Avaliação de OAD

Nome:Rafael Nogueira Rodrigues **Matrícula**: 2210100445

Introdução

Neste trabalho, falarei das diversas funções de ordenação trabalhadas em aula, no qual mostrarei e explicarei o funcionamento de cada uma, mostrando e comparando em gráfico o comportamento delas nos casos em que um vetor é randômico, e com tamanhos variantes, sendo assim, as funções a serem trabalhadas foram separadas em grupo 1 e grupo 2, além da Radix Sort(que será falada no final da avaliação).

No grupo 1, temos o bubble sort, selection sort e insertion sort. Esses algoritmos são considerados mais simples e representados pela fórmula ($O(n^2)$). Enquanto o grupo 2 será composto por quick sort, merge sort e heap sort, sendo representados pela fórmula (O(n xlog(x))).

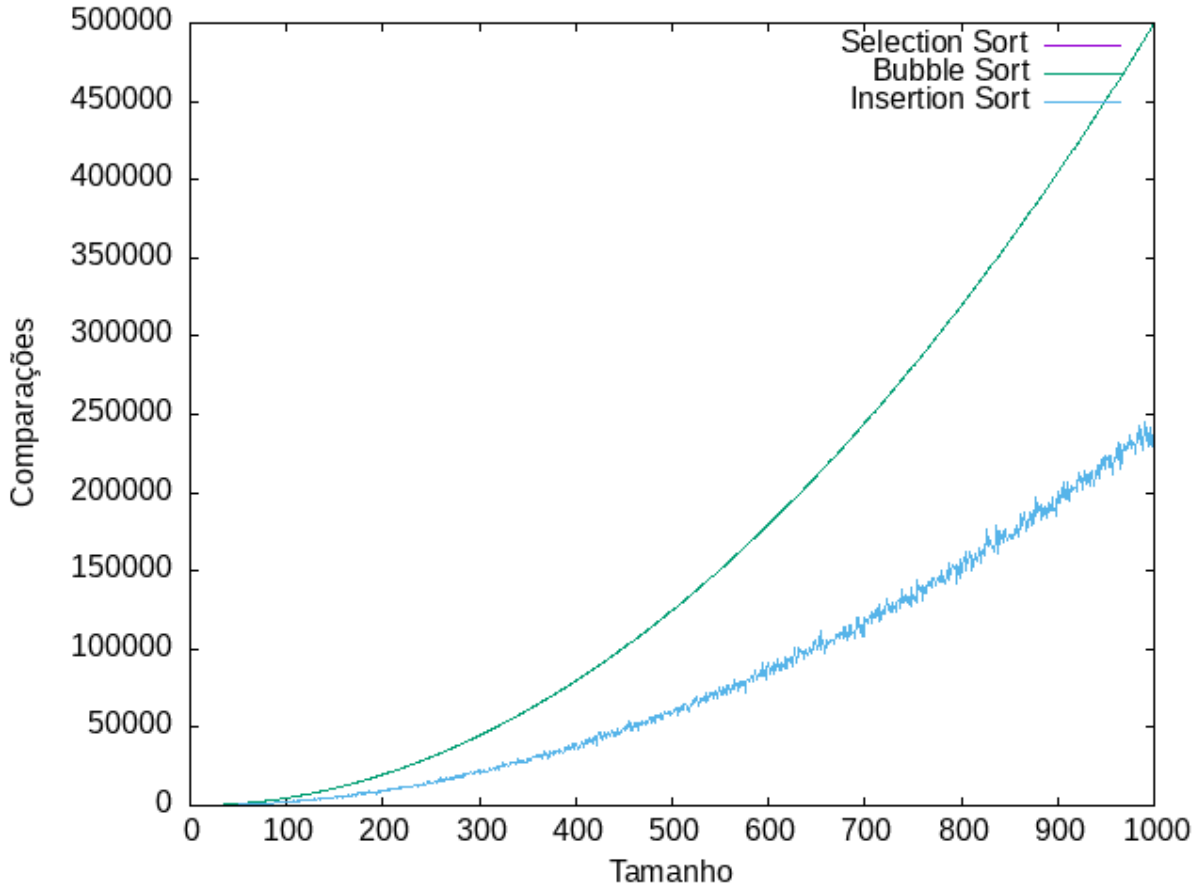
Grupo 1

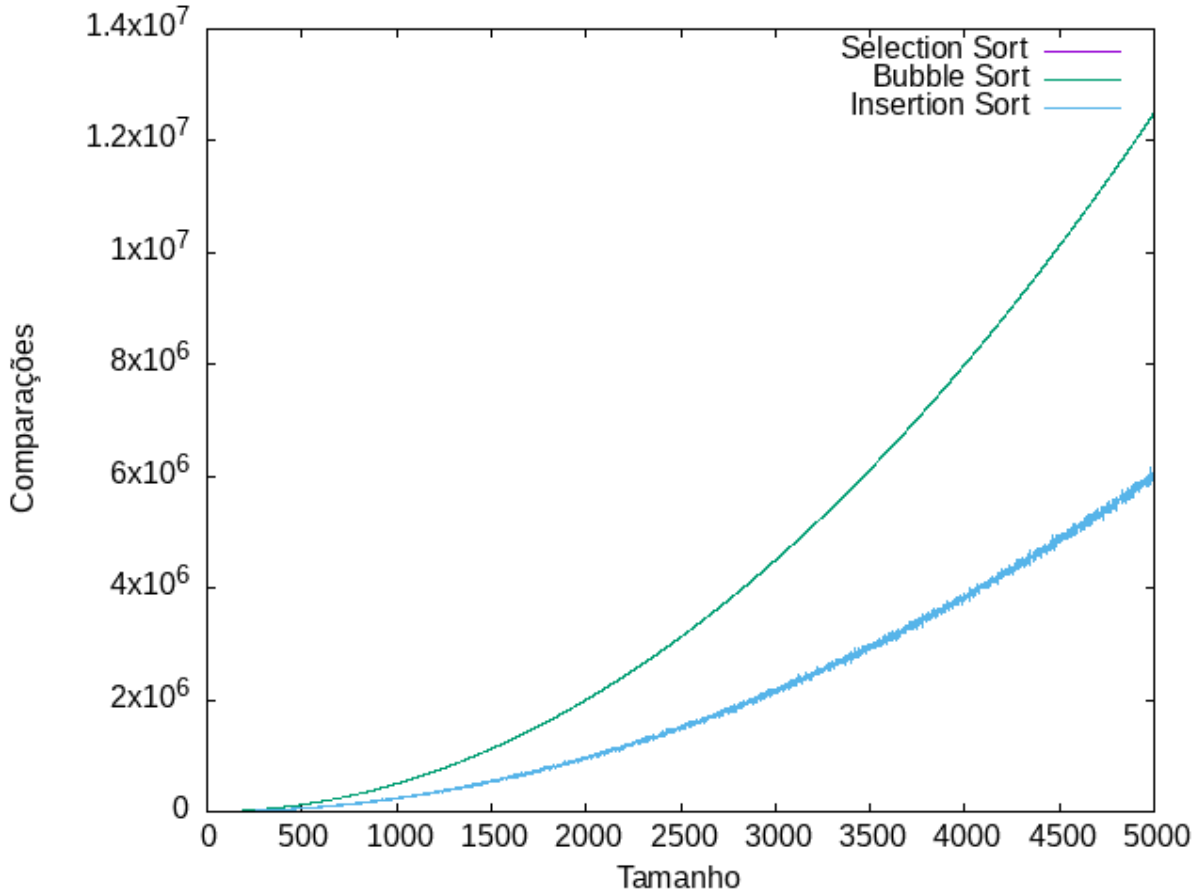
O grupo 1 é composto pelos algoritmos de ordenação bubble sort, selection sort e insertion sort. por mais que esses algoritmos sejam considerados mais simples em comparação aos outros do grupo 2, eles ainda desempenham um papel importante em determinados contextos e podem ser úteis em certas situações.

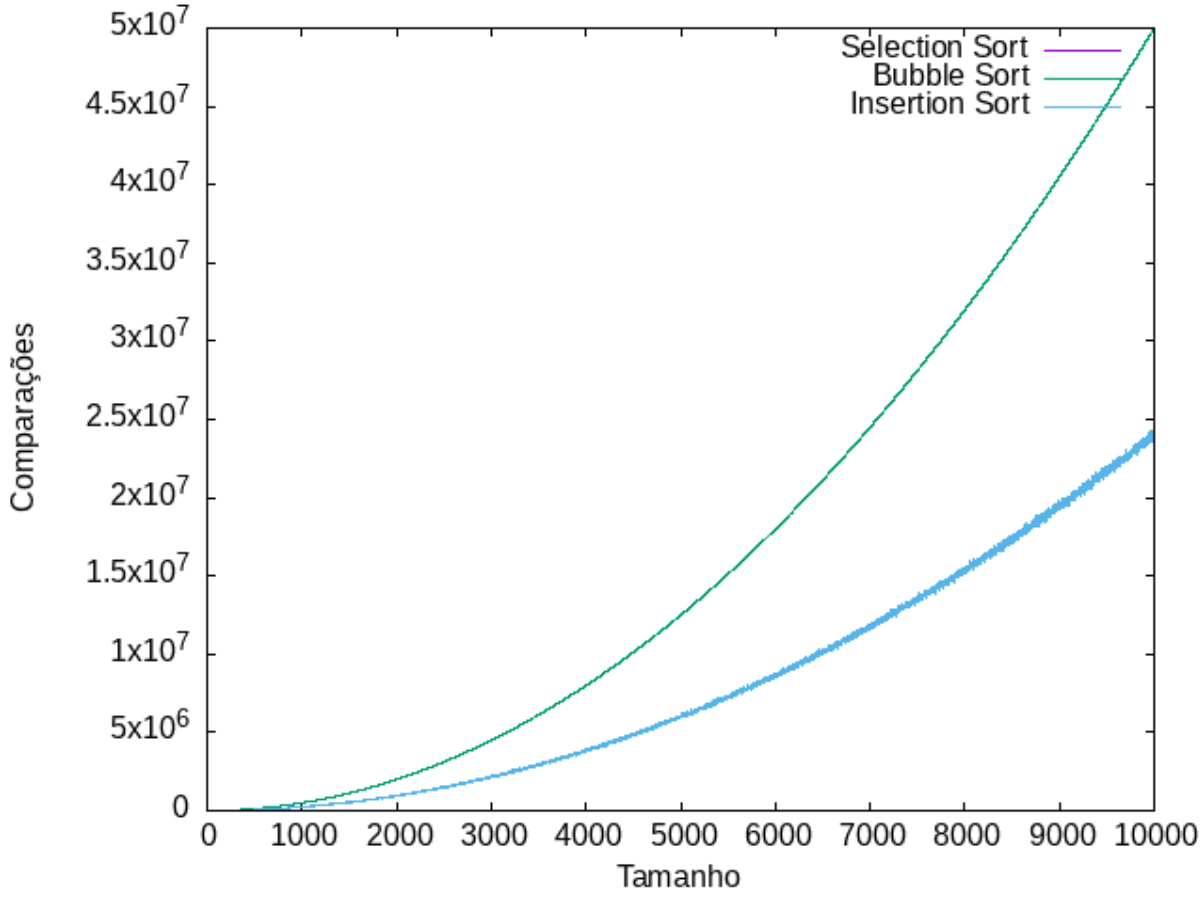
O bubble sort é considerado o algoritmo de ordenação mais simples, pois, sua função é unicamente comparar os elementos com os seus adjacentes, por exemplo, ele irá comparar o primeiro elemento com o segundo, caso ele seja menor e a ordenação esta sendo feita de forma crescente, o elemento será trocado, no caso do código desenvolvido, ele entrará na função swap para que isso ocorra, como ja foi dito, ele é simples, logo fará diversas comparações consideradas ineficientes e isso o torna inviável em algumas situações.

O selection sort também é um algoritmo simples de ordenação e sua característica é a que ele se baseia na posição(Index) de cada elemento, sendo assim, cada passo, o algoritmo encontra o menor elemento não classificado e o coloca na posição correta na sub lista ordenada. Isso se dará até que o vetor esteja ordenado, mais para frente do trabalho esse sort será descrito com mais detalhes.

O insertion sort é outro algoritmo de ordenação básico. uma de suas caracteristicas é a condição dentro do 2 loop “j > 0 && vetor[j - 1] > vetor [j]”, isso faz com que ele compare o elemento com seus antecessores, e para que não haja problema o loop inicial começa pela segunda posição.
Aqui está os gráficos de comparação do Grupo 1, alguns deles foram gerados tanto no linux quanto no windows, então fiz nos 2 Sistemas operacionais


Selection Sort
Bubble Sort
Insertion Sort
Comparações
Tamanho
0
50000
100000
150000
200000
250000
300000
350000
400000
450000
500000
0
100
200
300
400
500
600
700
800
900
1000


Selection Sort
Bubble Sort
Insertion Sort
Comparações
Tamanho
0
2x10^6
4x10^6
6x10^6
8x10^6
1x10^7
1.2x10^7
1.4x10^7
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000



Primeira observação sãos os dados, cada imagem representa a variação no tamanho, primeiro no windows foi feito com o tamanho que chamaremos de P,ou seja, o vetor iniciará com tamanho 1, irá até ter o tamanho 1000, cada loop, com uma incrementação de 1.O segundo tamanho chamaremos de M, começará com tamanho 1, irá até tamanho 5000, e com incrementação de 1.O terceiro tamanho chamaremos de G, começará com tamanho 1, irá até tamanho 10000 e com incrementação de 1.

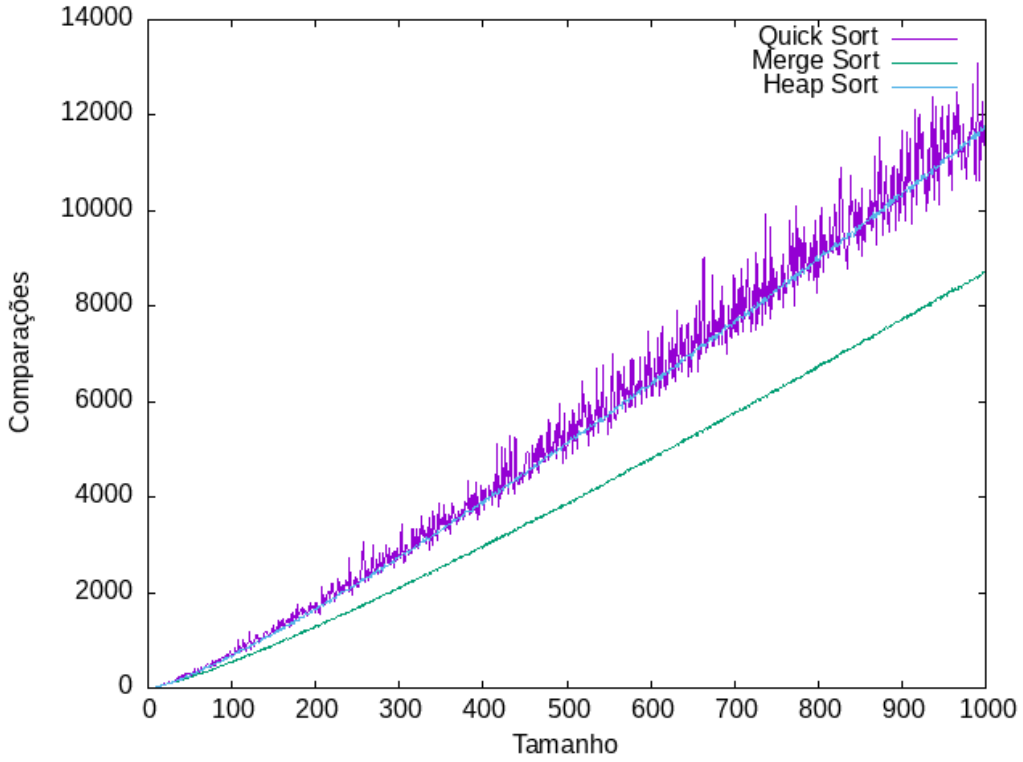
Segunda observação que podemos notar nos gráfico, é possivel ver que o Selection “não aparece” no gráfico, isso deve porque no numero de comparações ele tem o mesmo que o bubble sort, logo como as 2 linhas estão juntas o gnuplot mostrou apenas 1, ao ver nos arquivos dentro da pasta “txt”, terá os arquivos do grupo um nomeados de “g1” neles poderá ver os dados gerados, (“g1p.txt”, “g1m.txt”, “g1g.txt”)

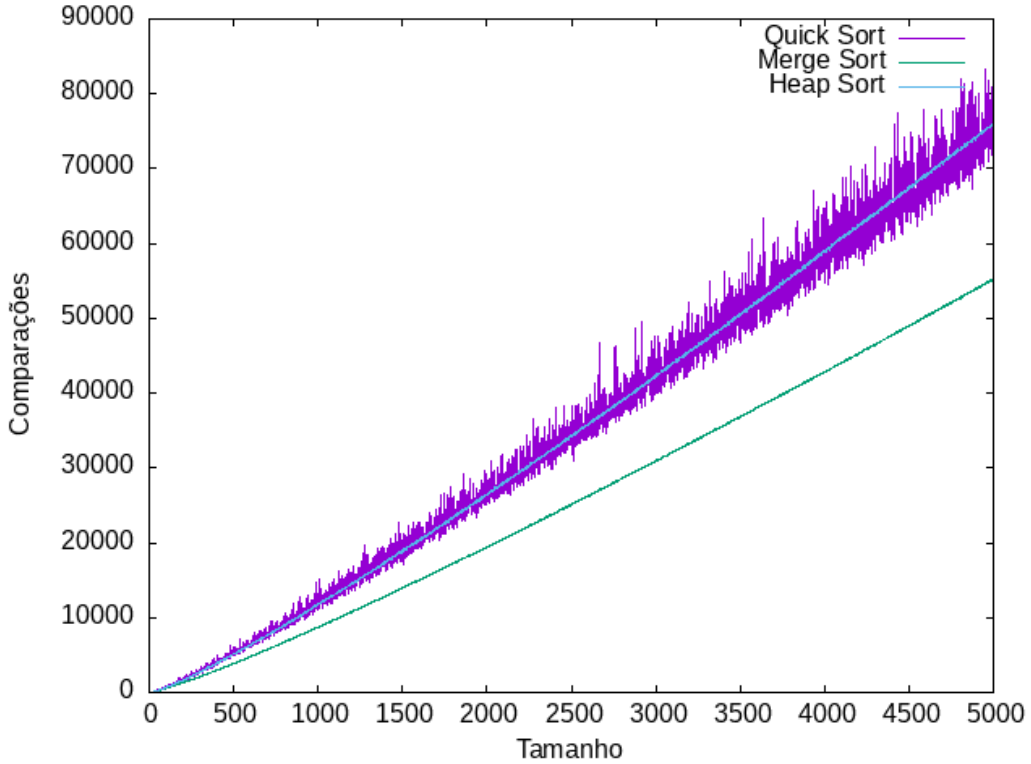
Grupo 2

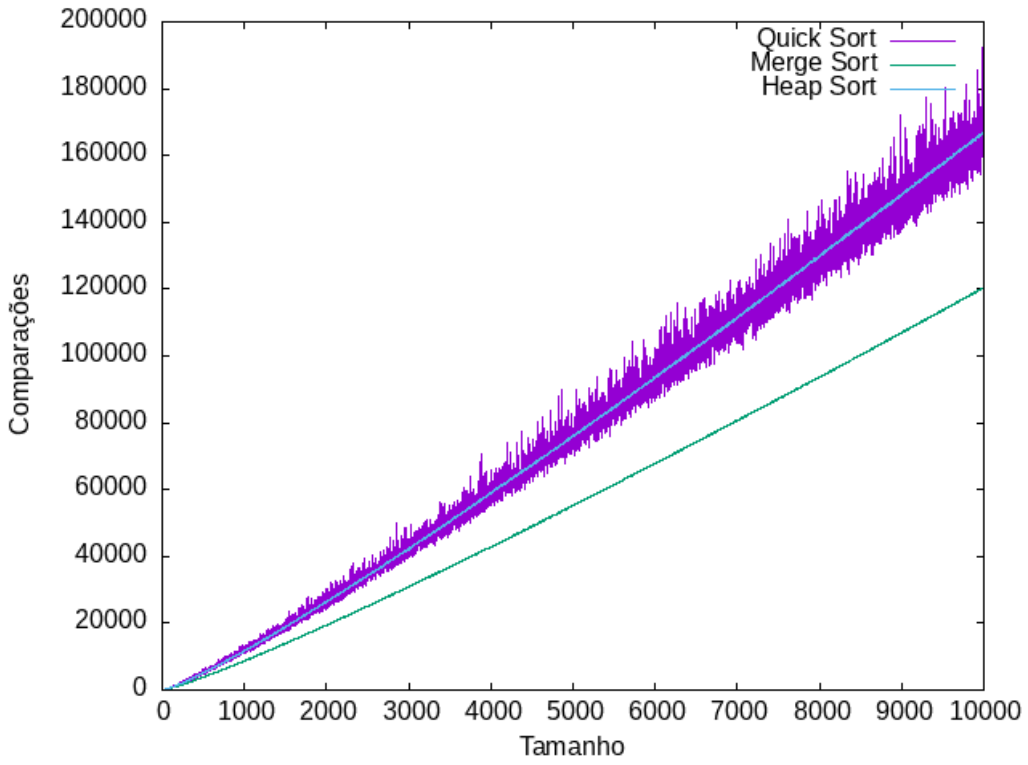
O grupo 2 possui sim, uma complexidade a mais que o grupo 1, eles são formados pelo quick sort, merge sort e heap sort.

Basicamente tanto o quick sort , quanto o merge sort, possuem a prática de “dividir para conquistar”. Quick sort ele tem como grande caracteristica a escolha de um pivô, no qual ele verificará se esse pivo é maior ou menor, então começar a ordenação de forma recursiva. Quanto ao Merge sort, sua maior característica é fato de que ele divide um vetor várias e várias vezes, até que ele possa começar a fazer trocas, isso permite com que ele desempenhe bem em grande numero de dados.

Heap Sort possui uma dinâmica diferente, e é comumente utilizado para descoberta de maior e menor valor.Isso se dá pois, ao pesquisarmos por ele, observamos que é julgado como uma arvore binária especial, no qual ele sempre extrai o valor da raiz, descobrindo assim o maior ou menor termo.





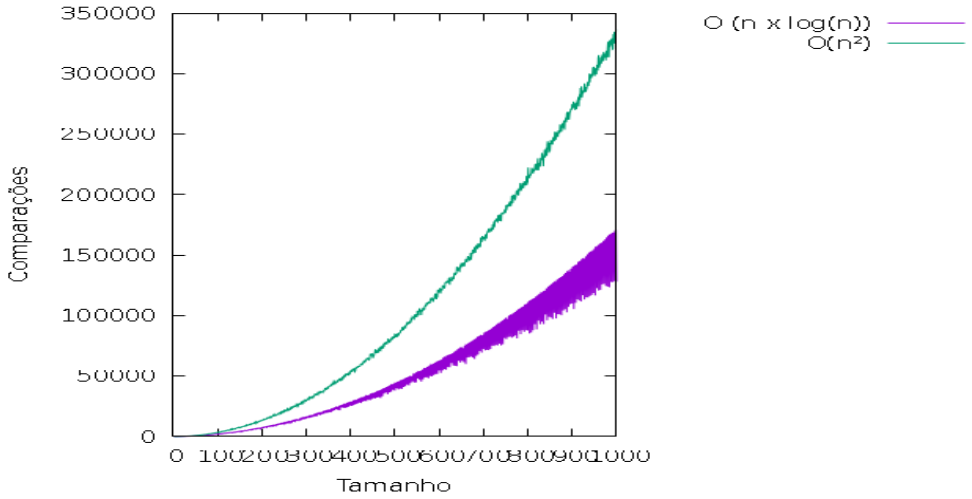


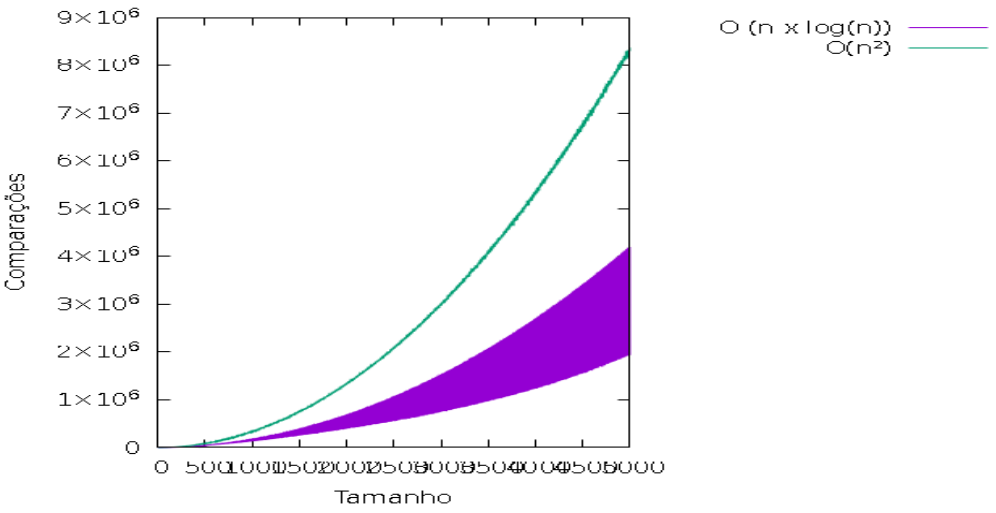
Esses gráficos também foram gerados naquele padrão visto anteriormente(P,M,G)
Ao observarmos esses gráficos é perceptível que o quick Sort possui uma grande variância comparado aos outros sorts e o Merge Sort possui uma grande eficiência por causa da sua estratégia de divisão, e a tendência de diferença entre ele e os demais sort tende a aumentar conforme o tamanho do vetor.

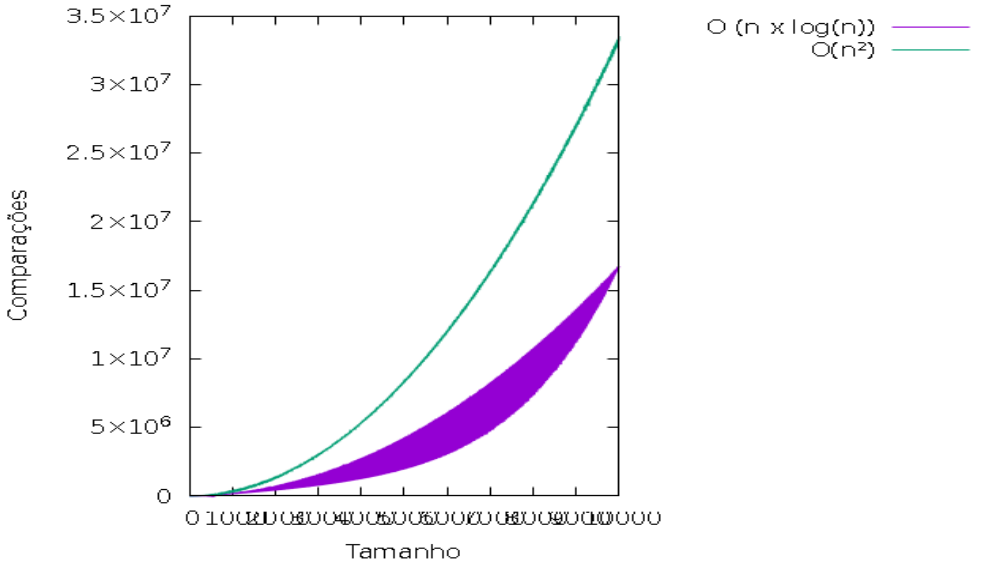
Grupo 1 x Grupo 2

Com base nos ultimos dados foi feita uma média de comparações dentro de cada grupo e os resultados obtidos foram:

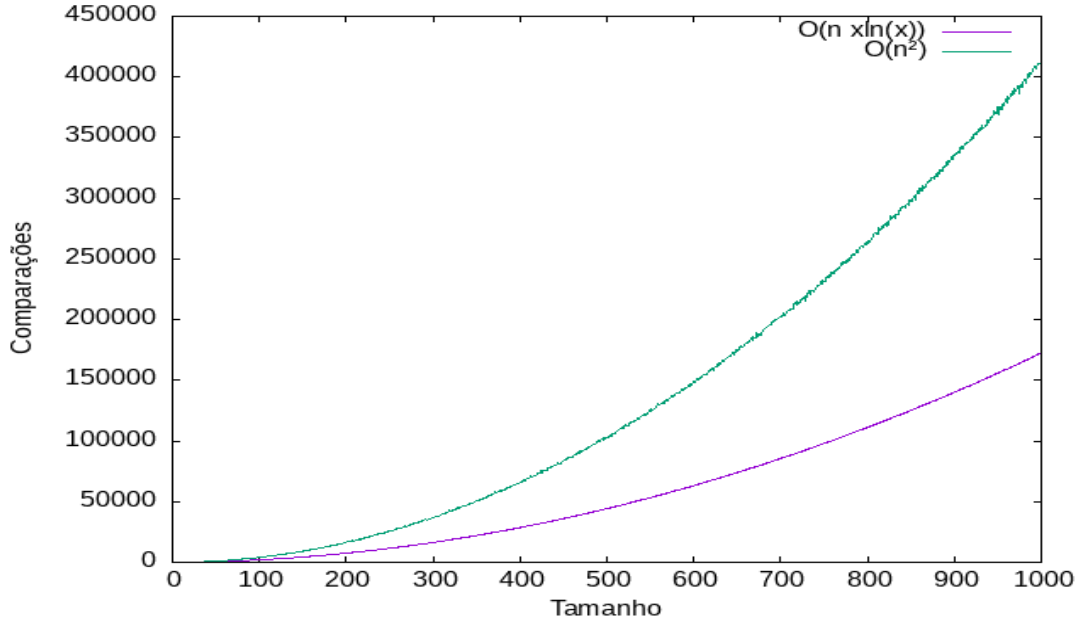
Imagens do Windows:

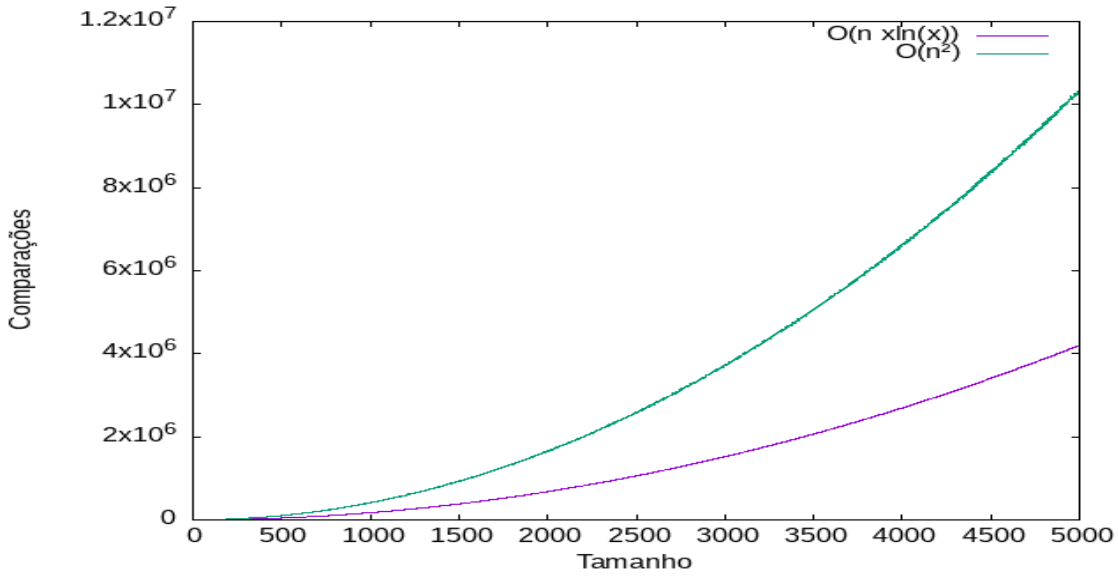


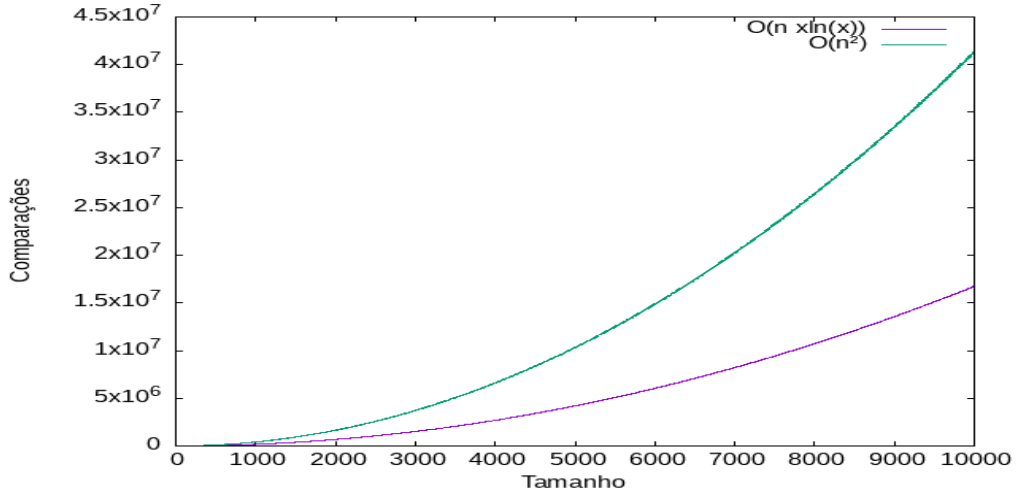

O (n x log(n))
O(n²)
Comparações
9×10⁶
8×10⁶
7×10⁶
6×10⁶
5×10⁶
4×10⁶
3×10⁶
2×10⁶
1×10⁶
0
Tamanho


O (n x log(n))
O(n²)
Comparações
3.5×10⁷
3×10⁷
2.5×10⁷
2×10⁷
1.5×10⁷
1×10⁷
5×10⁶
0
Tamanho

Imagens do linux:





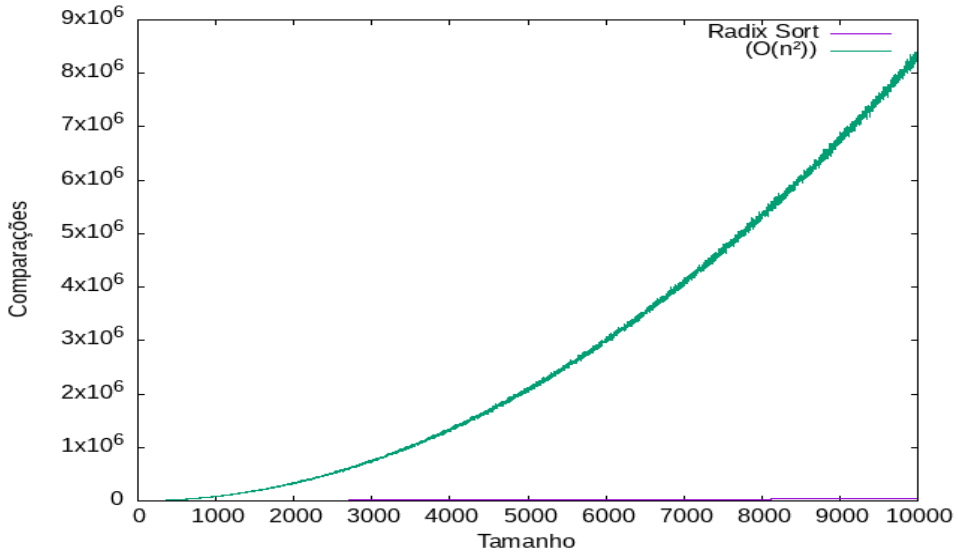


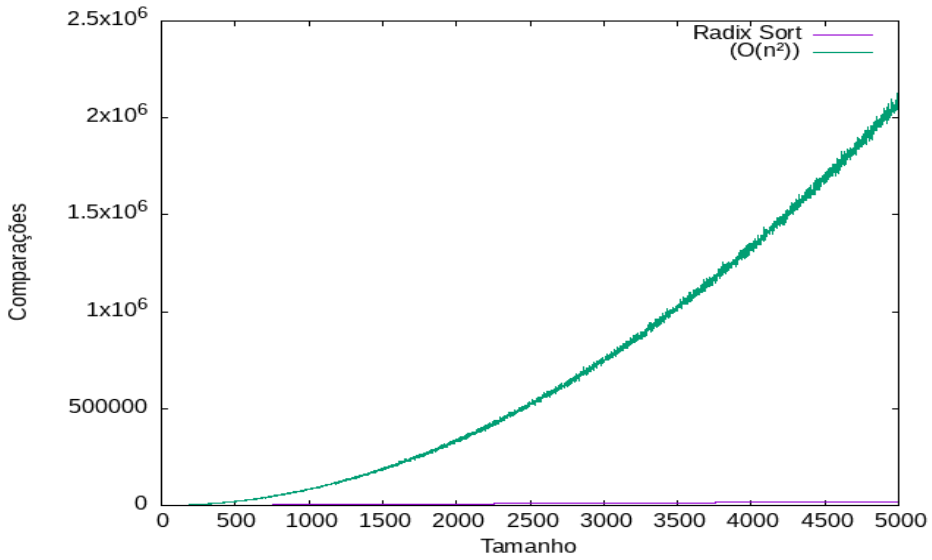
Observamos que há uma diferença nos gráficos gerados do windows e do Linux, porém ambos ainda mostram a grande eficiência das funções do Grupo 2, tanto para pequenos

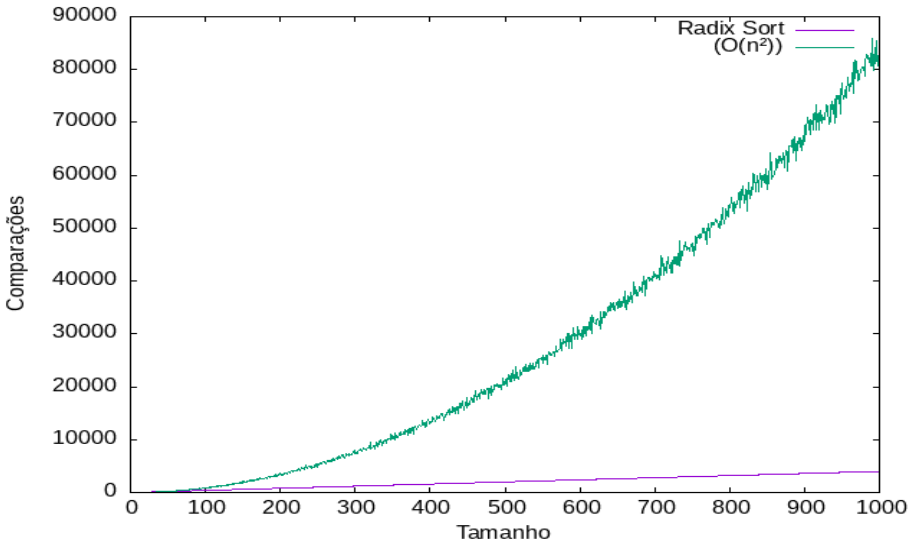
vetores quanto para grandes vetores, é importante ressaltar que o impacto que a quick sort faz na média do grupo 2 é muito mais aparente no windows.
Podemos concluir também que programas podem se tornar muito mais lentos se utilizarem grande números de dados juntamente as funções de O(n²), saber atribuir a função de ordenação certa melhorará a aplicação concerteza.

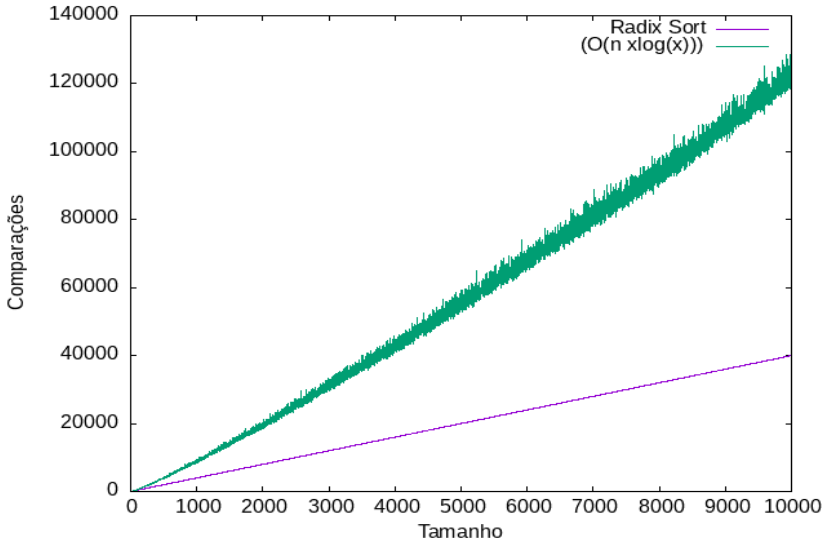
Radix Sort

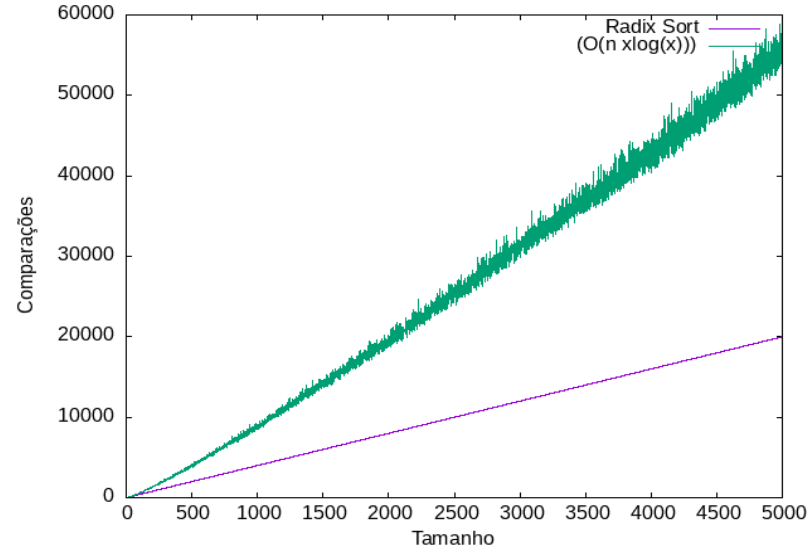
Essa função de ordenação é representada pela fórmula O(w + n), no qual ele ordena com base nos números de digitos de cada um.

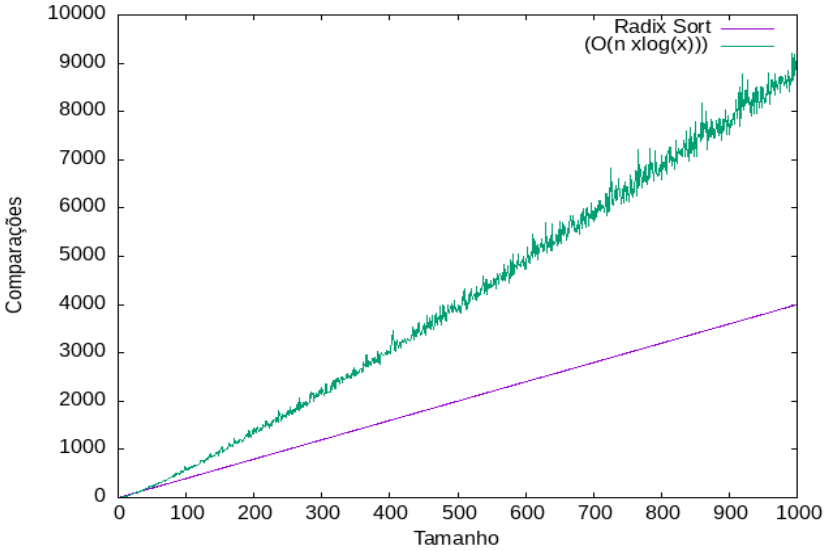





Radix Sort
(O(n²))
Comparações
Tamanho
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
0
100
200
300
400
500
600
700
800
900
1000


Radix Sort
(O(n xlog(x)))
Comparações
Tamanho
0
20000
40000
60000
80000
100000
120000
140000
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000





Uma característica interessante do Radix Sort é que ele é aparentemente uma função linear, separei entre Radix **X** O(n xlog(x)) e Radix **X** (O(n²)). independente de seu tamanho ele continua linear.

Conclusão

Podemos concluir que o uso das funções mais simples com complexidade O(n²) é útil apenas para baixos volumes de dados. Conforme o volume de dados aumenta, é recomendado utilizar funções com complexidade O(n log n), como por exemplo, as funções de ordenação baseadas em comparação. Por outro lado, em casos específicos em que a ordem dos números depende apenas da quantidade de dígitos, pode ser mais eficiente utilizar uma função de complexidade O(w + n).Essa abordagem mantém os números organizados de forma eficiente com base em sua quantidade de dígitos.