Quicksort – algoritmo de ordenação

Estrutura de Dados – turma E - Professor: George Teodoro

Autores:

Bruno Fernandes Carvalho – 150007159

Leonardo Nunes Cornelio Rêgo- 150015046

Introdução

A principal finalidade desse trabalho é testar e analisar o desempenho do algoritmo de ordenação Quicksort para diferentes estruturas de dados, tamanhos de entrada e variações do algoritmo. Serão avaliados o tempo ordenação, a quantidade de comparações entre chaves feita e o número de cópias realizadas. O usuário deverá determinar num arquivo de entrada quantos conjuntos de elementos deverão ser gerados e o tamanho de cada conjunto, além de escolher uma semente para gerar esses dados aleatórios. Depois, será processada a média das métricas de desempenho para analisar esses dados.

A análise do algoritmo foi dividida em dois cenários, descritos a seguir:

Cenário 1: Impacto de diferentes estruturas de dados

Nessa parte, será avaliado o desempenho do Quicksort quando este ordena um vetor e uma lista duplamente encadeada com os mesmos valores gerados aleatoriamente.

Cenário 2: Impacto de variações do Quicksort usando um vetor

Nesse cenário, compara-se o desempenho do algoritmo para pequenas variações feitas nele. Primeiro será avaliado o algoritmo com a escolha do pivô na primeira posição do vetor, depois será analisado escolhendo o pivô como sendo a mediana de k elementos do vetor e, por último, modifica-se o Quicksort para ordenar vetores com m ou menos elementos usando o Insertion Sort.

Como executar o programa

Para executar o programa, foi usado o utilitário Make, já instalado no sistema operacional Linux. Como foram utilizados quatro módulos, foi necessário criar quatro objetos (arquivo .o) para relacioná-los e criar um executável chamado “main”. A seguir, mostra-se como deve ser gerado o arquivo makefile para a correta compilação dos módulos.

```
main: main.o logica.o out_in.o estruturas.o
	gcc logica.o main.o out_in.o estruturas.o -o main
main.o: main.c logica.h estruturas.h
	gcc -g -c main.c
logica.o: logica.c logica.h
	gcc -g -c logica.c
out_in.o: out_in.c out_in.h
	gcc -g -c out_in.c
estruturas.o: estruturas.c estruturas.h
	gcc -g -c estruturas.c
```

Como foi usado no programa a função getopt para receber argumentos da linha de comando, é necessário definir todos eles na hora de executar a “main”. A seguir, é mostrado o menu de ajuda para executar o programa.

```
[uso]./main <opcoes>
-e Nome do arquivo de entrada   Nome do arquivo onde serao lidas as entradas.
-s Nome do arquivo de saida     Nome do arquivo onde serao registrados os resultados.
-n Numero que representa a semente   Numero que gera as entradas aleatorias.
-c Numero do cenario que deseja utilizar  1 - Impacto de estruturas de dados diferentes. 2 - Impacto das variacoes do QuickSort.


 DIGITE OS QUATRO ARGUMENTOS PARA INICIAR
```

Assim, na linha de comando, deve ser escrito para executar:

./main -n 10 -e entrada.txt -s saida.txt -c 1

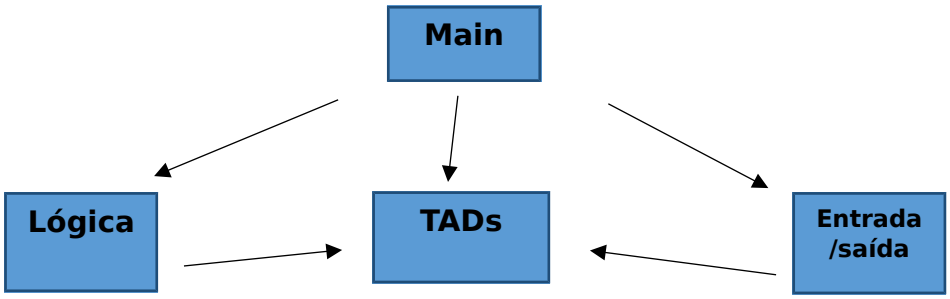
10 é o valor da semente, entrada.txt o arquivo texto contendo os dados de entrada com a quantidade de conjuntos que serão gerados, além de seus tamanhos, saida.txt, correspondente ao arquivo texto de saída das métricas analisadas e 1, que determina o cenário de testes que serão feitos. O arquivo de entrada deve seguir a formatação: <número de conjuntos>, <quantidade de elementos em um conjunto>, cada um desses dados numa linha do arquivo, conforme mostrado a seguir:

```
7 <número de conjuntos>
1000  <quantidade de elementos em um conjunto>
5000
10000
50000
100000
500000
1000000
```

Para analisar da melhor maneira possível a influência das estruturas de dados e das variações do algoritmo, prefiriu-se dividir essa análise nos dois cenários já descritos. Assim, toda vez que o usuário executar o programa, ele deve escolher qual cenário deseja avaliar. Se o cenário 1 for escolhido, os mesmos elementos em estruturas de dados diferentes serão ordenados, e no arquivo de saída estará apenas a análise entre essas duas variações. Já se o cenário 2 for escolhido para ser avaliado, o arquivo de saída conterá as comparações da ordenação de um vetor usando o Quicksort recursivo sem mediana, Quicksort com mediana usando k=3 e k=5, e Quicksort com Insertion Sort usando m=10 e m=100.

Descrição dos módulos

O algoritmo foi dividido em quatro módulos: um que contém o programa principal (main), um que contém a lógica do algoritmo de ordenação, um que gerencia a entrada e saída de dados e o último que contém os TADs (tipo abstrato de dados) e as estruturas de dados utilizadas. O programa principal depende de todos os módulos, além destes serem interdependentes, como mostrado na figura a seguir:



-Programa principal (main)

A main é responsável por articular as chamadas da maioria das funções para o funcionamento correto do código. Nela, também é usado a primitiva getopt para receber argumentos da linha de comando do terminal. Essa função recebe o nome do arquivo texto de entrada, o nome do arquivo de saída, a semente do gerador de valores aleatórios e o cenário de análise. Esse módulo também é responsável por declarar as estruturas de dados usadas, chamar as funções que gerenciam a entrada e saída de dados, assim como as funções que contém a lógica do programa e a função que dá o tempo de execução de um determinado trecho do programa.

-Lógica do programa

Essa parte do programa contém as funções responsáveis por terem toda a lógica dos algoritmos de ordenação e suas variações, além de funções que geram dados aleatórios para análise. Ele tem funções do Quicksort responsáveis por ordenar vetores e listas duplamente encadeadas, uma função do Insertion Sort e outra que calcula mediana para um determinado valor k. O cabeçalho das funções é mostrado a seguir:

void quicksort_array(int inicio, int final, int vetor[], int k, int m, int* copias, int* comp);
int mediana(int k, int vetor[], int inicio, int final);
void insertion_sort(int inicio, int final, int vetor[], int* comp, int* copias);
void quicksort_lista(apontador inicio, apontador final, int* copias, int* comp);
void gerar1(int vetor[], int n, int semente, tipolista* lista);
void gerar2(int vetor[], int vetor2[], int vetor3[], int vetor4[], int vetor5[], int n, int semente);

-Entrada e saída de dados

Esse módulo é responsável somente por organizar a entrada e saída de dados, além de mostrar o menu de ajuda relacionado com a função getopt. O cabeçalho das funções é mostrado abaixo:

void mostra_ajuda(char *name);
void saida1(FILE** fp1, char saida[], int tmili, int tmili2, int comp1, int comp2, int copias1, int copias2, int n);
void saida2(FILE** fp1, char saida[], int tmili, int tmili2, int tmili3, int tmili4, int tmili5, int comp1, int comp2, int comp3, int comp4, int comp5, int copias1, int copias2, int copias3, int copias4, int copias5, int n);
void entrada_arquivo(FILE** fp, int *valor);
void limpar_arquivo(FILE** fp1, char saida[], int cenario);

-TADs e estrutura de dados

Nessa parte do algoritmo estão os tipos abstratos de dados (especificação matemática que determina a implementação das estruturas de dados) e as próprias estruturas. Dois tipos são criados: tipolista que define a lista e tipocelula contendo os elementos da célula e dois apontadores para a célula seguinte e anterior da lista. Também foi criado um tipo apontador, que aponta para as células criadas. Essas estruturas são mostradas a seguir:

```
typedef struct tipocelula *apontador;
typedef struct tipocelula{
   int elemento;
   apontador prox, ant;
 }celula;
typedef struct{
   apontador primeiro, ultimo;
}tipolista;
```

Além disso, diversas operações matemáticas são feitas com as estruturas declaradas no programa principal. Entre elas, pode-se citar a verificação da posição de dois elementos numa lista, a inserção, desalocação, troca de posição e criação de uma lista por ponteiros vazia. Além destas, uma função opera com elementos de um vetor, trocando dois destes de posição. O cabeçalho de cada uma dessas funções é mostrado abaixo:

```
void flvazia(tipolista *lista);
void insere(tipolista *lista, int x);
void troca_array(int index1, int index2, int vetor[]);
void troca_lista(apontador *ptr1, apontador *ptr2);
int verifica_posicao(apontador big, apontador small);
void libera(tipolista *lista);
```

Análise dos algoritmos e complexidade

Cada módulo possui rotinas que apresentam papel fundamental na execução correta do programa, e serão analisados os algoritmos com o objetivo de entender as escolhas feitas, entender como eles funcionam e seus pontos fortes e fracos. Além disso, será analisado a complexidade dessas funções para determinar a otimização do código.

-Programa principal (main)

Na main, além de pegar e atribuir as variáveis da linha de comando pelo getopt, um teste de condição essencial define qual será o rumo que o código irá tomar. Se o cenário 1 for escolhido, um vetor e uma lista duplamente encadeada são utilizados para fazer a ordenação usando diferentes tamanho de conjuntos. Tem um loop que ordenará e gerará diferentes dados enquanto houver algo para ser lido no arquivo de entrada. Cada vez que um vetor ou lista for ordenado, é usado a função “gettimeofday” para saber qual foi o tempo de execução desse trecho do programa, ou seja, tempo para ordenar determinada estrutura de dado. Essa função pega o instante de tempo antes e depois da ordenação, e calcula essa diferença para saber o tempo de execução. Toda vez é passado para a função de ordenação as variáveis que armazenam as

métricas que analisam o número de comparações feitas e o número de cópias realizadas.

Já no cenário 2, cinco vetores são criados igualmente para uma determinada quantidade de elementos em um conjunto. Esses vetores são ordenados usando variações diferentes do Quicksort, e em cada uma delas, é analisado o tempo de execução, o número de comparações feita e o número de cópias realizada. Optou-se por realizar cinco ordenações usando o método recursivo, usando a mediana para k=3 e k=5 valores e combinando o algoritmo com o Insertion Sort para ordenar subvetores com m=10 e m=100 elementos.

Análise de complexidade

Como a função principal será executada q vezes, sendo q o número de conjuntos a serem ordenados, conclue-se que a ordem de complexidade será q vezes a complexidade do algoritmo de ordenação. Além disso, a complexidade da ordenação será multiplicada por uma constante que representa quantas vezes a função de ordenação foi chamada, dependendo do cenário de testes. Considerando o caso médio, em que a ordenação usando Quicksort é O(n logn), sabe-se que a ordem de complexidade da main é O(qn logn).

-Lógica do programa

Foi necessário criar duas versões diferentes do algoritmo Quicksort: uma para arranjos e outra para listas duplamente encadeadas, sendo que as duas recebem sempre como parâmetro uma variável que será responsável por calcular o número de comparações feitas e outra que representará o número de cópias realizadas.

A função responsável por ordenar vetores recebe como parâmetro a posição inicial e final do subvetor que se deseja ordenar, além dos valores de k e m, relacionados, respectivamente, à quantidade de elementos que serão enviados para a função que calcula a mediana e à quantidade de elementos de um subvetor que deverão ser ordenados pelo Insertion Sort. Isso significa que toda vez que um subvetor tiver m ou menos elementos, eles serão ordenados por Inserção. Essa função é adaptável a todos os cenários de testes, já que preferiu-se fazer uma rotina genérica que abrange todas as variações do algoritmo. Vale deixar claro que se for escolhido k = 0, nenhum elemento será enviado para a função que calcula a mediana e o pivô será sempre o primeiro elemento do subvetor. Além dessa observação, outra coisa importante a ser dita é que se for escolhido m=0, a função Insertion Sort fará apenas um teste e o algoritmo de ordenação representará o caso recursivo.

A função que ordena uma lista duplamente encadeada teve que passar por muitas mudanças em relação ao algoritmo de arranjos. Agora não é possível comparar se um índice é maior ou menor que outro, tornando esse algoritmo mais complexo. Será sempre necessário comparar ponteiros e verificar se a posição inicial e final foram alteradas. Além disso, toda vez que o ponteiro responsável pelo “too_small_index” mudar de posição, tem que verificar se ele passou pela célula que o ponteiro “too_big_index” aponta. Essa informação verificada acima será fundamental para determinar a condição de parada do loop principal da função. Como esse algoritmo será utilizado somente no cenário 1 de testes, ele não recebe como parâmetro os valores relacionados à mediana e ao Insertion Sort.

Além dessas duas funções apresentadas acima, que contém o núcleo do trabalho, existem mais duas que se encaixam na lógica do programa: uma apresenta o algoritmo de ordenação do Insertion Sort e outra que calcula a mediana para k valores recebidos como parâmetro. Esta foi feita para casos genéricos, ou seja, independentemente da quantidade de elementos que participarão da escolha do pivô, o algoritmo funciona corretamente. Isso acontece já que a lógica dessa rotina consiste em ordenar k elementos em um vetor auxiliar usando o Insertion Sort e escolher o elemento central para ser a mediana.

Mais duas funções presentes nesse módulo são essenciais para que o algoritmo analise entradas aleatórias. Estas são responsáveis por gerar dados aleatórios para um certo tamanho de conjunto determinado no arquivo de entrada. Ainda, essa tarefa foi dividida em duas funções diferentes, uma responsável por cada cenário.

Análise de complexidade

As funções de Quicksort vistas acima apresentam ordem de complexidade para um caso geral de O(n logn), considerando que os elementos estão bem distribuídos e a escolha do pivô seja satisfatória. Vale deixar claro que n é a quantidade de elementos presente em uma lista ou em um vetor. Dependendo da forma que os elementos estão arranjados na estrutura e considerando uma má escolha do pivô, esse algoritmo pode chegar a $O(n^2)$, representam um resultado ruim. Para evitar o pior caso, a escolha da mediana entre k valores é uma opção boa, e o algoritmo apresentado nessa programa possui ordem de complexidade $O(k^2)$, pois sabe-se que esses k valores são ordenados usando Insertion Sort. Para valores de k pequeno, o programa em geral funciona bem e apresenta resultados satisfatórios. Porém, se k for muito grande, o tempo de execução pode aumentar bastante. Além disso, a função Insertion Sort apresenta $O(n^2)$ para casos médios, e no melhor caso, quando o vetor está quase ordenado, pode chegar a O(n). As

funções que geram dados aleatórios possuem O(n), sendo n o número de elementos a serem criados.

-Entrada e saída de dados

O algoritmo de entrada e saída de dados é bastante simples, contendo apenas funções responsáveis por receber os dados de entrada, imprimir os dados de saída e o menu de ajuda para executar corretamente o programa pelo terminal. A função que lê os dados de entrada lê o valor de uma linha do arquivo, que contém a quantidade de elementos a serem ordenados, e depois espera que seja chamada novamente, para ler a próxima linha deste. Para cada valor lido, é executado a ordenação para os diferentes casos no cenário de teste presente.

A função que mostra as métricas analisadas num arquivo de saída recebe esses valores já prontos e apenas mostra para o usuário de forma organizada. Vale deixar claro que a função de saída foi dividida para cada cenário em análise. Além disso, existe uma função que cria ou apaga tudo do arquivo de saída escolhido pelo usuário, e ainda indica qual cenário está sendo analisado.

Análise de complexidade

Como todas as funções desse módulo não apresentam iteração e são apenas responsáveis por ler ou mostrar dados para o usuário, todas apresentam O(1), indicando que a ordem de complexidade é constante e não é um custo alto no tempo de execução do programa.

-TADs e estrutura de dados

O algoritmo descrito nesse módulo é responsável por dar suporte à lógica do programa. Aqui são definidas diversas funções que fazem operações com os TADs, além do próprio tipo e das estruturas que estão contidas nele. Usou-se listas duplamente encadeadas e arranjo para representar os dados de entrada no cenário 1 de testes e apenas arranjos no segundo cenário.

Existem várias funções que operam sobre as listas criadas, já comentadas na descrição dos módulos. As funções de inserção

basicamente adicionam a célula à lista desejada, e as funções de liberação são responsáveis por desalocar algum espaço de memória. Existe também uma função para criar uma lista duplamente encadeada vazia e outra responsável por trocar dois elementos de posição. Essa última é a mais complexa do módulo e teve que ser dividida em dois casos: um que troca dois elementos adjacentes na lista encadeada e outro caso que representa a troca de dois elementos em qualquer outra posição da lista. Para essa troca de posição, dois apontadores auxiliares devem ser criados, representando um aspecto ruim, já que cópias na memória deverão ser feitas. Além dessas, a função Verifica_posicao é responsável por verificar se o ponteiro “too_small_index” já ultrapassou o ponteiro “too_big_index” na lista em ordenação.

Como visto acima, várias operações foram necessárias para manipular a lista e os tipos criados. Já a manipulação de vetores é mais simples, e a única função presente nesse módulo é a responsável por trocar dois elementos de posição. Nela, precisa-se criar apenas uma variável auxiliar para que ocorra a troca de forma correta, mostrando que essa troca custa menos memória que a descrita no parágrafo anterior.

Análise de complexidade

A complexidade da maioria dessas funções são simples e apresentam O(1), já que elas consistem em testes de condição e atribuições, que são operações que não têm custo alto para o tempo de execução. Apenas uma função tem iteração: a que desaloca as células de uma lista. Esta é responsável por percorrer toda a lista, apresentando, portanto, O(n), sendo n o número de elementos na lista.

Análise de dados

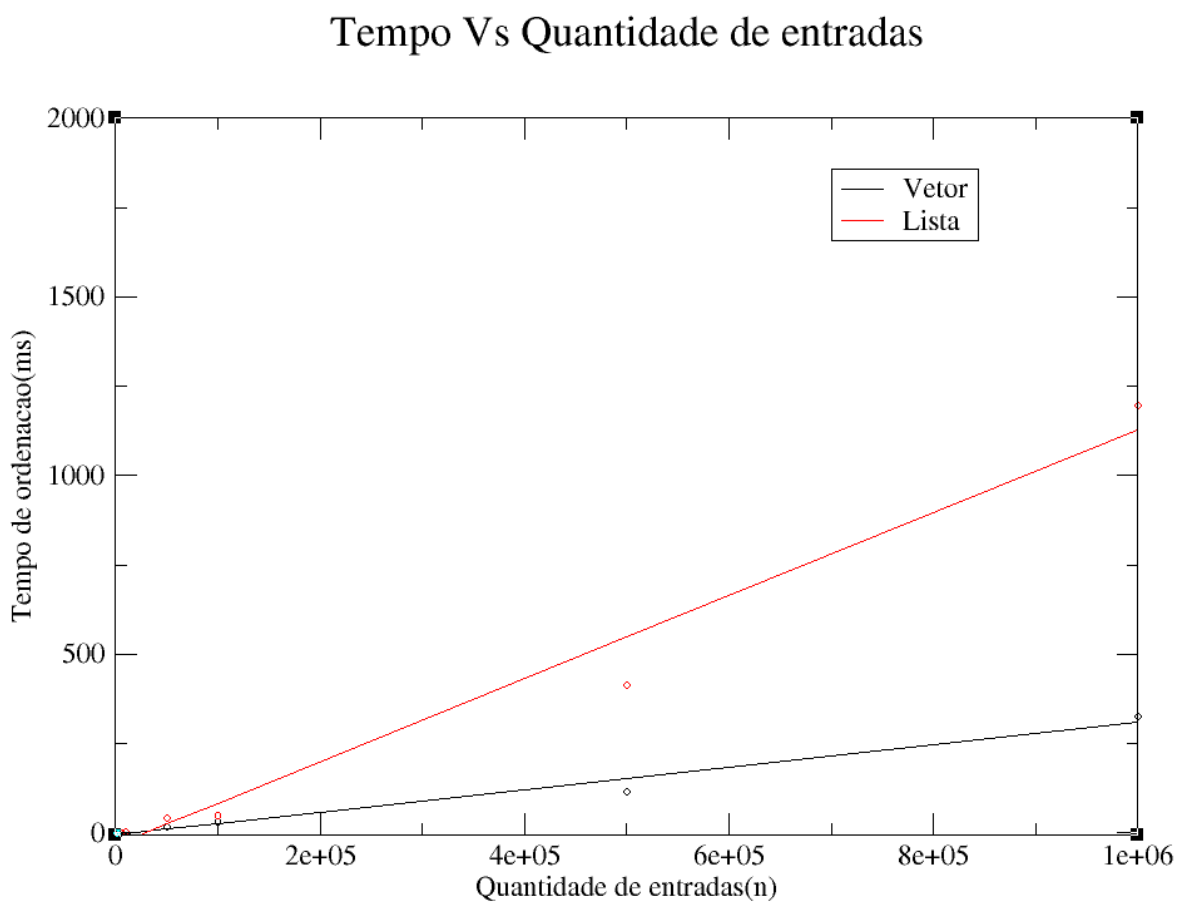
Cenário 1: Impacto de diferentes estruturas de dados.

O algoritmo Quicksort(Recursivo) foi utilizado em entradas de tamanho N variados. Para cada valor de N, gerou-se 5 entradas diferentes, ao se usar sementes diferentes no gerador de números aleatórios. A partir desses 5 resultados gerados para cada valor de N, calculou-se a média deles.

Quantidade de entradas	Tempo médio no vetor(ms)	Comparações no vetor	Cópias no vetor	Tempo médio na lista(ms)	Comparações na lista	Cópias na lista
1000	0	11580	2731.2	0	10411.1	5130
5000	0	73913.8	16372.4	0.6	68062.4	31102.8

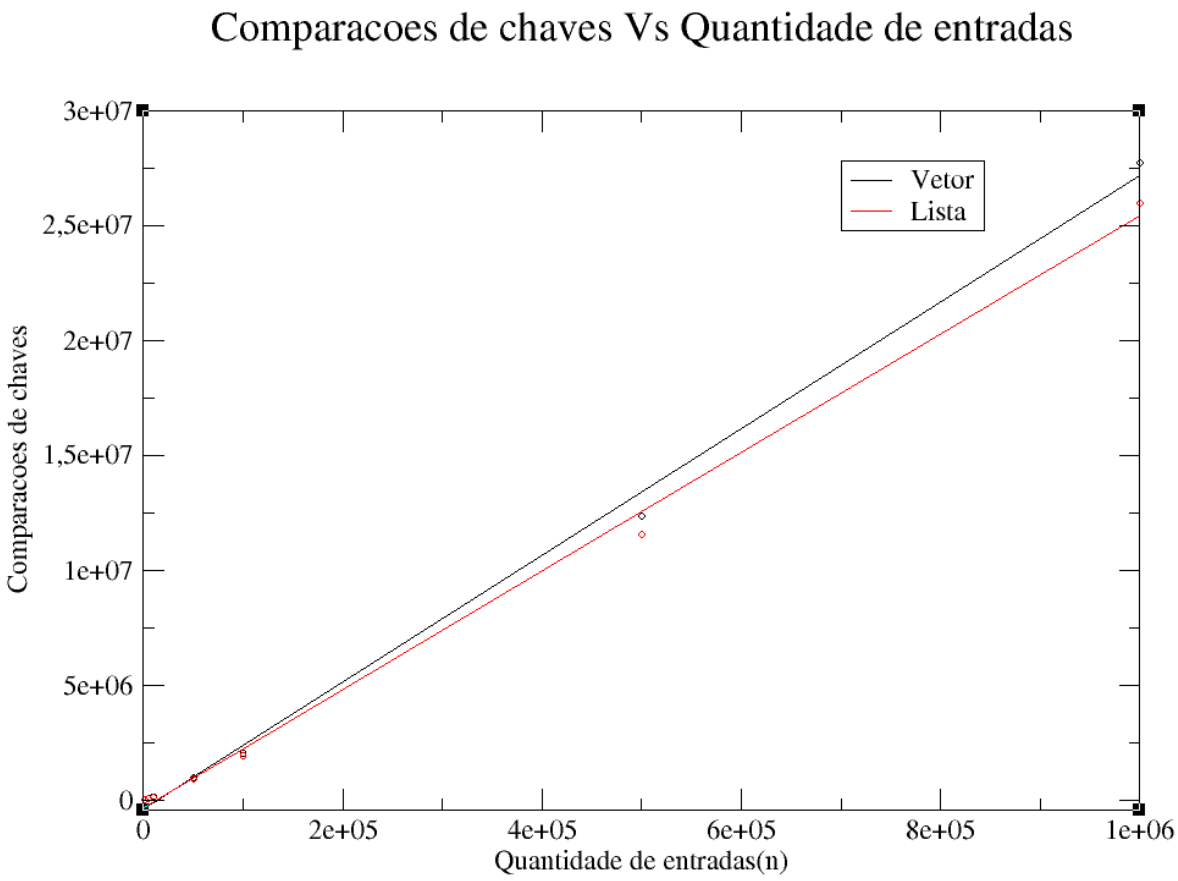
10000	1.4	162517	34988.8	3	150747.8	66886
50000	17	990901.6	200802.4	41.4	930380	389525.6
100000	30.8	2074092.8	426034.4	47.2	1948175.4	839373.6
500000	113.2	12387001.2	2339227	412.6	11578089	5107281.2
1000000	326.4	27780545.4	4768252	1195.8	25980418.2	10936922

Plotou-se gráficos para se realizar uma análise mais precisa ao se comparar os dois métodos.

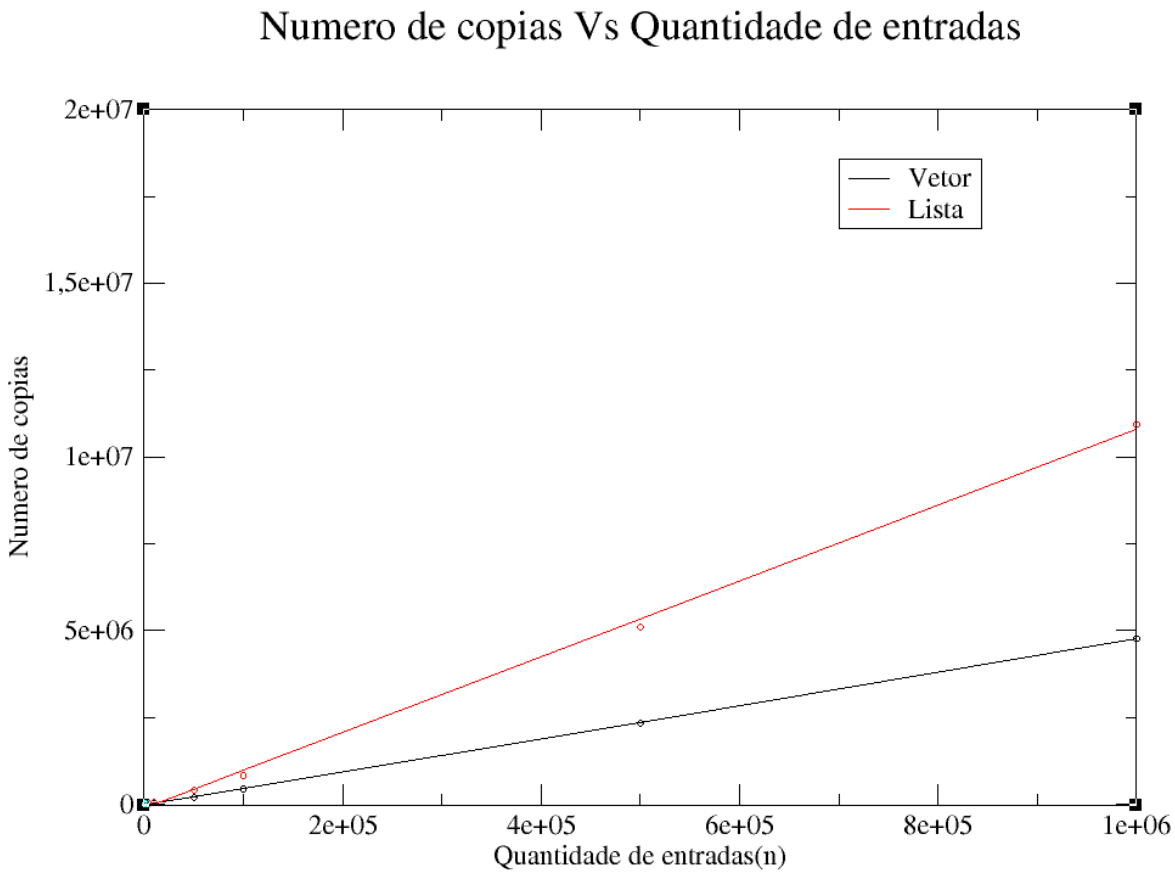


Com os dados obtidos da regressão, sabemos que o coeficiente angular da reta que representa o vetor é 0,00031244 e o da lista é

0,0011594. O coeficiente angular neste gráfico representa o quanto o tempo de ordenação varia, quando variamos em 1 unidade a quantidade de entradas, ou seja, se aumentarmos a quantidade de entradas em 1, o tempo de ordenação do vetor aumentará em 0,00031244 ms. Logo, o melhor método, em relação ao tempo, é o que possui o menor coeficiente angular, portanto, utilizar vetores para a ordenação é mais vantajoso se estivermos preocupados com o tempo de ordenação.



Com os dados obtidos da regressão, sabemos que o coeficiente angular da reta que representa o vetor é 27,577 e o da lista é 25,786. O coeficiente angular neste gráfico representa o quanto as comparações de chaves aumentam, quando variamos em 1 unidade a quantidade de entradas, ou seja, se aumentarmos a quantidade de entradas em 1, o número de comparações aumentará 27,577 para os vetores. Logo, o melhor método, em relação as comparações, é o que possui o menor coeficiente angular, portanto, utilizar listas é mais vantajoso se estivermos preocupados com número de comparações.



Com os dados obtidos da regressão, sabemos que o coeficiente angular da reta que representa o vetor é 4,7768 e o da lista é 10,921. O coeficiente angular neste gráfico representa o quanto o número de copias aumenta, quando variamos em 1 unidade a quantidade de entradas, ou seja, se aumentarmos a quantidade de entradas em 1, o número de copias aumentará 4,7768 para os vetores. Logo, o melhor método, em relação as copias, é o que possui o menor coeficiente angular, portanto, utilizar vetores é mais vantajoso se estivermos preocupados com número de copias.

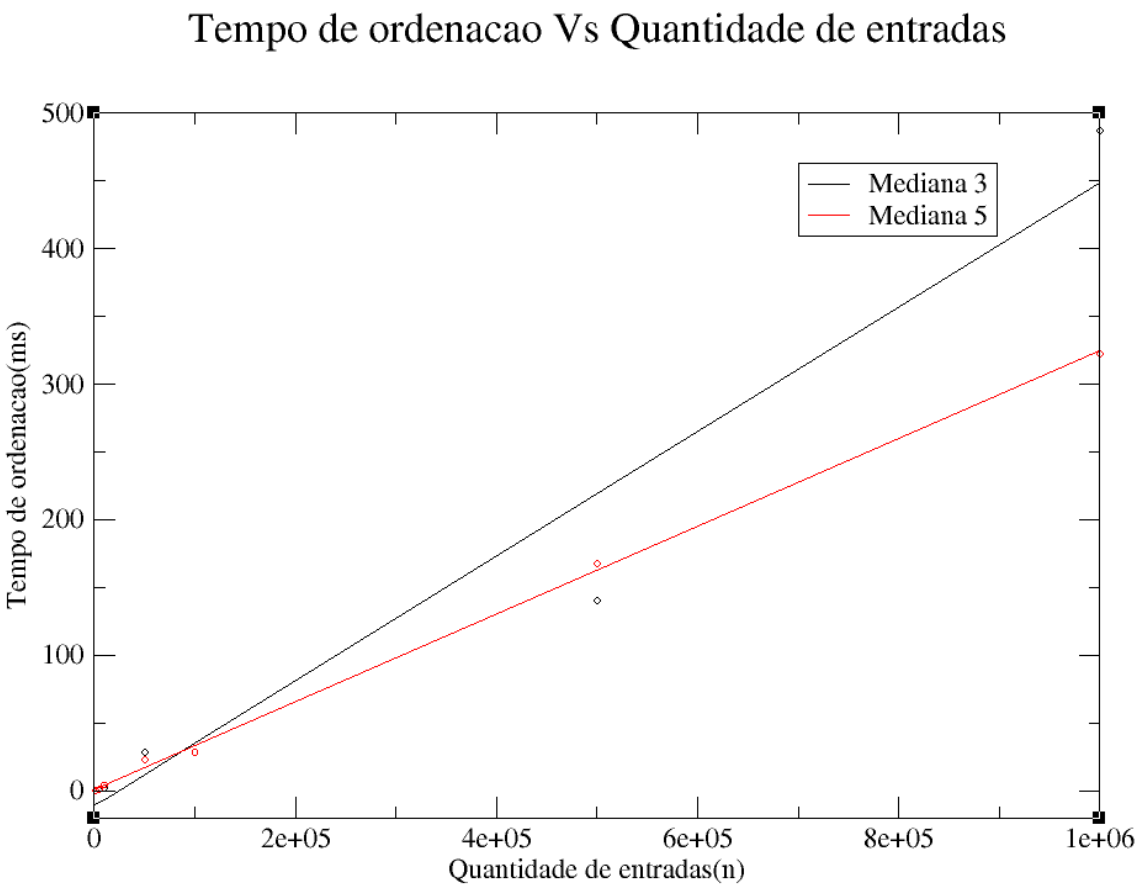
A partir dos resultados apresentados, pode-se concluir que não existe método mais eficiente em relação a todos as métricas analisadas. Por exemplo, se estivermos mais preocupados com o tempo de ordenação ou o número de cópias, opta-se pela implementação com vetores, entretanto, se estivermos mais preocupados com o número de comparações de chaves, pois pode ser o processo mais caro em determinada aplicação, deve-se optar pela implementação com listas.

Cenário 2: Impacto de variações do QuickSort.

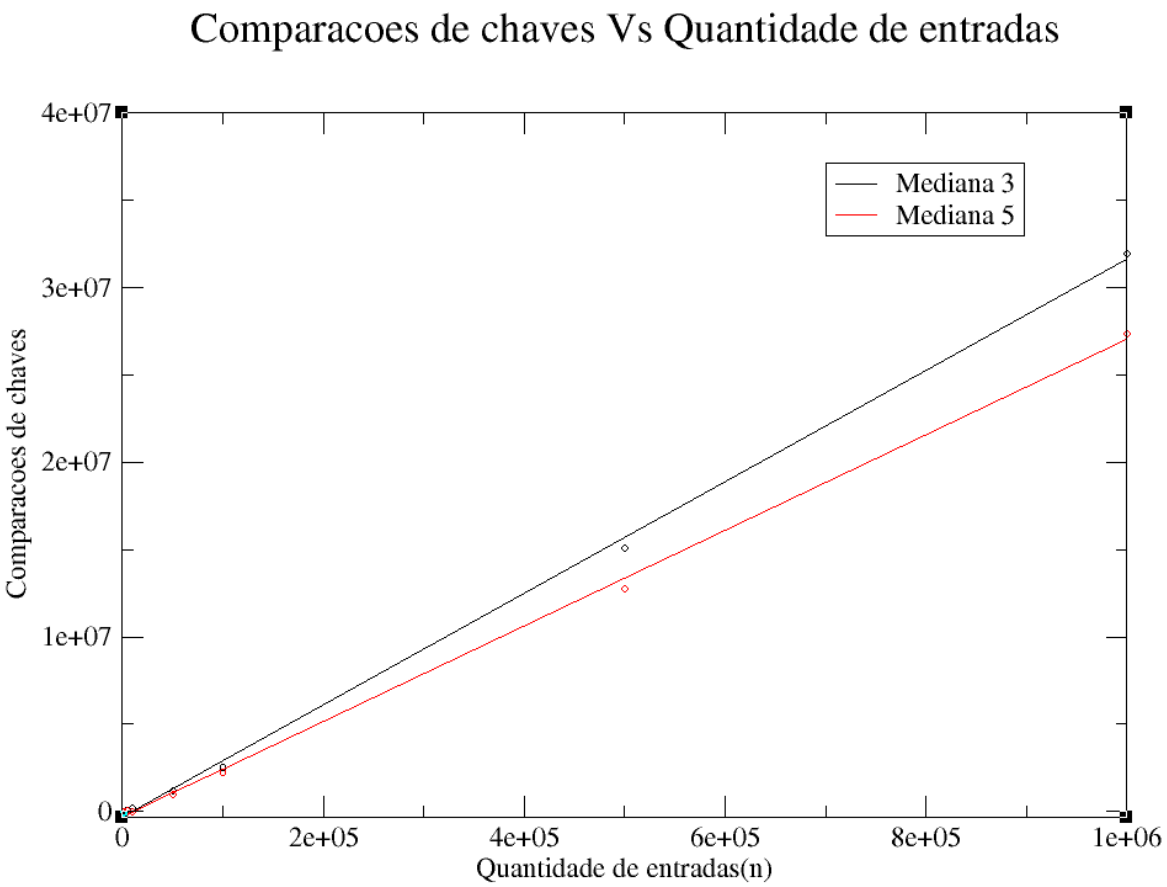
O algoritmo Quicksort(Mediana de 3 e 5) foi utilizado em entradas de tamanho N variados. Para cada valor de N, gerou-se 5 entradas diferentes, ao se usar sementes diferentes no gerador de números aleatórios. A partir desses 5 resultados gerados para cada valor de N, calculou-se a média deles.

Quantidade de entradas	Tempo médio mediana 3(ms)	Comparações mediana 3	Cópias mediana 3	Tempo médio mediana 5(ms)	Comparações mediana 5	Cópias mediana 5
1000	0	13715.4	3015.2	0	11758	2762.6
5000	1	92626	17830.4	1	80156.8	16631.4
10000	2	196684.6	36418.4	4.2	173116.6	35092.8
50000	28.2	1206308.4	206842.2	22.6	993927	197055
100000	28.6	2509571	427637.4	28	2247049.8	412652.8
500000	140.6	15076089.4	2313248.2	167.8	12780278.4	2249346.4
1000000	487.2	31939125.8	4745383.2	322	27345336	4657684.8

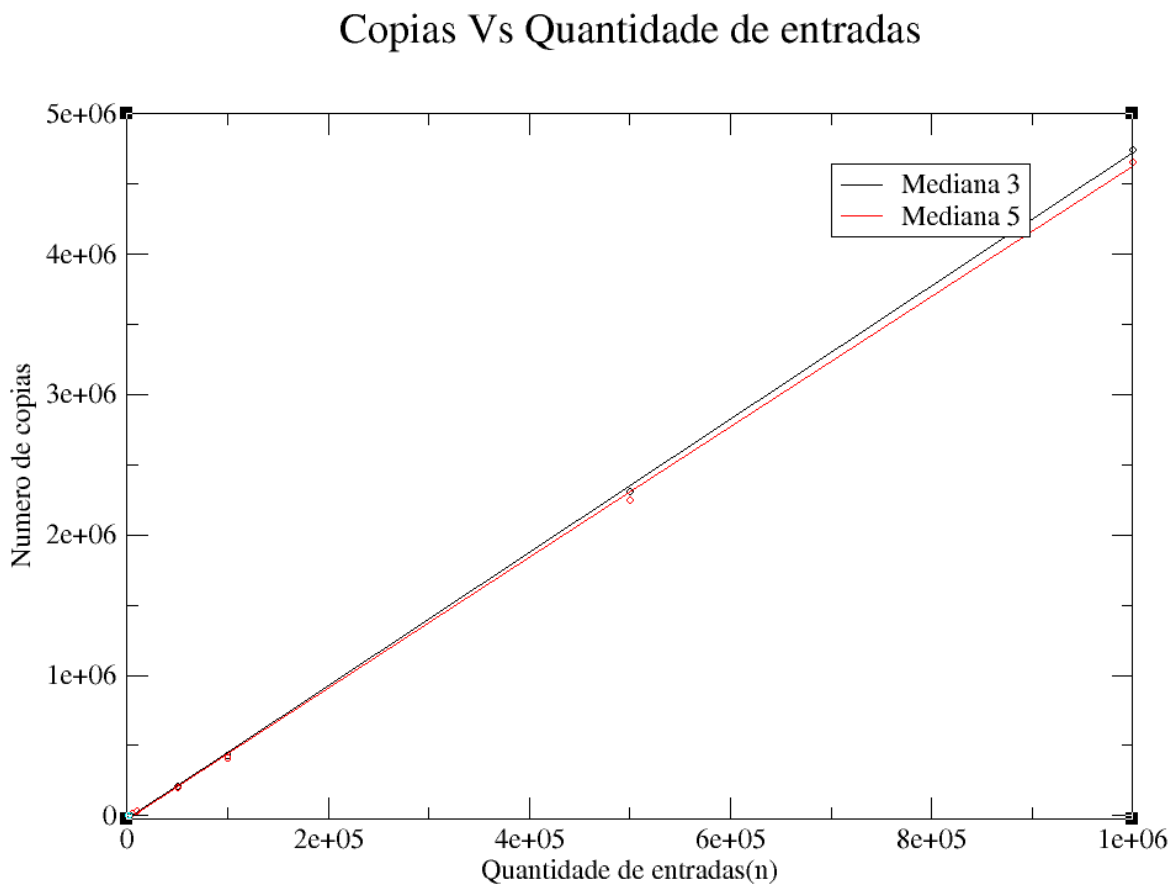
Plotou-se gráficos para se realizar uma análise mais precisa ao se comparar os dois métodos.



Com os dados obtidos da regressão, sabemos que o coeficiente angular da reta que representa a mediana de 3 é 0,00045969 e da mediana de 5 é 0,00032331. O coeficiente angular neste gráfico representa o quanto o tempo de ordenação varia, quando variamos em 1 unidade a quantidade de entradas, ou seja, se aumentarmos a quantidade de entradas em 1, o tempo de ordenação da mediana de 3 aumentará em 0,00045969 ms. Logo, o melhor método, em relação ao tempo, é o que possui o menor coeficiente angular, portanto, utilizar a mediana de 5 para a ordenação é mais vantajoso se estivermos preocupados com o tempo de ordenação.



Com os dados obtidos da regressão, sabemos que o coeficiente angular da reta que representa a mediana de 3 é 31,918 e da mediana de 5 é 27,321. O coeficiente angular neste gráfico representa o quanto as comparações de chaves aumentam, quando variamos em 1 unidade a quantidade de entradas, ou seja, se aumentarmos a quantidade de entradas em 1, o número de comparações aumentará 31,918 para a mediana de 3. Logo, o melhor método, em relação as comparações, é o que possui o menor coeficiente angular, portanto, utilizar a mediana de 5 é mais vantajoso se estivermos preocupados com número de comparações.



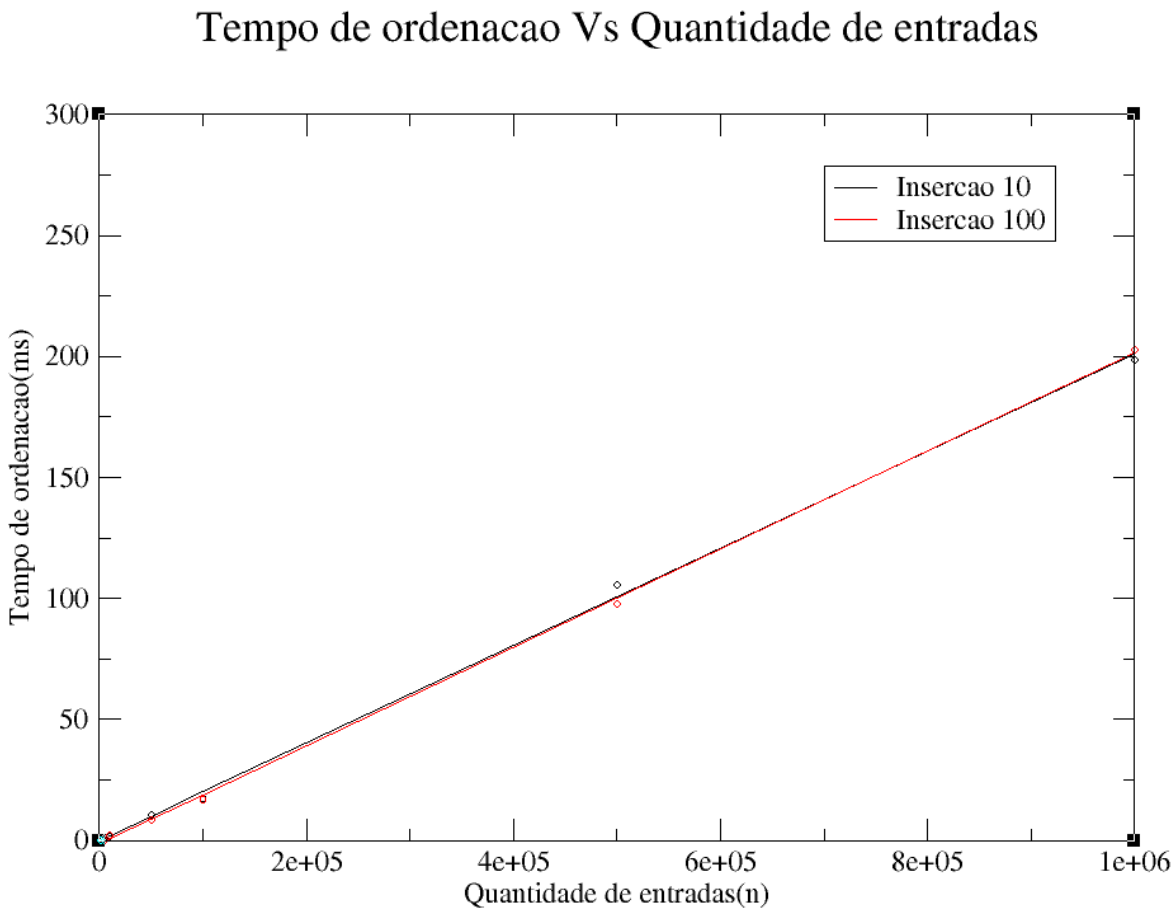
Com os dados obtidos da regressão, sabemos que o coeficiente angular da reta que representa a mediana de 3 é 4,7764 e da mediana de 5 é 4,6552. O coeficiente angular neste gráfico representa o quanto o número de copias aumenta, quando variamos em 1 unidade a quantidade de entradas, ou seja, se aumentarmos a quantidade de entradas em 1, o número de copias aumentará 4,7764 para a mediana de 3. Logo, o melhor método, em relação as copias, é o que possui o menor coeficiente angular, portanto, utilizar a mediana de 5 é mais vantajoso se estivermos preocupados com número de copias.

A partir da análise acima, pode-se concluir que a mediana de 5 é sempre mais eficiente que a mediana de 3 em relação as 3 métricas analisadas. Isso ocorre, pois com a mediana de 5, é escolhido um pivô melhor, que melhor divide o vetor, maximizando o QuickSort.

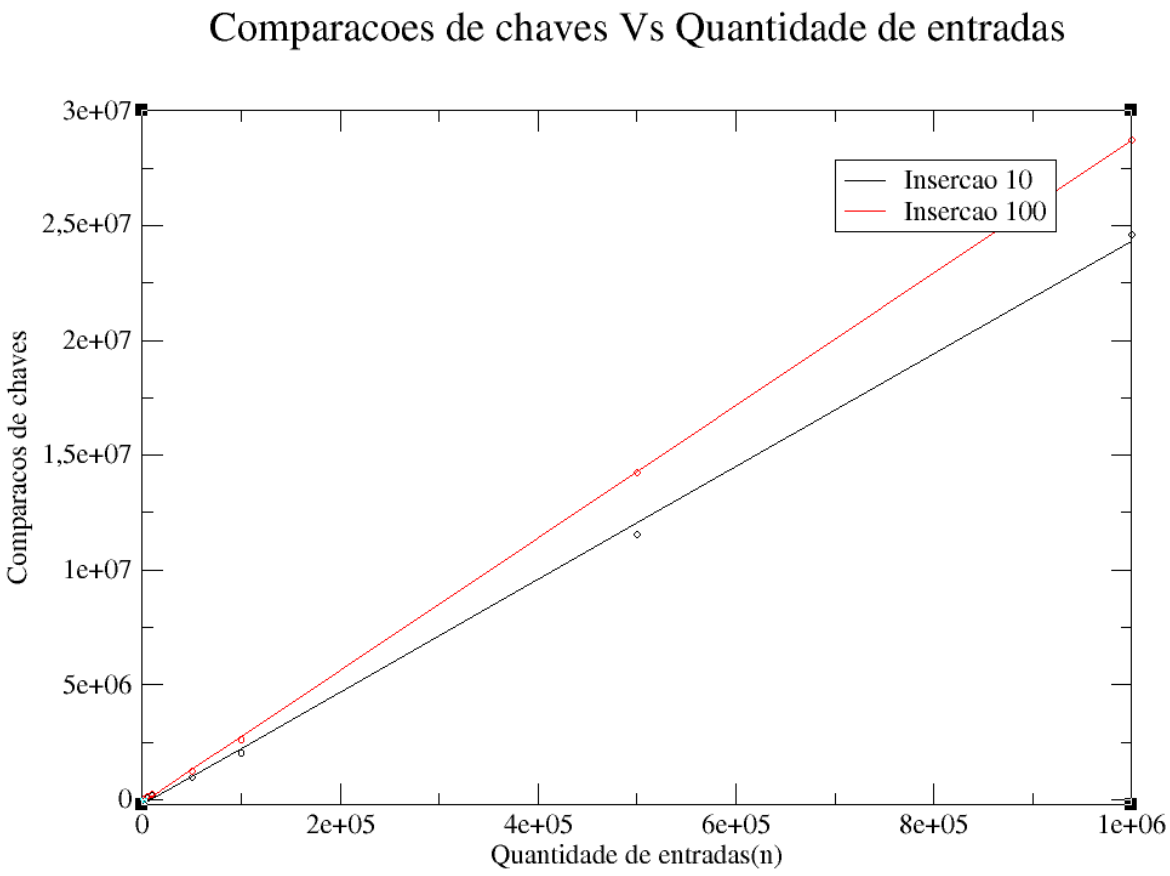
O algoritmo Quicksort(Inserção 10 e 100) foi utilizado em entradas de tamanho N variados. Para cada valor de N, gerou-se 5 entradas diferentes, ao se usar sementes diferentes no gerador de números aleatórios. A partir desses 5 resultados gerados para cada valor de N, calculou-se a média deles.

Quantidade de entradas	Tempo médio inserção 10(ms)	Comparações inserção 10	Cópias inserção 10	Tempo médio inserção 100(ms)	Comparações inserção 100	Cópias inserção 100
1000	0	11621.4	2638.4	0	17441	2089.4
5000	1	71575.6	15968.6	0	102025.4	13137.2
10000	1.4	155567.2	34203.6	2	215156.4	28612.2
50000	10.6	953334	197216.6	8.6	1254802.2	169193
100000	17.2	2010798.4	417434.8	17	2607528.8	361745.2
500000	105.4	11519072.4	2328847.8	97.8	14247695	2076697.8
1000000	198.6	24609162.6	4790795.8	202.8	28722000.4	4389422

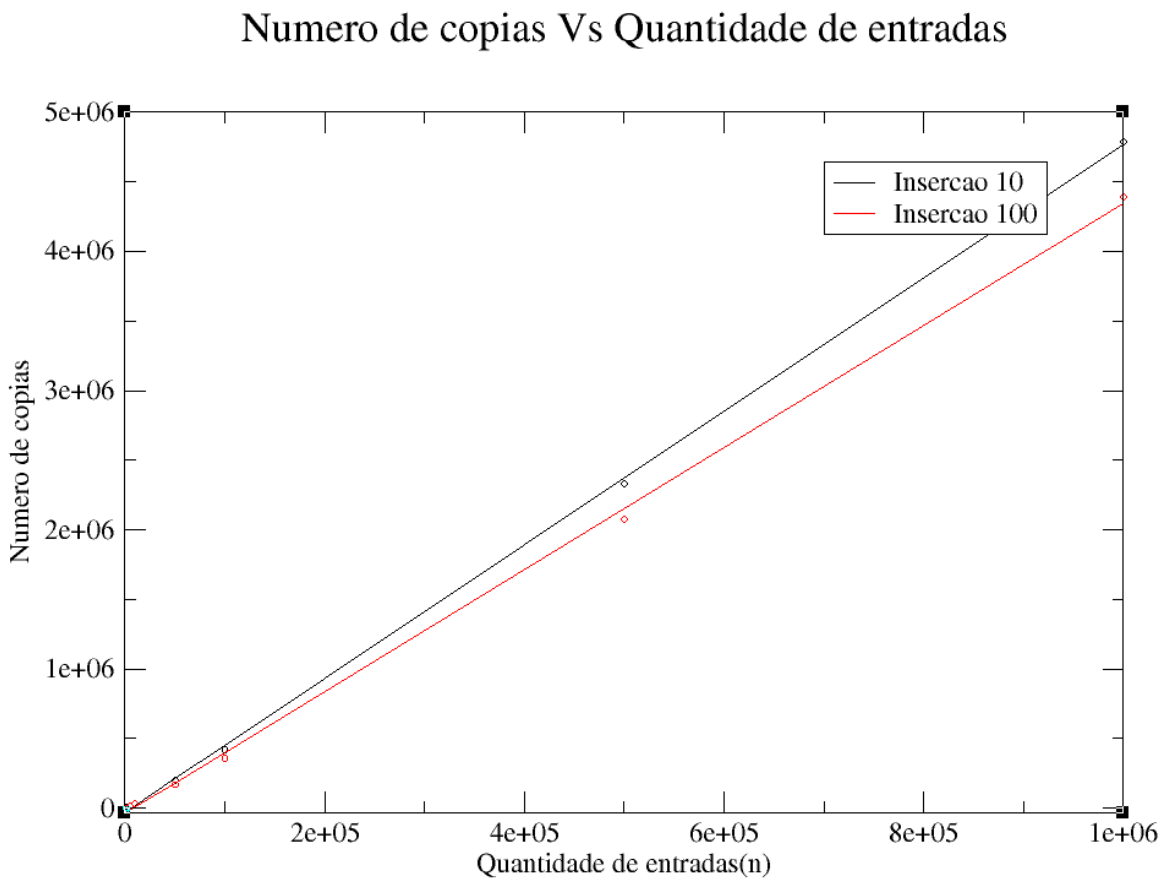
Plotou-se gráficos para se realizar uma análise mais precisa ao se comparar os dois métodos.



Com os dados obtidos da regressão, sabemos que o coeficiente angular da reta que representa a inserção de 10 é 0,00020093 e da inserção de 100 é 0,0002029. O coeficiente angular neste gráfico representa o quanto o tempo de ordenação varia, quando variamos em 1 unidade a quantidade de entradas, ou seja, se aumentarmos a quantidade de entradas em 1, o tempo de ordenação da inserção de 10 aumentará em 0,00020093 ms. Logo, o melhor método, em relação ao tempo, é o que possui o menor coeficiente angular, portanto, utilizar a inserção de 10 para a ordenação é mais vantajoso se estivermos preocupados com o tempo de ordenação. Note que os coeficientes são praticamente iguais e talvez esse análise de inserção de 10 ser, mesmo que pouco, melhor pode estar equivocada, pois utilizamos apenas 5 conjuntos de elementos para comparação.



Com os dados obtidos da regressão, sabemos que o coeficiente angular da reta que representa a inserção de 10 é 24,545 e da inserção de 100 é 28,802. O coeficiente angular neste gráfico representa o quanto as comparações de chaves aumentam, quando variamos em 1 unidade a quantidade de entradas, ou seja, se aumentarmos a quantidade de entradas em 1, o número de comparações aumentará 24,545 para a inserção de 10. Logo, o melhor método, em relação as comparações, é o que possui o menor coeficiente angular, portanto, utilizar a inserção de 10 é mais vantajoso se estivermos preocupados com número de comparações.

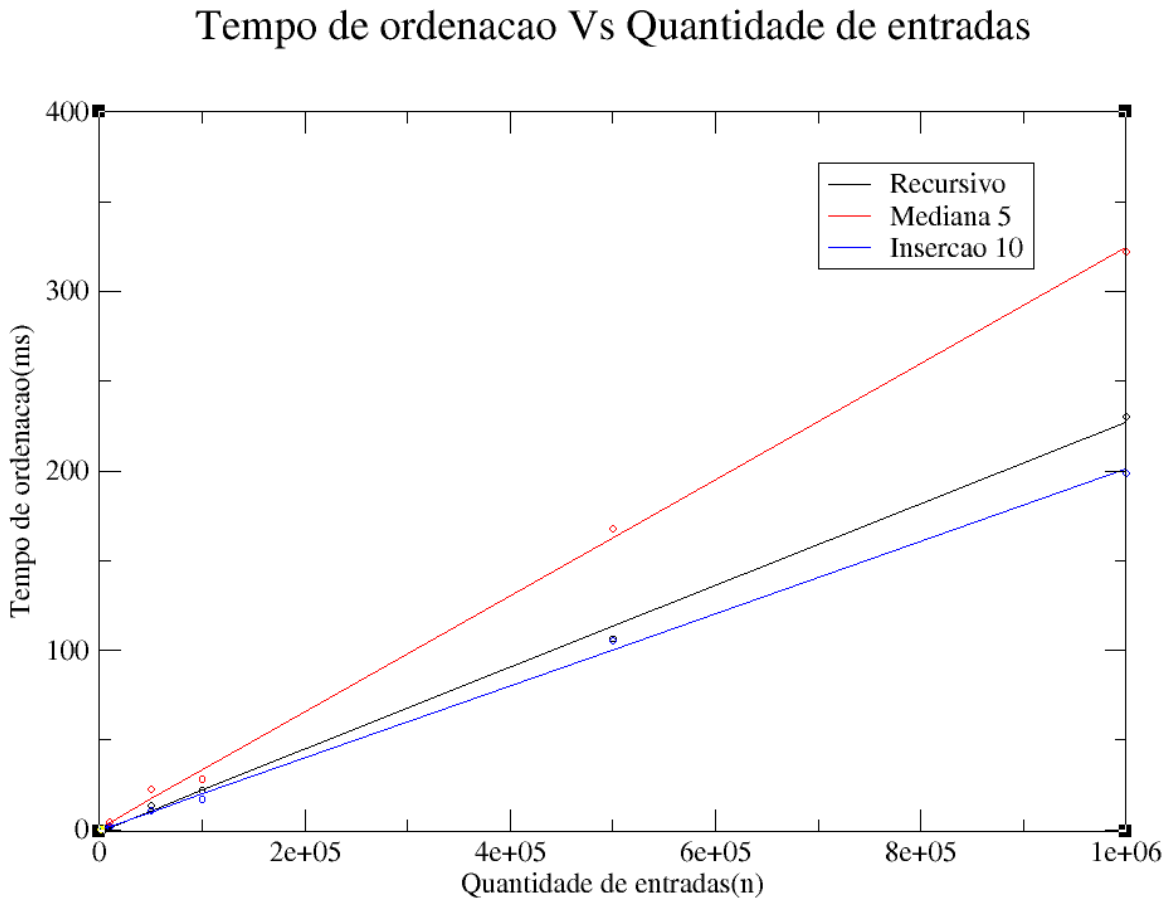


Com os dados obtidos da regressão, sabemos que o coeficiente angular da reta que representa a inserção de 10 é 4,796 e da inserção de 100 é 4,384. O coeficiente angular neste gráfico representa o quanto o número de copias aumenta, quando variamos em 1 unidade a quantidade de entradas, ou seja, se aumentarmos a quantidade de entradas em 1, o número de copias aumentará 4,796 para a inserção de 10. Logo, o melhor método, em relação as copias, é o que possui o menor coeficiente angular, portanto, utilizar a inserção de 100 é mais vantajoso se estivermos preocupados com número de copias.

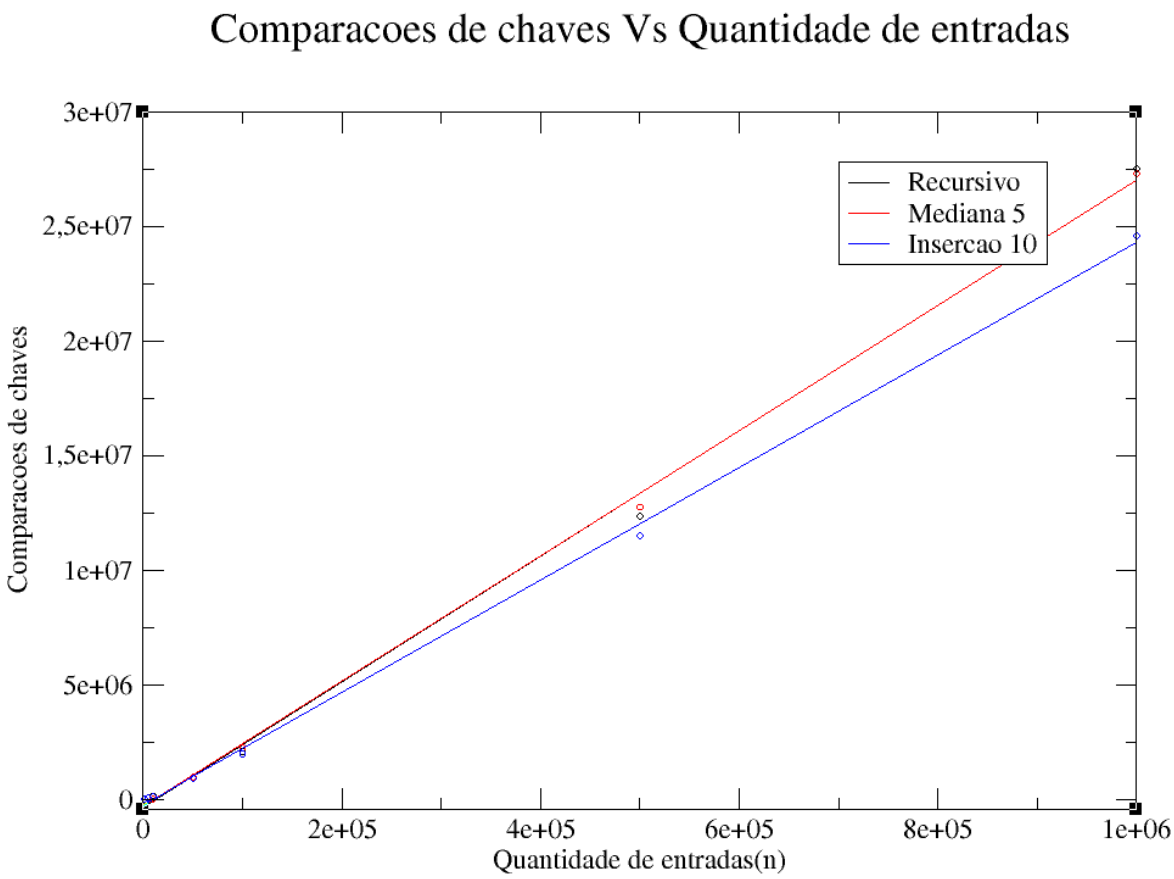
A partir dos resultados apresentados, pode-se concluir que não existe método mais eficiente em relação a todos as métricas analisadas. Por exemplo, se estivermos mais preocupados com o tempo de ordenação ou o número de comparações, opta-se pela implementação com inserção de 10, entretanto, se estivermos mais preocupados com o número de cópias, já que pode ser o processo mais caro em determinada aplicação, deve-se optar pela implementação com inserção de 100.

Por fim, comparou-se o QuickSort mediana de 5, QuickSort Inserção de 10 e o QuickSort Recursivo, já que esses algoritmos se destacaram

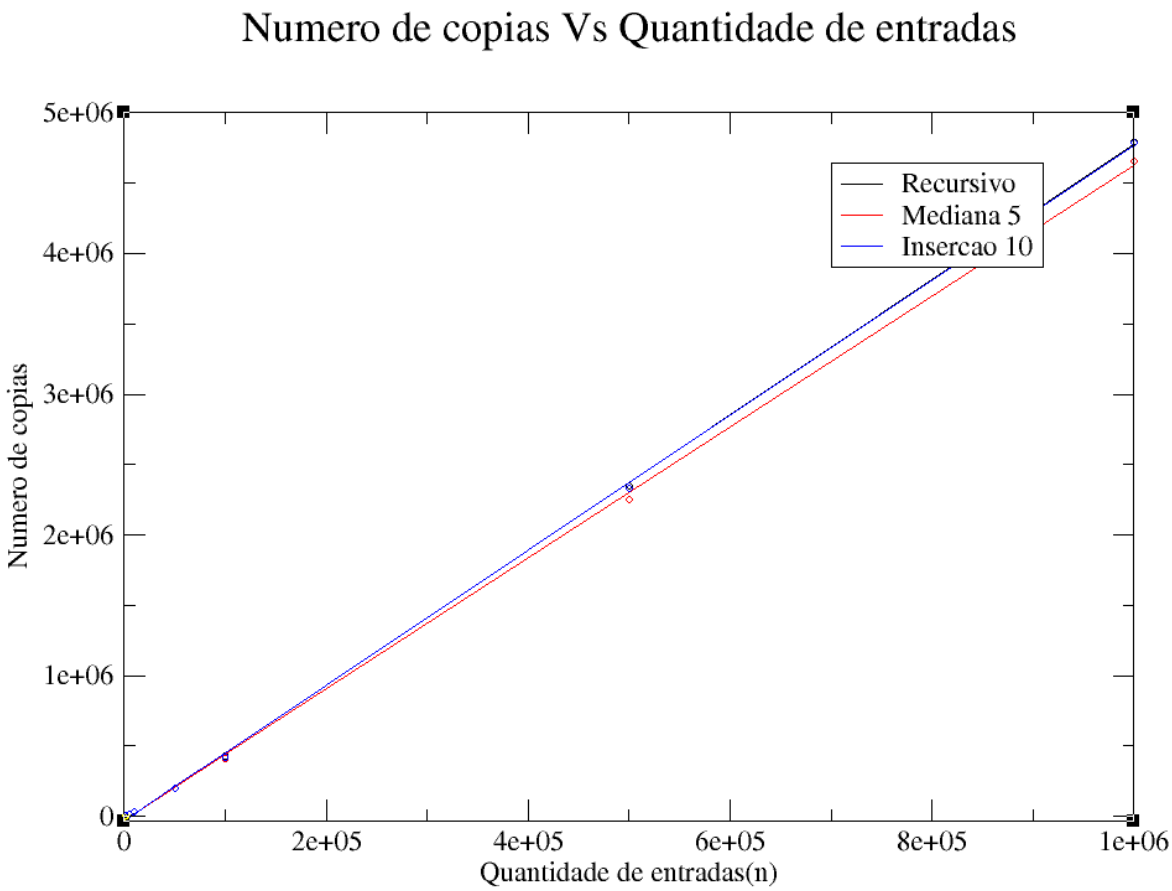
num balanço final das métricas. Plotou-se gráficos para se realizar uma análise mais precisa ao se comparar os três métodos.



Com os dados obtidos da regressão, sabemos que o coeficiente angular da reta que representa o recursivo é 0,00021792, da mediana de 5 é 0,00032331 e da inserção de 10 é 0,00020093. O coeficiente angular neste gráfico representa o quanto o tempo de ordenação varia, quando variamos em 1 unidade a quantidade de entradas, ou seja, se aumentarmos a quantidade de entradas em 1, o tempo de ordenação da inserção de 10 aumentará em 0,00020093 ms. Logo, o melhor método, em relação ao tempo, é o que possui o menor coeficiente angular, portanto, utilizar a inserção de 10 para a ordenação é mais vantajoso se estivermos preocupados com o tempo de ordenação.



Com os dados obtidos da regressão, sabemos que o coeficiente angular da reta que representa o recursivo é 27,381, da mediana de 5 é 27,321 e da inserção de 10 é 24,545. O coeficiente angular neste gráfico representa o quanto as comparações de chaves aumentam, quando variamos em 1 unidade a quantidade de entradas, ou seja, se aumentarmos a quantidade de entradas em 1, o número de comparações aumentará 24,545 para a inserção de 10. Logo, o melhor método, em relação as comparações, é o que possui o menor coeficiente angular, portanto, utilizar a inserção de 10 é mais vantajoso se estivermos preocupados com número de comparações.



Com os dados obtidos da regressão, sabemos que o coeficiente angular da reta que representa o recursivo é 4,7997, da mediana de 5 é 4,6552 e da inserção de 10 é 4,796. O coeficiente angular neste gráfico representa quanto o número de copias aumenta, quando variamos em 1 unidade a quantidade de entradas, ou seja, se aumentarmos a quantidade de entradas em 1, o número de copias aumentará 4,796 para a inserção de 10. Logo, o melhor método, em relação as copias, é o que possui o menor coeficiente angular, portanto, utilizar a mediana de 5 é mais vantajoso se estivermos preocupados com número de cópias.

Conclusão

Após a análise do resultados obtidos, não é possível eleger um método absoluto em relação as três métricas analisadas. Dependendo da situação, um método pode vir a ser melhor que outro, pois cada um possui suas vantagens e desvantagens. Por isso, é necessário saber quais operações são custosas para a aplicação e, assim, escolher uma estrutura de dados adequada e uma variação do algoritmo de ordenação vantajosa.