Trabalho Prático 2
Análise de algoritmos de ordenação

Estrutura de Dados

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)

Lorenzo Carneiro Magalhães

lorenzocarneirobr@gmail.com

Belo Horizonte - MG - Brasil

1. Introdução

O Trabalho teve como objetivo analisar alguns dos métodos de ordenação, classificando-os e apresentando resultados práticos acerca do desempenho dos algoritmos. Essa análise teve como foco o algoritmo de ordenação Quicksort, que é um dos mais utilizados atualmente devido à sua conhecida rapidez. Um ponto importante do projeto foi a procura de métodos de otimização para o quicksort tradicional.

2. Método

Todos os algoritmos de ordenação foram implementados em um único arquivo, o "*sort.hpp"*, e todos foram submetidos a análises similares utilizando diversas ferramentas, como a ferramenta *time*, a função *getrusage* e as bibliotecas *memlog* e *analisamem*.

Os algoritmos foram executados em um computador com 16gb RAM, processador i7-1165G7 2.80GHz com frequência máxima de 4.10GHz (64 bits CPU), no sistema operacional Kubuntu 22.04 LTS.

A implementação foi realizada em C++ utilizando o compilador G++.

3. Análise de Complexidade

Já é amplamente conhecidas as complexidades de cada algoritmo:

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)

Imagem retirada de https://github.com/tarcisio-marinho/sorting-algorithms

Com exceção do *Timsort* e do *Bubblesort*, que não foram implementados neste trabalho, todos os algoritmos seguiram uma complexidade de tempo e espaço esperados, conforme mostrado na imagem.

Começando pelos algoritmos com maior complexidade, temos o Selection Sort e o Insertion Sort. O primeiro sempre performa em $O(n^2)$, isso porque sempre realiza um número determinado de operações para achar o menor termo não ordenado e colocá-lo no lugar correto, de modo a percorrer por n-1 vezes o vetor, dado que *n* é o tamanho do vetor.

Já o Insertion Sort possui um melhor caso linear, que ocorre quando o vetor está ordenado ou praticamente ordenado. Para o restante dos casos, a complexidade é

O(n²), pois é necessário percorrer $n - i - 1$ termos a cada iteração do valor *i*, que vai de 1 até o tamanho *n*, que é o tamanho do vetor.

Passamos agora, portanto, para algoritmos mais eficientes. O Quicksort é um algoritmo do tipo dividir para conquistar e possui seu caso médio O($n \log n$) e seu pior caso O(n^2), que ocorre quando o pivô do algoritmo é um termo muito pequeno ou muito grande em relação aos valores do vetor. Isso ocorre porque as divisões não são eficientes, o que significa deixar muitos valores de um lado e poucos valores de outro.

O Mergesort também é considerado um algoritmo eficiente e do tipo dividir para conquistar, no entanto, diferentemente do quicksort, possui complexidade em todos os casos de O($n \log n$). Isso ocorre porque o mergesort sempre divide o vetor pela metade, de modo a desconstruí-lo até obter vetores de tamanho 1, que serão reconstruídos em ordem até obter o vetor de tamanho *n* ordenado.

Por fim, tem-se o Heapsort, que é um algoritmo muito interessante baseado na estrutura de heap. Essa estrutura se baseia na estrutura de árvore binária e possui um diferencial: os filhos de um nodo são sempre menores do que o pai. Desse modo, é possível estabelecer relações específicas nessa estrutura. Aproveitando disso, o algoritmo heapsort constrói um heap a partir do vetor previamente obtido e retira o maior termo e coloca-o no final do vetor. Após isso, o algoritmo reorganiza o heap levando o maior termo novamente para o topo e novamente o retira, de modo a repetir esse processo até o vetor estar completo e ordenado

4. Estratégias de robustez

O programa conta com algumas estratégias de robustez, como checagens de parâmetros, mas não garante que um eventual usuário não receba erros.

Entre as checagens, estão: validação do tamanho máximo do vetor, checagem da passagem de parâmetros obrigatórios e a validação dos parâmetros passados.

Essas checagens previnem a maior parte dos eventuais erros, mas também foram implementadas outras checagens relacionadas aos algoritmos de ordenação.

5. Análise Experimental

Será feita uma seção para cada algoritmo, começando do menos eficiente para o mais eficiente conforme análises pessoais. As comparações terão como base o quicksort.

Uma nota importante é que apesar de alguns algoritmos possuírem a mesma complexidade, o custo computacional pode ser drasticamente diferente.

Ademais, é necessário ressaltar que a implementação dos algoritmos não é, teoricamente, perfeita e pequenas otimizações são provavelmente possíveis.

Uma tabela com os respectivos tempos de execução de cada algoritmo será mostrada ao final, assim como os gráficos de complexidade de espaço e suas respectivas análises.

É importante ressaltar que a análise de padrão de acesso à memória não foi realizada com todos os algoritmos implementados.

5.1 Selection Sort

Esse algoritmo possui complexidade de $O(n^2)$ em todos os casos. No entanto, apesar da complexidade alarmante, o algoritmo performa relativamente bem em casos em que o número de elementos do vetor é pequeno.

5.2 Insertion Sort

Também foram feitos testes com o insertion sort de modo a compará-lo com o algoritmo citado acima. A partir dos resultados foi possível concluir que ambos algoritmos performam com um custo similar, de modo que o insertion sort performa

levemente melhor quando o vetor é muito pequeno e pior à medida em que o vetor aumenta de tamanho.

5.3 Mergesort

Entrando em uma categoria de algoritmos mais eficiente, com custo de *n * log n*, temos o Mergesort, que impressionou negativamente. O algoritmo foi o mais lento dessa categoria, além de ocupar muita memória e não ser tão rápido de implementar.

Esse fato pode ter ocorrido devido ao grande número de comparações e cópias do programa, que correspondem a 183 e 177, respectivamente, para um vetor de 15 termos.

Foi necessário implementá-lo utilizando alocação dinâmica para testá-lo com valores maiores, sendo bem visível, portanto, a pouca eficiência de memória do algoritmo, que corresponde a *n*.

5.4 Heapsort

O Heapsort é um algoritmo com a mesma complexidade do Mergesort, mas que superou em todos os aspectos o algoritmo anterior: mais fácil de ser implementado, mais rápido, e ocupa *log n* de memória.

Ainda sim, o algoritmo demora consideravelmente, de modo a realizar 141 cópias e 248 comparações para um vetor de tamanho 15.

5.5 Quicksort

A partir daqui é necessário uma análise mais rigorosa dos algoritmos, já que foram implementadas uma gama de variações deste algoritmo.
Em termos gerais, o quicksort, apesar de seu pior caso n^2, foi o algoritmo mais rápido. Quanto ao espaço de memória, o algoritmo necessita de *n*, de modo a ser pior do que o Heapsort nesse aspecto.

A velocidade do quicksort pode ser justificada pela menor quantidade de comparações, cópias e chamadas recursivas. Além disso, o pior caso do Quicksort é difícil de ocorrer, mas isso será discutido em seções futuras.

5.5.1 Quicksort Mediana

Ao início do projeto, esse modelo de quicksort me pareceu muito promissor, no entanto ele não atendeu a todas as expectativas iniciais. Foi pensado que o algoritmo seria muito superior ao quicksort tradicional, mas os resultados foram muito similares.

Foi raciocinado inicialmente que, escolhendo a mediana de *k* valores aleatórios, de modo que no mínimo $k < n/2$, de um vetor de modo a evitar o pior caso, que ocorre ao pegar termos pequenos ou grandes em relação aos termos do vetor, o Quicksort sempre estaria em um caso bom e portanto seria muito rápido.

Entretanto, foi notado que a medida que o *k* aumenta, mais lento fica. A explicação do porquê disso é simples: aliado ao custo de achar a mediana de *k* elementos, quando o Quicksort tradicional é executado em um vetor de tamanho grande, a probabilidade do pior caso ocorrer é muito baixa. Segue uma explicação disso:

Supondo que o Quicksort tem casos ruins na faixa de 25% dos primeiros termos e 25% dos últimos termos, tem-se que metade dos termos são ruins.

Então em um vetor de 11 termos, que possui valor próximo à média de chamadas da função igual a 7, caso seja escolhido o pior caso, serão chamados 11 vezes, o que não é muito mais do que 7. Assim, não há muita necessidade de se utilizar esse algoritmo, já que o ganho é mínimo.

No entanto, para casos grandes, parece que essa situação tende a favorecer o lado do quicksort mediana, de modo a evitar o pior caso, que é n^2. Porém, à medida que o *n* aumenta, o intervalo de números ruins “diminui”, já que a execução de algumas

chamadas a mais não fará tanta diferença, já que o número basal de chamadas já é muito maior do que essa diferença.

Houve uma tentativa de execução de uma prova formal dessa afirmação, entretanto ela não é fácil de ser elaborada, mas intuitivamente e com o auxílio de provas “informais”, foi possível chegar a essa conclusão.

Um adendo importante é que esse algoritmo é o que mais possui potencial de otimização, já que a solução implementada para achar a mediana é $O(n^2)$ e existe um algoritmo que tem seu caso médio linear.

Uma possível otimização, que também foi feita ao longo do estudo, foi realizar o Quicksort Mediana apenas nas primeiras execuções e com *k* pequeno. O resultado foi melhor do que o Quicksort tradicional, diferentemente do Quicksort Mediana comum. Entretanto, como não foi o algoritmo solicitado, ele não permaneceu no código e não foi utilizado nos testes apresentados nesta documentação

Essa otimização faz sentido porque como explicado anteriormente, não faz tanto sentido utilizar o cálculo de mediana em muitos casos, no entanto, é satisfatório utilizar nas execuções maiores, de maneira a eliminar possíveis piores casos.
Isso funciona apenas pois no vetor de tamanhos pequenos é trivial achar a mediana.
Esse algoritmo com $k = 3$ e com o tamanho de vetor igual a 15 realizou 51 cópias e 90 comparações.

5.5.2 Quicksort Empilha Inteligente

Essa versão, como esperado, não é tão rápida em relação aos modelos implementados de quicksort, isso se deve ao fato de que a *“inteligência”* da pilha tem um preço: se por um lado há uma otimização do espaço de memória, por outro há uma perda na velocidade.

Essa melhora é justificada pela não acumulação de processos na pilha, já que são executados os processos mais rápidos primeiro.

Essa perda é justificada pelas cláusulas *if's* nessa versão, que aumentam um pouco a performance do programa, além do tempo de alocação e desalocação dos blocos.

A performance desse algoritmo foi próxima às outras versões do quicksort, de modo a realizar 51 cópias e 106 comparações com um vetor de 15 posições.

5.5.3 Quicksort não Recursivo

Esse algoritmo também surpreendeu negativamente, já que tem-se no consenso que algoritmos recursivos são mais rápidos, devido à não necessidade de alocação na pilha de execução do computador. No entanto, nesse caso não houve uma melhoria em termos de tempo, apesar de ser claro a otimização no quesito espaço de memória.

O tempo de ordenação novamente ficou bem próximo em relação ao tempo do modelo tradicional e o número de cópias e comparações desse modelo foi de, respectivamente, 51 e 94 para um vetor de tamanho 15.

5.5.4 Quicksort Recursivo

Surpreendentemente, a versão tradicional do quicksort apresentou um tempo de execução muito baixo, sendo mais rápido do que muitas de suas versões "otimizadas".

Esse algoritmo foi o que realizou menos cópias e comparações, de modo a ter realizado 51 e 82 respectivamente.

5.5.5 Selection Quicksort

A versão Selection Quicksort apresentou o melhor desempenho. Quando o *m* é definido através de uma análise boa, o algoritmo funciona bem, caso contrário, não há nenhum benefício significativo.

É importante ressaltar que há uma leve melhoria no uso da memória, visto que o Selection Sort possui complexidade constante em relação à memória, mas nada muito significativo, já que à medida que havia essa melhora, o tempo de execução era prejudicado.

Em geral, o algoritmo se comporta bem com *m* seguindo aproximadamente relação $2 <= m <= 10^5$. Isso faz sentido pois o Selection Sort é relativamente rápido para lidar com vetores pequenos.

A complexidade desse algoritmo segundo minhas análises, permanece a mesma do Quicksort tradicional quando o *m* é ajustado ao tamanho da entrada e entre o intervalo citado.

A explicação disso vai ser baseada no exemplo utilizado nos testes, que consiste em um vetor de 10^6 posições e com m = 10^3.

Adotando a complexidade do Quicksort como O($n \log n$) e a do Selection Sort O(n^2), supondo que Quicksort faça apenas divisões ao meio no vetor e sabendo que o Selection Sort será executado quando o tamanho do vetor se torna menor ou igual a *m*, tem-se que:

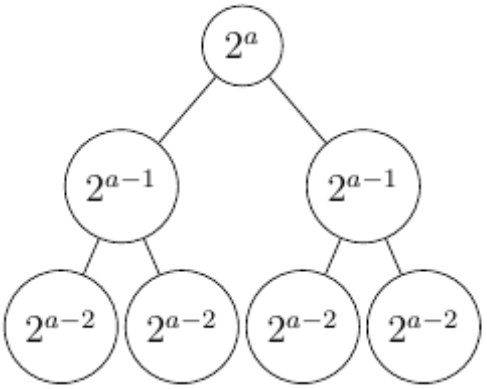
$$\frac{n}{2^x} \leq m, \ execution \ in \ O(n^2) \tag{1}$$

$$n \leq 2^x \cdot m, \ execution \ in \ O(n^2) \tag{2}$$

$$\frac{n}{m} \leq 2^x, \ execution \ in \ O(n^2) \tag{3}$$

$$x \geq \log_2 \frac{n}{m}, \ execution \ in \ O(n^2) \tag{4}$$

Estrutura de árvore binária



Análise de complexidade

Tomando y como o índice da linha atual, começando da linha 1, tem-se que a soma de todos os termos até a linha y é $\mathbf{2^y - 1}$

No caso do problema, o y é definido como:

$$\boldsymbol{y = \log_2 \frac{n}{m}}$$

Também é notório que a quantidade de termos da linha y é o termo y da P.G. de razão 2 que inicia em 1: 1, 2, 3, 4, ...

Assim, conclui-se que até a linha y é executado o Quicksort e na linha $y + 1$ é executado o Selection Sort. Portanto, o segundo algoritmo é executado uma vez a mais nesse caso, o que leva a crer que a complexidade converge para $O(n)$.

No entanto, o Quicksort realiza as operações com vetores de, no mínimo, o dobro do tamanho, de modo que à medida que y aumente, essa diferença de tamanho dos vetores fique cada vez maior, o que leva a crer que o algoritmo de ordenação predominante é o Quicksort.

Logo, para os exemplos testados, em que o y é relativamente grande, o algoritmo se comporta como $\boldsymbol{O(n \cdot logn)}$.

Uma observação importante é a de que a estrutura binária encontrada surgiu a partir do funcionamento suposto na análise de que o quicksort divide o vetor na metade.

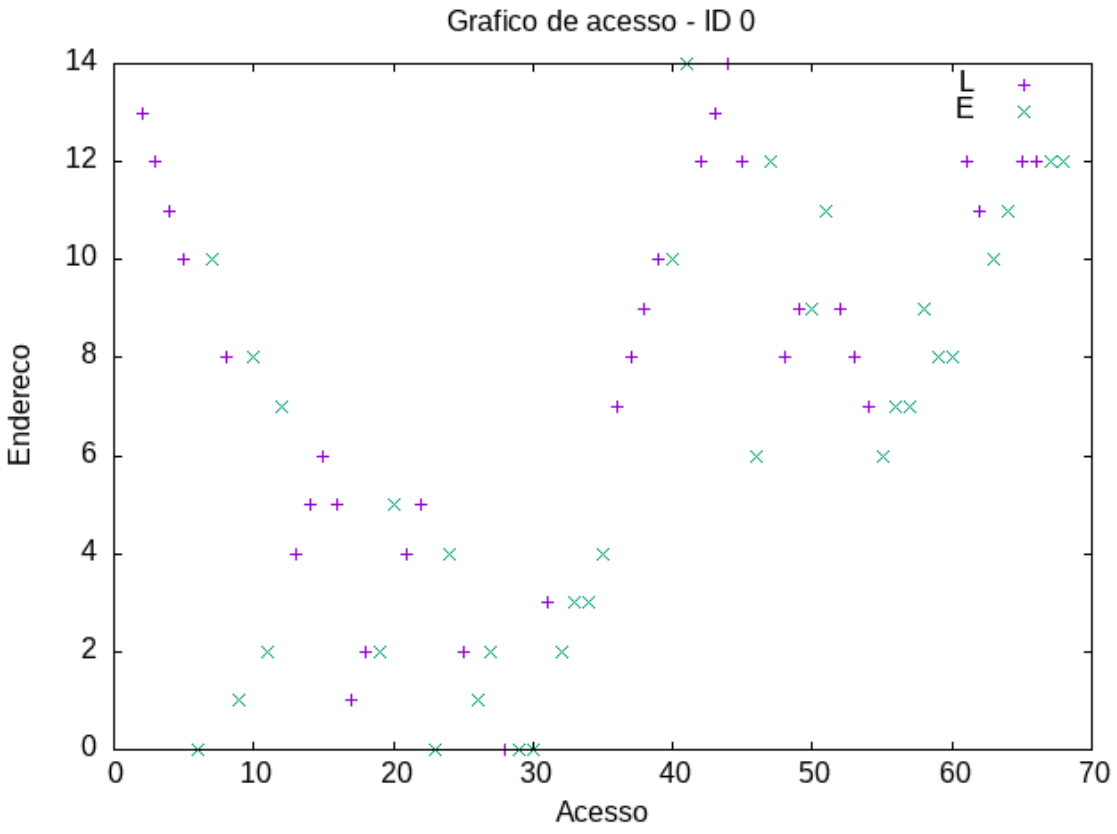
Por fim, esse algoritmo realizou 51 cópias e 83 comparações para um *m = 5* com um vetor de tamanho 15.

5.6 Localidade de referência

Os gráficos foram gerados a partir dos algoritmos de ordenação citados ao ordenarem um vetor de tamanho 15, com *k = 3* e *m = 5*.

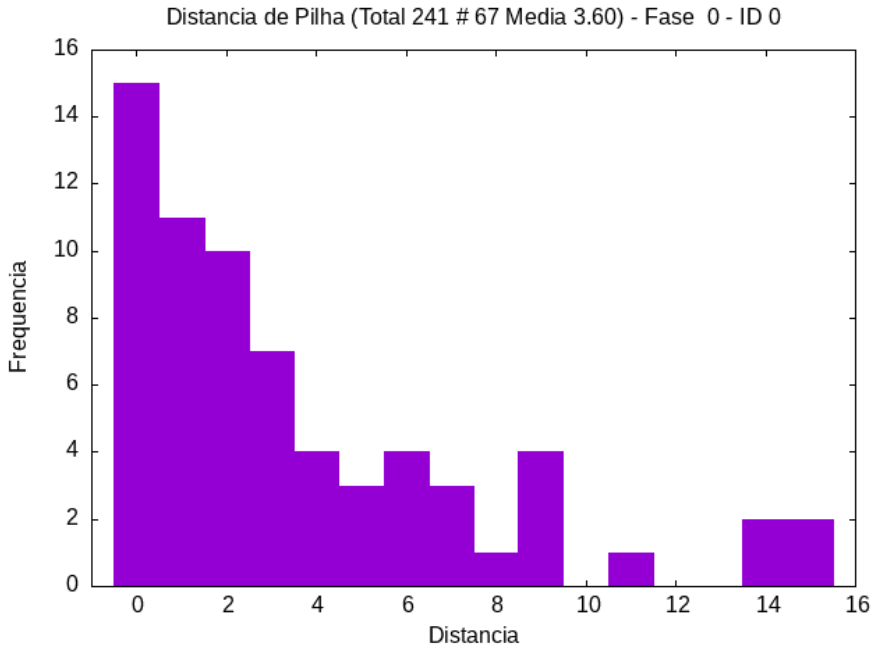
Os algoritmos estão em uma ordem arbitrária e não seguem a conformação anterior.

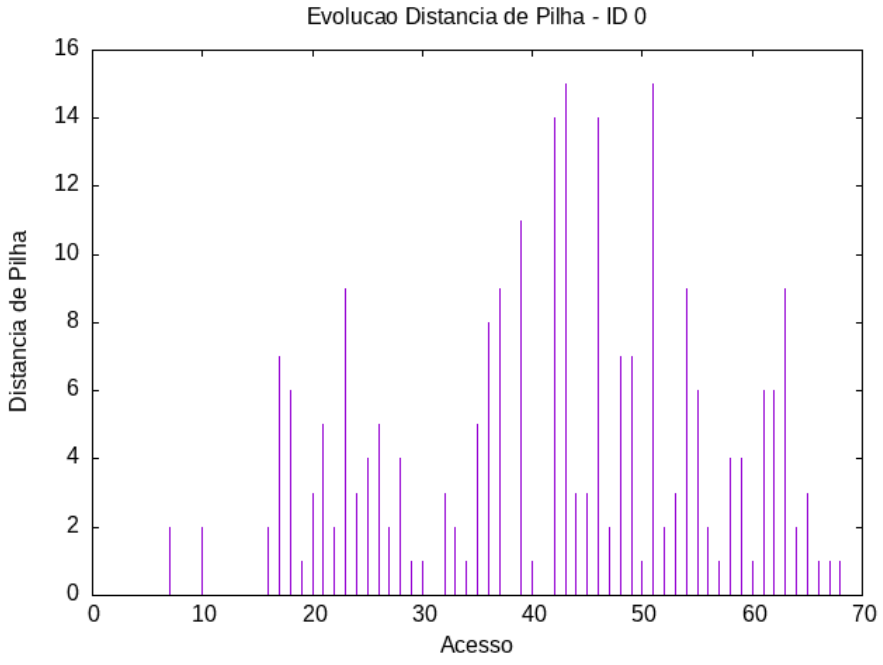
5.6.1 Quicksort Recursivo



De fato o gráfico corresponde à implementação, já que é visível cada parte do código:

Os pontos roxos descendentes no começo representam o acesso aos elementos da direita do vetor para checar se são menores do que o pivô. Os pontos verdes representam a troca dos registros e os pontos roxos ascendentes representam a checagem de caso o elemento da esquerda do vetor são maiores do que o pivô.

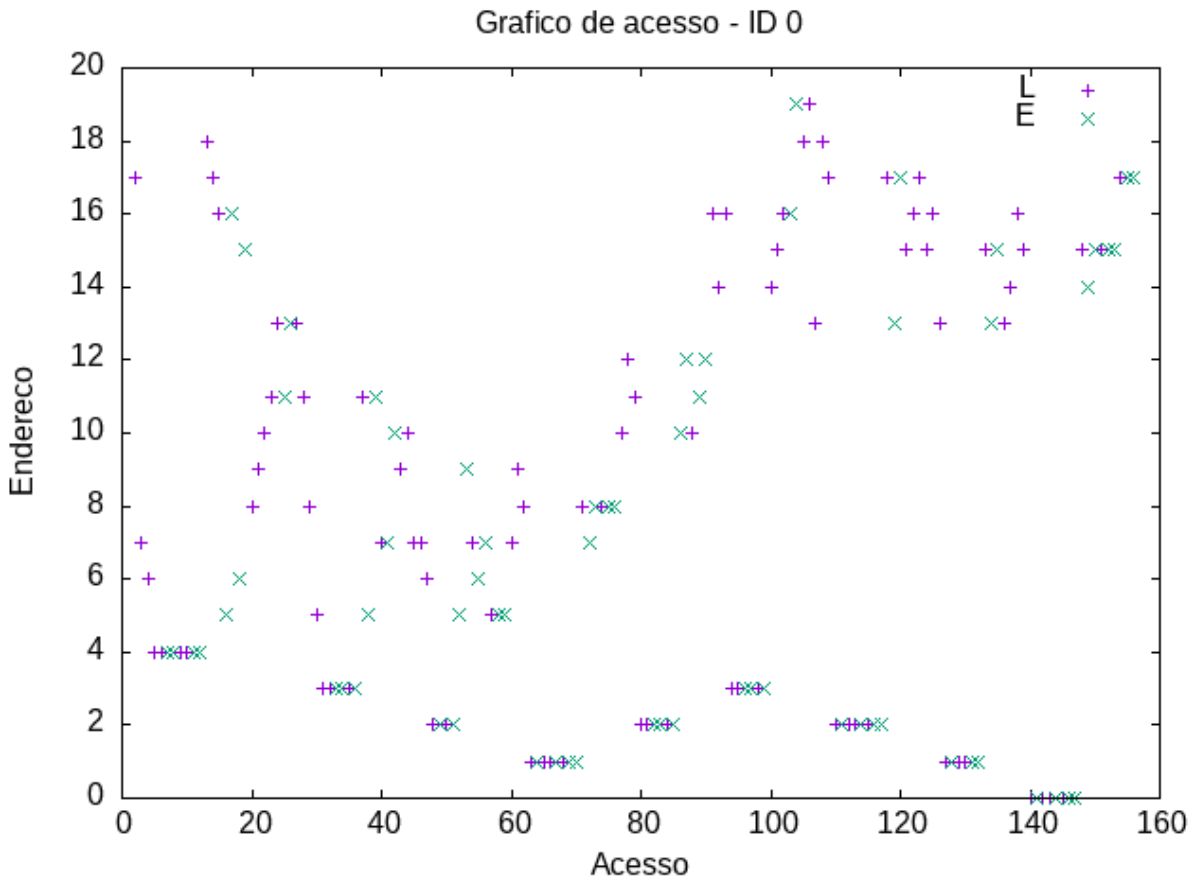




Esses serão os únicos gráficos desses tipos constando na documentação, já que todos os outros modelos resultaram em gráficos muito similares. Segundo análises próprias, realmente não deveria existir tanta modificação nesses gráficos de fato.

Há pequenas alterações nesses gráficos nos modelos não recursivos, mas nada muito impressionante ou interessante de se atentar.

5.6.2 Quicksort Mediana



É importante ressaltar que esse gráfico foi gerado a partir do algoritmo pedido pelo trabalho e não o otimizado que foi citado anteriormente.

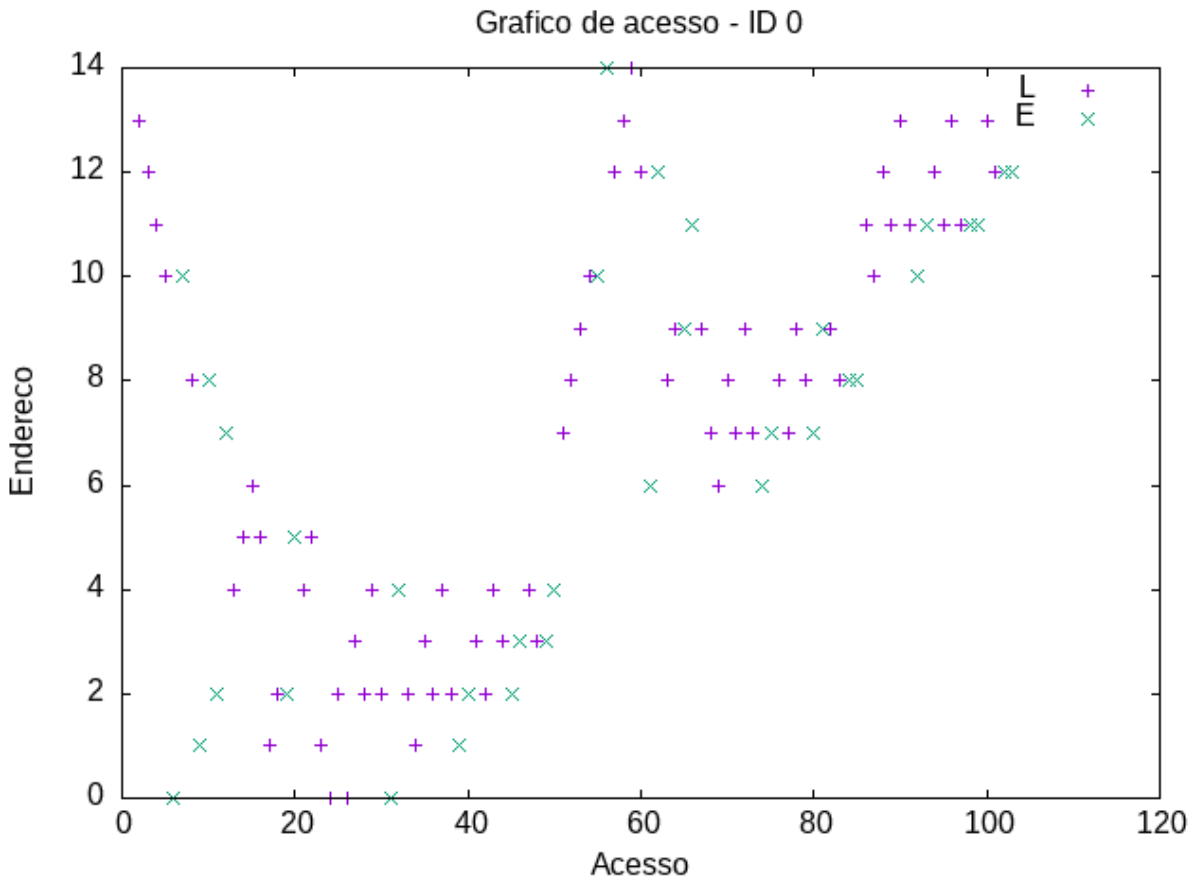
É claro notar que há uma grande diferença entre o Quicksort Mediana e o Quicksort recursivo. Primeiramente, é necessário falar que a função responsável por achar a mediana realiza tal objetivo através da ordenação do vetor de *k* termos. Desse

modo, percebe-se que a primeira coisa que ocorre é a ordenação de um vetor de 3 termos escolhidos aleatoriamente do vetor principal a ser ordenado.

Em seguida, o processo do Quicksort tradicional se repete até que chame novamente a função do Quicksort Mediana, que executa novamente o algoritmo para achar a mediana, de modo que o ciclo continue até que o vetor principal seja ordenado.

É importante observar que nesse gráfico, a concentração de símbolos verdes com formato de *x* é menor, de modo a mostrar que de fato há uma menor eficiência deste algoritmo em relação ao Quicksort tradicional.

5.6.3 Quicksort Seleção

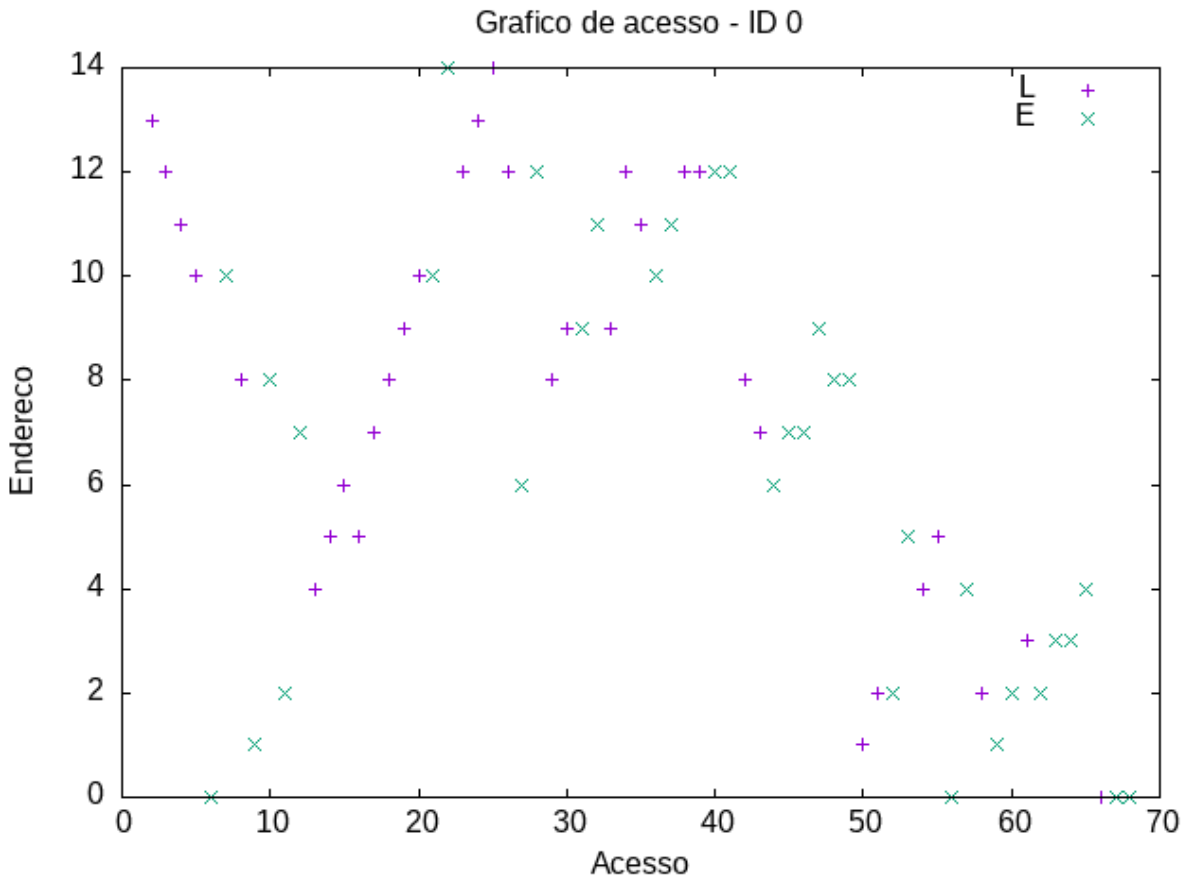


Esse gráfico, assim como o passado, ficou dissonante no quesito visual, no entanto, faz total sentido tendo em relação o código implementado.

Como dito anteriormente, é realizado um Quicksort tradicional até que o número de termos da iteração atual seja menor ou igual a *m*, de modo a ser executado o Selection Sort ao invés do Quicksort.

O Selection Sort é executado, por exemplo, no intervalo aproximado do acesso 20 ao acesso 40. Isso é perceptível pois há uma concentração de leituras que ficam em um intervalo de endereços bem definido, indicando que está ocorrendo uma varredura de valores de uma parte do vetor, que é o que acontece de fato.

5.6.4 Quicksort não recursivo

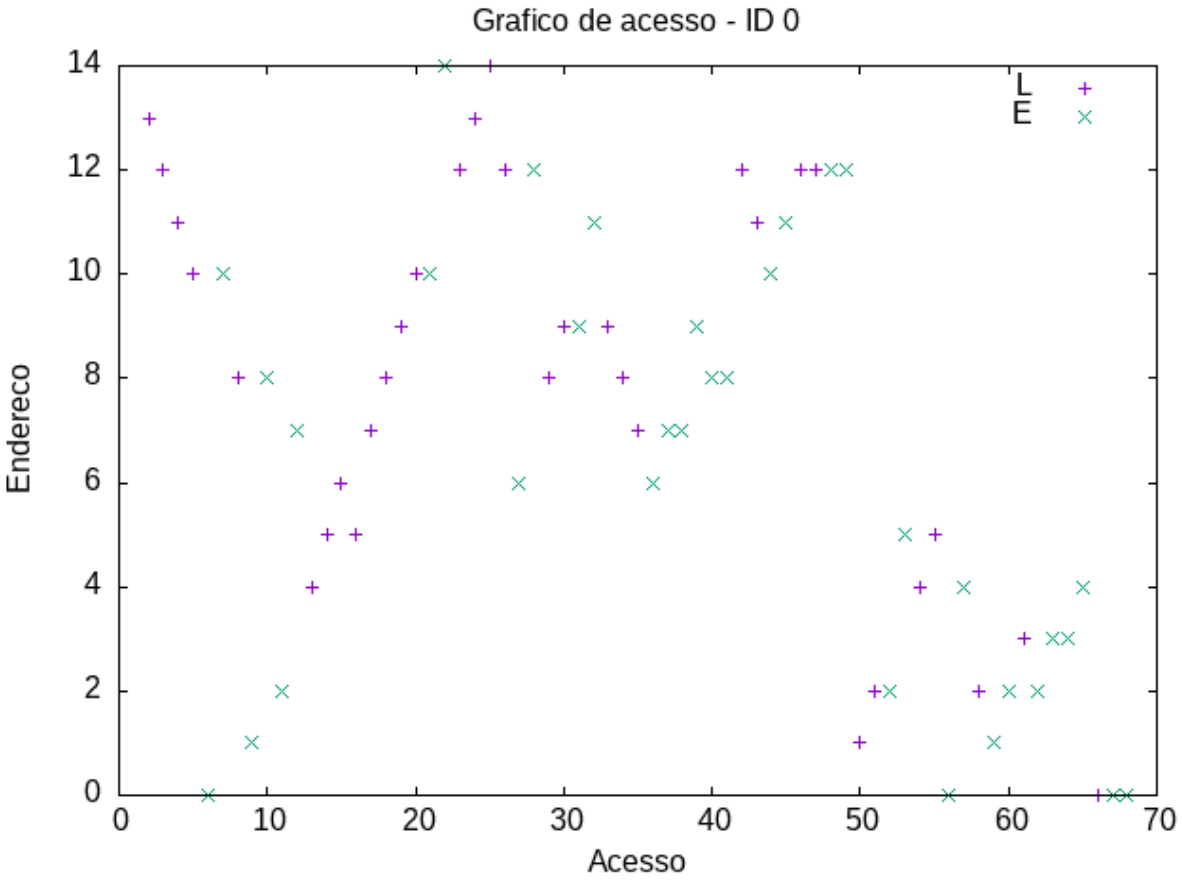


O gráfico do Quicksort Não Recursivo é muito parecido com o gráfico do Quicksort Recursivo tradicional, como esperado.

Isso ocorre devido a dois fatores:

- Não foi possível mostrar a estrutura de dados pilha nesse modelo, já que a alocação realizada é dinâmica e as bibliotecas *analisamem* e *memlog* não lidam muito bem com alocação dinâmica
- É executado os mesmos passos em ambos os algoritmos, com exceção da leitura e escrita na pilha, como citada anteriormente

5.6.5 Quicksort Empilha Inteligente



Como esperado, esse gráfico deveria seguir o mesmo padrão do modelo não recursivo, de modo a apresentar diferenças relacionadas à ordem que os padrões aparecem.

De fato, isso ocorre no gráfico gerado. Por volta do acesso 35, por exemplo, o modelo empilha inteligente realiza uma iteração que é realizada somente por volta do acesso 40. É difícil de visualizar isso, mas é o que ocorre na prática, já que o Quicksort Empilha Inteligente seleciona a menor partição antes de iterar novamente.

5.7 Tabela de performance

Como dito no início desta seção, a tabela contendo a média da performance iria constar no final. Então segue a tabela de performance sobre a qual as análises foram feitas, assim como a explicação de como se chegou nessa tabela final:

A tabela foi gerada a partir de uma média dos resultados das *seeds* 3310, 3311, 3312, 3313 e 3314, com entradas de 10^3 à 10^6, de modo que permaneceram somente os resultados de quando o tamanho era 10^6 para não ocupar muito espaço.

Esses testes foram gerados com $k = 3$ e $m = 1000$. Outros valores também foram testados e levam a resultados similares.

Sort algorithm	Size	Time
Recursive Quicksort	1000000	5.8066
Median of 3 Quicksort	1000000	9.0758
Selection 1000 Quicksort	1000000	5.4248
Non recursive Quicksort	1000000	6.0436
Smart Stack Quicksort	1000000	6.0934
Mergesort	1000000	47.0620
Heapsort	1000000	21.4604

6. Conclusões

O trabalho foi realizado com sucesso, de maneira a identificar os prós e contras de cada algoritmo implementado.
O algoritmo que mais surpreendeu positivamente foi o Quicksort Seleção, que contou com os melhores tempos em praticamente todas as execuções de teste. Já o que mais surpreendeu negativamente foi o Mergesort, que no quesito tempo deixou muito a desejar.

A qualidade dos gráficos de localidade de acesso à memória foi uma surpresa positiva, pois é possível observar de fato as diferenças de cada algoritmo e a implementação realizada.

Por fim, o trabalho proporcionou um entendimento satisfatório dos métodos mais conhecidos de ordenação.

7. Bibliografia

GNU make. GNU Project. Disponível em <https://www.gnu.org/software/make/manual/make.html>

MARINHO, Tarcisio. **Sorting Algorithms**. Github, 27 de out. de 2019. Disponível em <https://github.com/tarcisio-marinho/sorting-algorithms>

Sorting Algorithms. Geek for Geeks, 25 de out. de 2022 (última atualização). Disponível em <https://www.geeksforgeeks.org/sorting-algorithms/>

8. Instruções para compilação e execução

A lista de comandos do *makefile* e suas funções seguem abaixo:

- *testexec*: execução de caso teste exemplo
- *testmem*: execução de caso teste de memória exemplo
- *all*: executa os comandos *clean, bin, testexec, testmem e plot*
- *bin*: constrói o executável do programa principal
- *binmem*: constrói o executável do programa de teste de memória
- *plot*: traça os gráficos a partir de arquivos preexistentes
- *analisamem*: compila e gera o executável a partir da biblioteca analisamem
- *clean*: “limpa” os diretórios, mantendo os arquivos necessários e removendo os desnecessários

Os outros comandos são utilizados para *debug* ou são comandos auxiliares dos comandos citados.

Agora tratando dos executáveis que são gerados pelos comandos, começaremos pelo executável principal.

O comando *make bin* gera um executável do programa principal com o nome de *main* no diretório *bin*. Para checar os comandos disponíveis pelo próprio executável, insira a flag *-h*, que gerará o seguinte output:

Args:

*-v <int> version of quicksort**

1: ordinary quicksort

2: median quicksort (k required)

3: selection quicksort (m required)

4: non recursive quicksort

5: smart stack quicksort

6: all quicksorts

*-i <file> input file name**

*-o <file> output file name**

*-s <int> seed of the random numbers**

-k <int> median of k quicksort

-m <int> until which m array size will be used selection sort
-q always show recursive quicksort time
-p show progress in real time
-n not execute mergesort and heapsort
-h help
** -- obligatory flags*
<> -- expects a value

A partir desse menu de ajuda é possível utilizar todas as funções do executável.

É importante ressaltar que o comando *make testexec* gera um caso teste com todos os algoritmos com a entrada informada em *ent.txt*, contida no diretório *src*.

O programa de teste de memória possui comportamento similar ao programa principal. Basta executar o comando *make binmem* que será criado um executável do programa de teste de memória no diretório *bin* com o nome de *testMem*. A partir daí basta executar o executável com a flag *-h* para ser impresso as flags existentes e o modo de se usar. Segue o que é impresso:

Usage: ./sortMem [options]
Options:
-n <int> Number of elements to sort
-s <int> Seed for randomization
-v <int> Select algorithm:
0 - QuickSort
1 - Median of k QuickSort <int>
2 - Selection m QuickSort <int>
3 - Non Recursive QuickSort
4 - Smart Stack QuickSort
5 - MergeSort
6 - HeapSort
-h Show this help

A partir desse menu fica claro como executar e utilizar o programa.

Caso haja alguma dúvida de como executar o código, basta analisar os exemplos no *makefile* ou simplesmente executar o comando *make all*.