UNIVERSIDADE DE BRASILIA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
DISCIPLINA: Estrutura de Dados **TURMA E**
1º SEMESTRE 2016

Alunos: Matheus Feitosa de Castro – 15/0141297
Rodrigo Demetrio Palma – 15/0147384

PROJETO DE PROGRAMAÇÃO 02

Índice
Página

Parte 1, Cenário 1:

Parte 2, Cenário 2:

Apresentação

A ordenação é um problema fundamental, para o ramo da computação, não é uma exceção. Inúmeras são as situações que requerem um conjunto de dados ordenados, como por exemplo nomes em ordem alfabética, extratos bancários onde se ordena pelo número dos cheques, etc. Na ciência da computação, tem-se vários algoritmos que ordenam um conjunto de dados, chamados *registros*. Cada registro contém uma *chave*, que é o valor a ser ordenado. Neste trabalho, cada registro tem somente a chave associada a ele pois o objetivo é entender a eficiência do algoritmo em diferentes estruturas de dados, porém caso sejam necessários outros componentes, basta adicioná-los á estrutura de dados, visto que esses componentes são ordenados junto aos índices.

Cenário 1:

Descrição sucinta sobre o desenvolvimento do trabalho

Esta parte tem o simples objetivo de testar a eficiência do algoritmo *quick sort* implementado em diferentes tipos abstratos de dados.

Foi desenvolvido então o *quick_vetor* que ordena chaves em um *Vetor* e o *quick_lista* que por sua vez, ordena chaves de uma *Lista.*

quick_vetor:

- A implementação do algoritmo quicksort aplicado a vetores não possuiu nenhuma mudança significativa do apresentado em sala de aula, sendo esta função recebendo como parâmetro, o vetor a ser ordenado, o índice do primeiro elemento do vetor, o índice do último elemento do vetor e um vetor utilizado como contador. Além de ser usada uma função que calcula a mediana entre 3 elementos.

quick_lista:

Decisões de implementação:

- A implementação do algoritmo quicksort aplicado a lista, em seu início ocorreram diversas falhas de implementação, sendo uma delas e uma das principais a falta de praticidade em se acessar um elemento que se encontra no meio da lista, porém se analisarmos a estrutura de um vetor, este não passa de uma lista numerada. Tendo em vista esta informação, o primeiro passo para a implementação do algoritmo foi definir estratégias para definir o acesso a um elemento que se encontra no meio da lista. Para tal, criamos um novo elemento na lista, o índice, o qual, por meio de um laço é utilizado como parâmetro para se mover até determinado elemento da lista, porém ao utilizarmos deste método no quicksort, ao ordenarmos uma lista com grande quantidade de elementos este processo fica muito mais demorado, visto que pode ser necessário percorrer vários elementos da lista. Tal fato nos levou a duas mudanças de implementação do quicksort. A primeira mudança consiste na passagem por parâmetro de um apontador correspondente ao elemento com índice i. A segunda mudança consiste em setar o apontador auxiliar correspondente ao elemento de índice j a ser igual ao apontador auxiliar correspondente ao elemento de índice i, e a partir deste se mover até sua posição. Tal mudança gerou uma diminuição do tempo de execução em aproximadamente 40%.

Comentários sobre os testes executados:

-Apesar das decisões de implementação terem diminuído o tempo de execução, pode-se perceber que a utilização do quicksort em uma lista é extremamente ineficiente, tal fato pode ser observado nos testes executados nas quais em todas as situações o quicksort aplicado a lista obteve pior rendimento se comparado ao quicksort aplicado ao vetor.

Descrição dos módulos e sua interdependência

Aqui separamos o projeto em 4 módulos:

1. *Cenário1.c:* Onde estão todas as funções utilizadas no programa principal.
2. *Cenário1.h:* Onde estão definidos os Tipos Abstratos de Dados que foram usados e também contém os protótipos das funções usadas.
3. *Cenário1main.c:* Onde encontra-se o programa principal e a lógica implementada.
4. *Makefile:* Onde foi implementada a interligação entre todos os modos, pela rotina Make.

Descrição dos TADs e as estruturas de dados utilizadas

Nesta parte foram utilizadas as Listas Duplamente Encadeadas. Estas Listas, chamadas aqui de *TipoLista* contém **dois** *apontadores:* primeiro e ultimo que vão apontar para a primeira e para a última *célula,* aos quais possuem **dois** componentes *inteiros:* a chave e o índice, que serão usados para a ordenação e **dois** *apontadores* prox e ant, que apontam para a próxima célula e para a célula anterior.

As funções utilizadas são operações de uma lista, como inserção, deleção, criação de lista vazia, etc.

Descrição do formato de entrada dos dados

A entrada de dados neste cenário é dada pela linha de comando, onde o primeiro parâmetro é o *nome do programa*, o segundo é a *semente do gerador de números aleatórios,* o terceiro é o nome do *arquivo texto com os valores de entrada* e o quarto e último é o *nome do arquivo texto que terão as estatísticas*.

Exemplo: *quick 3 entrada.txt saida.txt* - Onde o arquivo texto *entrada.txt* será do seguinte formato:

7 -> número de valores de N que se seguem, um por linha
1000
5000
10000
50000
100000
500000
1000000

Descrição do formato de saída dos dados

O formato de saída dos dados será um arquivo texto contendo as estatísticas dos algoritmos implementados.No exemplo anterior este arquivo teria o nome de *saida.txt.* Além disso terão mais quatro arquivos textos contendo todos os vetores e todas as listas antes da ordenação e depois da ordenação que estarão dentro da pasta *Ordenação*, com os seguintes nomes:

- Listas Desordenadas.txt
- Listas Ordenadas.txt
- Vetores Desordenados.txt
- Vetores Ordenados.txt

Explicação sobre como utilizar o programa

Para utilizar o programa deve-se ter os 4 módulos e o arquivo texto de entrada do formato explicitado, além de uma pasta chamada *ordenação*, para onde os arquivos textos vão.Tudo isso deve estar em uma única pasta.

Em seguida, deve-se abrir o terminal e encontrar a pasta. Feito isso basta digitar *make* e depois digitar os parâmetros necessários. Não será necessário fazer mais nada.

Estudo da complexidade do programa

mediana_vetor:

-Esta função é usada para calcular a mediana de 3 elementos, sua complexidade é O(1).

quick_vetor:

-A ordem de complexidade é calculada pela soma da complexidade da função *mediana_vetor* realizada (*log n* vezes) mais o número de acessos nas partições realizada (log n vezes), logo temos: O(log n) + O(n log n)=O(n log n).

mediana_lista:

-Função usada para calcular a mediana da chave de 3 nós da lista, sua complexidade é O(n).

quick_lista:

-Ordem de complexidade é calculada pela soma da complexidade da função *mediana_lista* realizada (log n vezes) mais o número de acesso nas partições realizada (log n vezes), logo temos: O(n log n)+ O(n log n)=O(n log n).

Portanto, a ordem de complexidade final do cenário 1 é: O(n log n), em que n representa o número de elementos a ser ordenado.

Listagem dos testes executados

quick_vetor, semente =1:

N (semente=1)	Tempo de execução (milissegundos)	Número de comparação de chaves	Número de cópias de registro
1000	0	13424	2837
5000	2	86622	16858
10000	2	180100	37045
50000	11	1047363	230166
100000	23	2145848	510408
500000	120	12494510	3145011
1000000	237	25741062	6796272

quick_lista,semente=1:

N (semente=1)	Tempo de execução (milissegundos)	Número de comparação de chaves	Número de cópias de registro
1000	6	30291	38489
5000	84	201074	251400
10000	283	417184	526881
50000	7725	2369427	3057141
100000	33771	5073157	6588573
500000	1070106	28526190	37959562
1000000	5779149	60581199	81053516

quick_vetor, semente=2:

N (semente=2)	Tempo de execução (milissegundos)	Número de comparação de chaves	Número de cópias de registro
1000	0	13341	2791
5000	2	86685	17019
10000	2	173158	37309
50000	11	1031792	230326
100000	27	2179529	510245
500000	201	11912113	3166481
1000000	429	26724926	6783113

quick_lista,semente =2:

N (semente=2)	Tempo de execução (milissegundos)	Número de comparação de chaves	Número de cópias de registro
1000	6	31376	39573
5000	95	185212	235426
10000	284	388915	499026
50000	8199	2425504	3109757
100000	46233	5050046	6567026
500000	1594358	29131612	38539460
1000000	5085250	58915852	79457366

quick_vetor, semente=3:

N (semente=3)	Tempo de execução (milissegundos)	Número de comparação de chaves	Número de cópias de registro
1000	0	13704	2861
5000	2	80579	17118
10000	3	179543	37177
50000	17	1060554	230359
100000	38	2194278	508849
500000	177	12217378	3152787
1000000	385	25416255	6812585

quick_lista, semente=3:

N (semente=3)	Tempo de execução (milissegundos)	Número de comparação de chaves	Número de cópias de registro
1000	7	31385	39720
5000	109	187083	237380
10000	426	402179	512203
50000	12409	2318295	3010290
100000	52992	5068564	6592064
500000	1508585	28413085	37848213
1000000	5554237	58345812	78874973

quick_vetor, semente =4:

N (semente=4)	Tempo de execução (milissegundos)	Número de comparação de chaves	Número de cópias de registro
1000	0	12978	2846
5000	2	82479	17051
10000	2	178235	37087
50000	11	1044777	230387
100000	23	2117654	511986
500000	138	12608915	3133553
1000000	332	24829297	6849001

quick_lista, semente=4:

N (semente=4)	Tempo de execução (milissegundos)	Número de comparação de chaves	Número de cópias de registro
1000	7	29517	37744
5000	104	185599	236073
10000	283	407633	517729
50000	8058	2339959	3032413
100000	36746	5099977	6619682
500000	1138612	28652821	38095824
1000000	5651768	62109885	82426911

quick_vetor, semente=5:

N (semente=5)	Tempo de execução (milissegundos)	Número de comparação de chaves	Número de cópias de registro
1000	0	13263	2818
5000	3	83010	17043
10000	2	182121	37297
50000	11	1010347	231585
100000	23	2259599	504267
500000	121	12373798	3143369
1000000	247	25655429	6852638

quick_lista, semente=5:

N (semente=5)	Tempo de execução (milissegundos)	Número de comparação de chaves	Número de cópias de registro
1000	7	29012	37238
5000	106	193900	243722
10000	287	394998	505995
50000	7939	2283153	2976648
100000	35166	5124289	6636529
500000	1099523	27932347	37413794
1000000	4707100	58393892	78900877

quick_vetor, valores médios:

N	Tempo de execução (c=2,2 milissegundos)	Número de comparação de chaves (k=13342)	Número de cópias de registro (z=2830,6)
1000	0 c	1 k	1 z
5000	1 c	6,3 k	6 z
10000	1 c	13,4 k	13,1 z
50000	5,5 c	77,9 k	81,5 z
100000	12,2 c	163,3 k	179,9 z
500000	68,8 c	923,5 k	1112,2 z
1000000	148,1 c	1924,2 k	2408,9 z

quick_lista, valores médios:

N	Tempo de execução (c=2,2 milissegundos)	Número de comparação de chaves (k=13342)	Número de cópias de registro (z=2830,6)
1000	3 c	2,3 k	13,6 z
5000	45,3 c	14,3 k	85,1 z
10000	142,1 c	30,1 k	181 z
50000	4030 c	175,9 k	1073 z
100000	18628 c	381 k	2331,9 z
500000	582834,9 c	2138,5 k	13414,6 z
1000000	2434318,5 c	4472,3 k	28313 z

Gráfico 1: Tempo de execução x N

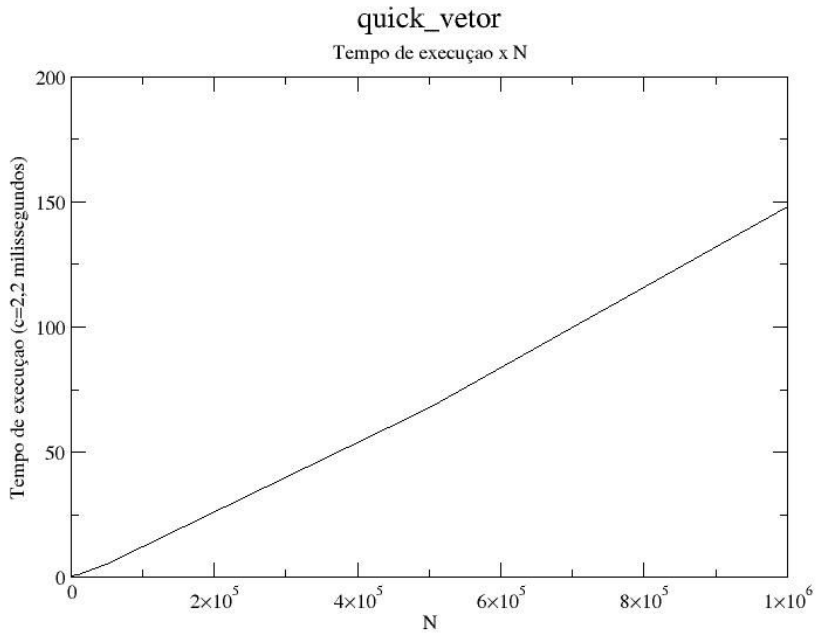


Gráfico 2: Tempo de execução x N

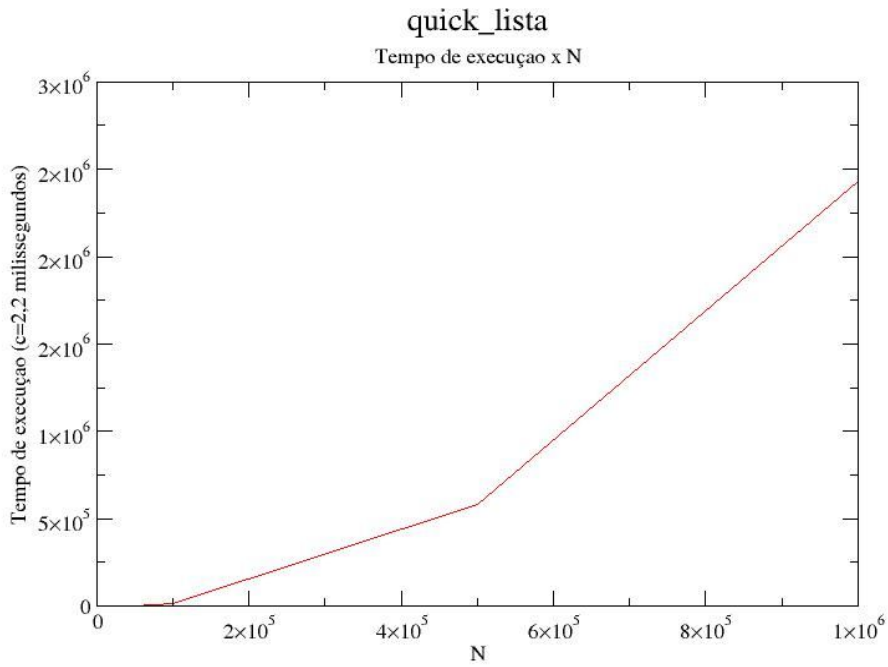


Gráfico 3: Comparação de chaves x N

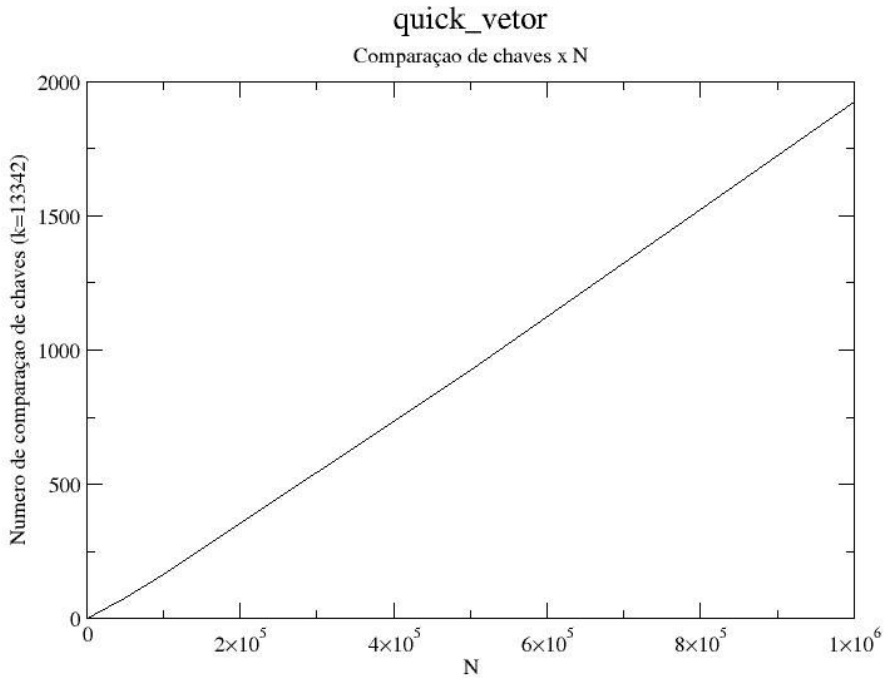


Gráfico 4: Comparação de chaves x N

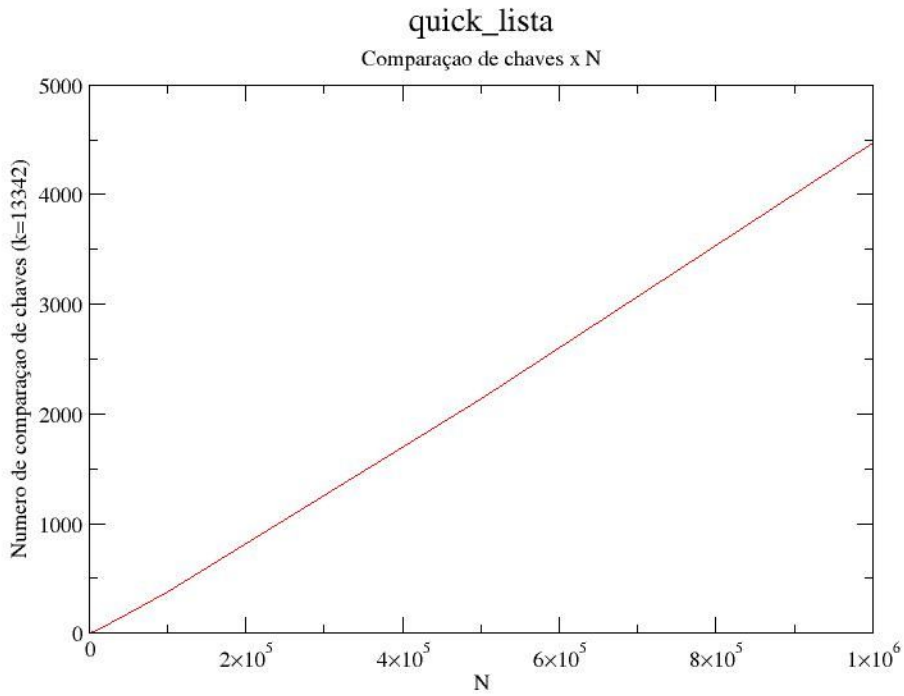


Gráfico 5: Copias de registro x N

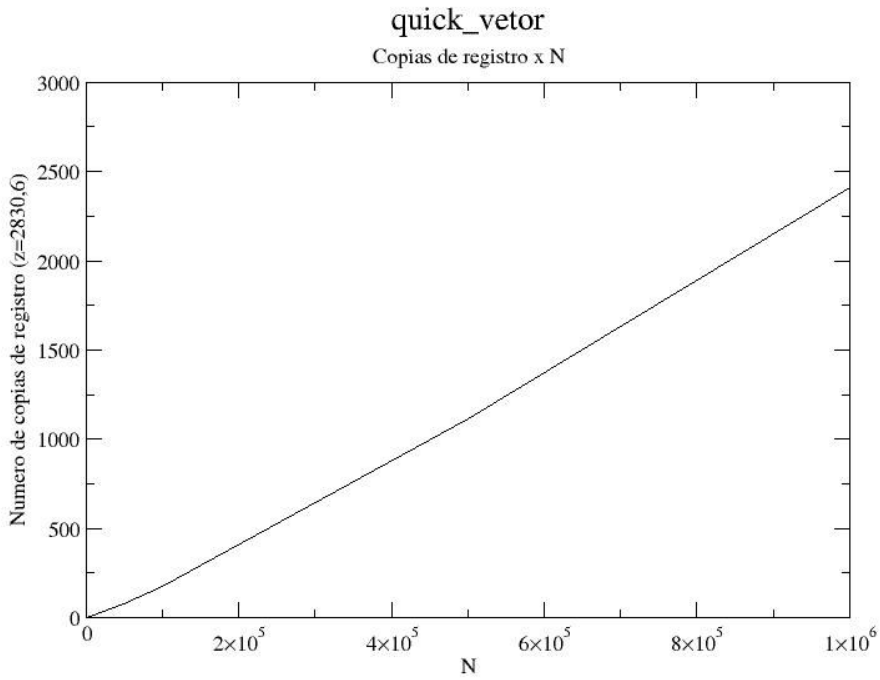
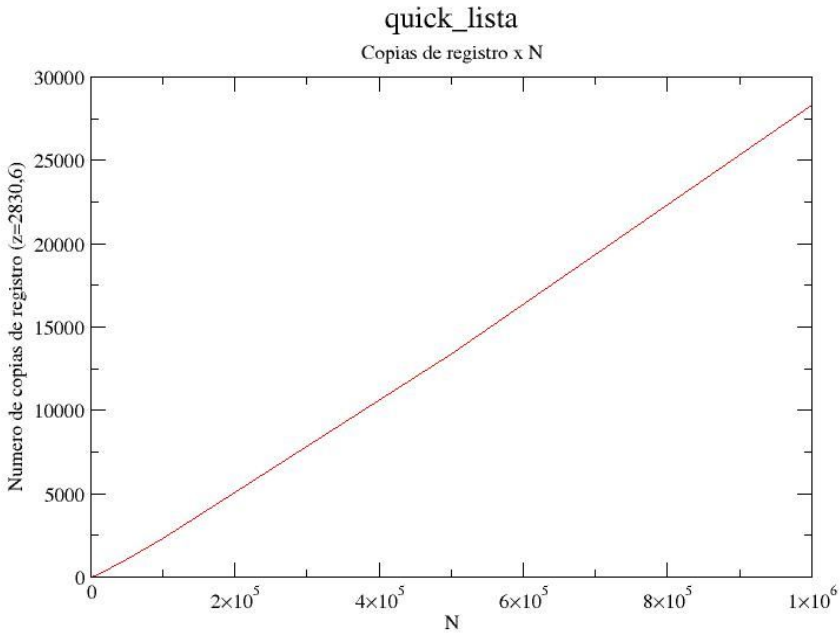


Gráfico 6:



Análise de resultados:

-Ao tomarmos nota da grande disparidade de tempo de execução, número de comparação de chaves e número de cópias de registros entre ordenar um vetor usando quicksort e ordenar uma lista usando o quicksort, é coerente afirmar que a ordenação da lista utilizando-se do algoritmo quicksort é ineficente, visto que este perdeu em todos os aspectos.

A principal causa do quicksort implementado a lista perder em tempo de execução, é a necessidade de percorrer toda a lista para se acessar um elemento que se encotra no meio da lista.

A razão do quicksort implementado a lista perder também em comparação de chaves é devido a necessidade de comparação quanto ao elemento índice da lista que é usado para se localizar determinado elemento da lista.

O motivo do quicksort implementado a lista perder em número de cópias de registro é devido ao mesmo fato de perder em tempo de execução, a necessidade de percorrer vários elementos de uma lista para se chegar a um elemento que se encontra no meio desta.

As tabelas e os gráficos acima comprovam tal fato, sendo uma saída mais eficiente a cópia dos valores da lista para um vetor e a ordenação deste vetor ou a escolha de outro método mais eficiente em listas como o heapsort.

Cenário 2

Descrição dos módulos e sua interdependência

Aqui separamos o projeto em 4 módulos:

5. *Cenário2.c:* Onde estão todas as funções utilizadas no programa principal.
6. *Cenário2.h:* Onde estão definidos os Tipos Abstratos de Dados que foram usados e também contém os protótipos das funções usadas.
7. *Cenário2main.c:* Onde encontra-se o programa principal e a lógica implementada.
8. *Makefile:* Onde foi implementada a interligação entre todos os modos, pela rotina Make.

Descrição sucinta sobre o desenvolvimento do trabalho

Esta parte tem o objetivo de comparar a eficiência nas variações do *quick sort* visando sua otimização. Foi implementado então três tipos deste algoritmo de ordenação para N elementos:

- *Quick Sort Recursivo:* Este é o algoritmo padrão visto em sala de aula
- *Quick Sort Mediana(k):* Esta variação utiliza *k* elementos aleatórios do vetor e pega sua mediana para ser o pivô.
- *Quick Sort Inserção(m):* Esta variação ordena normalmente até chegar a um sub-vetor de *m* elementos onde, a partir dele, começa a ordenar pelo algoritmo *Insertion Sort.*

Descrição dos TADs e as estruturas de dados utilizadas

Foram utilizados vetores e funções relacionadas a eles, as quais se seguem abaixo:

- *EncheVetor* - Preenche um vetor vazio com números aleatórios;
- *quick_recursivo* - é o Quick Sort Recursivo descrito acima;
- mediana_k - calcula a mediana de *k* números aleatórios do vetor
- *quick_mediana_k* - é o Quick Sort Mediana(k) descrito acima;
- *quick_insertion* - é o Quick Sort Inserção(m) descrito acima;
- *insertion_sort* - é o algoritmo de ordenação por inserção;
- *mediana_vetor* - calcula a mediana entre o primeiro, o meio e último valores de um vetor*;*
- *ImprimeVetorNoArquivo* - imprime o vetor em um arquivo.

Descrição do formato de entrada dos dados

Neste cenário, a entrada se diferencia do primeiro por dois novos parâmetros na linha de comando: o k e o m. O k é o número que será utilizado no algoritmo *quick_mediana_k* e o m é o número utilizado no algoritmo *quick_insercao*.

Exemplo de entrada: *quick 3 entrada.txt saida.txt 3 10* - Onde o arquivo *entrada.txt* terá o mesmo formato do cenário anterior.

Descrição do formato de saída dos dados

O formato de saída dos dados será um arquivo texto contendo as estatísticas dos algoritmos implementados. Além disso terão mais seis arquivos textos contendo todos os vetores antes da ordenação e depois da ordenação que estarão dentro da pasta *Ordenação*, com os seguintes nomes:

- *Vetores Desordenados no Quick Recursivo.txt*;
- *Vetores Ordenados pelo Quick Recursivo.txt*;
- *Vetores Desordenados no Quick Mediana(k).txt*;
- *Vetores Ordenados no Quick Mediana(k).txt*;
- *Vetores Desordenados pelo Quick Inserção(m).txt* e
- *Vetores Ordenados pelo Quick Inserção(m).txt.*

Explicação sobre como utilizar o programa

Para utilizar o programa do cenário 2 deve-se ter os 4 módulos e o arquivo texto de entrada do formato explicitado, além de uma pasta chamada *ordenação*, para onde os arquivos textos irão.Tudo isso deve estar em uma única pasta.

Em seguida, deve-se abrir o terminal e encontrar a pasta. Feito isso basta digitar *make* e depois digitar os parâmetros necessários. Não será necessário fazer mais nada.

Estudo da complexidade do programa

quick_recursivo:

Esta função de ordenação recursiva trabalha de modo que, a cada interação, ela divide o vetor na metade, fazendo isso n vezes, portanto esta função é O(n log n).

mediana_k:

Esta função pega k elementos aleatórios em um vetor e calcula a mediana dele, por meio do algoritmo de inserção, que é O(n^2), portanto esta função é O(k^2).

quick_mediana_k:

Esta função tem complexidade dada pela soma da função *mediana_k (O(k^2))* com a função *quick_recursivo (O(n log n)),* portanto têm complexidade O (n log n) para k's pequenos.

quick_insertion:

A ordem de complexidade desta função é O(n log n) + O(n*m), como a usamos para m's relativamente pequenos, então sua ordem de complexidade é: O(n log n).

insertion_sort:

Esta função tem ordem de complexidade O(n^2) no seu pior caso, porém a usamos aqui para n's pequenos.

Portanto, a complexidade neste cenário é de O (n log n), em que n é a quantidade de elementos a serem ordenados.

Listagem dos testes executados

quick_recursivo, semente 1

N	Tempo de Execução (milissegundos)	Número de comparações de chaves	Número de cópias de Registros
1000	0	13424	2837
5000	0	82580	17032
10000	1	178141	37100
50000	7	1066905	229827
100000	14	2145848	510408
500000	75	12296483	3152534
1000000	152	25741062	6796272

quick_mediana_k, semente = 1, k = 3

N	Tempo de Execução (milissegundos)	Número de comparações de chaves	Número de cópias de Registros
1000	1	13263	2877
5000	7	81051	17272
10000	15	173931	37268
50000	82	990800	231327
100000	169	2062312	512258
500000	917	11773035	3157270
1000000	1865	25192271	6809183

quick_mediana_k, semente = 1, k = 5

N	Tempo de Execução (milissegundos)	Número de comparações de chaves	Número de cópias de Registros
1000	3	12823	2911
5000	13	75597	17589
10000	15	162298	38065
50000	84	949284	234583
100000	174	2073327	514321
500000	962	11324057	3186150
1000000	1928	24084299	6877544

quick_insertion, semente 1, m = 10

N	Tempo de Execução (milissegundos)	Número de comparações de chaves	Número de cópias de Registros
1000	0	13043	2861
5000	0	86604	16859
10000	1	180101	37050
50000	7	1047367	230174
100000	15	2251679	506309
500000	80	12494544	3145022
1000000	161	25519279	6812507

quick_insertion, semente 1, m = 100

N	Tempo de Execução (milissegundos)	Número de comparações de chaves	Número de cópias de Registros

1000	0	16188	5360
5000	0	88261	18576
10000	1	183276	40200
50000	8	1050442	232182
100000	16	2254345	507291
500000	88	12494896	3145414
1000000	166	25520214	6813332

quick_recursivo, semente = 2

N	Tempo de Execução (milissegundos)	Número de comparações de chaves	Número de cópias de Registros
1000	0	13341	2791
5000	1	80836	17088
10000	1	179982	37239
50000	7	1053373	231369
100000	15	2179529	510245
500000	75	12166685	3152944
1000000	155	26724926	6783113

quick_mediana_k, semente = 2, k = 3

N	Tempo de Execução (milissegundos)	Número de comparações de chaves	Número de cópias de Registros
1000	4	13163	2917
5000	7	78773	17354
10000	15	170755	37483

50000	81	967821	232675
100000	182	2103647	509306
500000	932	11714452	3163998
1000000	1867	24817352	6835433

quick_mediana_k, semente = 2, k = 5

N	Tempo de Execução (milissegundos)	Número de comparações de chaves	Número de cópias de Registros
1000	2	12760	2881
5000	10	75985	17530
10000	16	170741	37454
50000	85	937066	235017
100000	174	2032867	514114
500000	952	11560879	3169060
1000000	1924	23767794	6888624

quick_insertion, semente = 2, m = 10

N	Tempo de Execução (milissegundos)	Número de comparações de chaves	Número de cópias de Registros
1000	0	13429	2868
5000	0	86736	17044
10000	1	173210	37337
50000	7	1053373	231369
100000	15	2179529	510245
500000	75	12166685	3152944

1000000	163	24810612	6864563

quick_insertion, semente = 2, m = 100

N	Tempo de Execução (milissegundos)	Número de comparações de chaves	Número de cópias de Registros
1000	0	15892	4697
5000	0	89723	19298
10000	1	175989	39937
50000	8	1032378	230903
100000	16	2271702	513659
500000	83	11912147	3166501
1000000	166	24830521	6862518

quick_recursivo, semente = 3

N	Tempo de Execução (milissegundos)	Número de comparações de chaves	Número de cópias de Registros
1000	0	13704	2861
5000	1	80924	17136
10000	1	172891	37404
50000	7	1022786	231243
100000	14	2194278	508849
500000	76	12039762	3157517
1000000	151	25416255	3152791

quick_mediana_k, semente=3, k = 3

N	Tempo de	Número de	Número de

	Execução (milissegundos)	comparações de chaves	cópias de Registros
1000	5	13096	2894
5000	8	79655	17324
10000	15	176033	37121
50000	81	1000829	231416
100000	172	2073346	513029
500000	960	11922634	3149579
1000000	1895	24399207	6865187

quick_mediana_k, semente=3, k = 5

N	Tempo de Execução (milissegundos)	Número de comparações de chaves	Número de cópias de Registros
1000	1	13021	2834
5000	7	75861	17486
10000	15	167131	38004
50000	85	938867	234319
100000	175	2018681	515450
500000	959	11434813	3177364
1000000	2000	23820260	6892569

quick inserção, semente =3, m =10

N	Tempo de Execução (milissegundos)	Número de comparações de chaves	Número de cópias de Registros
1000	0	13660	2839

5000	0	80527	17324
10000	1	179560	37198
50000	7	1060558	230361
100000	15	2170385	509562
500000	80	12217384	3152791
1000000	165	25384493	6836732

quick inserção, semente =3, m =100

N	Tempo de Execução (milissegundos)	Número de comparações de chaves	Número de cópias de Registros
1000	0	17043	5511
5000	0	80768	17349
10000	1	180396	37865
50000	8	1061789	231491
100000	16	2171217	510358
500000	82	12218167	3153458
1000000	168	25385269	6837437

quick_recursivo, valores médios

N	Tempo de Execução (milissegundos)	Número de comparações de chaves	Número de cópias de Registros
1000	0	13342	2830,6
5000	2,2	84054,6	16983,6
10000	2,2	178782,8	37080,86
50000	12,1	1039341,8	230693,9

100000	26,84	2178748,6	509224,9
500000	151,4	12321337	3148193,3
1000000	325,8	25672676,4	6818632,3

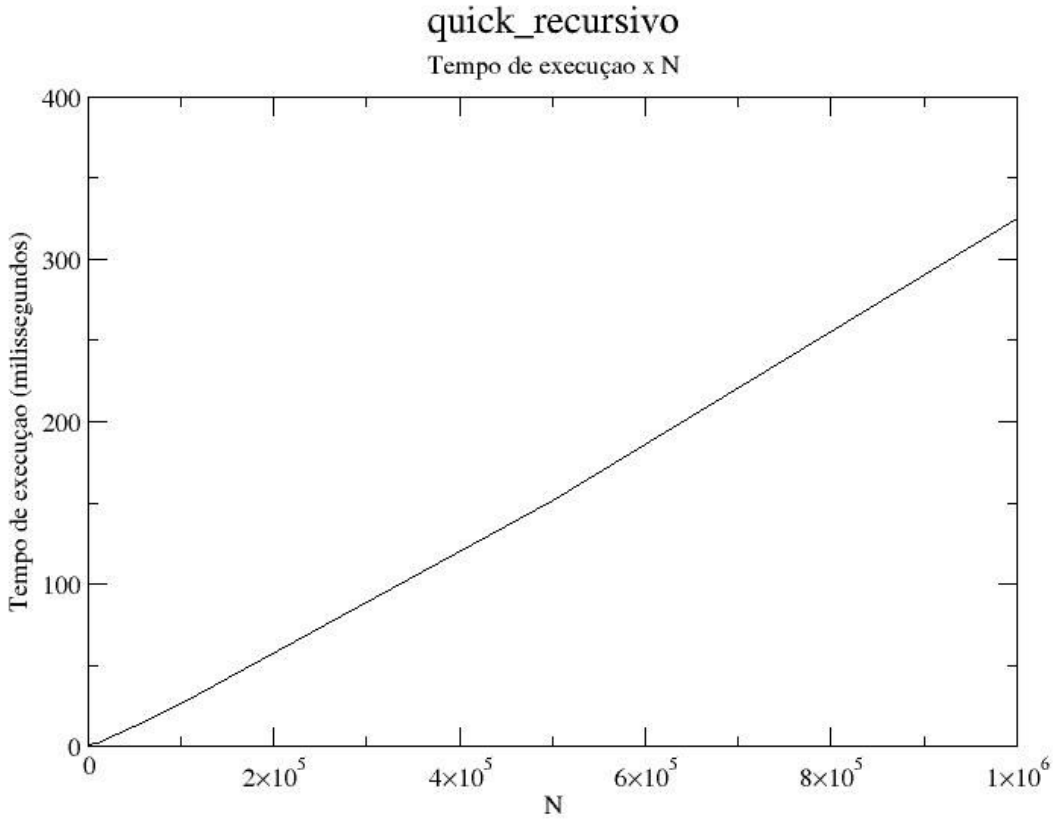
quick_mediana_k, k=3, valores médios

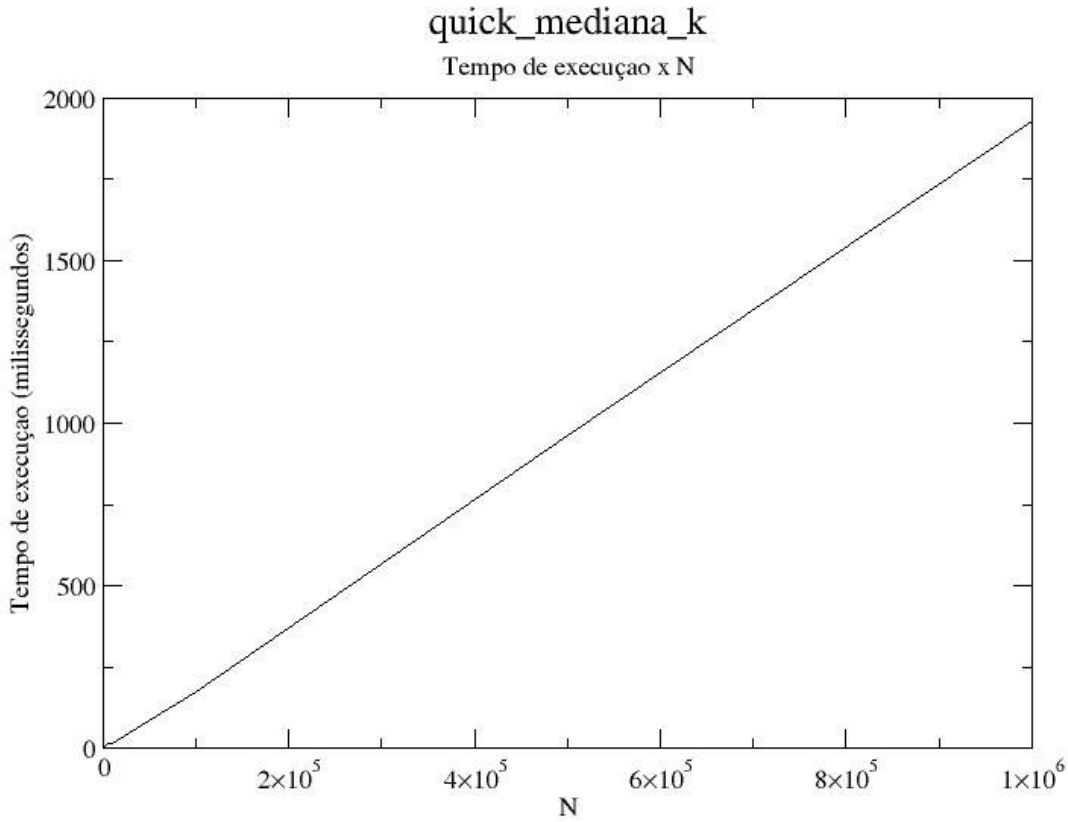
N	Tempo de Execução (milissegundos)	Número de comparações de chaves	Número de cópias de Registros
1000	3,3	13174	2896
5000	7,3	79826,3	17316,7
10000	15	173573	37290,7
50000	81,3	986483	231806
100000	174,3	2079768,3	511531
500000	936,3	11803373,7	3156949
1000000	1875,7	24802943,3	6836601

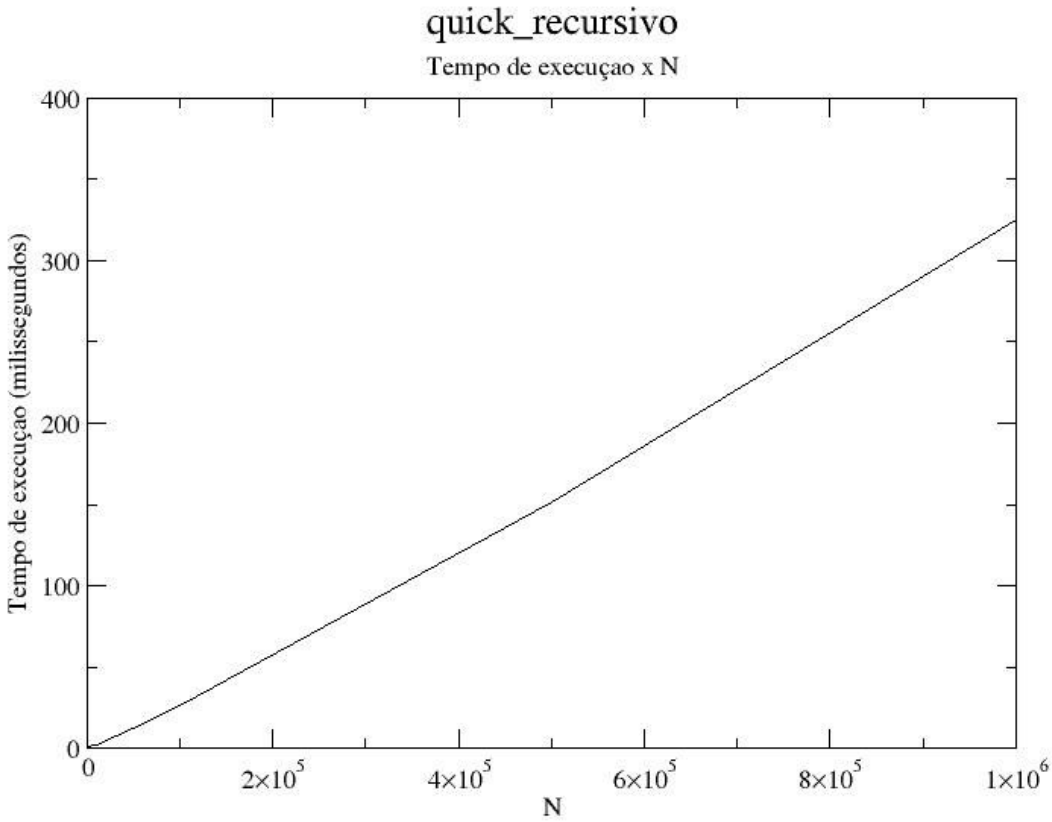
quick_insertion, valores médios, m = 10

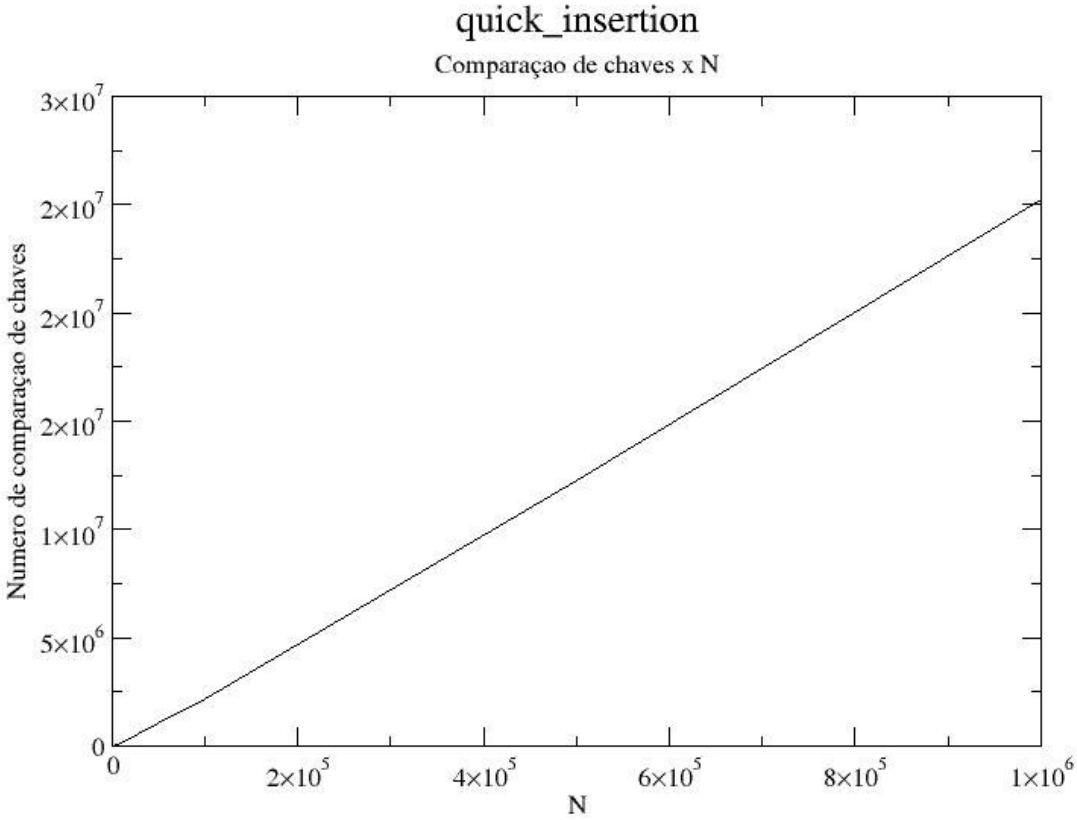
N	Tempo de Execução (milissegundos)	Número de comparações de chaves	Número de cópias de Registros
1000	0	13377,3	2856
5000	0	84622,3	17075,7
10000	1	177623,7	37195
50000	7	1053766	230634,7

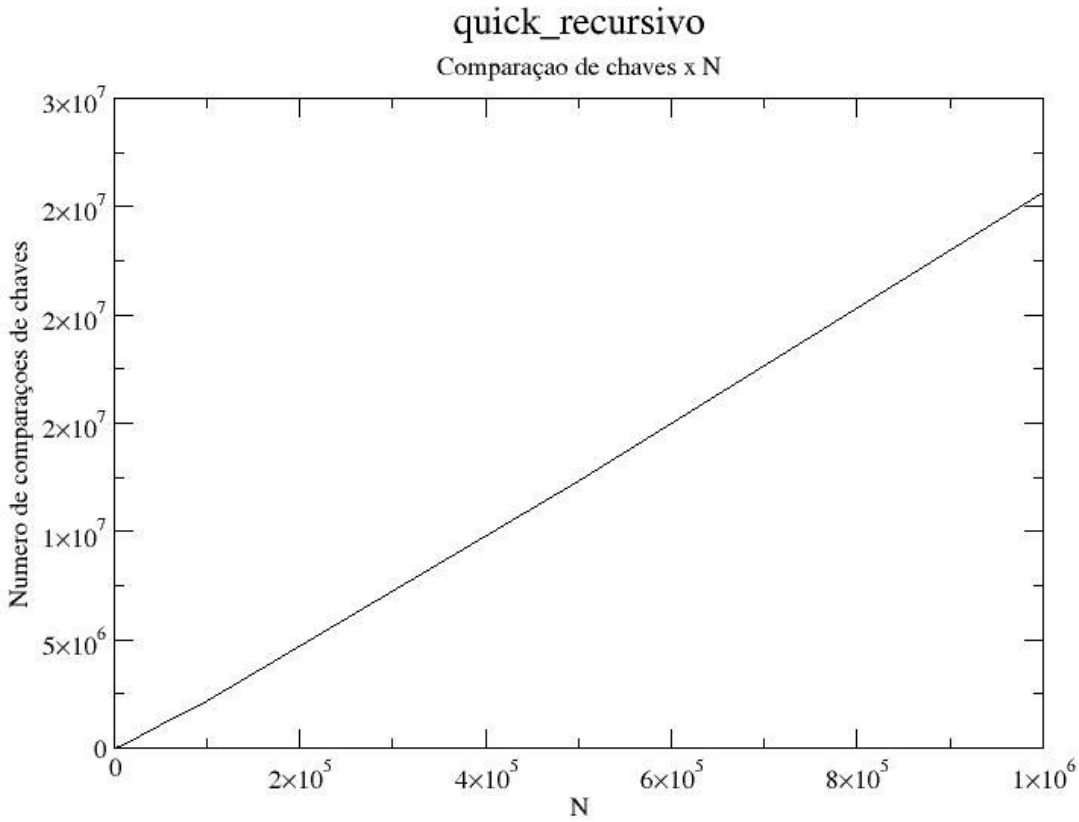
100000	15	2200531	508705,3
500000	78,3	12282971	3150252
1000000	163	25238128	6837934

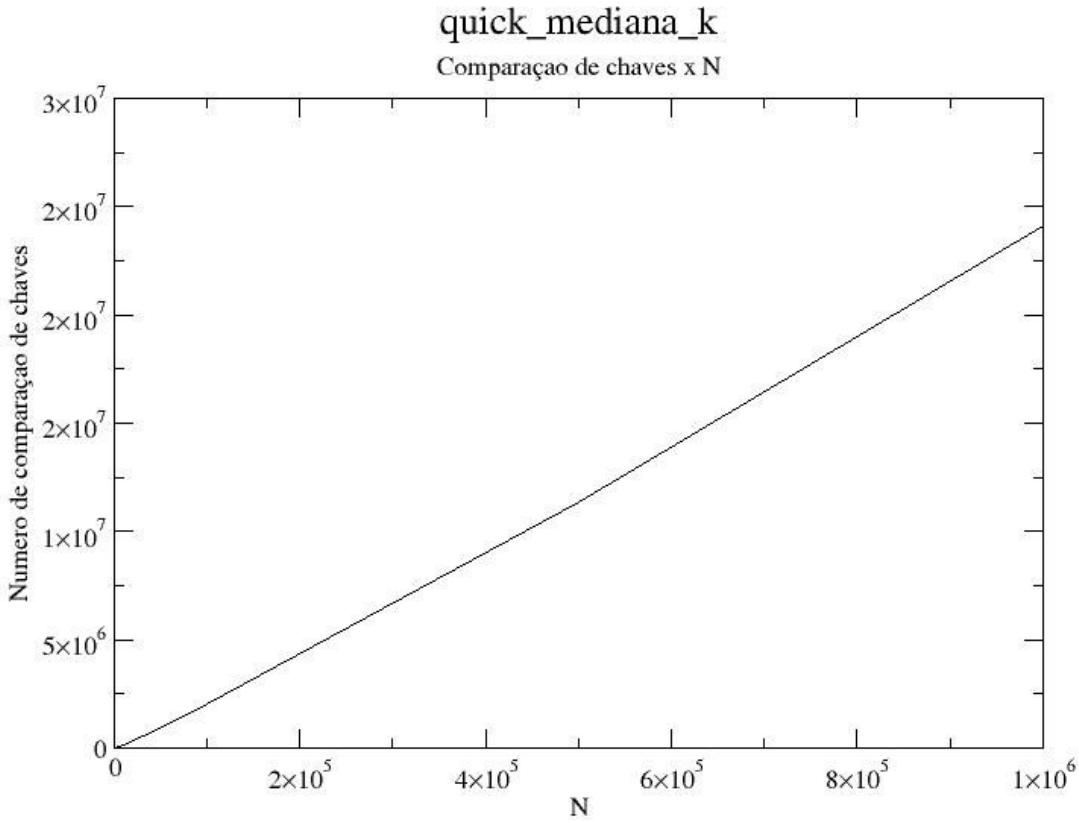


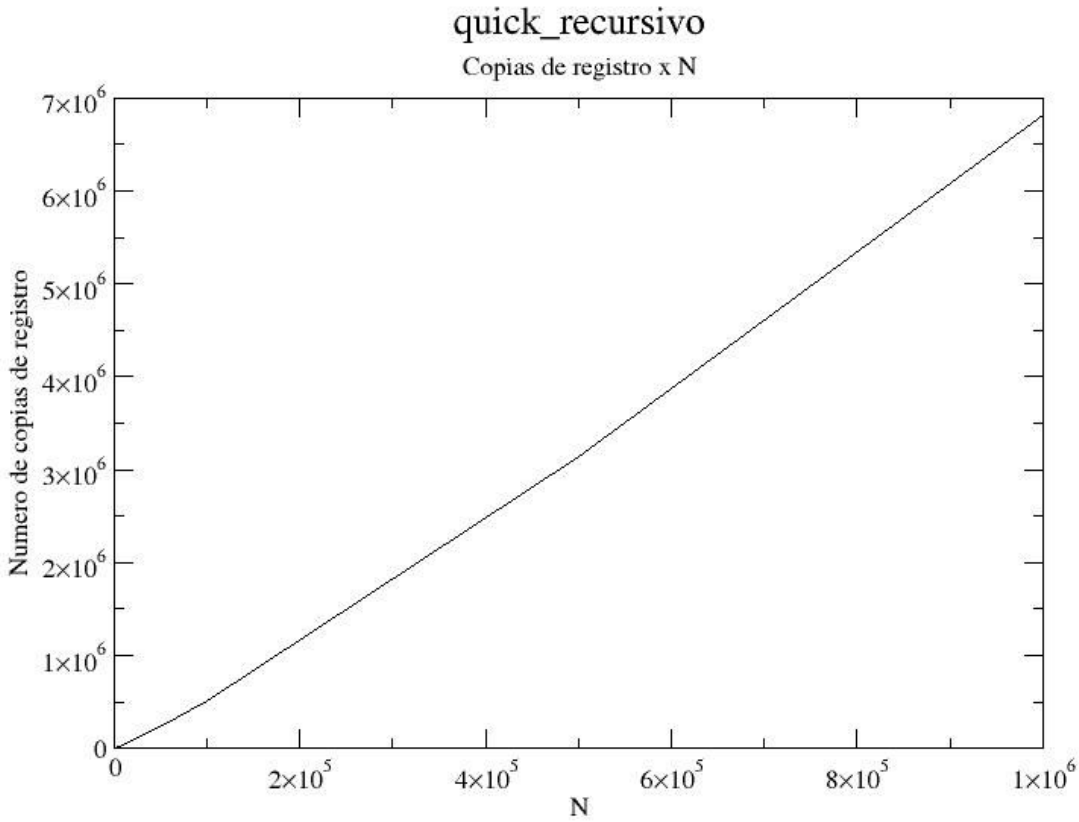

quick_mediana_k
Tempo de execuçao x N
Tempo de execuçao (milissegundos)
2000
1500
1000
500
0
0
2×10^5
4×10^5
6×10^5
8×10^5
1×10^6
N

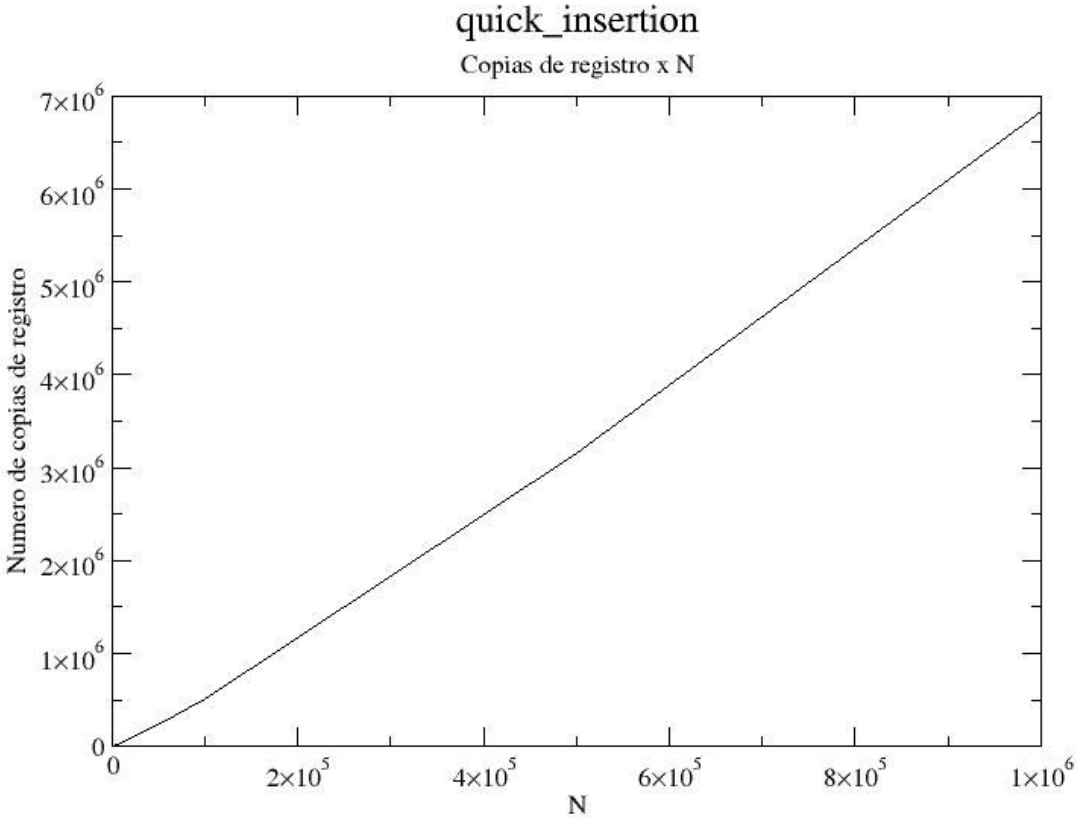

quick_recursivo
Tempo de execuçao x N
Tempo de execuçao (milissegundos)
400
300
200
100
0
0
2×10^5
4×10^5
6×10^5
8×10^5
1×10^6
N

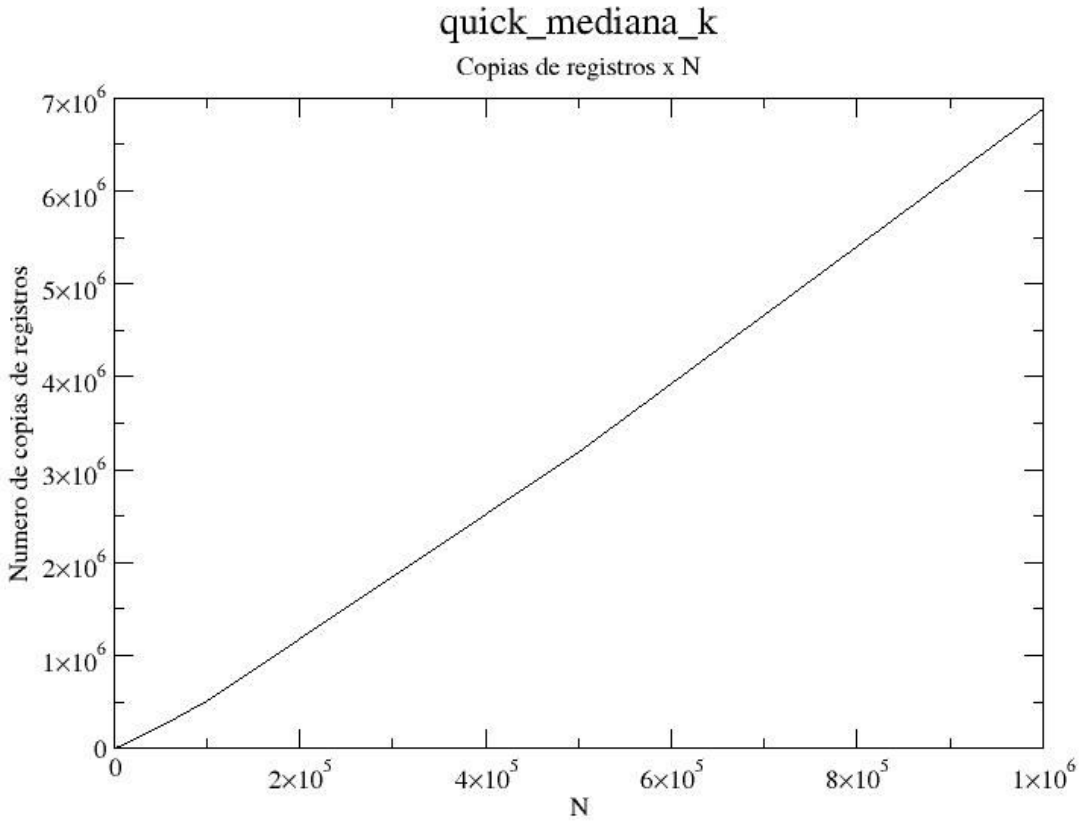

quick_insertion
Comparaçao de chaves x N
Numero de comparaçao de chaves
N
0
5×10^6
1×10^7
2×10^7
2×10^7
2×10^7
3×10^7
0
2×10^5
4×10^5
6×10^5
8×10^5
1×10^6


quick_recursivo
Comparaçao de chaves x N
Numero de comparaçoes de chaves
3×10^7
2×10^7
2×10^7
2×10^7
1×10^7
5×10^6
0
0
2×10^5
4×10^5
6×10^5
8×10^5
1×10^6
N


quick_mediana_k
Comparaçao de chaves x N
Numero de comparaçao de chaves
N
0
5×10^6
1×10^7
2×10^7
2×10^7
2×10^7
3×10^7
0
2×10^5
4×10^5
6×10^5
8×10^5
1×10^6


quick_recursivo
Copias de registro x N
Numero de copias de registro
N
0
1×10^6
2×10^6
3×10^6
4×10^6
5×10^6
6×10^6
7×10^6
0
2×10^5
4×10^5
6×10^5
8×10^5
1×10^6


quick_insertion
Copias de registro x N
Numero de copias de registro
N
0
1×10^6
2×10^6
3×10^6
4×10^6
5×10^6
6×10^6
7×10^6
0
2×10^5
4×10^5
6×10^5
8×10^5
1×10^6



Análise de dados

Quick_Recursivo:

Este algoritmo apresentou-se muito estável em relação aos três critérios estudados pois teve pouca variação quando mudou-se a semente.

Quick_Mediana_K:

A partir das tabelas acima e dos gráficos, foi observado que o *quick_mediana_k* teve um número menor de comparações de chaves, oque pode ser explicado pelo diferente método de escolher o pivô, fazendo com que este seja melhor escolhido na maioria das vezes. Por outro lado, este mesmo algoritmo teve uma defasagem no tempo de execução, visto que

foi o mais demorado em todos os casos. Isso se deve ao fato de que a função mediana_k tem $O(k^2)$, oque atrasa o algoritmo significativamente.

Percebeu-se que quando aumentou-se o k de 3 para 5, o número de comparações de chave diminuiu, oque é interessante quando se tem um alto custo para comparar algo. Os outros parâmetros não tiveram diferença significativa segundo nossas tabelas.

Quick_Insertion:

Esta variação do *quick Sort* apresentou-se muito interessante, pois foi a mais rápida se comparada aos outros dois algoritmos. Quanto aumentou-se o m de 10 para 100, o algoritmo começou a apresentar desvantagens somente nos casos em que o N era 1000. Isso é de fato, verdade, pois como foi utilizado o *insertion_sort*, no caso em que o N era pequeno, a maioria da ordenação foi feita pelo próprio *insertion_sort*, oque não é bom, visto que este algoritmo é $O(n^2)$.