Instituto Tecnológico de Aeronáutica
Divisão de Ciência da Computação

CTC-12

Laboratório 03

Sorting

Aluno:

Daniel Araujo Cavassani

Professor:

Luiz Gustavo Bizarro Mirisola

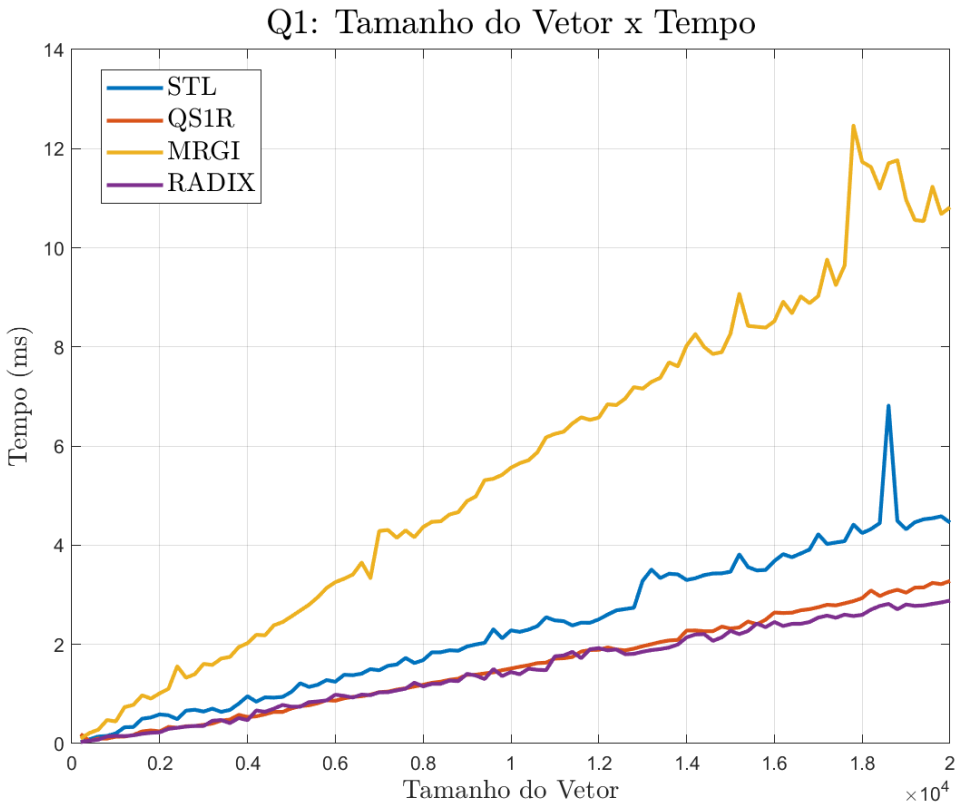
1) QuickSort x MergeSort x RadixSort x std::sort

Nesta análise, comparamos os tempos de execução de quatro algoritmos de ordenação aplicados a vetores aleatórios: STL, QuickSort com uma recursão e mediana de 3 (QS1R), MergeSort Iterativo (MRGI) e RadixSort (RADIX). Contrariando as expectativas teóricas, o QuickSort e o RadixSort tiveram tempos de execução muito próximos e similares.

A explicação para esse resultado inesperado pode estar no fato de que, mesmo tendo complexidade média de O(n.logn), o QuickSort utiliza operações rápidas, o que lhe confere um desempenho superior ao MergeSort. Este último, por sua vez, tem operações mais lentas devido à manipulação de subvetores.

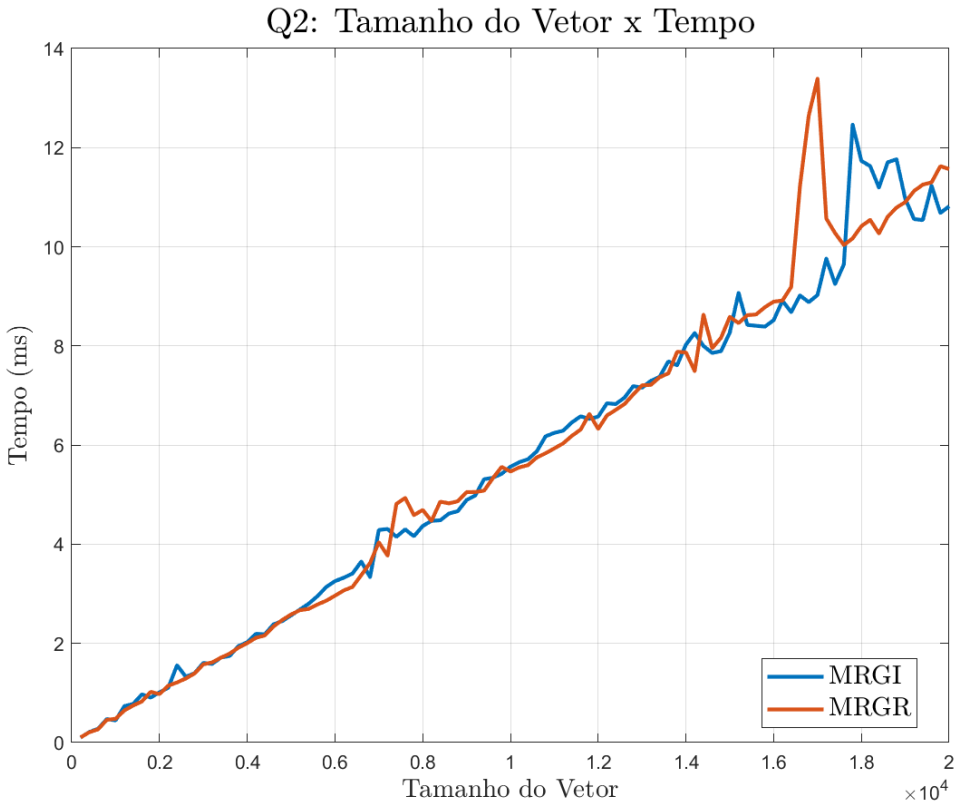
Quanto à comparação entre QuickSort e RadixSort, observamos que as constantes elevadas do RadixSort não prejudicaram seu desempenho em relação ao QuickSort. No início da análise, os dois algoritmos tiveram desempenho semelhante, mas o RadixSort acabou se mostrando ligeiramente melhor no final.

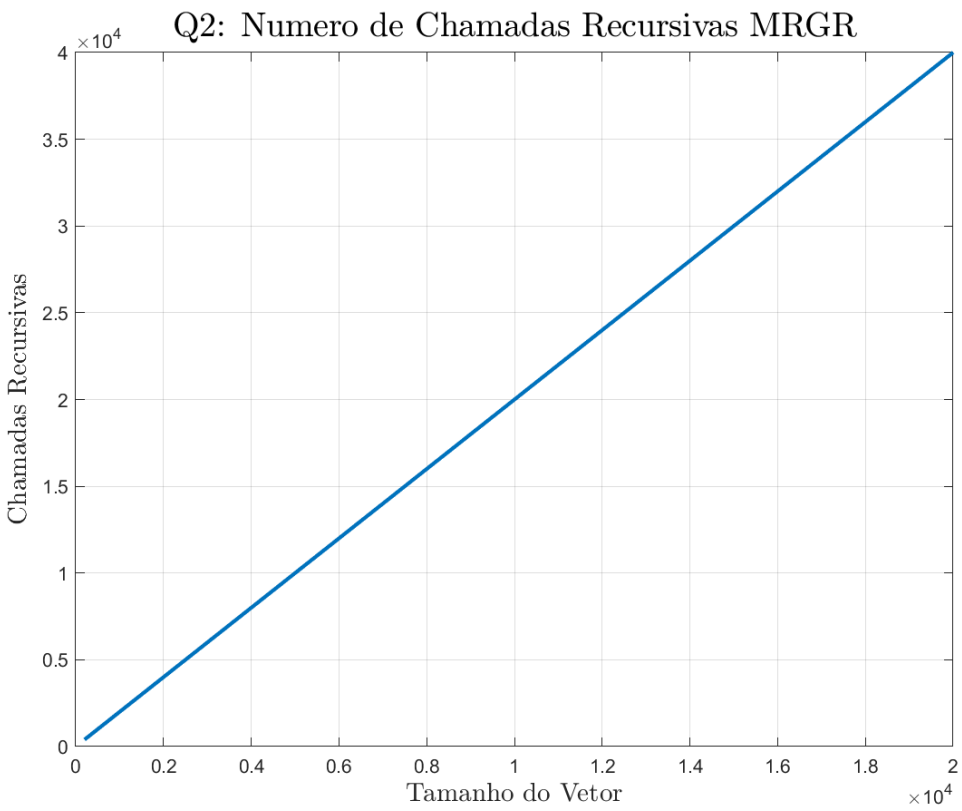
Em síntese, a classificação dos algoritmos em termos de desempenho, do melhor para o pior, foi: RadixSort, QuickSort, STL e, por último, MergeSort Iterativo. No entanto, é importante ressaltar que, apesar de ser uma versão otimizada com apenas uma chamada recursiva, o QuickSort ainda possui a limitação da pilha de recursão gerada, o que pode causar problemas de memória em alguns casos. Essa restrição deve ser considerada na avaliação dos resultados.

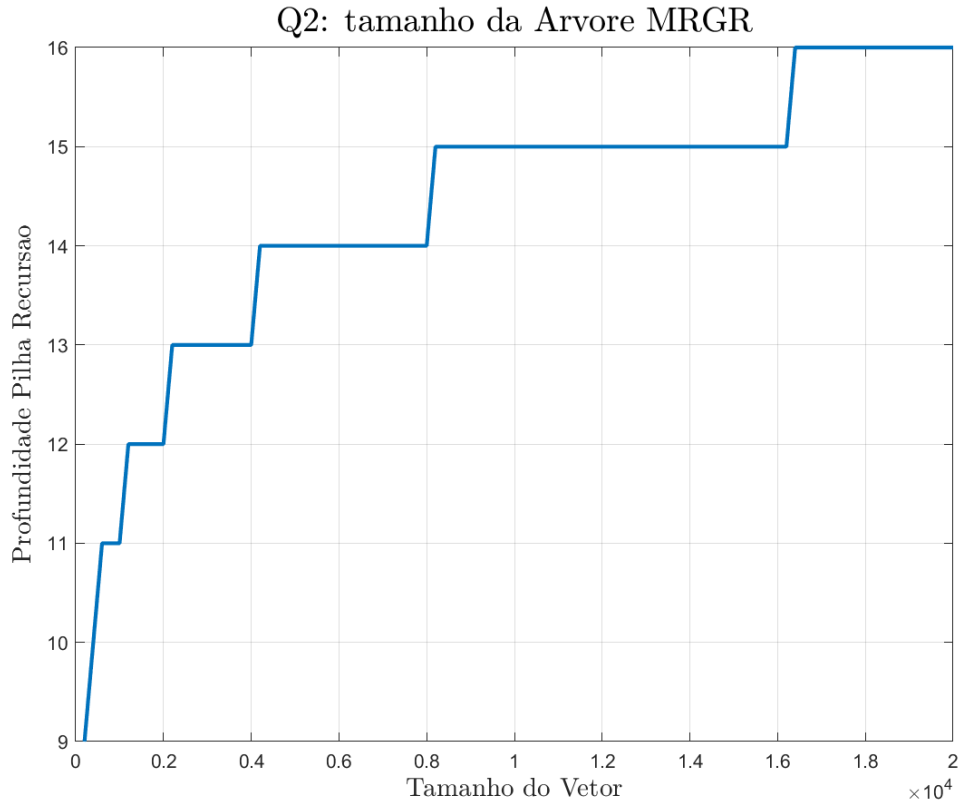


2) MergeSort: Recursivo x Iterativo

Nesta análise, comparamos o tempo de ordenação do MergeSort Recursivo (MRGR) e do MergeSort Iterativo (MRGI) aplicados a vetores aleatórios. Observamos que o MRGI apresenta um desempenho ligeiramente superior em termos de tempo. Além disso, o MRGI possui uma vantagem geral em relação ao MRGR, visto que não necessita de chamadas recursivas, o que economiza espaço na árvore de recursão.

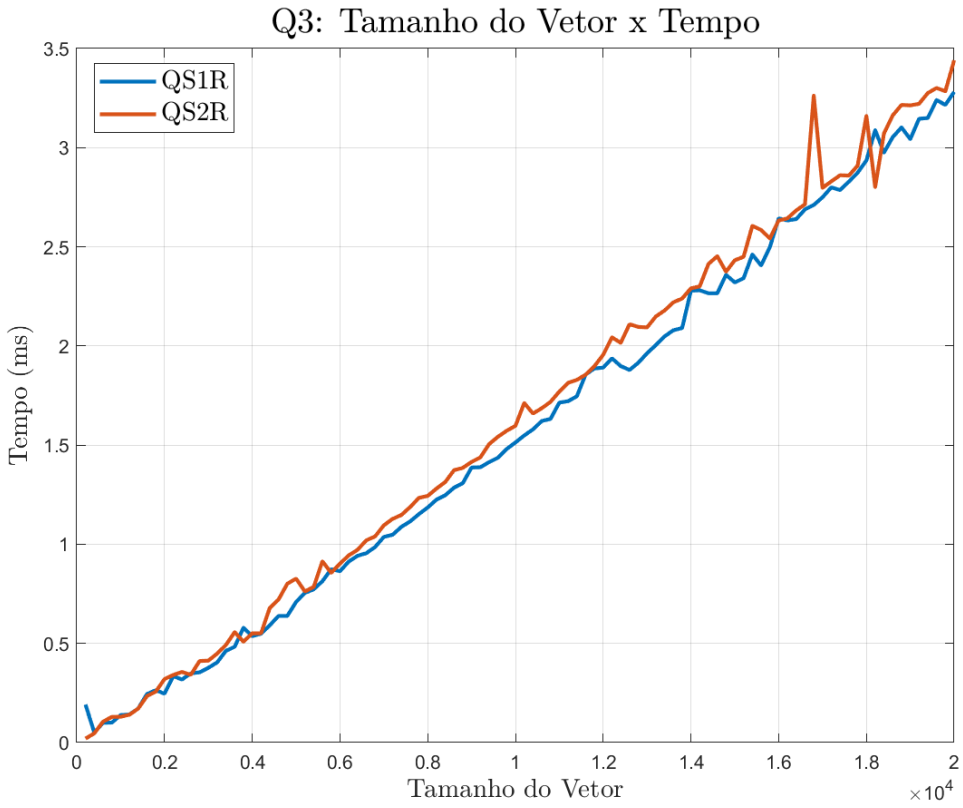


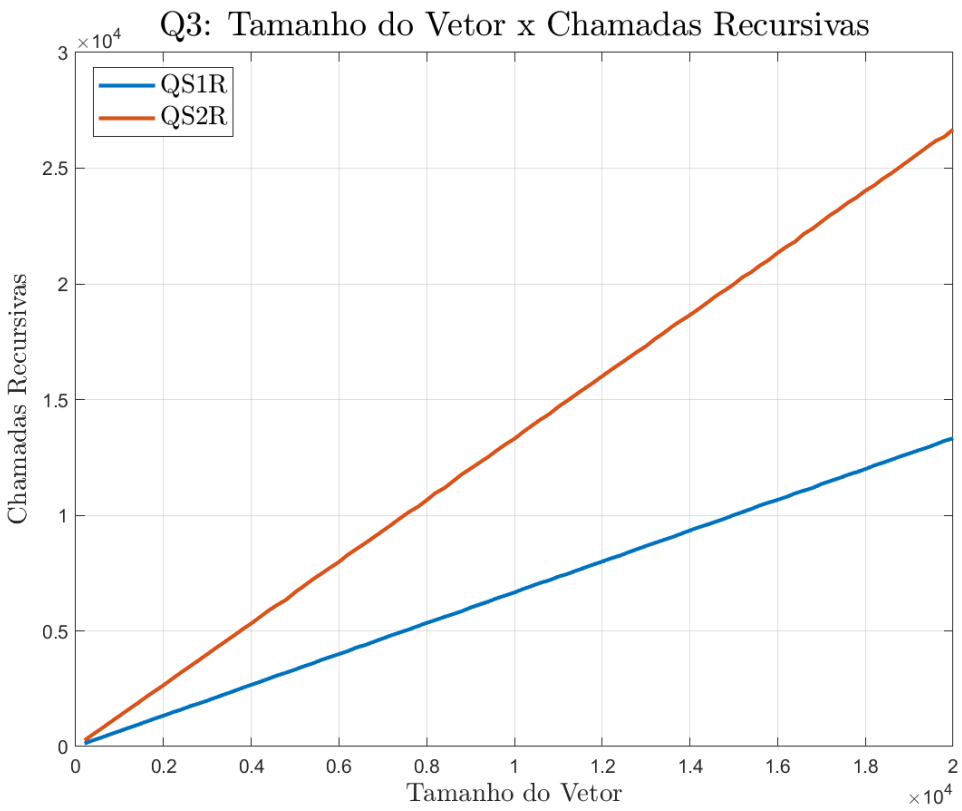

Q2: Numero de Chamadas Recursivas MRGR
×10^4
4
3.5
3
2.5
2
1.5
1
0.5
0
Chamadas Recursivas
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
Tamanho do Vetor
×10^4

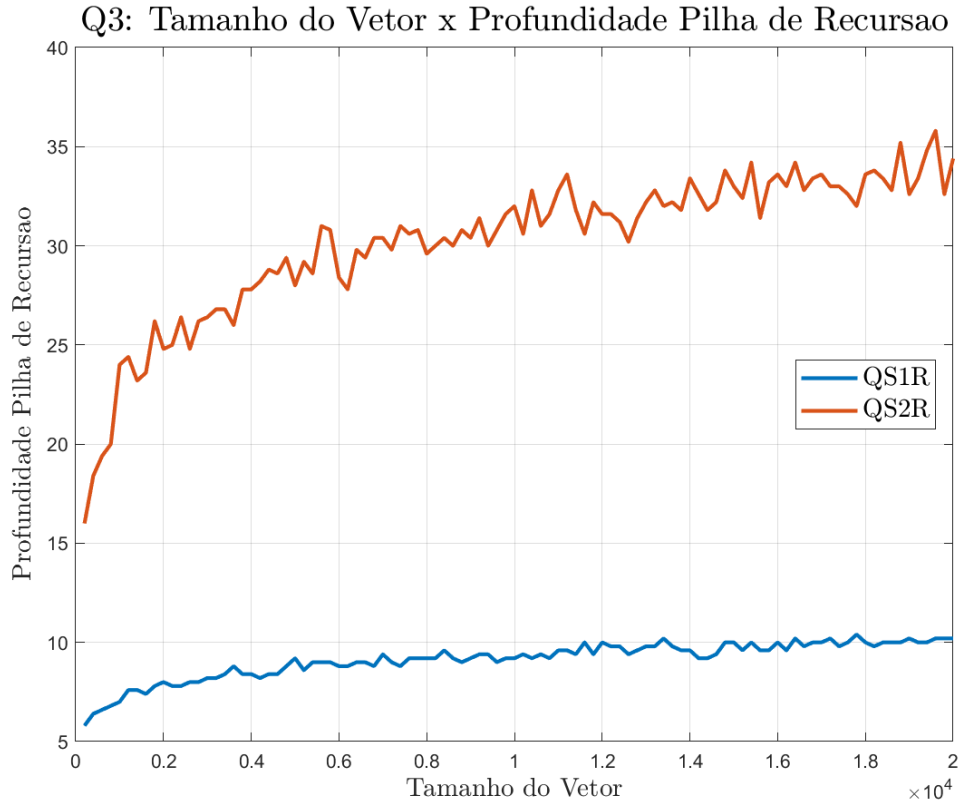

Q2: tamanho da Arvore MRGR
16
15
14
13
12
11
10
9
Profundidade Pilha Recursao
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
Tamanho do Vetor
×10^4

3) QuickSort com 1 recursão x QuickSort com 2 recursões

Nesta comparação, analisamos o QuickSort com 1 recursão (QS1R) e o QuickSort com 2 recursões (QS2R), ambos utilizando mediana de 3, e avaliando-os com vetores randômicos. Em relação ao tempo de execução, ambos apresentam desempenhos muito similares. Entretanto, observa-se uma alocação de memória mais eficiente na pilha de execução do QS1R. Ao compararmos o número de chamadas recursivas e o tamanho da árvore de ambos, a árvore do QS1R é praticamente três vezes menor que a do QS2R, o que indica uma vantagem do QS1R em termos de consumo de memória.



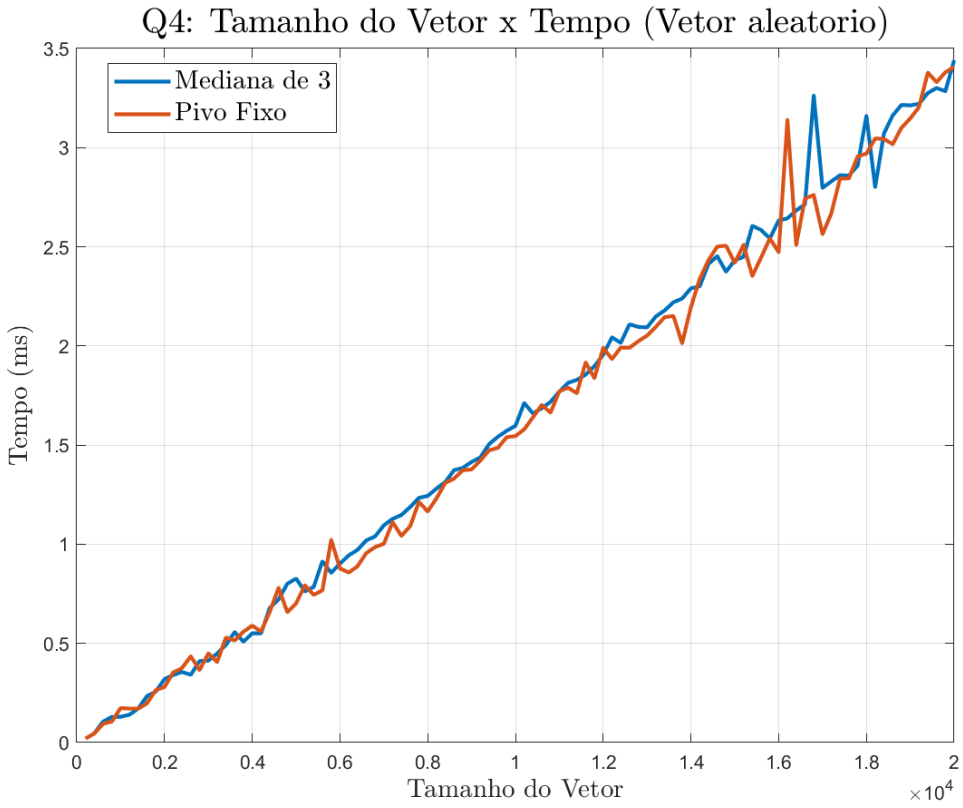

Q3: Tamanho do Vetor x Chamadas Recursivas
×10^4
3
2.5
2
1.5
1
0.5
0
QS1R
QS2R
Chamadas Recursivas
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
Tamanho do Vetor
×10^4

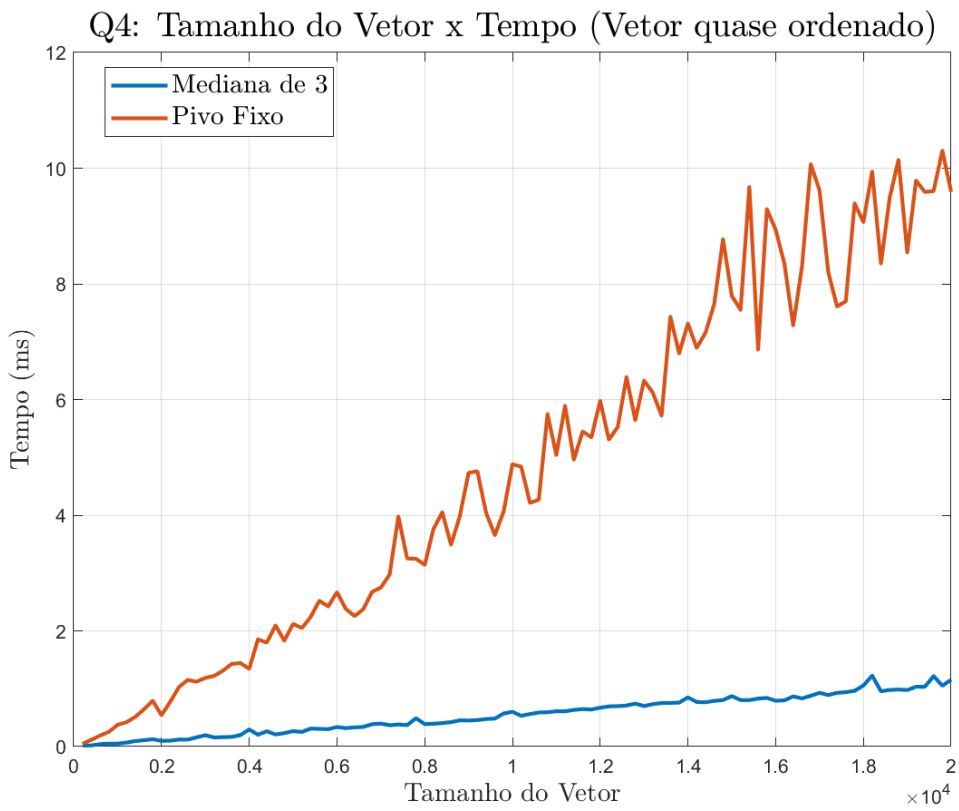

Q3: Tamanho do Vetor x Profundidade Pilha de Recursao
40
35
30
25
20
15
10
5
Profundidade Pilha de Recursao
QS1R
QS2R
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
Tamanho do Vetor
×10^4

4) QuickSort com mediana de 3 x QuickSort com pivô fixo para vetores quase ordenados

Nesta comparação, analisamos o QuickSort com mediana de 3 e o QuickSort com pivô fixo, ambos realizando duas chamadas recursivas. Inicialmente, avaliamos o desempenho dos algoritmos para vetores aleatórios, onde notamos um ligeiro melhor desempenho do QuickSort com mediana de 3 em relação ao pivô fixo. Essa diferença é esperada, pois a mediana de 3 reduz a probabilidade de o QuickSort apresentar um tempo de execução quadrático.

Ao analisarmos os algoritmos para vetores quase ordenados, a diferença de desempenho se torna ainda mais evidente. O pivô fixo apresenta um desempenho pior nesse caso (quadrático), realizando muitas trocas desnecessárias e sendo consideravelmente mais lento que o QuickSort com mediana de 3, que consegue evitar essas trocas ineficientes e mantém um bom desempenho mesmo em vetores quase ordenados.




Q4: Tamanho do Vetor x Tempo (Vetor quase ordenado)
Mediana de 3
Pivo Fixo
Tempo (ms)
Tamanho do Vetor
×10^4
0
2
4
6
8
10
12
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2

5) QuickSort com mediana de 3 x QuickSort com pivô fixo para vetores quase ordenados

De fato, não faz sentido implementar o QuickSort com pivô fixo, considerando que a implementação com mediana de 3 é uma correção simples e torna extremamente raros os casos de tempo de execução quadrático. Entretanto, caso a implementação empregue algo eficiente como a mediana de 3, e os casos quadráticos sejam realmente específicos e incomuns, não vejo problema em utilizá-la.

Nos casos reais, vetores quase ordenados são raros e, se por algum motivo eles se tornarem parte do projeto, cabe à equipe de desenvolvimento solucionar esse problema específico. É preferível resolver um problema específico de maneira pontual do que trocar uma solução muito boa para o caso geral e com ótimo custo-benefício por uma extremamente complexa e com custo-benefício ruim, apenas para atender a uma exceção.