Trabalho Prático 2
Estruturas de Dados - Ordenação

Wallace Eduardo Pereira - 2021032013

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

wallaceep@ufmg.br

1. Introdução

Esta documentação lida com a comparação das diferentes formas de se ordenar um conjunto de elementos, tomando a chave como índice de comparação. Tem como objetivo apresentar uma série de comparações entre os algoritmos *quicksort, mergesort e heapsort*, além das comparações entre as variações do *quicksort*, trazendo dados de qual é o melhor algoritmo a ser utilizado considerando número de comparações, número de cópias, tempo de execução, tamanho do vetor, etc. Para isso, foi criado um vetor de registros de tamanho N que contém como atributos uma chave gerada aleatoriamente, um conjunto de 15 *strings* de tamanho 200 geradas aleatoriamente e um conjunto de 10 números reais também gerados aleatoriamente. Para a ordenação, foi tomada a chave como elemento de comparação e aplicada uma série de experimentos.

A seção 2 trata de como foi feita a implementação dos registros, dos algoritmos de ordenação e a forma utilizada para serem realizados os testes, já na seção 3 é apresentado a análise de complexidade dos algoritmos, que refletem diretamente no desempenho. Já na seção 5 foi realizada a análise experimental, onde foram desenvolvidas as comparações nos diferentes testes realizados e análise de localidade e referência.

2. Implementação

O programa foi desenvolvido na linguagem C++ e compilado pelo compilador G++ da GNU Compiler Collection.

2.1. Estruturas de Dados

A implementação do programa teve como base a estrutura de dados de uma lista linear, ou seja, um vetor estático. Assim, foi criado um vetor de registros que eram armazenados em posições contíguas de memória, e a inserção é realizada após o último registro com custo constante. A ilustração da estrutura de dados pode ser visualizado na figura a seguir:

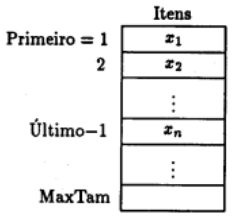


Figura 2.1: Implementação de uma lista através de arranjo

2.2. Classes

Para modularizar a implementação, foram utilizadas cinco classes principais, sendo elas: Quicksort, que tem os atributos do algoritmo e as funções que irão realizar os cinco tipos de variações do mesmo, Heapsort, que possui os atributos do algoritmo e as funções que compõem sua forma de ordenação, Mergesort, que assim como os outros possuem seus atributos e as funções que farão a ordenação, e, por fim, a classe Registros, que compõe os registros que serão ordenados, e inicializa seus atributos por meio de valores aleatórios. Além disso, possui a classe métricas, que auxilia retornando métricas importantes para análise e comparação dos diferentes algoritmos de ordenação.

2.2.1. Algoritmos de Ordenação

Dentre as funções presentes no programa, a maioria delas faz praticamente a mesma coisa de formas diferentes, ou seja, o objetivo é sempre ordenar o vetor de Registros de tamanho N tomando sua chave como índice. Porém, cada método de ordenação utiliza formas diferentes e com custos diferentes para realizar tal operação, sendo elas:

- **Quicksort:** Divide o problema de ordenar um conjunto de N itens em dois problemas menores, que são ordenados independentemente, e combinam os resultados para produzir o resultado final (dividir para conquistar).
    - **Quicksort Recursivo:** Realiza chamadas recursivas na função que ordena os elementos do vetor até que estejam todos ordenados.
    - **Quicksort Mediana:** Escolhe o pivô para partição como sendo a mediana de k elementos aleatórios do vetor, sendo k um número passado como parâmetro.
    - **Quicksort Seleção:** Versão do quicksort recursivo que utiliza a ordenação por seleção para ordenar partições do vetor com tamanho menor ou igual a m, sendo m um número inteiro passado como parâmetro.
    - **Quicksort não Recursivo:** Funciona da mesma forma que o quicksort recursivo, mas, ao invés das chamadas serem realizadas recursivamente, utiliza uma pilha para armazenar as partições que serão processadas posteriormente.
    - **Quicksort Empilha Inteligente:** Assim como o quicksort não recursivo, faz uso da pilha para armazenar as partições que serão processadas posteriormente, com a diferença que processa primeiro a menor partição.
- **Mergesort:** O algoritmo mergesort consiste no método “dividir para conquistar” baseado em *merging*, ou seja, é a combinação de dois vetores ordenados em um vetor maior que também está ordenado. Diferente do quicksort, o mergesort une dois vetores para criar um único, logo, faz duas chamadas recursivas, para a divisão e tem o procedimento de união dos vetores.
- **Heapsort:** O heapsort possui o mesmo princípio de funcionamento da ordenação por seleção, que consiste em selecionar o menor ou maior item do vetor e trocá-lo com o elemento na primeira ou última posição, com a diferença que o heapsort faz uso de fila de prioridade para reduzir o número de comparações, e, consequentemente, o custo da ordenação.

3. Análise de Complexidade

Esta seção apresenta a análise de complexidade de tempo e de espaço do programa, comparando as complexidades dos algoritmos de ordenação em alguns casos específicos.

3.1. Tempo

- **Quicksort:** Começando pelo quicksort, podemos dividir sua complexidade em melhor caso, pior caso e caso médio. Começando pelo melhor caso, no quicksort, ocorre quando cada partição divide o conjunto em duas partes iguais. Dessa forma, sua complexidade fica **T(n) = 2T(n/2) + n C(n) = O(n log n)**. No caso médio, segundo Sedgewick e Flajolet (1996, p. 17): **C(n) ≈ 1,386n log n – 0,846n**, ou seja, em média o tempo de execução do quicksort é **O(n log n)**. Já para o pior caso, ele ocorre quando o pivô escolhido é um dos extremos do vetor, fazendo com que o procedimento de ordenação seja chamado n vezes, eliminando apenas um item em cada chamada, ficando, assim, sua complexidade **T(n) = n + T(n-1) C(n) = O(n^2)**. Resumidamente, a análise de complexidade do quicksort fica da seguinte maneira:
    - Melhor caso: **O(n log n)**
    - Caso médio: **O(n log n)**
    - Pior caso: **O(n^2)**

Vale ressaltar que, dentre as variações implementadas do quicksort, temos o quicksort seleção, que utiliza o método de ordenação por seleção para ordenar partições de tamanho menor ou igual a m. Portanto, nesse caso, devemos também considerar a complexidade da ordenação por seleção, que é **C(n) = O(n^2)**.

- **Mergesort:** O custo do algoritmo de ordenação mergesort é obtido somando o custo da operação de junção (merge) + o custo de ordenação, ou seja, temos que o custo do mergesort é dado pela equação **T(n) = 2T(n/2) + n = O(n log n)**. Resolvendo a equação de recorrência por teorema mestre, chegamos no segundo caso, ou seja, a função de complexidade do mergesort é **T(n) = Θ (n log n)**.

- **Heapsort:** Para o heapsort, o custo de tempo é resultado dos custos das funções constrói, que constrói o heap, e refaz, que reconstrói o heap, sendo assim, o custo do heapsort é de **C(n) = O(n log n).**

3.2. Espaço

- **Quicksort:**
    - Melhor caso: **Θ(log_2n)**
    - Caso médio: **Θ(log_2n)**
    - Pior caso: **Θ(n)**

Assim como na complexidade de tempo, para o caso do algoritmo quicksort seleção, é preciso também considerar a complexidade de espaço do algoritmo de ordenação por seleção, sendo esse: **O(1)**, como a única memória extra usada é para variáveis temporárias durante a

troca de dois valores no Array. A classificação por seleção nunca faz mais do que trocas **O(n)** e pode ser útil quando a gravação na memória é uma operação cara.

- **Mergesort: Θ(n).** O Mergesort usa um vetor auxiliar de tamanho n para fazer a intercalação, mas o espaço ainda é **Θ(n)**. O Mergesort é útil para ordenação externa, quando não é possível armazenar todos os elementos na memória primária.

- **Heapsort: O(1).**

4. Estratégias de Robustez

Para que o sistema funcione corretamente e não apresente erros inesperados, foram implementadas algumas estratégias de robustez. Elas são divididas entre as que comprometem o funcionamento do programa, e por isso devem parar a execução e as que não comprometem e podem fazer a execução seguir normalmente. Para esse primeiro caso, existem os seguintes comportamentos:

- Caso o usuário não passe algum argumento necessário para execução do algoritmo, o funcionamento do programa é interrompido e é apresentada uma mensagem de erro na tela.
- Caso o usuário não defina o tipo de algoritmo necessário para a ordenação do vetor de registros, ou utilize um algoritmo que não esteja disponível dentre as opções, o programa tem sua execução interrompida e é apresentada uma mensagem de erro na tela.
- Caso o usuário passe como parâmetro um valor impossível de ser calculado no quicksort seleção ou quicksort mediana (ou seja, um valor menor ou igual à zero ou maior que N), o programa tem sua execução interrompida e é apresentada uma mensagem de erro na tela.
- Caso ocorra algum problema ao abrir o arquivo de entrada/saída, é interrompida a execução e apresentada uma mensagem de erro.

Para o segundo caso, ou seja, casos em que a execução não é comprometida pelo erro, segue-se normalmente sua execução, e emite os seguintes avisos:

- Se o usuário não passar o parâmetro **-o** seguido do nome de saída, por padrão, o arquivo será nomeado como **saida.txt**

Como o programa em si é muito simples, não foi necessário a utilização de muitas estratégias de robustez.

5. Análise Experimental

A análise experimental foi dividida em duas partes, sendo a primeira a comparação das cinco variações do quicksort citadas acima (recursivo, mediana, seleção, iterativo, empilha inteligente). As comparações levam em consideração o número de cópias, o número de comparações, o tempo de execução e a análise de localidade de referência. A partir das análises da primeira parte, a “melhor” variação do quicksort será comparada novamente, mas

com os algoritmos mergesort e heapsort, levando em consideração os mesmos critérios citados.

Ademais, é importante salientar que a análise experimental foi realizada inicializando somente o atributo chave de Registro, pois ao inicializar os demais, a máquina não conseguiu processar para grandes valores de N.

5.1. Variações do quicksort

Nessa seção serão analisadas as cinco variações do quicksort tomando como critérios de comparação o número de cópias, número de comparações, tempo de execução e localidade de referência.

Levando em consideração as 3 métricas, foram realizadas uma série de análises nas variações do quicksort, considerando a semente do *rand* que gera chaves aleatórias como 10, 50, 125, 250 e 500, foram obtidos os seguintes resultados:

*Deixando claro que não foi realizada a análise do empilha inteligente porque o não recursivo já é o empilha inteligente.

quicksort recursivo

Versão	Semente	N	Num Comparações	Num Cópias	Tempo Execução
1	10	1000	8505	3449	0.008573
1	10	5000	51960	19903	0.020398
1	10	10000	103555	42471	0.043382
1	10	50000	663453	237948	0.172301
1	10	100000	1369522	500261	0.433494
1	10	500000	8139847	2770060	2.82067
1	10	1000000	16328013	5804890	5.76416
1	50	1000	7859	3494	0.009334
1	50	5000	47344	20167	0.022018
1	50	10000	106361	42391	0.04565
1	50	50000	596818	240908	0.176461
1	50	100000	1374131	500464	0.437767
1	50	500000	7607236	2783495	1.92111
1	50	1000000	16830760	5786434	5.49499
1	125	1000	8981	3412	0.008788
1	125	5000	48434	20027	0.020791
1	125	10000	109398	42526	0.045993
1	125	50000	600567	241434	0.17364
1	125	100000	1538324	496801	0.436185
1	125	500000	7641198	2786088	2.2751
1	125	1000000	17655371	5725459	6.96016
1	250	1000	7132	3483	0.006326
1	250	5000	48358	20135	0.017998
1	250	10000	118152	41947	0.043772
1	250	50000	735235	234399	0.165916
1	250	100000	1406505	499571	0.433508
1	250	500000	8477280	2757443	1.8714
1	250	1000000	17308806	5781718	5.02285
1	500	1000	8000	3447	0.007775
1	500	5000	52741	19864	0.019396
1	500	10000	117394	42167	0.0454
1	500	50000	629058	239386	0.171347
1	500	100000	1586200	493614	0.428505
1	500	500000	7801166	2775443	1.86873
1	500	1000000	16039105	5801405	4.83512
Média		1000	8095,4	3457	0.0081592
		5000	49767,4	20019,2	0.0201202
		10000	110972	42300,4	0.0448394
		50000	645026,2	238815	0.171933
		100000	1454936,4	498142,2	0.433892
		500000	7933345,4	2774505,8	2.1514
		1000000	16832411	5779981,2	5.61546

quicksort mediana

Versão	Semente	N	m Comparaçõ	Num Cópias	empo Execuçã	k
2	10	1000	7957	3602	0.00872	3
2	10	5000	48320	20807	0.020002	3
2	10	10000	108555	43736	0.045911	3
2	10	50000	608232	247758	0.173545	3
2	10	100000	1396999	514065	0.43623	3
2	10	500000	7719963	2850294	2.68048	3
2	10	1000000	17056116	5919850	6.04892	3
2	10	1000	7936	3456	0.00891	5
2	10	5000	50333	19987	0.023105	5
2	10	10000	106281	42411	0.050355	5
2	10	50000	608881	239973	0.17969	5
2	10	100000	1302306	505291	0.444185	5
2	10	500000	7688117	2781748	1.96063	5
2	10	1000000	17142608	5760767	5.49272	5
2	10	1000	7117	3505	0.006838	7
2	10	5000	44402	20164	0.018622	7
2	10	10000	111819	42224	0.042964	7
2	10	50000	617674	239414	0.175759	7
2	10	100000	1359364	501046	0.515993	7
2	10	500000	7853465	2786119	2.10849	7
2	10	1000000	17657476	5747265	5.6538	7
2	50	1000	8710	3431	0.008878	3
2	50	5000	47188	20112	0.02052	3
2	50	10000	114737	42173	0.047405	3
2	50	50000	633946	238764	0.174904	3
2	50	100000	1541886	495163	0.487408	3
2	50	500000	8090778	2777004	2.11393	3
2	50	1000000	17744938	5769186	5.58157	3
2	50	1000	8774	3469	0.007855	5
2	50	5000	53026	19923	0.022271	5
2	50	10000	97664	42827	0.046426	5
2	50	50000	618459	239745	0.180993	5
2	50	100000	1406739	499754	0.467946	5
2	50	500000	8360703	2766332	1.97092	5
2	50	1000000	17205249	5769434	6.02276	5
2	50	1000	9561	3421	0.005785	7
2	50	5000	48552	20148	0.01815	7
2	50	10000	119115	42256	0.041721	7
2	50	50000	626242	240083	0.169379	7
2	50	100000	1303234	504486	0.43259	7
2	50	500000	7804130	2776468	2.14161	7
2	50	1000000	17185514	5788107	5.74685	7
2	125	1000	6450	3534	0.008597	3
2	125	5000	53003	19945	0.019986	3
2	125	10000	116312	41955	0.042572	3
2	125	50000	616437	239575	0.178619	3

quicksort seleção

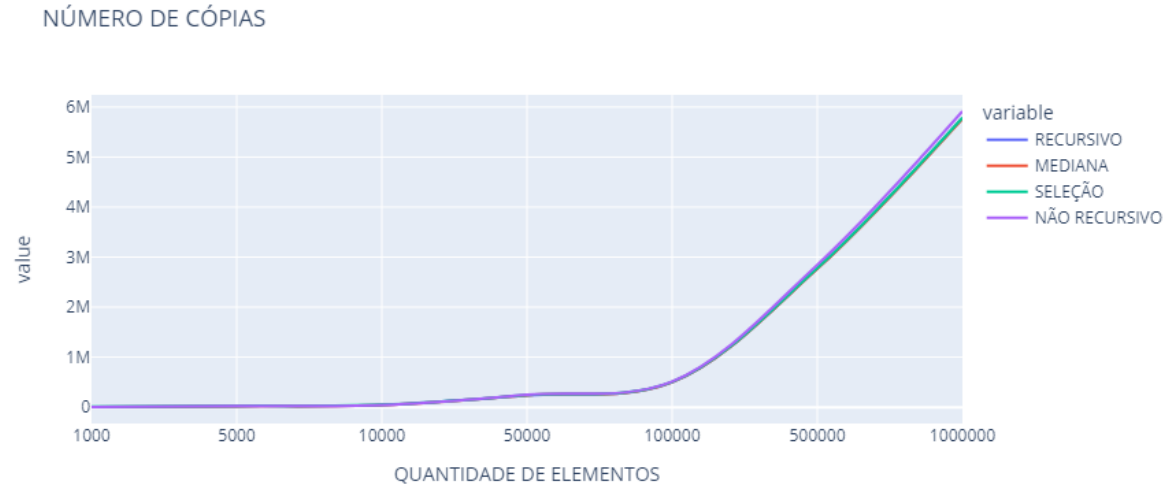
Versão	Semente	N	m Comparaçõ	Num Cópias	empo Execuçã	m
3	10	1000	8506	3510	0.008233	10
3	10	5000	51963	19944	0.024128	10
3	10	10000	103559	42536	0.056941	10
3	10	50000	663457	237996	0.224033	10
3	10	100000	1369529	500320	0.485041	10
3	10	500000	8139853	2770163	2.10681	10
3	10	1000000	16328018	5804952	5.54797	10
3	10	1000	8828	9995	0.011072	100
3	10	5000	52169	26590	0.024704	100
3	10	10000	103835	49537	0.052825	100
3	10	50000	663638	245015	0.19106	100
3	10	100000	1369793	505887	0.526744	100
3	10	500000	8140070	2776255	2.34755	100
3	10	1000000	16328254	5811381	5.41784	100
3	50	1000	7862	3553	0.006332	10
3	50	5000	47349	20227	0.018362	10
3	50	10000	106361	42453	0.04861	10
3	50	50000	596820	240985	0.180525	10
3	50	100000	1374141	500538	0.434535	10
3	50	500000	7607239	2783561	1.97617	10
3	50	1000000	16830762	5786494	5.68896	10
3	50	1000	8080	10574	0.012689	100
3	50	5000	47492	26635	0.028824	100
3	50	10000	106598	49153	0.056434	100
3	50	50000	597007	247894	0.19234	100
3	50	100000	1374368	506753	0.449478	100
3	50	500000	7607441	2789987	1.91246	100
3	50	1000000	16830973	5792891	5.43461	100
3	125	1000	8990	3458	0.007908	10
3	125	5000	48446	20079	0.020738	10
3	125	10000	109403	42604	0.047617	10
3	125	50000	600576	241518	0.183593	10
3	125	100000	1538335	496880	0.448899	10
3	125	500000	7641204	2786164	1.92148	10
3	125	1000000	17655375	5725541	5.5668	10
3	125	1000	9176	10192	0.009772	100
3	125	5000	48741	26247	0.025056	100
3	125	10000	109590	49055	0.05486	100
3	125	50000	600854	248270	0.191609	100
3	125	100000	1538727	503329	0.46291	100
3	125	500000	7641374	2792907	2.12911	100
3	125	1000000	17655618	5732113	5.46663	100
3	250	1000	7137	3593	0.008504	10
3	250	5000	48365	20193	0.021307	10
3	250	10000	118152	42022	0.045025	10
3	250	50000	735240	234455	0.178986	10

quicksort iterativo

Versão	Semente	N	m Comparaçõ	Num Cópias	empo Execuçã
4	10	1000	8715	3578	0.00669
4	10	5000	53064	20581	0.018287
4	10	10000	105796	43854	0.041338
4	10	50000	674833	244587	0.171892
4	10	100000	1392695	513679	0.455632
4	10	500000	8250863	2836376	1.99939
4	10	1000000	16551493	5938973	5.30605
4	50	1000	7957	3602	0.00872
4	50	5000	48320	20807	0.020002
4	50	10000	108555	43736	0.045911
4	50	50000	608232	247758	0.173545
4	50	100000	1396999	514065	0.43623
4	50	500000	7719963	2850294	2.68048
4	50	1000000	17056116	5919850	6.04892
4	125	1000	9200	3552	0.006059
4	125	5000	49815	20813	0.018779
4	125	10000	111777	43869	0.042517
4	125	50000	612022	247874	0.170211
4	125	100000	1560401	510120	0.464008
4	125	500000	7753677	2853530	2.03539
4	125	1000000	17882817	5858947	5.46096
4	250	1000	7957	3602	0.00872
4	250	5000	48320	20807	0.020002
4	250	10000	108555	43736	0.045911
4	250	50000	608232	247758	0.173545
4	250	100000	1396999	514065	0.43623
4	250	500000	7719963	2850294	2.68048
4	250	1000000	17056116	5919850	6.04892
4	500	1000	8231	3591	0.007427
4	500	5000	53948	20582	0.023788
4	500	10000	119905	43603	0.049532
4	500	50000	640039	245928	0.190827
4	500	100000	1608617	507061	0.458708
4	500	500000	7913647	2841929	2.17902
4	500	1000000	16263138	5935550	5.7001
Média		1000	8412	3585	0.0075232
		5000	50693,4	20718	0.0201716
		10000	110917,6	43759,6	0.0450418
		50000	628671,6	246781	0.176004
		100000	1471142,2	511798	0.450162
		500000	7871622,6	2846484,6	2.31495
		1000000	16961936	5914634	5.71299

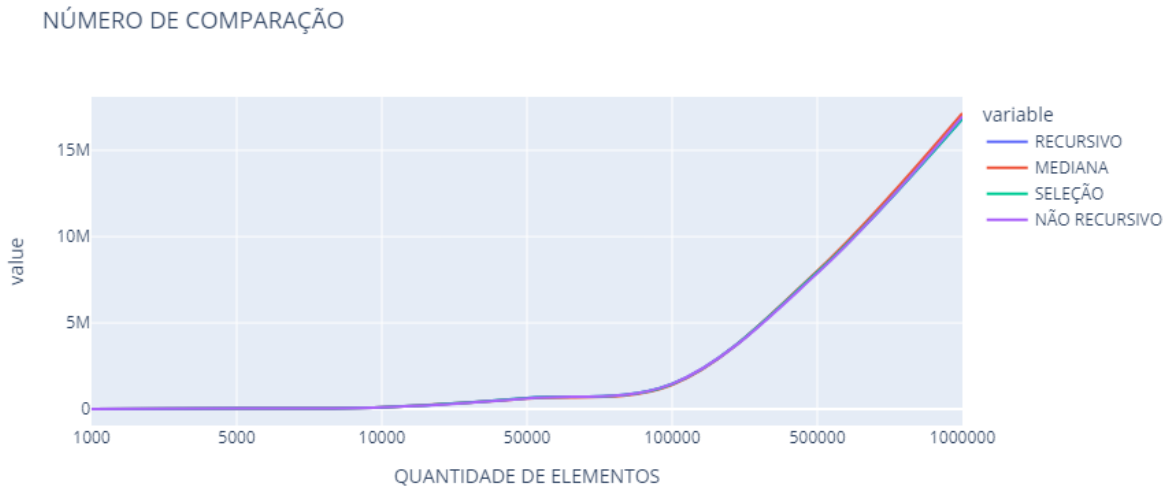
As análises foram feitas utilizando 5 sementes diferentes para gerar os vetores aleatórios de tamanhos 1000, 5000, 10000, 50000, 100000, 500000 e 1000000 e obtendo a média dos resultados. Além disso, também foram realizados experimentos alterando o valor de k para o quicksort mediana, testando com os números k = 3, 5 e 7, e o valor de m para o quicksort seleção, testando com os números m = 10 e 100. À partir das médias dos resultados, foram gerados gráficos de comparação entre os algoritmos considerando os três fatores:

5.1.1. Número de Cópias



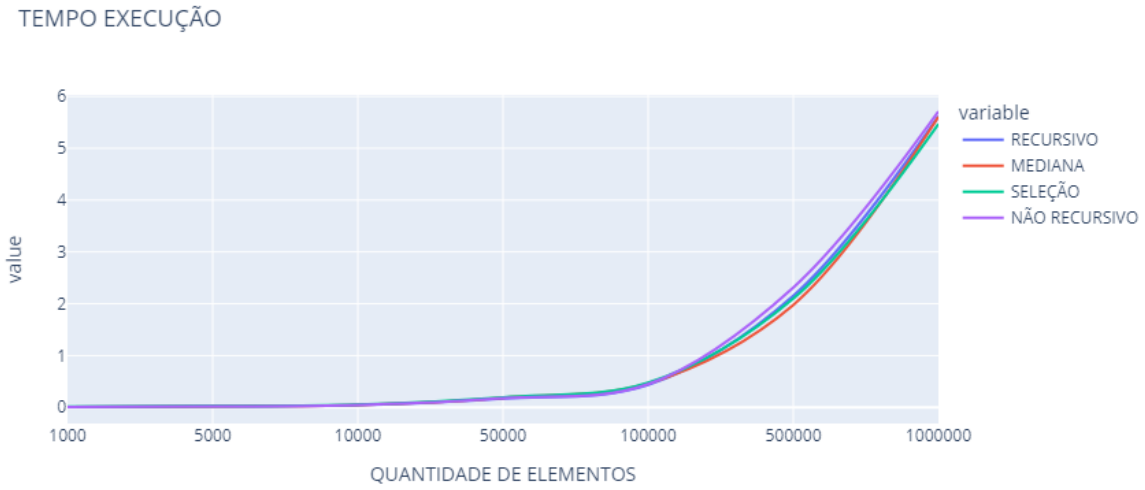
É possível observar, por meio do gráfico, que a quantidade de cópias realizadas pelos algoritmos é muito similar, não tendo muita alteração conforme varia o tamanho do vetor.

5.1.2. Número de Comparações



É possível observar, por meio do gráfico, que assim como o número de cópias, a quantidade de comparações realizadas pelos algoritmos é muito similar, não tendo muita alteração conforme varia o tamanho do vetor.

5.1.3. Tempo de Execução

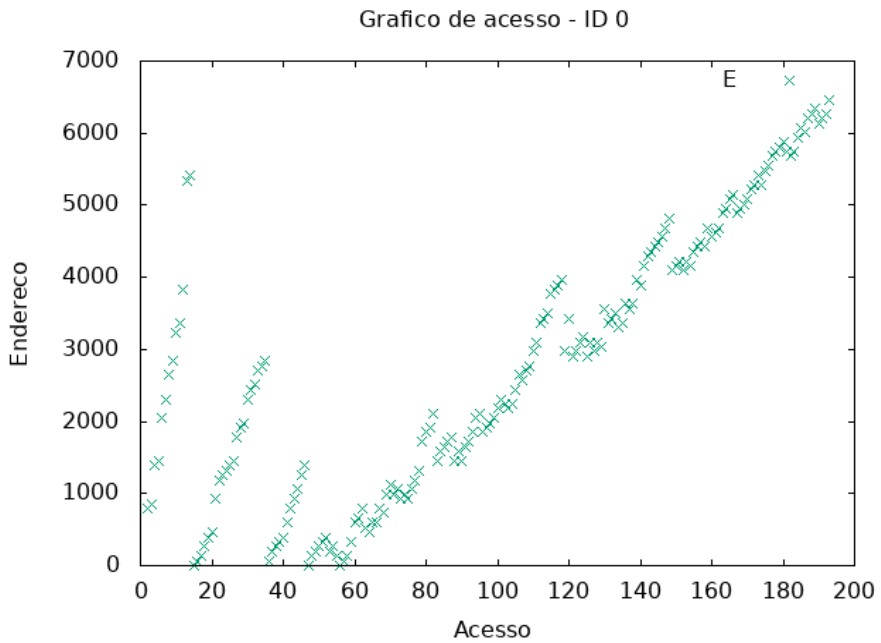


De acordo com o tempo de execução, é possível observar, por meio do gráfico, que assim como o número de comparações e o número de cópias, o tempo de execução realizados pelos algoritmos é muito similar para pequenos valores de N, porém ocorre uma leve alteração conforme N cresce, sendo o mediana com menor tempo de execução para maiores valores de N.

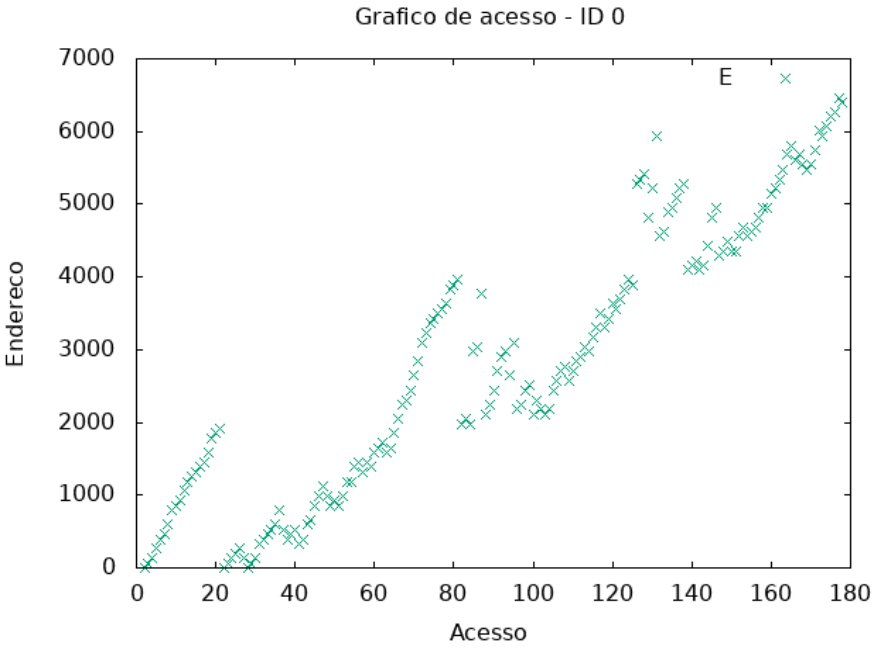
5.1.4. Localidade de Referência

Seguindo a localidade de referência, como mostrado nos gráficos a seguir, todas as variações do quicksort seguem o mesmo padrão, portanto, o que será explicado a seguir é referente ao quicksort recursivo, mas também se aplica às outras variações. Os gráficos foram gerados a partir da execução do programa no acesso de memória em suas comparações de chaves dos registros.

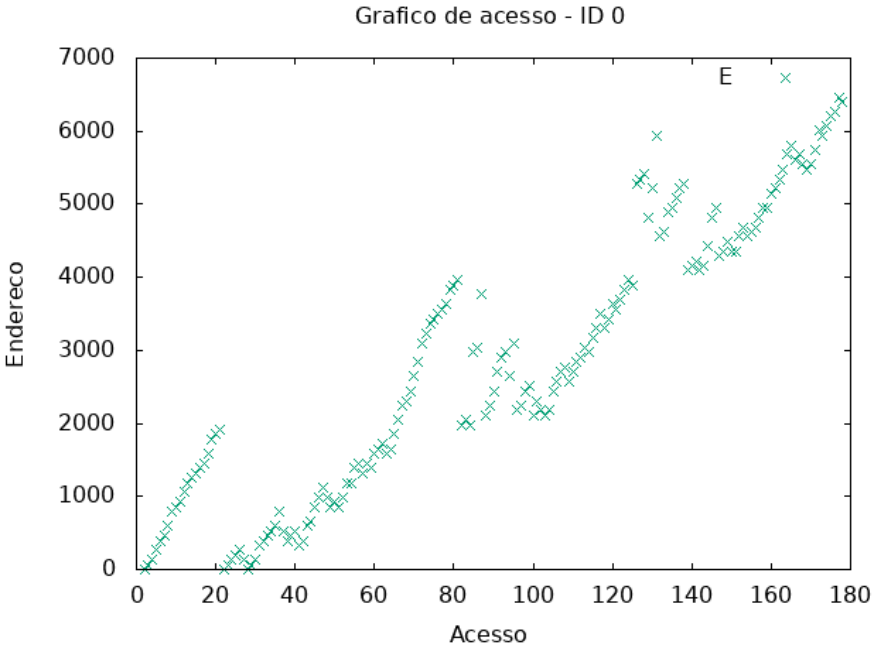
Quicksort Recursivo:



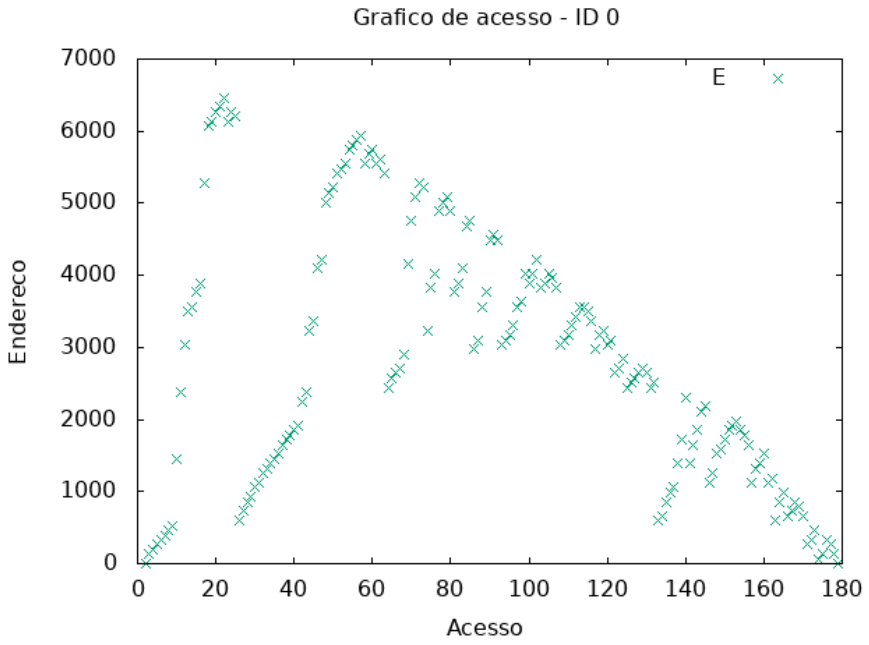
Quicksort Mediana:



Quicksort Seleção:



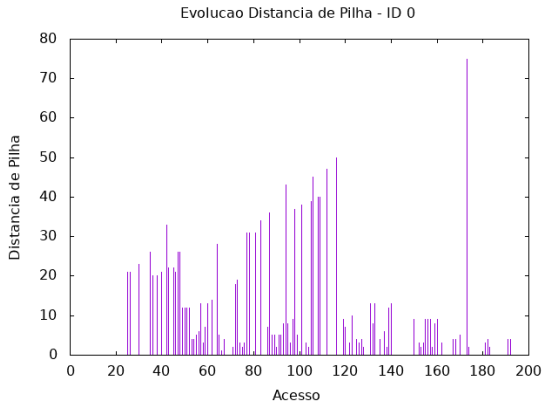
Quicksort Não Recursivo / Empilha Inteligente:

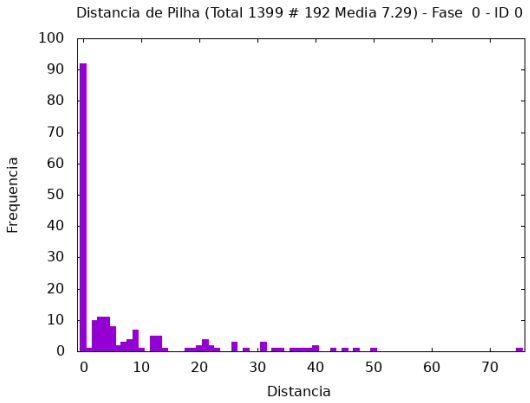


É possível visualizar que os endereços de memória são acessados conforme são realizadas as comparações, e, conforme um registro está na sua devida posição, ele não é mais acessado, enquanto algum outro que no processo de ordenar a partição foi trocado, mas ainda não está em seu devido lugar, é comparado novamente (ou seja, mais uma vez é acessado seu endereço de memória) e trocado de lugar até que esteja na posição correta. O processo ocorre até que o vetor esteja ordenado.

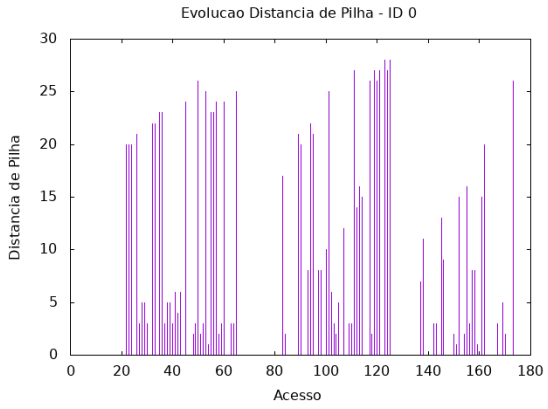
Assim como nos gráficos de acesso, os gráficos de evolução e distância de pilha são muito semelhantes para todos os algoritmos, já que todos realizam as mesmas operações, ainda que de maneiras diferentes.À partir do gráfico, é perceptível a diminuição da distância de pila conforme os acessos, que ocorre por meio da ordenação, ou seja, se um elemento já foi comparado e já está em sua devida posição, ele não precisa ser acessado de novo. Isso ocorre até que o vetor esteja totalmente ordenado. Assim como a evolução da distância de pilha, a frequência de acesso também diminui pelo mesmo motivo.

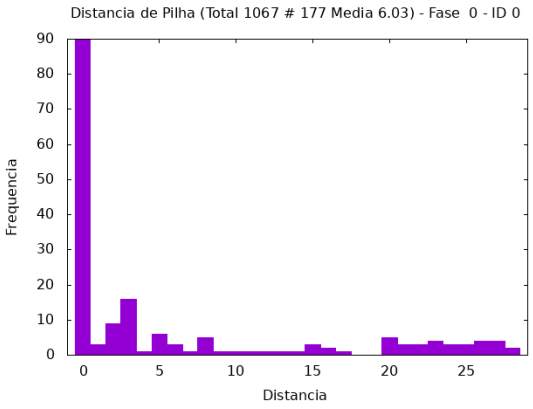
Quicksort Recursivo:



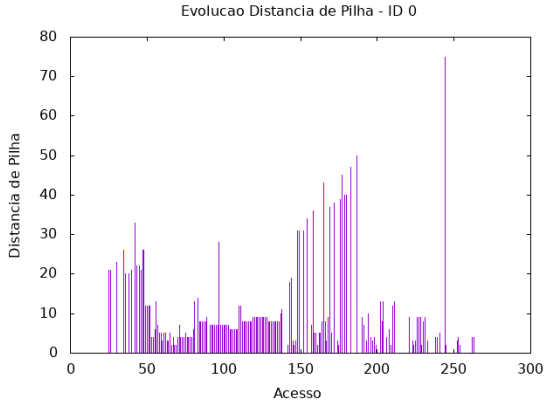


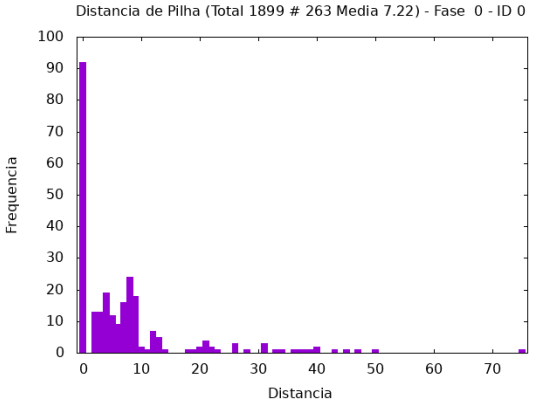
Quicksort Mediana:



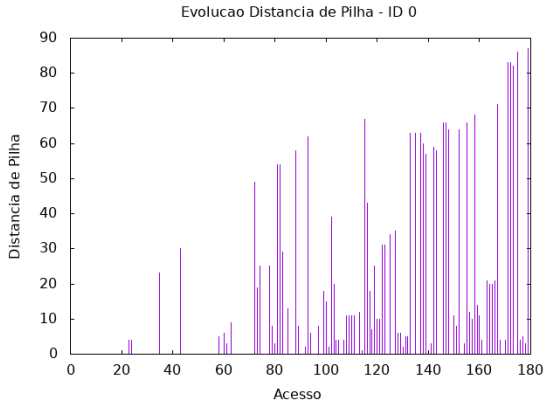


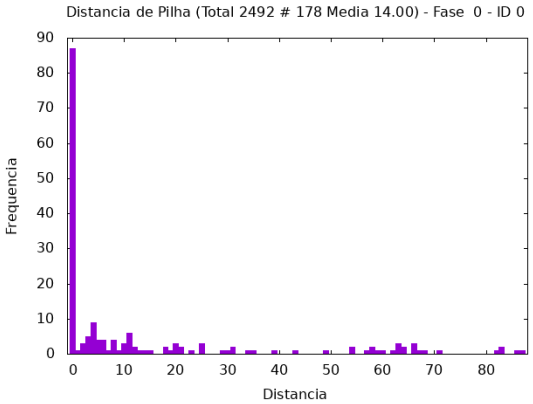
Quicksort Seleção:





Quicksort Não Recursivo / Empilha Inteligente:





Dado que o número de comparações, o número de cópias e a localidade de referência dentre as variações do quicksort apresentadas foram muito próximos, é perceptível que independente da variação, o quicksort é um algoritmo de ordenação com custo baixo. Partindo do tempo de execução, para valores maiores de N, é perceptível que o quicksort mediana teve menor tempo de execução, tomando mediana de 5, e será comparado com os outros algoritmos de ordenação Heapsort e Mergesort nos mesmos critérios apresentados.

5.2. Quicksort X Mergesort X Heapsort

Analisando as métricas do Mergesort e do Heapsort, e, em seguida, comparando com o Quicksort mediana (k=5).

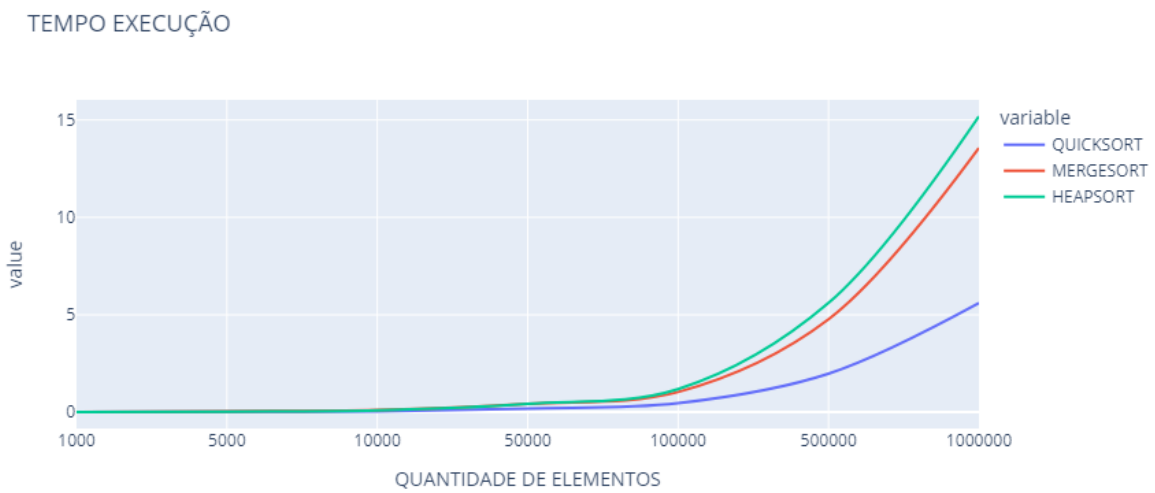
Mergesort:

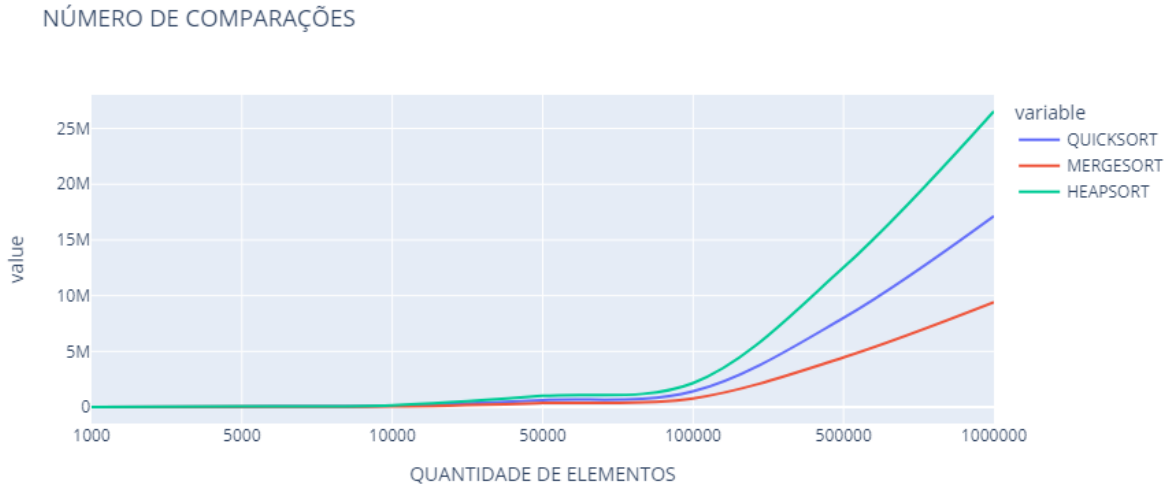
Versão	Semente	N	Num Comparações	Num Cópias
1	10	1000	8505	3449
1	10	5000	51960	19903
1	10	10000	103555	42471
1	10	50000	663453	237948
1	10	100000	1369522	500261
1	10	500000	8139847	2770060
1	10	1000000	16328013	5804890
1	50	1000	7859	3494
1	50	5000	47344	20167
1	50	10000	106361	42391
1	50	50000	596818	240908
1	50	100000	1374131	500464
1	50	500000	7607236	2783495
1	50	1000000	16830760	5786434
1	125	1000	8981	3412
1	125	5000	48434	20027
1	125	10000	109398	42526
1	125	50000	600567	241434
1	125	100000	1538324	496801
1	125	500000	7641198	2786088
1	125	1000000	17655371	5725459
1	250	1000	7132	3483
1	250	5000	48358	20135
1	250	10000	118152	41947
1	250	50000	735235	234399
1	250	100000	1406505	499571
1	250	500000	8477280	2757443
1	250	1000000	17308806	5781718
1	500	1000	8000	3447
1	500	5000	52741	19864
1	500	10000	117394	42167
1	500	50000	629058	239386
1	500	100000	1586200	493614
1	500	500000	7801166	2775443
1	500	1000000	16039105	5801405
Média		1000	8095,4	3457
		5000	49767,4	20019,2
		10000	110972	42300,4
		50000	645026,2	238815
		100000	1454936,4	498142,2
		500000	7933345,4	2774505,8
		1000000	16832411	5779981,2

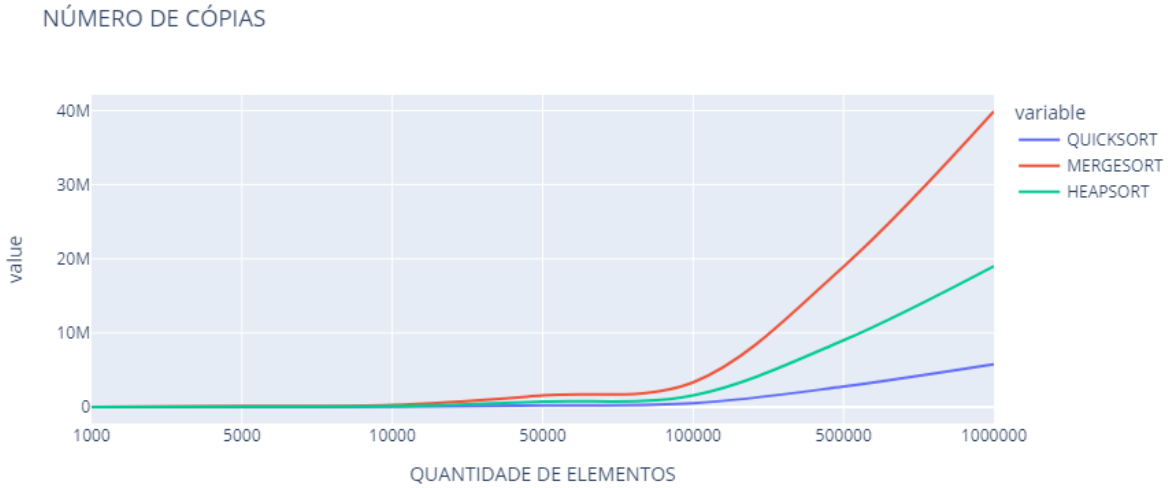
Heapsort:

Versão	Semente	N	Num Comparações	Num Cópias
1	10	1000	8505	3449
1	10	5000	51960	19903
1	10	10000	103555	42471
1	10	50000	663453	237948
1	10	100000	1369522	500261
1	10	500000	8139847	2770060
1	10	1000000	16328013	5804890
1	50	1000	7859	3494
1	50	5000	47344	20167
1	50	10000	106361	42391
1	50	50000	596818	240908
1	50	100000	1374131	500464
1	50	500000	7607236	2783495
1	50	1000000	16830760	5786434
1	125	1000	8981	3412
1	125	5000	48434	20027
1	125	10000	109398	42526
1	125	50000	600567	241434
1	125	100000	1538324	496801
1	125	500000	7641198	2786088
1	125	1000000	17655371	5725459
1	250	1000	7132	3483
1	250	5000	48358	20135
1	250	10000	118152	41947
1	250	50000	735235	234399
1	250	100000	1406505	499571
1	250	500000	8477280	2757443
1	250	1000000	17308806	5781718
1	500	1000	8000	3447
1	500	5000	52741	19864
1	500	10000	117394	42167
1	500	50000	629058	239386
1	500	100000	1586200	493614
1	500	500000	7801166	2775443
1	500	1000000	16039105	5801405
Média		1000	8095,4	3457
		5000	49767,4	20019,2
		10000	110972	42300,4
		50000	645026,2	238815
		100000	1454936,4	498142,2
		500000	7933345,4	2774505,8
		1000000	16832411	5779981,2

Comparando a média dos algoritmos com o quicksort mediana temos:







6. Conclusão

Após a implementação do programa e as análises feitas, pode-se notar que a escolha do algoritmo de ordenação influencia diretamente nos fatores de memória e desempenho. Após analisar os gráficos, pode-se concluir que o quicksort é considerado o algoritmo mais eficiente por apresentar um número de cópias e um tempo de execução e, consequentemente, um custo computacional muito mais baixo que o Mergesort e o Heapsort.

Analisando as variações do quicksort, todas apresentaram dados parecidos nos quesitos de número de cópias, número de comparações e tempo de execução, além da localidade de referência. Para pequenos valores de N é praticamente imperceptível a diferença, que se torna mais clara quando N tende a ficar muito grande, no caso do tempo de execução, em que ocorrem pequenas variações de valores de tempo de execução entre os 5 algoritmos e o quicksort mediana (k=5) se tornou o mais rápido.

Também é válido analisar que no quicksort mediana, testando os valores de k=3. 5 e 7, também ocorrem valores muito parecidos, e nos quesitos de número de comparações e

número de cópias, k=3 apresenta melhor desempenho, ou seja, quanto menor o número de elementos aleatórios usados para se escolher a mediana, melhor o desempenho. No caso do quicksort seleção, tomando-se m=10 e 100, m=10 apresentou melhores resultados na análise de desempenho, ou seja, quanto menor o valor da partição, mais rápido e eficiente é a execução, o que faz sentido, já que o algoritmo seleção é muito eficiente para pequenos valores de N.

Quicksort seleção com m=10 e m=100.

81						
82	Média	1000	8099,4	3524,4	0.0073842	10
83		5000	49773,2	20075,6	0.0208566	10
84		10000	110975	42371,8	0.0487272	10
85		50000	645030,8	238882,4	0.188888	10
86		100000	1454945,4	498209,2	0.45296	10
87		500000	7933349,2	2774578	2.0245	10
88		1000000	16832414,6	5780050,6	5.57342	10
89						
90	Média	1000	8315	10025,2	0.0115416	100
91		5000	50025,6	26421,6	0.0265382	100
92		10000	111201,8	49145,4	0.0554226	100
93		50000	645273,2	245476	0.194139	100
94		100000	1455193,6	504589	0.476302	100
95		500000	7933562	2781007	2.09988	100
96		1000000	16832639,4	5786531,4	5.46908	100

Quicksort mediana com k=3, k=5 e k=7.

121						
122	Média	1000	7392,6	3516,8	0.0083386	3
123		5000	49102,6	20182,4	0.0211798	3
124		10000	109342,2	42525,8	0.0473766	3
125		50000	643544,6	239947,4	0.180146	3
126		100000	1430538,6	501862,8	0.454935	3
127		500000	7808051	2793110,8	2.10844	3
128		1000000	16855834,4	5810269,6	5.53022	3
129						
130	Média	1000	8032	3470	0.0081428	5
131		5000	49752,8	20028	0.0219062	5
132		10000	105921	42615,2	0.047961	5
133		50000	640207,8	239237	0.188209	5
134		100000	1405613,4	500376	0.463317	5
135		500000	8150855,4	2766132,8	1.97813	5
136		1000000	17127930,6	5766012,6	5.60671	5
137						
138	Média	1000	7791,6	3475,6	0.0075982	7
139		5000	48872	20066,4	0.0213174	7
140		10000	113827,4	42251	0.0463164	7
141		50000	623534,4	239836,2	0.179104	7
142		100000	1408152,4	500023,6	0.463194	7
143		500000	8003538,4	2773495,2	2.01667	7
144		1000000	17144669,6	5775578,6	5.5783	7

Comparando o quicksort mediana com os algoritmos heapsort e mergesort, é observável que para pequenos valores de N não tem muita diferença nas métricas calculadas, porém, conforme N cresce, é perceptível que o quicksort é mais eficiente que os demais, por ter um menor tempo de execução e um menor número de cópias, além de possui um número médio de comparações e o melhor custo assintótico.

Da mesma forma, é perceptível que o mergesort, embora possua menor número de comparações, possui maior número de cópias e um tempo médio de execução, enquanto o heapsort se torne o algoritmo com maior tempo de execução e número de comparações, se tornando o menos eficiente dentre os três.

7. Bibliografia

Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciencia da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição, Cengage Learning, 2011.

https://www.geeksforgeeks.org/heap-sort/

https://www.geeksforgeeks.org/merge-sort/

https://www.geeksforgeeks.org/iterative-quick-sort/

https://pt.wikipedia.org/wiki/Quicksort

8. Instruções para compilação e execução

- Acesse o diretório TP
- Por meio de um terminal. execute o comando **make all**
    - Observe que por meio deste comando os arquivos são compilados, gerando os arquivos ".obj" e, na pasta bin, o executável run.out
- A seguir os comandos a serem executados e seus respectivos parâmetros:

*Observe que o valor referente à -v, -s, -i e -o devem ser substituídos pelos parâmetros que deseja passar, sendo -v a versão, -s a semente, -i o nome do arquivo de entrada e -o o nome do arquivo de saída. O mesmo vale para os seguintes:

- **./bin/run.out quicksort -v 1 -s 10 -i entrada.txt -o saida10.txt**
- **./bin/run.out quicksort -v 2 -k 3 -s 10 -i entrada.txt -o saida10.txt**
    - -k se refere à quantidade de elementos para encontrar a mediana.
- **./bin/run.out quicksort -v 3 -m 100 -s 10 -i entrada.txt -o saida10.txt**
    - -m se refere ao tamanho das partições que serão chamadas o Seleção
- **./bin/run.out quicksort -v 4 -s 10 -i entrada.txt -o saida10.txt**
- **./bin/run.out quicksort -v 5 -s 10 -i entrada.txt -o saida10.txt**
- **./bin/run.out mergesort -s 10 -i entrada.txt -o saida10.txt**
- **./bin/run.out quicksort -s 10 -i entrada.txt -o saida10.txt**