da matriz PGM, enquanto a última ocupa m x n posições de memória, a primeira ocupa m x n \* 3. Assim, o custo de espaço pode ser considerado O(n²), pois mesmo variando os valores m e n, ainda é uma função quadrática.

**4. Conclusão**

Ao finalizar o trabalho, fica claro a divisão das etapas e as capacidades treinadas por cada uma. O desenvolvimento do algoritmo em si, é usado para treinar a capacidade de montar estruturas de dados corretas e eficientes, com modularização e tratamento de exceções.

A outra parte treinada foi a de análise de complexidade de tempo e espaço, que são extremamente importantes de serem consideradas a desenvolver um algoritmo. Evitar soluções triviais que gastam muitos recursos computacionais é uma tarefa bem difícil dependendo do problema, assim, é uma habilidade que deve ser constantemente refinada.

Logo, o projeto trabalhou uma gama de habilidades que são essenciais para qualquer programador no geral, que vai muito além de apenas solucionar o problema em questão, mas sim a melhor forma de realizar tal ato.

**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.