Trabalho Prático 2
Métodos de Ordenação

Nome: Matheus Alves Kühl; Matrícula: 2015105713
Universidade Federal de Minas Gerais
Belo Horizonte-MG-Brasil

makalvesk@ufmg.br

Introdução

Este projeto foi implementado pensando em testar diferentes métodos de ordenação para um vetor com chaves escolhidas arbitrariamente a partir de uma semente gerada com o médoto srand. A implementação foi toda feita em c++. As métricas utilzadas foram: tempo total de execução do método até completar a ordenação, número de cópias do registro de memória e número de comparações entre chaves. Os métodos de comparação testados foram quicksort usual com recursão, quicksort mediana onde o pivô escolhido é definido como a mediana entre x elementos diferentes randomicamente escolhidos do intervalo a ser ordenado, quicksort seleção onde é feito o quicksort recursivo até que se chegue a subintervalos de tamanho menor que n em seguida é feito o método de ordenação por seleção, quicksort pilha que utiliza uma pilha para guardar as etapas de ordenação que antes eram chamadas por recursão e quicksort pilha inteligente que utiliza a pilha e faz ordenação a partir do menor intervalo. Na segunda parte do trabalho comparei o melhor método de quicksort com os métodos mergesort e heapsort para as mesmas sementes geradas.

Implementação

A árvore do projeto pode ser vista na figura 1, O projeto principal está dentro das pastas include e src. Na pasta include podemos observar todos os métodos e classes utilzadas, optei por fazer uma classe quicksort com as métricas e as outras quicksort com seus diferentes métodos herdando dela mas com as mesmas métricas, muitas funções do método quicksort principal (recursão) são iguais para os outros métodos portanto achei útil fazer uma classe pai chamada quicksort para os outros métodos. Os objetos abstratos que serão ordenados são do tipo Item Loaded que é um tipo Item com varias estruturas dentro dele, neste caso 15 strings com 200 caracteres cada e 10 tipo double. No método quicksort mediana cujo arquivo é quicksort_median.hpp utilizei o método de Inserção que foi definido em methods.hpp para ordenar os elementos aleatoriamente escolhidos que utlizarei para tirar a mediana. No método quicksort seleção definido no arquivo quicksort_selection.hpp, utilizei o método seleção como foi dito na introdução que também foi definido em methods.hpp. No método quicksort pilha e quicksort pilha inteligente, apesar do número médio de recursões não ultrapassar no melhor caso log (n), pode acontecer de haver no pior caso até n-1 recursões portanto determinei o tamanho da pilha como n-1.

O arquivo principal /src/main que chama os outros métodos foi estruturado de forma que recebe os parâmetros do usuário sendo eles no formato:

“método” -v “version” -s “seed” -k “k” -m “m” -i “entrada.txt” -o “saida.txt”

Onde “método” pode ser passado como “quicksort”, “mergesort” ou “heapsort” (sem as aspas), -v é uma opção válida somente para o quicksort e define qual dos métodos quicksort serão utilizados, 1: para o recursivo, 2: para o mediana, 3: para o seleção, 4: para o pilha, 5: para o pilha inteligênte. -s define o número da semente, -k é uma opção válida somente para o mediana e define o número de elementos aleatoriamente escolhidos para o cálculo da mediana, -m é uma opção válida somente para o método mediana e define o tamanho mínimo do subvetor utilizado. -i define o arquivo de entrada onde serão lidos os tamanhos dos vetores de objeto a serem ordenados. -o define o arquivo de saída onde serão impressas as métricas do método escolhido.

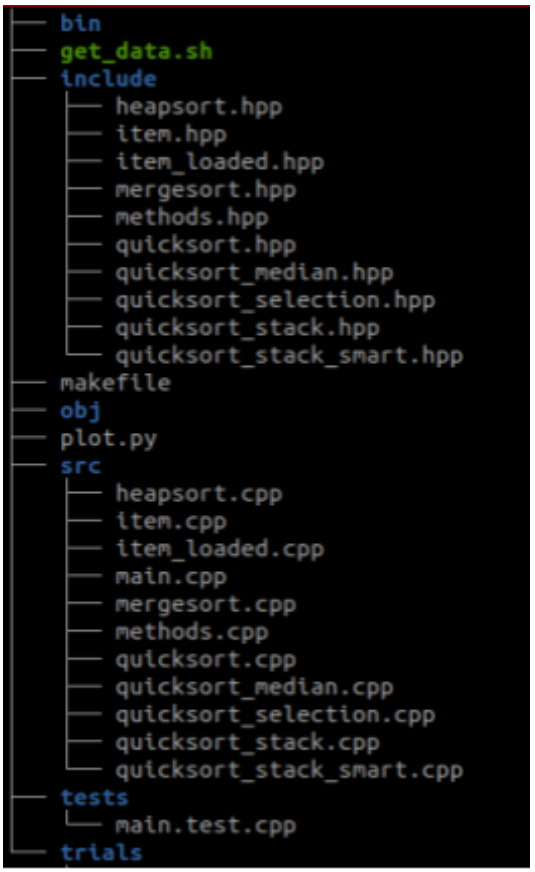


Figura 1: Árvore do Projeto

A pasta tests foi criada para realização de testes enquanto fazia o projeto, inicialmente havia mais testes mas terminei com apenas um teste main.test.cpp onde fiz os testes dos métodos para verificar se estavam ordenando os vetores.

A pasta trials, guarda alguns testes que fiz para as sementes que utilizei para analisar os métodos. O makefile foi feito de forma que os arquivos binários principais main e test fiquem na pasta /bin, e os arquivos tipo objeto na pasta /obj.

O arquivo get_data.sh é um script em bash utilizado para gerar métricas de uma quantidade arbitrária de métodos definidos no script. O arquivo plot.py foi um script em python utilizado para gerar as médias e os gráficos que serão apresentados nesta documentação. A partir das sementes testadas.

Formato da saída

O formato de sáida é da forma:

“version,k,m,size,comps,swaps,stime,utime,totaltime”

onde “version” corresponde ao método, sendo 1 para quicksort recursivo, 2 para quicksort mediana, 3 para quicksort seleção, 4 para quicksort pilha, 5 para quicksort pilha inteligênte, 6 para mergesort e 7 para heapsort. “k” e “m” é definido inicialmente como 0 e caso o método escolhido tenha “k” e “m” eles são mudados com seus valores respectivos. “comps” é o número de comparações de chaves, “swaps” o número de cópias feitas na memória, “stime” o tempo gasto pelo sistema enquanto executou o método em microsegundos,

“utime” o tempo utilzado pelo usuário no processamento do método em microsegundos e “totaltime” a soma desses intervalos de tempo.

.

Análise de Complexidade

Quicksort é um algoritmo de classificação rápida que funciona dividindo um grande array de dados em arrays menores. Isso implica que cada iteração funciona dividindo a entrada em dois componentes, classificando-os e recombinando-os. Para grandes conjuntos de dados, a técnica é altamente eficiente, pois sua complexidade temporal média e de melhor caso é O(n*log(n)), no entanto o pior caso pode acontecer quando o pivô escolhido é o maior elemento ou menor elemento do intervalo, o pior caso é da ordem $O(n^2)$. Sua complexidade de espaço também faz do método uma ótima escolha sendo da ordem O(log(n)). Uma desvantagem pode ser também o fato dele utilizar múltiplas recursões o que pode ser um problema caso recursão não seja permitida. Para remediar estas desvantagens foram inventadas variações do método quicksort com mesma complexidade de tempo e espaço. Uma desvantagem de todos os métodos quicksort é o fato deles não serem estáveis. O método quicksort mediana por exemplo pode evitar o pior caso do quicksort que acontece na escolha do pivô. Já o quicksort seleção se aproveita do fato do algoritmo de seleção ter um desempenho melhor para arrays de tamanho pequeno. Já o método quicksort pilha se utiliza de pilhas para superar o problema das recursões.

Mergesort e heapsort podem ser uma alternativa viável para o quicksort que supera os piores casos, pois a ordem de complexidade temporal desses casos é sempre da ordem de O(n*log(n)) mas a complexidade de espaço é sempre da ordem de O(n). Ambos os métodos não são estáveis.

Estratégias de Robustez

Foi implementado um teste para verificar se os algoritmos realmente ordenaram os vetores, para isto utilizei uma função simples em main.test.cpp onde verifica começando do último elemento se o elemento anterior é maior, caso seja maior o loop para e é printado um erro informando que o array não foi corretamente ordenado.

No programa principal main.cpp, inicializei a variável que guarda os argumentos com uma alocação o que define os argumentos inicialmente todos com valor 0, o que impede que o programa crash caso uma opção requerida não seja passada. Desta forma caso nenhuma opção seja passada o valor esperado para semente será 0, bem como para os arquivos de input e output. Se o arquivo de input não existir o programa vai retornar um erro.

Análise Experimental

As análises foram todas feitas com as sementes 3, 14, 15, 92 e 65. Os resultados podem ser verificados na pasta /trials. As médias foram feitas utilizando o script plot.py, bem como os gráficos (a explicação do nome das colunas podem ser encontrados na parte

Formato de Saída. O resultado das médias pode ser verificado nas imagens a seguir, o tempo foi contabilizado em segundos com precisão de micro segundo. Os gráficos tem tendência parecida em todos de O(n*log(n)) e podem ser encontrados no apêndice B. Um fato sobre a análise de memória é que como cada string tem 200 bytes se contabilizar que cada caracter tem 1 byte, e cada double tem 8 bytes foram manipulados os nos casos com array de tamanho um milhão mais de 3 gigabytes de dados.

Quicksort:

	version	k	m	size	comps	swaps	stime	utime	totaltime
0	1.0	0.0	0.0	1000.0	12476.8	8117.4	0.000390	0.040419	0.040809
1	1.0	0.0	0.0	5000.0	80704.8	48211.2	0.000567	0.198050	0.198617
2	1.0	0.0	0.0	10000.0	173408.0	103896.6	0.000040	0.391149	0.391189
3	1.0	0.0	0.0	50000.0	1050706.2	597637.8	0.000816	2.250958	2.251772
4	1.0	0.0	0.0	100000.0	2288447.4	1262196.6	0.000082	4.534986	4.535066
5	1.0	0.0	0.0	500000.0	12561216.6	7147505.4	0.000876	25.516360	25.517240
6	1.0	0.0	0.0	1000000.0	26671432.0	14990055.6	0.011327	53.413440	53.424780

Quicksort Mediana:

	version	k	m	size	comps	swaps	stime	utime	totaltime
7	2.0	3.0	0.0	1000.0	13239.0	8139.0	0.000820	0.039048	0.039868
8	2.0	3.0	0.0	5000.0	83099.2	48595.2	0.000132	0.184089	0.184221
9	2.0	3.0	0.0	10000.0	177238.6	104208.6	0.000044	0.381114	0.381157
10	2.0	3.0	0.0	50000.0	1070917.6	600326.4	0.000092	2.210324	2.210418
11	2.0	3.0	0.0	100000.0	2237399.8	1271386.8	0.000017	4.602932	4.602950
12	2.0	3.0	0.0	500000.0	12774537.8	7185460.2	0.000029	25.787440	25.787480
13	2.0	3.0	0.0	1000000.0	27244754.8	14999562.6	0.000822	53.611280	53.612100
	version	k	m	size	comps	swaps	stime	utime	totaltime
14	2.0	5.0	0.0	1000.0	13361.6	8092.8	0.000274	0.039857	0.040131
15	2.0	5.0	0.0	5000.0	81551.8	48646.8	0.000203	0.182547	0.182750
16	2.0	5.0	0.0	10000.0	178589.4	103724.4	0.000049	0.384178	0.384227
17	2.0	5.0	0.0	50000.0	1095109.2	597482.4	0.000104	2.207626	2.207730
18	2.0	5.0	0.0	100000.0	2270378.4	1271407.8	0.000054	4.665268	4.665320
19	2.0	5.0	0.0	500000.0	13020853.6	7134169.2	0.000100	25.846900	25.847000
20	2.0	5.0	0.0	1000000.0	27026069.2	15020543.4	0.000740	53.878740	53.879480
	version	k	m	size	comps	swaps	stime	utime	totaltime
21	2.0	7.0	0.0	1000.0	13573.8	8111.4	0.001001	0.042541	0.043542
22	2.0	7.0	0.0	5000.0	82985.8	48494.4	0.000157	0.181136	0.181293
23	2.0	7.0	0.0	10000.0	176779.2	104106.6	0.000108	0.383287	0.383395
24	2.0	7.0	0.0	50000.0	1065012.2	601287.6	0.000088	2.215208	2.215298
25	2.0	7.0	0.0	100000.0	2235103.4	1272348.6	0.000087	4.622608	4.622694
26	2.0	7.0	0.0	500000.0	12879252.0	7165393.8	0.000116	25.887720	25.887840
27	2.0	7.0	0.0	1000000.0	26732818.0	15039141.0	0.000113	53.909900	53.910040

Quicksort Seleção:

	version	k	m	size	comps	swaps	stime	utime	totaltime
28	3.0	0.0	10.0	1000.0	9991.2	7654.2	0.000970	0.030967	0.031937
29	3.0	0.0	10.0	5000.0	68350.6	46024.2	0.000132	0.144271	0.144404
30	3.0	0.0	10.0	10000.0	148680.0	99525.0	0.000088	0.311558	0.311646
31	3.0	0.0	10.0	50000.0	926548.4	575894.4	0.000136	1.898176	1.898314
32	3.0	0.0	10.0	100000.0	2040162.6	1218820.8	0.000128	3.871784	3.871908
33	3.0	0.0	10.0	500000.0	11320406.0	6929926.2	0.000988	22.177700	22.178720
34	3.0	0.0	10.0	1000000.0	24189011.0	14554344.6	0.000088	46.814780	46.814860
	version	k	m	size	comps	swaps	stime	utime	totaltime
35	3.0	0.0	100.0	1000.0	7635.0	5761.8	0.000479	0.025585	0.026064
36	3.0	0.0	100.0	5000.0	56041.2	36760.8	0.000248	0.111472	0.111720
37	3.0	0.0	100.0	10000.0	124181.6	80798.4	0.000081	0.244582	0.244662
38	3.0	0.0	100.0	50000.0	804530.0	482598.0	0.000056	1.517604	1.517660
39	3.0	0.0	100.0	100000.0	1795600.6	1031021.4	0.000074	3.223418	3.223494
40	3.0	0.0	100.0	500000.0	10097269.4	5993227.2	0.000828	18.754040	18.754860
41	3.0	0.0	100.0	1000000.0	21745227.4	12682397.4	0.000083	40.300500	40.300580

```
Quicksort Pilha:

    version    k    m       size       comps       swaps     stime      utime  totaltime
42      4.0  0.0  0.0     1000.0     12476.8      8117.4  0.000000   0.038021   0.038021
43      4.0  0.0  0.0     5000.0     80704.8     48211.2  0.000070   0.176590   0.176661
44      4.0  0.0  0.0    10000.0    173408.0    103896.6  0.000072   0.378835   0.378907
45      4.0  0.0  0.0    50000.0   1050706.2    597637.8  0.000036   2.185516   2.185552
46      4.0  0.0  0.0   100000.0   2288447.4   1262196.6  0.000020   4.533796   4.533818
47      4.0  0.0  0.0   500000.0  12561216.6   7147505.4  0.000002  25.493060  25.493060
48      4.0  0.0  0.0  1000000.0  26671432.0  14990055.6  0.000853  53.896640  53.897500
```

```
Quicksort Pilha Inteligente:

    version    k    m       size       comps       swaps     stime      utime  totaltime
49      5.0  0.0  0.0     1000.0     12476.8      8117.4  0.000898   0.043181   0.044080
50      5.0  0.0  0.0     5000.0     80704.8     48211.2  0.000121   0.202653   0.202774
51      5.0  0.0  0.0    10000.0    173408.0    103896.6  0.000047   0.378583   0.378630
52      5.0  0.0  0.0    50000.0   1050706.2    597637.8  0.000111   2.192114   2.192228
53      5.0  0.0  0.0   100000.0   2288447.4   1262196.6  0.000019   4.539186   4.539206
54      5.0  0.0  0.0   500000.0  12561216.6   7147505.4  0.000804  25.547880  25.548700
55      5.0  0.0  0.0  1000000.0  26671432.0  14990055.6  0.000038  53.736780  53.736820
```

```
Mergesort:

    version    k    m       size       comps       swaps      stime       utime   totaltime
63      6.0  0.0  0.0     1000.0      8695.2      19952.0   0.009099    0.103186    0.112285
64      6.0  0.0  0.0     5000.0     55238.8     123616.0   0.013182    0.438615    0.451796
65      6.0  0.0  0.0    10000.0    120409.6     267232.0   0.019328    0.984998    1.004327
66      6.0  0.0  0.0    50000.0    718181.0    1568928.0   0.143281    5.655902    5.799184
67      6.0  0.0  0.0   100000.0   1536377.6    3337856.0   0.404727   11.811500   12.216220
68      6.0  0.0  0.0   500000.0   8837344.4   18951424.0   3.481826   68.209920   71.691740
69      6.0  0.0  0.0  1000000.0  18674322.2   39902848.0  10.791500  141.494200  152.285800
```

```
Heapsort:

    version    k    m       size       comps       swaps     stime      utime  totaltime
56      7.0  0.0  0.0     1000.0      5434.2      15073.4  0.000488   0.080270   0.080758
57      7.0  0.0  0.0     5000.0     32705.2      87157.0  0.000367   0.482482   0.482849
58      7.0  0.0  0.0    10000.0     70531.6     184193.0  0.000775   1.032454   1.033227
59      7.0  0.0  0.0    50000.0    410724.2    1037378.2  0.002101   3.902672   3.904776
60      7.0  0.0  0.0   100000.0    870936.4    2174867.4  0.000350   7.380128   7.380474
61      7.0  0.0  0.0   500000.0   4950826.4   12024033.6  0.003027  41.132040  41.135040
62      7.0  0.0  0.0  1000000.0  10400792.0   25048457.4  0.001888  86.627020  86.628900
```

De acordo com as médias retiradas a partir das sementes escolhidas, podemos observar que o método quicksort seleção com m =100 tem o melhor desempenho de todos em todos os quesitos, inclusive em comparação com o método heapsort e mergesort.

Conclusão

A partir da análise experimental feita posso atestar que o método quicksort seleção com m = 100 é o que teve melhor desempenho, apesar de que para ser feito uma análise boa o suficiente seria necessário calcular o desvio da média de cada uma das sementes testadas para verificar também a consistência dos resultados ao se variar as sementes. Dito isto o método parece ser o que melhor desempenha a ordenação para um valor arbitrário de semente. Podemos verificar também analisando os métodos que quanto maior o número de

comparações e cópias na memória a complexidade de tempo cresce conforme esses valores.

Bibliografia

CORMEN, Thomas. Introduction to Algorithms. 4th Edition, MIT Press, ISBN-10:9780262033848, 223-339, 2022.

Apêndice A

Como Executar este programa?

O programa principal pode ser executado com o comando de make ou make main a partir do diretório principal onde o makefile está instalado. É necessário ter o compilador de c++ 11 standard instalado (g++), o programa foi testado somente em ambiente linux Ubuntu 21.

Como Executar os testes?

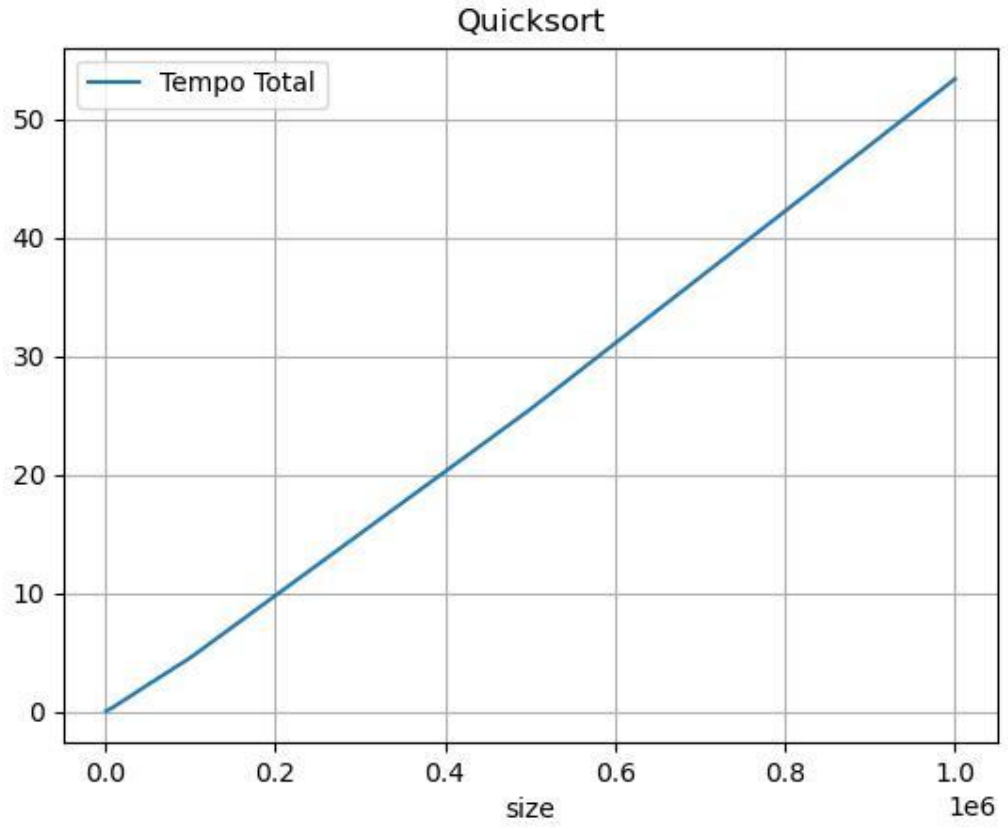
Pode ser executado com o comando de make test a partir do diretório principal onde o makefile está instalado.

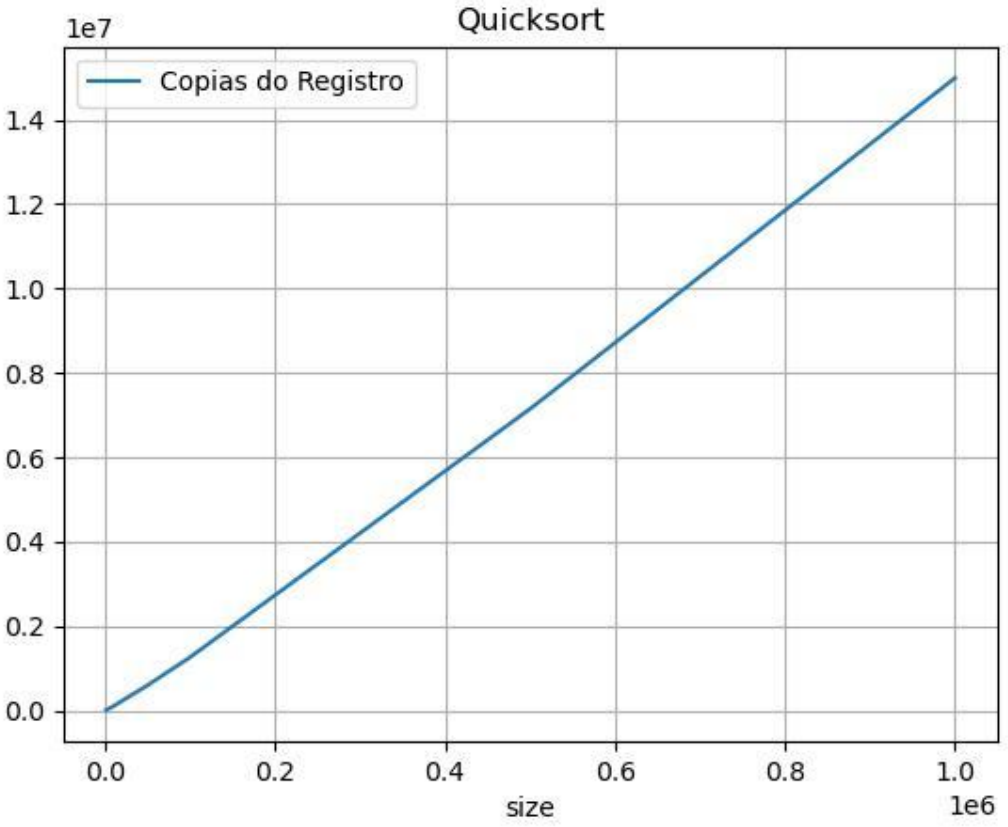
Como testar várias sementes ao mesmo tempo e vários métodos?

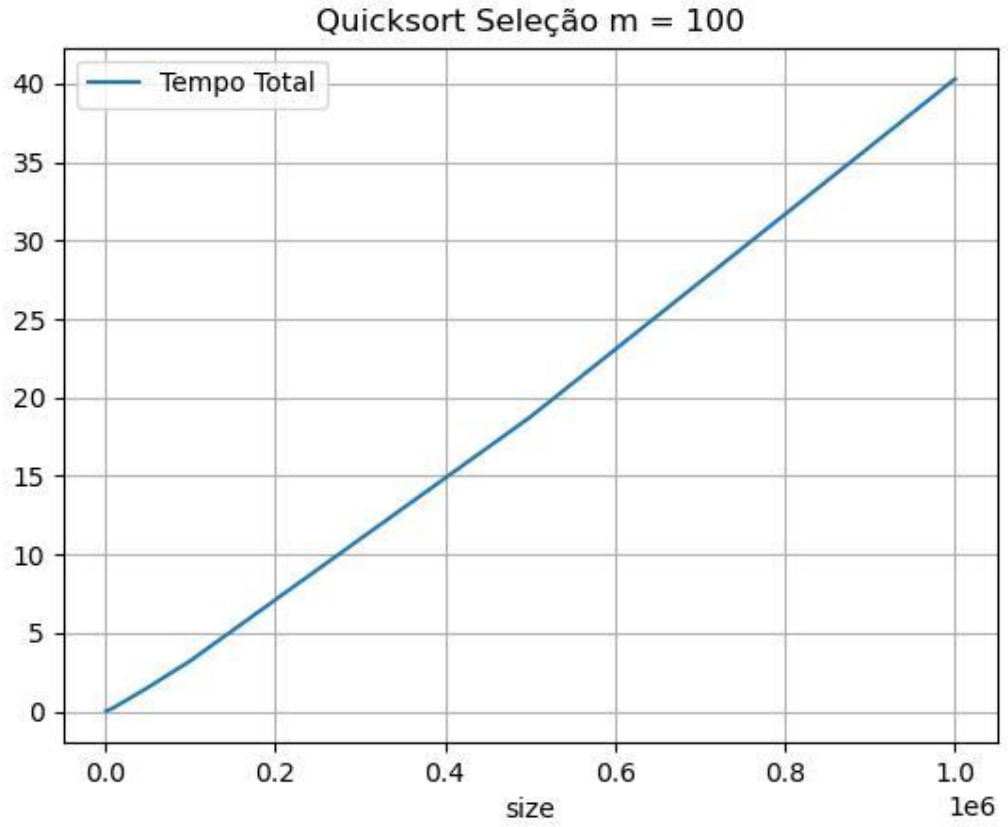
Modifique get_data.sh de acordo com seu gosto. Instruções são dadas dentro do arquivo.

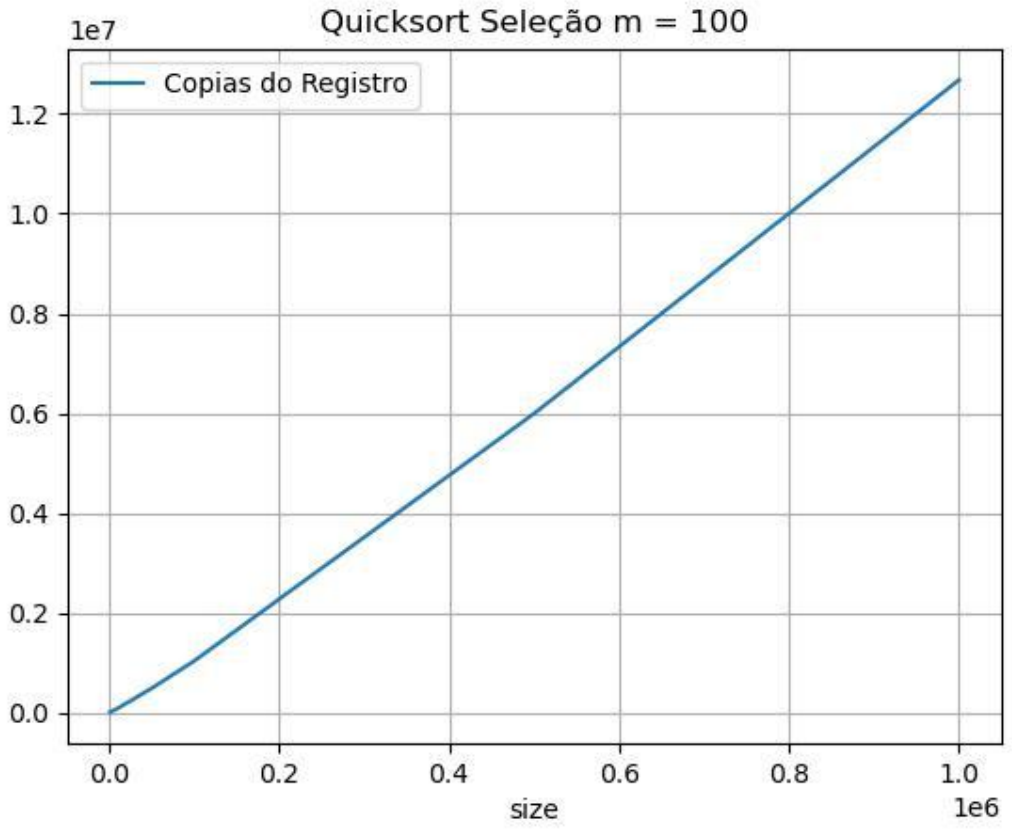
Apêndice B

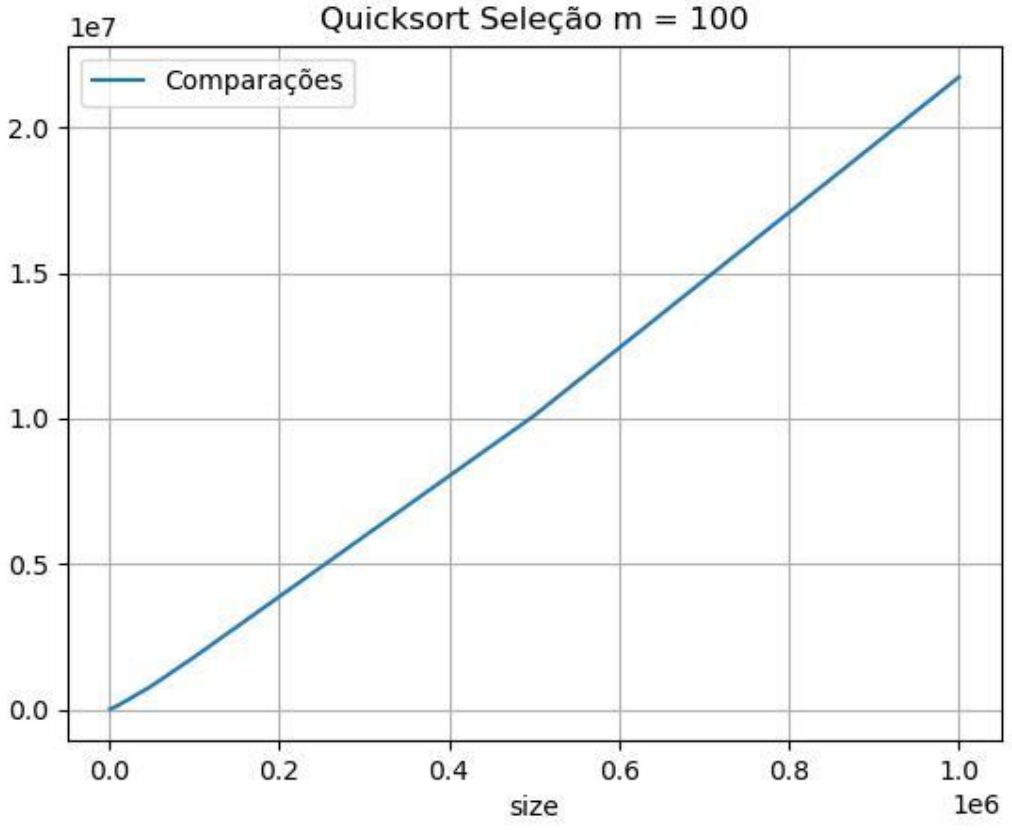
Gráficos dos métodos testados:

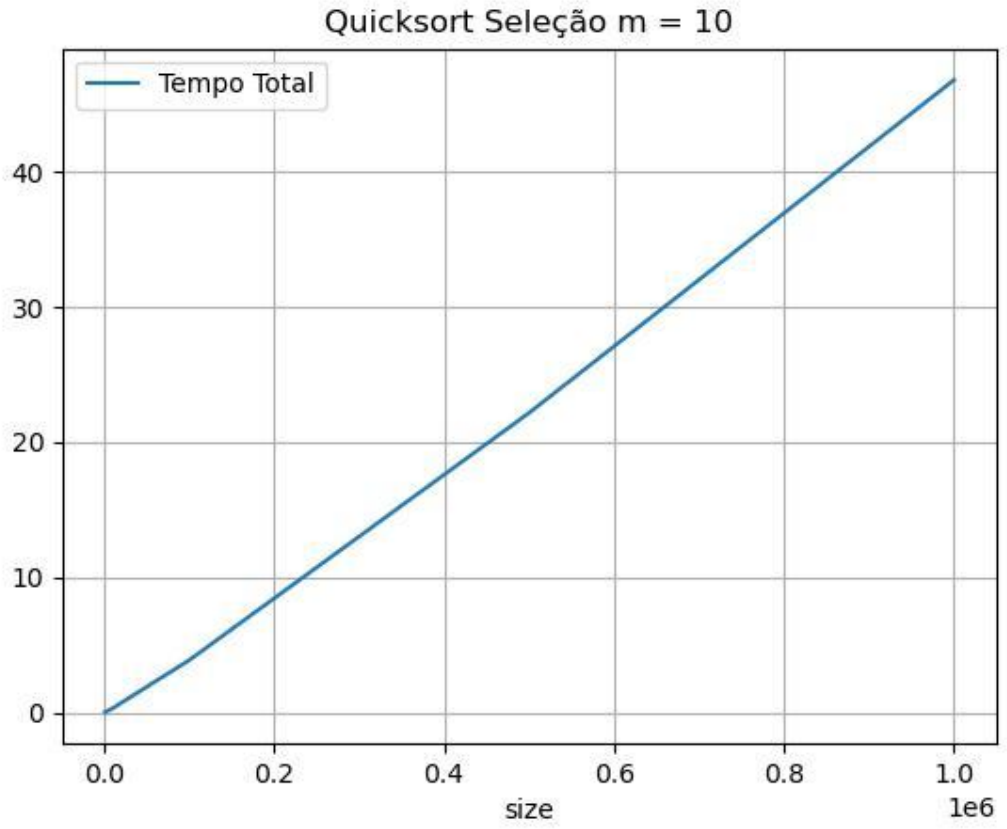


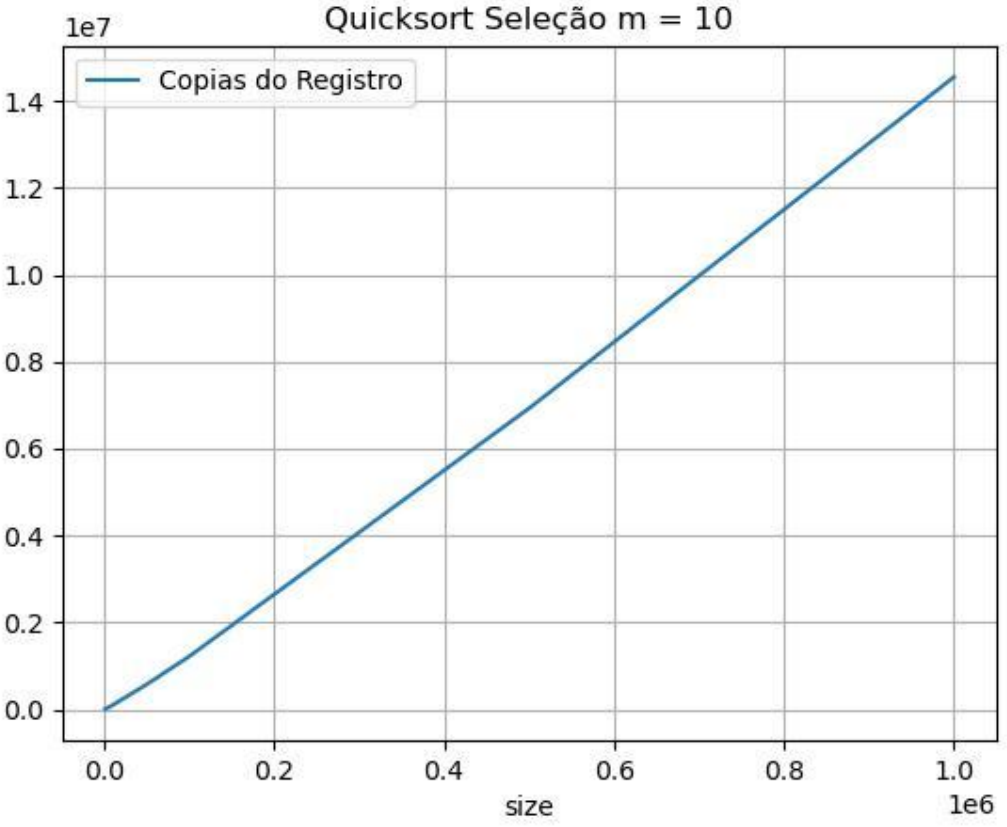

Quicksort
1e7
Copias do Registro
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1e6
size

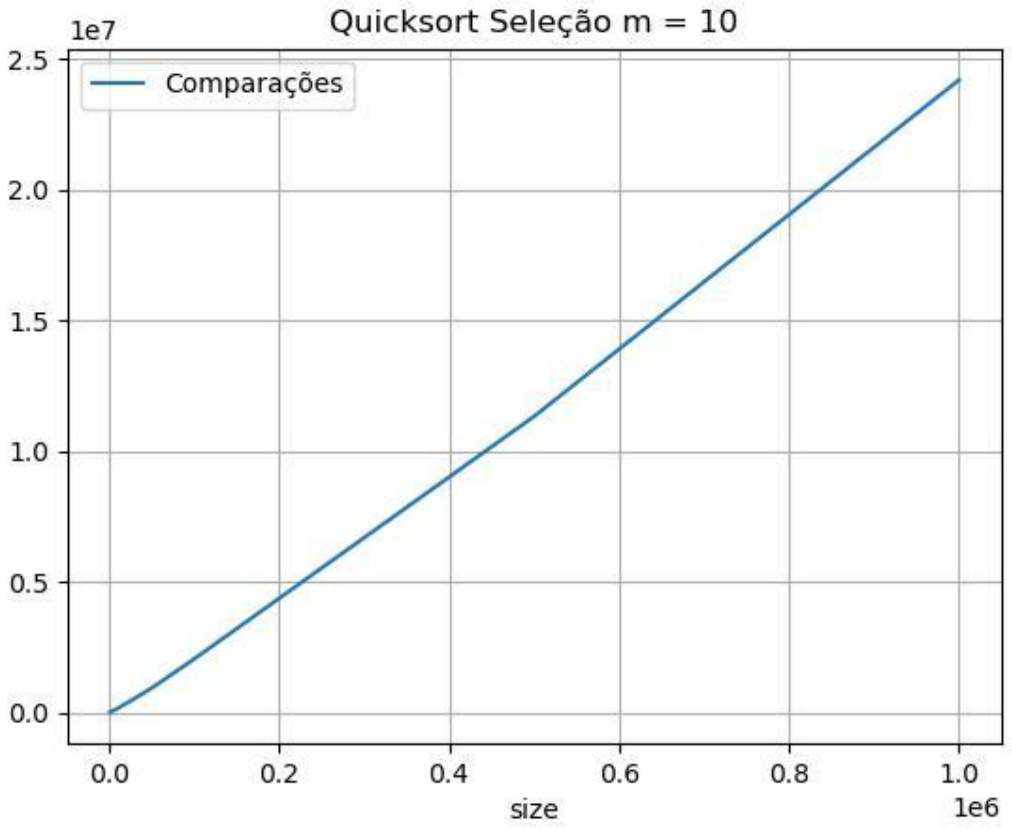

Quicksort Seleção m = 100
Tempo Total
40
35
30
25
20
15
10
5
0
0.0
0.2
0.4
0.6
0.8
1.0
1e6
size

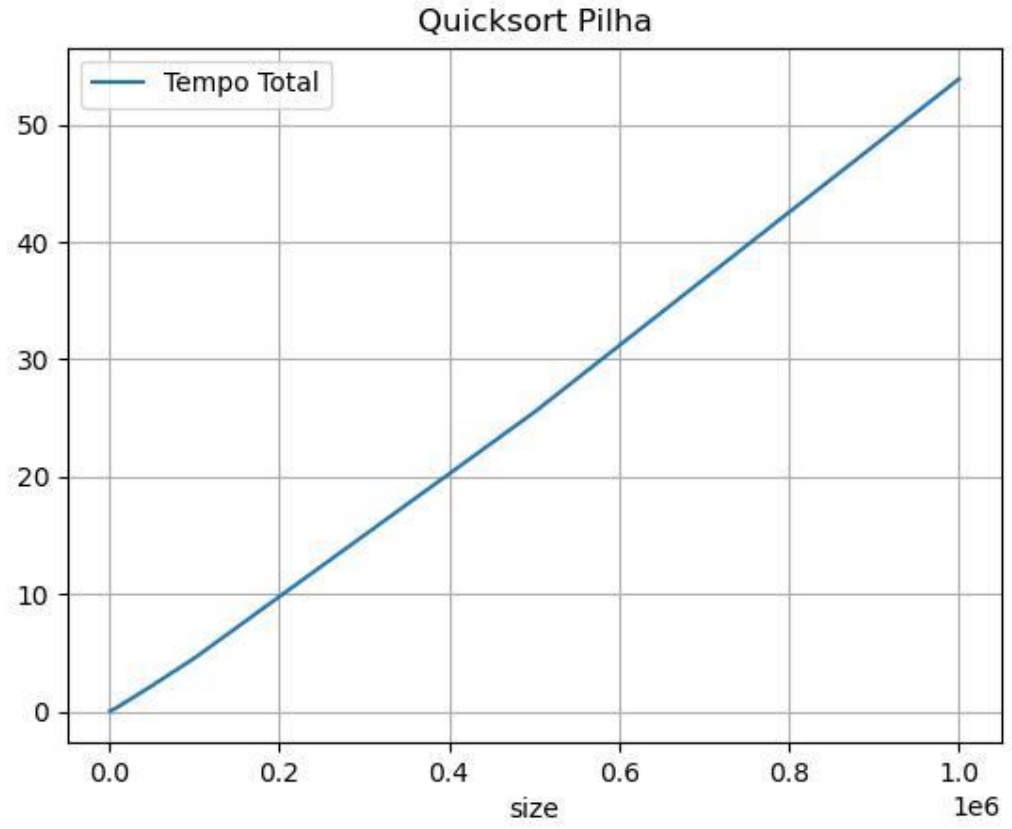

Quicksort Seleção m = 100
1e7
Copias do Registro
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
size
1e6

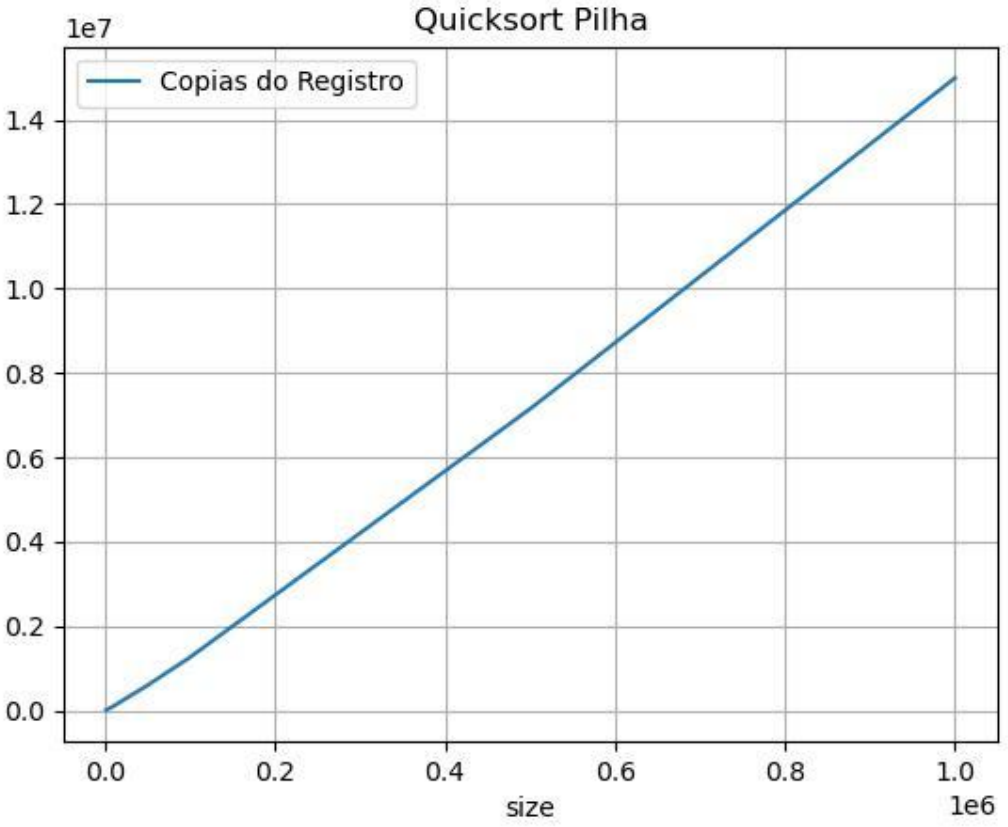

Quicksort Seleção m = 100
1e7
Comparações
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0
size
1e6

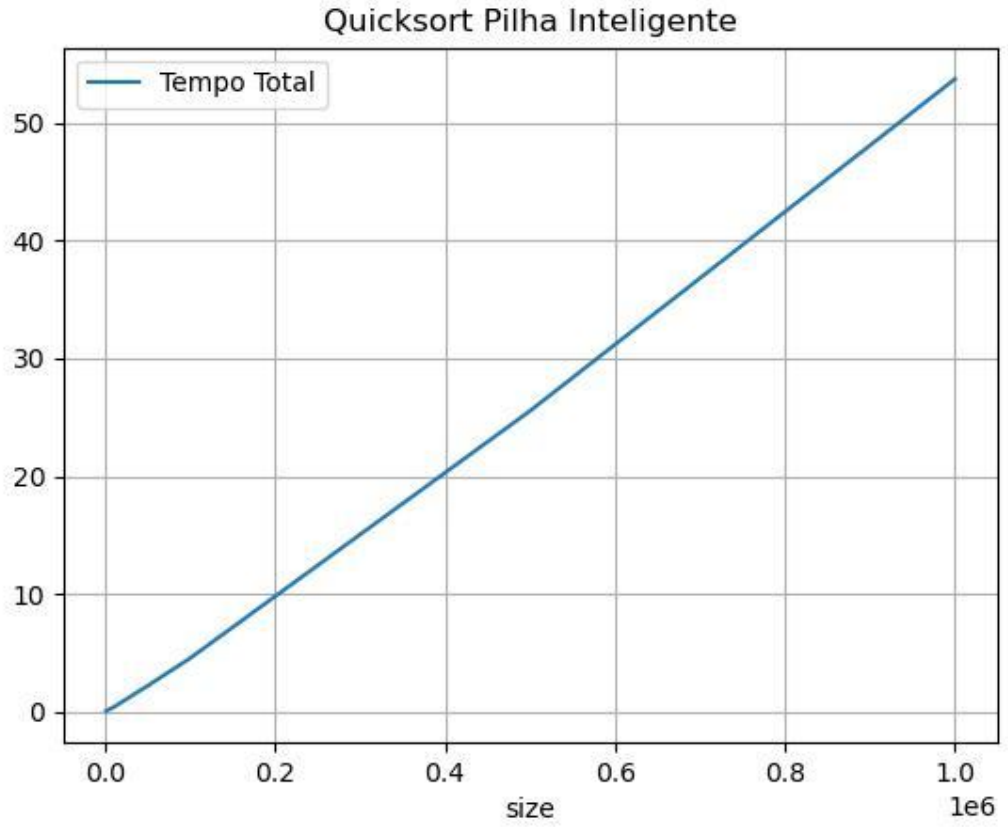

Quicksort Seleção m = 10
Tempo Total
40
30
20
10
0
0.0
0.2
0.4
0.6
0.8
1.0
1e6
size

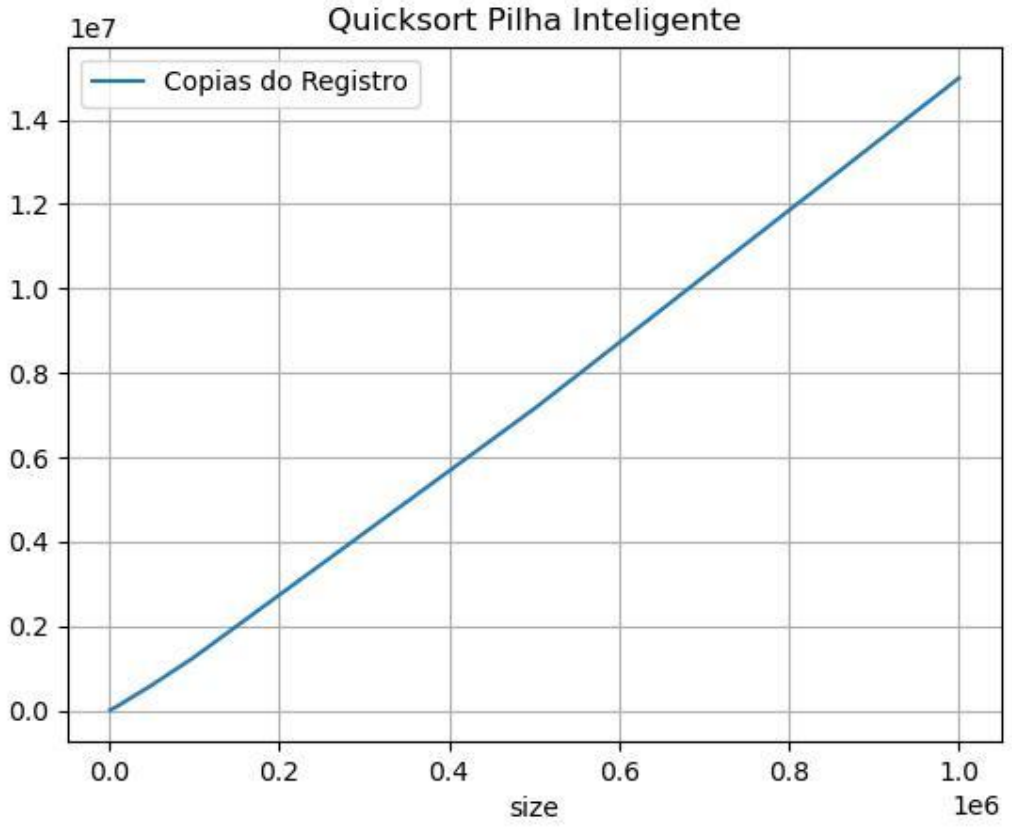

Quicksort Seleção m = 10
1e7
Copias do Registro
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1e6
size

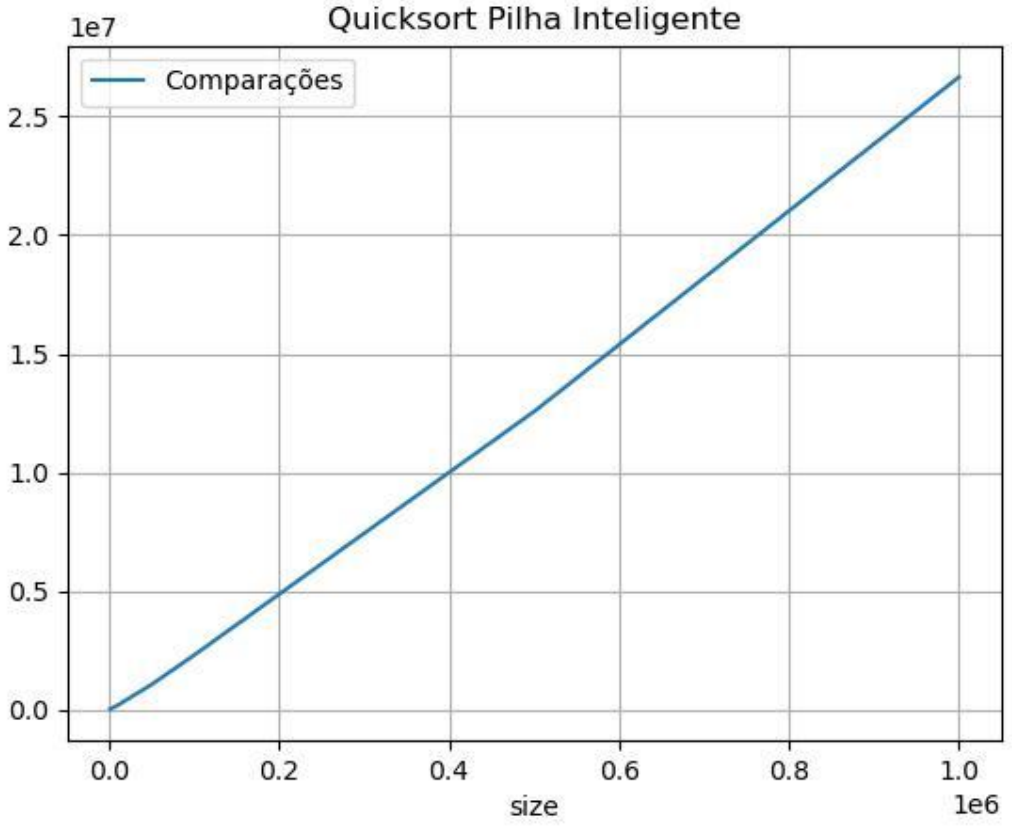

Quicksort Seleção m = 10
1e7
Comparações
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0
size
1e6

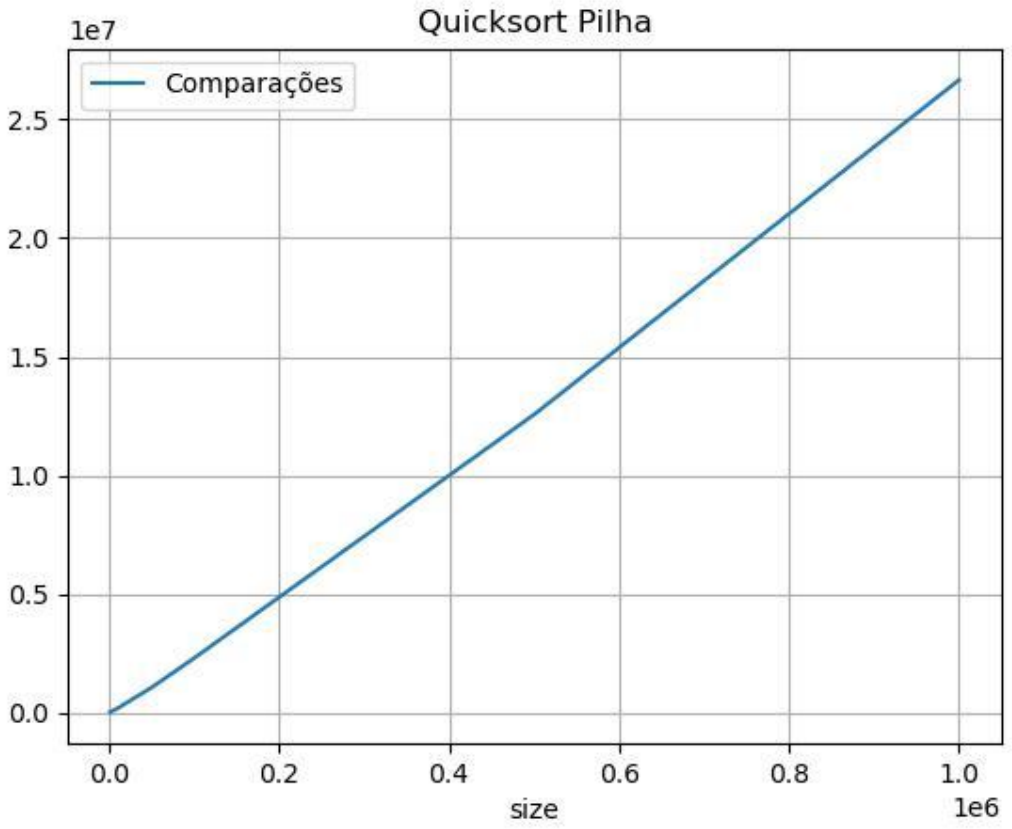

Quicksort Pilha
Tempo Total
50
40
30
20
10
0
0.0
0.2
0.4
0.6
0.8
1.0
size
1e6

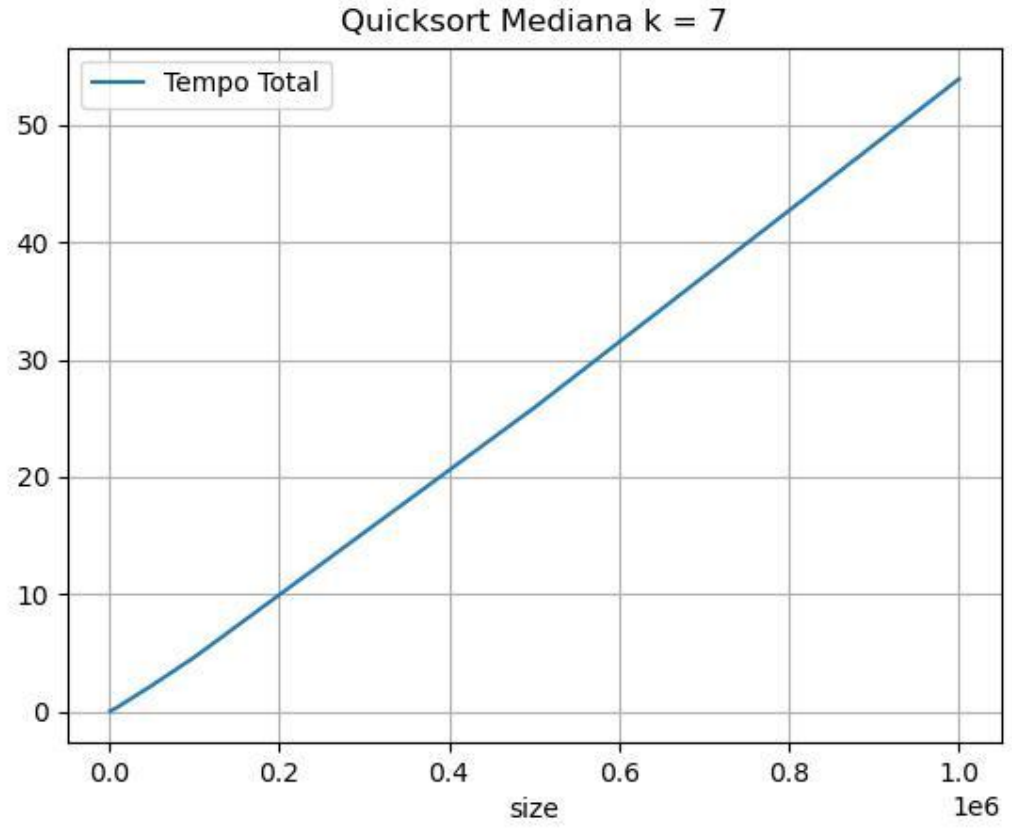

Quicksort Pilha
1e7
Copias do Registro
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
size
1e6

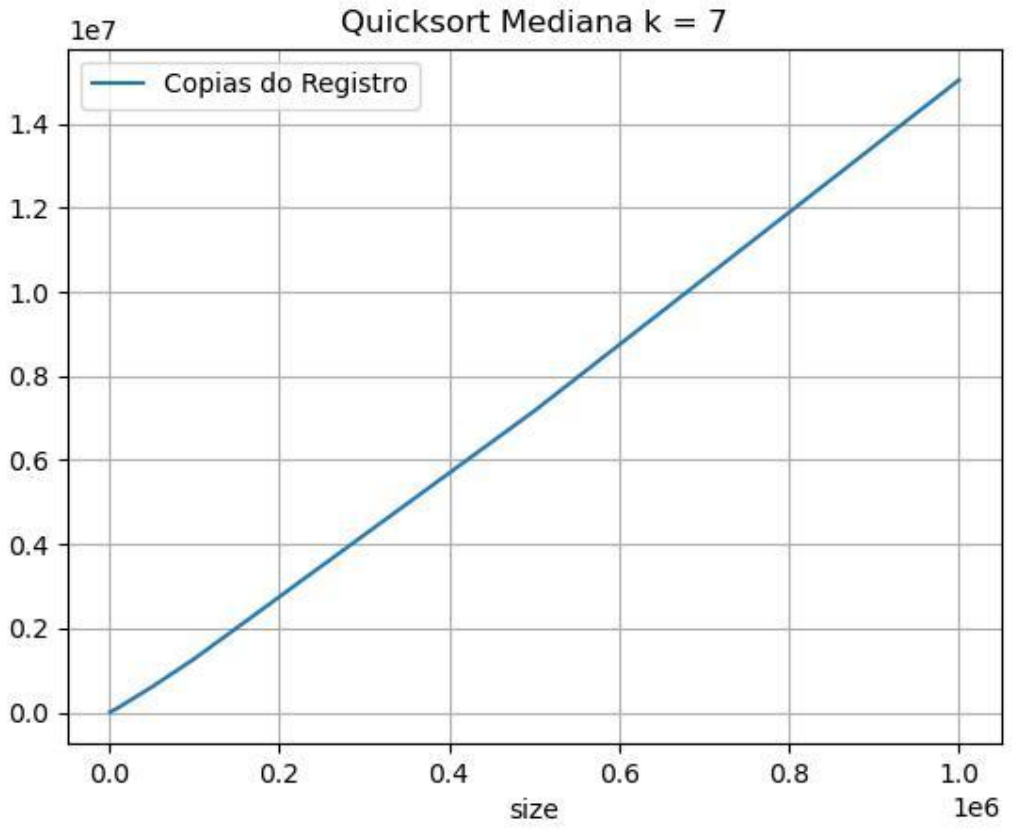

Quicksort Pilha Inteligente
Tempo Total
50
40
30
20
10
0
0.0
0.2
0.4
0.6
0.8
1.0
size
1e6

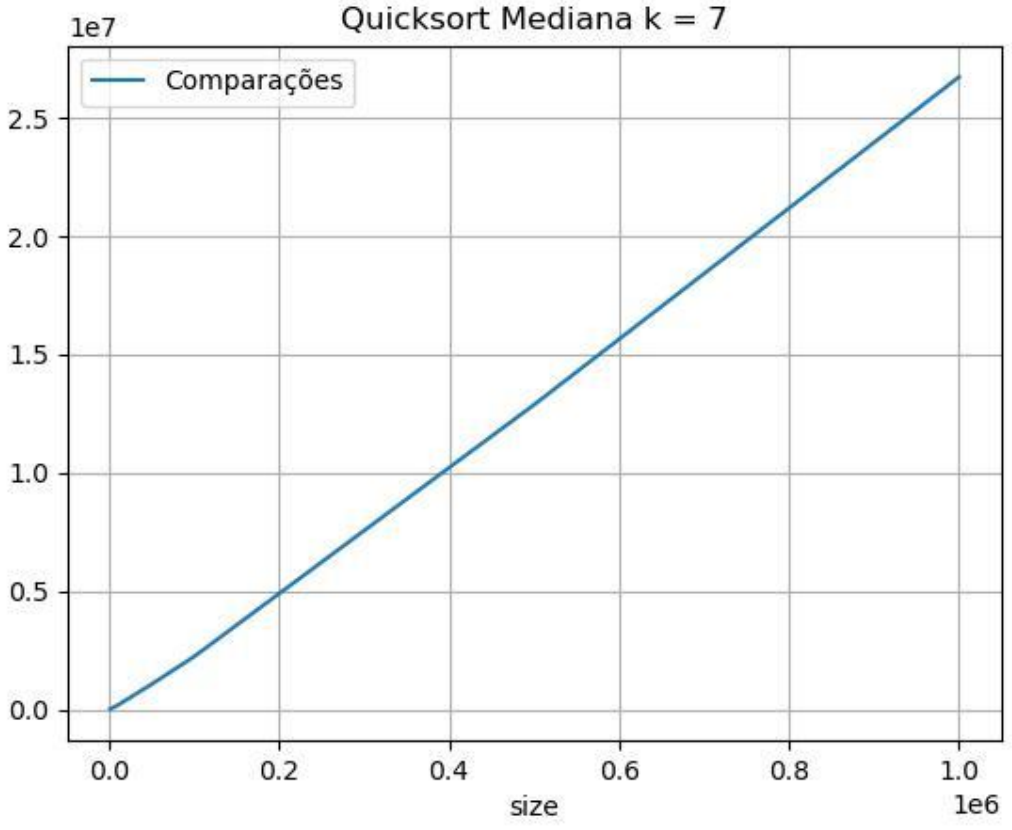

Quicksort Pilha Inteligente
1e7
Copias do Registro
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1e6
size

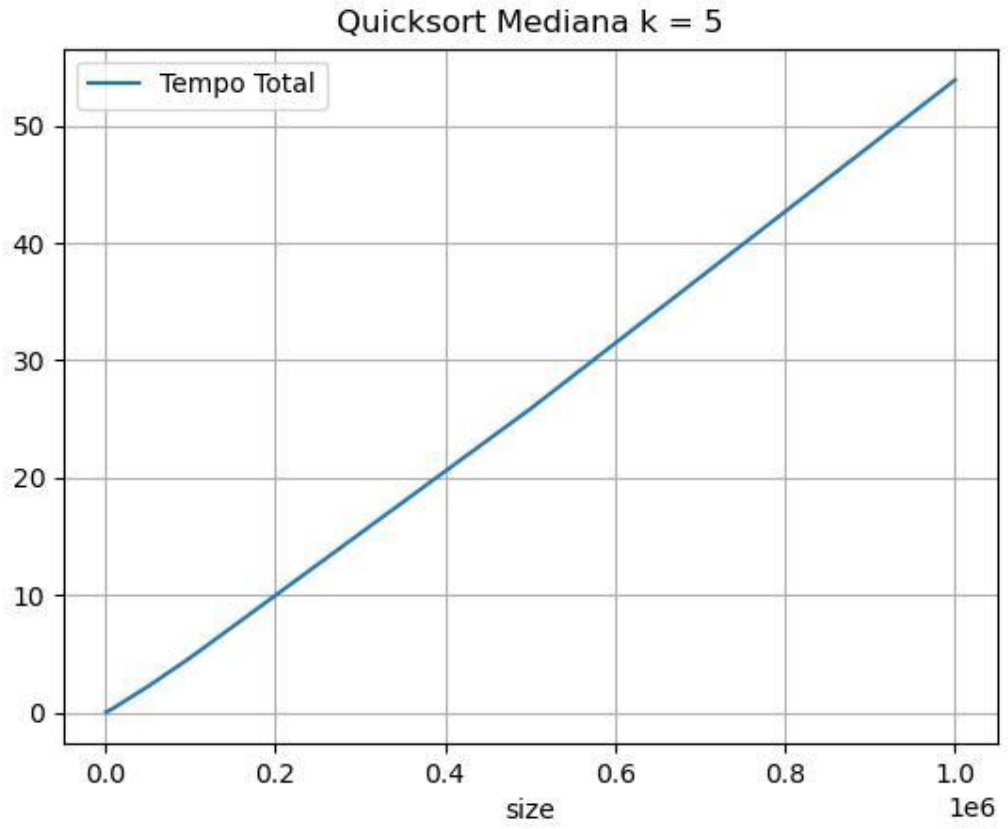

Quicksort Pilha Inteligente
1e7
Comparações
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1e6
size

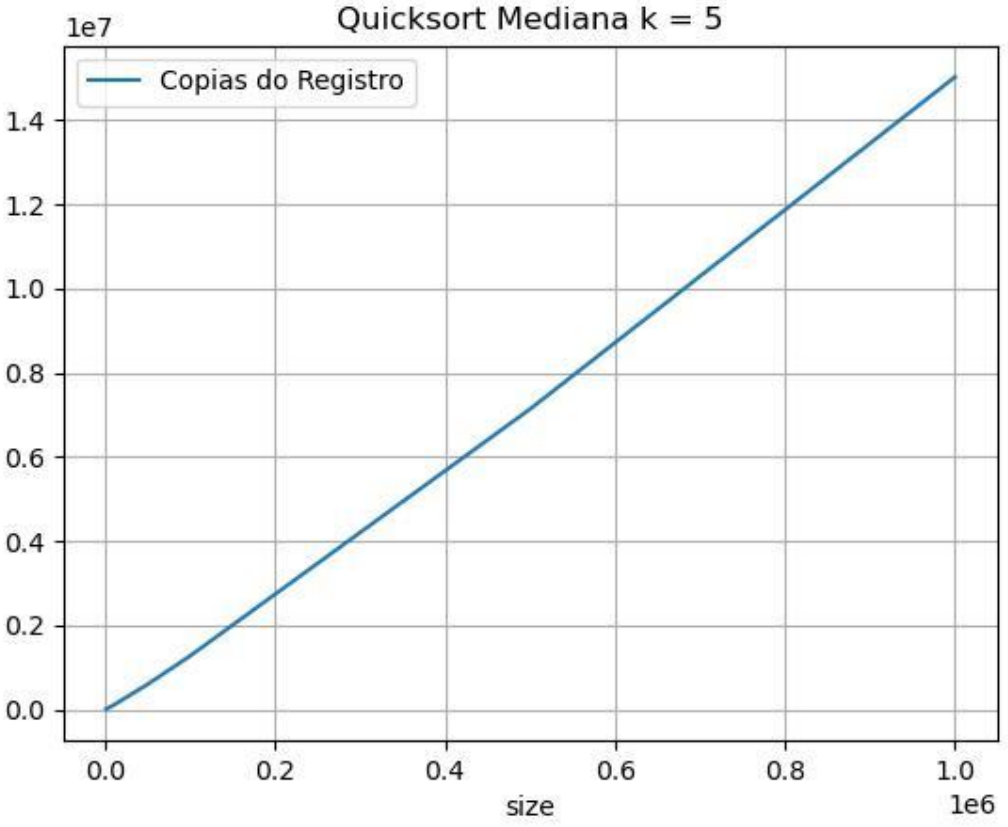

Quicksort Pilha
1e7
Comparações
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0
size
1e6

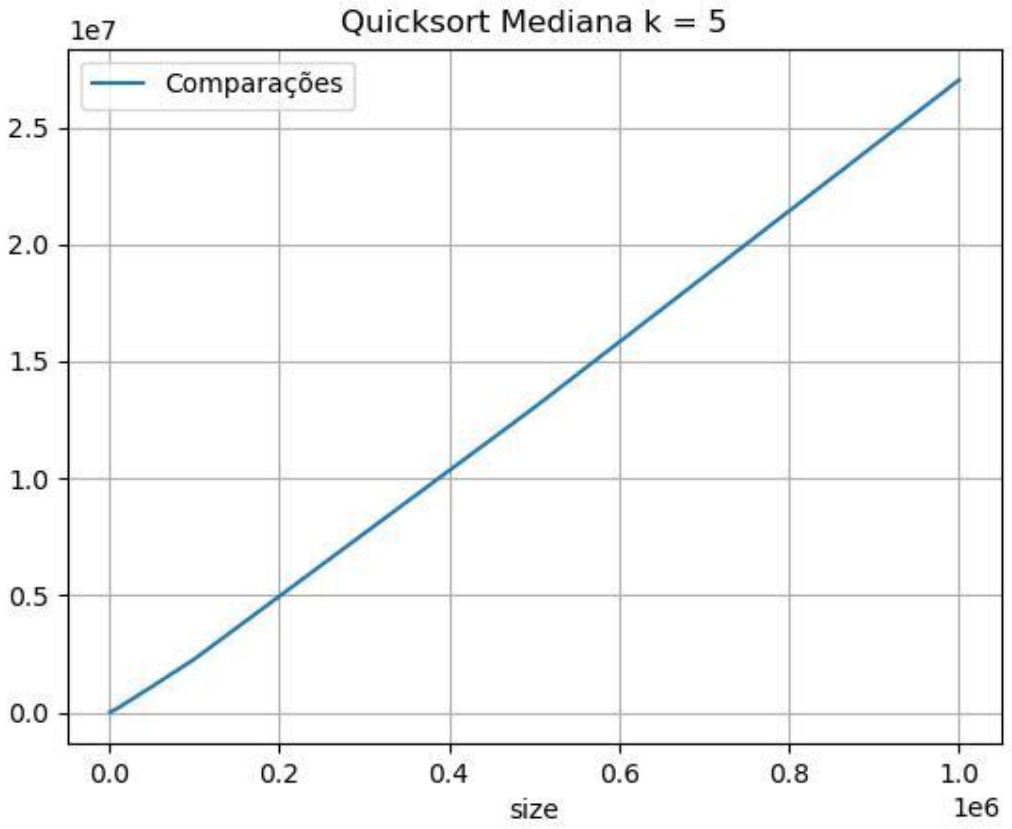

Quicksort Mediana k = 7
Tempo Total
50
40
30
20
10
0
0.0
0.2
0.4
0.6
0.8
1.0
size
1e6

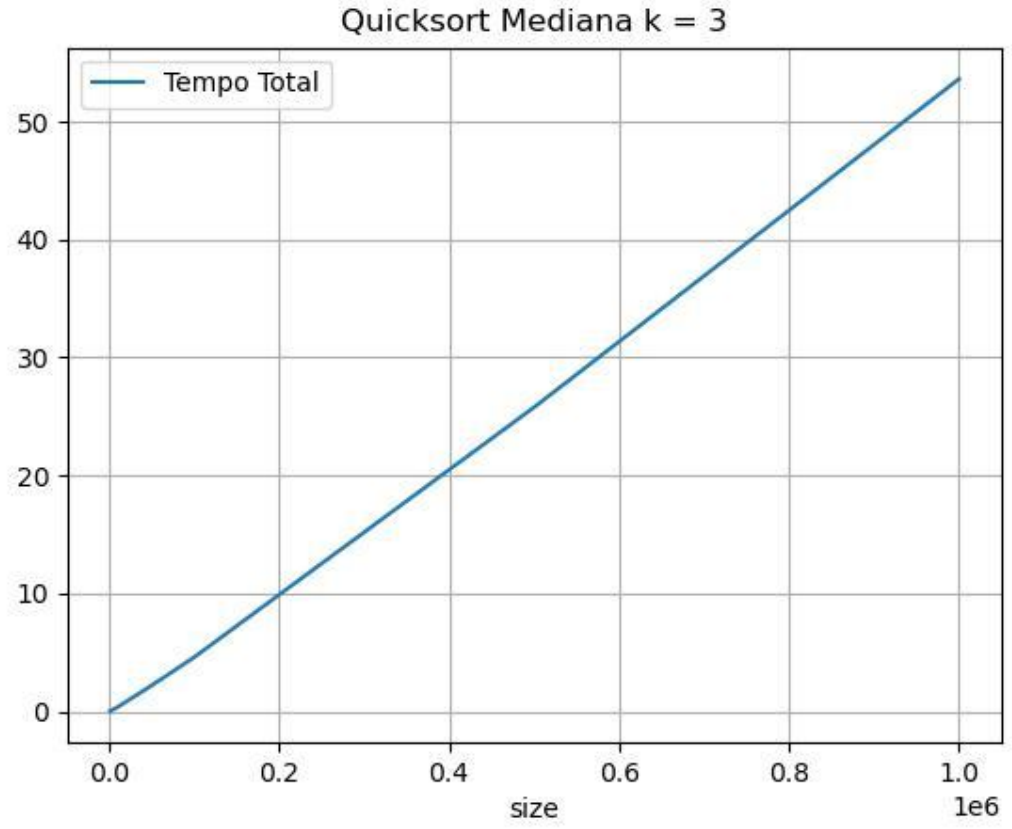

Quicksort Mediana k = 7
1e7
Copias do Registro
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1e6
size

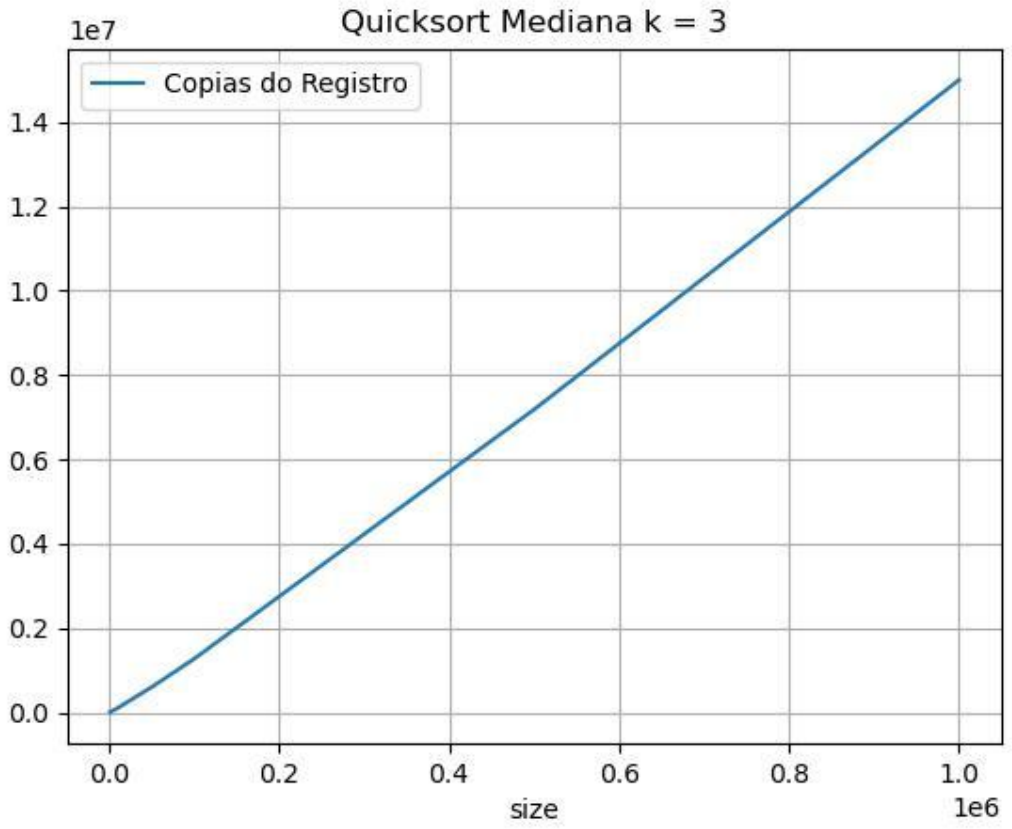

Quicksort Mediana k = 7
1e7
Comparações
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1e6
size

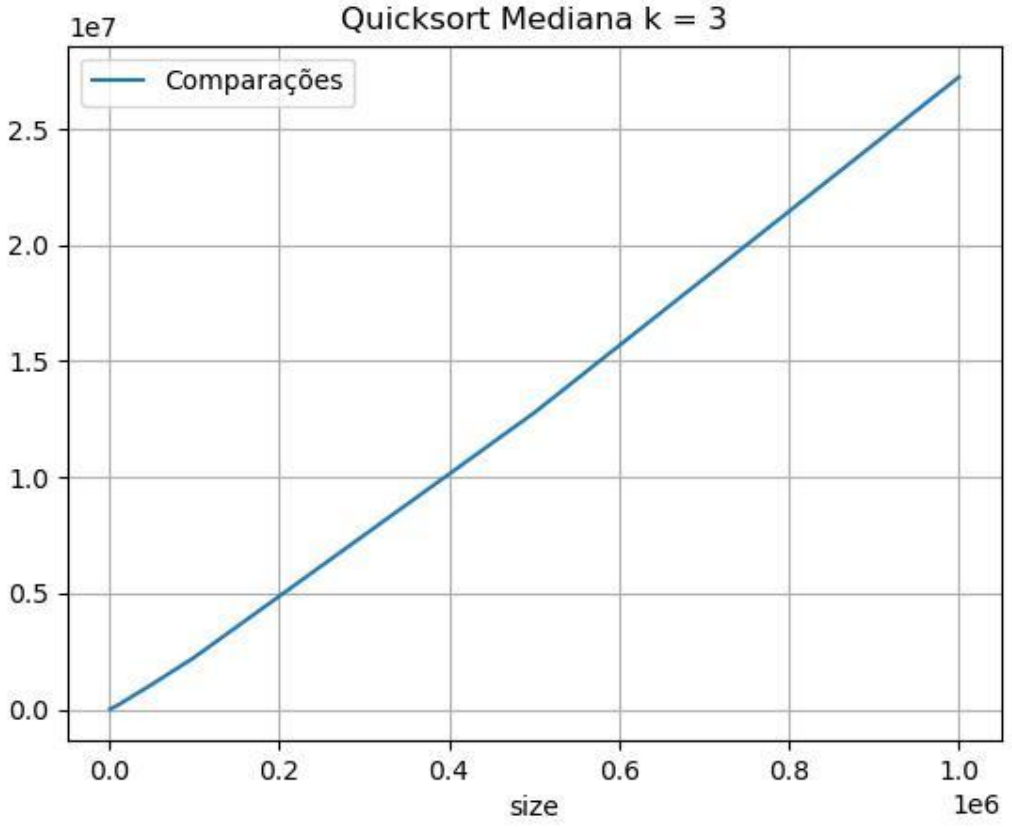

Quicksort Mediana k = 5
Tempo Total
50
40
30
20
10
0
0.0
0.2
0.4
0.6
0.8
1.0
1e6
size

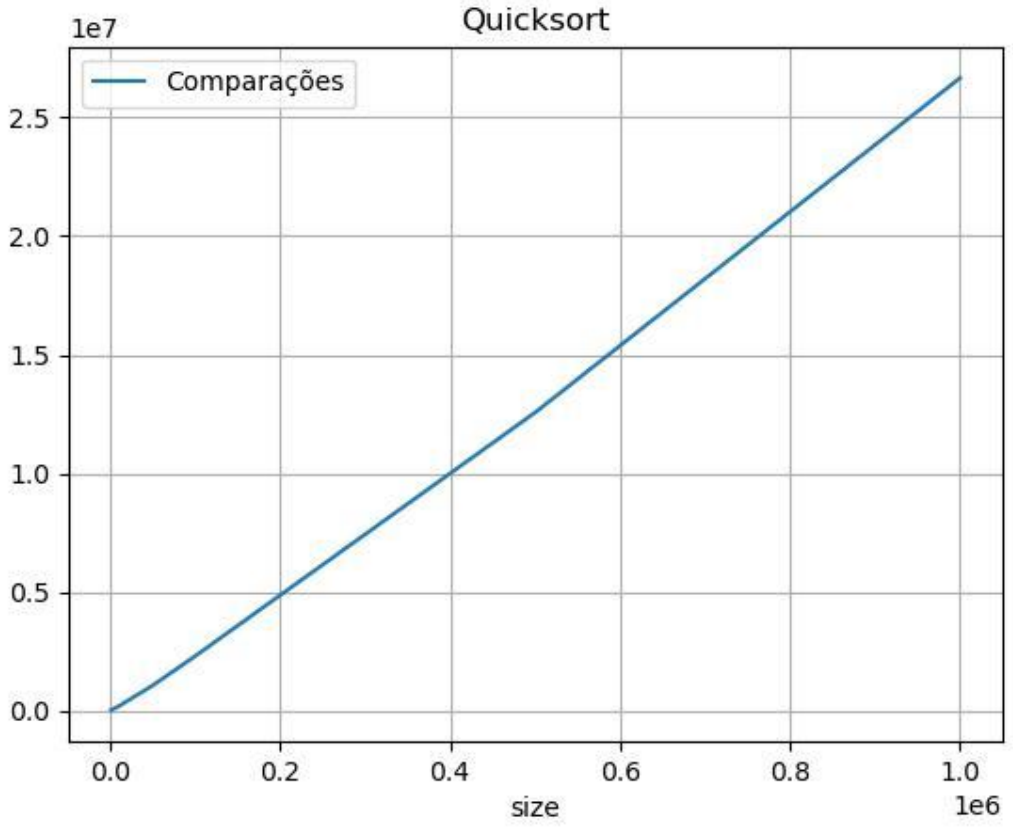

Quicksort Mediana k = 5
1e7
Copias do Registro
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1e6
size

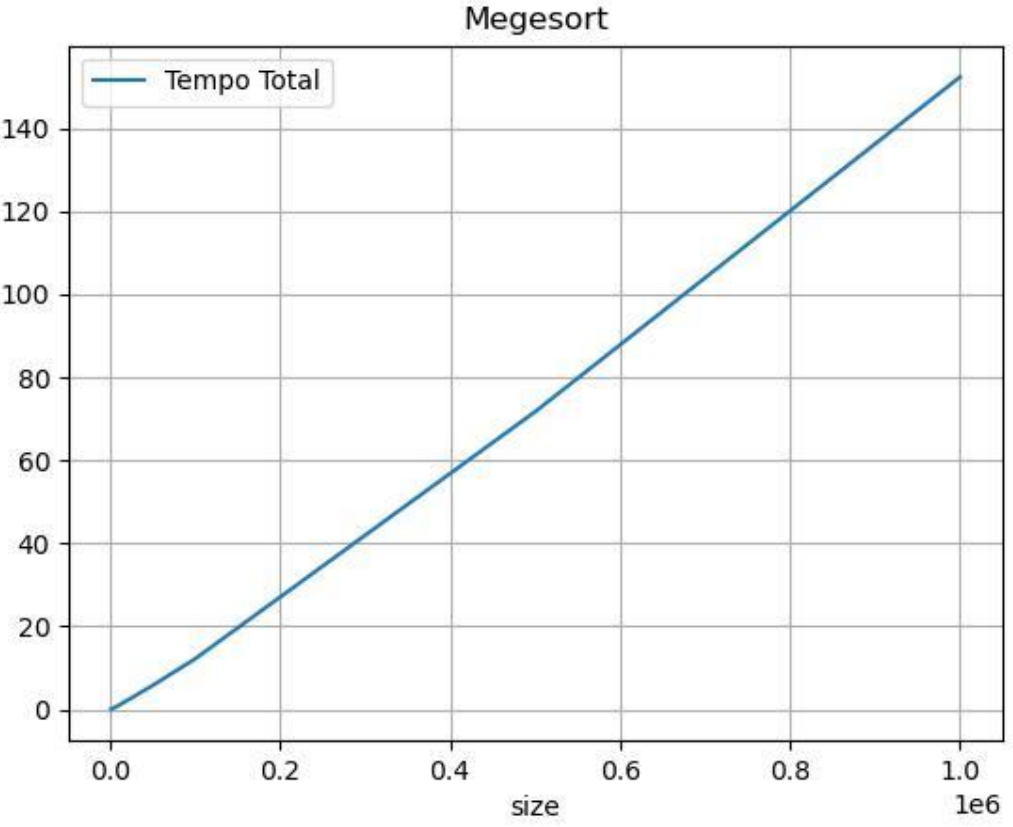

Quicksort Mediana k = 5
1e7
Comparações
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1e6
size

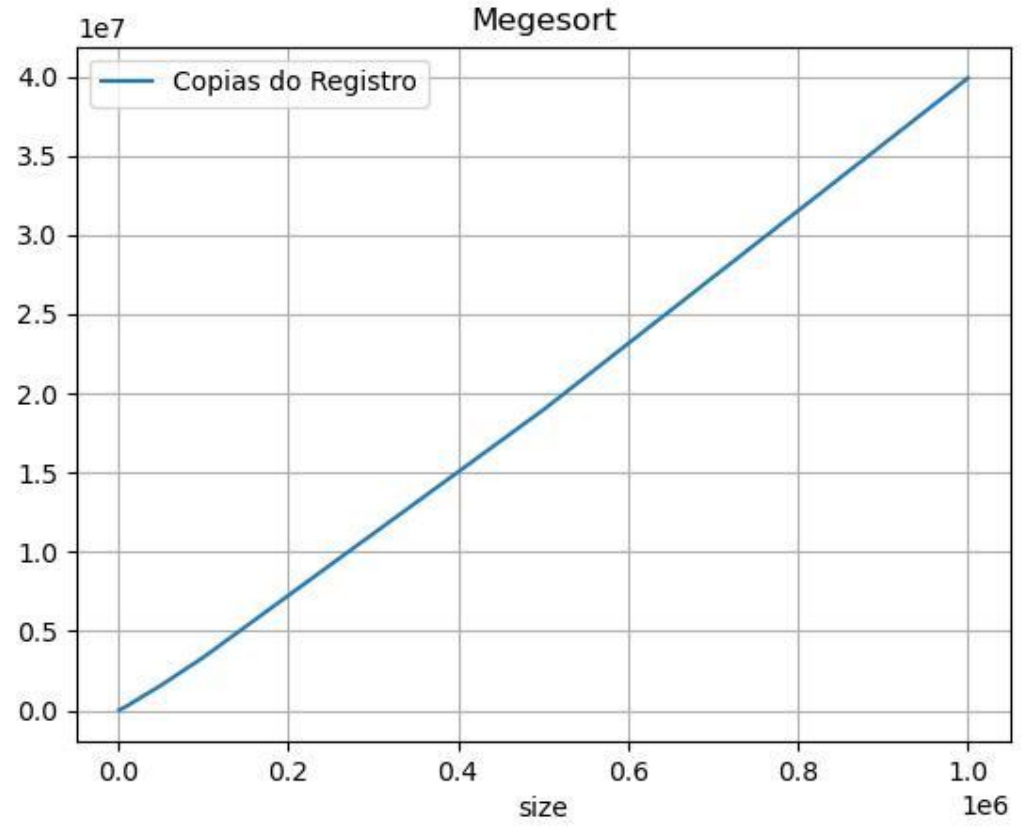

Quicksort Mediana k = 3
Tempo Total
50
40
30
20
10
0
0.0
0.2
0.4
0.6
0.8
1.0
1e6
size

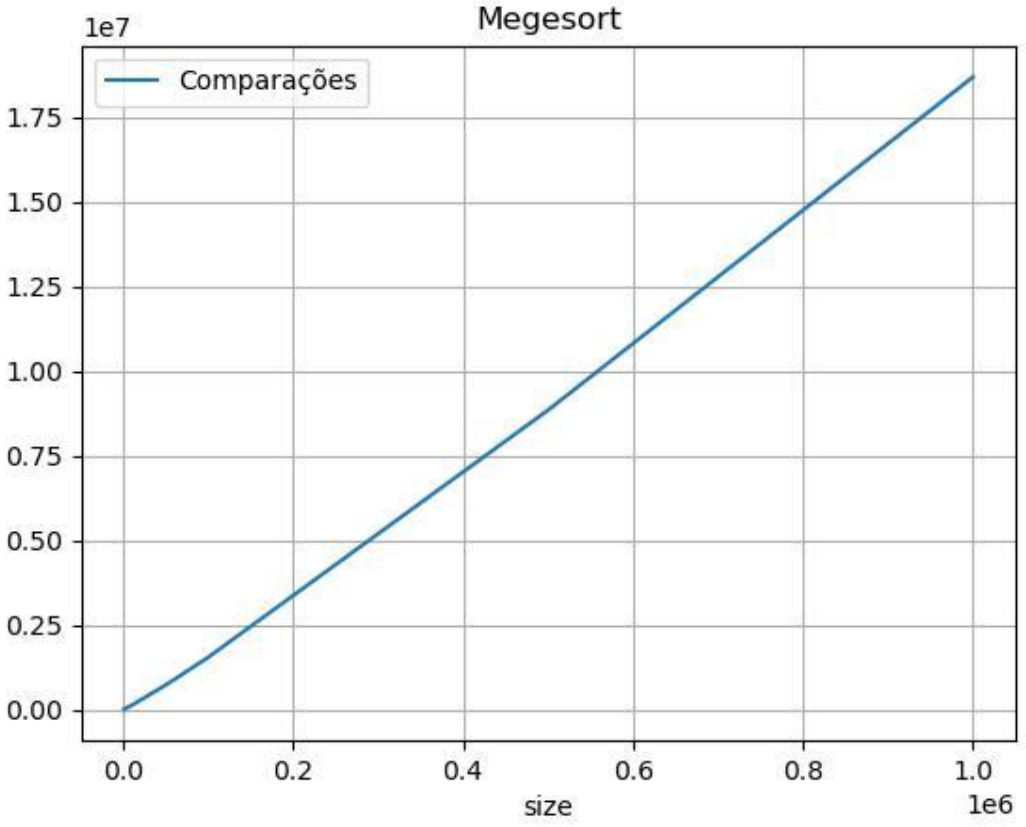

Quicksort Mediana k = 3
1e7
Copias do Registro
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
size
1e6

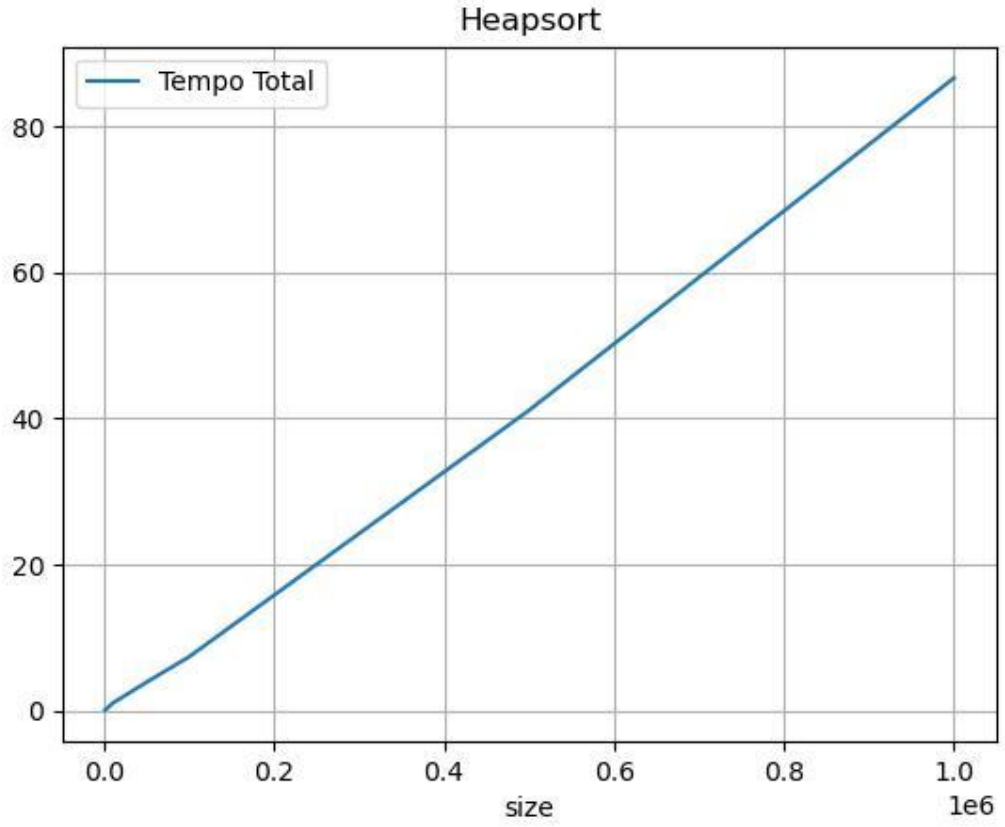

Quicksort Mediana k = 3
1e7
Comparações
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0
size
1e6

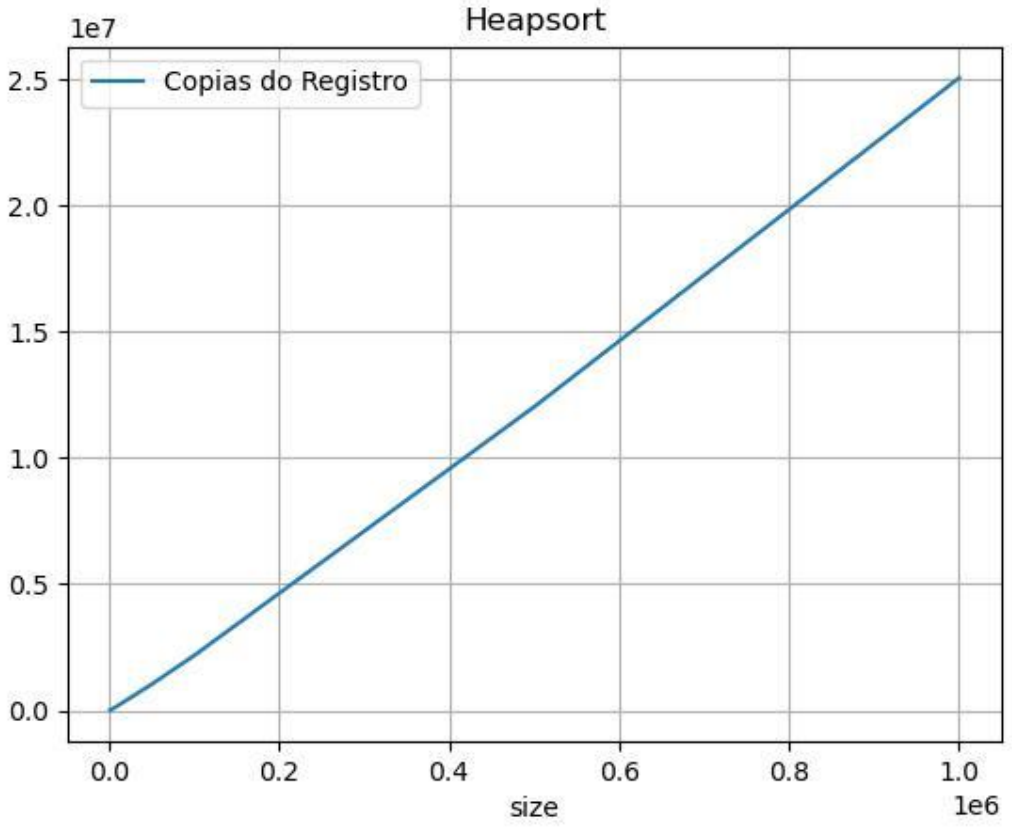

Quicksort
1e7
Comparações
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1e6
size

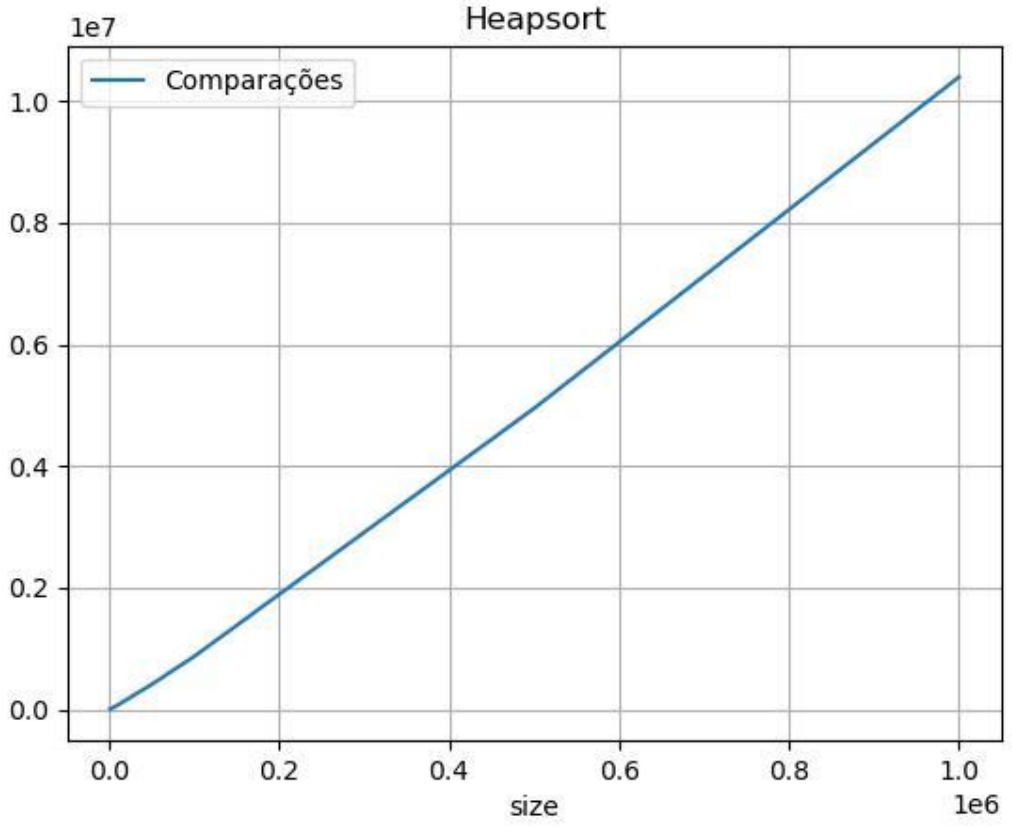

Megesort
Tempo Total
140
120
100
80
60
40
20
0
0.0
0.2
0.4
0.6
0.8
1.0
1e6
size


Megesort
1e7
Copias do Registro
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0
size
1e6


Megesort
1e7
Comparações
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.0
0.2
0.4
0.6
0.8
1.0
size
1e6


Heapsort
Tempo Total
80
60
40
20
0
0.0
0.2
0.4
0.6
0.8
1.0
size
1e6


Heapsort
1e7
Copias do Registro
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0
size
1e6


Heapsort
1e7
Comparações
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
size
1e6