Trabalho Prático 2

Thaís Ferreira da Silva - 2021092571

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

thaisfds@ufmg.br

1 Introdução

O problema proposto foi a implementação de um programa para ordenar um vetor aleatório de registro composto por uma chave (número inteiro), quinze cadeias de caracteres (strings) e 10 número reais. Éle deve ser capaz de ordenar o vetor através de um dos 7 métodos disponíveis: Quicksort Recursivo, Quicksort Mediana, Quicksort Seleção, Quicksort não Recursivo, Quicksort Empilha Inteligente, Mergesort e Heapsort.

Para isso é necessário que o usuário entre com um valor para a geração do vetor aleatório, o número da versão refente ao método de ordenação desejado (1 a 7), o arquivo .txt com os tamanhos dos vetores a serem gerados, e o nome do arquivo de saída .txt onde será informado o tempo de execução do método selecionado, juntamente com o numero de comparação de chaves e o numero de cópias de registros realizados. Além disso, alguns métodos necessitam de outras entradas como um valor para o tamanho da mediana e outro para decidir até quando o método de seleção será utilizado no lugar do quicksort.

2 Método

O programa foi desenvolvido na linguagem C++, compilado pelo compilador GCC da GNU Compiler Collection. O Computador utilizado tem as seguintes especificações:

- Sistema Operacional: Linux Ubuntu 22.04
- Processador: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz
- RAM: 8,00 GB (utilizável: 7,80 GB)

2.1 Namespace

Para armazenar alguns dados importantes que seriam utilizados em diversos locais, achei mais prático criar três namespaces par armazenar os dados de entrada, analise de comparação/cópias/tempo, e dados gerais utilizado na impressão no arquivo.

- **Entradas:** Esse namespace é responsável por armazenar os dados de entrada inseridos ao compilar o código, sendo eles:
  - Número da versão escolhido (define o método de ordenação);
  - Número da semente (seed) para definir os números aleatórios das chaves dos registros;

  - Valor de k utilizado no Quicksort Mediana que é utilizado para escolher o pivô para partição como sendo a mediana de k elementos do vetor;
  - Valor de m utilizado no Quicksort Seleção para ordenar partições com tamanho menor ou igual a m;
  - Nome do arquivo de entrada .txt com os tamanhos dos vetores a serem gerados e ordenados;
  - Nome do arquivo de saída .txt com o numero de comparações de chaves, cópias de registros e tempo de execução do método de ordenação ordenado.
- **Analise:** O namespace analise é responsável por armazenar os dados a serem analisados ao final da ordenação, sendo eles:
  - O número de comparações das chaves entre elementos do vetor;
  - O número de cópias de um registro realizadas;
  - O horário do inicio da ordenação do vetor;
  - O horário do fim da ordenação do vetor;
  - O tempo de execução do método de ordenação.
- **Geral:** Esse namespace é relevante para a impressão final dos dados no arquivo de saída, armazenando as médias para os 7 tamanhos solicitados no tp (vetores de 1000, 5000, 10000, 50000, 100000, 500000, e 1000000 elementos aleatórios):
  - Vetor com o nome das versões de ordenação da atividade;
  - Vetor com as 7 médias de comparações de chaves entre elementos do vetor;
  - Vetor com as 7 médias de copias de registro;
  - Vetor com as 7 médias de tempo de execução do método de ordenação.

2.2 Classes

O programa é composto de apenas três classes: Registro e Pilha.

- **Registro:** Essa classe é a responsável por armazenar o dado que é adicionado aleatoriamente no vetor a ser ordenado. Ele é composto por uma chave (número inteiro / int), um vetor de 10 números (números reais / float), um vetor de 15 cadeias de caracteres (15 strings de 200 caráteres).
- **Node:** Classe criada para armazenar o Item da pilha que é composto por dois inteiros (esquerda e direita), e o endereço do próximo node da pilha.
- **Pilha:** A pilha construída é utilizada no Quicksort Não Recursivo, e no Quicksort Empilha Inteligente. Ela armazena o tamanho da pilha, o endereço para o Node adicionado por último na pilha. Além disso é possível empilha, desempilha e limpar a pilha.

2.3 Funções

O programa é composto de dois principais arquivos com diversas funções, sendo eles o MetodosOrdenacao, e o MetodosAuxiliares.

- **MetodosOrdenacao**

- **Ordenar:** A função ordenar recebe o vetor de registro aleatórios e é responsável por chamar o método de ordenação desejado de acordo com a versão escolhida no momento de compilação. Para isso escolhi o padrão: 1 - Quicksorte Recursivo
  2 - Quicksorte Mediana (k)
  3 - Quicksorte Seleção (m)
  4 - Quicksorte Não Recursivo
  5 - Quicksorte Empilha Inteligente
  6 - Mergesort
  7 - Heapsort
- **SwpReg:** Essa função é responsável por trocar a posição de dois registros utilizando uma variável de registo auxiliar. Toda vez que essa função é chamada somas-se 3 no número de cópias de registro.
- **Particao:** A partição é utilizada em vários métodos de ordenação ela é responsável por comparar os elementos de uma parte do vetor, e chamar a função SwapReg para trocar um elemento desordenado de posição. Nessa função aumenta-se o número de comparações realizados na medida em que o pivô percorre o vetor.
- **ParticaoMediana:**
- **Selection:** O método Selection Sort rearranja as chaves do vetor sempre colocando o registro com menor valor de chave na primeira posição. Ele percorre o vetor inteiro, encontra o menor valor, coloca na primeira posição, e repete o processo com o resto do vetor desordenado. Esse link posse uma animação demonstrando o funcionamento do selectionsort.
- **QuicksortRecursivo:** O quicksort adota a estratégia de divisão e conquista. Nesse método o algoritmo rearranja as chaves recursivamente, colocando as menores chaves antes das maiores chaves considerando um pivô (normalmente esse pivô é o elemento do meio do vetor).
- **QuicksortMediana:** Semelhantemente ao quicksortRecursivo, esse método separa as menores e maiores chaves de acordo com um pivô. A diferença é que esse pivô é a mediana do vetor definido pelo valor k inicialmente escolhido pelo usuário.
- **QuicksortSelecao:** Semelhantemente ao quicksortRecursivo, esse método separa as menores e maiores chaves de acordo com um pivô. A diferença é que quando para uma partição de tamanho k ou menor escolhida pelo usuário, o vetor não realiza o quicksort e sim uma ordenação com o selectSort. Esse link posse uma animação demonstrando o funcionamento do quicksort.
- **QuicksortNaoRecursivo:** Semelhantemente ao quicksortRecursivo porem não recursivo. Sendo assim ele utiliza de uma pilha de itens (que armazenam um int esquerda e direita), que é utilizado para ordenar a pilha empilhando os registros.
- **QuicksortInteligente:** Semelhantemente ao quicksortNãoRecursivo, mas com uma pequena alteração que o torna mais inteligente. Ele empilha a maior partição e ordena a menor partição;
- **Merge:** O merge é uma função utilizada pelo método mergesort, que combina duas partes ordenadas de um mesmo array.
- **Mergesort:** O algorítimo Merge sort também adota a estratégia de divisão e conquista. Ele realiza "quebras"na metade do vetor recursivamente até que sobre apenas um ele-

mento. Depois ele utiliza da função merge para ordenar os seus dados quebrados. É importante notar que esse método tem um funcionamento semelhante a uma árvore binária. Esse link posse uma animação demonstrando o funcionamento do mergesort.

  - **Heap:** O heap é um algoritmo que enxerga o vetor com uma arvore binária
  - **Heapsort:** Esse link posse uma animação demonstrando o funcionamento do heapsort.

- **MetodosAuxiliares**
  - **Timer:** Função implementada pelo professor que retorna o horário atual do computador. Ela é utilizada para calcular o tempo total de execução ao fazer tempo final - tempo inicial.
  - **Argumentos:** Função que recebe os argumentos digitados durante a compilação do código, e armazena nas variáveis do namespace entradas.
  - **AnaliseGeral:** Essa função não faz parte das necessidades do tp2, mas foi desenvolvida para rodar e testar todos os 7 métodos de ordenação para 5 seeds diferentes, gerando um arquivo de saída para cada método, e uma saída com todas as médias obtidas. Para rodar esse método é necessário que o computador tenha mais de 8gb de ram, e que na hora de compilar o programa sejam passados todos os parâmetros normais e extras, escolhendo como versão o valor 0. Um exemplo será anexado ao final do documento.

3 Análise de Complexidade

- **SwapTeg:** Como essa função só realiza a troca de posição de dois elementos de um vetor, sua complexidade é a mesma para o melhor e pior caso $O(1)$.
- **Particao e ParicaoMediana:** Esse método precisa escolher um pivô, e depois percorre o vetor pela direita e pela esquerda até chegar no pivô ordenando o vetor. Por isso a complexidade sempre é a mesma para o pior e melhor caso $O(n/2 + n/2) = O(n)$.
- **Selection:** Como esse método sempre percorre o vetor inteiro, e compara cada registro com todos os outros registros do vetor, então a complexidade sempre é a mesma e é $O(n)$ no melhor e pior caso.
- **QuicksortRecursivo:** Esse método método sempre chama a função partição, o que já adiciona a sua complexidade $O(n)$. Depois ele tem duas condicionais que vão recursivamente chamar o método QuicksortRecursivo para as duas partições do vetor. Assim multiplicamos a complexidade por $O(logn)$ no melhor caso, e $O(n)$ no pior caso. Sendo assim:
  - Pior caso: $O(n)$.
  - Melhor caso: $O(nlogn)$.
- **QuicksortMediana:** Esse método tem a mesma complexidade do QuicksortRecursivo, e a única diferença é que o pivô sempre será a mediana o que evita o pior caso. Sendo assim:
  - Pior caso: $O(n)$.
  - Melhor caso: $O(nlogn)$.
- **QuicksortSelecao:** Esse método tem a mesma complexidade do QuicksortRecursivo, e a única diferença é que o a partir de um certo tamanho de partição o método para de chamar o quicksort e passa a ordenar os dados com o SelectionSort.
  - Pior caso: $O(n)$.

– Melhor caso: $O(nlogn)$.

$$T(n) = \begin{cases} O(nlogn), & n > m \\ \Theta(n), & n \leq m \end{cases} \quad (1)$$

- **QuicksortNaoRecursivo:** Nesse método não recursivo, é criado uma pilha que chamei de part que auxilia na ordenação dos registros. Ao longo do código, empilhamos na part o registro quando o valor da direita é maior do que o da esquerda, e desempilhamos quando o valor da direita é menor ou igual o da esquerda. Como esse processo ocorre até a part voltar a ser uma pilha vazia, as complexidades são:
  - Pior caso: $O(n)$.
  - Melhor caso: $O(nlogn)$.
- **QuicksortInteligente:** Esse método é semelhante ao método QuicksortNaoRecursivo, a diferença está na horá de empilhar os valores na part. Ao percebermos que a direita é maior do que a esquerda criamos uma partição e analisamos novamente os valores, porem agora comparando se o tamanho da partição da direita é maior que o tamanho da partição da esquerda. Sendo assim, caso a condição seja verdadeira empilhamos a partição da esquerda, e caso seja falsa, empilhamos a partição da direita. Sendo assim, a complexidade será:
  - Pior caso: $O(n)$.
  - Melhor caso: $O(nlogn)$.
- **Mergesort:** Como já explicado antes, esse método enxerga o vetor como uma arvore binária, dividindo os vetores sempre ao meio até chegar a um único registro. Depois ele vai reorganizando essas metades até o vetos ser recontruindo novamente de forma ordenado. Por isso a função tem o mesmo valor para o melhor e pior caso, sendo ele $O(nlogn)$.
- **Heapsort:** O método Heapsort também enxerga o vetor como uma arvore binário. Dessa forma ele organiza essa arvore colocando o maior valor na primeira posição, e depois manda esse maior valor de chave para o final do vetor, para que na próxima chamada o vetor a ser analisado tenha tamanho n-1. Dessa forma ele tem o mesmo valor para o melhor e pior caso, sendo ele $O(nlogn)$.

3.1 Complexidade de Espaço

Quase todos os tipos de ordenação estudados nesse trabalho prático possuem a mesma complexidade de espaço, pois nesses métodos não há alocação de memória. Entretanto os métodos não recursivos (QuicksortNaoRecursivo e QuicksortInteligente) são um pouco melhor nesse sentido, pois chamada de função gastam um certo espaço. Sendo assim a complexidade é $O(n)$. O único método que se destacou ao longo da análise foi o Mergesort, pois aparentemente possui um comportamento diferente dos outros. Como hà alocação de vetores cada vez menores (com o tamanho divididos pela metade) o esperado seria $O(nlogn)$, entretanto se considerarmos que o vetor sempre é desalocado antes de se alocar um novo pedaço a complexidade deveria ser $O(2n) = O(n)$.

4 Estratégias de Robustez

Para tornar o programa mais robusto foi utilizado do assert para verificar as principais exceções do programa. Foram implementadas algumas medidas para:

- Verificação se a versão desejada está entre 0 e 7
- Verificar se a seed, o k e o m são números positivos
- Verificar a extensão dos arquivos de entra e saída

Além disso, foram implementados outras estratégias de robustez onde o assert não foi necessário. As três estrategias foram:

- Definir como padrão os valores da seed, da versão, do k e do m para caso o usuário não entre com esses dados na hora de compilar o programa.

5 Análise Experimental

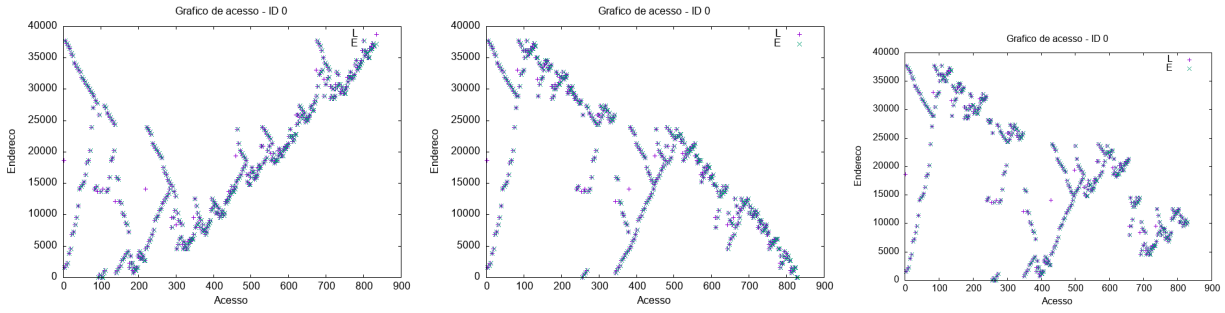
5.1 Impacto de variações do Quicksort

O Quicksort é um método popular, rápido e eficiente, inventado por C.A.R Hoare em 1960. O seu funcionamento adota a estratégia de divisão e conquista. Nesse trabalho testamos 5 variações do quicksort: 1 - Quicksorte Recursivo
2 - Quicksorte Mediana (k)
3 - Quicksorte Seleção (m)
4 - Quicksorte Não Recursivo
5 - Quicksorte Empilha Inteligente

Para isso, foram gerados gráficos de acesso à memória, distancia de pilha e evolução da distancia de pilha utilizando da biblioteca memlog e analisamem disponibilizados pelos professores. Os testes foram realizados com uma única entrada de tamanho 100, pois o objetivo é apenas de registrar o acesso à memória e comprovar o desempenho do programa.

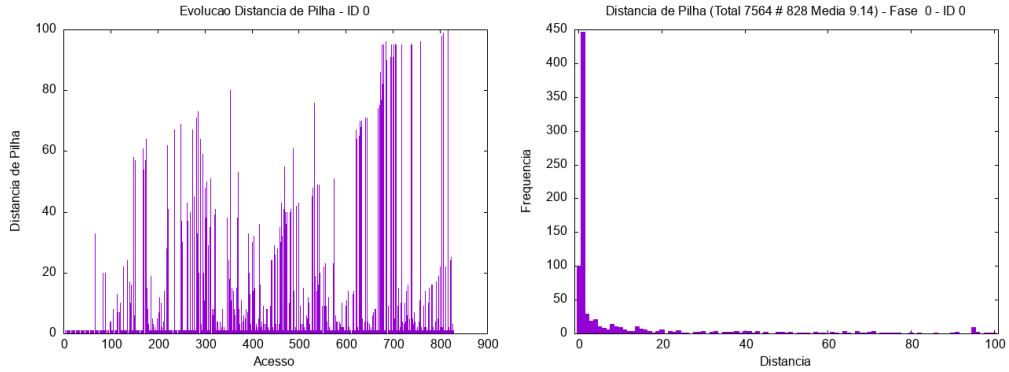
5.1.1 Quicksort Recursivo, Não Recursivo, e Empilha Inteligente

O Quicksort Recursivo, Não Recursivo e Empilha Inteligente possuem o mesmo padrão de comportamento. Podemos perceber através dos gráficos de acesso que a principal diferença está na ordenação das partições, onde o Recursivo ordena primeiro a subpartição da esquerda, o Não recursivo a subpartição da direita, e o Empilha Inteligente escolhe a menor partição para ordenar.



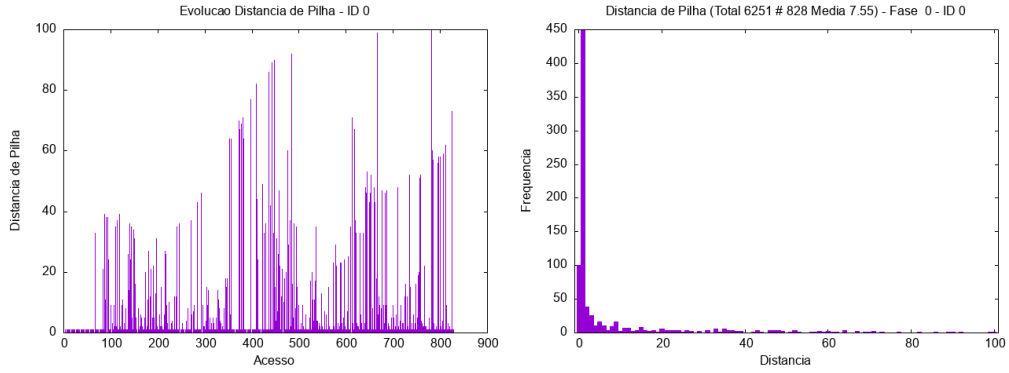
(a) Gráfico de Acesso Quicksort Recursivo (b) Gráfico de Acesso Quicksort Não Recursivo (c) Gráfico de Acesso Quicksort Empilha Inteligente

Figura 1: Gráficos de Acesso



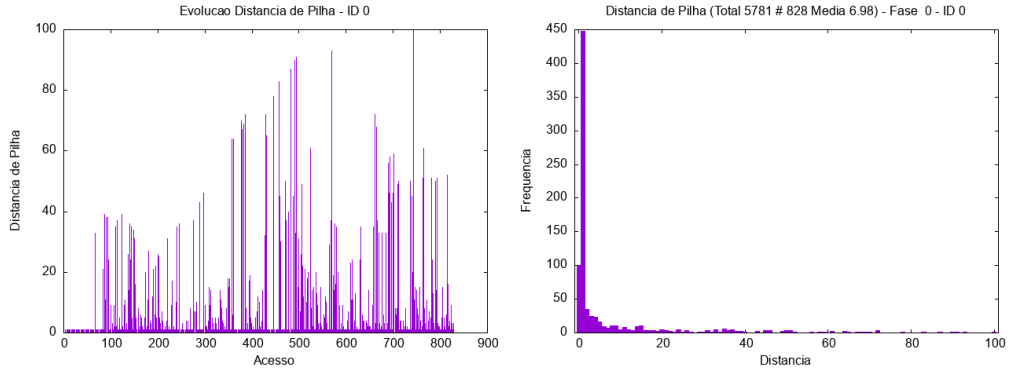
(a) Evolução de Distancia de Pilha (b) Distancia de Pilha

Figura 2: Gráfico QuicksortRecursivo



(a) Evolução de Distancia de Pilha (b) Distancia de Pilha

Figura 3: Gráfico QuicksortNaoRecursivo



(a) Evolução de Distancia de Pilha (b) Distancia de Pilha

Figura 4: Gráfico QuicksortInteligente

Já em relação a distancia de pilha, podemos perceber que o comportamento dos três métodos é extremamente parecido, mas podemos concluir que o Quicksort Inteligente é mais eficiente devido a menor distancia de pilha que foi 5781.

5.1.2 Quicksort Mediana (k)

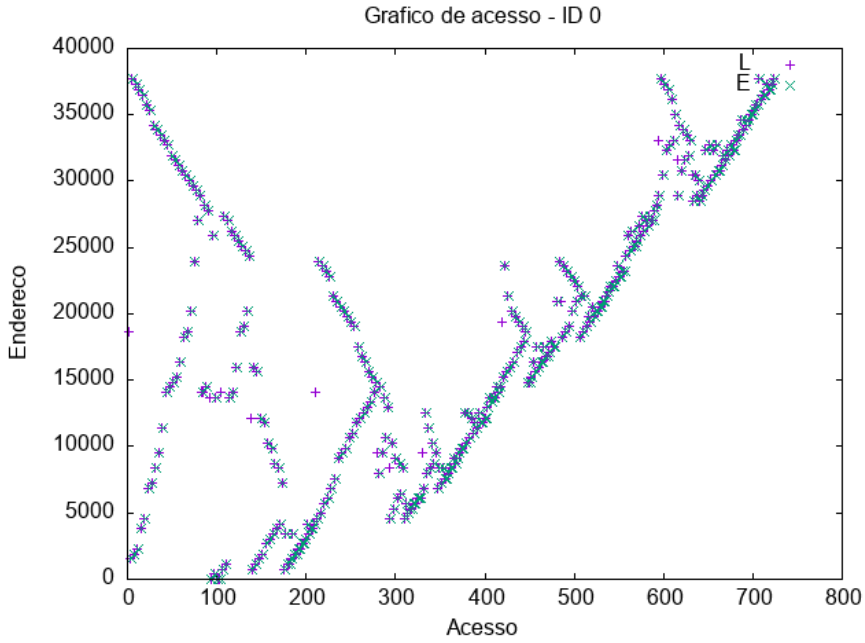
```
thais@ubuntu:~/Documentos/GitHub/CCOMP-UFMG/ED/TPs/TP2/analisaSH$ sh analisa.sh
rm -f bin/analisamem bin/testepilhaindexada obj/pilhaindexada.o obj/analisamem.o  obj/pilhaindexada.o obj/testepilhaindexada.o
gcc -g -Wall -c -Iinclude -o obj/pilhaindexada.o src/pilhaindexada.c
gcc -g -Wall -c -Iinclude -o obj/analisamem.o src/analisamem.c
gcc -g -o bin/analisamem obj/pilhaindexada.o obj/analisamem.o -lm
gcc -g -Wall -c -Iinclude -o obj/testepilhaindexada.o src/testepilhaindexada.c
gcc -g -o bin/testepilhaindexada obj/pilhaindexada.o obj/testepilhaindexada.o -lm
g++ -std=c++17 -g -Wall -c src/main.cpp -o obj/main.o -I./include/
g++ -std=c++17 -g -Wall -c src/memlog.cpp -o obj/memlog.o -I./include/
g++ -std=c++17 -g -Wall -c src/MetodosAuxiliares.cpp -o obj/MetodosAuxiliares.o -I./include/
g++ -std=c++17 -g -Wall -c src/MetodosOrdenacao.cpp -o obj/MetodosOrdenacao.o -I./include/
g++ -std=c++17 -g -Wall -c src/Pilha.cpp -o obj/Pilha.o -I./include/
g++ -std=c++17 -g -Wall -o ./bin/run.out  ./obj/main.o  ./obj/memlog.o  ./obj/MetodosAuxiliares.o  ./obj/MetodosOrdenacao.o  ./obj/Pilh
a.o
Enderecos:  [94817486788288-139709754418460] (44892267630172) #end 5611533453773
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
Aborted (core dumped)
thais@ubuntu:~/Documentos/GitHub/CCOMP-UFMG/ED/TPs/TP2/analisaSH$
```

Figura 5: Erro ao tentar gerar os gráfico para o Quicksort Mediana

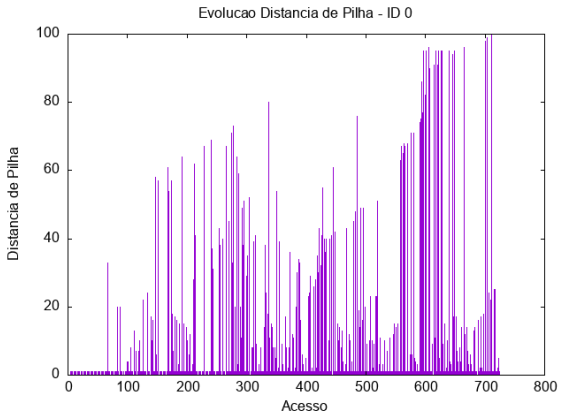
Infelizmente devido a problemas relacionados a pilhaindexada.c do analisamem do professor, não foi possível gerar os gráficos refentes ao Quicksort Mediana.

No entanto, podemos concluir alguns aspectos interessantes levando em consideração a construção do código. Como o valor para a mediana é escolhido aleatoriamente, podemos concluir que a função partição terá que subdividir mais o vetor, e por isso provavelmente terá um número maior de acessoas.

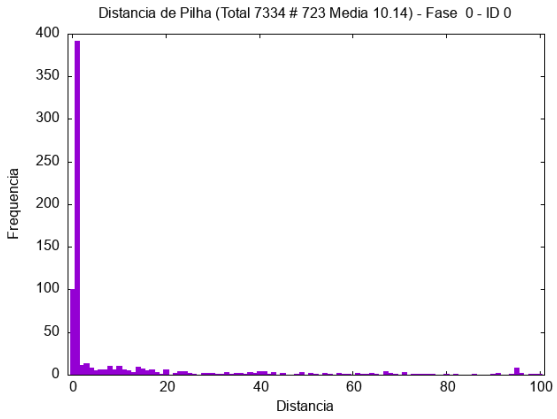
5.1.3 Quicksort Seleção (m)



(a) Grafico de Acesso



(b) Evolução de Distancia de Pilha

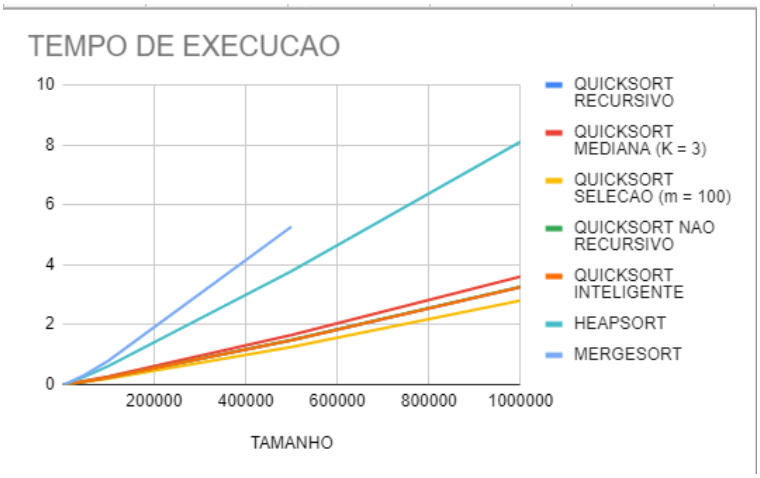


(c) Distancia de Pilha

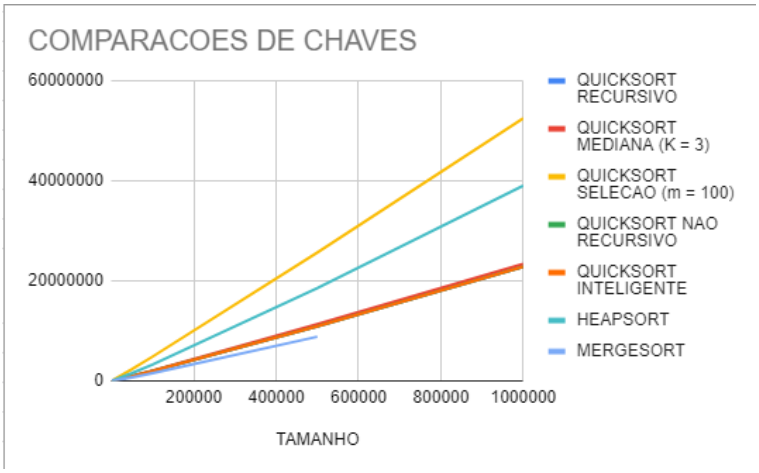
Figura 6: Gráfico QuicksortSelecao

Podemos perceber que através do gráfico de acessos que o método possui uma parte semelhante ao Quicksort Recursivo, e também um comportamento de acesso sequencial devido a utilização do Selectionsort. Já em relação a distancia de pilha, e sua evolução, podemos concluir que o seu funcionamento não é um dos mais eficientes entre os outros Quicksorts analisados.

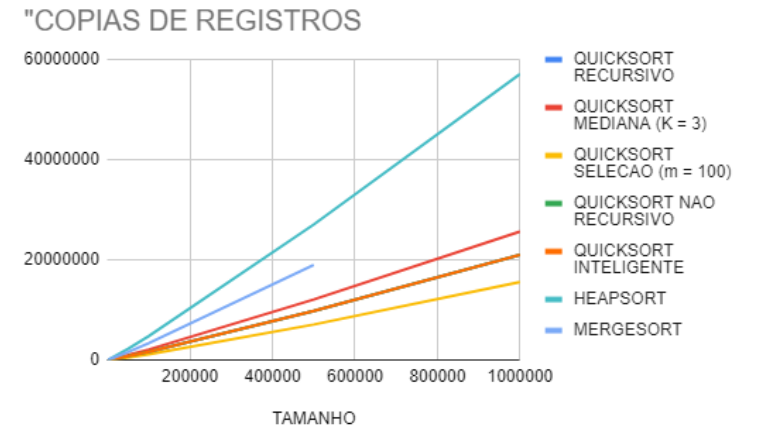
5.1.4 Tempo, Comparações e Cópias



(a) Tempos de Execução



(b) Comparaçoes de Chaves



(c) Copias de Registro

Figura 7: Gráfico QuicksortSelecao

Nessa parte do trabalho utilizamos da função timer() desenvolvida pelo professor e outras variáveis desenvolvidas no código para analisar o comportamento de cada um dos métodos.

Em relação ao tempo podemos perceber através da figura 7 que o Quicksorte Mediana possui o maior tempo entre os QuicksSorts devido ao custo de se ordenar o vetor mediana, já o Quicksort Seleção é o mais rápido o que comprova a eficiencia do SelectionSort para ordenar pequenos vetores.

Já em relação ao número de comparações, podemos concluir que os Quicksorts Recursivos, não recursivos e Empilha inteligente possuem o mesmo resultado para esse dado, enquanto o Quicksort seleção que é o mais rápido acabou por ter o maior custo nessa área.

Já em relação ao número de cópias o Quicksort seleção é também o mais eficiente. Tendo em vista que ele possui o menor número de cópias de registros entre as 3 opções citadas.

Também podemos perceber o comportamento semelhante do Quicksort Recursivo, Não Recursivo, e Empilha Inteligente, onde temos os valores de tempo, comparações e cópias bem próximos nos três casos.

QUICKSORT RECURSIVO

TAMANHO	TEMPO DE EXECUCAO	COMPARACOES DE CHAVES	COPIAS DE REGISTROS
1000	0.0011618	13331.6	8026.2
5000	0.0065168	74730.6	52368
10000	0.0152338	161652	116104
50000	0.105417	934677	735621
100000	0.23374	1.9695e+06	1.61061e+06
500000	1.4752	1.09296e+07	9.7647e+06
1000000	3.24884	2.28128e+07	2.10208e+07

QUICKSORT MEDIANA

TAMANHO	TEMPO DE EXECUCAO	COMPARACOES DE CHAVES	COPIAS DE REGISTROS
1000	0.0011954	14482.4	13685.4
5000	0.0080714	82589.6	77312.4
10000	0.0187156	170791	164810
50000	0.123056	953604	967669
100000	0.270297	2.01522e+06	2.06935e+06
500000	1.65131	1.13467e+07	1.20636e+07
1000000	3.60791	2.33885e+07	2.56502e+07

QUICKSORT SELECAO

TAMANHO	TEMPO DE EXECUCAO	COMPARACOES DE CHAVES	COPIAS DE REGISTROS
1000	0.0006116	39331	5704.2
5000	0.0050172	201057	37615.2
10000	0.0123568	426519	81779.4
50000	0.0874586	2.32729e+06	506094
100000	0.194453	4.84758e+06	1.11464e+06
500000	1.25783	2.56876e+07	7.07124e+06
1000000	2.80638	5.25295e+07	1.55721e+07

Figura 8: Dados do Quicksort

QUICKSORT NAO RECURSIVO

TAMANHO	TEMPO DE EXECUCAO	COMPARACOES DE CHAVES	COPIAS DE REGISTROS
1000	0.0008204	13331.6	8026.2
5000	0.0063736	74730.6	52368
10000	0.0155132	161652	116104
50000	0.106465	934677	735621
100000	0.235559	1.9695e+06	1.61061e+06
500000	1.48766	1.09296e+07	9.7647e+06
1000000	3.26891	2.28128e+07	2.10208e+07

QUICKSORT INTELIGENTE

TAMANHO	TEMPO DE EXECUCAO	COMPARACOES DE CHAVES	COPIAS DE REGISTROS
1000	0.0008166	13331.6	8026.2
5000	0.0061268	74730.6	52368
10000	0.0151218	161652	116104
50000	0.105224	934677	735621
100000	0.233361	1.9695e+06	1.61061e+06
500000	1.47541	1.09296e+07	9.7647e+06
1000000	3.25094	2.28128e+07	2.10208e+07

Figura 9: Dados do Quicksort

HEAPSORT

TAMANHO	TEMPO DE EXECUCAO	COMPARACOES DE CHAVES	COPIAS DE REGISTROS
1000	0.002101	19185.2	27277.8
5000	0.0151254	119137	171206
10000	0.037571	258241	372361
50000	0.271835	1.5243e+06	2.21145e+06
100000	0.614504	3.24841e+06	4.72262e+06
500000	3.78225	1.85359e+07	2.70538e+07
1000000	8.10262	3.90694e+07	5.71042e+07

Figura 10: Dados do Heapsort

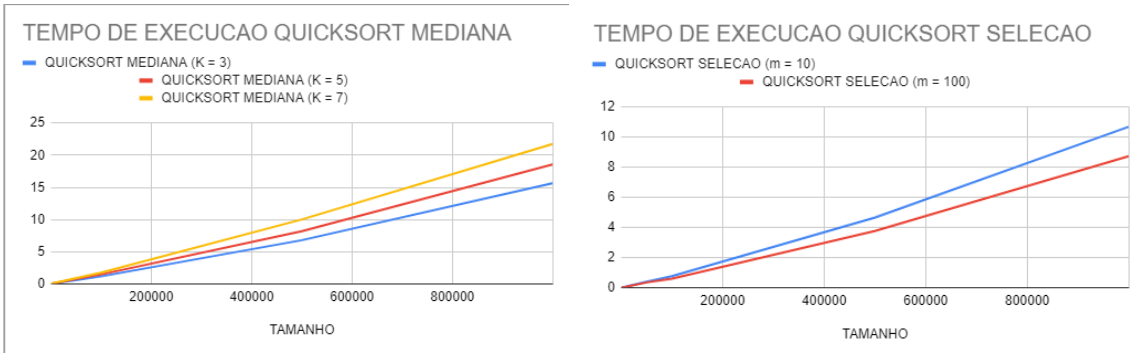
MERGESORT

TAMANHO	TEMPO DE EXECUCAO	COMPARACOES DE CHAVES	COPIAS DE REGISTROS
1000	0.0019866	8705.6	19952
5000	0.0170248	55219.2	123616
10000	0.043172	120462	267232
50000	0.333072	718041	1.56893e+06
100000	0.793925	1.53599e+06	3.33786e+06
500000	5.274	8.8353e+06	1.89514e+07
1000000	-	-	-

Figura 11: Dados do Mergesort

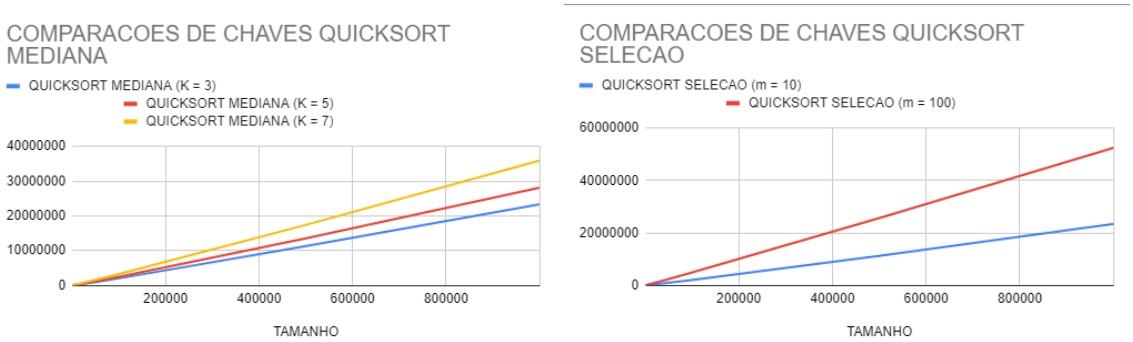
5.1.5 Impacto do valor de K e M

Outro aspecto interessante de analisarmos é os impactos causados pela mudança do valor do k no quicksort mediana, e de m no quicksort seleção.



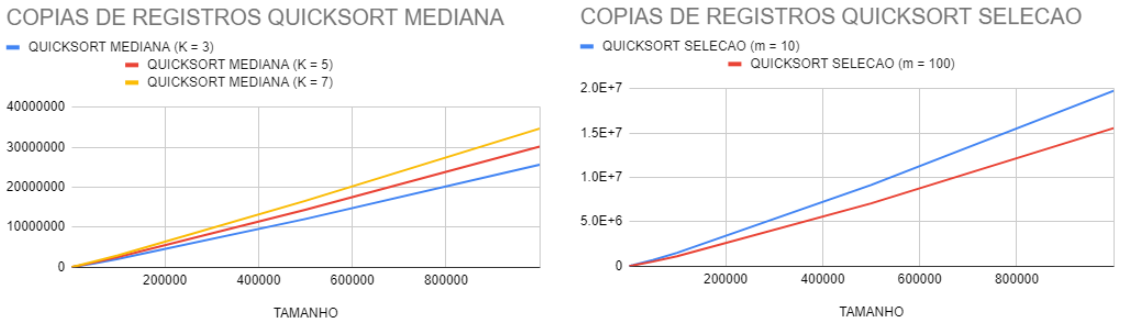
(a) Tempo de Execução Quicksort Mediana (b) Tempo de Execução Quicksort Seleção

Figura 12: Gráfico de Tempo de Execução



(a) Comparações de Chaves Quicksort Mediana (b) Comparações de Chaves Quicksort Seleção

Figura 13: Gráfico Comparações de Chaves



(a) Cópias de Registro Quicksort Mediana (b) Cópias de Registro Quicksort Seleção

Figura 14: Gráfico Cópias de Registro

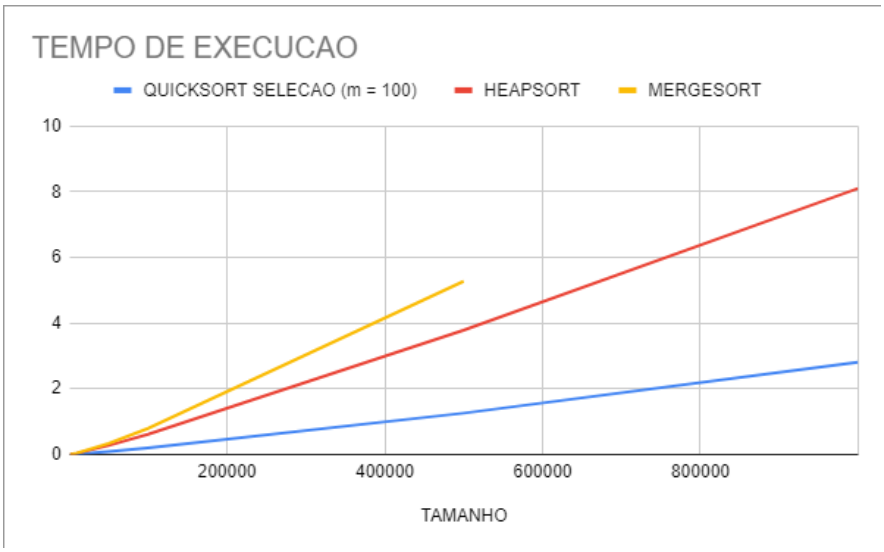
Em relação ao Quicksort mediana, quando maior o valor de k, maior o seu tempo de execução, comparações de chaves e copias de registro.

Já em relação ao Quicksort seleção podemos perceber que quando maior o valor de m, maior o número de comparações de chaves, mas menor é o seu tempo de execução e numero de cópias de registro.

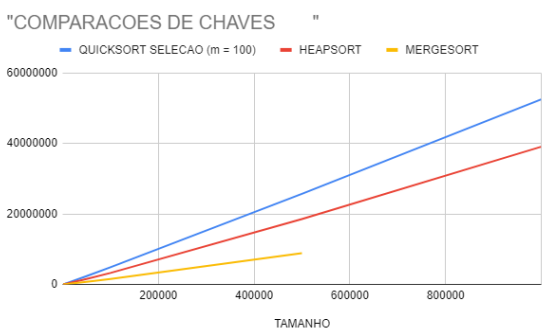
Esses comportamentos é causado pelo Selectionsort no Quicksort, tanto na hora de ordenar o vetor da mediana, quanto na hora de ordenar as pequenas partições da Seleção.

5.1.6 Quicksort X Mergesort X Heapsort

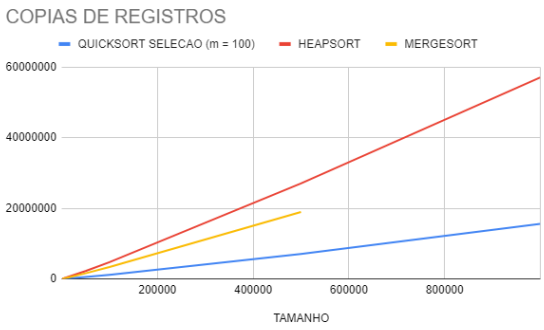
Por fim, podemos fazer uma comparação entre o Quicksort Seleção (Mais rápido e eficiente dos Quicksorts) com os métodos Heapsor e Mergesort.



(a) Tempo de Execução



(b) Comparações de Chaves



(c) Cópias de Registro

Figura 15: Quicksort X Mergesort X Heapsort

Analisando os gráficos da figura 15, podemos concluir que o Merge sort tem o maior tempo de execução, mas o menor numero de comparações. Já o Heapsort se destaca ao ter o maior número de cópias e ser o meio termo entre as três opções. E por fim o Quicksort Seleção (que é o melhor dos Quicksort) e de novo é o método mais eficiente, com o menor tempo de execução, e menor cópias de registro.

Infelizmente também é possível perceber que os dados do mergesort não foram calculados para todos os tamanho. Isso ocorreu pois os testes foram realizados em um notebook com 8gb de RAM,

o que foi insuficiente para ordenar um vetor de tamanho 1000000. Mas felizmente isso não impactou o resto da analise final do trabalho.

6 Conclusão

Pode-se perceber através desse trabalho a importância de realizar a analise de complexidade dos códigos implementados, para achar a melhor solução possível para um problema. Pudemos observar o comportamento de 7 métodos diferentes de se ordenar um vetor, e concluir que cada um possui um comportamento que o favorece ou desfavorece em determinados casos. Além disso, pude entender um pouco mais sobre alguns métodos de ordenação, mas diferentemente das aulas teóricas, o trabalho pratico 2 complementou o conhecimento adquirição em sala de aula construindo um ambiente onde pude testar as mudanças geradas em cada caso.

Outro aspecto importante é que mesmo não sendo solicitado, consegui utilizar e entender um pouco melhor o software Valgrind para descobrir se estava ocorrendo memory leaks na implementação dos métodos Quicksort Não Recursivo e Empilha Inteligente.

Por fim, as partes mais desafiadoras de tudo foram: a implementação do Quicksort Mediana por possuir várias pequenas etapas, a escolha e desenvolvimento do ambiente de teste, já que foi preciso testar o código várias vezes para diversos métodos, e várias modificações ao longo dos testes.

References

DOS, C. Quicksort.
Disponível em: <https://pt.wikipedia.org/wiki/Quicksort>.
Acesso em: 22 nov. 2022.

DOS, C. Selection sort.
Disponível em: <https://pt.wikipedia.org/wiki/Selection_sort>.
Acesso em: 22 nov. 2022.

WIKIPEDIA CONTRIBUTORS. Merge sort.
Disponível em: <https://en.wikipedia.org/wiki/Merge_sort>.
Acesso em: 22 nov. 2022.

JOÃO ARTHUR BRUNET. Estruturas de Dados e Algoritmos.
Disponível em: <https://joaoarthurbm.github.io/eda/posts/merge-sort/>.
Acesso em: 22 nov. 2022.

Merge Sort Algorithm - GeeksforGeeks.
Disponível em: <https://www.geeksforgeeks.org/merge-sort/>.
Acesso em: 22 nov. 2022.

C++ Program for Heap Sort - GeeksforGeeks.
Disponível em: <https://www.geeksforgeeks.org/cpp-program-for-heap-sort/>.
Acesso em: 22 nov. 2022.

DOS, C. Heapsort.
Disponível em: <https://pt.wikipedia.org/wiki/Heapsort>.
Acesso em: 22 nov. 2022.

Instruções de Compilação e Execução

1. Abrir o terminal
2. Entrar na pasta tp
   Comando: cd / <caminho para a pasta /tp >
3. Compilar o programa com o Makefile disponibilizado na pasta
   Comando: make
4. Executar o programa
   Comando: ./bin/<nome do arquivo.out>
   - i <nome do arquivo de entrada .txt> (nome do arquivo de entrada)
   - v <nome da versao desejada> (versão)
   - s <numero da seed> (valor da semente)
   - k <numero dos k elementos utilizados no QuicksortMediana> (valor de k)
   - m <numero m que limita o funcionamento do QuicksortSelecao> (valor de m)
   - o <nome do arquivo de saida .tx> (nome do arquivo de saída)
   - p <nome do arquivo de registro .out> (nome de saída do log)
   - l (ativa o memlog)
5. Após terminar de utilizar o programa, delete os arquivos desnecessários
   Comando: make clean
6. A versão 0 é extra e só funciona caso o computador tenha mais de 8gb de ram (O mergeSorte consome bastante ram do computador)

Exemplo de Compilação

Exemplos de compilação para todos os métodos disponíveis.

0 - Teste de todos os métodos
./bin/run.out -v 0 -k 3 -m 100 -s 10 -i ./entradas/input.txt -o ./saidas/output.txt
1 - Quicksorte Recursivo
./bin/run.out -v 1 -s 10 -i ./entradas/input.txt -o ./saidas/output.txt
2 - Quicksorte Mediana (k)
./bin/run.out -v 2 -k 3 -s 10 -i ./entradas/input.txt -o ./saidas/output.txt
3 - Quicksorte Seleção (m)
./bin/run.out -v 3 -m 100 -s 10 -i ./entradas/input.txt -o ./saidas/output.txt
4 - Quicksorte Não Recursivo
./bin/run.out -v 4 -s 10 -i ./entradas/input.txt -o ./saidas/output.txt
5 - Quicksorte Empilha Inteligente
./bin/run.out -v 5 -s 10 -i ./entradas/input.txt -o ./saidas/output.txt
6 - Mergesort
./bin/run.out -v 6 -s 10 -i ./entradas/input.txt -o ./saidas/output.txt
7 - Heapsort
./bin/run.out -v 7 -s 10 -i ./entradas/input.txt -o ./saidas/output.txt

Exemplo de Entrada

```
4
1000
5000
10000
50000
```

Figura 16: input.txt

Exemplo de Saída

```
--------------------------------------------------------------------------------
QUICKSORT RECURSIVO - TAMANHO: 1000

      SEED        TEMPO DE EXECUCAO    COMPARACOES DE CHAVES      COPIAS DE REGISTROS
        10                 0.007338                    12087                     8172
        15                 0.006206                    13982                     7947
        20                 0.004945                    12951                     8136
        25                 0.003708                    13654                     7959
        30                 0.006185                    13984                     7917
MEDIA GERAL:
     TODAS               0.0056764                  13331.6                    8026.2
--------------------------------------------------------------------------------
QUICKSORT RECURSIVO - TAMANHO: 5000

      SEED        TEMPO DE EXECUCAO    COMPARACOES DE CHAVES      COPIAS DE REGISTROS
        10                 0.041855                    74755                    52242
        15                 0.027342                    74497                    52371
        20                 0.024744                    71914                    52758
        25                 0.023054                    77442                    52278
        30                 0.023066                    75045                    52191
MEDIA GERAL:
     TODAS               0.0280122                  74730.6                    52368
```

Figura 17: output.txt