Trabalho Prático 2

Filipe de Araújo Mendes

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

flipeara@ufmg.br

1. Introdução

Vetores ou arranjos são estruturas de dados que armazenam uma sequência de objetos, todos do mesmo tipo. Isto é, um vetor de números inteiros é uma sequência de números inteiros, por exemplo {2, 1, 3, 6, 4, 1}. Algoritmos de ordenação, por sua vez, são algoritmos que recebem um vetor e ordenam o vetor recebido com base na sua chave. Pegando o exemplo anterior, um algoritmo de ordenação retornaria o vetor {1, 1, 2, 3, 4, 6}.

O trabalho prático 2 consiste na análise do desempenho de variados algoritmos de ordenação, sendo eles o QuickSort comum e 4 de suas variações (QuickSort Mediana, QuickSort Seleção, QuickSort Iterativo e QuickSort Empilha Inteligente), o MergeSort e o HeapSort. Essa análise é feita levando em conta 3 métricas: o número de comparações de chaves dos vetores, o número de cópias de registros e o tempo total gasto pelo algoritmo.

2. Método

2.1. Main.cc

Na hora de iniciar o programa, são passados alguns argumentos, sendo responsabilidade da função main registrá-los e executá-los.

Além disso, a função main fica encarregada de iniciar o memLog, para a análise em tempo de execução do programa, abrir o arquivo de texto que contém os tamanhos dos vetores a serem criados e chamar as funções para a criação dos vetores aleatórios e posteriormente para a ordenação de tais vetores.

2.2. Register (Registro):

O Registro é uma classe com 3 atributos (os dados que ela guarda) e 3 métodos (o que ela faz). Os atributos do Registro são privados, o que significa que só podem ser acessados e alterados pelo próprio Registro, sendo eles uma variável do tipo inteiro key (chave), uma matriz de variáveis

do tipo char de tamanho 15 x 200 e um vetor de variáveis do tipo float de tamanho 10. Os métodos, por sua vez, são públicos, que significa que podem ser chamados por fora da classe Registro, sendo esses o construtor padrão, que inicializa os objetos do tipo Registro, um método getKey, para pegar a chave do registro e um método setKey, para alterar a chave do registro. Os métodos get e set são necessários pelo fato de que o atributo key é privado, assim, é possível alterar e acessar tal atributo de fora da classe.

O Registro é a estrutura básica utilizada neste trabalho, os vetores a serem ordenados são vetores de Registros, os quais serão ordenados pelo valor do atributo key (a chave) de cada um deles. Os outros atributos, apesar de não serem inicializados, ainda ocupam um espaço na memória, totalizando 3044 bytes junto com a chave por Registro.

2.3. QuickSort:

O QuickSort é o primeiro dos métodos de ordenação que será analisado neste trabalho. Serão feitas 5 variações do método QuickSort, uma delas a original e as outras 4 são possíveis melhorias que podem ser adotadas.

2.3.1. Recursive (Recursivo)

Esse é o QuickSort original, o mais simples e sem nenhuma melhoria. O método do QuickSort é baseado na divisão e recursividade para fazer a ordenação de um vetor. A ordem de complexidade de tempo do QuickSort original é O(n*log n) no melhor caso e O(n^2) no pior caso.

No método QuickSort, é selecionado arbitrariamente um elemento do vetor (no trabalho foi escolhido o elemento do meio), que é chamado de pivô, que então é utilizado como base para fazer a ordenação. Todos os elementos menores que o pivô são colocados à esquerda dele, enquanto todos os maiores são colocados à direita. Após esse processo, chamado de partição, tem-se dois sub-vetores, o sub-vetor de elementos menores que o pivô e o sub-vetor de elementos maiores que o pivô. Assim, é feita a mesma coisa recursivamente com cada um desses sub-vetores, até que o vetor esteja ordenado.

A implementação específica neste trabalho dividiu o QuickSort Recursivo em duas funções: a função recursiveOrder (ordena recursivo) e a função partition (partição).

recursiveOrder: chama a função partition para o vetor recebido e chama recursivamente a função recursiveOrder duas vezes, uma para o vetor à esquerda do pivô e outra para o vetor à direita.

partition: executa o processo de partição do método QuickSort. Essa função é utilizada também por outras variações do método QuickSort implementadas.

2.3.2. Median (Mediana)

O QuickSort Mediana difere do original somente na parte de escolha do pivô no processo de partição. Enquanto no original é escolhido um elemento arbitrário para ser o pivô, no Mediana são escolhidos k elementos arbitrários e o pivô é escolhido como a mediana desses k elementos escolhidos. Assim, é garantido que o pior caso do QuickSort nunca ocorrerá, mas ao custo inicial da escolha da mediana, que em casos normais, torna essa variação do QuickSort mais lenta que o original.

A implementação específica neste trabalho dividiu o QuickSort Mediana em duas funções: a função medianOrder (ordena mediana) e a função medianPartition (partição mediana).

medianOrder: quase igual à recursiveOrder, com a diferença de chamar a medianPartition ao invés da partition.

medianPartition: cria um vetor de k elementos, ordena esse vetor e seleciona a mediana (elemento do meio) como pivo para depois executar normalmente o resto do processo de partição do método QuickSort.

2.3.3. Selection (Seleção)

Para entender o QuickSort Seleção, é necessário primeiro entender o SelectionSort. Esse algoritmo de ordenação consiste em varrer o vetor inteiro procurando o maior ou menor elemento, para depois colocá-lo em sua posição. Depois disso é feito a mesma coisa com o segundo, terceiro, assim sucessivamente, até que o vetor esteja ordenado. Esse algoritmo tem custo O(n^2) sempre, dessa forma ele é bem menos eficiente que o QuickSort para vetores grandes.

Agora é possível entender o QuickSort Seleção. A versão original do QuickSort, para ordenar vetores muito pequenos, possui uma grande ineficiência por possuir várias chamadas recursivas, que são computacionalmente caras. Dessa forma, para vetores de tamanho suficientemente pequeno, algoritmos comumente mais ineficientes, como o SelectionSort acabam sendo mais eficientes que o QuickSort por não possuir tais chamadas recursivas. Assim, o QuickSort Seleção utiliza dessa propriedade para fazer com que, a partir de um certo tamanho m, o vetor será ordenado pelo SelectionSort, eliminando várias chamadas recursivas e fazendo com que essa versão do QuickSort seja mais rápida que a original.

A implementação específica neste trabalho dividiu o QuickSort Seleção em duas funções: a função selectionOrder (ordena seleção) e a função partition (partição).

selectionOrder: quase igual à recursiveOrder, com a diferença de verificar se os sub-vetores possuem tamanho menor ou igual a m. Se sim, é feito o SelectionSort, se não, é chamado recursivamente o selectionOrder.

partition: já apresentado na parte do QuickSort Recursivo.

2.3.4. Iterative (Iterativo)

Essa variação é um QuickSort não-recursivo, que visa remover a pilha de recursão e fazer tudo por meio de uma classe pilha.

Resumidamente, a classe Stack (pilha) é uma estrutura de dados que armazena elementos em um formato sequencial, como vetores, mas com a diferença que novos elementos sempre são inseridos no topo da pilha e é sempre retirado o elemento que está no topo. Os elementos armazenados são da classe Node (nó), que guardam as posições de inicio e fim dos sub-vetores no vetor original.

Essa classe Stack é utilizada para simular a pilha de recursão do algoritmo QuickSort original, sem ter, de fato, a recursão. Por esse motivo, o tempo de execução dessa versão em comparação com o do original, é o mesmo.

A implementação específica neste trabalho dividiu o QuickSort Iterativo em duas funções: a função iterative (iterativo) e a função partition (partição).

iterative: inicializa a pilha, coloca as posições do vetor original na pilha, e vai chamando partition para empilhar os sub-vetores e desempilhando até que o vetor esteja ordenado.

partition: já apresentado na parte do QuickSort Recursivo.

2.3.5. Smart Stack (Empilha Inteligente)

Essa variação é quase igual ao QuickSort Iterativo, mas com a pequena diferença de que ele sempre processa o menor dos sub-vetores primeiro, enquanto o Iterativo processa sempre o sub-vetor da esquerda.

2.4. MergeSort:

Parecido com o método QuickSort, mas diferente. O MergeSort faz a divisão recursiva do vetor até que o tamanho dos sub-vetores seja 1 e então vai juntando os sub-vetores novamente, ordenando-os enquanto junta. Por exemplo, o vetor {6, 1, 3, 5} seria dividido em {6}, {1}, {3} e {5}, para então

fazer a primeira junção, formando os vetores {1, 6} e {3, 5} e após a segunda junção teria o vetor ordenado {1, 3, 5, 6}.

A complexidade de tempo do MergeSort é O(n*log n), não possuindo pior caso, ou seja, há casos em que o QuickSort seja pior, mas devido à várias cópias de vetor que o MergeSort faz, o caso normal é mais lento que o QuickSort, além de gastar mais memória.

2.5. HeapSort:

O HeapSort é parecido com o SelectionSort, mas aplicado a um Heap. A característica principal do Heap é que um pai de índice i é, necessariamente, maior que nenhum de seus filhos, com índice 2i e 2i + 1. Dessa forma, o método HeapSort consiste em pegar o primeiro elemento do Heap, que é, necessariamente, o maior de todos, colocar em último lugar do vetor, que é a posição correta de tal elemento, e refazer o Heap, repetindo esse processo até que o vetor esteja ordenado. Por exemplo, o vetor {1, 3, 2} seria transformado em um Heap, ficando {3, 1, 2}, para assim trocar de lugar o 3 com o dois, formando {2, 1, 3}, para reformar o Heap em {2, 1}, que já está feito, para então trocar o 2 com o 1 e formar o vetor ordenado {1, 2, 3}.

Assim como o MergeSort, o HeapSort tem complexidade de tempo O(n*log n), sendo abaixo do pior caso do QuickSort, gastando a mesma memória do QuickSort.

2.6. Funções Auxiliares:

getExtra: verifica qual o QuickSort escolhido, se for o Mediana, define o valor da variável extra como igual à variável k, se for o Seleção, define como igual à variável m.

defaultOutputName: cria um nome para o arquivo de saída baseado no método e semente utilizados.

fHeader: cria o cabeçalho do arquivo de saída.

fBody: cria o corpo do arquivo de saída.

randArray: cria um vetor aleatório a partir da semente fornecida.

getMean: retorna a média de n valores.

quickSort: chama a variante do QuickSort escolhida.

sort: ordena 5 vetores diferentes com o método selecionado e retorna a média de comparações, cópias de registro e tempo da execução das ordenações.

swap: troca 2 elementos do vetor.

endWith: verifica se uma string termina com a extensão fornecida.

2.7. Contagem das Métricas para Comparação:

A contagem do número de comparações de chave e número de cópias de registro foi feita por meio da utilização de variáveis globais acessíveis e modificáveis em todo o programa. A função swap contou como 3 cópias de registro por precisar de uma variável auxiliar.

A contagem do tempo foi feita chamando a função Timer, providenciada no enunciado do TP, antes e depois de ordenar, fazendo a diferença entre os dois.

3. Análise de Complexidade

3.1. Espaço

QuickSort Recursivo: utiliza somente o vetor original, sendo então O(n).

QuickSort Mediana: utiliza 2 vetores, o original e um de tamanho k, sendo então O(n+k).

QuickSort Seleção: igual ao QuickSort Recursivo.

QuickSort Iterativo: utiliza o vetor original e uma pilha cujo tamanho varia em relação a log n, assim, a complexidade continua sendo O(n).

QuickSort Empilha Inteligente: igual ao QuickSort Iterativo.

MergeSort: o espaço utilizado pelo algoritmo implementado possui um vetor v durante toda a execução, com outras alocações de vetores de tamanhos que chegam até o tamanho de v, assim, o espaço máximo é 2 vezes o tamanho do vetor, sendo então O(n).

HeapSort: o método HeapSort armazena somente um vetor de tamanho n, que é alterado durante a execução do algoritmo, sendo então O(n).

3.2. Tempo

Todas as variações do QuickSort possuem a mesma complexidade de tempo, por realizar as operações de maneira semelhante, mas alguns são mais rápidos ou mais lentos que outros.

Em melhor caso, o método de dividir e conquistar do QuickSort sempre divide o vetor na metade, chamando recursivamente um total de log n vezes, combinando com a ordenação, que é O(n). Dessa forma, o melhor caso possui complexidade O(n*log n).

O pior caso do QuickSort ocorre quando o pivô é sempre o maior ou menor elemento daquele sub-vetor, dividindo então em sub-vetores de tamanho 1 e n - 1. Dessa forma, o custo do QuickSort é:

$$T(n) = n + T(n - 1)$$

Assim, o pior caso é O(n^2). Somente o QuickSort Mediana possui um pior caso melhor que esse, mas ainda assim, com um custo ainda parecido.

Já os métodos MergeSort e HeapSort, ambos possuem complexidade fixa em O(n*log n). O MergeSort por fazer no máximo log n divisões do vetor e o HeapSort por simular, com o heap, o funcionamento de uma árvore binária de altura log n.

4. Estratégias de Robustez

Para evitar a interrupção desnecessária do programa, foi adicionada uma função para checar se os argumentos contendo o nome de entrada do programa contém a extensão do arquivo requerida pelo programa, não interrompendo a execução se esse não for o caso.

Outra forma foi a implementação de valores padrão para variáveis. A função defaultOutputName cria automaticamente um nome para o arquivo de saída caso o usuário não declarar, caso o método não seja definido, o programa executa todos os métodos e caso as variáveis k e m não sejam definidas, o programa utiliza, respectivamente, 3 e 100, que foram os melhores casos dos respectivos QuickSorts. A semente padrão também foi criada, com valor 1.

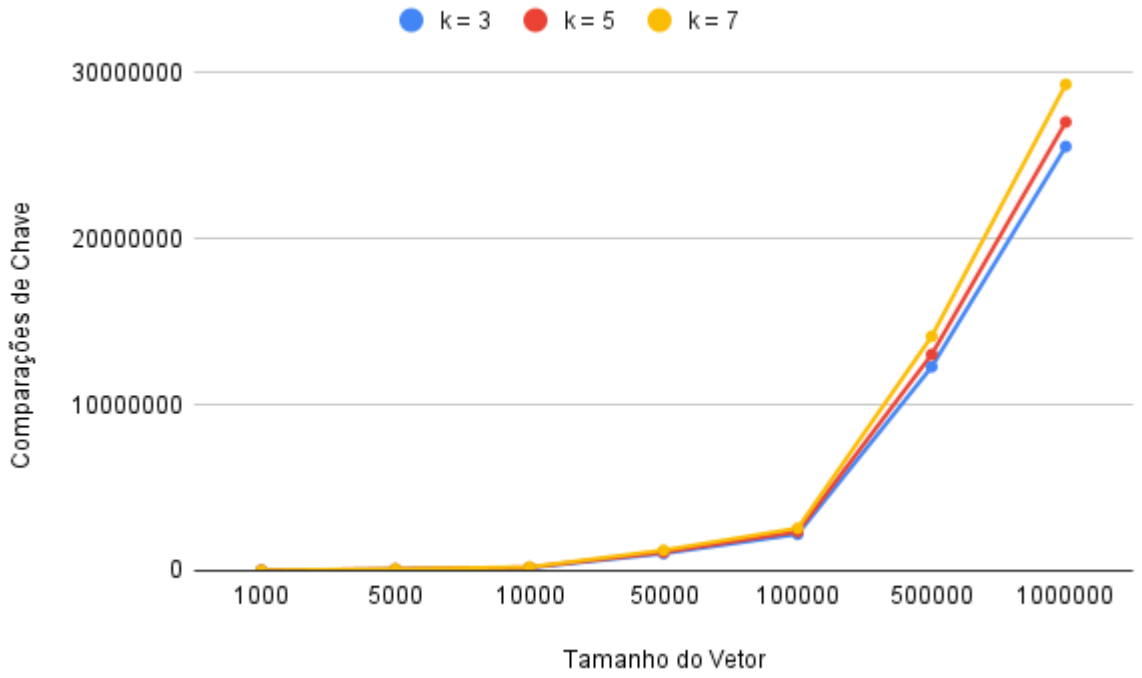
5. Análise Experimental

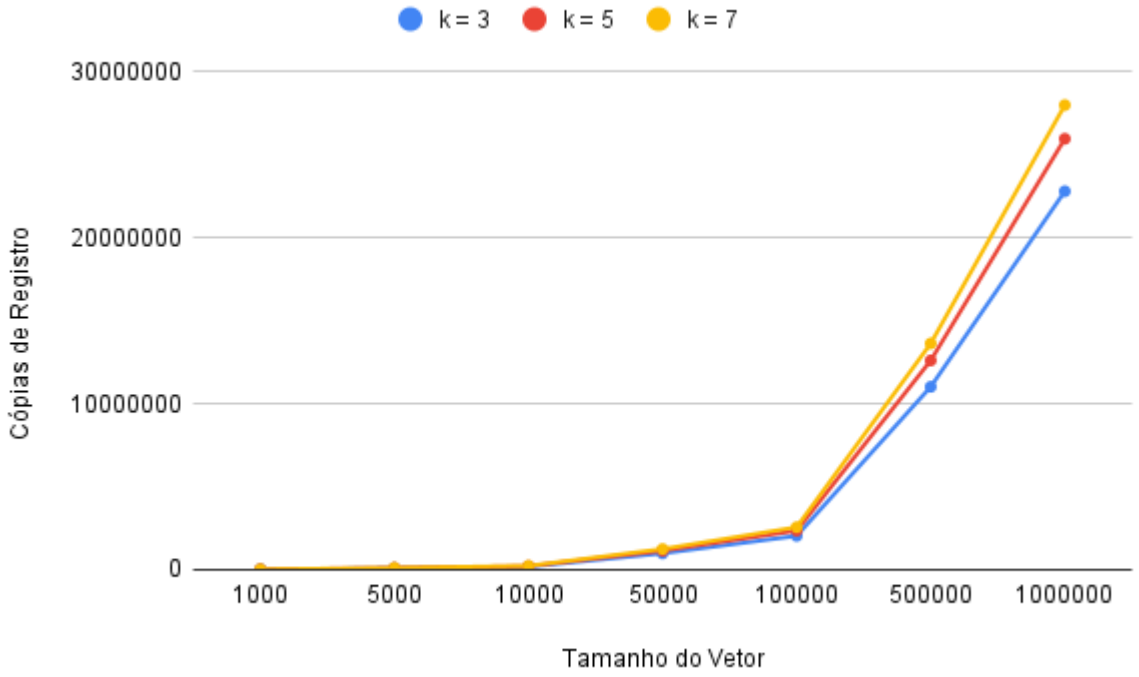
Foram analisados os diferentes métodos de ordenação, assim como variações nos valores das variáveis k e m, para o QuickSort Mediana e QuickSort Seleção, respectivamente.

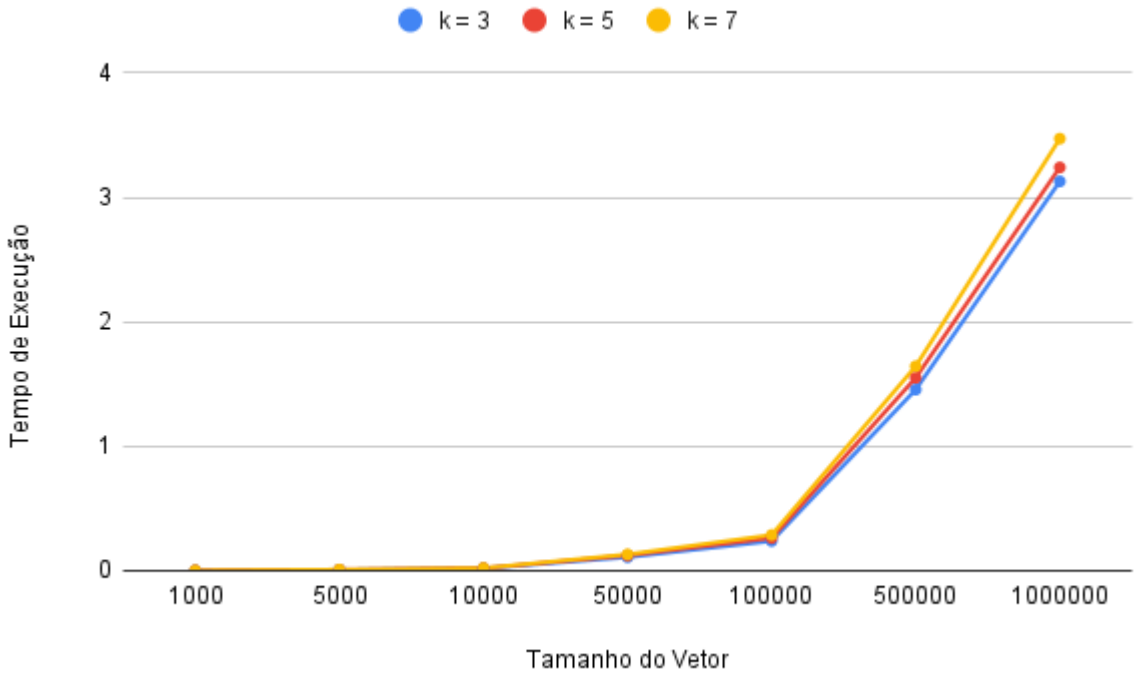
5.1. QuickSort Mediana com k = 3, 5, 7.

O QuickSort Mediana, por calcular a mediana do vetor por meio da ordenação do vetor de medianas com o SelectionSort, aumenta as comparações de chave e cópias de registro com o aumento de k.

Por causa desse aumento nas comparações de chave e de cópias de registro, o tempo de execução também aumenta, fazendo com que o QuickSort Mediana mais eficiente, em geral, seja o com k = 3.

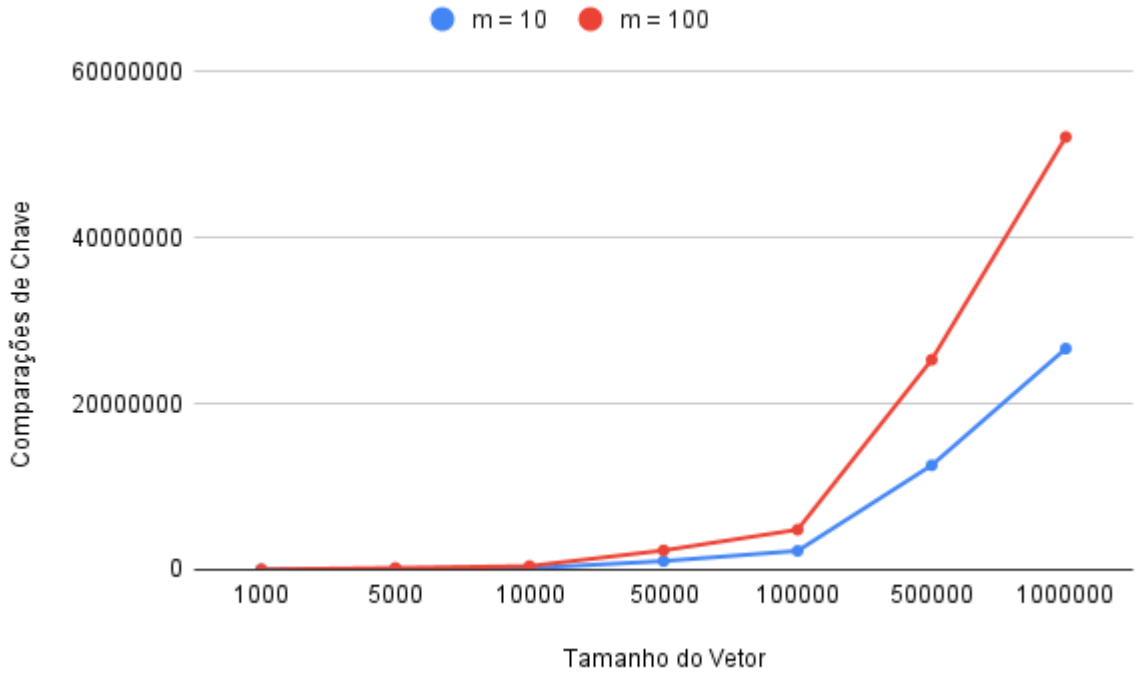

k = 3
k = 5
k = 7
30000000
20000000
10000000
0
Comparações de Chave
1000
5000
10000
50000
100000
500000
1000000
Tamanho do Vetor

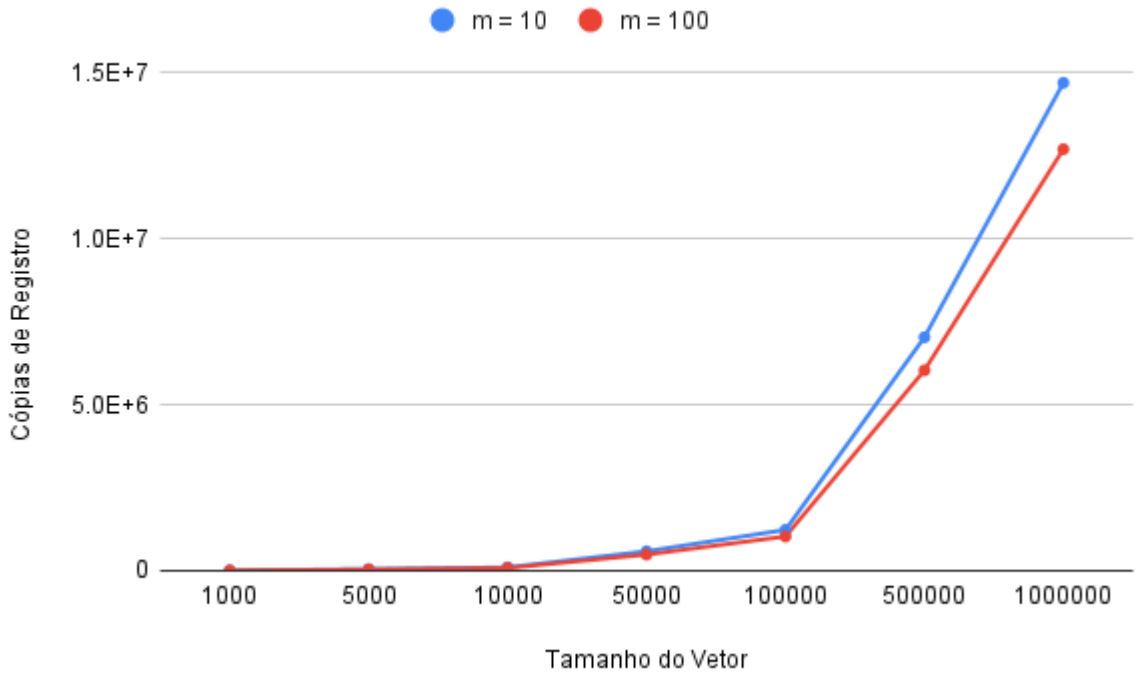

k = 3
k = 5
k = 7
30000000
20000000
10000000
0
Cópias de Registro
1000
5000
10000
50000
100000
500000
1000000
Tamanho do Vetor



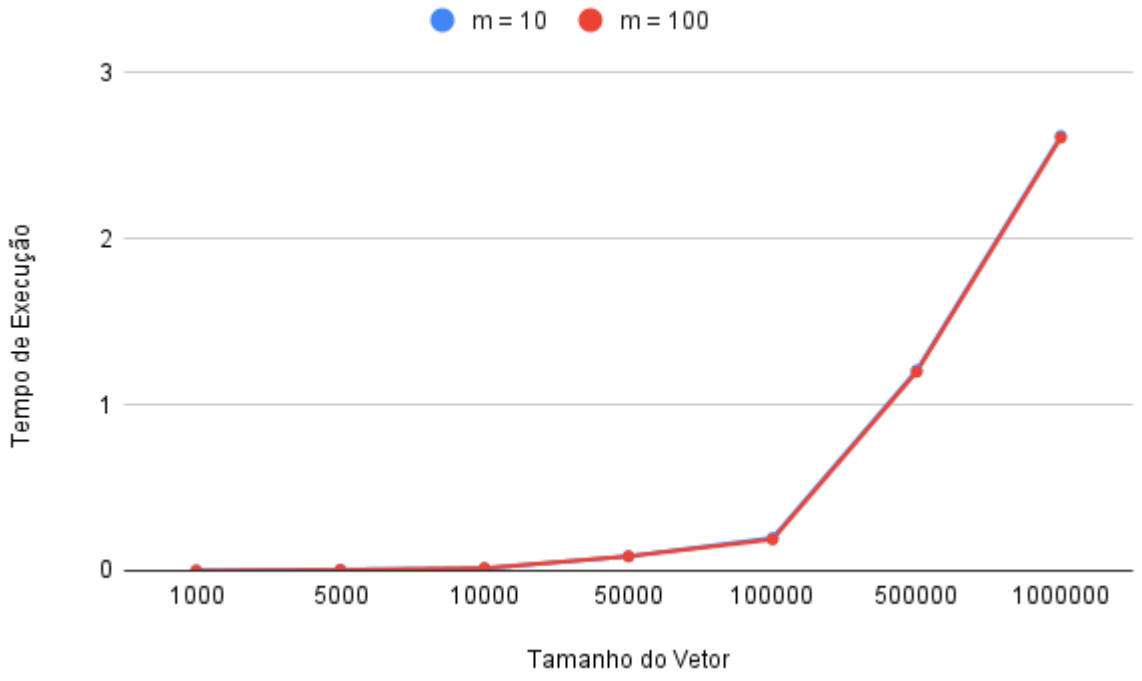
5.2. QuickSort Seleção com m = 10, 100

O QuickSort Seleção, ao alterar o valor de m, utiliza o SelectionSort com vetores de tamanhos variados. O SelectionSort, com maiores tamanhos, realiza bem mais comparações de chave que o QuickSort, mas realiza menos cópias de registro.





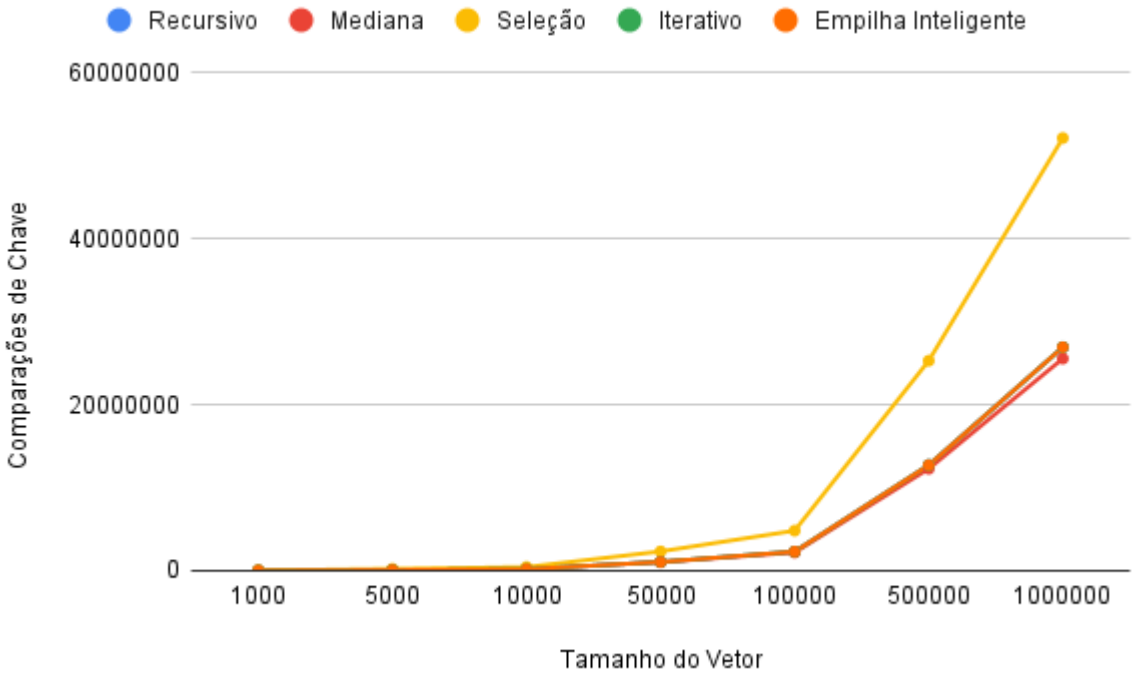
Em questão de tempo, a redução ainda maior da chamada recursiva com m = 100, junto com a redução nas cópias de registro, foi suficiente para fazer com que o QuickSort Seleção com m = 100 fosse um pouco mais rápido que o com m = 10 (apesar da dificuldade de se observar no gráfico, o com m = 100 foi 0.01 seg mais rápido que o com m = 10 para o maior tamanho), mas com um valor ainda maior de m, teria uma redução na eficácia dessa melhora, fazendo com que o custo do SelectionSort não compense as chamadas recursivas não feitas.



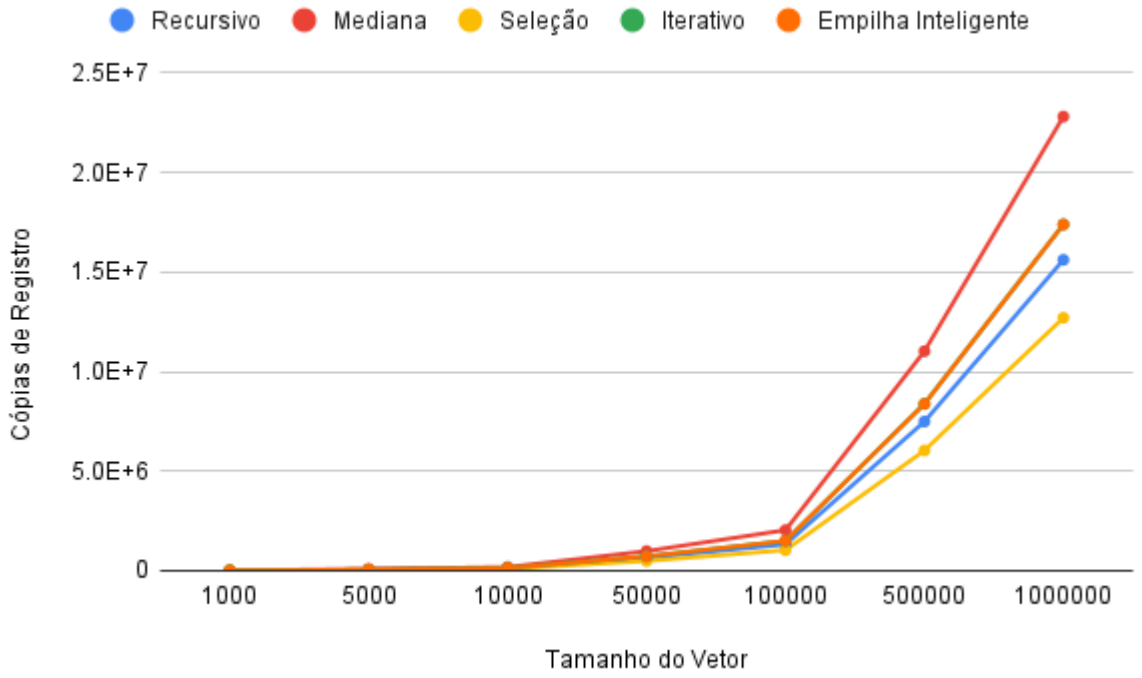
5.3. Variantes dos QuickSorts

A análise geral para comparar qual o melhor QuickSort, utilizando as mesmas métricas e utilizando o QuickSort Mediana com k = 3 e o QuickSort Seleção com m = 100, que foram as variações mais eficientes das testadas.

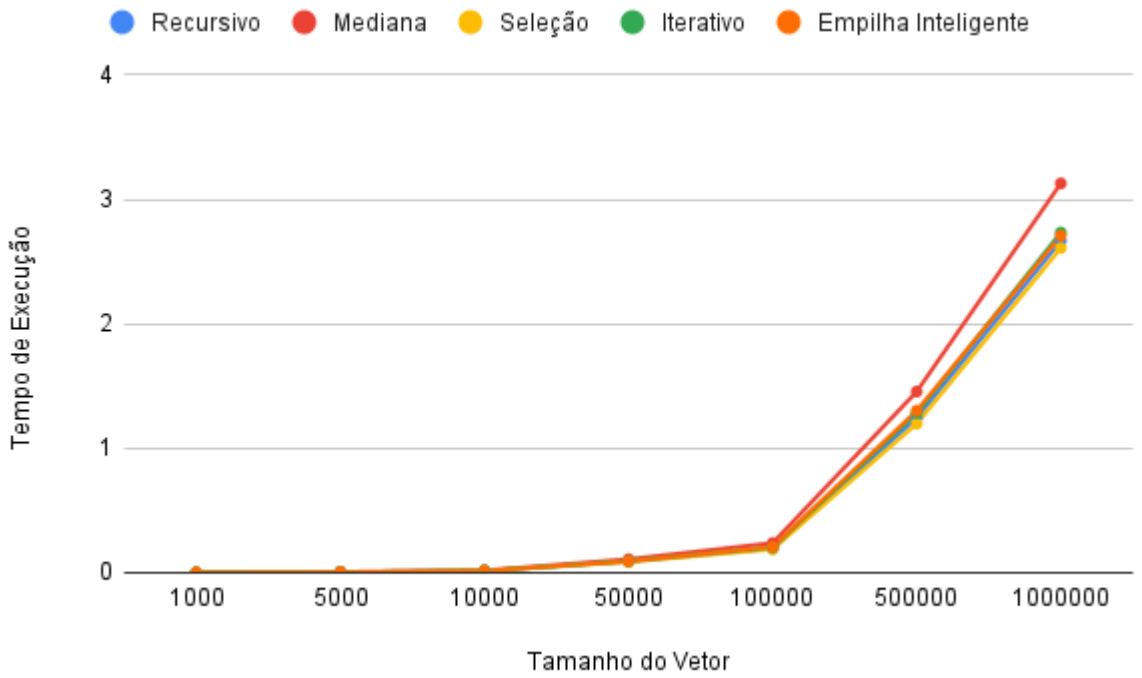
Comparações de Chave: o Seleção é o QuickSort com a maior quantidade de comparações, por utilizar o método SelectionSort, que faz muito mais comparações que o QuickSort, seguido do Iterativo, Empilha Inteligente e Recursivo que fazem exatamente o mesmo número de comparações e então o Mediana, que consegue fazer menos comparações por dividir o vetor em subvetores de tamanhos mais próximos que os outros.



Cópias de Registro: o Mediana é o QuickSort com a maior quantidade de cópias de registro, por criar vetores mediana tamanho 3 a cada chamada da função medianPartition, seguido do Iterativo e Empilha Inteligente, que fazem exatamente o mesmo número de cópias de registro, acima do Recursivo por fazer uma cópia de registro na hora de empilhar e na hora de desempilhar um elemento. Com o menor número de cópias de registro está o QuickSort Seleção, por causa da quantidade reduzida de cópias que o SelectionSort faz em relação ao QuickSort.



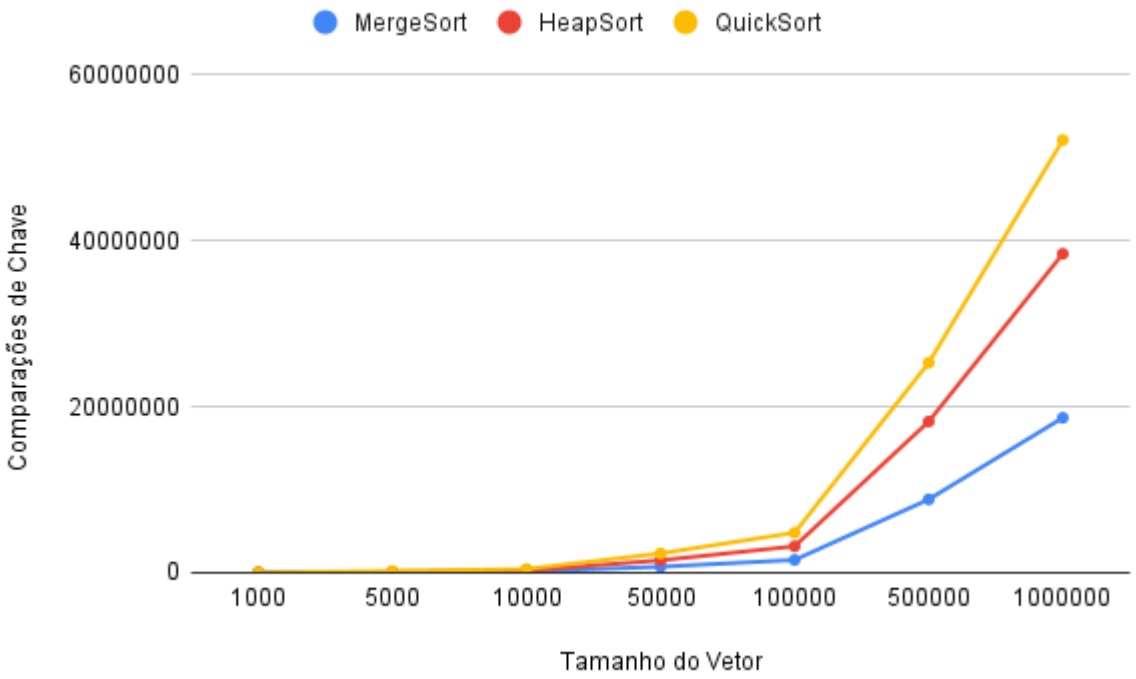
Tempo de Execução: o QuickSort mediana é o mais lento por fazer várias cópias de registro e ordenação dos vetores mediana, mas com certos vetores, possui o potencial de ser o mais rápido dos 5, por evitar o pior caso. O segundo mais lento é o Iterativo, seguido do Empilha Inteligente, que são mais lentos que o Recursivo por otimizações na hora de colocar na pilha recursiva em comparação com a pilha dos iterativos, mas com uma diferença pequena. O QuickSort Seleção é mais rápido que o Recursivo, por possuir menos chamadas recursivas e menos cópias de registro, sendo então o QuickSort mais eficiente dos 5.



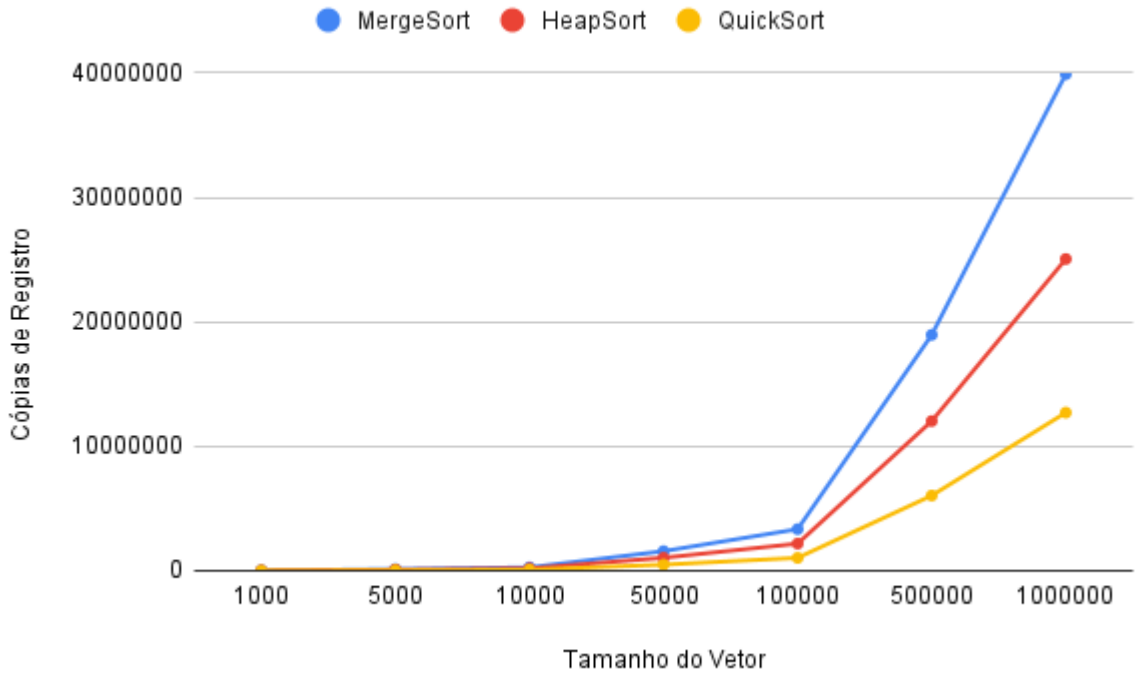
5.4. QuickSort, MergeSort e HeapSort

O variante do QuickSort escolhida para essa análise foi o QuickSort Seleção com m = 100, já que foi a variante mais rápida da análise passada.

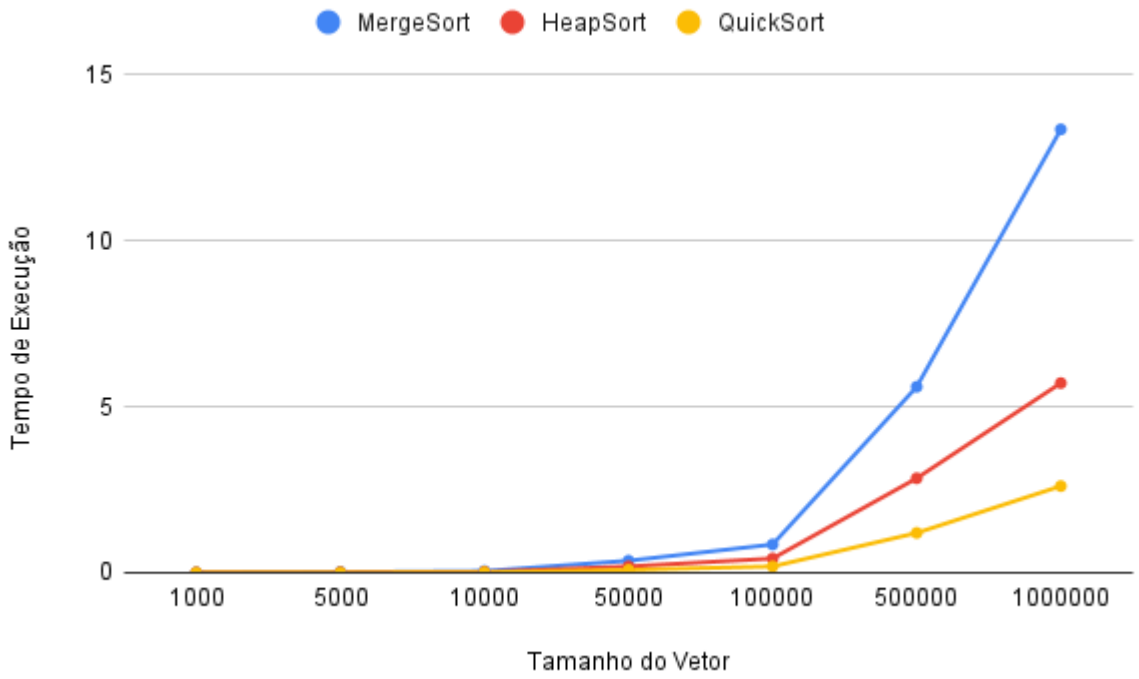
O MergeSort é o algoritmo com o menor número de comparações de chave, por realizá-las somente na hora de juntar os vetores. O maior número é feito pelo QuickSort Seleção, pelo SelectionSort possuir uma grande quantidade de comparações de chave. No meio se encontra o HeapSort, que seria maior que outras variantes do QuickSort por ser necessário refazer o Heap a cada troca, sempre fazendo várias comparações.



O MergeSort, na questão de cópias de registro, é o algoritmo mais custoso, por alocar novos vetores e copiar partes do vetor original dele a cada execução da função merge. Então tempos o HeapSort, com um número maior de cópias que o QuickSort Seleção novamente pela necessidade de refazer o Heap a cada vez. A economia nos custos de cópias de registro do SelectionSort faz com que o QuickSort Seleção tenha os menores números de cópias de registro entre todos eles.



Em última métrica, temos o tempo de execução dos algoritmos. O processo de cópia desses vetores, que ocupam até GigaBytes de memória, acaba sendo um processo extremamente caro, fazendo com que o MergeSort seja o mais lento dentre todos eles. Depois, temos o HeapSort que, novamente por ter que refazer o Heap, também é prejudicado na questão do tempo. O QuickSort acaba sendo o mais rápido nesse caso, mas lembrando que com certos vetores ele possui o potencial de ser bem mais lento, por possuir um pior caso, coisa que não existe nos outros dois métodos.



5.5. Análise da Distância de Pilha

Não foi possível realizar a análise da distância de pilha dos métodos de ordenação, devido a um erro durante a execução do analisamem:

```
bin/analisamem -i teste/log.out -p out/log
Enderecos:  [94792733856496-139901836998972] (45109103142476) #end 5638637892811
Fases: [0-0] (0) #fase 1
Ids: [0-0] #id 1
src/pilhaindexada.c:48: Erro 'p->pilha != NULL' - nao foi possivel alocar p->pilha
make: *** [Makefile:29: use] Aborted
```

Foram feitas várias tentativas para resolver esse problema, inclusive recorrendo a colegas de turma que obtiveram sucesso no uso da biblioteca. Examinei também a documentação do analisamem, mas não estava presente em nenhum lugar menções desse erro, muito menos qualquer instrução para corrigi-lo.

Assim, após várias horas tentando transformar o arquivo log.out gerado na execução do programa em gráficos para fazer a análise da distância de pilha, com tantas tentativas frustradas, finalmente desisti deste tipo de análise no trabalho.

6. Conclusões

O trabalho consistiu em implementar vários métodos de ordenação e analisar o desempenho de cada um deles.

Por meio das variantes do método QuickSort implementadas e analisadas, é possível perceber que os diferentes tipos possuem seus casos de uso específicos e que cada melhoria, apesar de melhorar algumas das métricas, possui seu ponto forte, sem que nenhuma delas seja simultaneamente a melhor nas três métricas observadas.

O QuickSort Mediana pode ser utilizado quando se deseja evitar a qualquer custo, o pior caso do algoritmo QuickSort, com o efeito de reduzir o número de comparações de chave na execução do método. O QuickSort Seleção pode ser utilizado quando se deseja um algoritmo mais rápido, que também realiza menos cópias de registro pela utilização do SelectionSort. Os dois QuickSorts iterativos podem ser utilizados quando é desejável remover a pilha de recursão, sendo ainda bem eficientes. A diferença entre os dois não foi explicitada pela incapacidade de ser feita a análise da distância de pilha, que proporcionaria um melhor entendimento dos dois. Por último, o QuickSort original, como observado, ainda é um algoritmo eficiente, não ficando muito atrás do Seleção e obtendo métricas médias nos testes.

A comparação entre os métodos QuickSort, MergeSort e HeapSort também serviu para mostrar ainda melhor a eficiência geral do método QuickSort, que acabou sendo muito mais rápido que os outros em tamanhos maiores dos vetores. Ainda assim, é sempre necessário lembrar que o

método QuickSort, ao contrário dos outros dois citados, possui pior caso, com complexidade O(n^2), sendo essa igual à complexidade de um método mais simples, o SelectionSort. Por isso, em certos casos é preferível utilizar métodos menos eficientes em geral, como o MergeSort e o HeapSort.

7. Bibliografia

Para funções padrão de c++, foi utilizado como base o link:
https://cplusplus.com/forum/

Para detalhes sobre o uso do analisaMem, foi utilizado o link:
https://docs.google.com/document/d/1h-yeWDz2FjBVvMgI1gfXyV_ogOmwlpneKEfq4yHihoA/edit

Para as implementações dos métodos de ordenação foram utilizados os slides no moodle, assim como o site:
https://www.geeksforgeeks.org/

8. Instruções para Compilação e Execução

Para a compilação do programa em sistema Linux, primeiro é necessário abrir uma janela do terminal dentro da pasta TP. Após feito isso, basta digitar a linha de comando “make” e o programa será compilado.

Para sua execução em ambiente Linux, é necessário também abrir a janela do terminal na pasta TP, para então digitar a linha de comando “./bin/run.out ARGUMENTOS”, onde “ARGUMENTOS” são os argumentos a seguir, somente o último sendo opcional:

- “-i entrada” ou “-i entrada.txt” - O argumento “-i” é seguido do nome do arquivo de texto que se deseja ler, que deve estar no diretório TP, com ou sem o “.txt”, que indica a extensão.
  Se esse argumento for omitido, o programa tenta abrir, por padrão, um arquivo “input.txt” no diretório TP. Caso o arquivo não exista, o programa produz um erro de arquivo não encontrado.

- “-v METODO” - O argumento “-i” é seguido do método de ordenação que se deseja utilizar, sendo este escolhido por um número de 1 a 7 que corresponde ao:

  1. QuickSort Recursivo.
  2. QuickSort Mediana.
  3. QuickSort Seleção.
  4. QuickSort Iterativo.
  5. QuickSort Pilha Inteligente.
  6. MergeSort.
  7. HeapSort.

  Se esse argumento for omitido, o programa executa, por padrão, com todos os métodos de ordenação, colocando, no arquivo de saída, todos os métodos juntos.

- “-k NUMERO” - O argumento “-k”, relevante somente para o QuickSort Mediana, define o tamanho do vetor mediana que o usuário deseja nesse tipo de QuickSort, sendo este o número passado pelo usuário.
  Se esse argumento for omitido, o tamanho do vetor mediana, por padrão, é 3.

- “-m NUMERO” - O argumento “-m”, relevante somente para o QuickSort Seleção, define o tamanho mínimo do vetor em que, a partir desse tamanho, será feito o SelectionSort do vetor, sendo este o número passado pelo usuário.

Se esse argumento for omitido, o tamanho mínimo, por padrão, é 100.

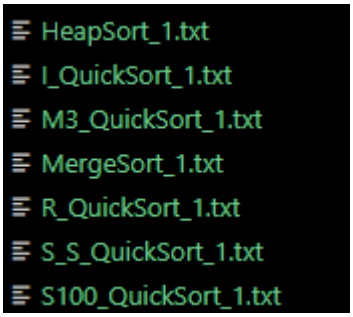
- .”-s NUMERO” - O argumento “-s” define a semente utilizada para gerar o vetor aleatório no programa.
  Se esse argumento for omitido, a semente do programa, por padrão, é 1.

- “-l” - O argumento “-l” deve ser adicionado somente se o usuário desejar que seja registrado os acessos à memória durante a execução do programa, caso contrário, será registrado apenas o tempo de execução do programa.

- “-o saida” ou “-o saida.txt” - O argumento “-o” define o nome do arquivo de saída criado pelo programa, com ou sem o “.txt”, que indica a extensão.
  Se esse argumento for omitido, o nome do arquivo de saída é gerado automaticamente baseado no método de ordenação, no k, no m e na semente utilizados para o teste no formato “METODO(K, M)_SEMENTE.txt”.

  “R_QuickSort_1.txt” indica que foi utilizado o método recursivo com semente = 1.
  “M3_QuickSort_1.txt” indica que foi utilizado o método mediana com k = 3 e semente = 1.
  “S100_QuickSort_1.txt” indica que foi utilizado o método seleção com m = 100 e semente = 1.
  “I_QuickSort_1.txt” indica que foi utilizado o método iterativo, com semente = 1.
  “S_S_QuickSort_1.txt” indica que foi utilizado o método empilha inteligente com semente = 1.
  “MergeSort_1.txt” indica que foi utilizado o MergeSort com semente = 1.
  “HeapSort_1.txt” indica que foi utilizado o HeapSort com semente = 1.

  Além disso, se o argumento “-v” também for omitido, o programa cria um arquivo de saída para cada método testado, com os nomes no formato descrito acima.



Exemplo de arquivos criados com a linha de comando:
./bin/run.out
(Necessário que exista o arquivo input.txt no diretório TP)

```
Recursive QuickSort

Array Size    Mean Key Comparisons    Mean Register Copies    Mean Execution Time
    1000                   13251                    8706              0.0006306
    5000                   83907                   51363              0.0055348
   10000                  181325                  109728              0.0133184
   50000                 1051512                  630390              0.0895146
  100000                 2298078                 1322027               0.194942
  500000                12727623                 7490210                1.24463
 1000000                26933347                15614249                2.66678
```

Exemplo de arquivo de saída.