UDESC UNIVERSIDADE DO ESTADO DE SANTA CATARINA | JOINVILLE CENTRO DE CIÊNCIAS TECNOLÓGICAS

CAL0001

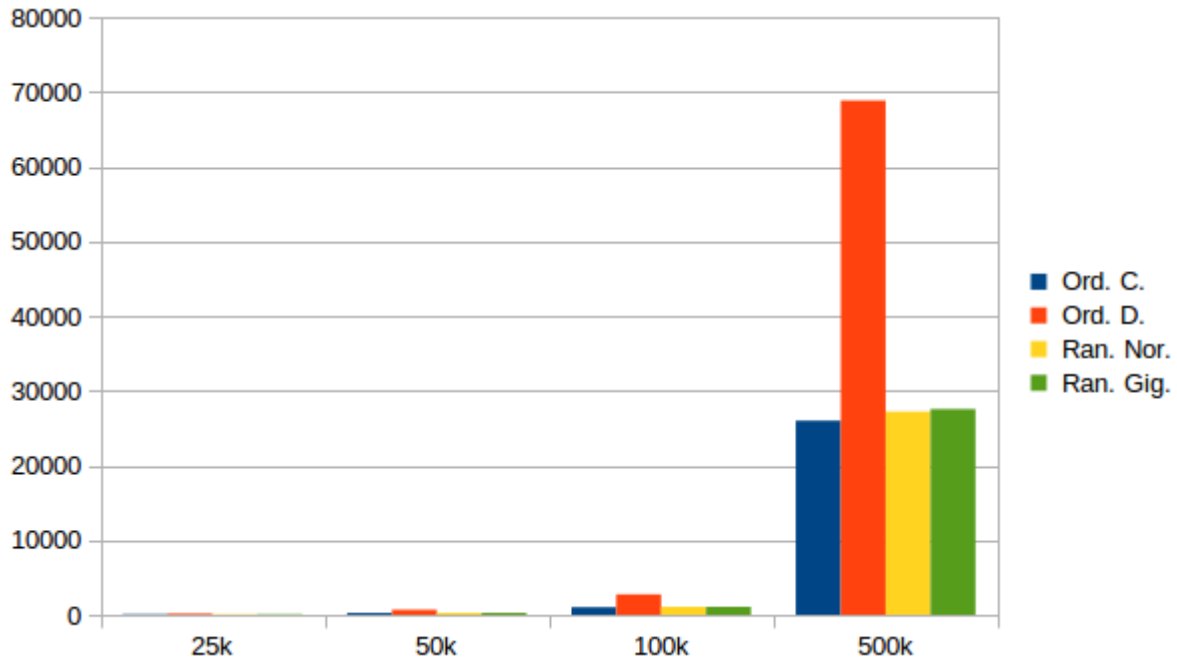
COMPLEXIDADE DE ALGORITMOS

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ACADÊMICO MARLON HENRY SCHWEIGERT

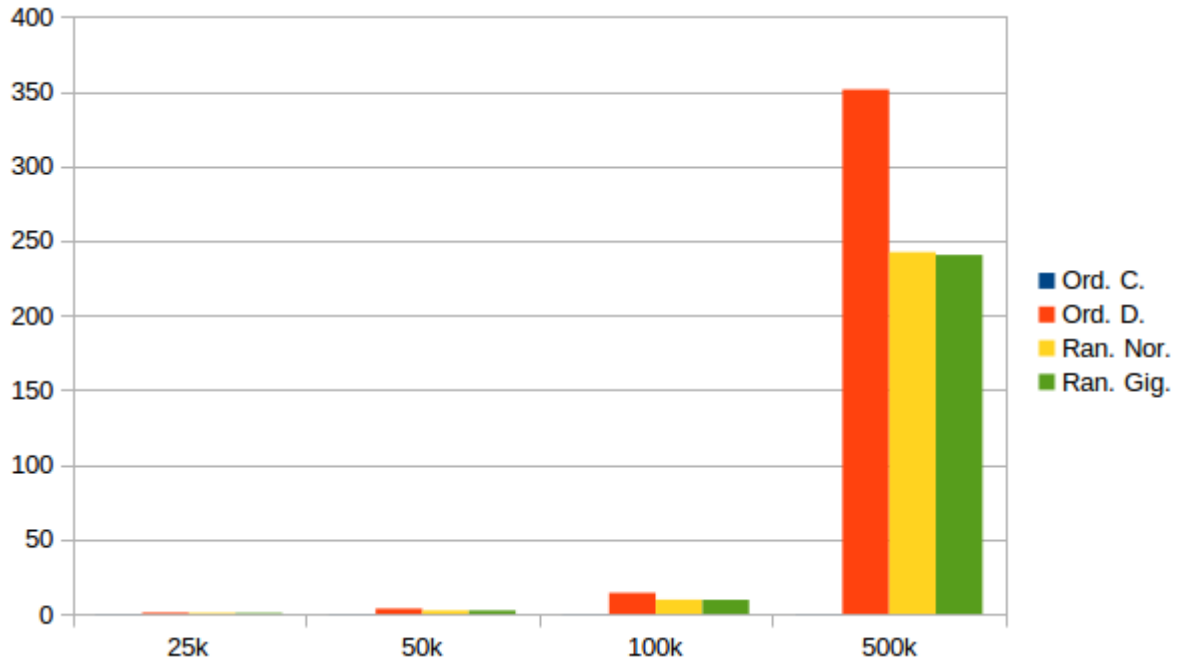
BUBBLE SORT

Este algoritmo trabalha com comparações diretas entre todos os objetos $O(N^2)$. Por este motivo, o modelo decrescente é o mais lento visto que precisa trocar todos os elementos comparados.



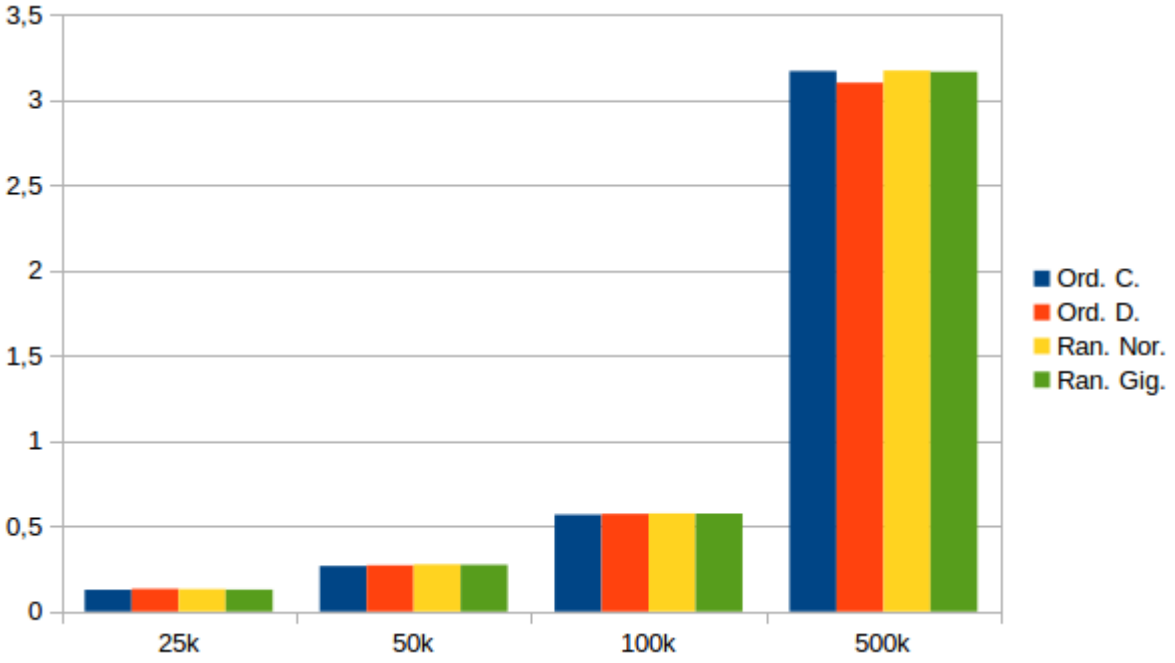
INSERT SORT

Analisa o nó e insere ele no local correto da lista. Para listas quase ordenadas, torna-se muito eficiente, com complexidade Ω(N). Porém, para listas reversas, o seu tempo de execução também é ruim obtendo a complexidade igual ao Bubblesort, sendo $O(N^2)$.

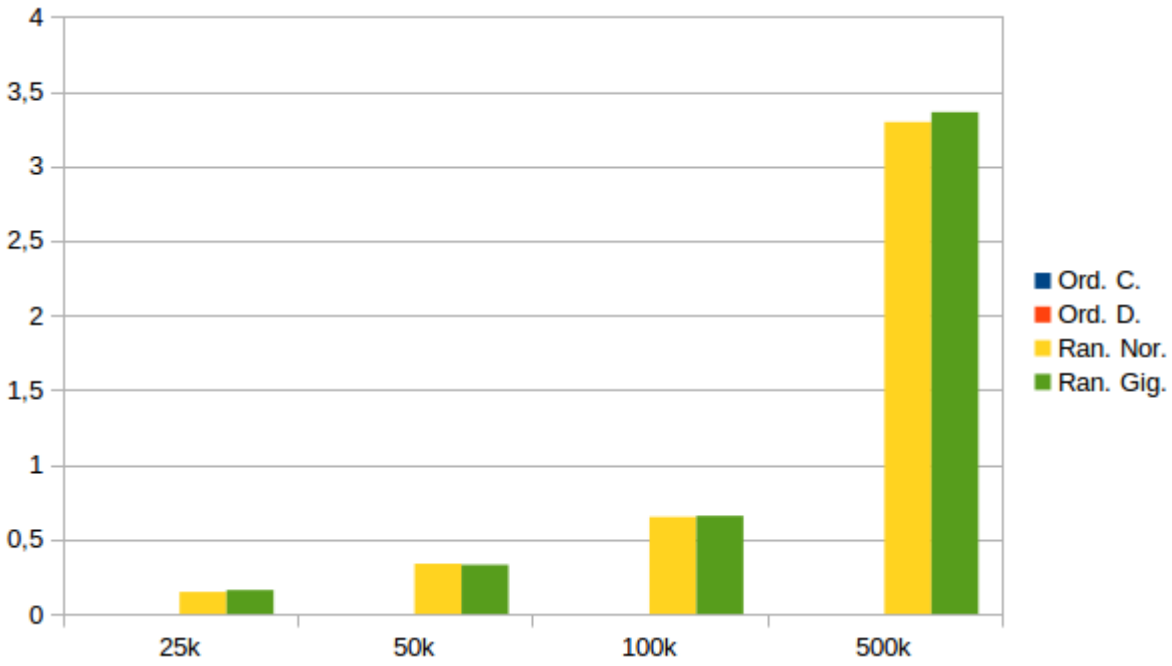


QUICK SORT

Divide de forma recursiva o vetor para ordenar os dados. Não necessita comparar todos com todos por este motivo. A seleção do pivô para divisão da lista é o grande problema do Quicksort. Caso seja selecionado um pivô descentralizado, a divisão será pouco eficiente, tornando o algoritmo de ordem quadrática. Já para os melhores casos, teremos Ω(n.log(n)).

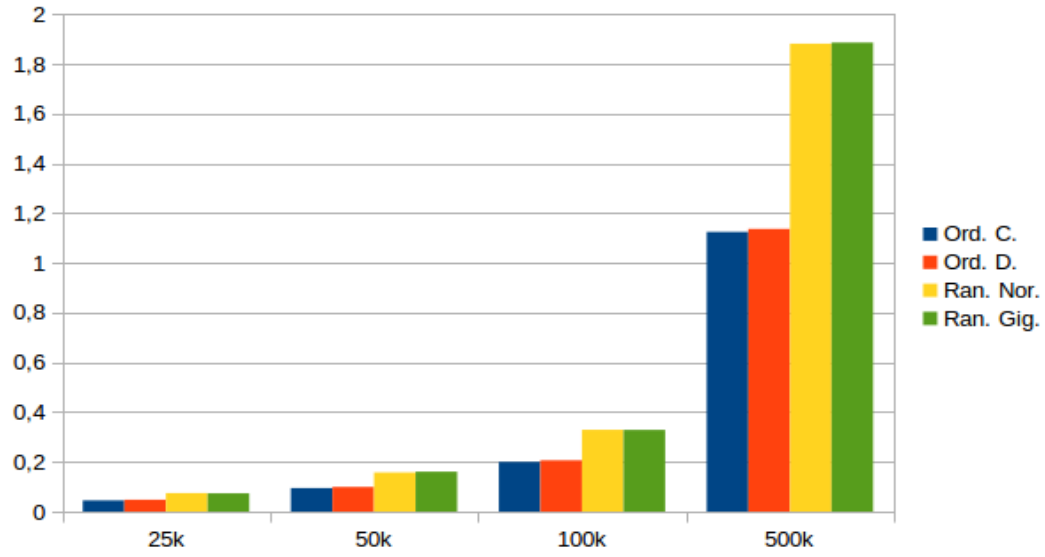


Por comparativo, temos o QuickSort com o pivô sendo o primeiro elemento da lista. Dessa forma, a quantia de chamadas recursivas aumenta absurdamente, tornando o algoritmo em $O(n^2)$. Nos casos onde a recursão aumenta drasticamente, o interpretador estoura a pilha de recursão, não contabilizando tempo.



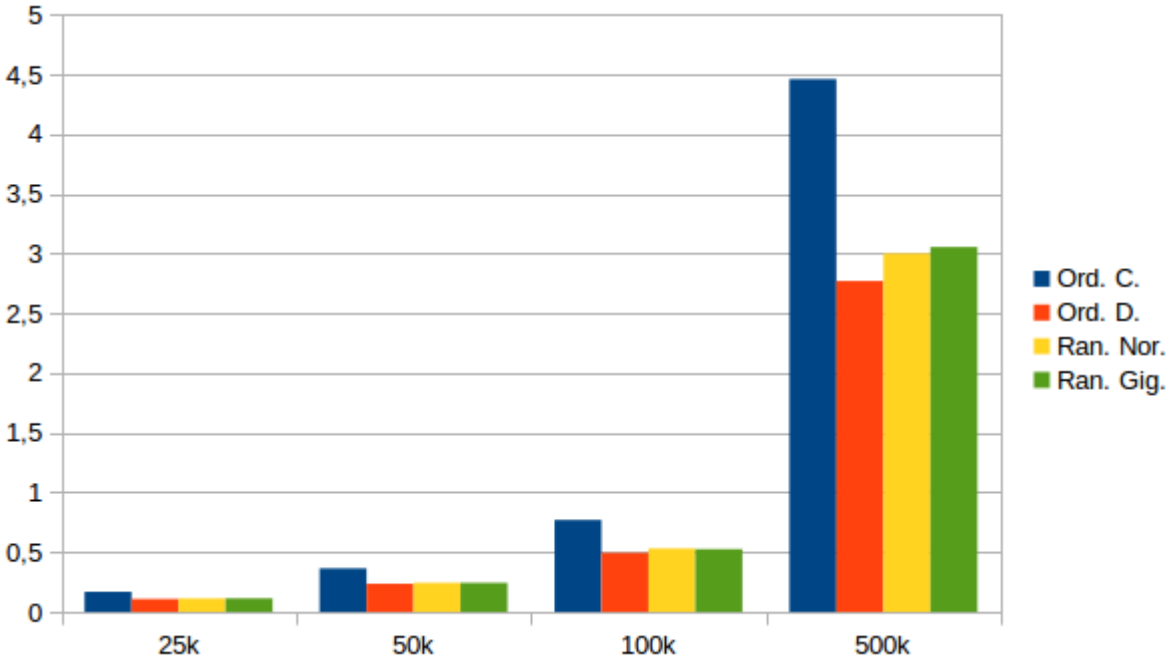
MERGE SORT

Também trabalha com a ideia de divisão para conquistar, porém resolve o problema do pivô encontrado no Quicksort. Ele ordena os elementos por comparação a duas listas já ordenadas, tendo a eficiência do InsertionSort para vetores já ordenados. A sua complexidade é Ω(n.log(n)).



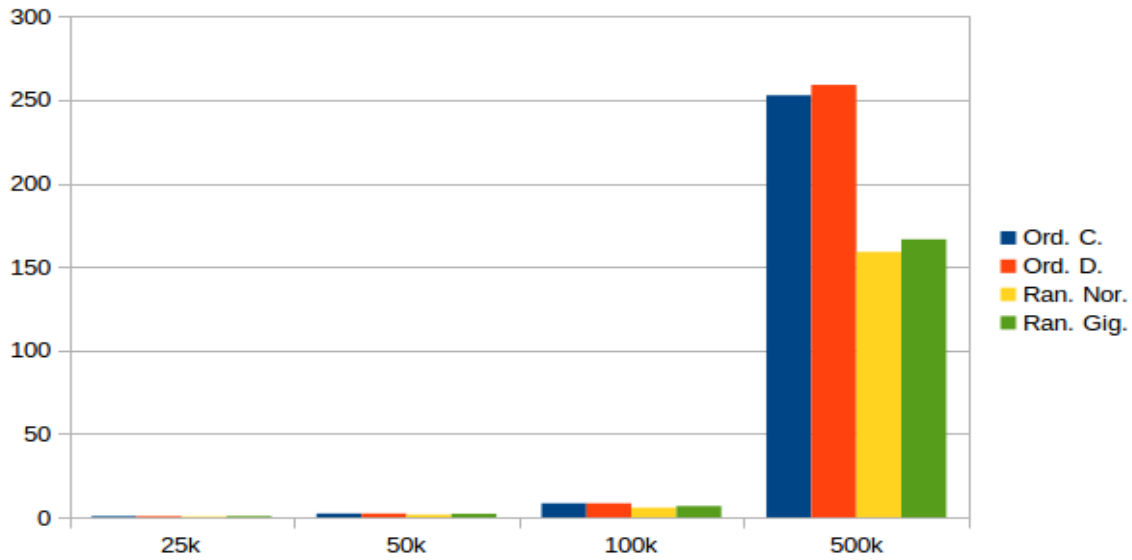
HEAP SORT

Utiliza a lógica da normalização de árvores binárias para elevar os menores valores ao topo, e deixar os maiores valores nas folhas da árvore. Dessa forma, o pior caso é onde o vetor já está ordenado, visto que o número de movimentos cresce consideravelmente. Além disso, ele possui um tempo adicional para a construção inicial (linear), a qual o merge e quick não possuem. Sua complexidade também é O(n.log(n)). Em objetos já ordenados, a quantia de trocas na árvore torna o algoritmo mais custoso em tempo, mas não prejudica sua complexidade.

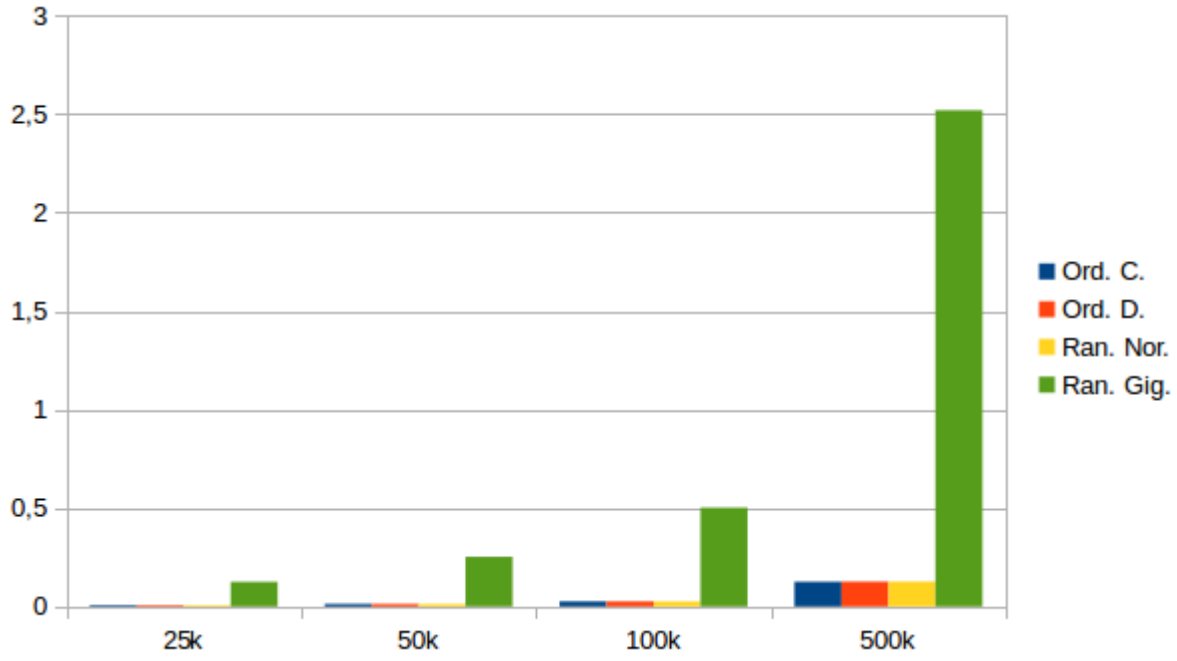


COUTING SORT

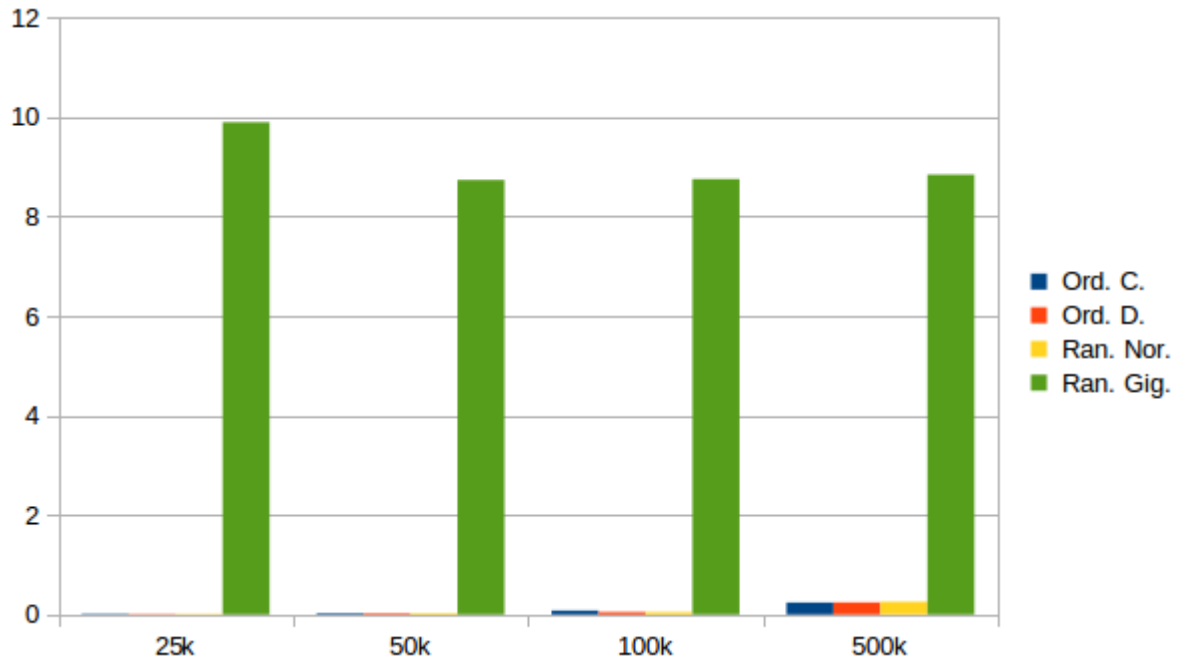
Na linguagem ruby, tornou-se inviável recriar a lista para a resposta. Este método mostrou que a complexidade da criação da lista é muito maior comparado ao algoritmo couting sort.



Para compensar, criamos um algoritmo a qual não gera uma nova lista conforme os valores obtidos da contagem. O resultado final aplicado demonstra-se esperado conforme o estudo em sala de aula, onde sua complexidade é $\Omega(N)$.



Em um caso extremo, onde o valor seja de 100000000 no teste Rând. Gigante, teremos os seguintes tempos:

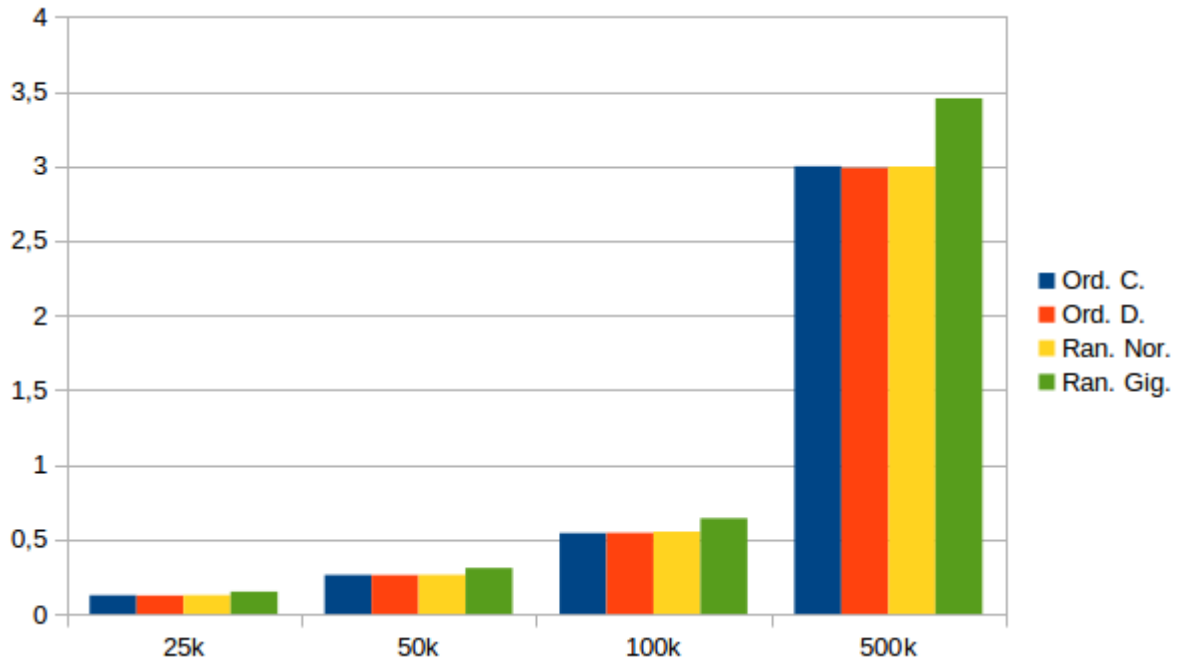


BUCKET SORT

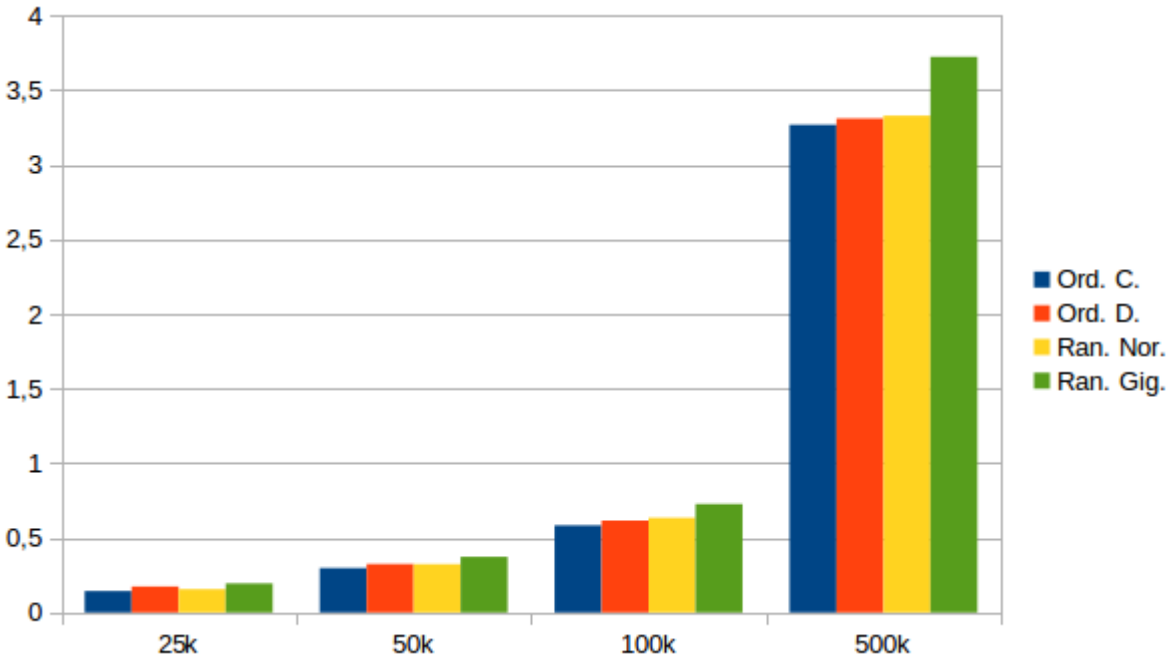
O bucket sort tornou-se mais eficiente que o quick sort (o qual foi utilizado para ordenar os baldes). A sua complexidade tornou-se linear para números bem distribuídos, porém para números mal distribuídos, mostra-se ruim visto que criará buckets com valores distribuídos de forma não uniforme. Fora do seu pior caso, o BucketSort tornou-se mais eficiente que o subalgoritmo utilizado, o QuickSort.

O número de buckets utilizados é 3.3*log(n), onde n é o tamanho do vetor. 3.3*log(n) é o tamanho de buckets utilizado no sort pela linguagem Lua, dado que este número é utilizado na estatística para obter um número adequado de amostras de uma população finita.

A complexidade do algoritmo Bucket é Ω(N), sendo o algoritmo de ordenação interno é O(n.log(n)) (quicksort).



Para testar o pior caso do BucketSort, colocamos no caso do Rând. Gigante, o maior valor de 100000000. Os tempos obtidos foram:



Nesse caso, a sua complexidade fica O(n.log(n)) igual ao subalgoritmo utilizado, tornando-se menos eficiente que o Quicksort.