Universidade Federal do Rio Grande do Norte
Centro de Ensino Superior do Seridó
Departamento de Computação e Tecnologia
Curso de Bacharelado em Sistemas de Informação

Relatório I

Jonathan Tauan Pereira Maia

Caicó-RN
Junho, 2022

Jonathan Tauan Pereira Maia

Relatório I

Trabalho apresentado à disciplina de Estrutura de Dados do Departamento de Computação e Tecnologia da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção da nota da primeira unidade.

Orientador

Prof. Dr. João Paulo de Souza Medeiros
Universidade Federal do Rio Grande do Norte - UFRN

BSI – Bacharelado em Sistemas de Informação
DCT – Departamento de Computação e Tecnologia
CERES – Centro de Ensino Superior do Seridó
UFRN – Universidade Federal do Rio Grande do Norte

Caicó-RN

Junho, 2022

Relatório I

Autor: Jonathan Tauan Pereira Maia
Orientador(a): Prof. Dr. João Paulo de Souza Medeiros

Resumo

Este trabalho tem como objetivo objetivo apresentar um relatório sobre os algoritmos Insertion-Sort, Merge-Sort e Quick-sort, que são utilizados para resolver problemas de ordenação em um conjunto de dados linear que possa ser ordenado. Os algoritmos serão aqui analisados em forma de gráficos que representam o tempo de execução de seus casos em função do tamanho da entrada, juntamente com uma análise analítica sobre cada algoritmo e análise de desempenho em relação ao custo de tempo e memória, além disso, será feita uma comparação entre os mesmos, visando entender melhor suas particularidades e casos mais performáticos.

Palavras-chave: Algoritmo, Complexidade, Ordenação, Estrutura de Dados.

Report I

Author: Jonathan Tauan Pereira Maia
Supervisor: João Paulo de Souza Medeiros, Ph.D.

Abstract

This work aims to present a report on the Insertion-Sort, Merge-Sort and Quick-sort algorithms, which are used to solve sorting problems in a linear dataset that can be sorted. The algorithms will be analyzed here in the form of graphs that represent the execution time of their cases as a function of the size of the input, together with an analytical analysis on each algorithm and performance analysis in relation to the cost of time and memory, in addition, it will be A comparison was made between them, in order to better understand their particularities and more performative cases.

Keywords: Algorithm, Complexity, Sorting, Data Structure.

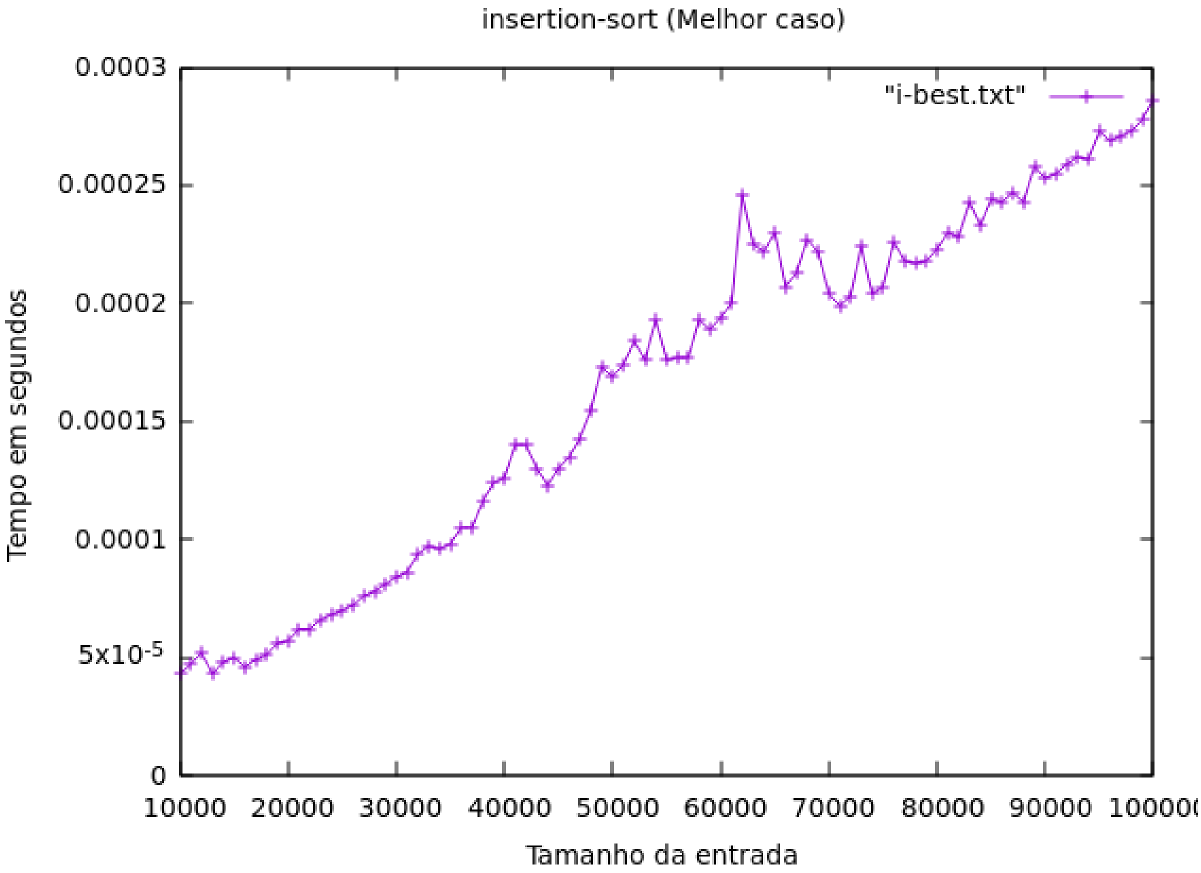
Sumário

1 Insertion Sort

1.1 Gráficos

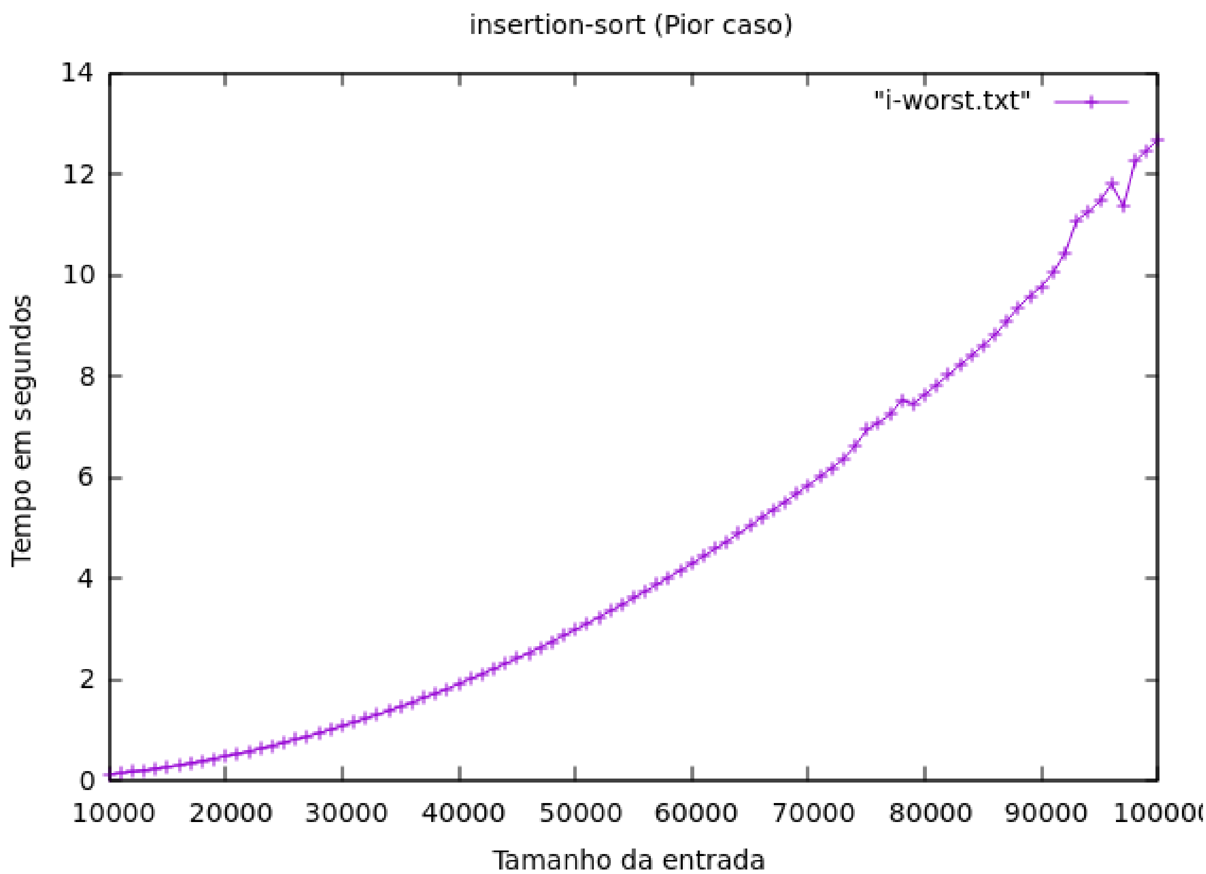
1.1.1 Melhor caso

O gráfico abaixo representa o tempo de execução do melhor caso do insertion-sort em função do tamanho da entrada. Como entrada para gerar o gráfico, foram utilizados 91 vetores já ordenados. Pela análise do gráfico podemos notar que o algoritmo no melhor caso, desconsiderando a variação de tempo gerada por processos concorrentes no momento da execução do programa, é linear. Tb(n) pertence a O(n).



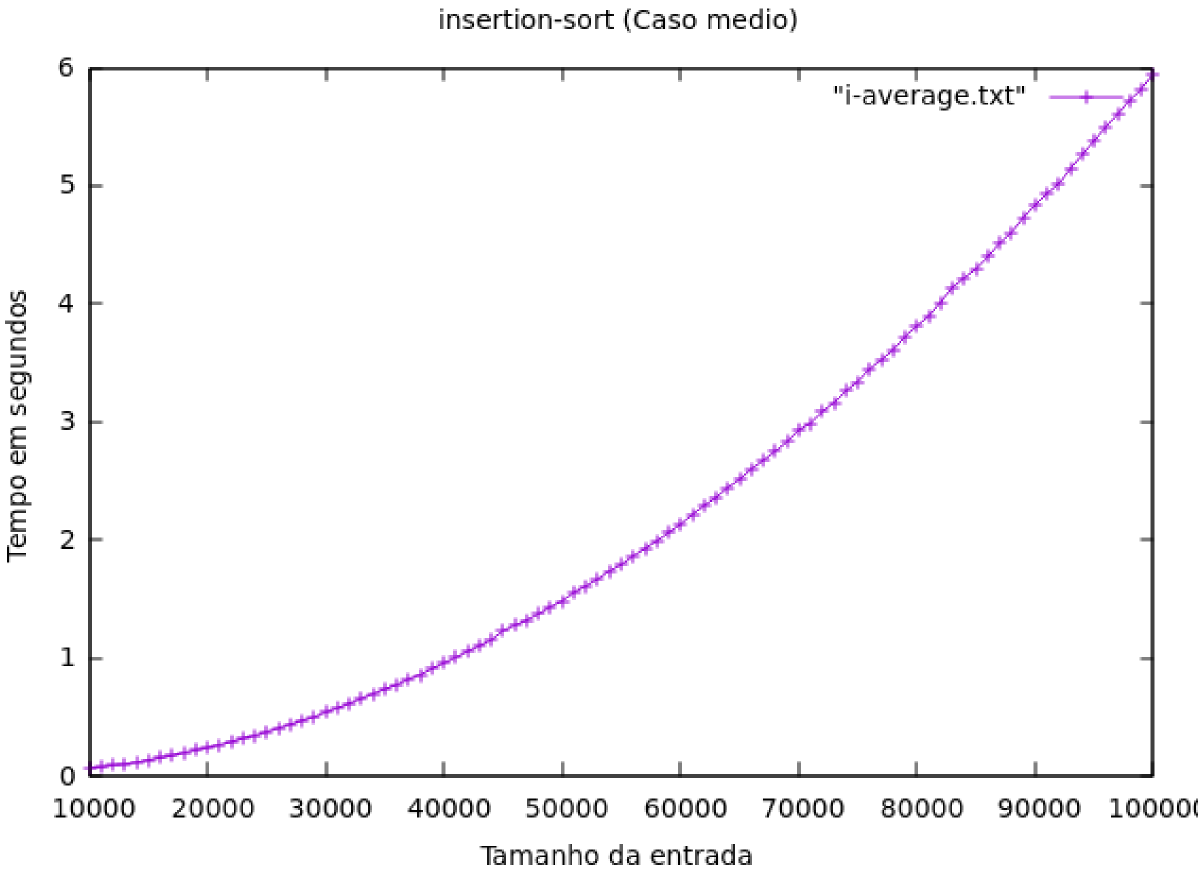
1.1.2 Pior caso

O gráfico abaixo representa o tempo de execução do pior caso do insertion-sort em função do tamanho da entrada. Como entrada para gerar o gráfico, foram utilizados 91 vetores ordenados em ordem decrescente. Pela análise do gráfico podemos notar que o algoritmo no pior caso, desconsiderando a variação de tempo gerada por processos concorrentes no momento da execução do programa, é quadrático. Tw(n) pertence a $O(n^2)$.



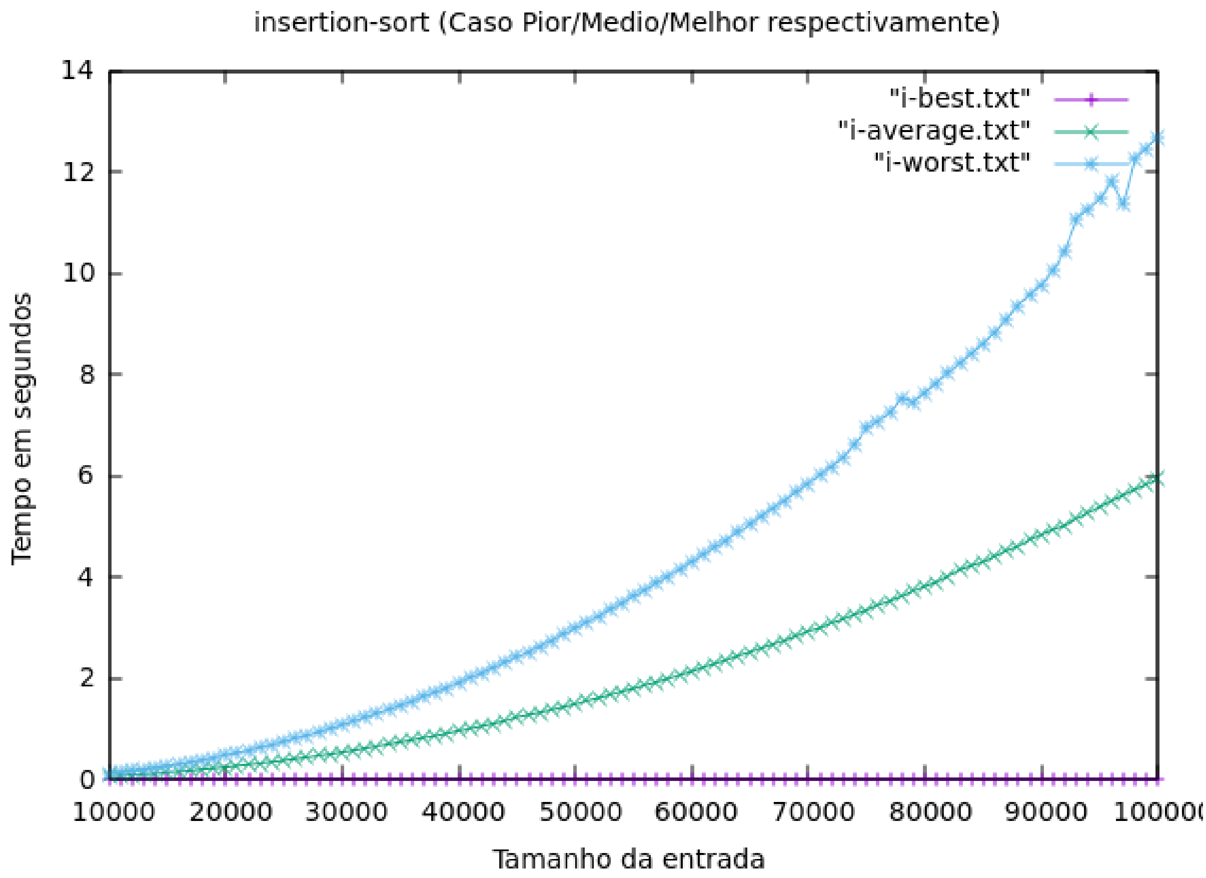
1.1.3 Caso médio

O gráfico abaixo representa o tempo de execução esperado do insertion-sort em função do tamanho da entrada. Como entrada para gerar o gráfico, foram utilizados 91 vetores de tamanho n, preenchidos com números de 0 até n gerados aleatoriamente. Pela análise do gráfico podemos notar que o algoritmo no caso médio, desconsiderando a variação de tempo gerada por processos concorrentes no momento da execução do programa, é quadrático. Ta(n) pertence a $O(n^2)$.



1.1.4 Comparação dos gráficos

No gráfico abaixo podemos ver a comparação entre o caso médio, o melhor e o pior caso do algoritmo.



1.2 Análise analítica do tempo de execução

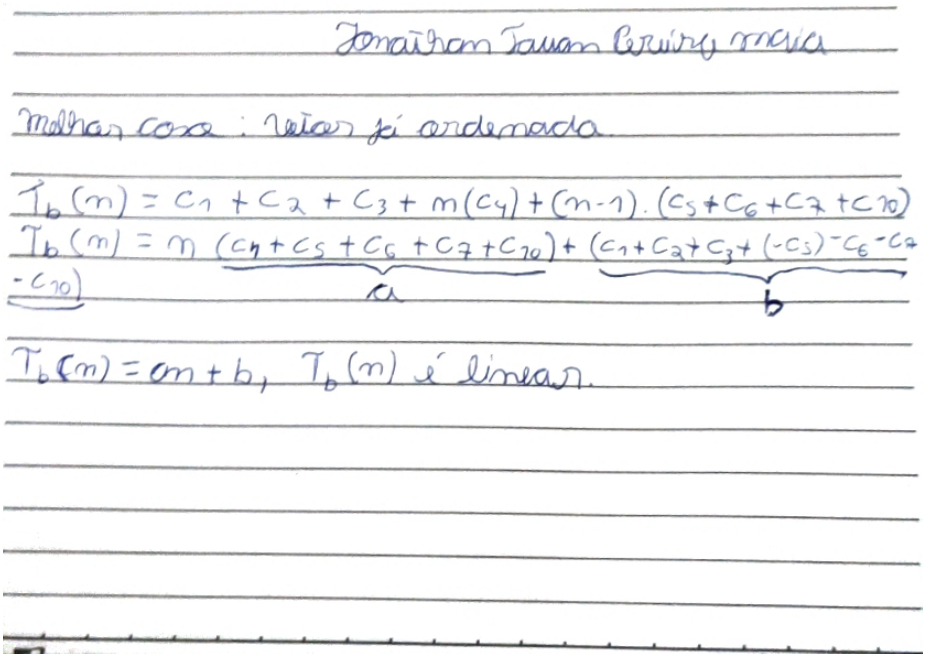
1.2.1 Algoritmo

```
void insertion_sort(int*v, unsigned int n){
    long int i, j, temp;
    for(i=1; i<=n; i++){
        temp = v[i];
        j = i-1;
        while (j>=0 && v[j] > temp){
            v[j+1] = v[j];
            j = j-1;
        }v[j+1] = temp
    }
```

Jonathan Tauan Pereira Maia

1.2.1.1 Melhor caso

A análise analítica do tempo de execução do insertion-sort no melhor caso, que se dá quando o vetor de entrada já está ordenado, permitiu perceber novamente que este é da forma linear, f(n) = an + b.



Jonathan Tauan Pereira Maia

Melhor caso: vetor já ordenado.

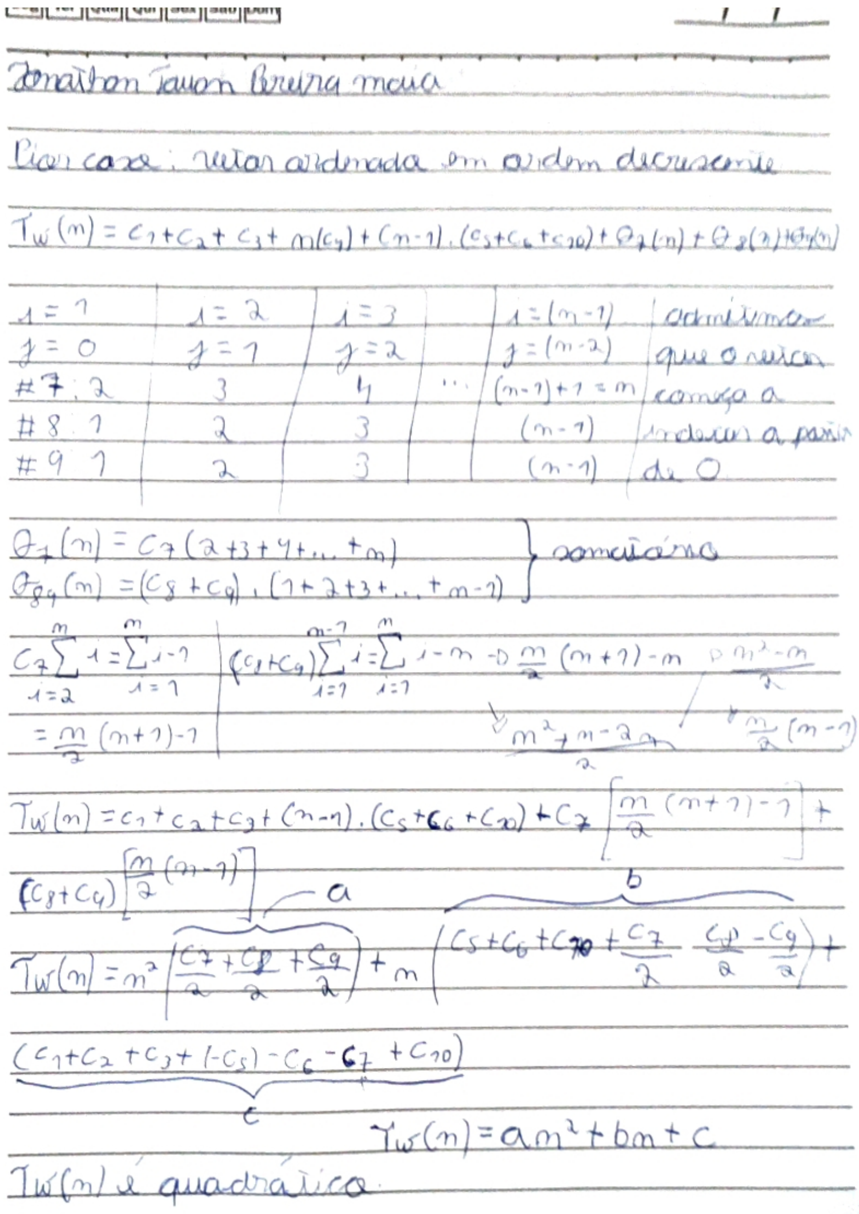
$$T_b(n) = c_1 + c_2 + c_3 + n(c_4) + (n-1).(c_5+c_6+c_7+c_{10})$$

$$T_b(n) = n\underbrace{(c_4+c_5+c_6+c_7+c_{10})}_{a} + \underbrace{(c_1+c_2+c_3+(-c_5)-c_6-c_7-c_{10})}_{b}$$

$T_b(n) = an+b$, $T_b(n)$ é linear.

1.2.1.2 Pior caso

A análise analítica do tempo de execução do insertion-sort no pior caso, que se dá quando o vetor de entrada está ordenado em ordem decrescente, permitiu perceber também que este é da forma quadrática, f(n) = an^2 + bn + c.



Jonathan Tauan Pereira maia

Pior caso: vetor ordenado em ordem decrescente

$T_w(n) = c_1 + c_2 + c_3 + n(c_4) + (n-1).(c_5 + c_6 + c_{10}) + \theta_7(n) + \theta_8(n) + \theta_9(n)$

$i = 1$	$i = 2$	$i = 3$		$i = (n-1)$	admitimos
$j = 0$	$j = 1$	$j = 2$		$j = (n-2)$	que o vetor
#7: 2	3	4	...	$(n-1)+1 = n$	começa a
#8: 1	2	3		$(n-1)$	ordenar a partir
#9: 1	2	3		$(n-1)$	de 0

$\left.\begin{array}{l}\theta_7(n) = c_7(2+3+4+\dots+n) \\ \theta_{89}(n) = (c_8+c_9).(1+2+3+\dots+n-1)\end{array}\right\}$ somatório

$c_7\sum_{i=2}^{n} i = \sum_{i=1}^{n} i - 1 = \frac{n}{2}(n+1) - 1$

$(c_8+c_9)\sum_{i=1}^{n-1} i = \sum_{i=1}^{n} i - n \to \frac{n}{2}(n+1) - n \to \frac{n^2-n}{2}$

$\frac{n^2+n-2n}{2}$ $\quad \frac{n}{2}(n-1)$

$T_w(n) = c_1 + c_2 + c_3 + (n-1).(c_5 + c_6 + c_{10}) + c_7\left[\frac{n}{2}(n+1) - 1\right] + (c_8+c_9)\left[\frac{n}{2}(n-1)\right]$

$T_w(n) = n^2\underbrace{\left(\frac{c_7}{2} + \frac{c_8}{2} + \frac{c_9}{2}\right)}_{a} + n\underbrace{\left(c_5 + c_6 + c_{10} + \frac{c_7}{2} - \frac{c_8}{2} - \frac{c_9}{2}\right)}_{b} + \underbrace{(c_1 + c_2 + c_3 + (-c_5) - c_6 - c_7 + c_{10})}_{c}$

$T_w(n) = an^2 + bn + c$

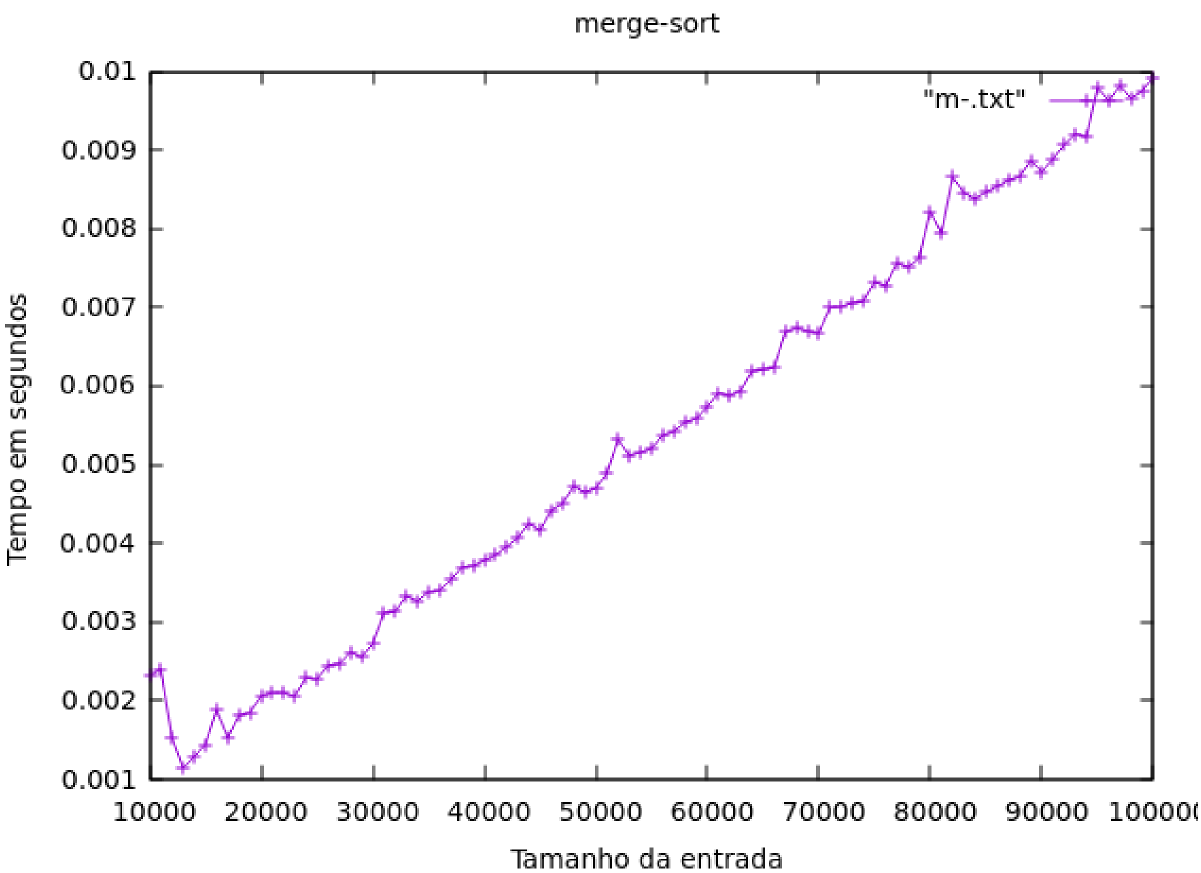
$T_w(n)$ é quadrático

2 Merge Sort

2.1 Gráfico

O gráfico abaixo representa o tempo de execução esperado do merge-sort em função do tamanho da entrada. Como entrada para gerar o gráfico, foram utilizados 91 vetores de tamanho n, preenchidos com números gerados aleatoriamente de 0 até n. Pela análise do gráfico podemos notar que o algoritmo, desconsiderando a variação de tempo gerada por processos concorrentes no momento da execução do programa, pertence a O(n*log(n)).

O merge-sort não apresenta distinção de tempo dependendo da entrada, portanto, seu tempo é consistente e não pode ser análisado por melhor, pior e médio caso.



2.2 Análise analítica do tempo de execução

2.2.1 Algoritmo

/ /

Seg Ter Qua Qui Sex Sáb Dom

```
void merge_sort(int *v, int r, int l) {
    int m;
    if (r < l) {
        m = floor((r+l)/2);
        merge_sort(v, r, m);
        merge_sort(v, m+1, l);
        merge(v, r, m, l);
    }
}
```

Jonathan Jauan Barbosa Maia

2.2.1.1 Tempo esperado

A análise analítica do tempo de execução do merge-sort permitiu perceber que em todos os casos o seu tempo de execução será sempre (n*log (n)).

Qua Qui Sex Sab Dom / /

O algoritmo merge-sort não tem melhor, pior e médio caso pois apresenta a mesma complexidade em todos os casos.

$T(n) = (c_1 + c_2) + T(n/2) + T(n/2) + c_3$

$T(n) = 2T(n/2) + n$

$T(n/2) = 2T(n/4) + n/2$

$T(n) = 2[2T(n/4) + n/2] + n$

$T(n) = 4T(n/4) + 2n$

$T(n/4) = 2T(n/8) + n/4$

$T(n) = 4[2T(n/8) + n/4] + 2n$

$T(n) = 8T(n/8) + 3n$

$T(n) = 2^x T(n/2^x) + x.n$

$T(n) = 2^{\log_2 n} T(1) + \log_2 n \cdot n$ (substituindo caso base)

$T(n) = n \log_2 n + n$

Caso base para $n = 1$,

$\frac{n}{2^x} = 1$

$2^x = n \rightarrow \log_2 2^x = \log_2 n \rightarrow x \cancel{\log_2 2} = \log_2 n$

$x = \log_2 n$

$T(n)$ é $n \log_2 n$

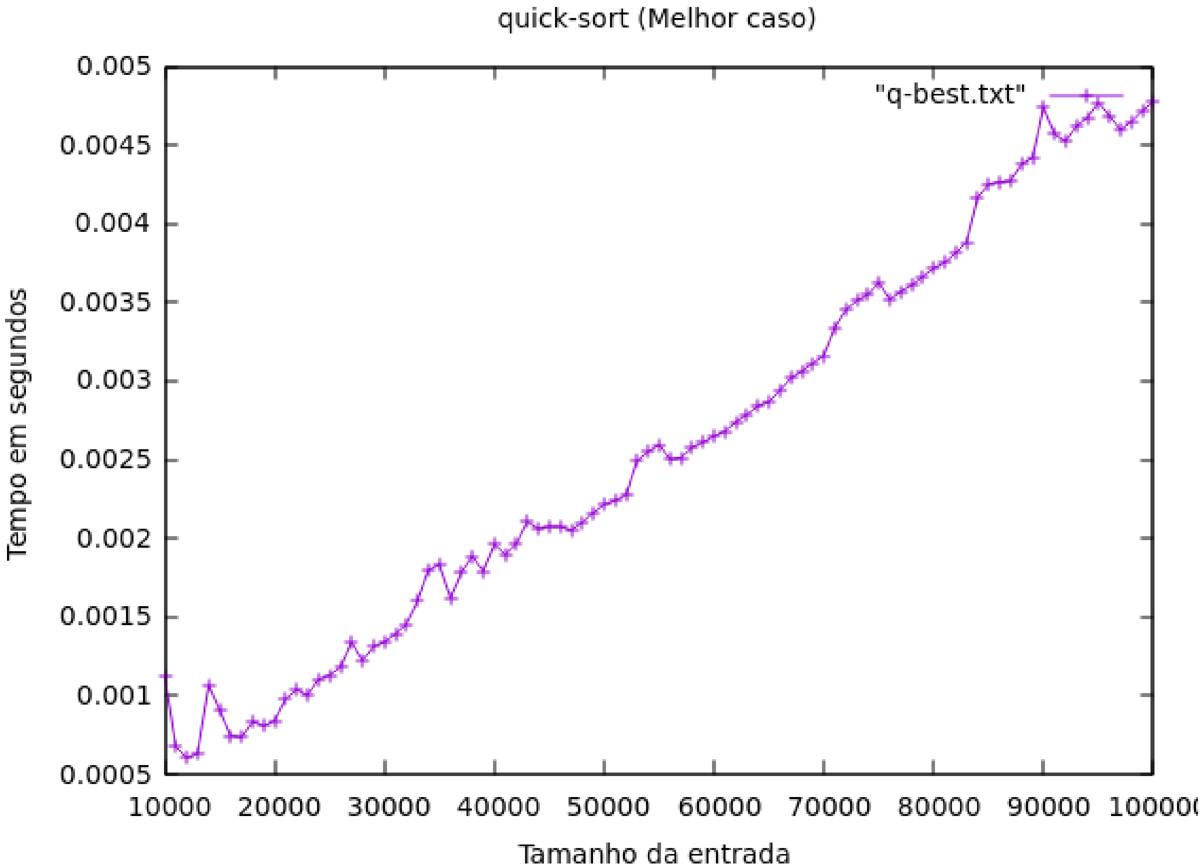
3 Quick Sort

A função partition necessária para o funcionamento do quick-sort foi implementada pegando como pivô o último elemento da entrada para todos os casos.

3.1 Gráficos

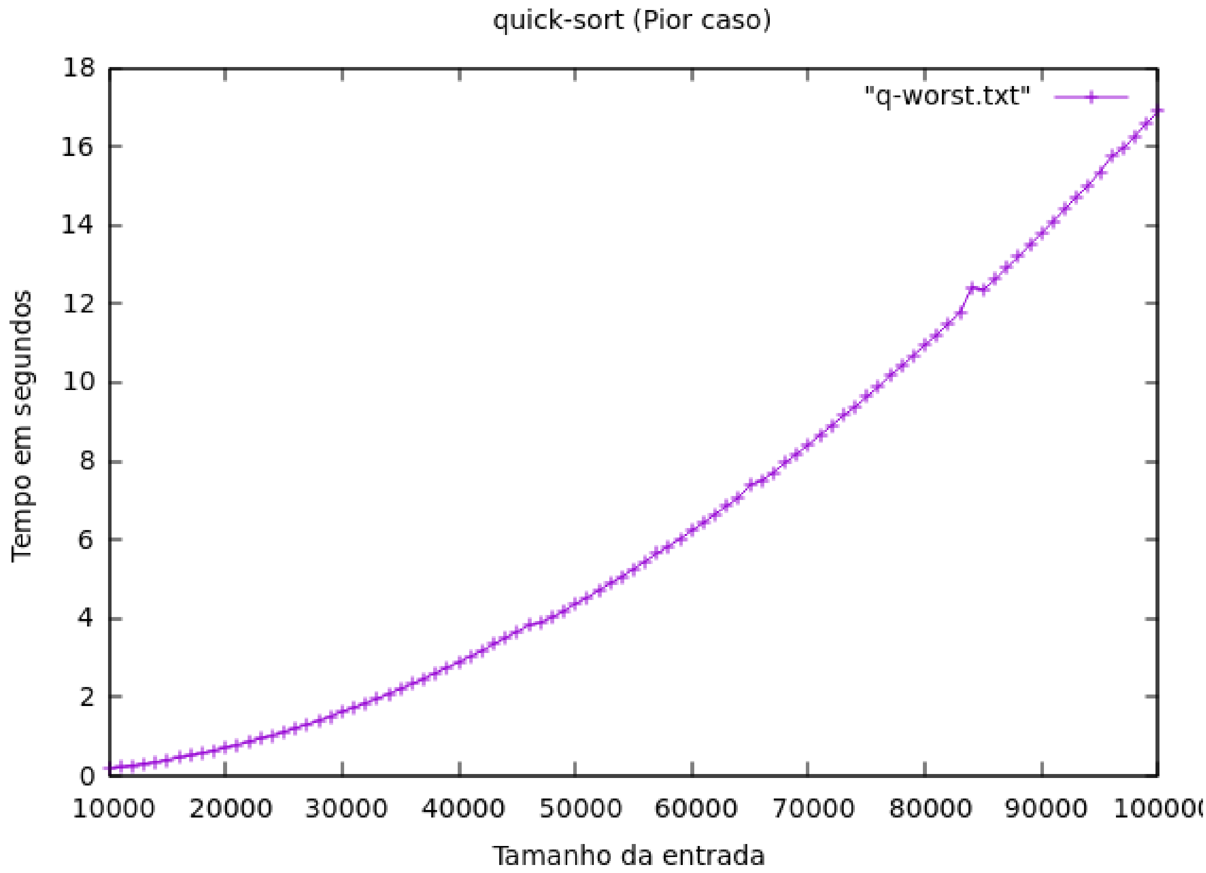
3.1.1 Melhor caso

O gráfico abaixo representa o tempo de execução do melhor caso do quick-sort em função do tamanho da entrada. Como entrada para gerar o gráfico, foram utilizados 91 vetores já ordenados, porém, com as posições do meio trocadas com a última posição para forçar o melhor caso. Pela análise do gráfico podemos notar que o algoritmo no melhor caso, desconsiderando a variação de tempo gerada por processos concorrentes no momento da execução do programa, pertence a O(n*log(n)).



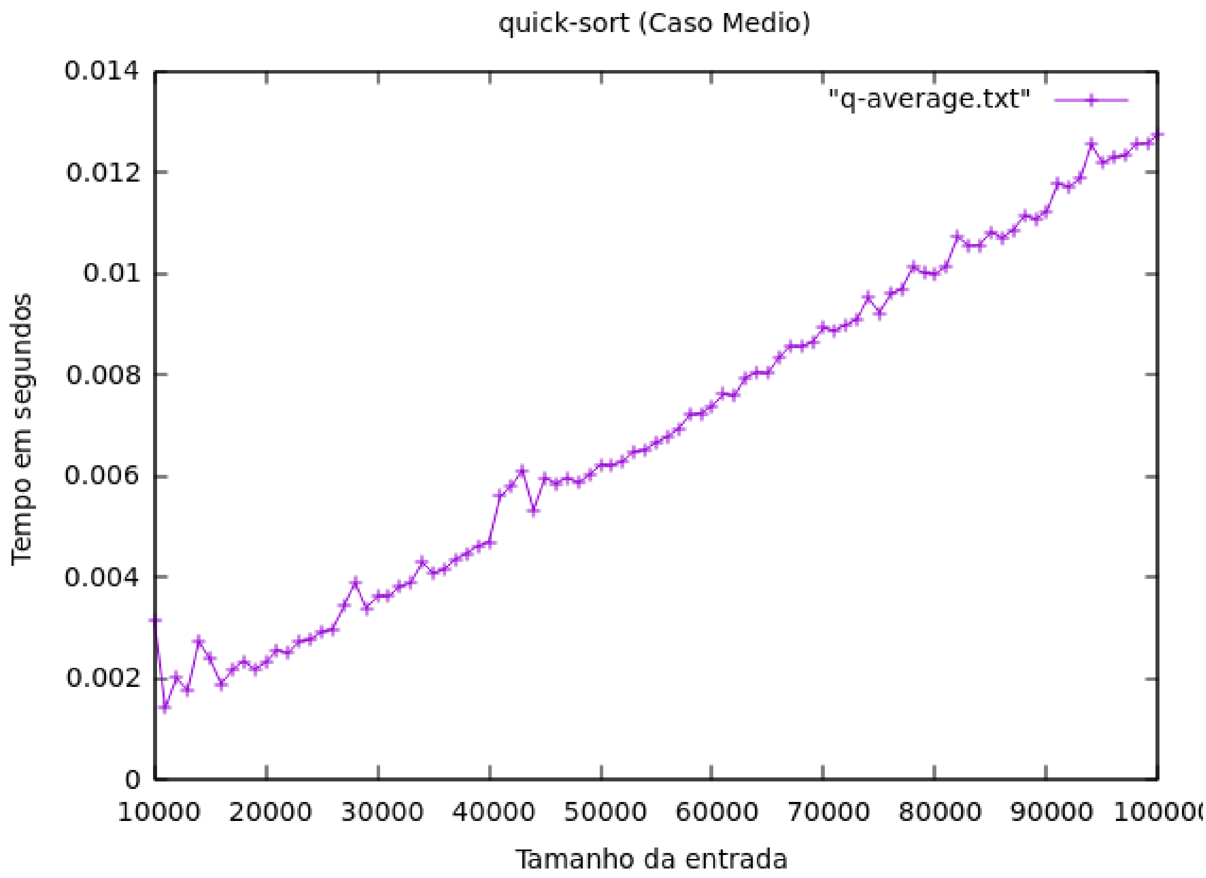
3.1.2 Pior caso

O gráfico abaixo representa o tempo de execução do pior caso do quick-sort em função do tamanho da entrada. Como entrada para gerar o gráfico, foram utilizados 91 vetores já ordenados em ordem crescente. Pela análise do gráfico podemos notar que o algoritmo no pior caso, desconsiderando a variação de tempo gerada por processos concorrentes no momento da execução do programa, é quadrático. Tw(n) pertence a O(n²).



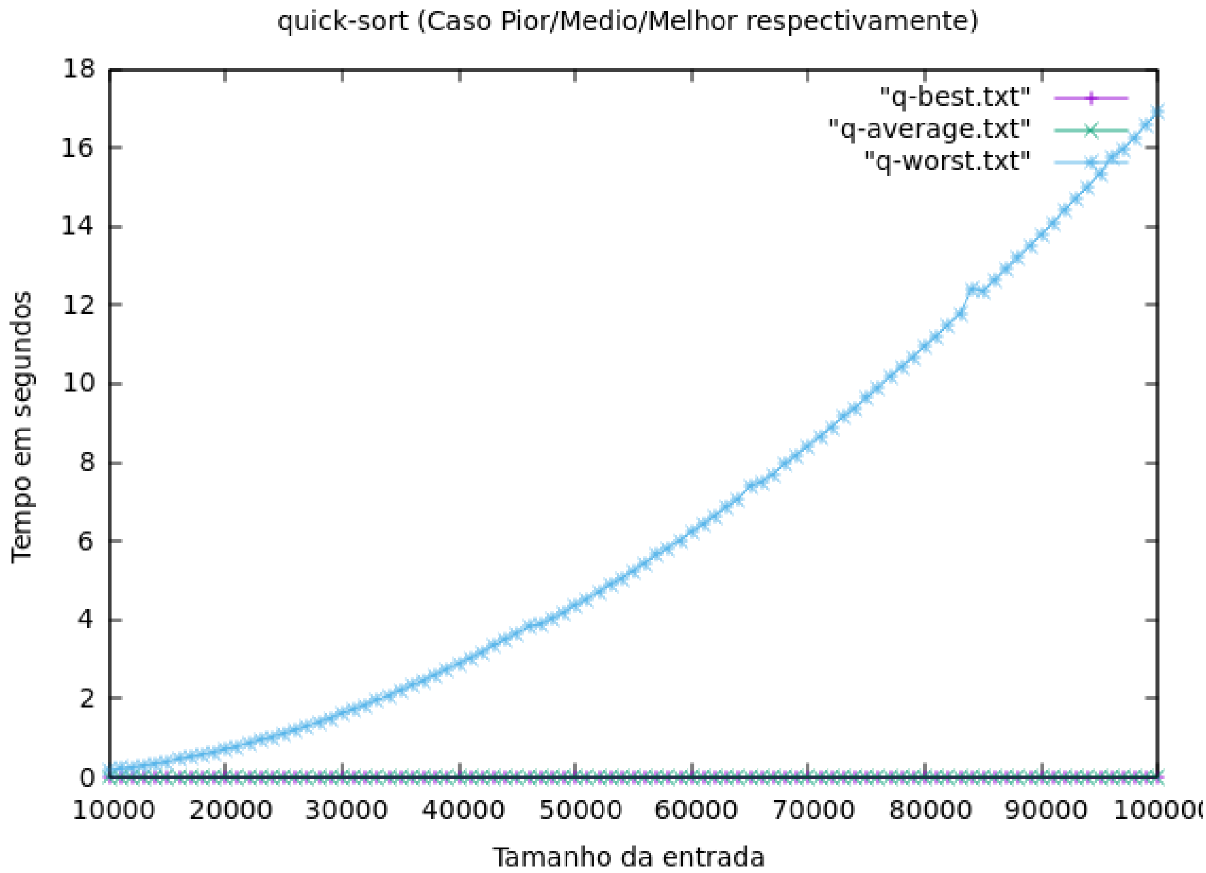
3.1.3 Caso médio

O gráfico abaixo representa o tempo de execução esperado do quick-sort em função do tamanho da entrada. Como entrada para gerar o gráfico, foram utilizados 91 vetores de tamanho n, preenchidos com números de 0 até n gerados aleatoriamente. Pela análise do gráfico podemos notar que o algoritmo no caso médio, desconsiderando a variação de tempo gerada por processos concorrentes no momento da execução do programa, pertence a O(n*log(n)).



3.1.4 Comparação dos gráficos

No gráfico abaixo podemos ver a comparação entre o caso médio, o melhor e o pior caso do algoritmo. É possível perceber que o tempo esperado e o melhor caso tem tempo muito inferior ao tempo do pior caso.



3.2 Análise analítica do tempo de execução

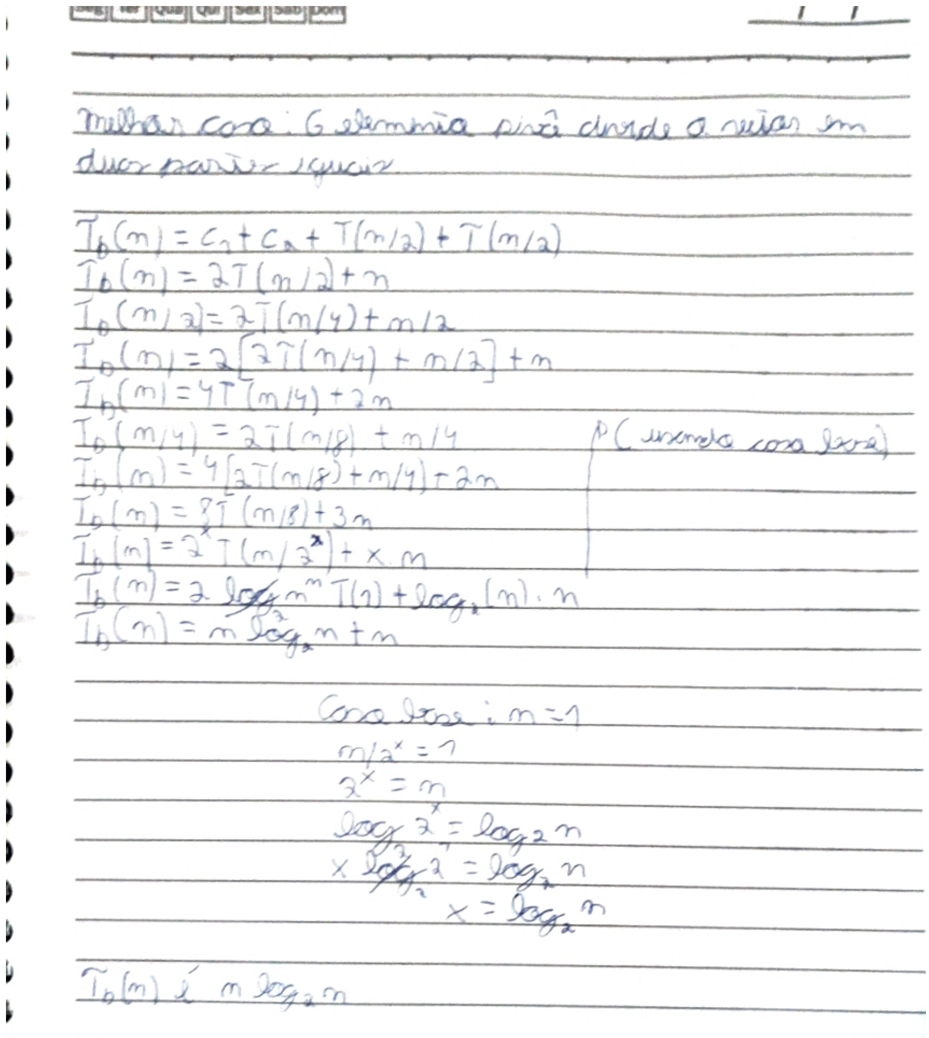
3.2.1 Algoritmo

```
void quick_sort(int*v, int r, int l){
    int p;
    if(r<l){
        p=partition(v,r,l);
        quick_sort(v,r,p-1);
        quick_sort(v,p+1,l)
    }
}
```

Jonathan Tauan Pereira maia

3.2.1.1 Melhor caso

A análise analítica do tempo de execução do quick-sort no seu melhor caso, que se dá quando o vetor já está ordenado, porém, com o elemento da posição do meio invertido com o último elemento do vetor, e temos como pivô a última posição, permitiu perceber que seu tempo de execução será (n*log (n)).



Melhor caso: O elemento pivô divide o vetor em duas partes iguais.

$T_b(n) = c_1 + c_2 + T(n/2) + T(n/2)$
$T_b(n) = 2T(n/2) + n$
$T_b(n/2) = 2T(n/4) + n/2$
$T_b(n) = 2[2T(n/4) + n/2] + n$
$T_b(n) = 4T(n/4) + 2n$
$T_b(n/4) = 2T(n/8) + n/4$ — p (usando caso base)
$T_b(n) = 4[2T(n/8) + n/4] + 2n$
$T_b(n) = 8T(n/8) + 3n$
$T_b(n) = 2^x T(n/2^x) + x.n$
$T_b(n) = 2^{\log_2 n} T(1) + \log_2(n).n$
$T_b(n) = n \log_2 n + n$

Caso base: $n = 1$
$n/2^x = 1$
$2^x = n$
$\log_2 2^x = \log_2 n$
$x \log_2 2 = \log_2 n$
$x = \log_2 n$

$T_b(n)$ é $n \log_2 n$

3.2.1.2 Pior caso

A análise analítica do tempo de execução do quick-sort no seu pior caso, que se dá quando o vetor já está ordenado, e pegamos o pivô também como o último elemento do vetor, permitiu perceber que seu tempo de execução será quadrático, f(n) = $an^2 + bn + c$.

Pior caso: O vetor pode já estar ordenado e o pivô será sempre o último elemento

$$T_w(n) = c_1 + c_2 . n + T(1) + T(n-1)$$
$$T_w(n) = c_1 + c_2 . n + T(n-1)$$
$$T_w(n-1) = c_1 + c_2 (n-1) + T(n-2)$$
$$T_w(n) = c_1 + c_2 . n + [c_1 + c_2 (n-1) + T(n-2)]$$
$$T_w(n) = 2c_1 + c_2 . (n + (n-1)) + T(n-2)$$
$$T_w(n-2) = c_1 + c_2 (n-2) + T(n-3)$$
$$T_w(n) = 2c_1 + c_2 . (n + (n+1)) + [c_1 + c_2 . (n-2) + T(n-3)]$$
$$T_w(n) = 3c_1 + c_2 (n + (n-1) + (n-2)) + T(n-3)$$
$$T_w(n) = x c_1 + c_2 . (n + \dots + (n - x - 1)) + T(n - x)$$

Caso base: $n = 1$,
$n - x = 1$
$x = n - 1$

$$T_w(n) = (n-1) . c_1 + c_2 (n + (n-1) + \dots + (n-(n-1))) + T(1) + 0$$
$$T_w(n) = (n-1) . c_1 + c_2 (n + (n-1) + \dots + 0)$$
$$c_2 . (n + (n-1) + \dots + 2 + 1 + 0), \; c_2 . 0 = 0$$
$$n + 1, \; (n-1) + 2 = n + 1 \rightarrow \frac{n}{2}(n+1)$$

$$T_w(n) = (n-1) . (c_1 + c_2) \left(\frac{n}{2} (n+1) \right)$$

$$T_w(n) = n^2 \underbrace{\left(\frac{c_2}{2} \right)}_{a} + n \underbrace{\left(c_1 + \frac{c_2}{2} \right)}_{b} + \underbrace{(-c_1)}_{c}$$

$$T_w(n) = an^2 + bn + c$$

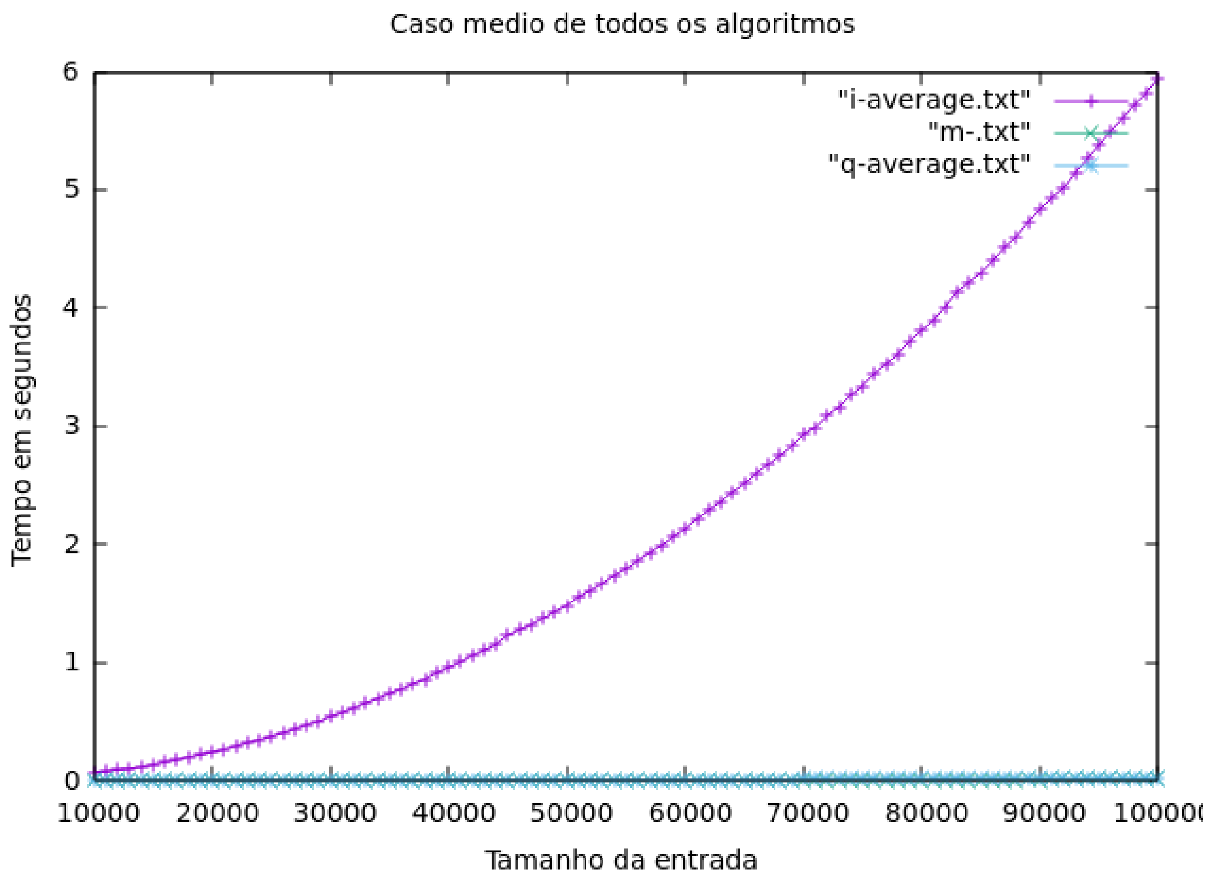
$T_w(n)$ é quadrático

4 Comparação de desempenho em relação ao custo de tempo e memória

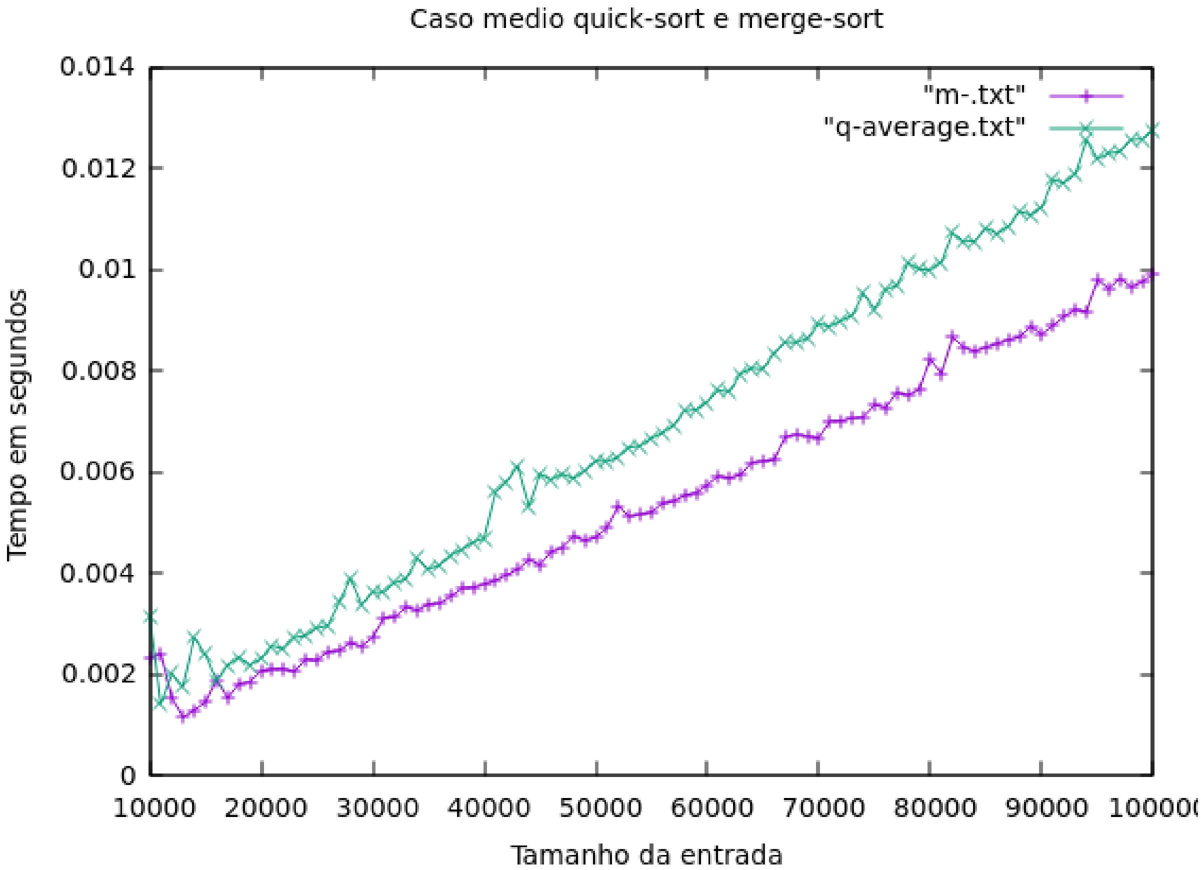
4.0.1 Gráficos

4.0.1.1 Comparação do tempo esperado dos algoritmos

O gráfico abaixo representa o tempo esperado dos três algortimos. É possível notar que o insertion-sort apresenta um tempo esperado muito maior do que os outros dois algoritmos.

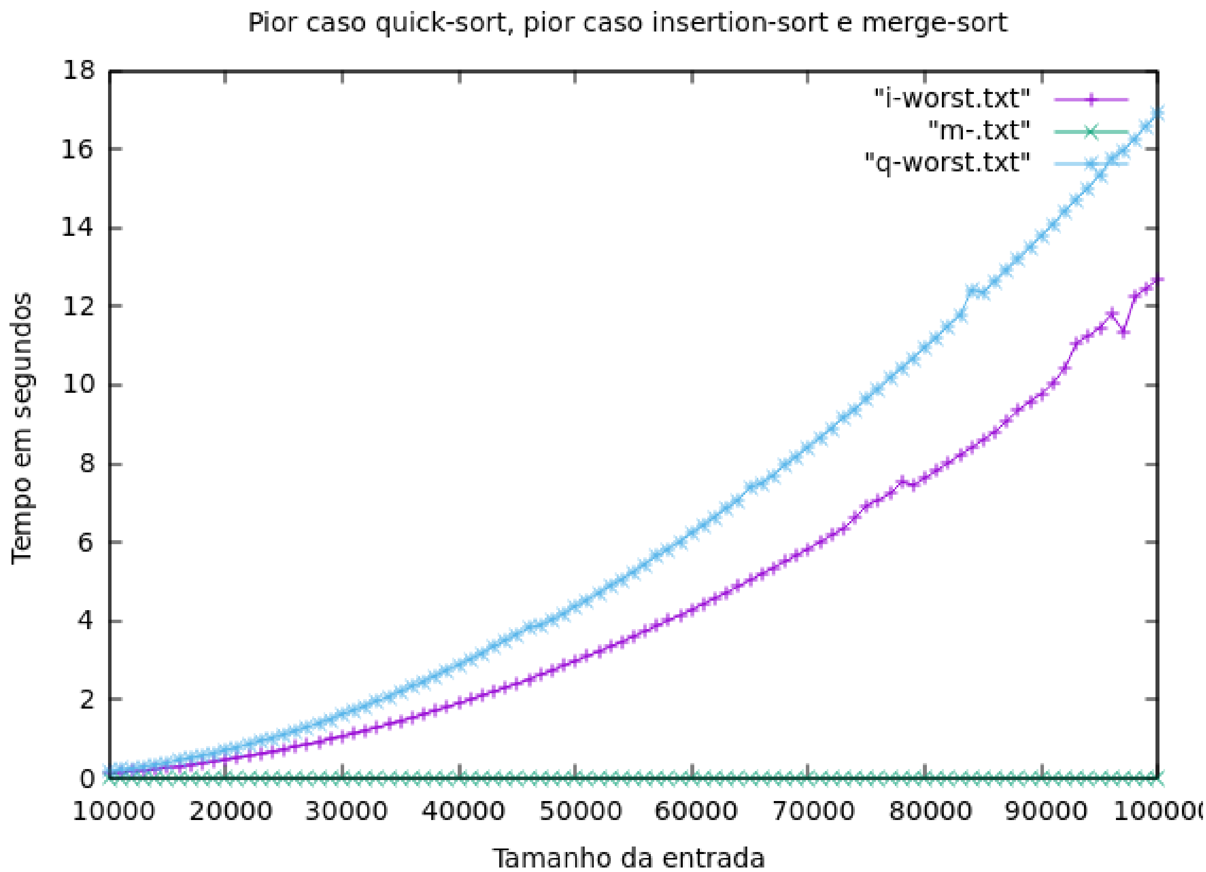


No gráfico abaixo excluímos o insertion-sort para podermos ver uma comparação melhor do tempo esperado do quick-sort com o tempo do merge-sort.



4.0.1.2 Comparação do tempo do pior caso dos algoritmos

Aqui abaixo temos a comparação do tempo do pior caso dos algoritmos quick-sort e insertion-sort com o o tempo do merge-sort.



4.0.2 Conclusão

Pela a análise dos gráficos de cada algoritmo levando em consideração seu melhor, pior e caso médio, foi possível observar que dentre os três, o algoritmo insertion-sort foi o que apresentou menor desempenho em relação ao tempo de execução esperado.

Porém, como não foi implementado de forma recursiva, o mesmo pode custar a menor quantidade de memória em comparação aos outros dois e seu pior caso ainda consegue ser melhor do que alguns casos específicos de pior caso do quick-sort.

Ainda analisando os algoritmos quick-sort e o merge-sort, podemos notar que, de acordo com o gráfico, o algoritmo quick-sort em seu melhor caso e seu caso médio, ainda é se apresentou rápido que o merge-sort.

Entretanto, o algoritmo merge-sort ainda se apresenta mais estável, pois, para todos os casos apresenta complexidade (n*log(n)), ao contrário do quick-sort que é volátil e mais instável, e apesar de ser mais rápido na maioria das vezes, no seu pior caso, pode chegar a complexidade de tempo $O(n^2)$, sendo bem inferior quando comparado ao tempo de execução do merge-sort.

Além disso, devemos considerar a implementação desses dois últimos de forma recursiva, que para grandes entradas podem apresentar uma sobrecarga na pilha de execução. Mas em relaçâo a memória, o quick-sort sai na frente pois o seu concorrente, o merge-sort, aloca um novo vetor a cada chamada recursiva da função merge, o que gera um gasto adicional e ainda mais notável em linguagens de mais alto nível.

4.0.2.1 Tabelas

Melhor caso

Tamanho da entrada	Tempo (s)		
	insertion-sort	merge-sort	quick-sort
10000	0.000043	0.002320	0.001124
20000	0.000057	0.002062	0.000842
30000	0.000084	0.002724	0.001341
40000	0.000126	0.003782	0.001962
50000	0.000193	0.004701	0.002213

Pior caso

Tamanho da entrada	Tempo (s)		
	insertion-sort	merge-sort	quick-sort
10000	0.124559	0.002320	0.187187
20000	0.476691	0.002062	0.718812
30000	1.074601	0.002724	1.630973
40000	1.908465	0.003782	2.887971
50000	2.982496	0.004701	4.367089

Tempo esperado

Tamanho da entrada	Tempo (s)		
	insertion-sort	merge-sort	quick-sort
10000	0.070669	0.002320	0.003140
20000	0.239225	0.002062	0.002307
30000	0.537542	0.002724	0.003623
40000	0.955588	0.003782	0.004691
50000	1.482152	0.004701	0.006215