TRABALHO PRÁTICO 02

ARTHUR LINHARES MADUREIRA
2021031599

1) INTRODUÇÃO

O objetivo do trabalho foi implementar diferentes estratégias de ordenação - quicksort recursivo, quicksort mediana, quicksort seleção, quicksort não recursivo, quicksort empilha inteligente, mergesort e heapsort. Com a implementação desses métodos, espera-se ser capaz de avaliá-los com base nos seguintes critérios: tempo de processamento, número de comparações e quantidade de cópias geradas. Primeiramente, essa análise será realizada com as cinco variações de quicksort testadas. Posteriormente, espera-se definir a melhor versão dos quicksorts testados e, então, compará-lo com o mergesort e o heapsort.

Para resolver esse desafio, foi criada uma classe denominada “Sort” que contém um array da struct “Data”, composta por um inteiro, 15 cadeias de caracteres e 10 números reais. Essa classe apresenta métodos responsáveis por executar os diferentes tipos de ordenação requeridos.

Além disso, foi criado um header responsável por contabilizar o tempo de processamento de cada sort por meio da biblioteca “sys/resource.h”.

2) MÉTODO

- **STRUCT DATA**

➢ **int key:** inteiro que é a chave para ordenação;
➢ **char strings:** 15 cadeias de strings com 200 caracteres cada;
➢ **float numbers:** 10 número reais

- **CLASSE SORT**

➢ **Sort:** método construtor que recebe os parâmetros seed, numberElements, output;
➢ **~Sort:** método destrutor;
➢ **void swap:** método para trocar dois elementos por meio de seus índices;
➢ **void randomArrays:** gera o array de “Data” utilizando a seed;
➢ **int getComparisons:** retorna o número de comparações realizadas;
➢ **int getCopies:** retorna o número de cópias realizadas;
➢ **int getNumberElements:** retorna o número de elementos do vetor;

- ➢ **int partition:** essa função pega o último elemento como pivô. Então, posiciona o elemento pivô em sua posição correta e coloca todos os menores (menores que pivô) à esquerda do pivô e todos os elementos maiores à direita do pivô.
- ➢ **int randomPartition:** define o pivô da quicksort mediana utilizando um laço “for” que soma números aleatórios dentro do intervalo dado e, por fim, computa a média entre eles.
- ➢ **int heapify:** compara um nó, à sua esquerda e a sua direita. O maior dos três deve ser a raiz dessa sub-árvore. Caso o maior dos três seja o próprio nó, o algoritmo para. Caso contrário, segue a comparação da árvore.
- ➢ **void merge:** cria dois vetores auxiliares (R e L). Esses vetores copiam as duas metades da partição do vetor fornecido. Então, o vetor original recebe os elementos desses dois vetores auxiliares de forma que o elemento de menor chave seja atribuído primeiro até que sejam esgotados os elementos de R e L.
- ➢ **void recursiveQuickSort:** chama recursivamente a função “partition” para ordenar o vetor.
- ➢ **void selectionQuickSort:** utiliza a mesma lógica do recursiveQuickSort de chamar recursivamente a função “partition”. No entanto, caso o tamanho da partição seja menor que o parâmetro fornecido, é utilizado um selection sort para ordenar tal partição.
- ➢ **void medianQuickSort:** utiliza função randomPartition para definir o pivô do quicksort de forma aleatória.
- ➢ **void noRecursiveQuickSort:** utiliza a lógica de dividir o vetor em partições para ordená-las. No entanto, diferentemente do quicksort recursivo, utiliza estrutura de dados para armazenar os elementos em vez de utilizar recursão.
- ➢ **void stackSmartQuickSort:** funcionamento similar ao do noRecursiveQuickSort. No entanto, ordena primeiro a menor partição.

- **CLOCK.HPP**

Esse header possui funções responsáveis por contabilizar o tempo de processamento de cada método de ordenação. Além disso, essas funções registram as informações coletadas em um arquivo de saída.

3) ANÁLISE DE COMPLEXIDADE

3.1) TEMPO

OBS: o pior caso do quicksort ocorre na o pivô escolhido é o maior ou o menor elemento. Nesse caso, a complexidade de tempo do algoritmo fica em $O(n^2)$. Como a possibilidade dessa ocorrência é muito pequena, vamos atribuir ao quicksort a complexidade de seu caso médio: O(n log n).

- **QUICKSORT RECURSIVO**

Como discutido acima, a complexidade desse algoritmo no caso médio é definida com **O(n log n).**

O(n log n)

- **QUICKSORT MEDIANA**

Nesse quicksort, é utilizada uma função que contém um laço “for” com uma quantidade definida de iterações, ou seja, possui complexidade O(1). Somado isso ao algoritmo quicksort em seu caso médio, temos:

O(n log n) + O(1) = O(n log n)

- **QUICKSORT SELEÇÃO**

Nesse quicksort, é acionado o selection sort sempre que uma partição fica menor que determinado tamanho estabelecido. Tendo em visto que o selection sort é realizado com um tamanho constante, é nítido que tal operação possui O(1). Somado isso ao algoritmo quicksort em seu caso médio, temos:

O(1) + O(n log n) = O(n log n)

- **QUICKSORT NÃO RECURSIVO**

Bem como o quicksort recursivo, esse algoritmo possui complexidade O(n log n) no caso médio.

- **QUICKSORT EMPILHA INTELIGENTE**

Bem como o quicksort recursivo, esse algoritmo possui complexidade O(n log n) no caso médio.

- **MERGESORT**

A complexidade de tempo do mergesort é dada por O(n log n).

- **HEAPSORT**

A complexidade de tempo do heapsort é dada por O(n log n).

Para sintetizar os dados supracitados, vamos utilizar a tabela abaixo:

ALGORITMO	TEMPO MÉDIO	MELHOR TEMPO	PIOR TEMPO
QUICKSORT RECURSIVO	O(n log n)	O(n log n)	O (n^2)
QUICKSORT NÃO RECURSIVO	O(n log n)	O(n log n)	O (n^2)
QUICKSORT MEDIANA	O(n log n)	O(n log n)	O (n^2)
QUICKSORT EMPILHA INTELIGENTE	O(n log n)	O(n log n)	O (n^2)
MERGESORT	O(n log n)	O(n log n)	O(n log n)
HEAPSORT	O(n log n)	O(n log n)	O(n log n)

3.2) ESPAÇO

No caso médio, os quicksorts possuem complexidade de espaço definida em O(log n). No entanto, isso não se aplica ao pior caso, haja vista que é criada uma partição de tamanho n, sendo assim O(n).

OBS: no pior caso, o quicksort empilha inteligente continua com complexidade de espaço O(log n).

O heapsort é um algoritmo de ordenação *in-place,* por isso não requer espaço adicional e, consequentemente, apresenta complexidade de espaço dada por O(1).

O mergesort precisa realizar uma cópia inteira do vetor na memória. Por isso, sua complexidade de espaço é definida por O(n).

Para sintetizar os dados supracitados, vamos utilizar a tabela abaixo:

ALGORITMO	ESPAÇO MÉDIO	PIOR ESPAÇO
QUICKSORT RECURSIVO	O(log n)	O(n)
QUICKSORT NÃO RECURSIVO	O(log n)	O(n)
QUICKSORT MEDIANA	O(log n)	O(n)
QUICKSORT EMPILHA INTELIGENTE	O(log n)	O(log n)
MERGESORT	O(n)	O(n)
HEAPSORT	O(1)	O(1)

4) ESTRATÉGIAS DE ROBUSTEZ

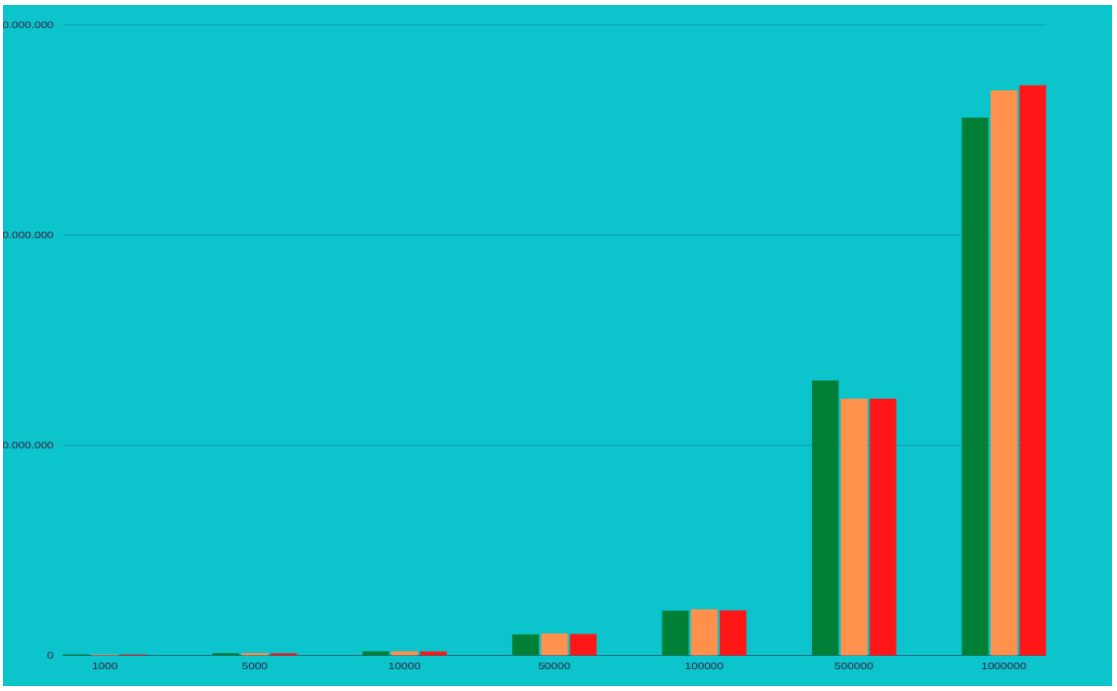
- ➔ **CHECAGEM DE ARQUIVO:** caso o arquivo de entrada não seja aberto corretamente, é acionado um *erroAssert.*
- ➔ **TAMANHO DO ARRAY:** caso o tamanho do array seja menor ou igual à zero, é acionado um *erroAssert.*
- ➔ **ANÁLISE DAS OPÇÕES:** caso a opção da flag -v seja inválida, é acionado um *erroAssert.*

5) ANÁLISE EXPERIMENTAL

5.1) COMPARAÇÃO ENTRE OS QUICKSORTS MEDIANA

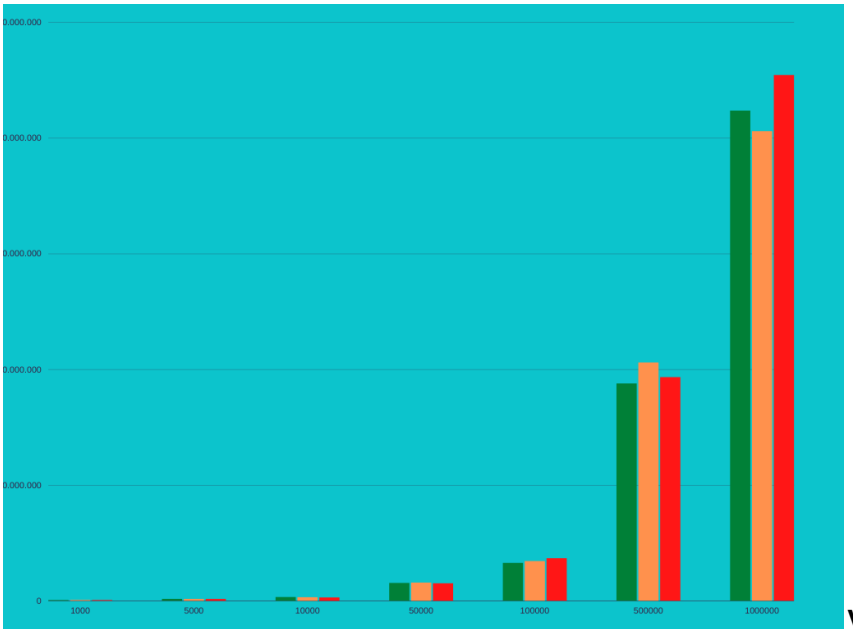
Para realizar essa análise, foram testados os quicksorts mediana com parâmetros k = 3, k = 5 e k = 7. Os resultados estão apresentados nos gráficos abaixo:

NÚMERO DE COMPARAÇÕES



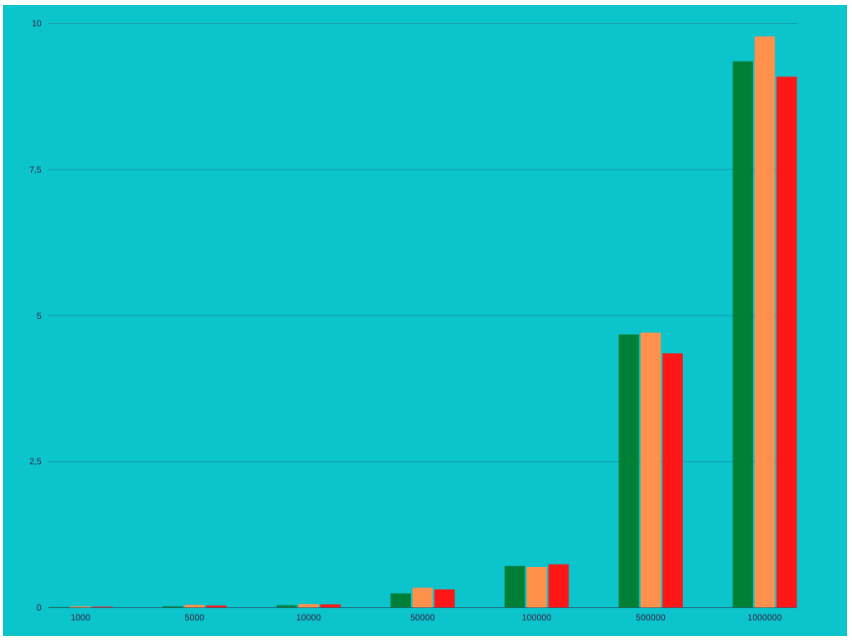
VERDE: K = 3
LARANJA: K = 5
VERMELHO: K = 7

NÚMERO DE CÓPIAS



VERDE: K = 3
LARANJA: K = 5
VERMELHO: K = 7

TEMPO



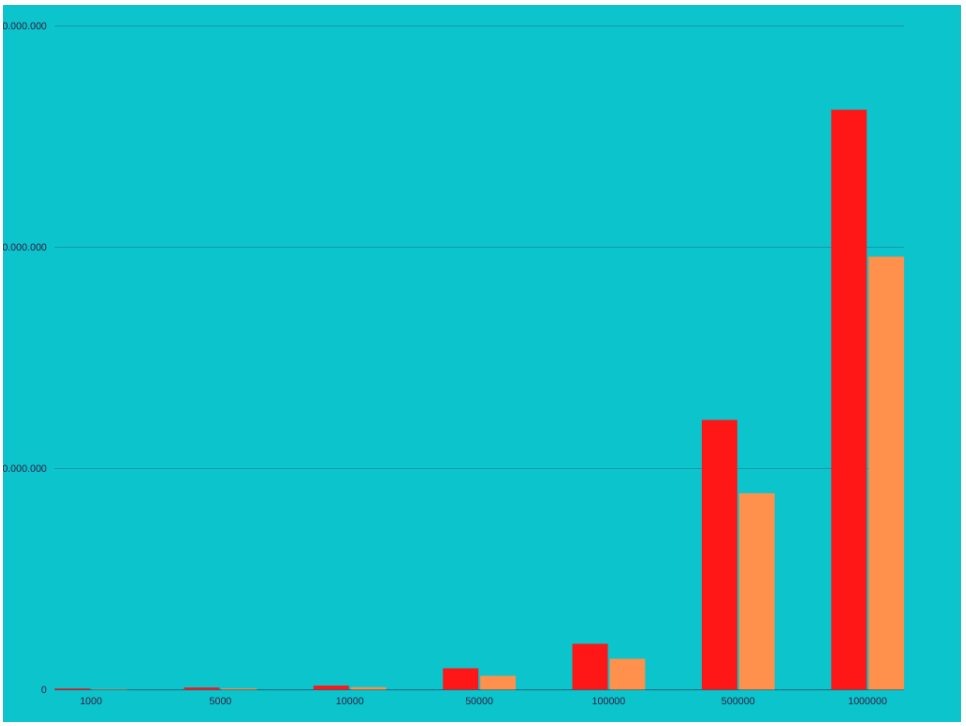
VERDE: K = 3
LARANJA: K = 5
VERMELHO: K = 7

Como pode ser observado nos gráficos, a mudança de parâmetro não impacta significativamente para o desempenho do algoritmo. Isso se deve ao fato de que o pivô acaba sendo escolhido randomicamente, não sendo possível ,assim, prever como essa escolha afetará o programa.

5.2) COMPARAÇÃO ENTRE OS QUICKSORTS SELEÇÃO

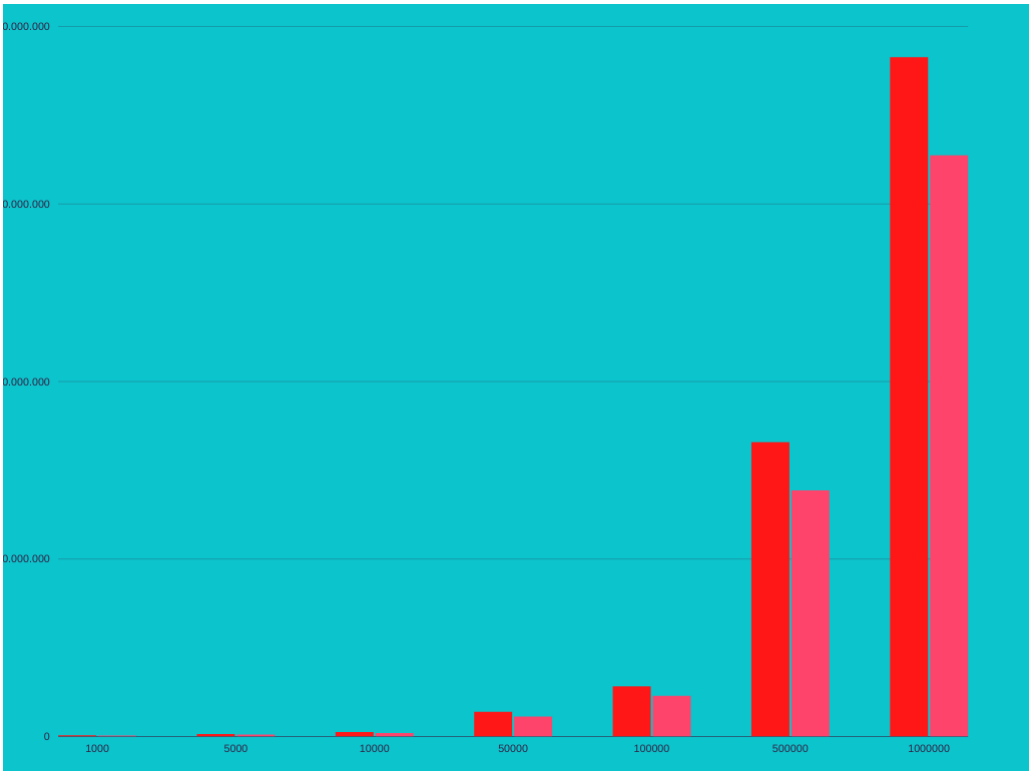
Para realizar essa análise, foram testados os quicksorts seleção com parâmetros m = 10 e m = 100. Os resultados estão apresentados nos gráficos abaixo:

NÚMERO DE COMPARAÇÕES



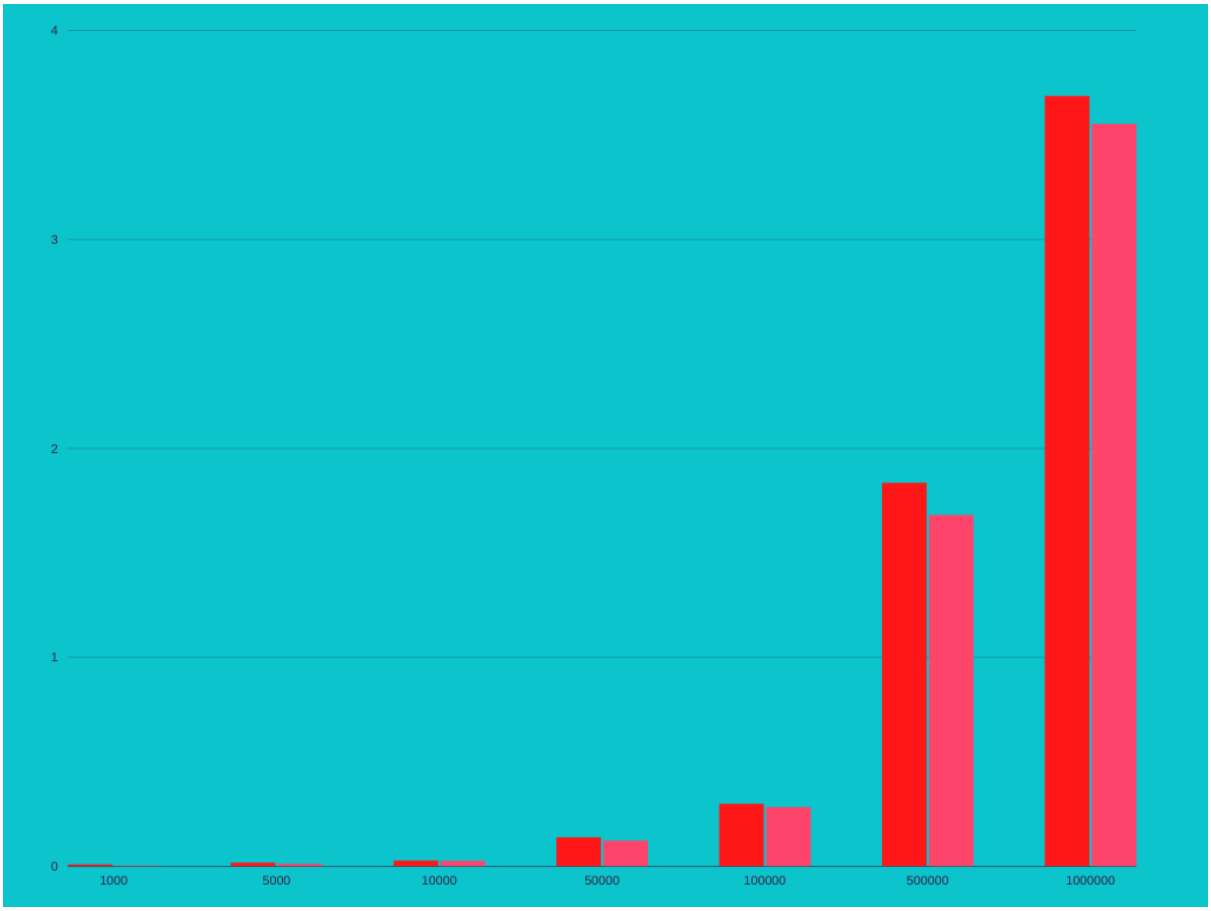
VERMELHO: M =10
LARANJA: M = 100

NÚMERO DE CÓPIAS



VERMELHO: M = 10
ROSA: M = 100

TEMPO



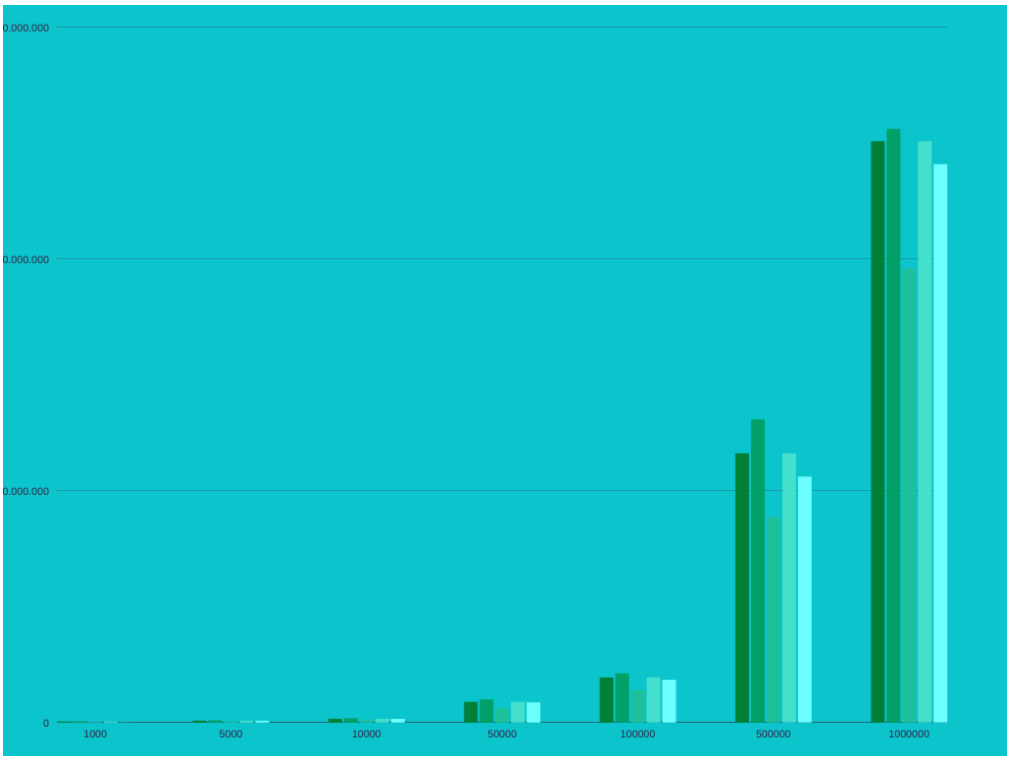
VERMELHO: M = 10
ROSA: M = 100

Como observado, o algoritmo apresenta melhor desempenho na situação em que m= 100. O número de cópias e o número de comparações fica consideravelmente menor do que em relação a quando m = 10.

5.3) COMPARAÇÃO ENTRE OS QUICKSORTS

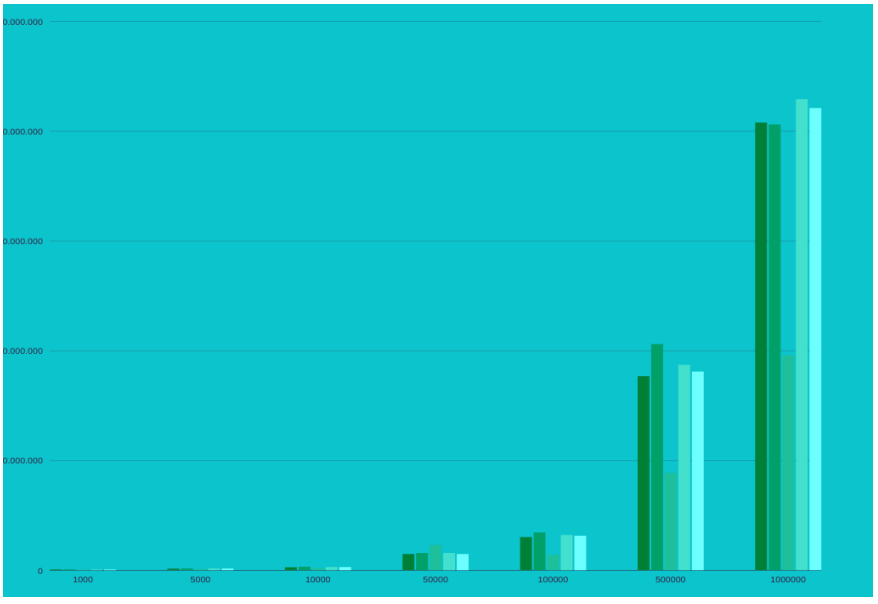
Para facilitar a comparação, utilizaremos a média obtida entre as variações do quicksort mediana, haja vista que todos apresentam resultados similares. Além disso, vamos considerar o quicksort selection com m = 100, já que esse desempenhou melhor o algoritmo.

NÚMERO DE COMPARAÇÕES



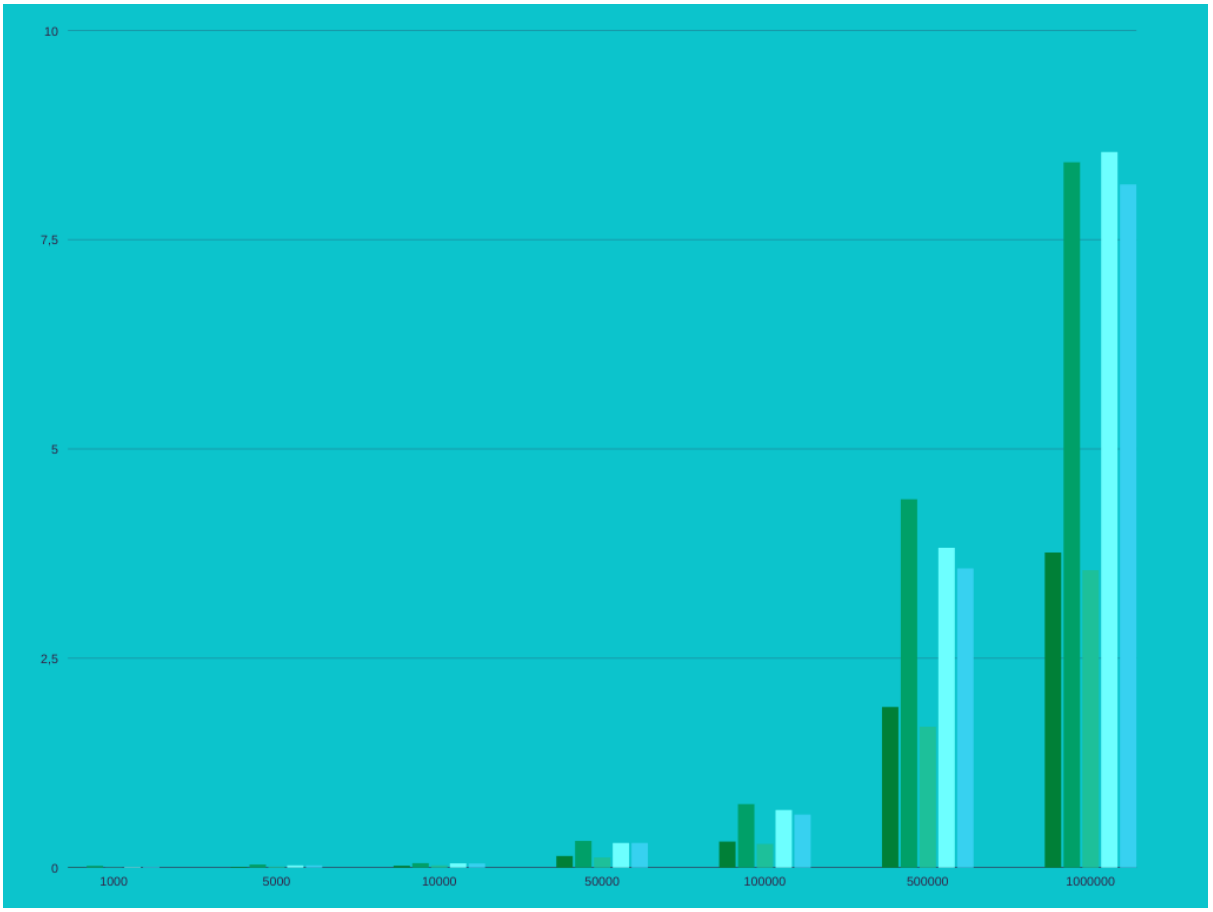
VERDE MUITO ESCURO: RECURSIVO
VERDE ESCURO: MEDIANA
VERDE: SELEÇÃO
AZUL: NÃO RECURSIVO
AZUL CLARO: EMPILHA INTELIGENTE

NÚMERO DE CÓPIAS



VERDE MUITO ESCURO: RECURSIVO
VERDE ESCURO: MEDIANA
VERDE: SELEÇÃO
AZUL: NÃO RECURSIVO
AZUL CLARO: EMPILHA INTELIGENTE

TEMPO



VERDE MUITO ESCURO: RECURSIVO
VERDE ESCURO: MEDIANA
VERDE: SELEÇÃO
AZUL: NÃO RECURSIVO
AZUL CLARO: EMPILHA INTELIGENTE

Diante da análise dos gráficos, pode-se chegar a algumas conclusões.

Primeiramente, é evidente que o versão “inteligente” do quicksort não recursivo não apresenta melhoras de performance em relação a sua versão mais simples no que diz respeito ao tempo de processamento.

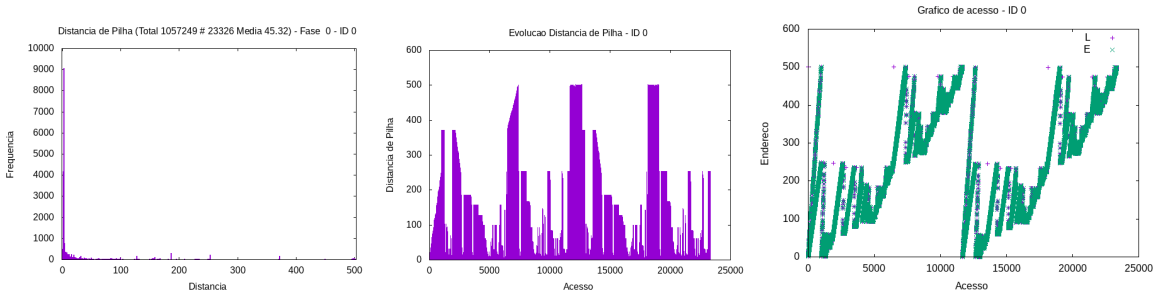
Além disso, é notável que os quicksorts mediana possuem queda de desempenho em relação ao quicksort normal.

Por fim, chegamos a conclusão que a melhor versão do quicksort é o seleção. Isso se deve ao fato de que o algoritmo selection é mais eficiente que o quicksort para partições pequenas, que é quando o selection sort é acionado.

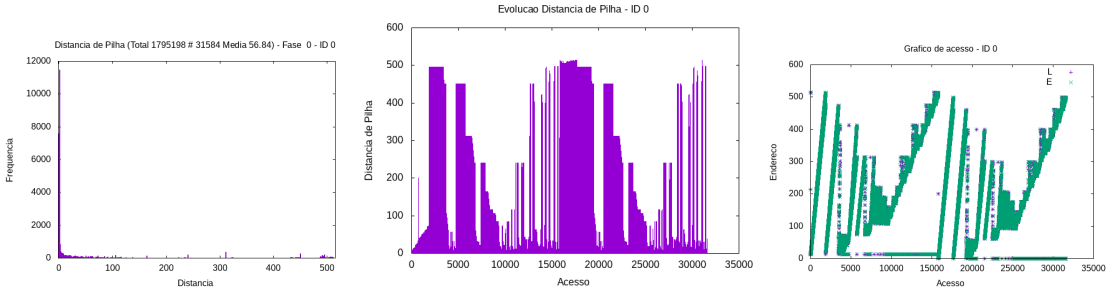
5.4) LOCALIDADES DE REFERÊNCIA

Para analisar as localidades de referência, foi utilizada a ferramenta *analisamem.* Abaixo, pode-se observar os gráficos dessa experimentação:

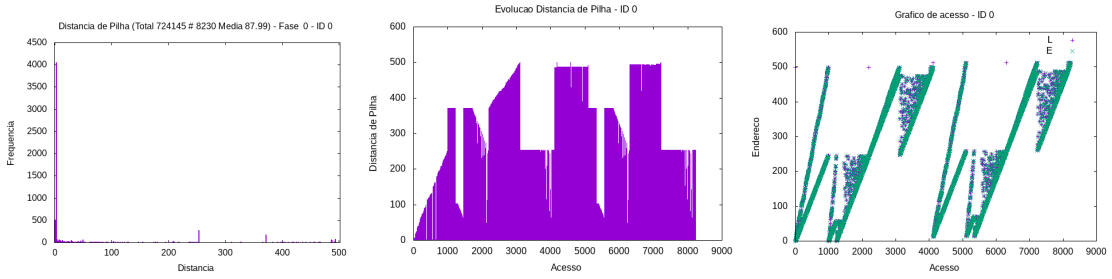
QUICKSORT RECURSIVO



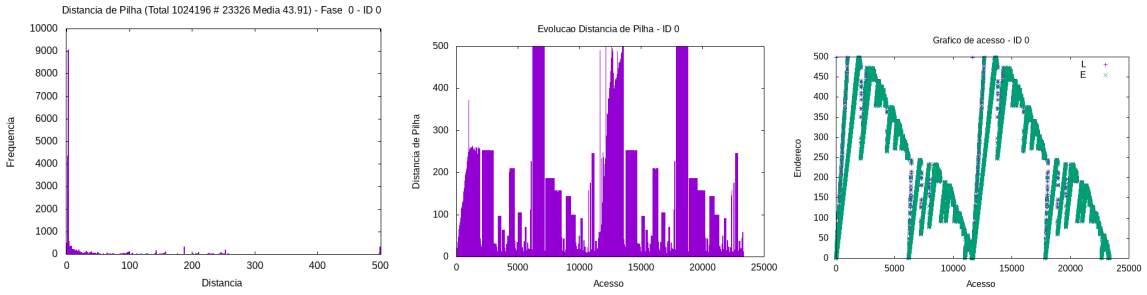
QUICKSORT MEDIANA



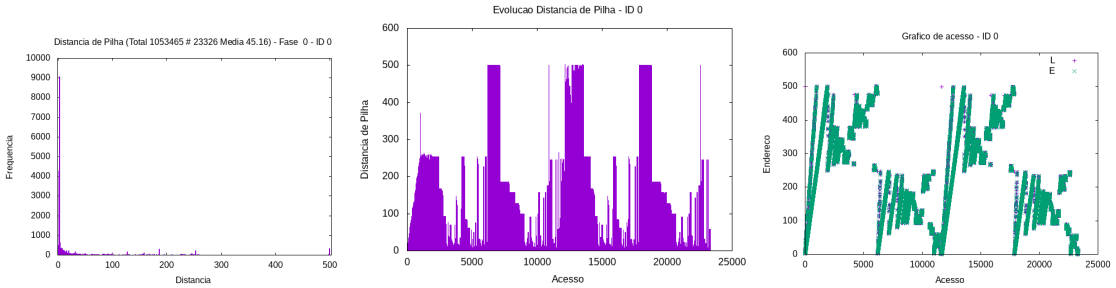
QUICKSORT SELEÇÃO



QUICKSORT NÃO RECURSIVO



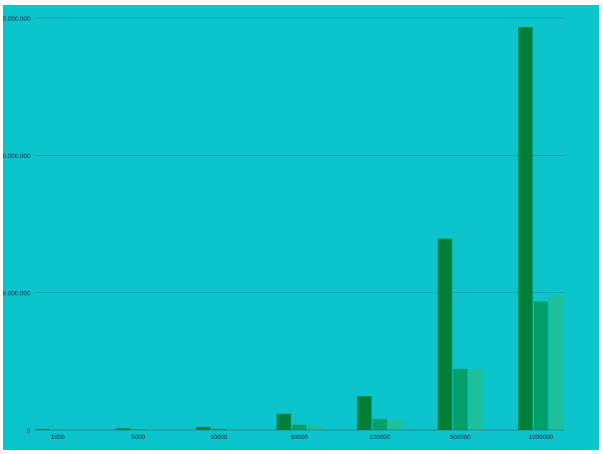
QUICKSORT EMPILHA INTELIGENTE



É notório que o único método de ordenação que se distingue dos demais no quesito de acesso à memória é selection quicksort. Isso se deve ao fato de que esse algoritmo é híbrido, apresentando uma parte que não segue a lógica proposta pelo quicksort.

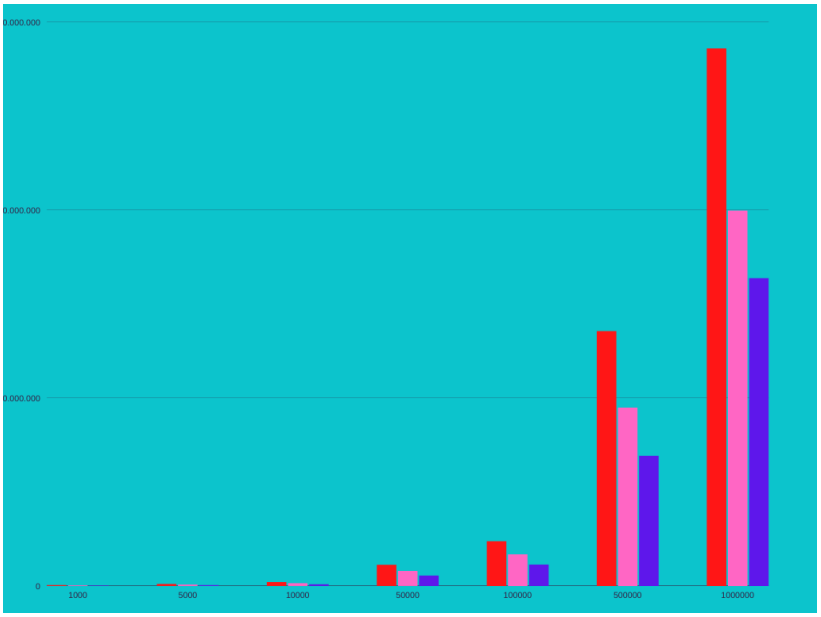
5.5) HEAPSORT x MERGESORT x SELECTION QUICKSORT

NÚMERO DE COMPARAÇÕES



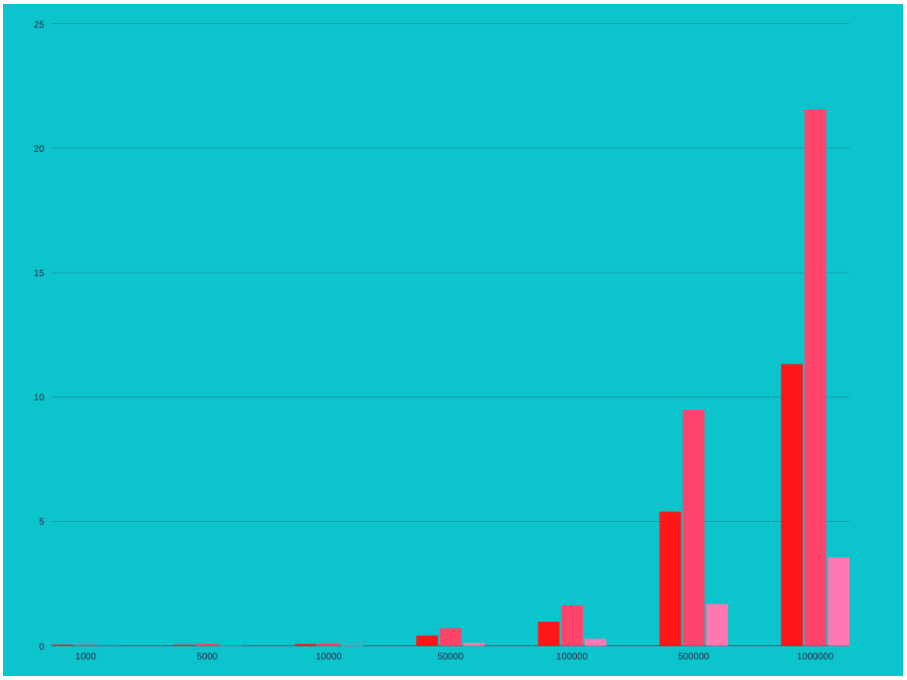
VERDE ESCURO: HEAPSORT
VERDE: MERGESORT
VERDE CLARO: SELECTION QUICKSORT

NÚMERO DE CÓPIAS



VERMELHO: HEAPSORT
ROSA: MERGESORT
ROXO: SELECTION QUICKSORT

TEMPO



VERMELHO: HEAPSORT
ROSA: MERGESORT
ROXO: SELECTION QUICKSORT

Como esperado, o selection quicksort apresentou resultados melhores que o heapsort e o mergesort. O mergesort é o algoritmo que possui os tempos de processamento mais elevados.

6) CONCLUSÃO

Com a realização desse trabalho, foi possível aprender a implementar diferentes métodos de ordenação diferentes (variações do quicksort, o mergesort e o heapsort).
Além disso, foi possível a análise desses algoritmos para definir a ordem de complexidade de espaço e tempo de cada um.
Nota-se que todos possuíam a mesma complexidade de tempo no caso médio: O(n log n). No entanto, mesmo assim os quicksorts acabam sendo significativamente mais rápidos que as demais implementações.
No que diz respeito à complexidade de espaço, o heapsort possui a vantagem de possuir complexidade O(1). Por outro lado, o mergesort apresentou a pior complexidade espacial (O(n)). Isso ocasionou problemas na máquina em certos momentos quando o valor de N era 1 milhão.

Diante dos fatos supracitados, é evidente, portanto, que a realização desse trabalho foi muito importante para melhor entendimento dos métodos de ordenação estudados em sala de aula.

INSTRUÇÕES PARA COMPILAÇÃO E EXECUÇÃO

- Utilize o comando make
- Então digite ./bin/run.out com as seguintes flags:
  - ➢ “-v”: coloque o inteiro correspondente ao método de ordenação logo após essa flag (consulte a legenda no final).
  - ➢ “-m”: caso seja escolhido o selectionQuickSort, utilize essa flag para definir o valor do parâmetro.
  - ➢ “-k”: caso seja escolhido o medianQuickSort, utilize essa flag para definir o valor do parâmetro.
  - ➢ “-s”: coloque o inteiro que será utilizado como a semente logo após essa flag.
  - ➢ “-i”: coloque o nome do arquivo de entrada logo após essa flag.
  - ➢ “-o”: coloque o nome do arquivo de saída logo após essa flag.
  - ➢ “-p” (opcional): coloque o nome do arquivo que será registrado os dados do memlog logo após essa flag.

1: Quicksort Recursivo
2: Quicksort Mediana
3: Quicksort Seleção
4: Quicksort Não Recursivo
5: Quicksort Empilha Inteligente
6: Mergesort
7: Heapsort