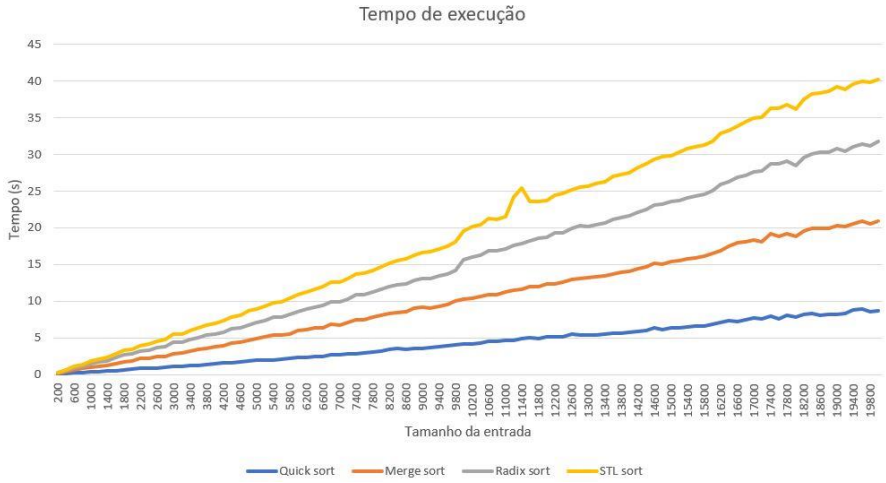
INSTITUTO TECNOLÓGICO DE AERONÁUTICA

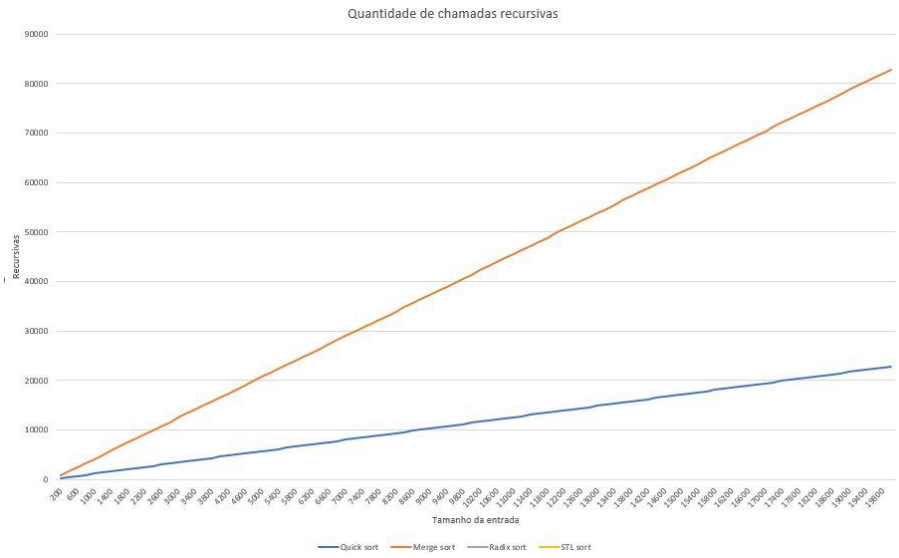
# LABORATÓRIO 3 – SORT

Aluno: Kalil Georges Balech

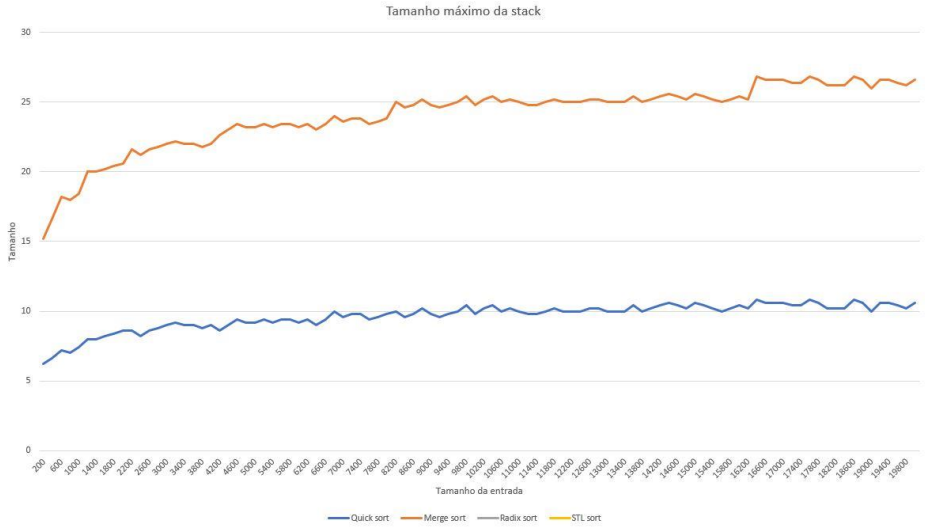
# Q1 QuickSort X MergeSort X RadixSort x std::sort




Com relação ao tempo de execução, percebe-se, claramente que o quick-sort possui o melhor tempo devido às baixas constantes de sua complexidade (n.logn). Ele é seguido diretamente pelo merge-sort, que, apesar de possuir complexidade (n.logn) possui constantes multiplicativas mais altas. Em seguida, tem-se o radix-sort, que, apesar de possuir complexidade linear, possui constantes multiplicativas muito altas, o que faz seu tempo de execução ser desfavorável. Por fim, com o pior tempo, tem-se o sort da lib STL.

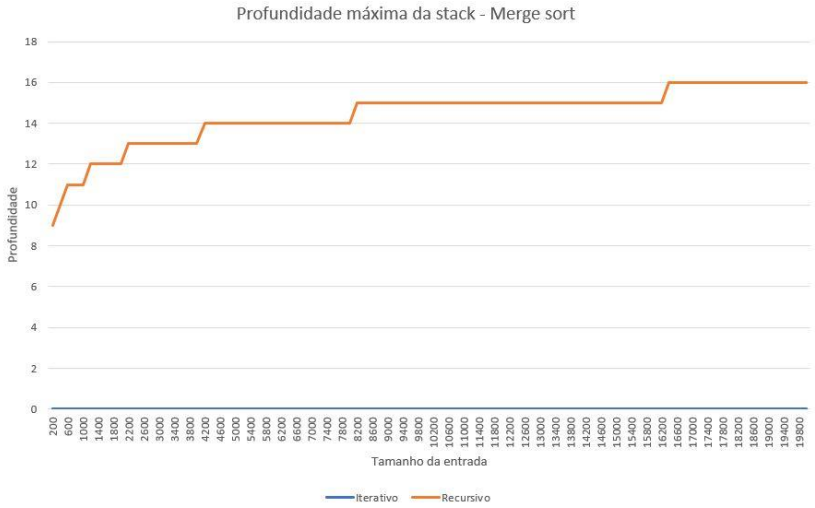



Tanto o algoritmo da STL quanto o Radix sort não são implementados recursivamente, logo ambos possuem total de chamadas recursivas chamadas igual a zero. Comparando quick-sort e merge-sort, temos que o primeiro possui um menor numero de chamadas recursivas dado que tal fator está relacionado de forma direta com a efetividade do algoritmo. Dessa forma, por ser menos flexível, o merge-sort possui um maior numero de chamadas recursivas.

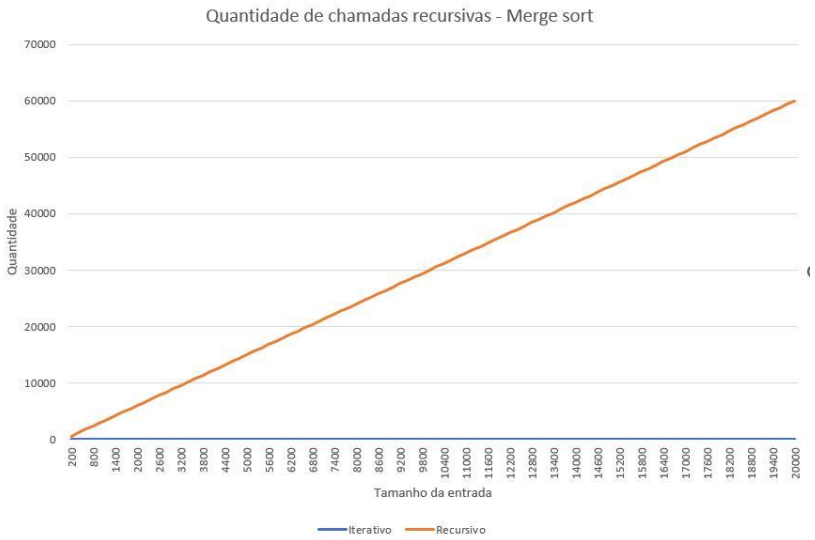



Tanto o merge-sort quanto o quick-sort apresentam tamanhos máximos de Stack com complexidade logarítmica. O tamanho da profundidade da Stack varia bastante para o quick-sort, de forma que para entradas aleatórias, ele possui um tamanho máximo de Stack menor. No entanto, esse comportamento é alterado para entradas de vetores quase ordenados.

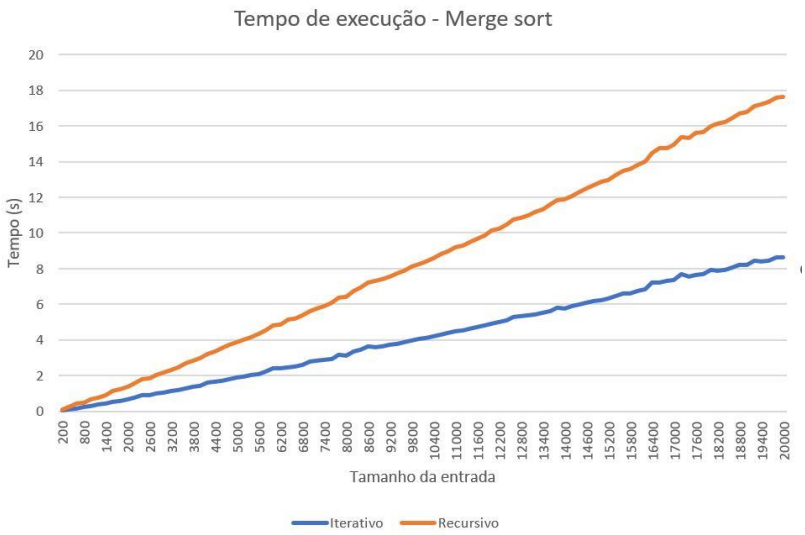
# Q2 MergeSort: Recursivo x Iterativo




Percebe-se que a profundidade máxima da Stack para o algoritmo recursivo possui uma sequencia de pontos de inflexão devido à natureza da recursão.

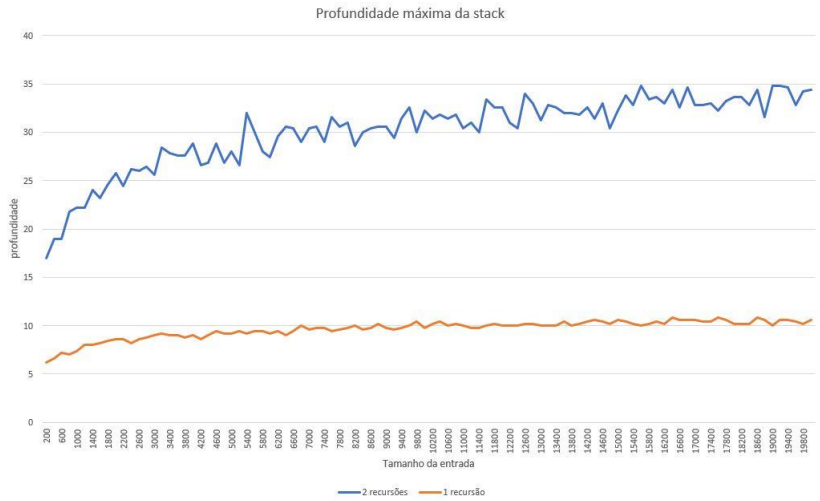



O algoritmo iterativo não realiza nenhuma chamada recursiva de funções por natureza.

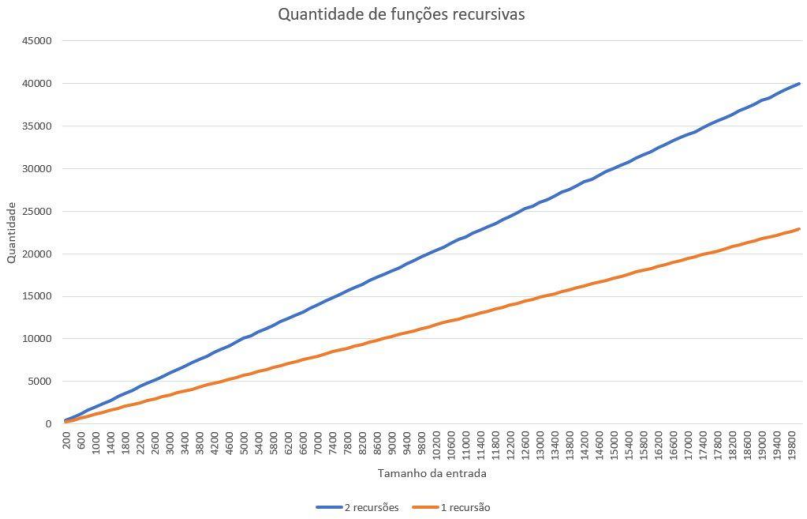



O tempo de execução do algoritmo iterativo é claramente menor devido a não necessidade de gastar tempo com empilhamento e desempilhamento das funções recursivas na Stack.

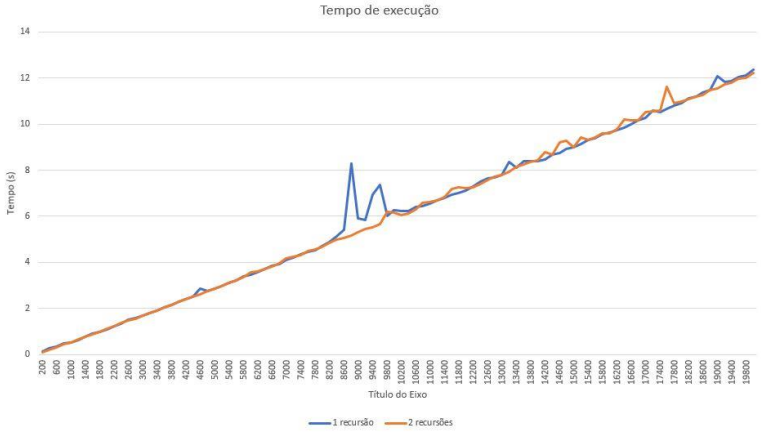
# Q3 - QuickSort com 1 recursão x QuickSort com 2 recursões




A implementação da chamada recursiva apenas para a menor parte de vetor separada pelo pivô é realizada justamente para a diminuição do tamanho máximo da Stack de empilhamento das funções recursivas.



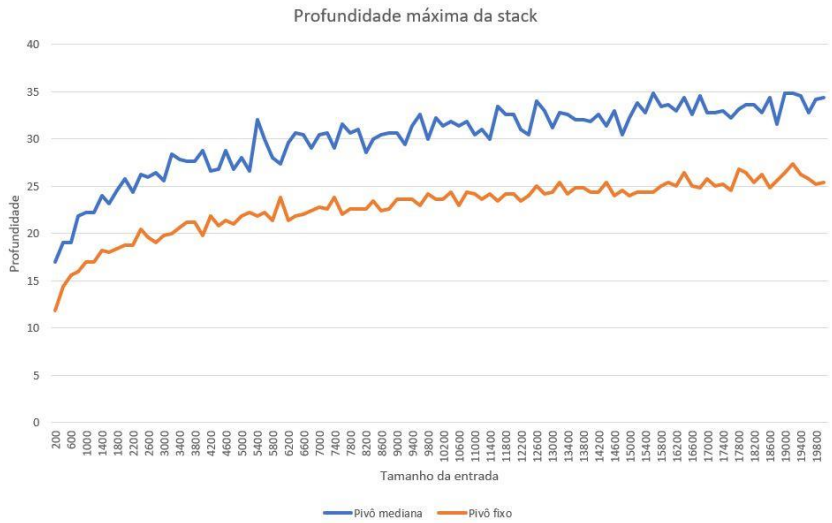

Naturalmente, o algoritmo de chamada recursiva para as duas partes do vetor possui um maior número de chamadas recursivas em sua implementação.



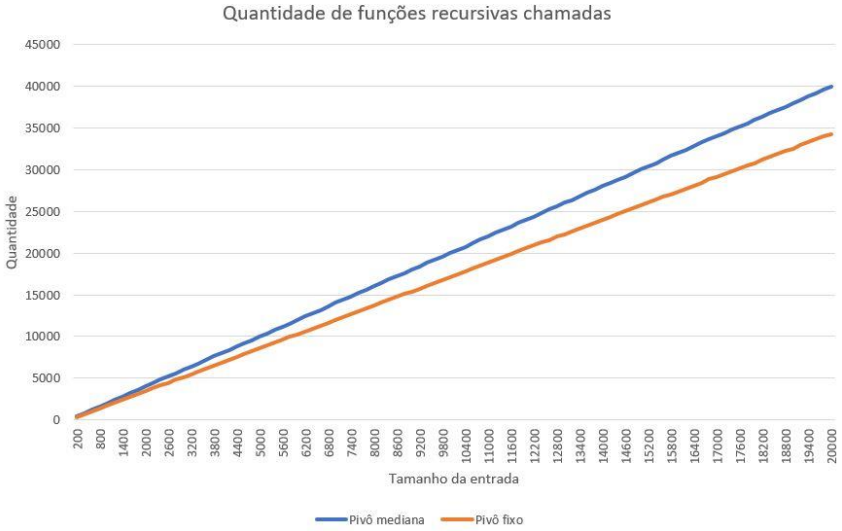

A chamada somente da menor parte do vetor não altera o tempo de execução do algoritmo. Tal recurso apenas possui impacto no tamanho da pilha de execução formada durante a implementação. Dessa forma, os tempos de ambos algoritmos são bastante similares.

# Q4 QuickSort com mediana de 3 x Quicksort com pivô fixo para vetores quase ordenados

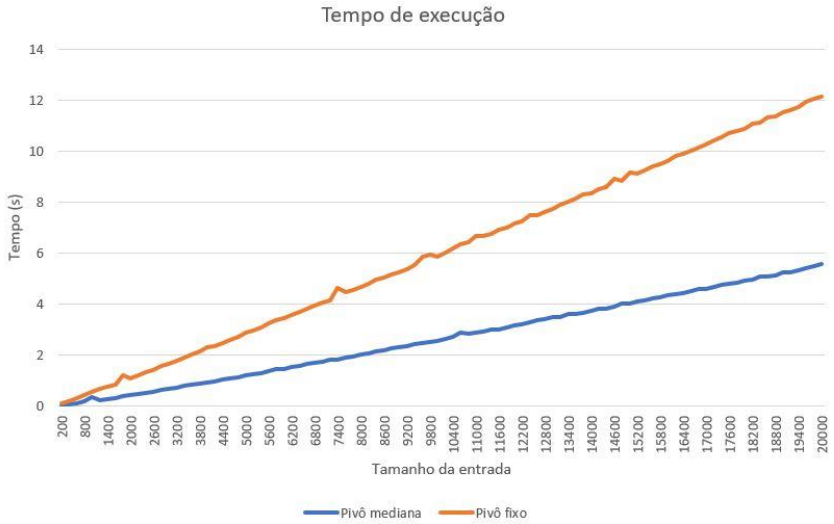
## Dados aleatórios




A profundidade da Stack é menor com o pivô sendo fixo, pois nesse caso, a distribuição dos elementos em torno dele faz com que haja uma tendência de um dos lados ser significativamente menor do que o outro. Dessa forma, a chamada da função quick sort para esse lado tende a se encerrar mais rápido, de modo a não permitir que a pilha de execução cresça tanto.

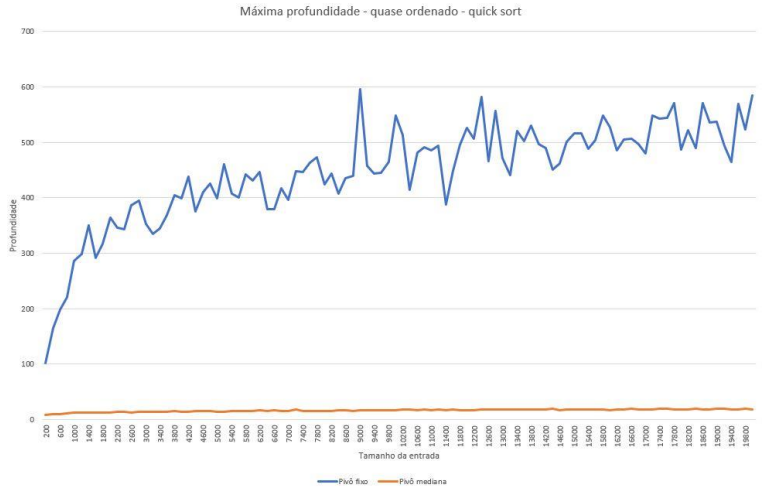


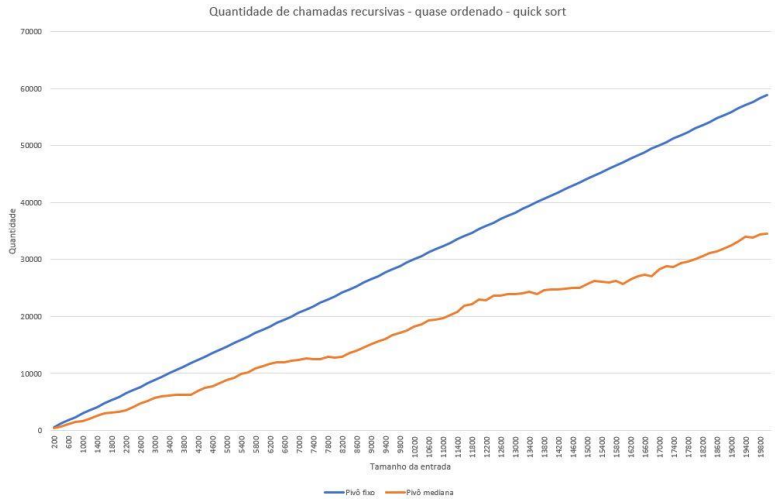

Para o pivô sendo a mediana de 3 elementos, há a geração de um maior número de chamadas recursivas, pois o vetor fica dividido de maneira mais igualitária.

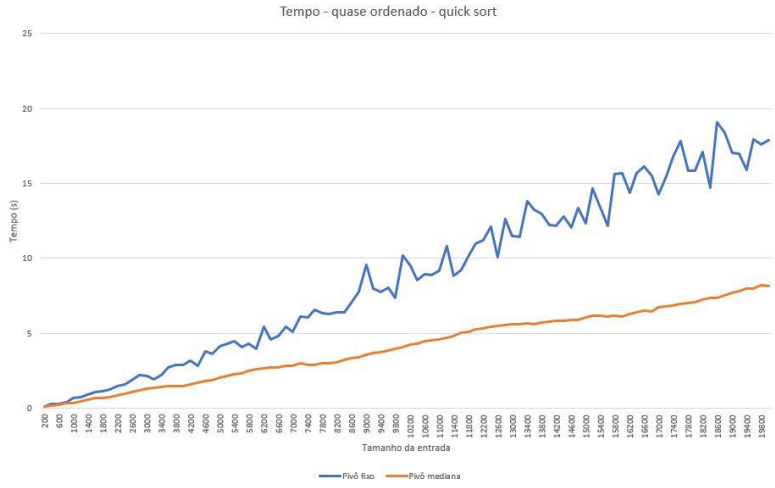



A chamada da função Partition com o Pivô sendo a mediana de 3 elementos melhora a distribuição dos elementos em torno do pivô. Dessa forma, o algoritmo se torna mais efetivo e demora menor tempo de execução.

# Dados quase ordenados




Para casos em que o vetor está quase ordenado, o algoritmo cujo pivô é tido pela mediana entre 3 elementos se mostra mais eficiente em tornas os piores casos do quick sort ainda mais raros.

# Q5 Questão sobre caso real

Não é plausível que uma biblioteca de tanta importância possua um comportamento quadrático quando o vetor estiver quase ordenado. De maneira melhor estruturada, essa biblioteca deveria realizar algum teste prévio que gere alguma informação a respeito da ordenação do vetor. E, assim, escolher o algoritmo mais apropriado considerando essa informação. Assim sendo, como chefe de desenvolvimento de uma equipe, eu não aprovaria tal algoritmo que se mostra bastante ineficiente em casos em que o vetor está quase ordenado, considerando que um pequeno teste pode melhorar o algoritmo nos casos mais extremos de pior desempenho.