Trabalho Prático 2

Análise de desempenho de algoritmos de ordenação

Matheus Guimarães Couto de Melo Afonso
Matrícula: 2021039450

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

matheusgcma@ufmg.br

1. Introdução

O trabalho prático 2 consistiu em implementar cinco variações diferentes do algoritmo de ordenação QuickSort, além da implementação dos algoritmos MergeSort e HeapSort. As implementações do QuickSort realizadas foram:

- recursivo: variação convencional e utilizada de maneira geral.
- recursivo com mediana: o pivô é escolhido como a mediana de k elementos aleatórios do vetor.
- recursivo com seleção: o algoritmo usa a ordenação por seleção a partir de partições de um tamanho m.
- não recursivo: utiliza uma pilha para armazenar as partições a serem ordenadas.
- não recursivo com pilha inteligente: a pilha ordena as menores partições primeiro.

Cada um dos algoritmos de ordenação foi executado com tamanhos de vetores diferentes e contabilizando a quantidade de cópias, comparações e tempo de execução de cada um deles. Inicialmente, foi comparado o impacto de cada uma das variações do QuickSort e, posteriormente, a melhor delas foi comparada aos algoritmos MergeSort e HeapSort.

A seção 2 contém uma explicação mais profunda sobre as estruturas de dados e métodos implementados, bem como informações sobre o ambiente de desenvolvimento. A seção 3 apresenta uma análise de complexidade de cada um dos algoritmos de ordenação, enquanto a seção 4 se aprofunda nas estratégias de robustez aplicadas no desenvolvimento. Na seção 5, mostro os resultados dos testes de desempenho dos algoritmos e as comparações entre eles, e por fim a seção 6 conclui a documentação do trabalho.

2. Método

2.1. Organização do projeto

O projeto é estruturado no modelo sugerido pelos professores: a pasta **src** armazena os arquivos de código (*.cpp) e a pasta **include** têm os arquivos de cabeçalho (*.h). Ao compilar o código por meio do **Makefile**, que está na pasta raiz do projeto, os arquivos *.o são gerados no diretório **out** e o executável **quicksort** é gerado na pasta **bin**. A única diferença na organização é a existência da pasta **assets**, onde coloquei um arquivo padrão de entrada **entrada_padrao.txt**, caso o usuário não tenha um arquivo para fornecer.

2.2. Estruturas de dados

A única estrutura de dados implementada foi a *struct* **Item**, que corresponde aos registros que serão ordenados pelos algoritmos. A ordenação é baseada no atributo **id**.

```
struct Item {
    int id;
    std::string str[15];
    double num[10];
};
```

O programa também implementa a *struct* Flags, que serve apenas para salvar os diferentes argumentos de linha de comando que o programa pode receber, como o caminho para arquivos de entrada e saída ou se devem ser salvos os *logs* de registro de desempenho, além das opções de seed, tipo do algoritmo e k ou m para os algoritmos do tipo 2 e 3, respectivamente.

```
struct Flags {
    std::string input_file;
    std::string output_file;
    char log_file[100];
    bool regmem;
    int type;
    int random_seed;
    int median_k;
    int select_stop;

    Flags()
        : input_file("assets/entrada_padrao.txt"),
          output_file("bin/saida.txt"),
          log_file("bin/log.out"),
          regmem(false),
          type(1),
          random_seed(77),
          median_k(3),
          select_stop(10) {}
};
```

2.3. Algoritmos de ordenação

Como apresentado na introdução, foram implementados sete algoritmos de ordenação diferentes, sendo cinco deles variações do QuickSort, uma variação de MergeSort e uma de HeapSort. Cada uma delas pode ser utilizada por meio de uma opção da linha de comando.

1. QuickSort recursivo: ordena o vetor recursivamente, utilizando como pivô o elemento na posição central do vetor inicial.
2. QuickSort recursivo com mediana: recebe como parâmetro um inteiro k, que corresponde à quantidade de elementos do vetor selecionados aleatoriamente cuja mediana será o pivô da ordenação.
3. QuickSort recursivo com SelectionSort: recebe como parâmetro um inteiro m. A partir de partições de tamanho m, o algoritmo utiliza o SelectionSort para ordená-las, uma vez que o QuickSort não é tão eficiente em partições pequenas.
4. QuickSort não recursivo: armazena as partições do vetor em pilhas, de forma que dispensa a recursividade convencional do QuickSort.
5. QuickSort não recursivo com pilha inteligente: de maneira semelhante à opção 4, armazena as partições do vetor em pilha, mas processa as menores partições primeiro.
6. MergeSort
7. HeapSort

Uma vez que a estrutura dos algoritmos é facilmente encontrada na Internet, não julgo necessário um aprofundamento nas minhas escolhas de implementação.

2.4. Ambiente de desenvolvimento

- Sistema operacional: WSL Ubuntu 20.4
- Linguagem de programação: C++
- Compilador: g++ Ubuntu 9.4.0-1ubuntu1
- Processador: Intel Core i5-9400F @2.90GHz
- Memória RAM: 2 x 8GB

3. Análise de Complexidade

A complexidade dos algoritmos de ordenação depende do tamanho dos vetores a serem ordenados. A seção está dividida em duas partes – complexidade de tempo e espaço – e em cada uma delas analiso os algoritmos individualmente.

3.1. Complexidade de tempo

1. QuickSort recursivo: no caso médio, tem complexidade $O(n \log n)$. Contudo, no pior caso (quando o pivô é sempre o maior ou menor elemento), tem complexidade $O(n^2)$.

2. QuickSort recursivo com mediana: a utilização da mediana como pivô impede a ocorrência do pior caso da opção 1, portanto tem complexidade $O(n \log n)$.
3. QuickSort recursivo com SelectionSort: tem complexidade semelhante ao caso médio da opção 1; contudo o tamanho das partições que utilizam o SelectionSort (m) também influencia a complexidade do algoritmo. Dessa forma, a complexidade é dada por $O(n \log n) + O(m^2)$.
4. QuickSort não recursivo: complexidade de tempo semelhante à opção 1.
5. QuickSort não recursivo com pilha inteligente: complexidade de tempo semelhante ao caso 1.
6. MergeSort: é um algoritmo que se baseia no paradigma de divisão e conquista, portanto a complexidade de tempo é $O(n \log n)$ em todos os casos.
7. HeapSort: comportamento $O(n \log n)$ em todos os casos.

3.2. Complexidade de espaço

1. QuickSort recursivo: não utiliza memória adicional; $O(1)$.
2. QuickSort recursivo com mediana: não utiliza memória adicional na ordenação, mas utiliza no cálculo da mediana, que cria um vetor de tamanho k, portanto a complexidade é $O(k)$.
3. QuickSort recursivo com SelectionSort: não utiliza memória adicional; $O(1)$.
4. QuickSort não recursivo: no pior caso, tem complexidade de espaço $O(n)$, já que realizaria n partições de tamanho 1.
5. QuickSort não recursivo com pilha inteligente: soluciona o problema de espaço da opção 4, tendo complexidade $O(\log n)$ no pior caso.
6. MergeSort: armazena cada elemento do vetor em uma partição individual, portanto $O(n)$.
7. HeapSort: utiliza espaço $O(n)$ para construir o heap.

4. Estratégias de Robustez

A robustez de um código diz respeito à sua capacidade de funcionar corretamente mesmo em condições anormais, por meio de mecanismos de programação defensiva e tolerância a falhas. A robustez do programa foi implementada por meio da biblioteca **msgassert.h**, disponibilizada pelos professores de Estrutura de Dados.

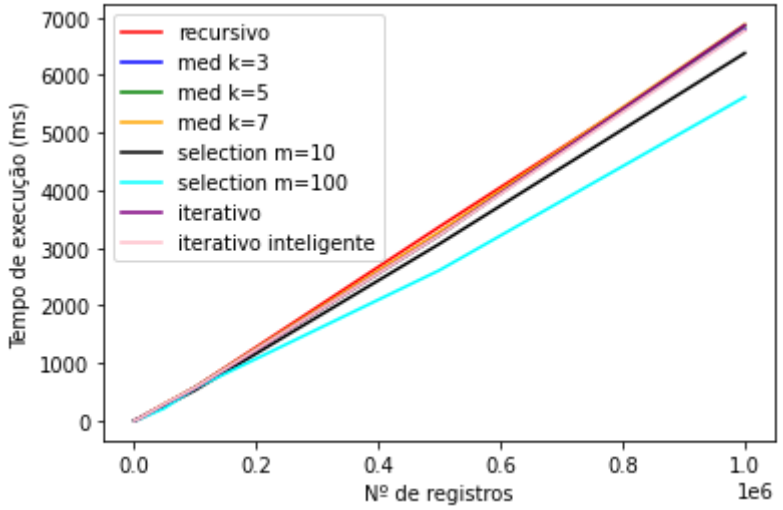
Após receber e interpretar os argumentos recebidos na linha de comando, é utilizada a função *avisoAssert*, para informar o usuário onde estão o arquivo de entrada, saída e os logs de execução padrões, que são utilizados no caso de o usuário não passar estas informações como parâmetro. De forma semelhante, caso o usuário não escolha o tipo do quicksort e a semente, é utilizado o quicksort tipo 1 (recursivo) e a semente 77, que escolhi arbitrariamente. Quando o usuário escolhe os tipos 2 ou 3 e não passa os valores de k ou m, são utilizados $k = 3$ e $m = 10$ por padrão. Se o usuário escolher uma opção de ordenação não reconhecida (>7 ou <0) o programa é abortado.

5. Análise Experimental

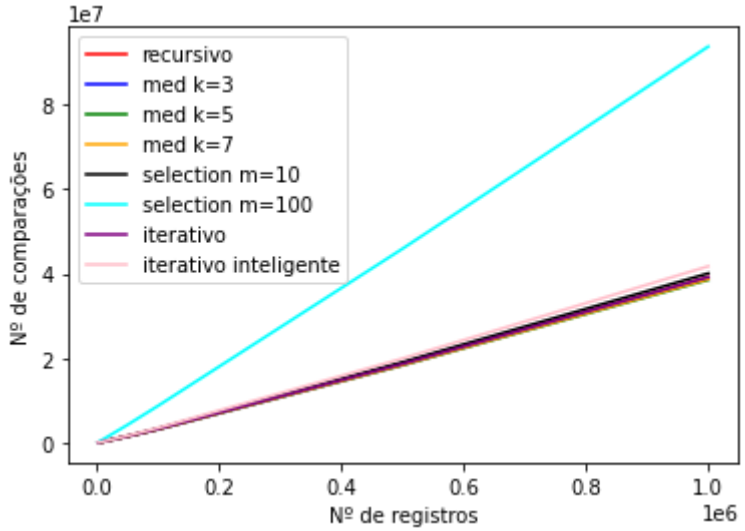
A análise experimental consiste em executar cada um dos algoritmos de ordenação, comparar seus resultados no que diz respeito a número de trocas, comparações e tempo de execução. O tempo de execução foi medido por meio da biblioteca **<chrono>** do C++. Para cada um dos algoritmos, foram realizados 5 testes com vetores de tamanhos 1.000, 5.000, 10.000, 50.000, 100.000, 500.000 e 1.000.000 utilizando as seeds 100, 101, 102, 103 e 104.

Essa seção será dividida em duas partes. Na primeira serão comparados as cinco versões do QuickSort, e na segunda a que tiver o melhor desempenho será comparada aos MergeSort e HeapSort.

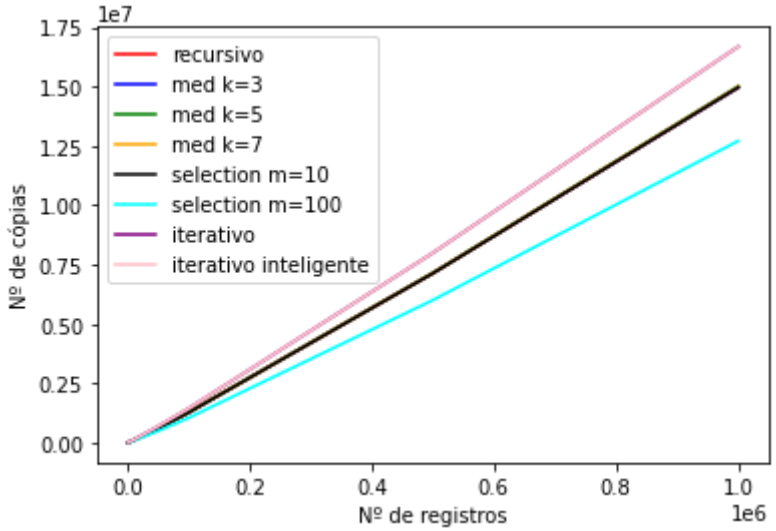
5.1. Impacto de variações do QuickSort



O primeiro gráfico mostra o tempo de execução em milissegundos de cada uma das variações do QuickSort. É possível perceber um emparelhamento dos tipos 1, 2, 4 e 5 quanto à esta métrica. Os algoritmos que se saíram melhor foram os do tipo 3, que utilizam o algoritmo de seleção para ordenar partições de tamanhos menores. Isso é esperado, uma vez que o QuickSort se mostra ineficiente para ordenar vetores de tamanhos pequenos. Dentre os algoritmos do tipo 3, a variação com $m = 100$ teve um desempenho melhor do que com $m = 10$, o que numa primeira observação pode parecer inesperado, mas uma vez que os vetores de entrada tem tamanhos que chegam até 1 milhão, há uma diminuição considerável no número de partições feitas pelo QuickSort recursivo.



O segundo gráfico leva em consideração o número de comparações feitas por cada um dos algoritmos. Novamente, é perceptível o emparelhamento da maioria dos algoritmos, com apenas um deles fugindo à regra: o tipo 3 com $m = 100$. Essa diferença exorbitante ocorre porque o algoritmo de seleção por natureza realiza um grande número de comparações se comparado com o QuickSort.

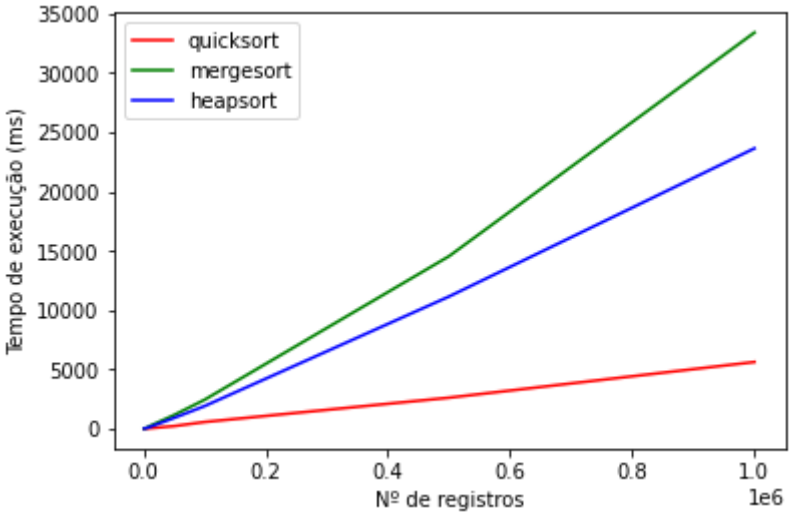


Por fim, o terceiro gráfico mostra a quantidade de cópias de cada um dos algoritmos. Houveram dois casos notáveis: o QuickSort do tipo 5 realiza um número de cópias ligeiramente maior que a maioria, enquanto o do tipo 3 com $m = 100$ realiza um número ligeiramente menor.

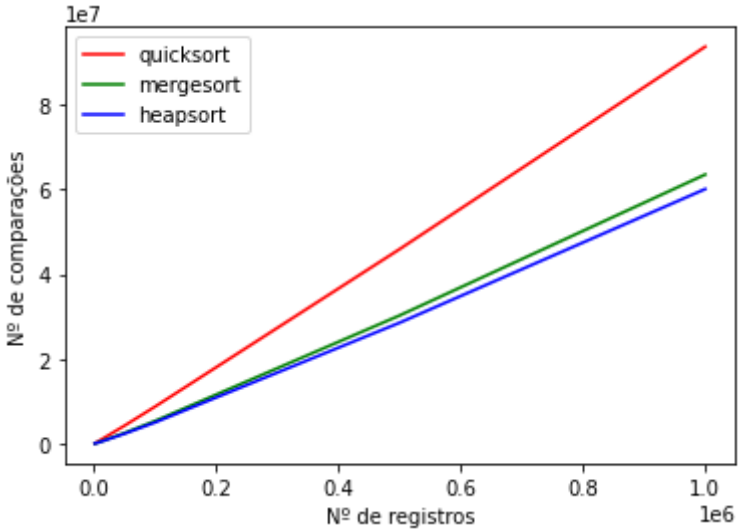
Ao considerar os três testes, considero que o algoritmo com desempenho mais satisfatório foi o QuickSort recursivo com SelectionSort para partições de tamanho 100. Apesar de ter um número de comparações expressivamente maior em relação aos outros, seu tempo de execução e gasto de memória foram bem menores.

5.2. QuickSort x MergeSort x HeapSort

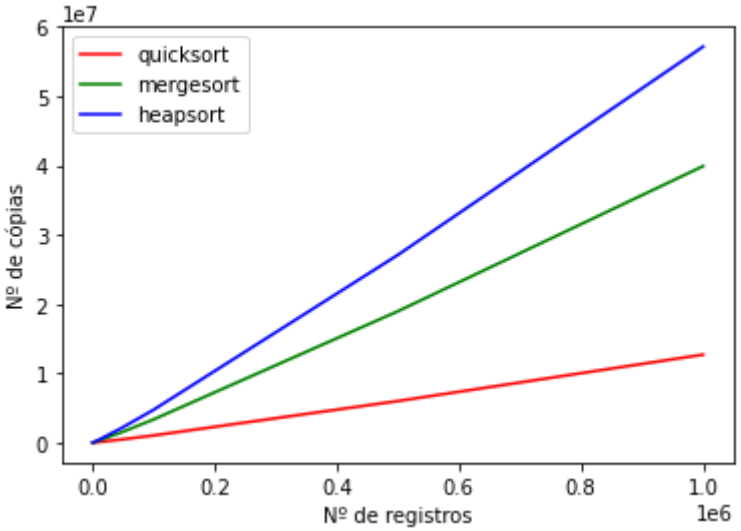
A segunda parte da análise experimental consiste em comparar a melhor versão do QuickSort – no meu caso, a recursiva com Selection a partir de partições de tamanho 100 – e comparar com os algoritmos MergeSort e HeapSort.



Quanto à métrica de tempo de execução, o QuickSort leva uma vantagem considerável. Por meio da análise do gráfico, é possível perceber que a ordenação de um vetor de 1 milhão de registros tem o mesmo tempo de execução que de um vetor com média de 200 mil registros pelos outros algoritmos.



Na análise do número de comparações, contudo, o QuickSort já é o que realiza um número maior. Isso ocorre pois a versão considerada aplica o SelectionSort a partir de certo momento, o que aumenta muito a quantidade de comparações realizadas.



Pela última métrica, o número de cópias, vemos mais uma vez uma larga vantagem do QuickSort em relação aos outros algoritmos, uma vez que ele não utiliza memória adicional para ordenar os registros.

Assim, é possível concluir que o QuickSort, apesar da sua desvantagem na métrica de número de comparações, ainda assim é o algoritmo de ordenação que se apresenta mais eficiente, uma vez que sua desvantagem nessa métrica é compensada por uma ampla vantagem nos outros dois critérios.

6. Conclusões

O trabalho prático 2 tratou da implementação de um programa em C++ que implementa diferentes algoritmos de ordenação amplamente difundidos na comunidade de computação. Em seguida, foram realizadas análises dos algoritmos em relação às métricas de tempo de execução, número de cópias de registros e número de comparações de registros, buscando definir qual dos algoritmos é mais ou menos eficiente dado um certo problema de ordenação. Por meio da minha análise, defini que o algoritmo mais eficiente na maioria dos casos estudados foi o QuickSort com seleção a partir de partições de tamanho menor que 100.

O trabalho permitiu entender mais a fundo como funcionam os algoritmos estudados, além de saber quais critérios devem ser utilizados na escolha de um algoritmo de ordenação para aplicar em um projeto.

7. Bibliografia

Slides e códigos da disciplina Estrutura de Dados. Disponibilizados via Moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Como Medir Tempo de Execução com a Biblioteca Chrono do C++. Disponível em: https://youtu.be/YePlZK84yFg. Acesso em 20 nov. 2022.

8. Instruções para compilação e execução

- Acesse o diretório /tp do projeto por meio de um terminal
- Digite o comando **make** para compilar o programa e gerar o arquivo executável **quicksort** na pasta /bin
- Ao executar o comando **quicksort**, o programa é executado com a entrada (assets/entrada_padrao.txt), saída (bin/saida.txt) e arquivo para registro de logs (bin/log.out) padrões.
- Para utilizar arquivos além dos padrões, o usuário pode inserir algumas opções na linha de comando:
    - -i <arquivo_de_entrada> para mudar a entrada
    - -o <arquivo_de_saida> para mudar o local da saída
    - -p <arquivo_de_logs> para mudar onde serão escrito os logs
    - -l para registrar ou não os acessos à memória nos logs
- Para escolher os parâmetros relacionados ao algoritmo desejado, são utilizadas as seguintes opções:
    - -v <número do algoritmo desejado> para escolher qual o algoritmo de ordenação a ser executado
    - -s <semente> para escolher a semente aleatória
    - -k <número inteiro> para escolher o k do quicksort com mediana
    - -m <número inteiro> para escolher o m do quicksort com seleção
- Caso deseje executar os casos de teste utilizados na análise experimental, basta utilizar o comando **make use** no diretório /tp após a utilização do comando **make**.