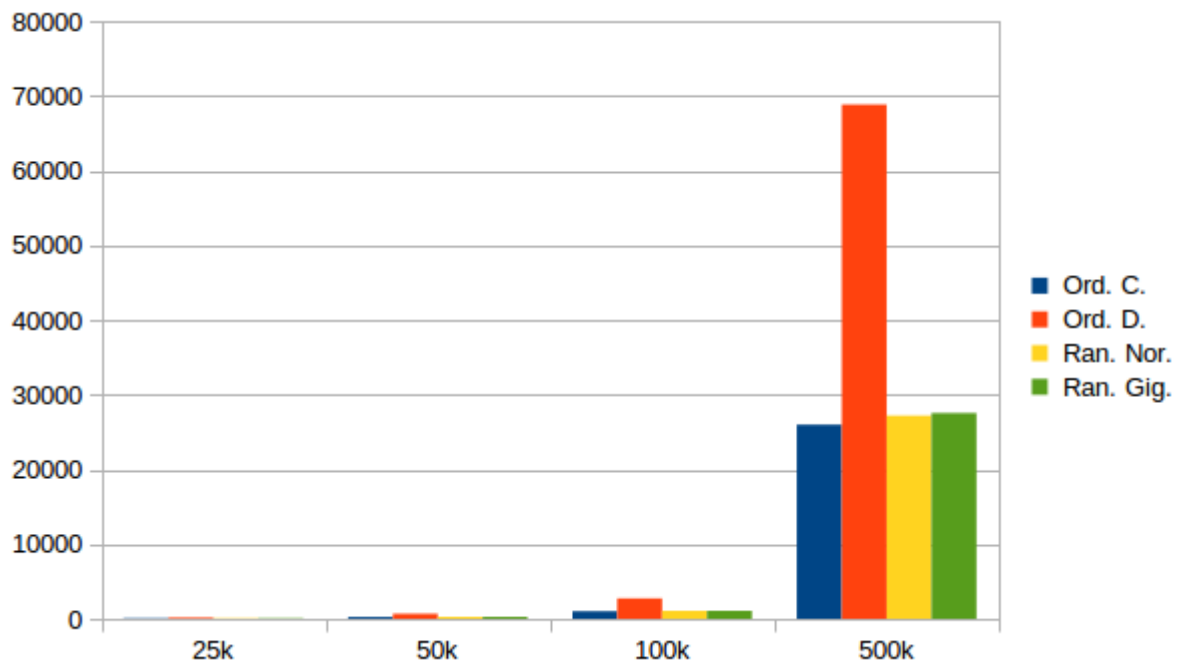


CAL0001
COMPLEXIDADE DE ALGORITMOS
UNIVERSIDADE DO ESTADO DE SANTA CATARINA
CENTRO DE CIÊNCIAS TECNOLÓGICAS

ACADÊMICO MARLON HENRY SCHWEIGERT

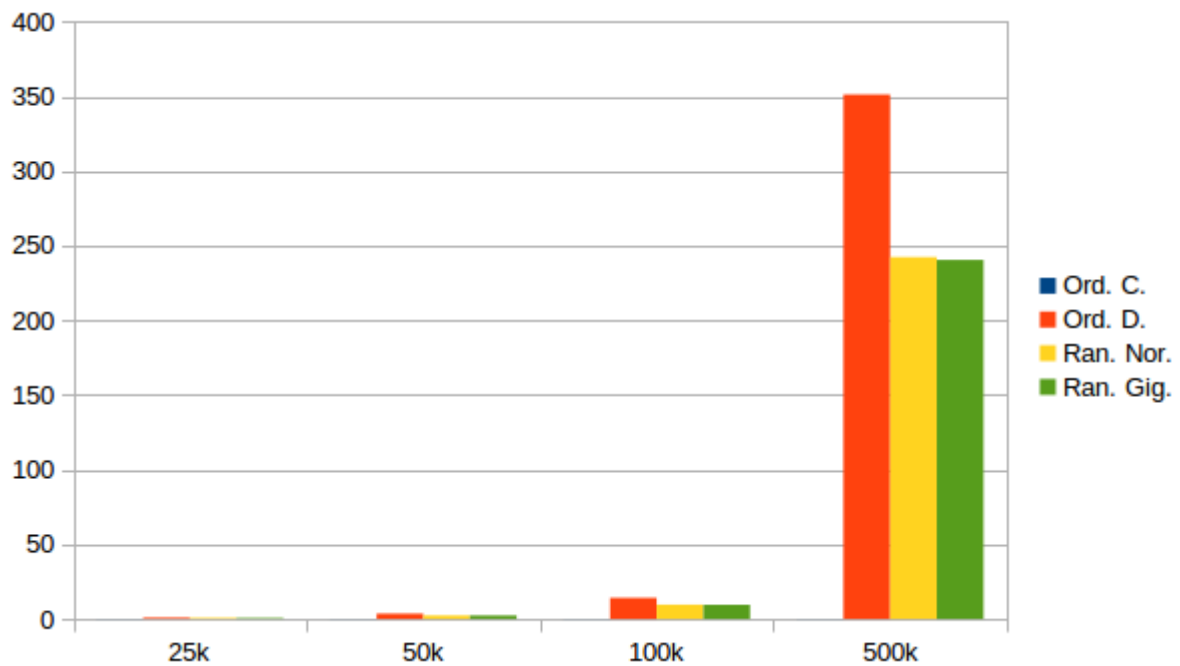
BUBBLE SORT

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.



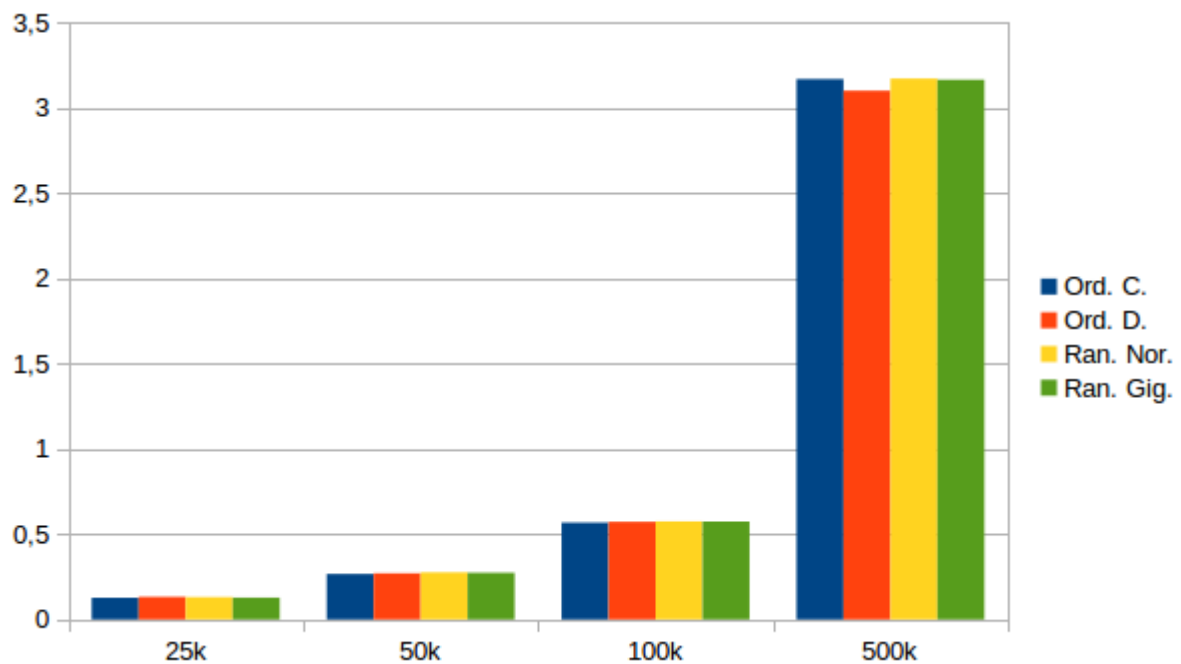
INSERT SORT

Analisa o nó e insere ele no local correto da lista. Para listas quase ordenadas, torna-se muito eficiente, com complexidade $O(N)$. Porém, para listas reversas, o seu tempo de execução também é ruim comparado a outros métodos de ordenação.

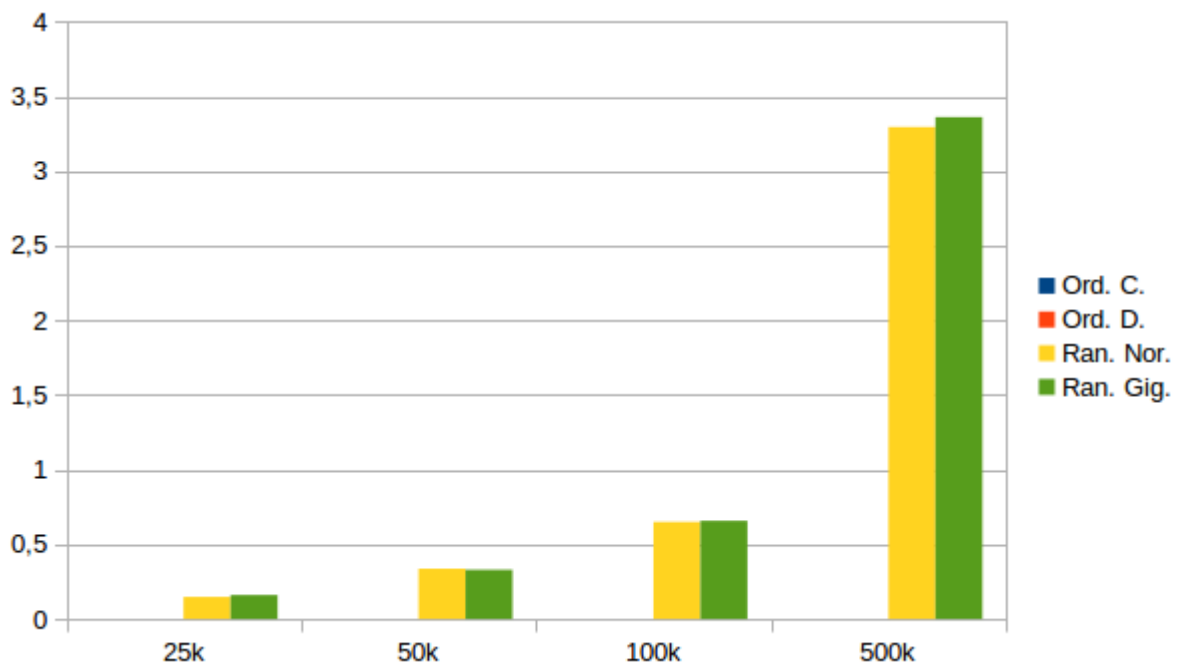


QUICK SORT

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 muito pequeno, a divisão será pouco eficiente, tornando o algoritmo de ordem quadrática. Já para os melhores casos, teremos $O(n \cdot \log(n))$.

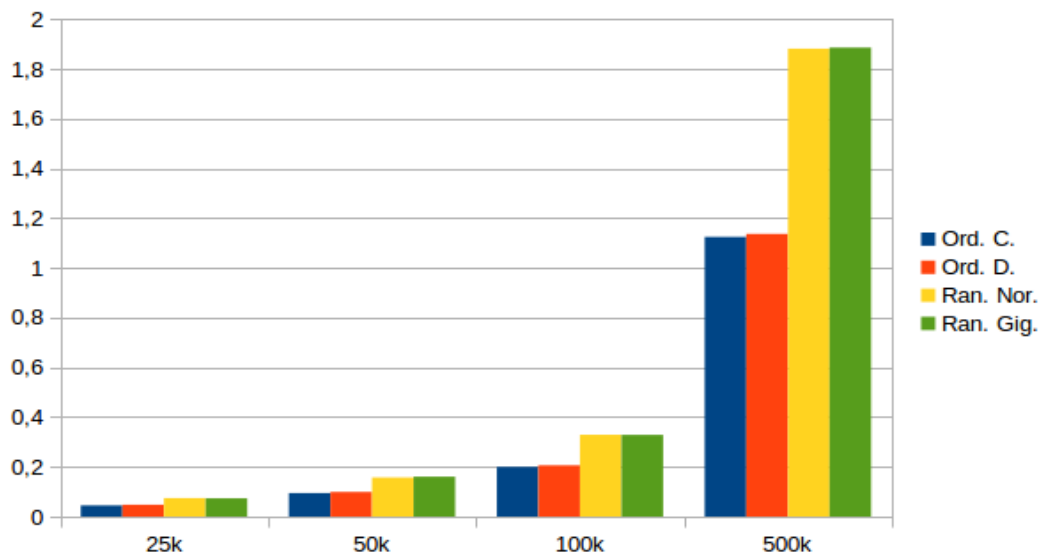


Para o caso ruim, como no gráfico seguinte, temos o primeiro elemento da lista como pivô. Isso faz nos casos dos vetores já ordenados ou até mesmo próximo de ordenação estourarem o limite da pilha. Além disso, os demais casos demoraram consideravelmente mais do que o caso onde o pivô era selecionado randomicamente.



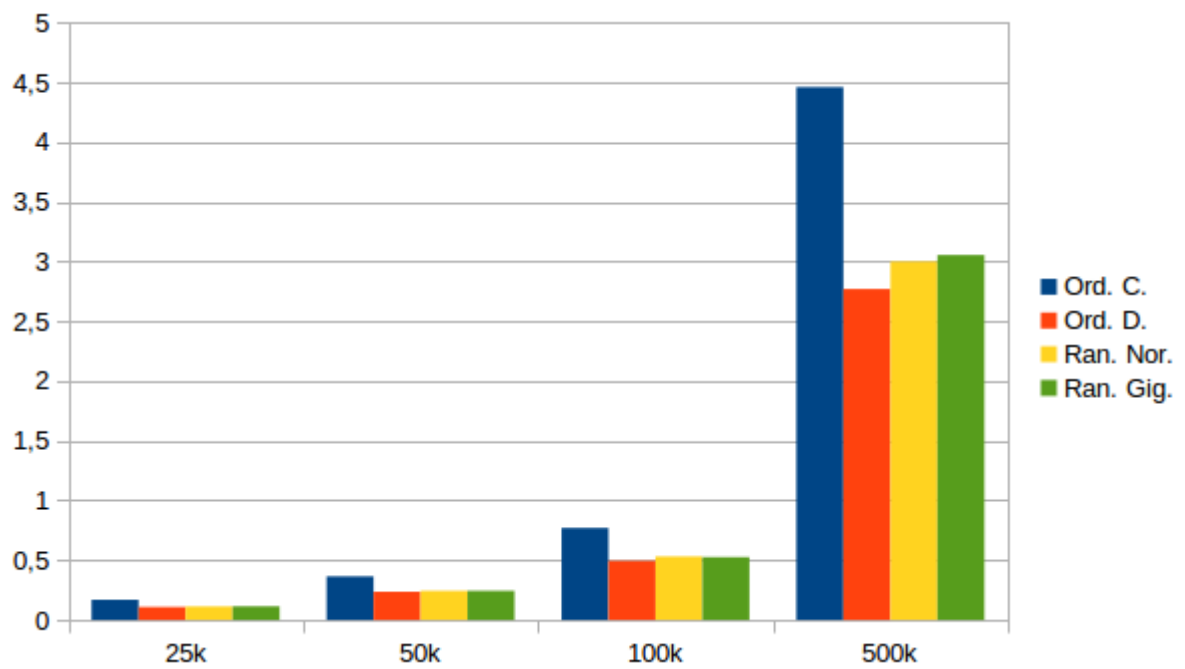
MERGE SORT

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 $O(n \cdot \log(n))$.



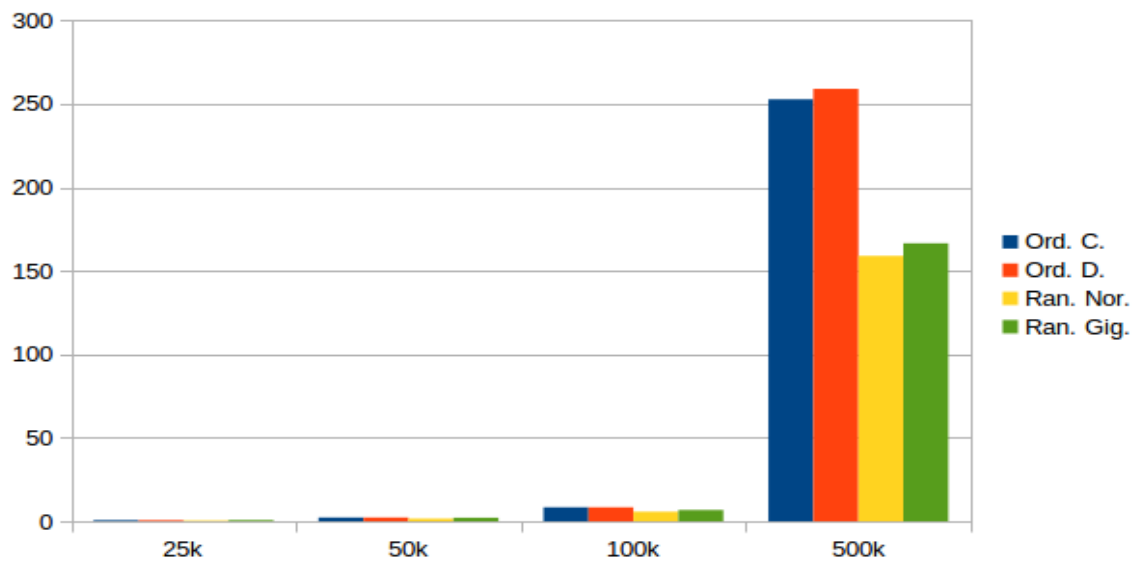
HEAP SORT

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 $O(n \cdot \log(n))$.



COUTING SORT

O algoritmo tornou-se instável utilizando a linguagem Ruby visto que se utiliza listas, e não vetores para armazenar todos os dados.



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 muitos valores e outros com poucos valores. O tempo em qualquer caso, ficou mais eficiente que o Quicksort puro, utilizado abaixo.

O número de buckets utilizados é $3.3 \cdot \log(n)$, onde n é o tamanho do vetor. $3.3 \cdot \log(n)$ 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 $O(N)$, sendo o algoritmo de ordenação interno é $O(n \cdot \log(n))$ (quicksort).

