UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
CIÊNCIA DA COMPUTAÇÃO

Trabalho I:
Projeto e Análise de Algoritmo

Juan Cardoso da Silva

Caio Nogueira

Presidente Prudente – SP
19/10/2018

Sumário

I: BubbleSort (Sem melhoria)

O processo de organização desse algoritmo , é fazer uma varredura no vetor e fazer com que os elementos mais "leves" "flutuem" para cima do vetor e os mais "pesados" "afundem" e cada elemento procuraria ficar em seu nível apropriado , em outras palavras é feito trocas por varredura até que o vetor esteja organizado por comparações do termo atual para o próximo termo.

Verificação de troca do código:

if(A[i] > A[i+1]).

• Complexidade do Algoritmo

A complexidade do BubbleSort não melhorado é apenas de $O(n^2)$ onde cada execução não é verificado qualquer tipo de organização de vetor, fazendo o custo de verificação ser grande e utilizar mais tempo do que devia.

Ao observar o gráfico abaixo, é notável a semelhança do formato das execuções devido a todas elas serem curvas quadráticas, os vetores de elementos aleatórios e decrescentes quase compartilham de mesmo tempo e execuções, mudando apenas para o vetor com elementos crescente que mesmo com vetores em ordem, a varredura é feita resultando na demora de execução do programa.

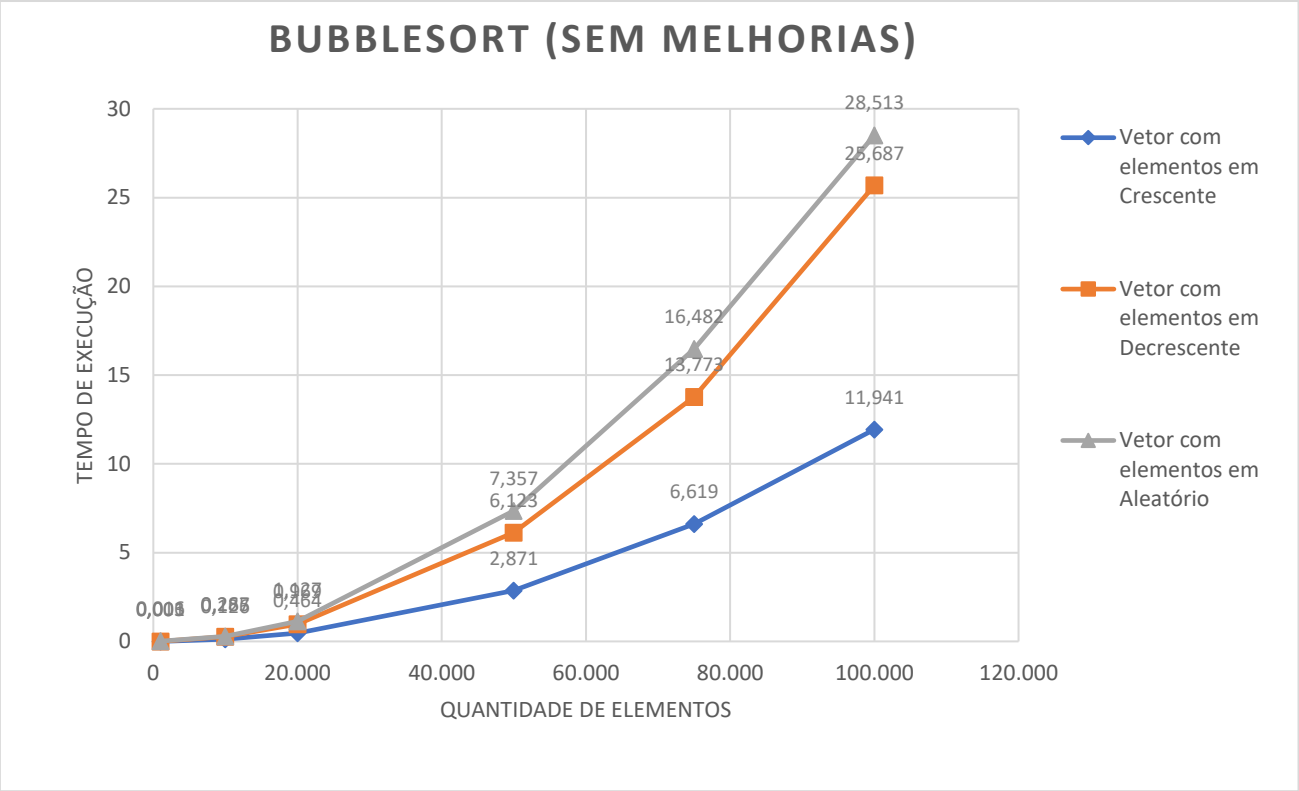

Figura 1 bubblesort sem melhoria

Concluindo, esse algoritmo é bem ineficaz para ordenações de vetores com muitos elementos e com algoritmos melhores já desenvolvidos, restringindo o algoritmo para casos onde a quantidade de elementos é pequena, devido a facilidade de implementação do código.

II: BubbleSort (Melhorado)

Derivado do BubbleSort original (versão sem melhoria), esse código possui um aprimoramento do qual permite verificar a ordem do vetor que passa em sua varredura , gerando dois tipos de complexidades assintóticas e dois casos respectivamentos

• Complexidade do Algoritmo

Temos a complexidade $O(n^2)$ para o pior caso e $O(n)$ para o melhor caso.

O melhor caso é dado quando o vetor já vem organizado (em ordem crescente) , permitindo que a variável que verifica a disposição dos elementos funcione ,sem a necessidade de executar as trocas.

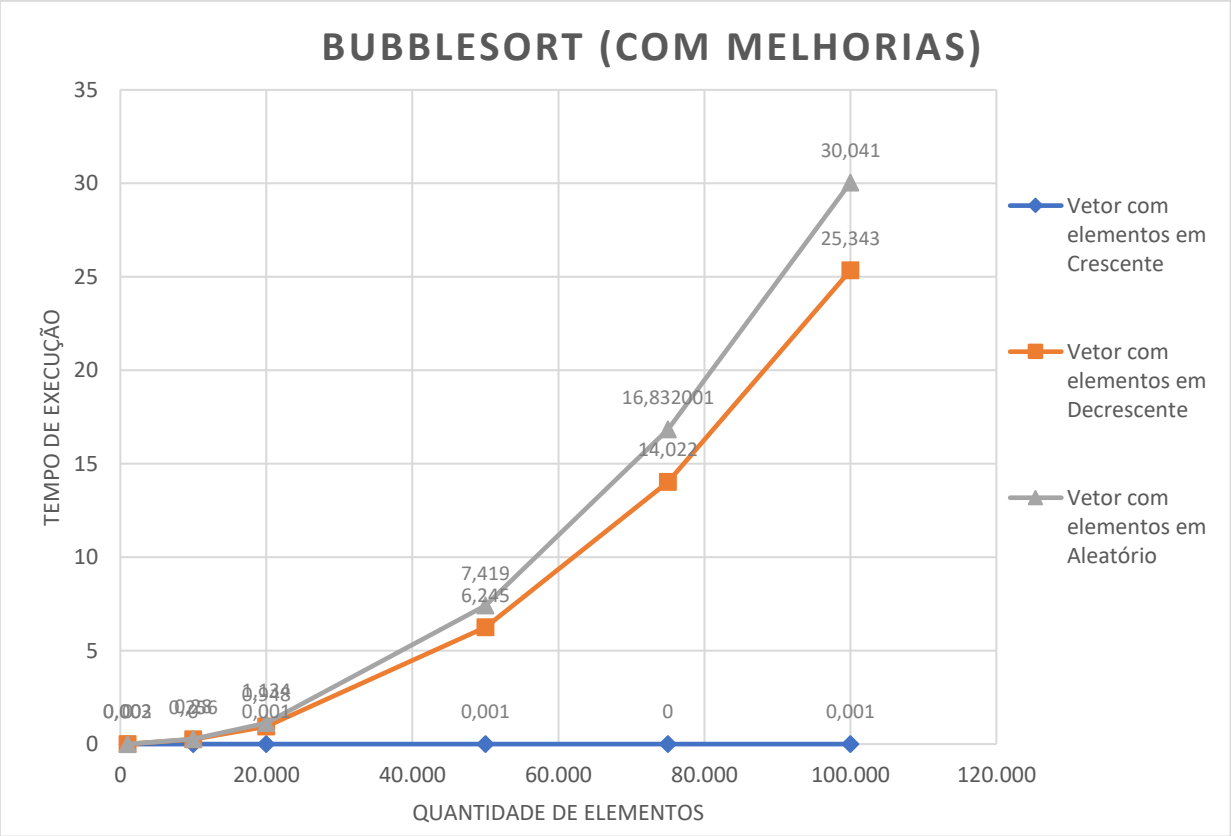

Figura 2 bubblesort com melhoria

Ao analisar o gráfico acima ambos vetores com elementos em ordem aleatória ou em ordem decrescente possuem uma curva de ordem quadrática e mesmo com tempo de execuções relativamente diferentes, é feito a dupla varredura para organizar, enquanto o vetor organizado cai no caso de curva linear onde o vetor não tem a necessidade da troca para organiza-lo.

Concluindo, mesmo com o aprimoramento que melhorou o algoritmo, ele ainda não é a escolha apropriada para vetores com uma quantidade grande de elementos devido à grande quantidade de tempo para executar a ordenação, mas sua facilidade de implementação é um ponto a ser considerado caso os elementos sejam em quantidades pequenas.

III: InsertionSort (Inserção Direta)

Seu método de organização é basicamente o algoritmo que cria uma vetor final com cada elemento do vetor original que é passado em sua varredura, sendo cada elemento adicionado nessa matriz, organizado, para exemplificar, imagine um baralho sendo organizado após a inserção de uma carta nova, o que o algoritmo faz é comparar a valor da carta atual com as outras para achar sua posição correta.

• Complexidade do Algoritmo

Este algoritmo possui dois tipos de complexidades, sendo duas delas para o caso médio e pior caso, $O(n)$ e $O(n^2)$.

- Para o melhor caso temos: $O(n)$.
- Para o pior e médio caso temos: $O(n^2)$.

O melhor caso do algoritmo que está presente em uma curva linear, dado o vetor ordenado e portanto não há necessidade de fazer qualquer tipo de troca de posição entre os elementos do vetor ,o médio caso é quando o vetor é decrescente e este compartilha uma curva quadrática com o pior caso(quando os elementos do vetor estão desorganizados), mudando apenas o formato da curva, como pode ser observado no gráfico abaixo.

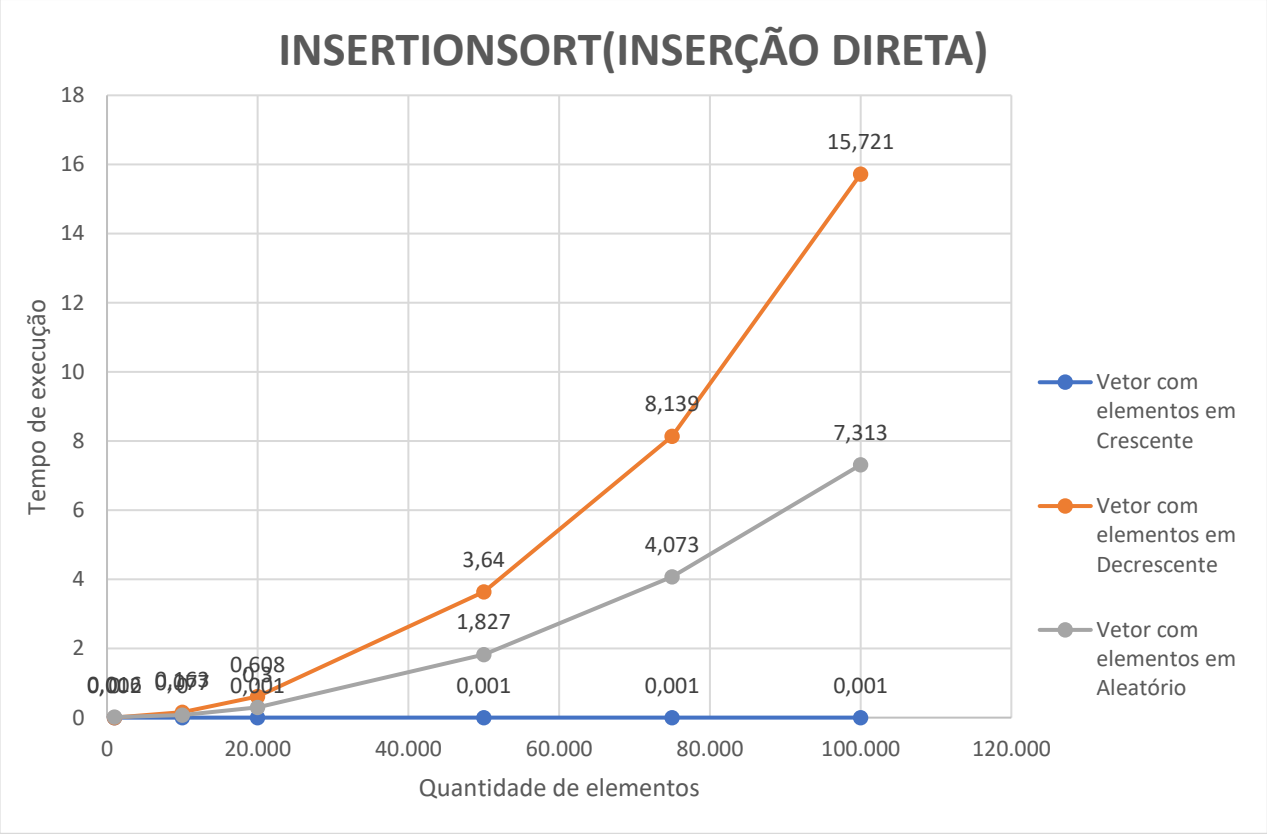

Figura 3 insertion sort (inserção direta)

Concluindo, o InsertionSort é um algoritmo que quando comparados a outros algoritmos existentes é difícil considerar sua utilidade para um código, entretanto possui algumas vantagens consideráveis quando for pensar num caso onde a quantidade de elementos é pequena, como possuir uma quantidade constante para espaço de memória adicional, algoritmo estável para quantidades muito grandes de elementos e não altera a ordem de elementos com valores iguais.

IV: MergeSort

Baseado na teoria do dividir e conquistar, o MergeSort divide os elementos do vetor até que estes estejam isolados e recursivamente vai juntar cada elemento e compara-los de uma maneira que o menor elemento sempre esteja atrás do elemento maior até que o vetor original esteja refeito de maneira organizada.

• Complexidade do Algoritmo

Este algoritmo tem apenas um caso de complexidade que é $O(nlog(n))$.

- Caso único: $O(nlog(n))$

Apesar do caso de complexidade ser única, as curvas para cada maneira que os elementos do vetor foram distribuídos (crescente, decrescente e aleatório) são relativamente diferentes devido ao tempo de execução que cada caso teve.

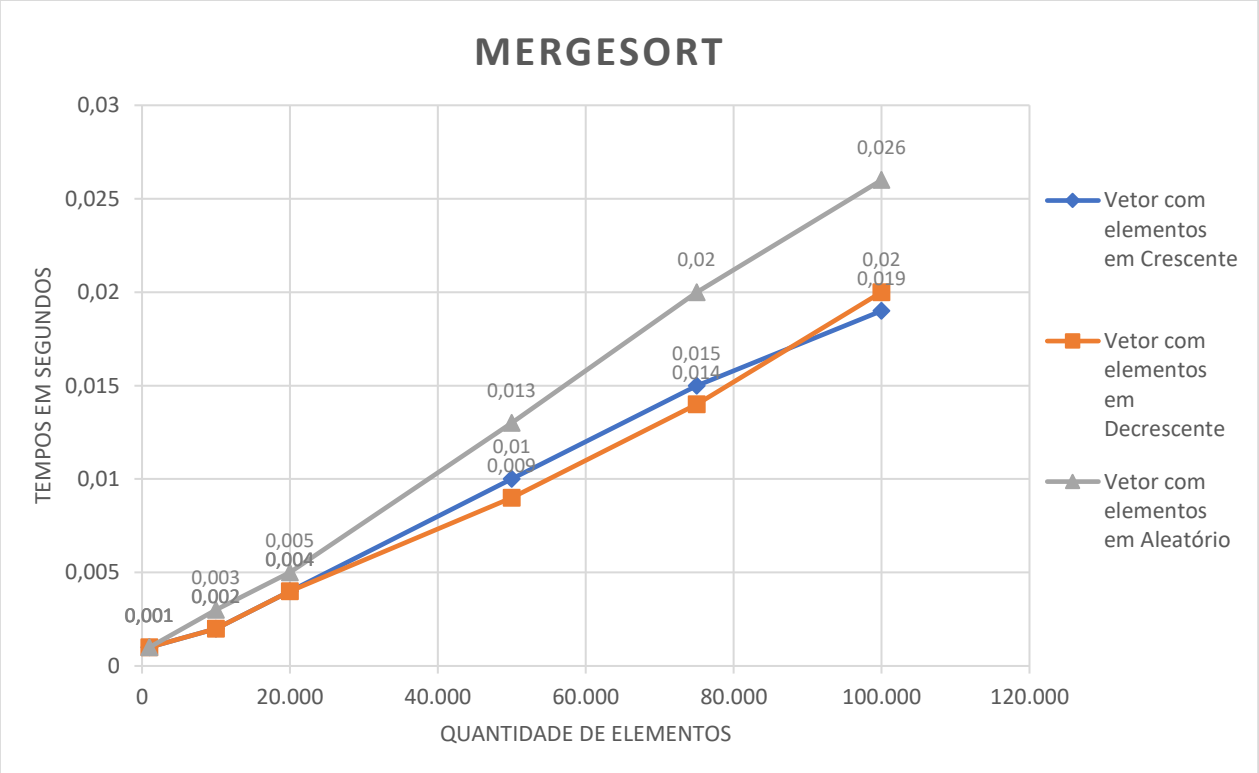

Figura 4 mergesort

Ao olhar para o gráfico, a curva com elementos crescentes é o melhor caso, pois sua velocidade para dividir o vetor e criar novamente o vetor é mais rápida e o pior caso é a curva com elementos aleatório e decrescentes que tomariam mais tempo para dividir os vetores e reorganizar o vetor ordenado.

Concluindo a estabilidade do MergeSort e sua velocidade para ordenar vetores com elementos de qualquer tipo de ordem é muito eficaz e sendo uma ótima escolha para análises com bastantes elementos disponíveis (devido a velocidade que sua curva cresce), o caso aonde utilizar ele é não aconselhável é para vetores com elementos organizados , já que o MergeSort irá organizar o vetor mesmo assim, fazendo com que algoritmos como InsertionSort sejam mais preferíveis nesse caso.

V: QuickSort(Pivô com o elemento médio do vetor)

Inventado por C.A.R Hoare em 1960, esse algoritmo foi desenvolvido para reduzir o problema original em subproblemas e estes possam ser resolvidos mais fácil e rápido que o problema original. Adotando a estratégia de divisão e conquista, escolhendo um pivô para a separação do vetor, sendo os elementos menos que o pivô uma lista e os maiores, outra lista e logo em seguida as duas sub-listas são criadas e após ordenar cada sub-lista, o algoritmo recursivamente começa a reorganizar o vetor original, só que agora organizado.

• Complexidade do Algoritmo

Este algoritmo possui complexidade para:

- Pior caso: $O(n^2)$
- Médio caso: $O(nlog(n))$
- Melhor caso: $O(nlog(n))$

Os médio e melhor caso é quando o vetor tem seus elementos particionados corretamente pelo algoritmo, existindo apenas uma pequena diferença no tempo de execução possuem curvas semelhantes , esse particionamento ocorre quando os vetores possuem os elementos dispostos em ordem crescente e decrescente , enquanto o pior caso é representado pela curva quadrática , este ocorre quando o particionamento do vetor não é feito corretamente ,mais ocasionalmente o vetor com elementos em disposição aleatória pode cair nesse caso dependendo de como ocorra a seleção do elemento do meio que será o pivô(os elementos em ordem crescente e decrescente também podem cair nesse caso)

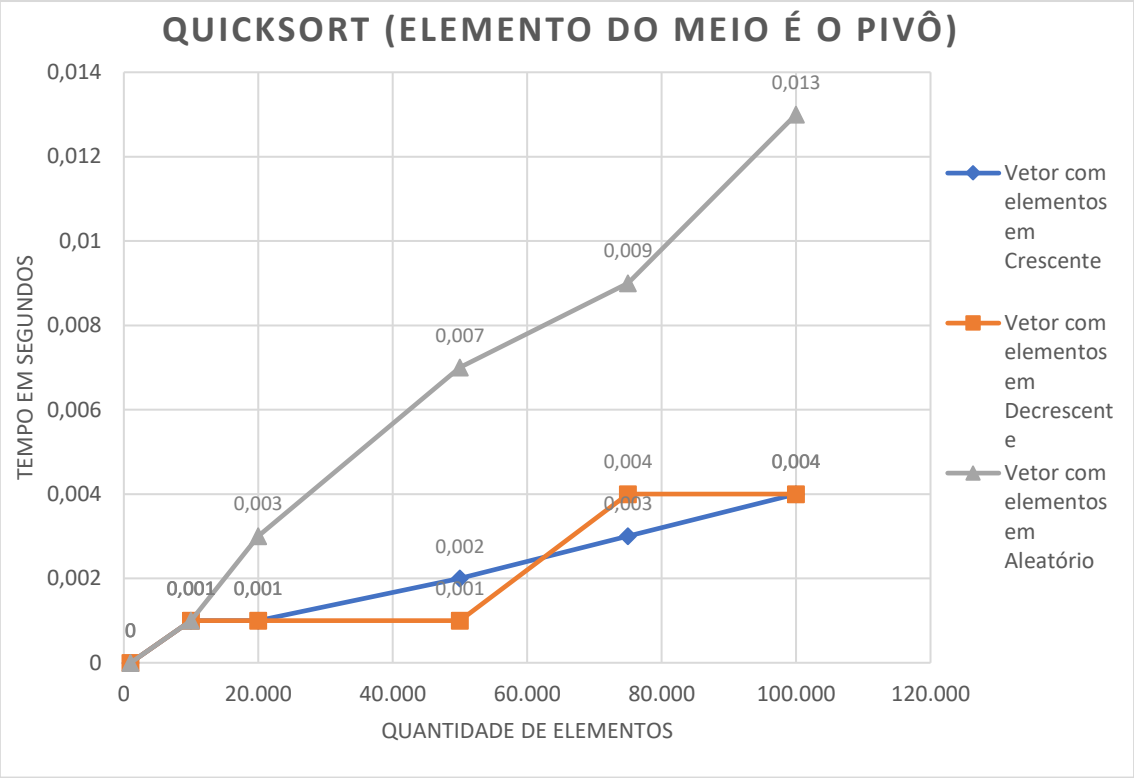

Figura 5 QuickSort com pivô de elemento médio do vetor

Ao observar o gráfico acima, o melhor caso é o vetor em ordem crescente, médio caso o vetor em ordem decrescente e o pior caso é o vetor com elementos em disposição aleatória.

Concluindo, o quicksort é um algoritmo com um método de organização preciso e rápido , porém apresenta instabilidade(muito de seus elementos originais perdem seus valores de chaves originais) e por esse motivo , dependendo da data que será organizada , o quicksort não é recomendado.

V-i): QuickSort(Pivô com o primeiro elemento do vetor)

Escolhendo o primeiro elemento do vetor como pivô resulta no caso onde a partição do quicksort não é bem aproveitada (caso onde o vetor já tem seus elementos em ordem crescente) resultando no pior caso.

• Complexidade do Algoritmo

Este algoritmo possui complexidade para:

- Pior caso: $O(n^2)$
- Médio caso: $O(nlog(n))$
- Melhor caso: $O(nlog(n))$

O pior caso ocorre quando os elementos do vetor estão dispostos em ordem crescente e isso gera uma curva quadrática, mas os outros casos ainda se mantem iguais aos casos anteriores.

Devido à instabilidade do algoritmo com elementos maiores que 20.000, o vetor com o pivô no primeiro elemento, não conseguiu atingir um gráfico completo quanto o quicksort original.

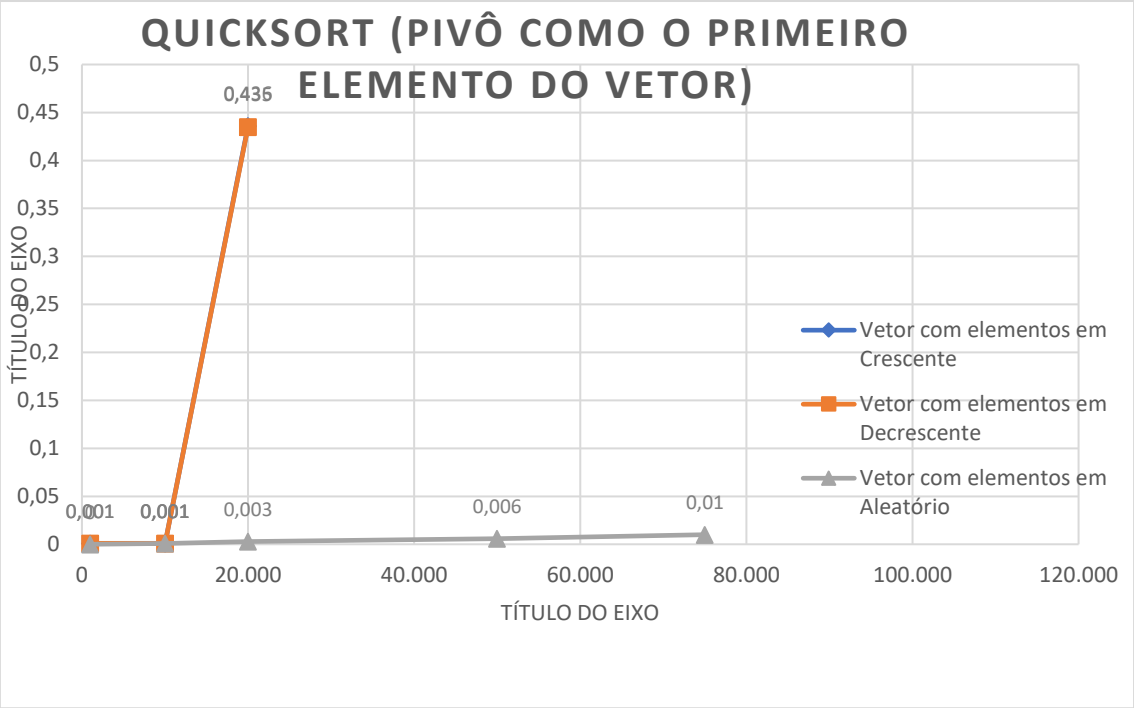

Figura 6 QuickSort com primeiro elemento do vetor como pivô

Concluindo, quando o quicksort é implementado com o pivô sendo o primeiro elemento do vetor, a sua instabilidade e a possibilidade de sempre cair no pior caso é uma desvantagem para utilizar o algoritmo desta maneira.

VI: HeapSort

Considerado um algoritmo do tipo de seleção, o HeapSort trabalha com a divisão e conquista, mas diferente dos outros, o heapsort utiliza uma estrutura de dados por heap, esta compara cada heap pai e filho inserido para realizar a troca e ordenar o vetor.

• Complexidade do Algoritmo

Este algoritmo possui complexidade para:

Como o HeapSort não é sensível a disposição dos elementos do vetor, sua complexidade é única.

- $\boldsymbol{\theta(nlog(n))}$

Como o algoritmo é indiferente pra a disposição dos elementos no vetor, suas curvas são semelhantes , diferindo apenas no tempo de execução, podemos dizer que mesmo compartilhando o mesmo tipo de complexidade ,ainda há melhores e piores casos sendo, o pior caso o vetor com elementos aleatórios (o heap demora mais tempo para organizar), médio caso os elementos por ordem crescente (a sua demora pode ser exemplificado quando os heaps são formados e o vetor necessita comparar termos de heaps pais e filhos e fazer as trocas independente de como eles foram organizados) e o melhor caso os elementos em ordem decrescente, essa análise pode ser observada no gráfico abaixo.

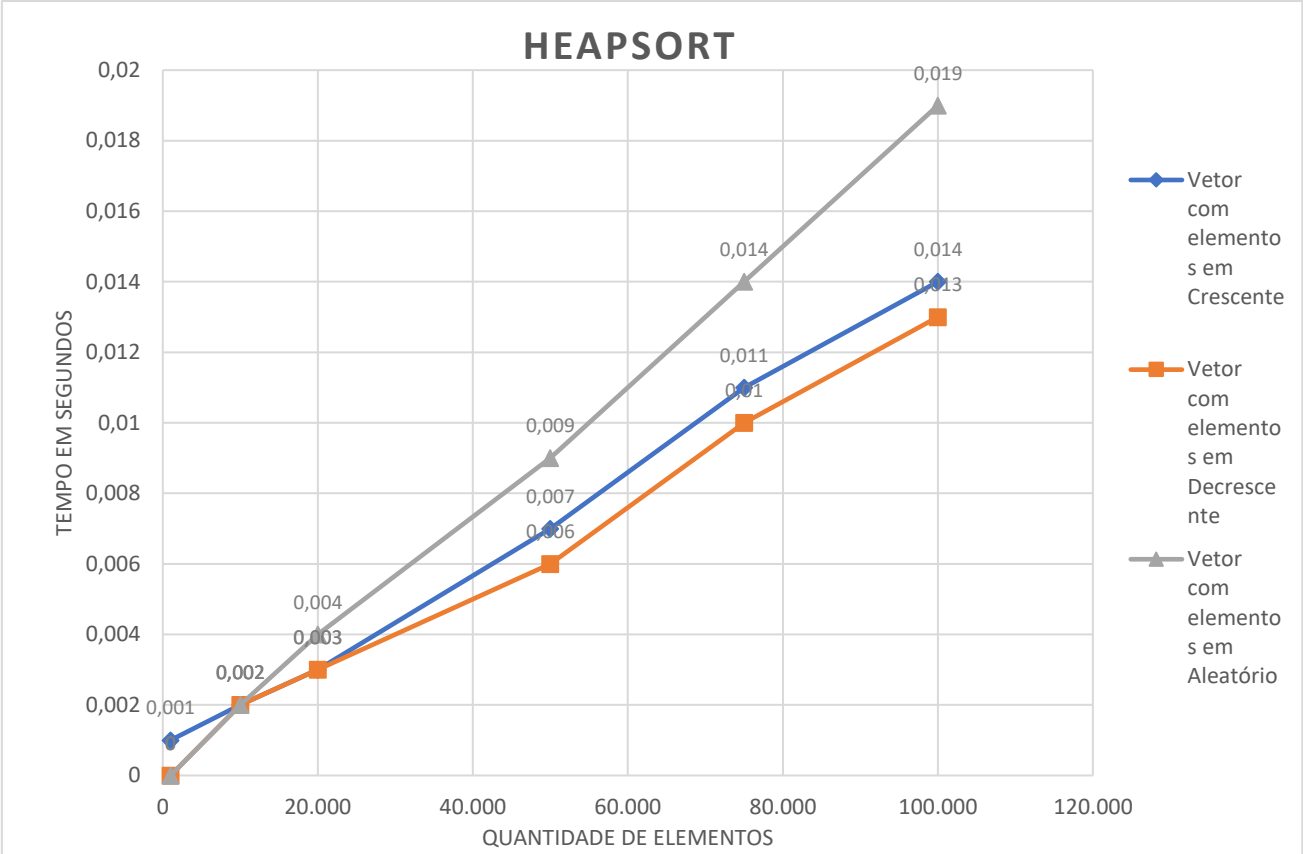

Figura 7 HeapSort

Concluindo o HeapSort é muito apropriado para ordenação de vetores , porem possui um problema de estabilidade que o quicksort compartilha em relação a seus elementos e suas chaves relacionadas, mas com uma adaptação correta(onde cada elemento da estrutura que é formado deve ficar em pares) permite que o algoritmo seja uma excelente escolha para ordenar vetores com grandes quantidades.

VII: SelectionSort

O algoritmo ordena um vetor varrendo o mesmo completamente, encontrando o menor valor de todos e o coloca na primeira posição, então repete para encontrar o segundo menor e o coloca na segunda posição, e assim por diante para todas as posições.

• Complexidade do Algoritmo

Temos a complexidade $O(n^2)$ para qualquer caso, já que o algoritmo sempre executa uma varredura completa do vetor para cada posição, independente de ordenação ou iteração.

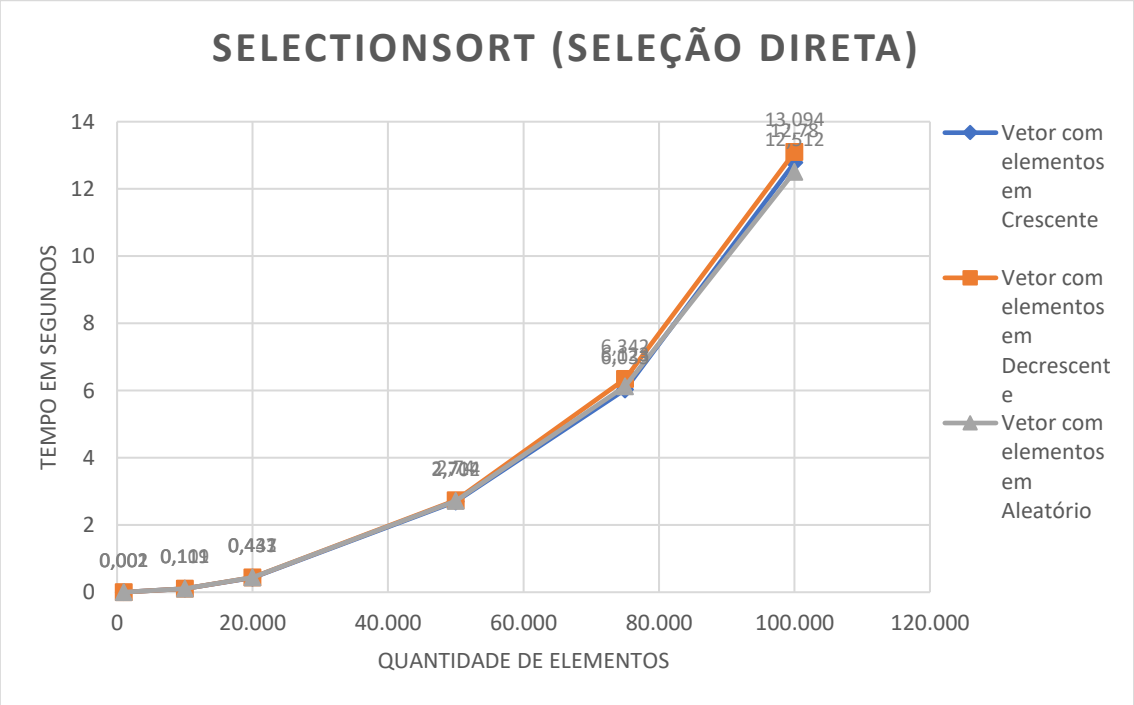

Figura 8 selectionsort

Percebe-se com o gráfico acima que o tempo de execução cresce exponencialmente com o tamanho do vetor, com pouca variação independente da ordenação prévia do mesmo.

Concluindo, o selection sort não é especialmente eficiente com tempo de execução em relação a outros métodos de ordenação, mas compensa por ser um dos algoritmos de mais simples implementação, o que pode economizar tempo dos programadores.

VIII: ShellSort

O algoritmo do ShellSort comporta-se como o método da inserção direta, porém refina-o ao dividir o vetor em grupos menores para organizá-los separadamente, tornando o programa mais eficiente no geral para vetores com mais elementos.

• Complexidade do Algoritmo

A complexidade depende do tamanho dos gaps, considerando que, se um gap for muito grande e sobrar no total sobre o vetor, a complexidade se aproxima da do pior caso de um InsertionSort padrão.

- Melhor caso: $O(nlog^2(n))$
- Pior caso: $O(n^2)$

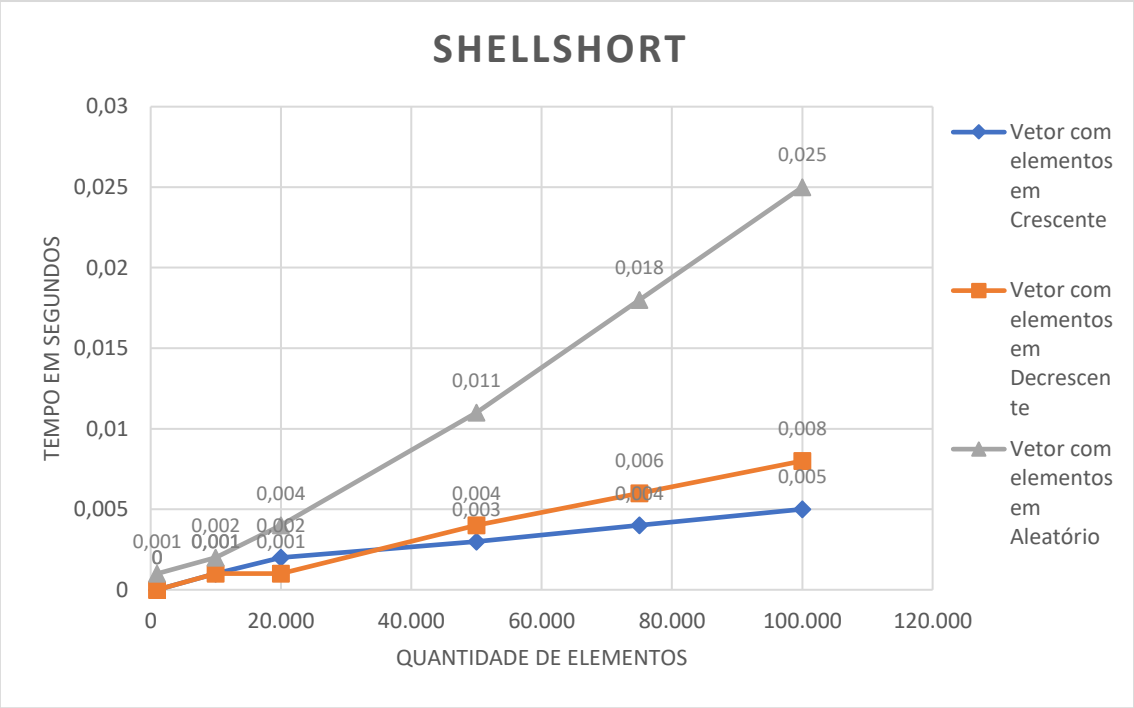

Figura 9 shellsort

Com o gráfico acima, percebe-se que o algoritmo é mais eficiente que o de inserção direta para vetores maiores, enquanto ainda se mantém relativamente simples de se implementar.

Conclui-se que o ShellSort mantém as características do InsertionSort de ser ineficiente com vetores grandes e ser extremamente instável para vetores diferentes, mas também é de fácil implementação e de maior viabilidade.

X: Comparações Finais

- Agora uma comparação geral entre os códigos

Comparando todos os códigos considerando o objeto de estudo o vetor de elementos em ordem aleatória, os algoritmos de divisão e conquista são os melhores em desempenho, por quebrar o tempo utilizando métodos de divisão e conquista ou até com estrutura de dados.

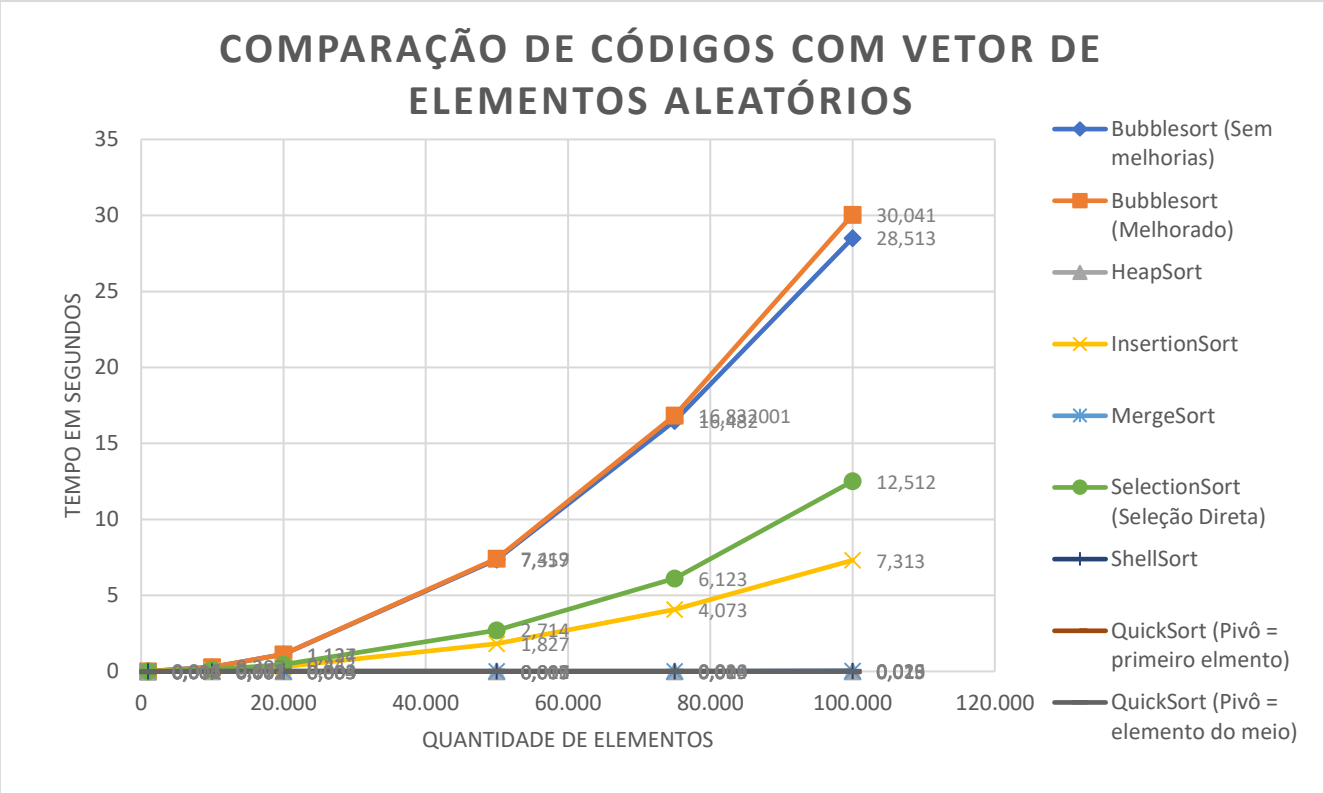

Figura 10 Comaprativo 1

Ao comparar os elementos de vetor que estejam em ordem crescente, é notado que os métodos de ordenação apresentados onde a complexidade é dada por uma curva linear, são mais apropriados nesse caso e são melhores tempos de execução chegando ao nível dos algoritmos de execução do divisão de conquista, fazendo estes serem mais apropriados para esse tipo de caso, como pode ser observado em baixo.

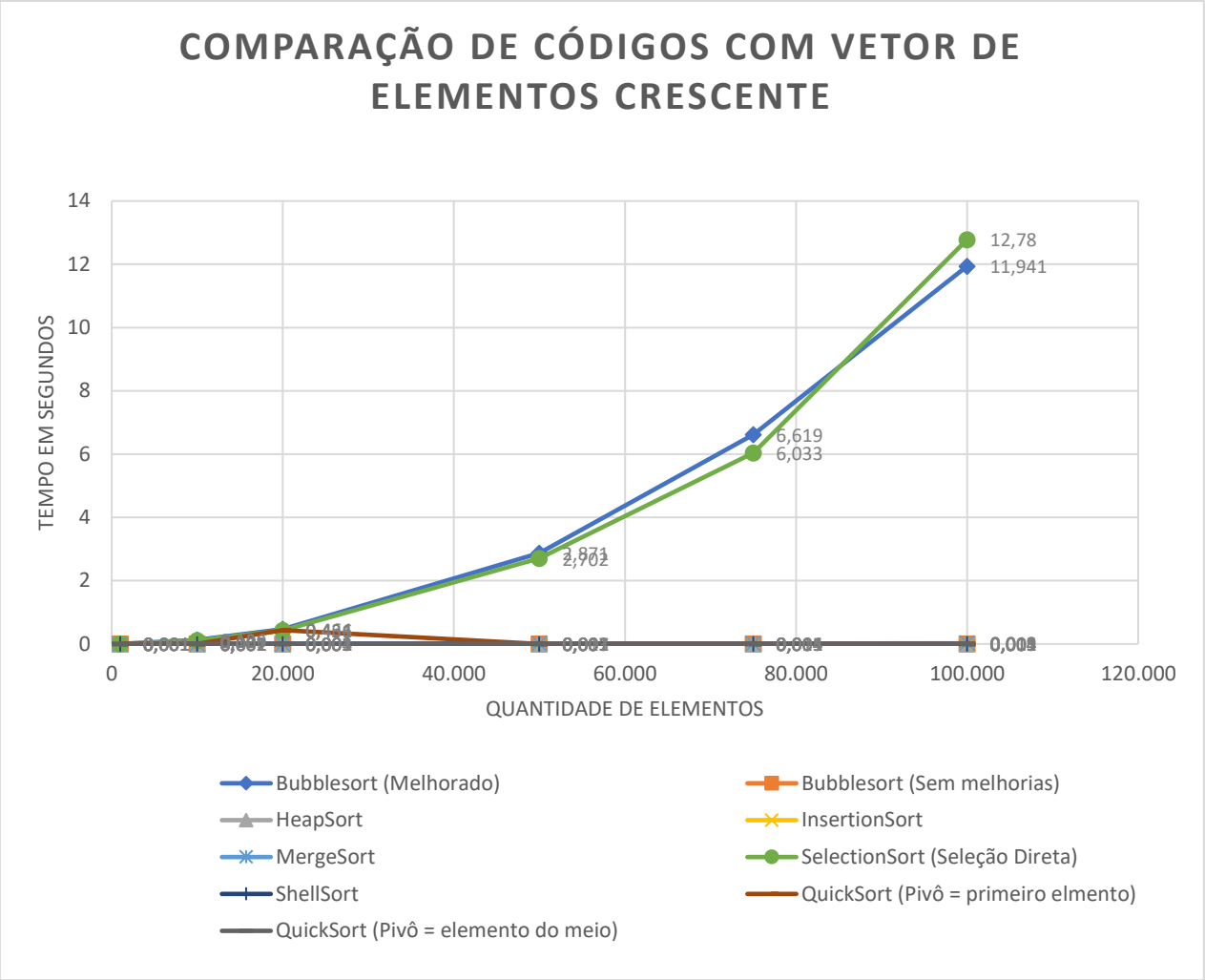

Figura 11 Comparativo 2

Por fim a ultima comparação a ser feita é de os elementos dispostos no vetor estarem em ordem decrescente. Os melhores casos estão relacionado aos algoritmos de divisão e conquista e de estrutura de dados com árvores , uma vez que estes não precisão verificar a disposição de ordem do elemento e simplesmente ordenam os vetores e ao incrível tempo de execução, fazendo que códigos como insertionsort sejam desconsiderados para utilização, como pode ser observado no gráfico abaixo.

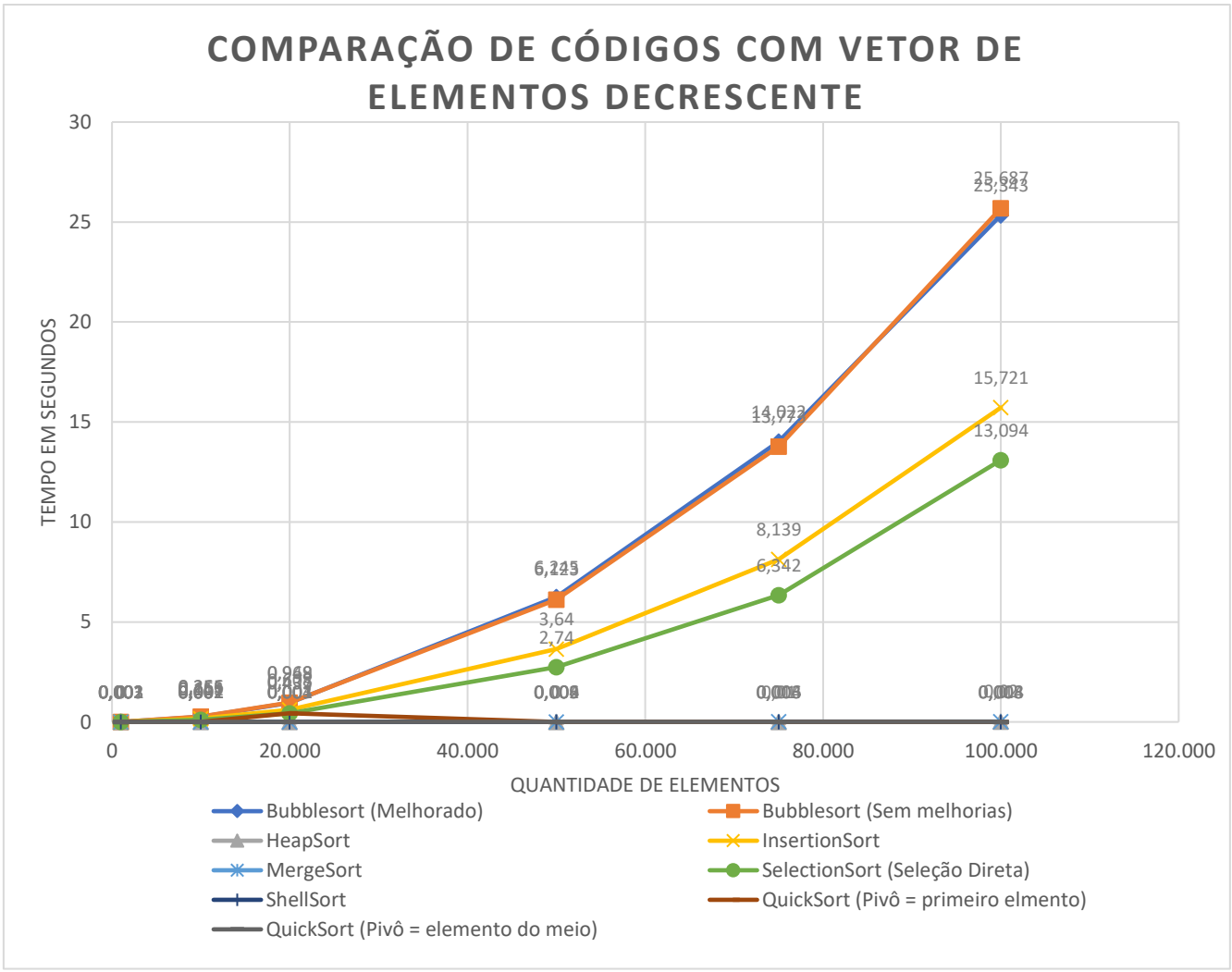

Figura 12 Comparativo 3