CES-12 - Relatório Lab 3 - Sorting

Aluno: Bruno Costa Alves Freire

1 de Junho de 2020

1 Comparações entre os algoritmos de ordenação

1.1 Tempo de execução: QuickSort vs. MergeSort vs. RadixSort

Para realizar uma primeira comparação de desempenho entre os algoritmos estudados, plotamos o tempo de execução de cada um para vetores de diferentes tamanhos, no gráfico da figura 1.

No gráfico da figura 1, podemos constatar a grande disparidade entre o RadixSort e os outros dois, tendo o QuickSort o melhor desempenho geral. Cabe ressaltar que as versões de cada algoritmo utilizadas na comparação foram: a versão iterativa do MergeSort, a versão com pivô fixo e 2 recursões do QuickSort, e a implementação do RadixSort foi feita dividindo-se os elementos do vetor em duas filas conforme o $d-$ésimo dígito binário, no $d-$ésimo passo.

A escolha da versão "primo pobre" do QuickSort nesse cenário ocorre pois se trata de vetores aleatoriamente gerados. Nesses casos, as variantes com mediana de 3 apresentam um custo adicional que não se mostra necessário.

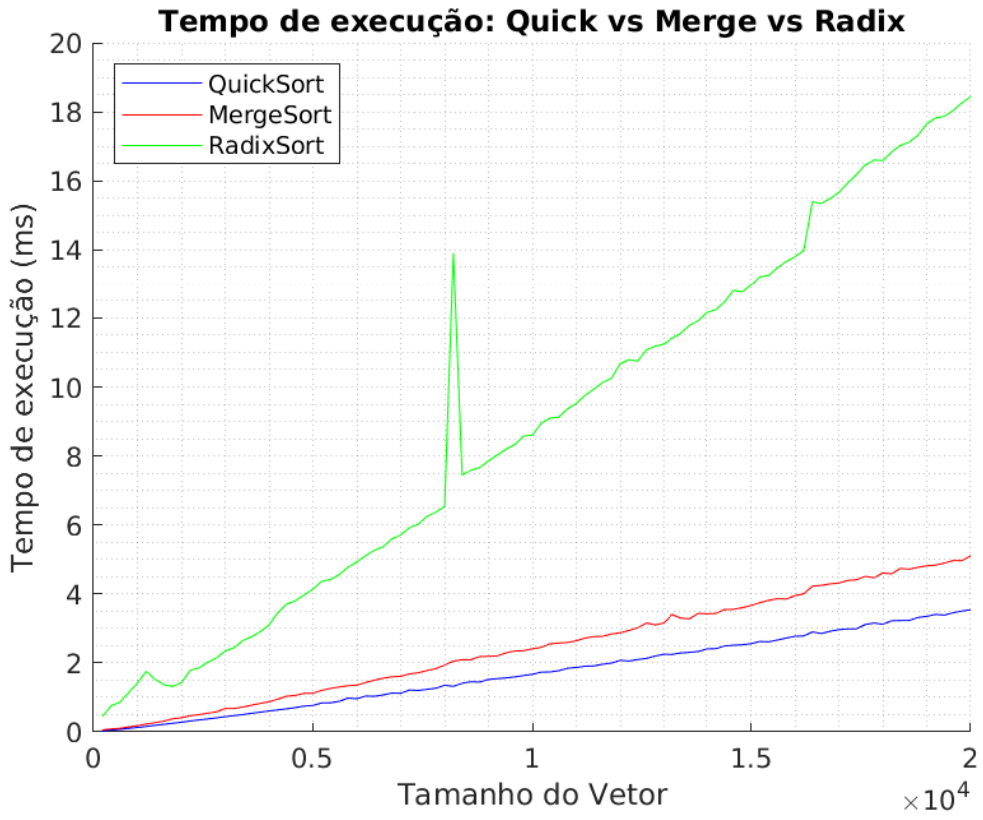

Figura 1: Comparativo de tempo de execução entre os algoritmos QuickSort, MergeSort e RadixSort.

1.2 Desempenho do MergeSort: Recursivo vs. Iterativo

Seguimos a comparar as variantes de cada algoritmo, começando pelo MergeSort. Implementamos sua versão recursiva e sua versão iterativa. Como não faz sentido comparar o uso da pilha de recursão nesse caso, uma vez que a versão iterativa não faz uso da mesma, plotamos apenas os tempos de execução de cada variante para vetores aleatórios, conforme o gráfico da figura 2.

Podemos notar uma pequena vantagem da versão iterativa sobre a versão recursiva, em termos de tempo. A outra vantagem, é claro, é a economia na pilha de recursão.

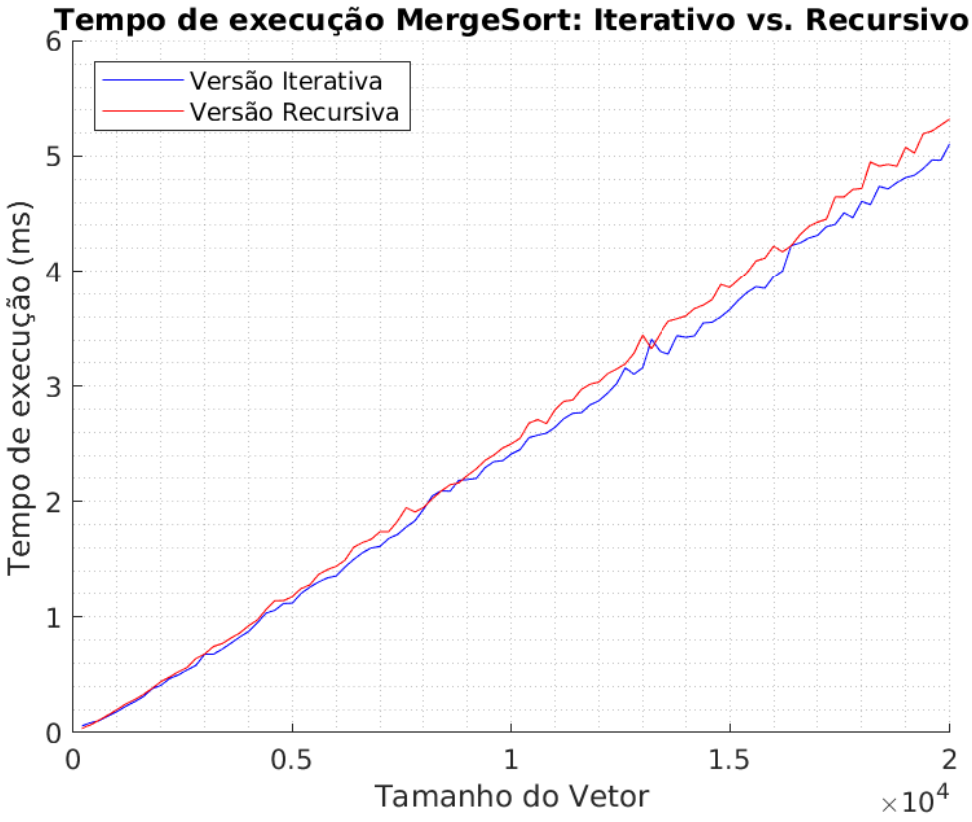

Figura 2: Comparativo de tempo de execução entre as variantes recursiva e iterativa do MergeSort.

1.3 Uso da pilha de recursão no QuickSort: 2 recursões vs. 1 recursão

Dentre as variantes do QuickSort, temos uma alternativa que busca reduzir o crescimento da pilha de recursão do algoritmo, realizando apenas uma chamada recursiva após o particionamento do vetor, na menor partição.

No gráfico da figura 3, temos a comparação do crescimento da pilha de recursão em profundidade, juntamente com a quantidade total de chamadas recursivas. No gráfico da figura 4, mostramos uma comparação de tempo de execução entre as duas variantes.

Podemos observar, do primeiro gráfico, que a variante com 1 recursão de fato consegue reduzir drasticamente a profundidade da pilha de recursão, bem como reduz o número de chamadas recursivas aproximadamente pela metade. Contudo, tiramos do segundo gráfico que a diferença em termos de tempo de execução é imperceptível, de modo que essa variante não impacta o desempenho de tempo do algoritmo.

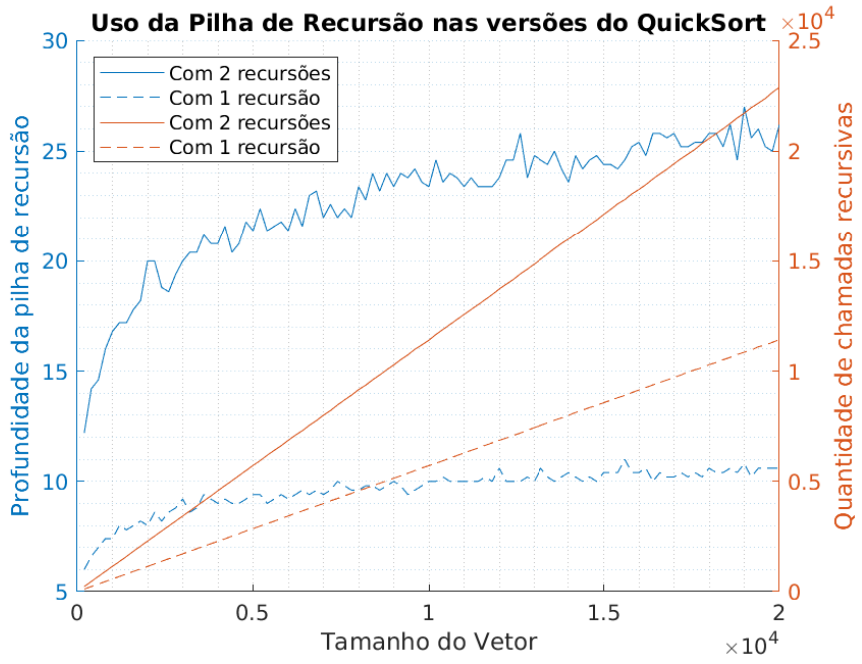

Figura 3: Comparativo de crescimento da pilha de recursão para as variantes do QuickSort com mediana de 3.

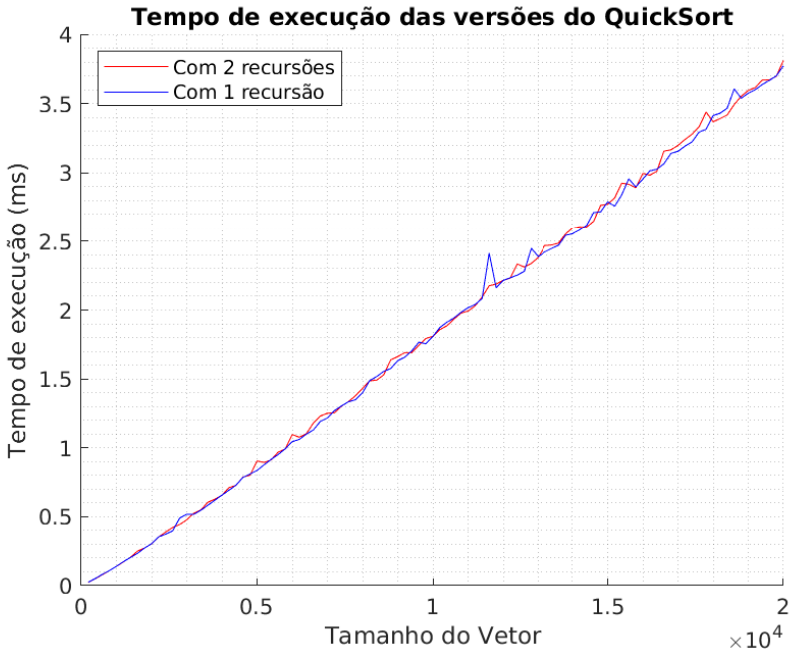

Figura 4: Comparativo de tempo de execução para as variantes do QuickSort com mediana de 3.

1.4 Evitando casos $O(n^2)$ no QuickSort: mediana de 3 vs. pivô fixo

Um aspecto problemático do QuickSort é o fato de sua complexidade de pior caso ser $O(n^2)$, uma ordem subótima, a qual, no entanto, só se pronuncia em casos bem particulares. Para tornar mais raros os casos em que o QuickSort apresenta tal comportamento, e melhorar seu desempenho em casos próximos, uma variação do algoritmo implementa a escolha do pivô com mediana de 3.

Para analisar o impacto dessa escolha no desempenho do algoritmo, temos os gráficos das figuras 5 e 6, onde comparamos respectivamente os tempos de execução e o crescimento da pilha de recursão, para os algoritmos executados com vetores *quase ordenados*. Note que aqui estamos comparando a variante "primo pobre" com a variante que usa 2 recursões e mediana de 3.

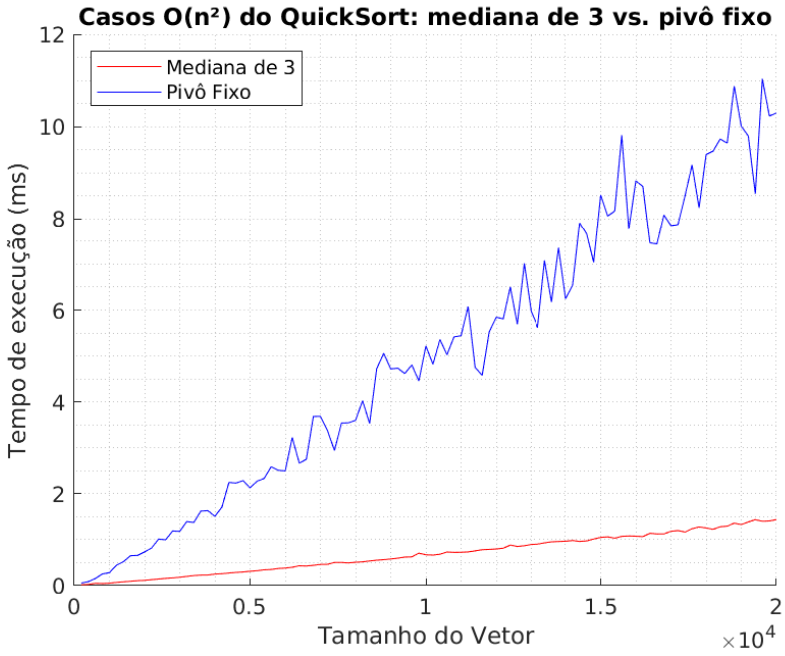

Figura 5: Comparativo de tempo de execução entre as variantes do QuickSort com pivô fixo e mediana de três para vetores *quase ordenados*.

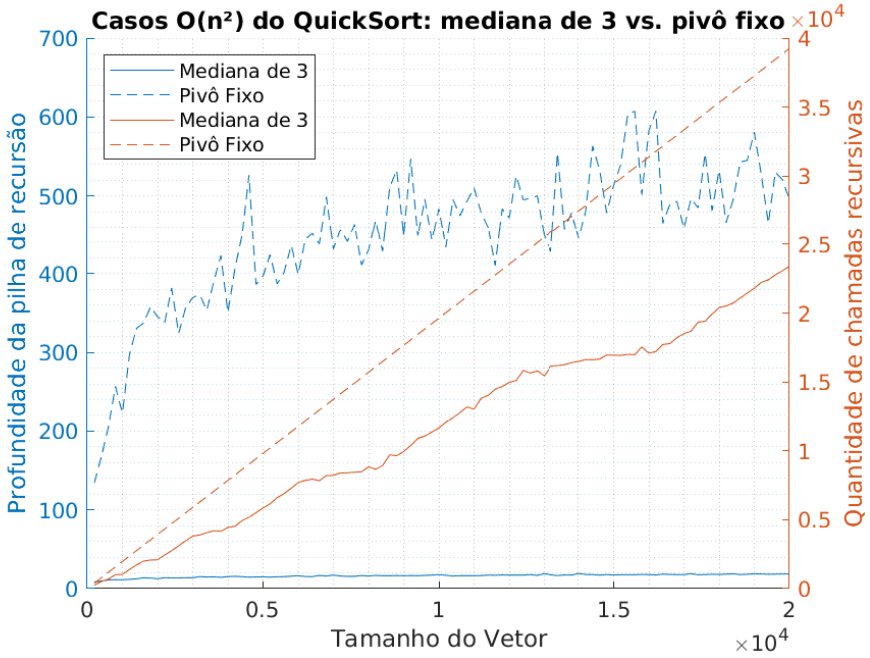

Figura 6: Comparativo de tempo de execução entre as variantes do QuickSort com pivô fixo e mediana de três para vetores *quase ordenados*.

Olhando para a figura 5 é bastante evidente o impacto no tempo de execução do uso da mediana de 3, visto que o pivô fixo transforma uma tarefa que a princípio quase não seria necessária numa grande perda de tempo. Na figura 6 também vemos uma diferença significativa no crescimento da pilha de recursão, um aspecto que parece ser beneficiado pelo uso da mediana de 3.

Para investigar um pouco mais a fundo o impacto dessa escolha do pivô no crescimento da pilha, plotamos o gráfico da figura 7, onde temos um comparativo das 3 variantes implementadas no tocante ao crescimento da pilha. Novamente, tratamos do caso em que os vetores estão *quase ordenados*.

Na figura 7 podemos constatar a relação de desempenho entre as três variantes no tocante ao crescimento da pilha de execução. O gráfico nos permite deduzir que tanto a escolha do pivô pela

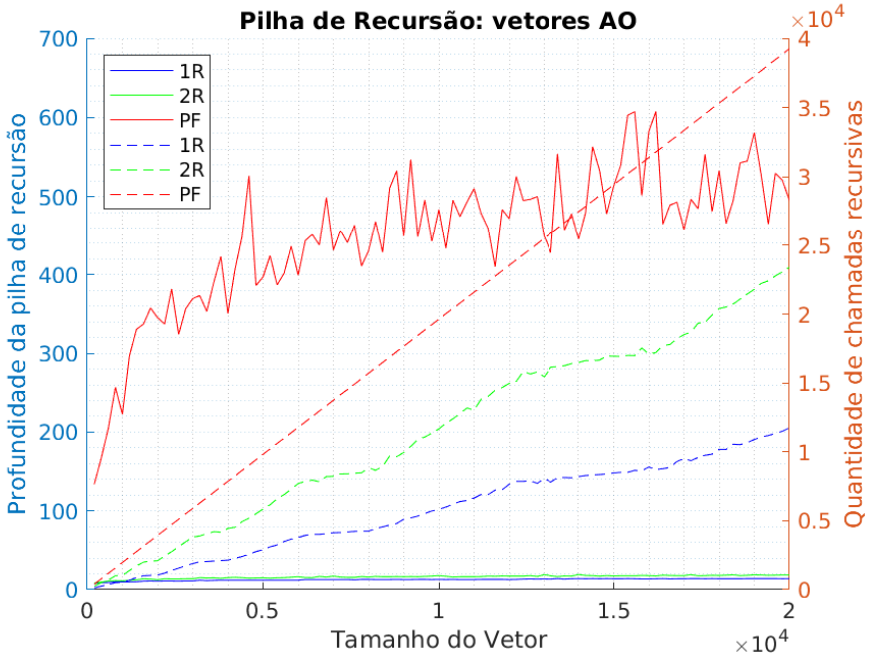

Figura 7: Comparativo do crescimento da pilha de recursão para as três variantes do QuickSort.

mediana de 3 quanto a chamada recursiva única são decisões que melhoram o desempenho do algoritmo nos piores casos, relativamente ao crescimento da pilha. Por outro lado, sabemos que a variante "primo pobre" é apenas um pouco mais rápida que as demais quando tratamos de vetores aleatórios.

2 Questão opinativa

Para basear a discussão dessa questão, realizamos um comparativo de tempo entre as três variantes do QuickSort aqui implementadas, e a versão disponível da STL, para os casos de vetores aleatórios e *quase ordenados*.

Na figura 8 vemos que a versão da STL tem performance inferior a todas as três variantes implementadas, enquanto na figura 9, vemos que a versão da STL é melhor que a versão "primo pobre", porém ainda perde para as duas variantes com mediana de 3.

Infelizmente não temos acesso aos dados sobre o crescimento da pilha na versão da STL para comparamos esse aspecto. Contudo, observando o comportamento da implementação STL face a vetores quase ordenados, podemos afirmar que ela implementa alguma medida contra o crescimento quadrático nesses casos, tal qual as variantes com mediana de 3.

Complementando a discussão anterior, as variantes com mediana de 3 têm performance levemente prejudicada em casos médios, por realizar mais processamento, no entanto, o ganho de desempenho que elas exibem nos piores casos é muito significativo. É um pequeno preço a se pagar para evitar perdas de tempo colossais em casos que seriam melhor resolvidos por algoritmos mais lentos, como o MergeSort.

Da mesma forma, sabendo que a STL é uma biblioteca que objetiva ser portável e ao mesmo tempo dinâmica (no sentido de que todas as classes são implementadas com *templates*), é compreensível que esta seja prejudicada por uma pequena perda de desempenho no caso médio em relação a uma implementação para um tipo fixo. Além disso, mesmo com essa desvantagem no caso médio, a implementação STL ainda se mostra satisfatória nos piores casos, mostrando que certamente houve a preocupação com esses casos no design da biblioteca.

Por fim, na minha opinião, a implementação nativa do C/C++ certamente teria medidas de projeto pensadas para lidar com os casos quase ordenados de modo satisfatório. Em outras palavras, considerando o custo benefício de uma implementação que ganha por pouco no caso médio, mas perde vergonhosamente nos piores casos (que a depender do problema, não são tão raros assim), eu não aprovaria tal implementação em detrimento de outra que lide adequadamente com os piores casos.

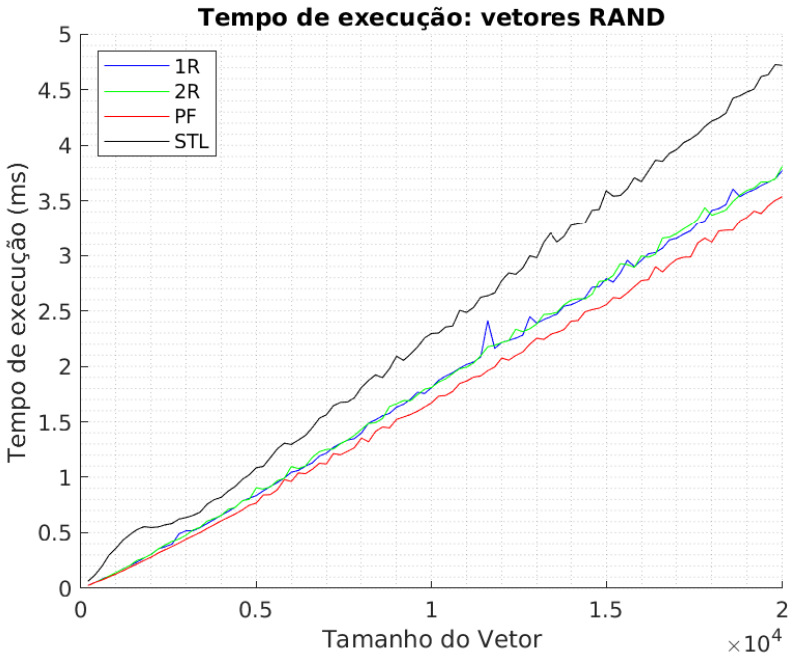

Figura 8: Comparativo de tempo entre as 3 variantes do QuickSort implementadas e a versão da STL para vetores aleatórios.

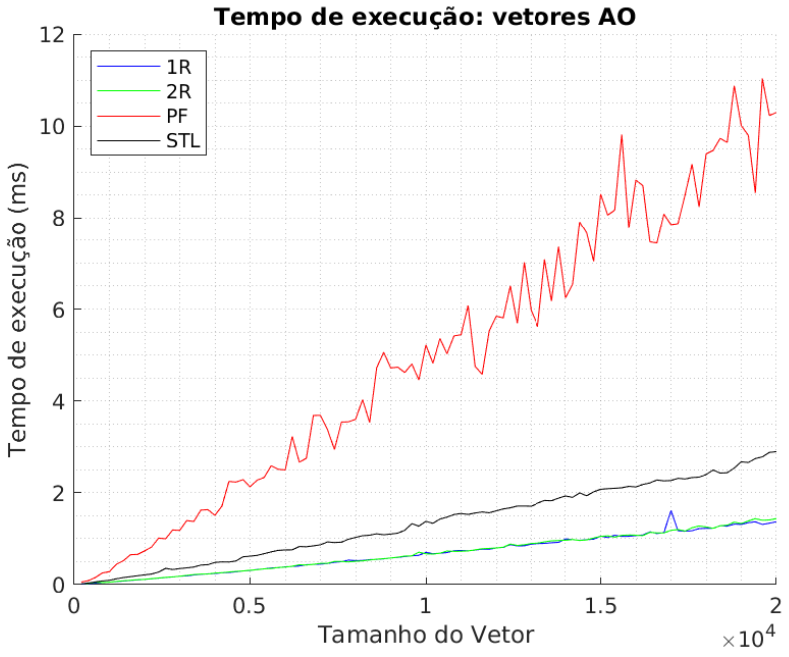

Figura 9: Comparativo de tempo entre as 3 variantes do QuickSort implementadas e a versão da STL para vetores quase ordenados.