PUC Minas – Engenharia de Software – Noturno
Laboratório de Computação II – Felipe Cunha
José George Melgaço Souza - 529264

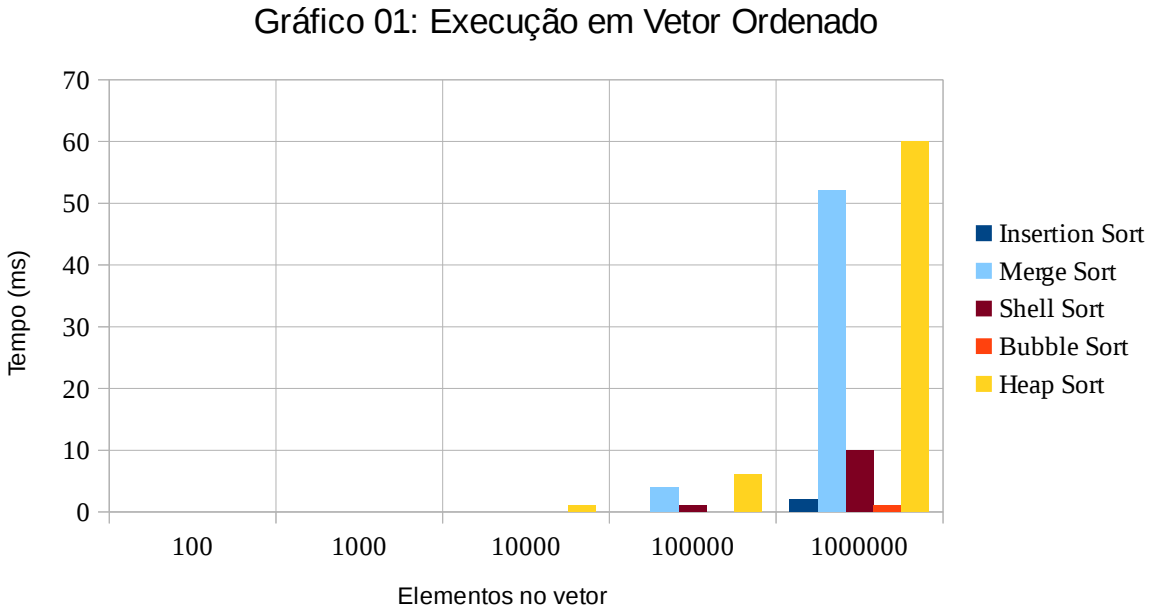
Trabalho Prático 03

O trabalho consiste da análise da eficiência em tempo de execução de sete algoritmos de ordenação básicos na ciência da computação, sendo eles: *Insertion Sort*, *Selection Sort*, *Merge Sort*, *Shell Sort*, *Bubble Sort*, *Heap Sort* e *Quick Sort*.

As implementações usadas baseiam-se nos algoritmos básicos de cada método de ordenação, sem melhorias adicionais, com exceção do *Quick Sort*, no qual foi usado o esquema de particionamento proposto por C.A.R. Hoare ao invés do método proposto por Nico Lomuto[i]. Todas as implementações usam a linguagem de programação Java versão 8.

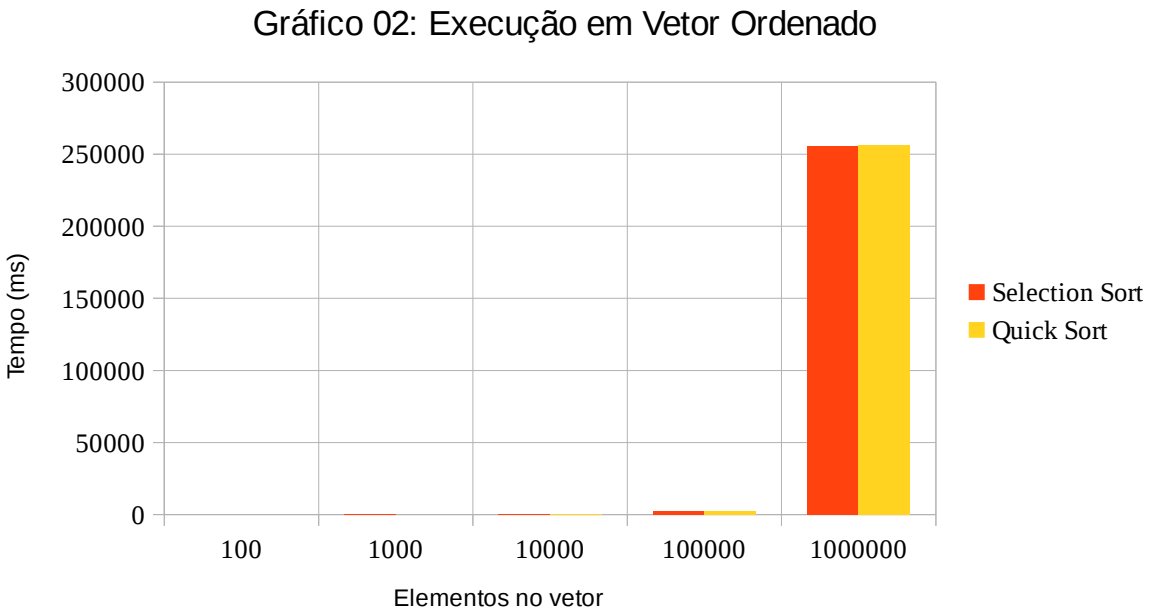
Os dados de entrada são vetores de números inteiros positivos com 100, 1 k, 10 k, 100 k e 1 M elementos, sendo que cada vetor é clonado sete vezes a fim de que todos os algoritmos de ordenação possam ser comparados em relação à mesma entrada. Além, são usados três classes de vetores, sendo elas: 1. vetores pseudo-aleatórios, ou seja, com elementos desordenados; 2. vetores com elementos ordenados; 3. vetores com elementos inversamente ordenados. Tais classes visam cobrir os melhores casos, os casos médios e os piores casos da maioria dos sete algoritmos. Para cada ordenação, o tempo gasto, em *ms*, é computado e arquivos *.tsv* são criados com os dados.

Abaixo, seguem seis gráficos com os dados referentes a todos os métodos de ordenação para as três classes de vetores definidas anteriormente. Para cada gráfico, seguem comentários que comparam os resultados de cada algoritmo em função do número de comparações e de seu caso de complexidade.



O Gráfico 01: Execução em Vetor Ordenado mostra que, quando a entrada de dados já está ordenada, tem-se o melhor caso dos algoritmos *Insertion Sort*, *Merge Sort*, *Shell Sort*, *Bubble Sort* e *Heap Sort*. Em relação ao *Bubble Sort* e *Insertion Sort*, ambos não fazem trocas com os elementos do vetor, havendo gasto mínimo relacionado ao gasto básico de percorrer o vetor uma única vez e comparar os elementos dois a dois no processo, complexidade *O(n)*. O *Shell Sort*, sendo uma variação do *Insertion Sort*, tem um custo um pouco mais elevado que o último, pois o vetor é comparado em camadas, havendo a necessidade de percorrer o vetor mais vezes, complexidade

O(nlogn). *Merge Sort* e *Heap Sort* possuem complexidade *O(nlogn)* para qualquer caso, como é mostrado no Gráfico 01: Execução com Vetor Ordenado e nos demais gráficos, sendo seus custos fixos os de divisão e posterior fusão do vetor para o primeiro algoritmo e o de construção da estrutura *heap* e sua quebra e correção *n* vezes para o segundo algoritmo.



O Gráfico 02: Execução em Vetor Ordenado mostra os resultados dos algoritmos *Selection Sort* e *Quick Sort* separadamente dos demais. Para o *Selection Sort* não existe diferenciação entre o pior, melhor e caso médio: todos tem complexidade quadrática $O(n^2)$ independente do estado dos dados de entrada, sendo considerado o pior algoritmo para grandes quantidades de dados. Diferentemente, apesar do *Quick Sort*, dependendo da implementação, ser mais rápido que o *Heap Sort*, essa rapidez só é verificada para entradas desordenadas com complexidade *O(nlogn)* como é mostrado nos gráficos para resultados com vetores desordenados. Na situação atual, tem-se o pior caso caso do *Quick Sort* que é para entradas ordenadas ou inversamente ordenadas, o que obriga o algoritmo a fazer o máximo de partições quando a escolha do pivot é feita no primeiro ou último elemento de cada subseção do vetor (maior ou menor), havendo partições de tamanhos desproporcionais, como *0* e *n – 1*. Para o pior caso, a complexidade do *Quick Sort* é $O(n^2)$.

Assim, para uma entrada com elementos ordenados ascendentemente, o melhor método de ordenação é o *Bubble Sort*. Entretanto, como esse caso é bastante raro, não é recomendado o seu uso, havendo algoritmos muito melhores para o caso médio.

No Gráfico 03: Execução em Vetor Inversamente Ordenado, tem-se novamente o pior caso do *Quick Sort* com complexidade $O(n^2)$, o pior caso do *Insertion Sort* com o máximo de comparações e trocas com complexidade de $O(n^2)$ *e* também o pior caso do *Bubble Sort*, no qual o vetor inversamente ordenado obriga o algoritmo a fazer o máximo de trocas. Assim, nesse caso, apesar de ter a mesma complexidade que o *Insertion Sort*, o *Bubble Sort* realiza mais trocas, gastando quase o triplo do tempo do *Insertion Sort* para grandes quantidades de elementos.

Observando o Gráfico 04: Execução em Vetor Inversamente Ordenado, tem-se os algoritmos com melhores resultados. Como já dito, *Heap Sort* e *Merge Sort* tem complexidade assintótica única para todos os casos de *O(nlogn)*, sendo que o *Merge Sort* apresentou o melhor resultado para esse caso. O *Shell Sort*, usando a distribuição de tamanho de camas proposta por Ciura, Marcin (2001)[ii], não tem uma complexidade assintótica definida para o pior caso. Tal complexidade depende do tamanho escolhido para as camadas. Entretanto, considerando que o algoritmo levou cerca de 14 vezes mais tempo que o *Merge Sort*, pode-se considear que este é o seu pior caso com

complexidade assintótica próxima de $O(n\log_2^2 n)$. Isso se deve ao fato do *Shell Sort* realizar o máximo de trocas em um vetor inversamente ordenado.

Gráfico 03: Execução em Vetor Inversamente Ordenado

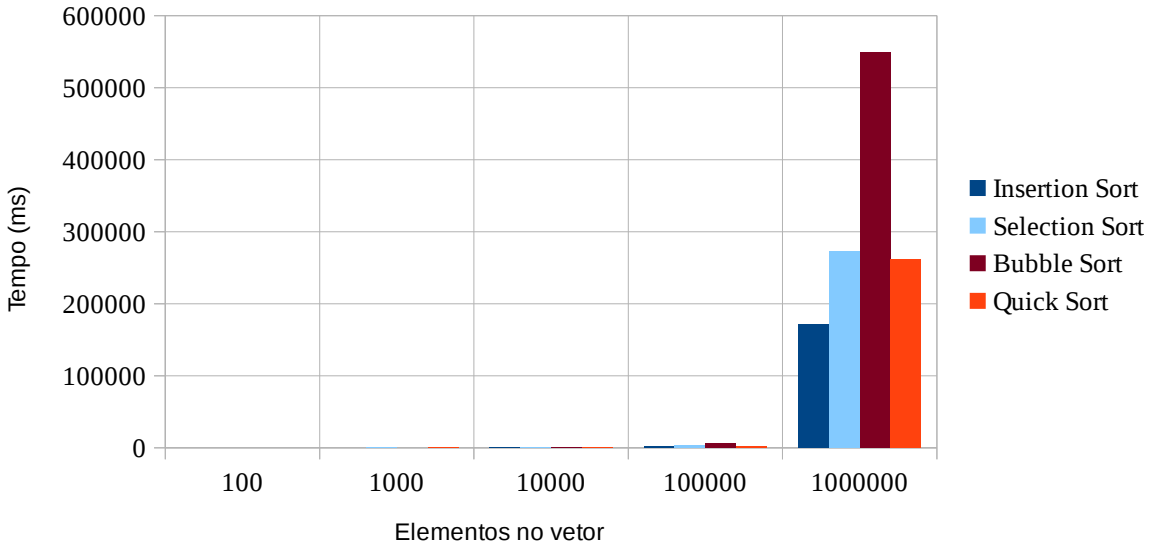
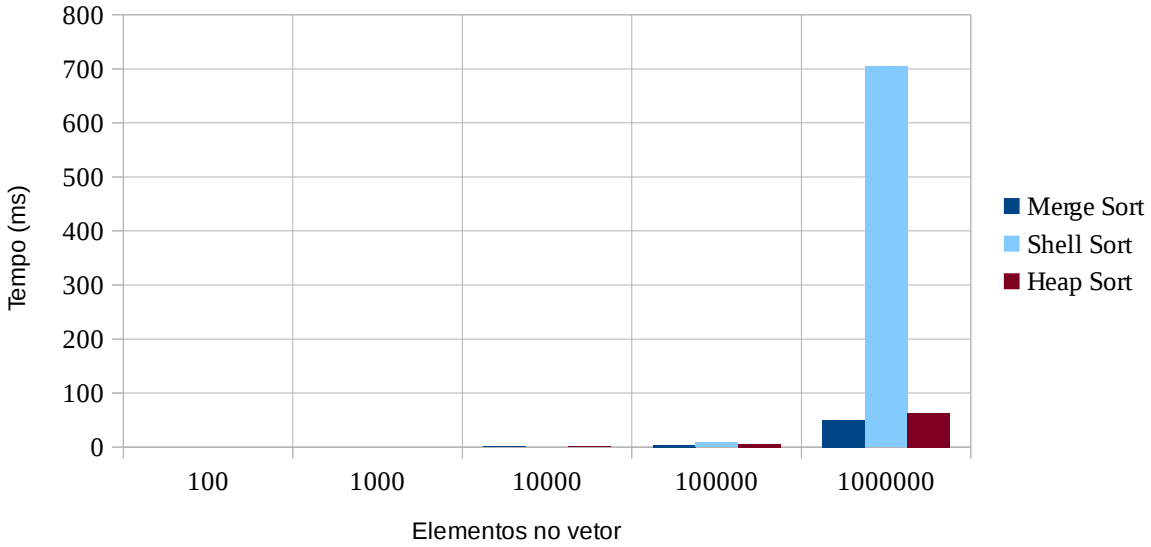


Gráfico 04: Execução em Vetor Inversamente Ordenado



Finalmente, para o caso do vetor desordenado, tem-se os gráficos 05 e 06. No Gráfico 05: Execução em Vetor Desordenado, tem-se o caso médio do *Insertion Sort* com complexidade de $O(n^2)$ e o caso médio do *Bubble Sort*, também com complexidade de $O(n^2)$. Entretanto, o gráfico mostra um gasto exagerado de tempo, mesmo não sendo o pior caso. Algumas fontes não acadêmicas encontradas na internet dizem que o *Bubble Sort* tem uma interação ruim com CPUs modernas, produzindo mais escritas que o *Insertion Sort*, mais erros de cache e mais erros de predição de caminhos condicionais no *hardware*. Entretanto, fontes confiáveis não foram encontradas. Conclui-se que o *Bubble Sort* é o pior algoritmo para vetores desordenados, seguido do *Selection Sort* e *Insertion Sort*.

É no Gráfico 06: Execução em Vetor Desordenado que o *Quick Sort* mostra o seu poder e rapidez, sendo o melhor algoritmo para ordenação de entradas desordenadas, com complexidade de $O(n\log n)$. Como o melhor caso do *Quick Sort* depende da escolha do pivot e como a implementação

feita neste trabalho não considerou melhorias, possivelmente o melhor caso não é alcançado. Seguido pelo *Merge Sort* e *Heap Sort*, ambos com complexidade de *O(nlogn)* para qualquer caso, apesar de serem mais lentos que o *Quick Sort* para esse caso. Tem-se também o caso médio do *Shell Sort*, com complexidade assintótica de *O(nlogn)*, porém realizando mais comparações e trocas que seus concorrentes e, consequentemente, gastando mais tempo no processo de ordenação.

Gráfico 05: Execução em Vetor Desordenado

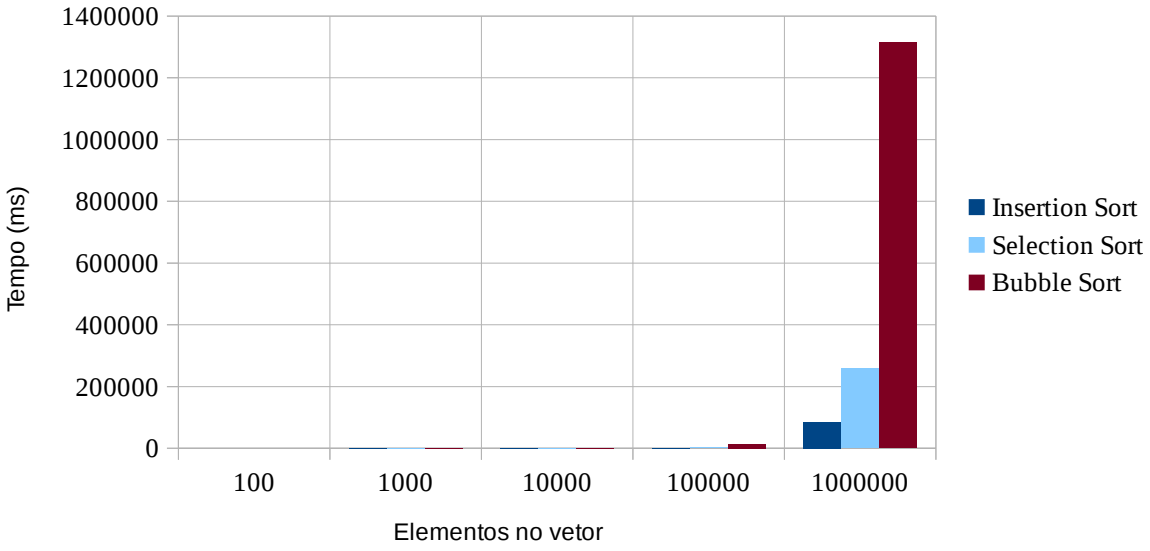
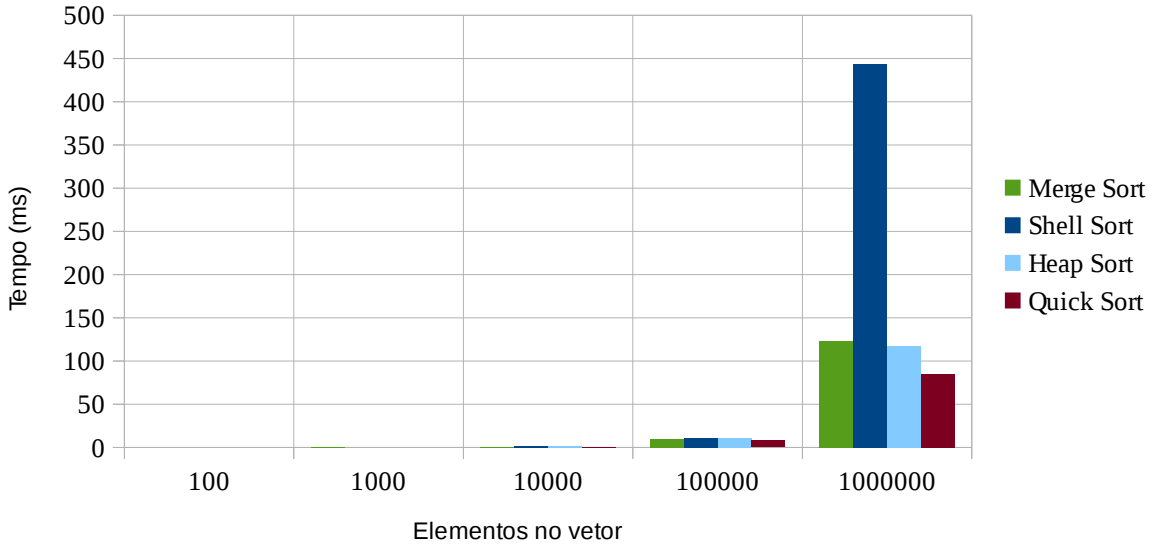


Gráfico 06: Execução em Vetor Desordenado



Conclui-se, assim, que dentre os métodos de ordenação analisados, o melhor algoritmo para o caso de entradas ordenadas ascendentemente é o *Bubble Sort* e o pior é o *Quick Sort*. Para o caso de entradas ordenadas descendentemente, o melhor é o *Merge Sort* e o pior é o *Bubble Sort*. Para o caso padrão com entradas desordenadas, o melhor é o *Quick Sort* e o pior é o *Bubble Sort*, seguido do *Selection Sort*.

i https://cs.stackexchange.com/questions/11458/quicksort-partitioning-hoare-vs-lomuto/11550#11550

ii Ciura, Marcin (2001). "Best Increments for the Average Case of Shellsort" (http://sun.aei.polsl.pl/~mciura/publikacje/shellsort.pdf). In Freiwalds, Rusins. Proceedings of the 13th International Symposium on Fundamentals of Computation Theory. London: Springer-Verlag. pp. 106–117. ISBN 3-540-42487-3.