Nome Ketlly A. de Medeiros* Nome João Paulo de S. Medeiros†

11 de junho de 2022

Resumo

Trabalho desenvolvido com o objetivo de observar os tempo de execução de algoritmos de ordenação, sendo eles o insertion-sort, o merge-sort e o quick-sort. Será possível analisar esses tempos graficamente, e compará-los, além de fazer a analíse assintótica.

1 Insertion-sort

O inserion Sort é a forma de ordenação que utiliza apenas um vetor, e não utiliza chamadas recursivas em seu código. Ele vai deixa o vetor em ordem crescente vendo um pedaço do vetor, ordendando, e aumentando a visualização. Exemplo: temos um vetor [7,1,4,5,8], ele primeiro observa: [7,1] organiza: [1,7] e aumenta sua visão: [1,7,4], e continua essa repetição até todo o vetor está organizado. E pode haver casos que o código execute mais rápidamente, ou mais lentamente.

1.1 Algoritmo:

1.1.1 Insertion-sort()

algoritmo insertion-sort(v, n)

para i de 2 até n **faça**
    $k \leftarrow v[i]$
    $j \leftarrow i - 1$
    enquanto $j >= 1$ e $v[j] > k$ **faça**
        $v[j+1] \leftarrow v[j]$
        $j \leftarrow j - 1$
    fim enquanto
    $v[j+1] \leftarrow k$
fim para

1.2 Melhor Caso

O melhor caso do insertion-sort consiste quando o vetor ja está ordenado, pois não haverá nescessidade de realizar as trocas, com isso, ele rodará rapidamente, tendo uma solução lineare de análise assintótica $\Theta(n)$.

*Aluno do Bacharelado em Sistemas de Informaçãoda Universidade Federal do Rio Grande do Norte. Membro do . (e-mail: ketlly.azevedo.090@ufrn.edu.br)

†Professor do Departamento de Computação e Tecnologiada Universidade Federal do Rio Grande do Norte. Da matéria de Estrutura de Dados

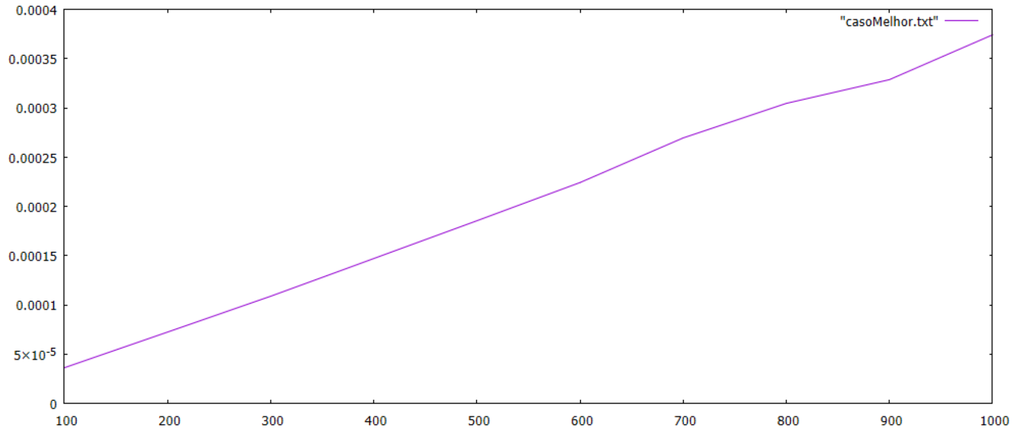

Figura 1: Melhor Caso do insertion sort

$$T_b(n) = nC_1 + (n-1)(C_2 + C_3 + C_4 + C_7)$$

$$T_b(n) = n(C_1 + C_2 + C_3 + C_4 + C_7) + (-C_2 - C_3 - C_4 - C_7)$$

$$na + b$$

linear $\Theta(n)$

26 1.3 Pior Caso

O pior caso resulta quando o vetor está em ordem decrescente, pois assim, terá que realizar todas as trocas, fazendo o algorítmo rodar seu maior número de vezes. Ele tem uma solução quadrática de analise assintótica $\Theta(n^2)$.

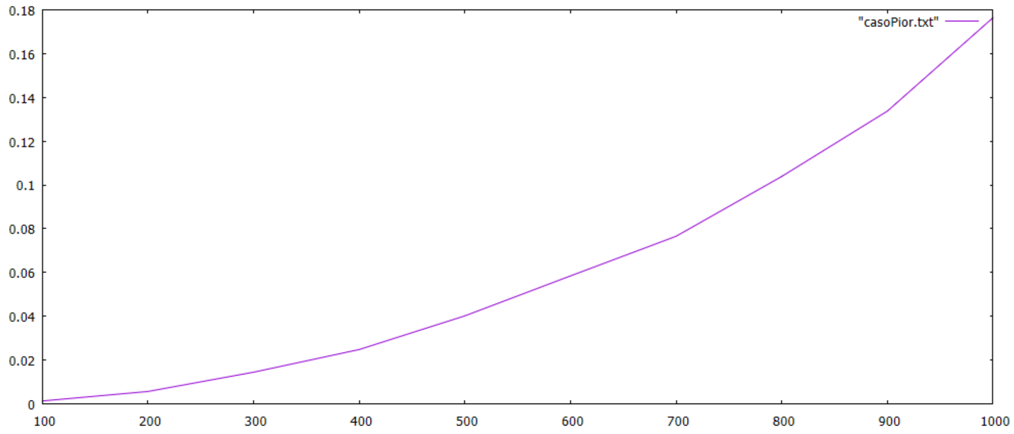

Figura 2: Pior Caso do insertion sort

$$T_w(n) = nC_1 + (n-1)(C_2+C_3+C_7) + C_4\left(\sum_{i=2}^{n} i\right) + (C_5+C_6)\left(\sum_{i=1}^{n-1} i\right)$$

$$T_w(n) = nC_1 + (n-1)(C_2+C_3+C_7) + C_4\left(\frac{n^2+n-2}{2}\right) + (C_5+C_6)\left(\frac{n^2-n}{2}\right)$$

$$T_w(n) = n(C_1+C_2+C_3+C_7) - (C_2+C_3+C_7) + n^2\frac{C_4}{2} + n\frac{C_4}{2} - C_4 + n^2\frac{(C_5+C_6)}{2} - n\frac{(C_5+C_6)}{2}$$

$$T_w(n) = n^2\left(\frac{C_4+C_5+C_6}{2}\right) + n\left(C_1+C_2+C_3+C_7+\frac{C_4}{2}-\frac{C_5+C_6}{2}\right) - (C_2+C_3+C_4+C_7)$$

$$n^2a + nb + c \rightarrow \text{quadratica} \quad \Theta(n^2)$$

1.4 Medio Caso

O médio caso vem com vetores aleatórios, sem ser crescente ou decrescente, e para sabermos seu tempo de execução, é uma conta com todos os seus possíveis caso, o que resulta em uma equação quadrática, tendo assim, uma analise assintótica $\Theta(n^2)$.

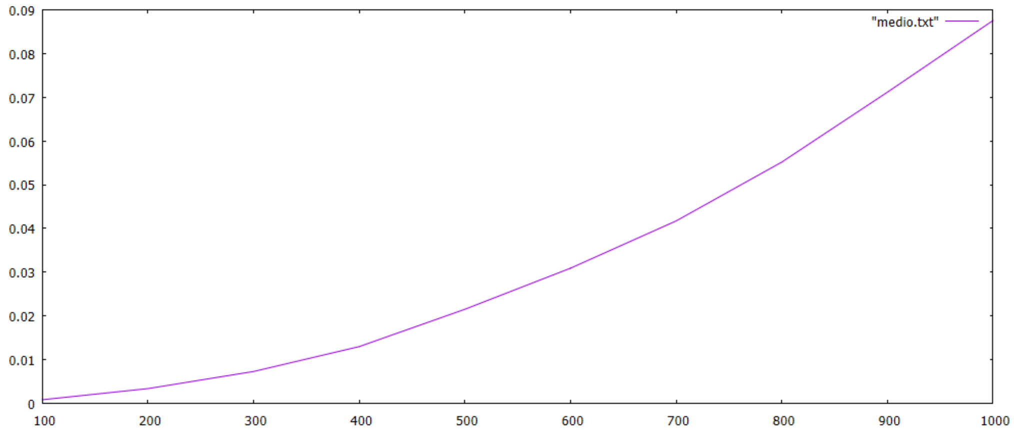

Figura 3: Medio Caso do insertion sort

1.5 Conclusão

Aqui podemos observar claramente os tempos de execução dos casos apresentados acima. O pior caso se destaca, e o melhor quase não consegumios vêr, ja que por ser linear se sobressai em velocidade que uma quadrática.

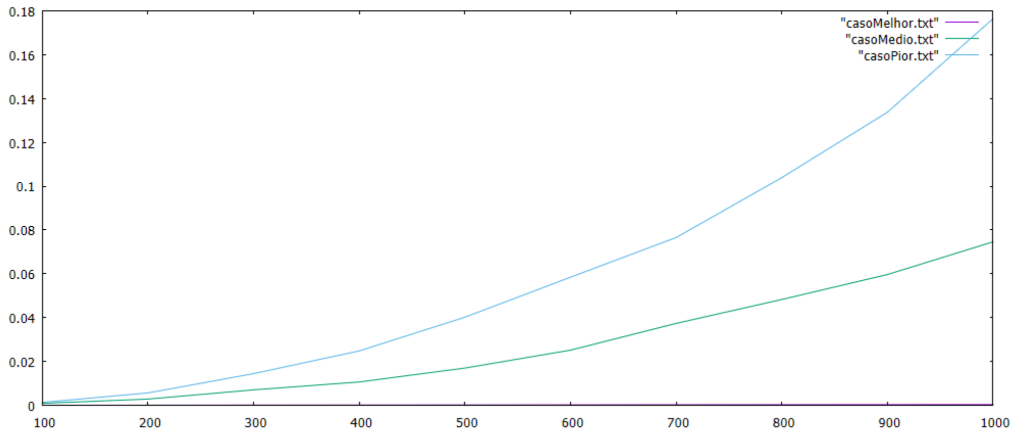

Figura 4: Commparação do insertion sort

2 Merge-Sort

O merge-sort é um algoritmo recursivo, que necessita do auxílio do código merge para fazer a ordenação. Eles funciona da seguinte maneira: ele vai dividindo o vetor ao meio n vezes até termos apenas as posições separadas (Ex: [3,5,8,6] - [3,5] [8,6] - [3][5] [8][6]), pois assim já estarão ordenados, em seguida junta-se as posições novamente, aos poucos, ordenando-as (Ex:[3,5] [6,8] - [3,5,6,8]) tendo ao final um vetor ordenado.

2.1 Algoritmo:

2.1.1 Merge-sort()

algoritmo merge-sort(v, s, e)

se $s < e$ **então**
    $m \leftarrow (s + e)/2$
    $merge - sort(v, s, m)$
    $merge - sort(v, m + 1, e)$
    $merge(v, s, m, e)$
fim se

2.1.2 Merge()

algoritmo merge(v, s, m, e)

$i \leftarrow s$
$j \leftarrow m + 1$
para k de 1 até $(e - s + 1)$ **faça**
    se $(s < e)$ ou $(i <= m$ e $v[i] < v[j])$ **então**
        $w[k] \leftarrow v[i]$
        $i \leftarrow i + 1$
    else
        $w[k] \leftarrow v[j]$
        $j \leftarrow j + 1$
    fim se
fim para
para k de 1 até $(e - s + 1)$ **faça**
    $v[s + k - 1] \leftarrow w[k]$
fim para

63 ## 2.2 merge-sort

Como podemos analiza, independente da forma que o vetor está organizado, ele rodará todas as vezes, ou seja, não teremos melhor ou pior caso, sendo assim tremos uma função polilogaritmo, de analise assintótica $\Theta(n \cdot \log_n)$.

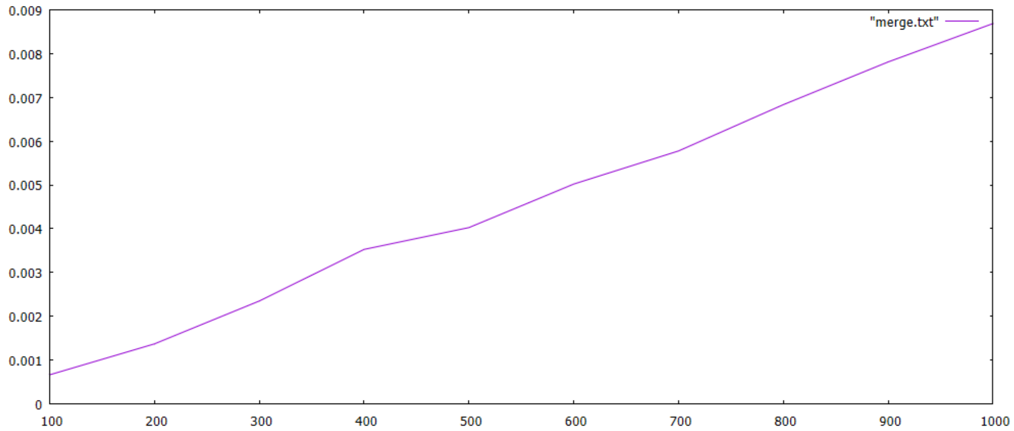

Figura 5: Tempo de execução do Merge Sort

$T(1) = C_1$ ← caso base

$T(n) = C_1 + C_2 + C_3 + C_4 + C_5 + T(n/2) + T(n/2) + Tm(n)$

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

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

merge é linear

$b\frac{n}{2^i} + C$

$T(n) = 3a + 4T(n/4) + Tm(n/2) + Tm(n)$

$T(n) = (x-1)a + x\,T(n/x) + \sum_{i=0}^{(\log_2 x)-1} 2^i\, Tm(n/2^i)$

caso base

$T(n) = (n-1)a + n\,T(n/n) + \sum_{i=0}^{\log_2 n} 2^n\, Tm(n/2^n)$

propriedade

$x^{\log_x n} = n$

$T(n) = (n-1)a + nC_1 + b\,n \cdot \log_2 n + C\,(2^{\log_2 n} - 1)$

$T(n) = (n-1)a + nC_1 + b\log_2 n + C\,(n-1)$

$bn\log_2 n + n(a + C_1 + C) - a - c$

polilogaritima $\Theta(n \cdot \log n)$

3 Quick-Sort

O quick-sort utiliza de chamadas recursivas e do codigo partition para encontrar o pivô do vetor. Utilizamos o pivô, que foi assumido que seria a última posição, para assim ordenar o vetor, pois ele sempre vai comparando com as demais posições, com o objetivo de deixar sempre os menores a direita e os maiores a esqueda, a fim de conseguir o vetor inteiramente ordenado (Ex: [3,5,7,2,1,9] - [3,5,7,2,1,9'] - [3,5,7,2,1',9] - [1,3,5,7,2',9] - [1,2,3,5,7',9] - [1,2,3,5,7,9]). Além de que pode haver situações que o algoritmo compilará mais rápido oou mais devagar.

3.1 Algoritmo:

3.1.1 quick-sort()

algoritmo quick-sort(v, s, e)

se $s < e$ **então**
    $p \leftarrow partition(v, s, e)$
    $quick - sort(v, s, p - 1)$
    $quick - sort(v, p + 1, e)$
fim se

3.1.2 partition()

algoritmo partition(v, s, e)

$k \leftarrow v[e]$
$i \leftarrow s - 1$
para j de s até $e - 1$ **faça**
    se $v[j] <= k$ **então**
        $i \leftarrow i + 1$
        $v[i] \leftrightarrow v[j]$
    fim se
fim para
$v[i + 1] \leftrightarrow v[e]$
retorne $i + 1$

Lembrando que o pivô está sendo o ultimo elemento do vetor.

92 3.2 Melhor Caso

O melhor caso do quick-sort() se dá de maneira forçada, já que para isso é nescessário que o pivô escolhido seja aquele que divide o vetor ao meio, em outras palavras, o elemento central do vetor ordenado. Com isso conseguimos uma equação polilogaritima e de analise assintótica $\Theta(n \cdot \log_n)$.

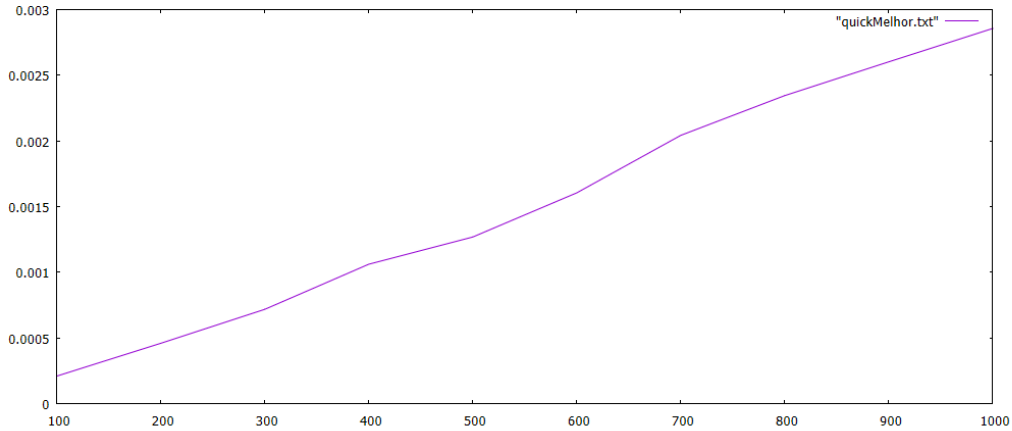

Figura 6: Melhor caso do Quick Sort

$T(0) = T(1) = C_1 \leftarrow$ Caso base

$$T_b(n) = \underbrace{C_1 + C_2 + C_3 + C_4}_{a} + T_p(n) + 2T_b\left(\frac{n-1}{2}\right)$$

$$T_b\left(\frac{n-1}{2}\right) = a + T_p\left(\frac{n-1}{2}\right) + 2T_b\left(\frac{n-3}{4}\right)$$

$$T_b(n) = 3a + T_p(n) + 2T_p\left(\frac{n-1}{2}\right) + 4T_b\left(\frac{n-3}{4}\right)$$

$$T_b\left(\frac{n-3}{4}\right) = a + T_b\left(\frac{n-3}{4}\right) + 2T_b\left(\frac{n-7}{8}\right)$$

$$T_b(n) = 7a + T_p(n) + 2T_p\left(\frac{n-1}{2}\right) + 4T_p\left(\frac{n-3}{4}\right) + 8T_b\left(\frac{n-7}{8}\right)$$

$$T_b(n) = (2^x - 1)a + \left(\sum_{i=0}^{x-1} 2^i T_p\left(\frac{n-(2^i-1)}{2^i}\right)\right) + 2^x T_b\left(\frac{n-(2^x-1)}{2^x}\right)$$

$$\frac{n-(2^x-1)}{2} = 1$$

$$n - 2^x + 1 = 2$$

$$n - 1 = 2^{x+1}$$

$$\log_2(n-1) = \cancel{\log_2 2}^{x+1}$$

$$x = \log_2(n-1) - 1$$

encontrar o valor de x para o caso base

Partition é linear

$$Tb(n) = \left(2^{\log(n-1)-1} - 1\right)a + \sum_{i=0}^{\log(n+1)-2} 2^i T_p\left(\frac{n-(2^i-1)}{2^i}\right) + 2^{\log(n-1)-1}\, Tb\left(n - \left(\frac{2^{\log(n+1)-1}}{2^{\log(n+1)-1}} - 1\right)\right)$$

$$Tb(n) = \left(\frac{n-1}{2}\right)a + \left(\frac{n+1}{2}\right)c_1 + \sum_{i=0}^{\log(n+1)-2}\left[b\left(n-(2^i-1)\right) + 2^i c\right]$$

$$Tb(n) = \left(\frac{n-1}{2}\right)a + \left(\frac{n+1}{2}\right)c_1 + \left(\log_2(n+1)-1\right)bn + \left(\log_2(n+1)-1\right)b + bn + cn$$

$$Tb(n) = \left(\frac{n-1}{2}\right)a + \left(\frac{n+1}{2}\right)c_a + bn\log_2(n+1) - 2bn + b \cdot \log_2(n+1) - b + nc$$

Polilogaritima $\Theta(n \cdot \log_2 n)$

93 3.3 Pior Caso

O pior caso resulta quando o pivô é o maior ou menor valor do vetor, pois assim, não terá como deixar os menore a esquerda e maiores a direita, ja que ele é um extremo, e isso ocasiona do algoritmo rodar inumeras vezes, gerando uma função quadratica de analise assintótica $\Theta(n^2)$

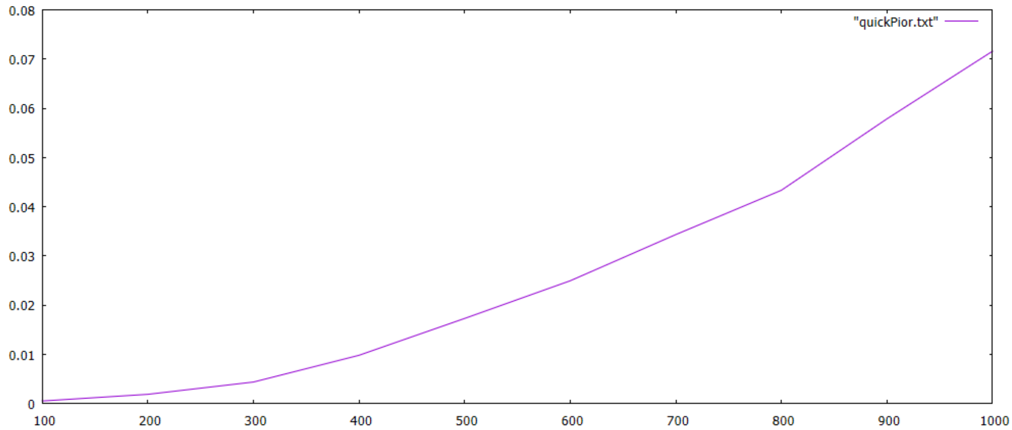

Figura 7: Pior caso do Quick Sort

$$T_w(n) = C_1 + C_2 + T_p(n) + T_w(n-1) + T_w(0)$$

$T_w(0) = T_w(1) = C_1 \leftarrow$ caso base

$$T(n-1) = a + T_p(n-1) + T(n-2) + T_w(0)$$

$$T_w(n) = 2a + T_p(n) + T_p(n-1) + T_w(n-2) + 2T_w(0)$$

$$T(n-2) = a + T_p(n-2) + T_w(n-3) + T_w(0)$$

$$T_w(n) = 3a + T_p(n) + T_p(n-1) + T_p(n-2) + T_w(n-3) + 3T_w(0)$$

$$T_w(n) = Xa + \sum_{i=0}^{X-1} T_p(n-i) + T_w(n+X) + XT_w(0)$$

$$T_w(1) = \quad X = (n-1)$$

$$T_w(n) = (n-1)a + T_w(n-n-1)) + (n-1)T_w(0) + \sum_{i=0}^{n-2} T_p(n-i)$$

$$T_w(n) = (n-1)a + T_w(1) + (n-1)C_1 + \sum_{i=0}^{n-2} [b(n-i)+c]$$

$$T_w(n) = (n-1)a + nC_1 + \sum_{i=0}^{n-2} C + \sum_{i=0}^{n-2} bn - \sum_{i=0}^{n-2} bi$$

$$T_w(n) = (n-1)a + nC_1 + (n-1)c + b\left[n^2 - n - \frac{(n-2)}{2}(n-1)\right]$$

quadratica $\Theta(n^2)$

3.4 Medio Caso

No médio caso, é utilizado valores aleatórios, tendo a possibilidade de ocorrer, também, o pior caso ou melhor caso. E para saber seu tempo de execução é necessário saber todos os possíveis caso, no qual, será obtido uma função polilogaritima, de analise assintótica $\Theta(n \cdot \log_n)$.

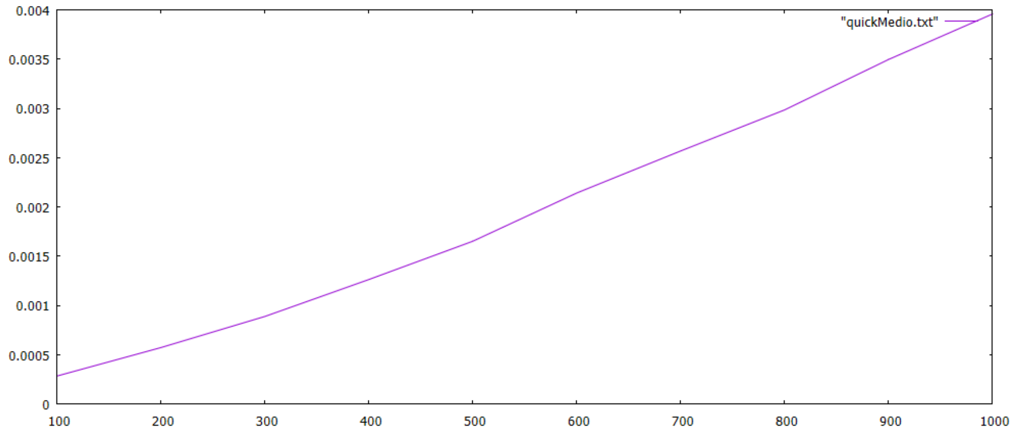

Figura 8: Medio caso do Quick Sort

3.5 Conclusão

Como é possível observar, as funções de melhor e médio caso são bem próximas por apresentar a mesma analise assintótica, já o pior caso, por ser quadrático se sobressai em lentidão, em comparação aos outros.

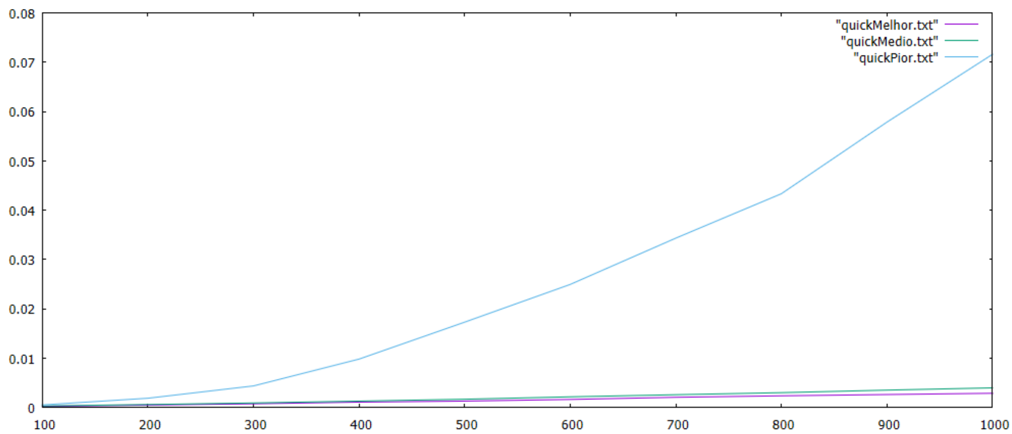

Figura 9: Commparação do quick sort

4 Conclusão

Pela observação dos aspectos analisados, em relação ao algoritmos de ordenação trabalhados, é possível observar qual se torna mais eficiente.

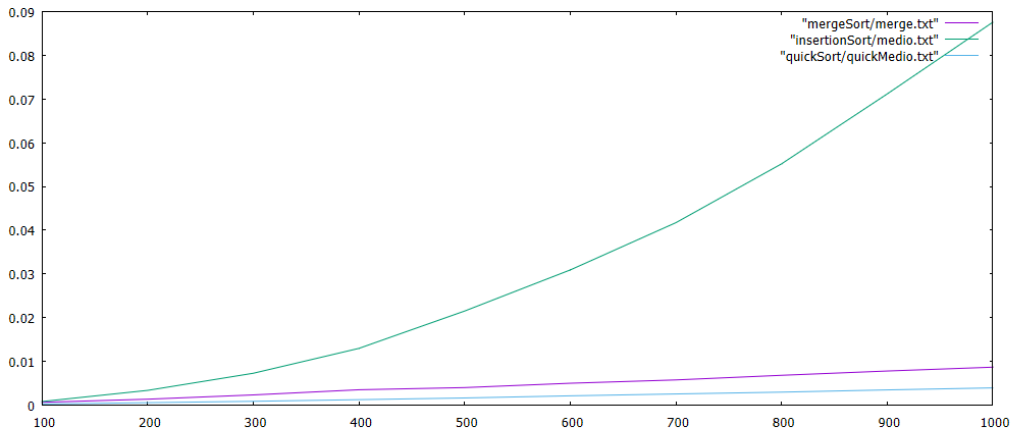

Figura 10: Commparação do algoritmos sort

O insertion-sort ele se torna o mais lento entre os trabalhado, ele se torna mais útil para pequenas entradas, ou seja, se ja tiver um vetor ordenado e queira acrescentar uma nova pequena parcela que precisa ser ordenada, ele se torna uma opção demasiadamente viável. Já o merge-sort ele mesmo tendo a mesma analise assintótica do quick, é possível observar que ele se torna um pouco mais lento, e isso se dá pela criação de um vetor auxiliar, e o fato de, no final, ter que transferi-lo ao vetor principal. E temos o quick-sort como uma das melhores opções, principalmente para vetores longos, pois ele tem um laço interno simples, o que o torna mais rápido.