Lab CTC12 - 2025

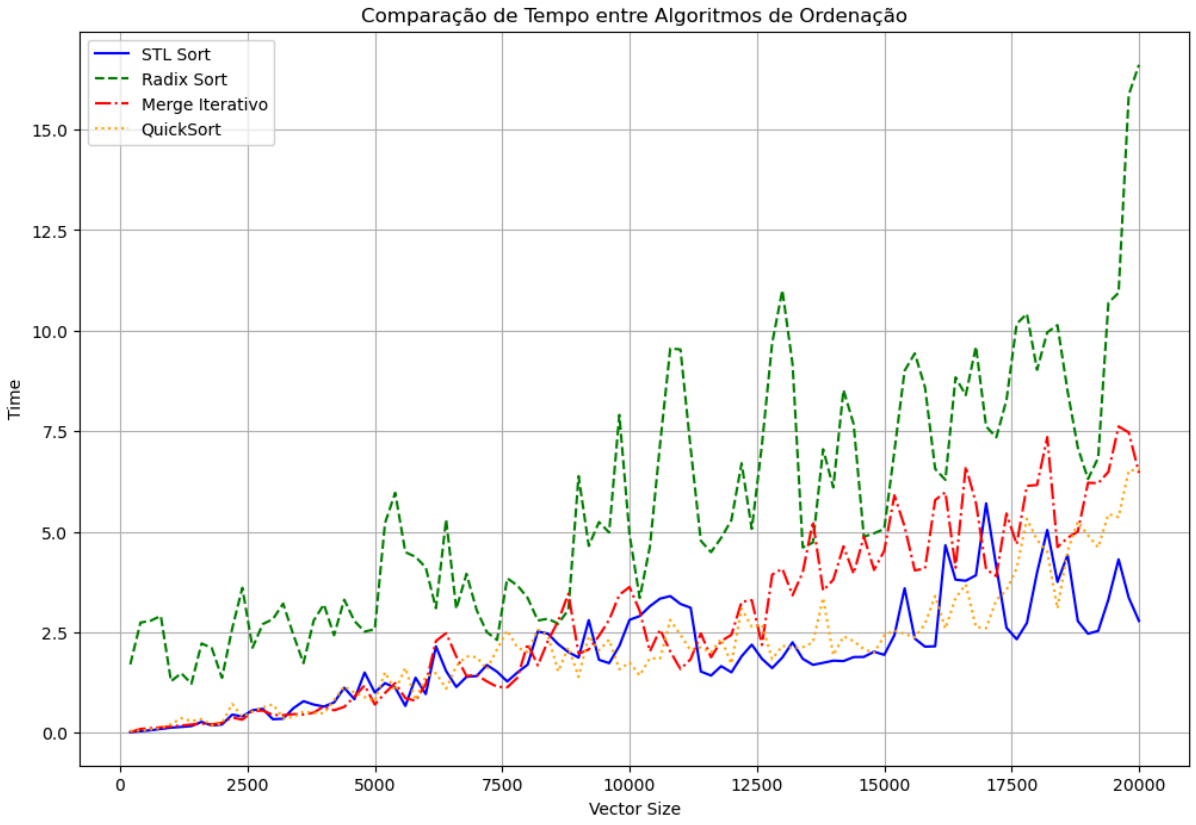
Variações de algoritmos de ordenação

Aluno: Douglas Bergamim Fernandes

Objetivo:

Implementar variações de algoritmos de ordenação, realizar comparações e entender os resultados.

Q1 QuickSort X MergeSort-Iterativo X RadixSort X std::sort



De início, para tamanhos de entrada pequenos, as diferenças de desempenho entre todos os algoritmos, com exceção do RadixSort, são menos acentuadas. Isso está relacionado ao custo fixo de cada implementação. No caso do RadixSort, pode-se observar que as operações realizadas por ele são custosas e tem constantes significantemente maior que dos outros algoritmos. Ademais, para o QuickSort, MergeSort e o próprio STLSort tem diferenças menos acentuadas, pois o custo fixo de cada implementação (chamadas recursivas ou alocação de memória) incide de forma similar para pequenos tamanhos de entrada.

No entanto, à medida que o tamanho do vetor cresce, o gerenciamento de memória e a complexidade do algoritmo impactam mais. Na tabela abaixo, tem os valores de tempo médio obtido para cada algoritmo:

	STLSort	RadixSort	Merge Iterativo	QuickSort 1R
Tempo(s)	1,9379	5,57269	2,798	2,129

O RadixSort, apesar de ter complexidade linear (O(n)), apresentou o pior desempenho em termos de tempo na prática para esses casos simulados, devido ao alto custo constante associado à alocação e manipulação frequente de memória adicional (filas) durante o processamento dos buckets. Apesar disso, observa-se que, com o aumento do tamanho da entrada, a diferença de tempo do RadixSort para os demais algoritmos diminuiu.

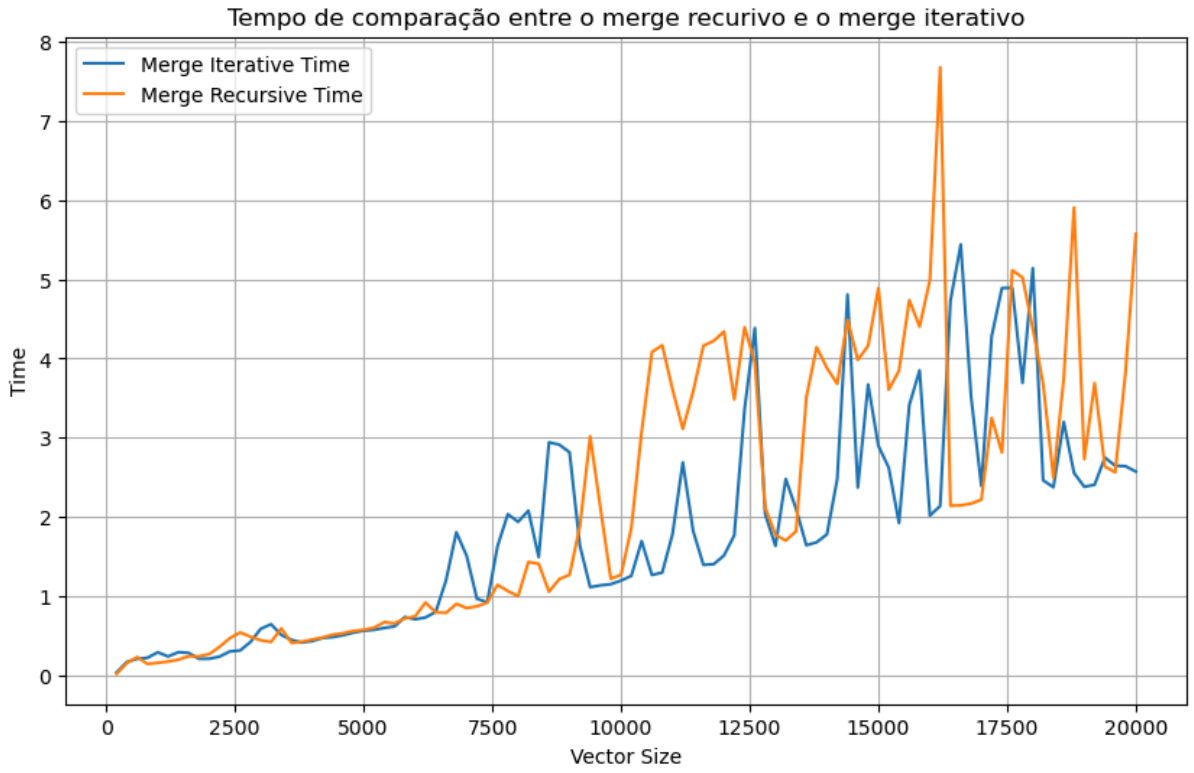
O MergeSort iterativo, com complexidade O(n log n), mostra um comportamento intermediário, em que geralmente tem um desempenho pior que o QuickSort e que o STLSort, mas melhor que o RadixSort. Isso é esperado devido à necessidade constante de alocação de memória proporcional ao tamanho do vetor, o que é mais custoso em questão de tempo e memória. No entanto, em alguns casos, ele possui desempenho melhor que o QuickSort. Isso acontece porque, no pior caso do QuickSort (em um vetor quase ordenado) sua complexidade é O(n^2), enquanto para o MergeSort é sempre O(n log n). Além disso, também percebe-se maior estabilidade para o algoritmo, sem mudanças muito bruscas de gasto de tempo para entradas próximas.

Por último, dos algoritmos implementados neste relatório, o que teve melhor desempenho foi o QuickSort. Com uma complexidade O(n log n) na média, apresentou um comportamento estável e bastante próximo do STL Sort, o que reflete o uso mais eficiente da memória e da pilha de recursão ao utilizar a estratégia de minimizar as chamadas recursivas. Assim como já falado, o algoritmo, para alguns casos, teve desempenho pior que o MergeSort devido ao seu pior caso gastar tempo O(n^2).

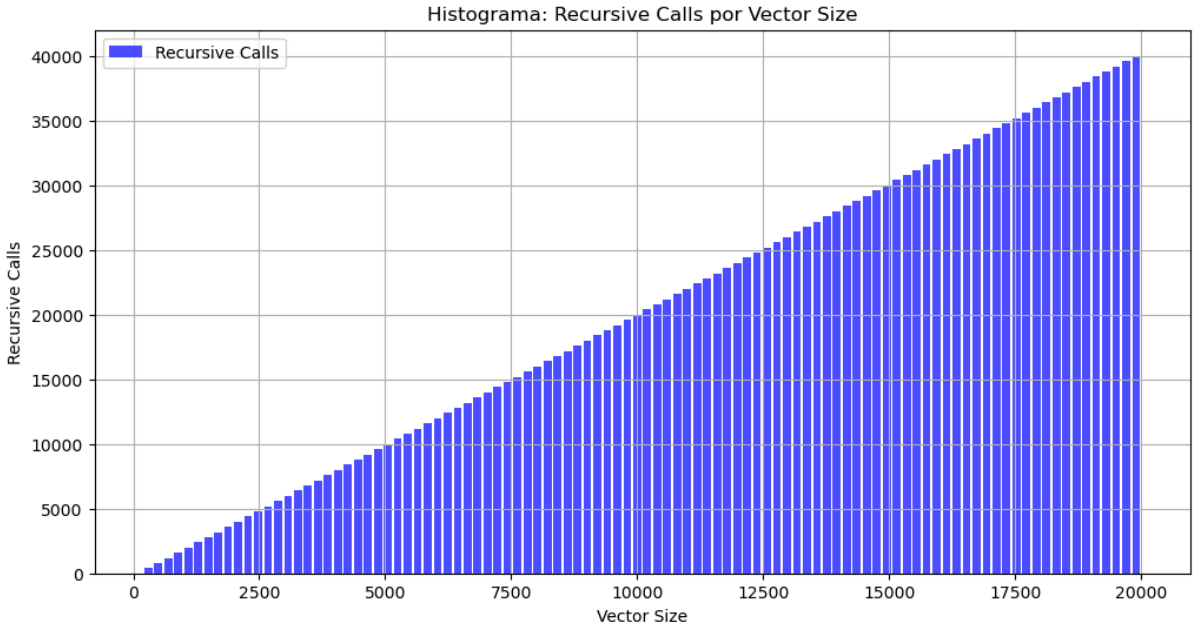
Por último, o STL Sort, que utiliza uma combinação do QuickSort e outros algoritmos, obteve consistentemente o melhor desempenho geral. Em síntese, para pequenos vetores as diferenças entre os algoritmos são pouco significativas. Porém, para vetores maiores, fica evidente que o STL Sort e o QuickSort personalizado têm desempenho mais satisfatório, especialmente devido ao uso eficiente da memória e da pilha. O MergeSort iterativo é uma solução intermediária aceitável quando estabilidade é necessária, enquanto o RadixSort só é competitivo para grandes valores de entrada.

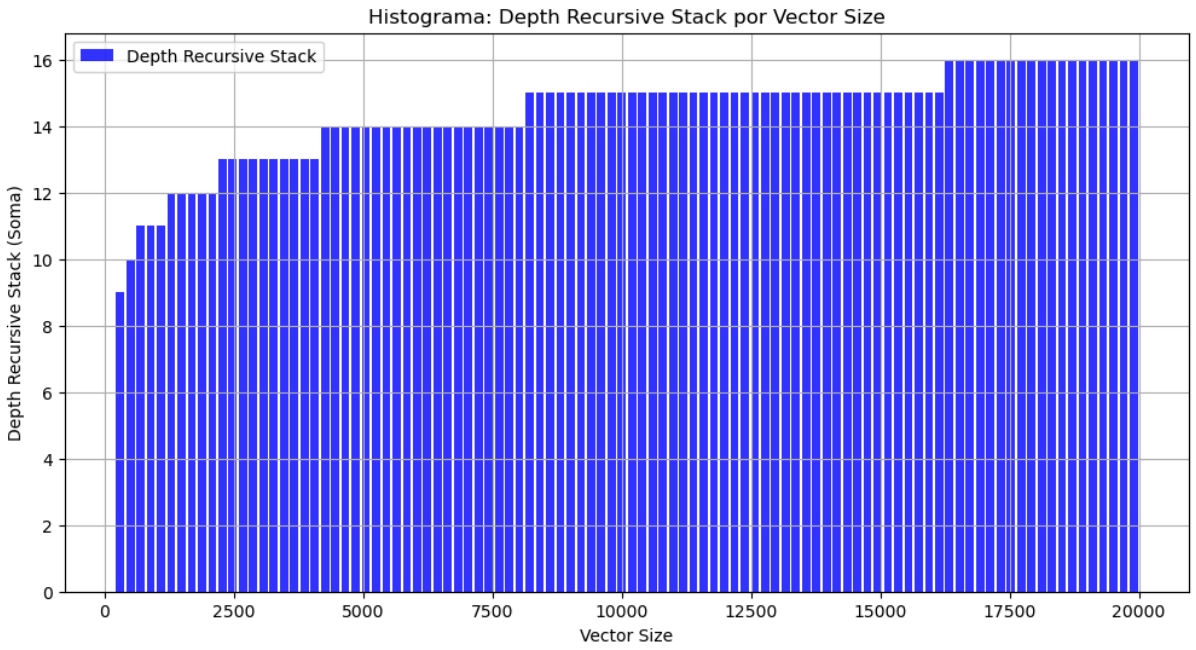
Q2 MergeSort: Recursivo x Iterativo

Seguem os gráficos obtidos do tempo de execução para o Merge Iterativo e o Merge Recursivo.



Além disso, segue a plotagem dos valores obtidos para o número de chamadas recursivas e para a profundidade máxima da árvore de recursão, respectivamente.





Observando-se o gráfico do tempo gasto pelos dois algoritmos, verifica-se que o MergeSort recursivo e o MergeSort iterativo apresentam desempenhos semelhantes para vetores menores, uma vez que a sobrecarga de chamadas de função no recursivo não chega a impactar tanto nessa faixa de entrada e o custo de alocação no iterativo também se mantém reduzido.

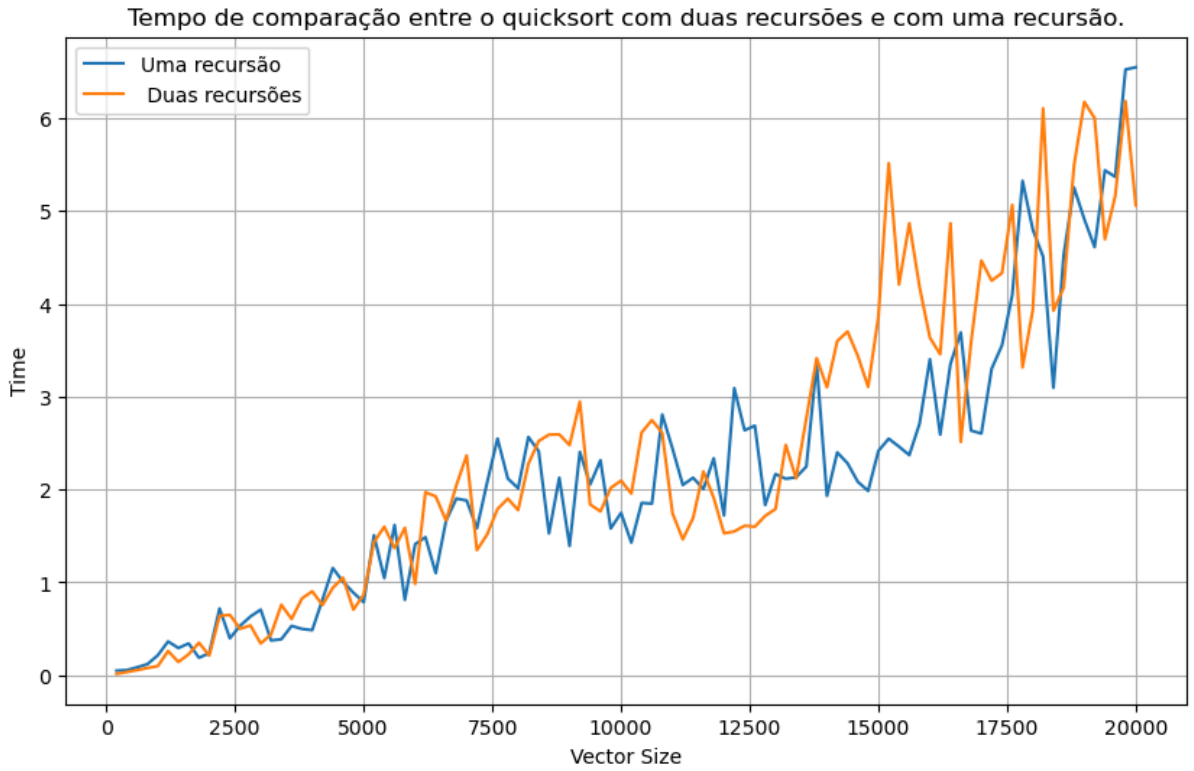
Apesar das complexidade de tempo e espaço não serem alteradas entre os dois algoritmos, à medida que o tamanho do vetor cresce, o MergeSort iterativo mostra-se com melhor desempenho, pois tem menor uso da pilha de execução.

Os histogramas que mostram o tamanho máximo da pilha de recursão e o número de chamadas recursivas por tamanho do vetor justificam o gasto de mais tempo para o MergeSort recursivo. O segundo gráfico, que evidencia a quantidade de chamadas recursivas efetuadas, mostra uma elevação proporcional ao tamanho do vetor (o que é trivial de comprovar teoricamente): enquanto o vetor se expande, o número de partições cresce.

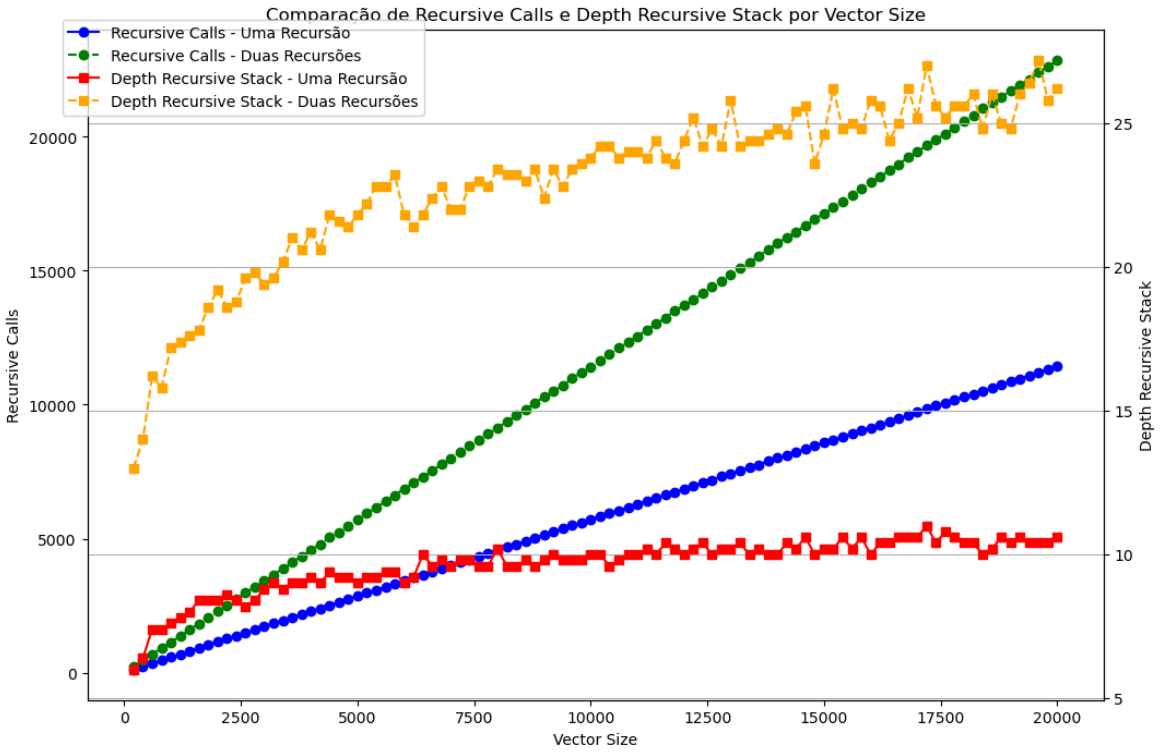
Já o terceiro gráfico revela que a profundidade máxima da árvore de recursão cresce de forma logarítmica o que, embora seja eficiente em termos teóricos, implica em consumo de memória adicional para a pilha de chamadas à medida que o tamanho do vetor aumenta. Em contrapartida, o MergeSort iterativo não incorre na mesma quantidade de custo adicional para chamadas de funções e gerenciamento de pilha.

Q3 QuickSort com 1 recursão x QuickSort com 2 recursões

O gráfico obtido para o tempo gasto pelos dois algoritmos durante a ordenação dos vetores gerados randomicamente:



O segundo gráfico registra a quantidade de chamadas recursivas e do tamanho máximo da pilha de recursão para os dois algoritmos. O eixo esquerdo refere-se à quantidade de chamadas recursivas e o eixo esquerdo, à profundidade máxima da pilha de recursão.



Usando apenas uma chamada recursiva, o QuickSort concluiu a ordenação mais rápido porque reduz drasticamente o tempo dedicado a empilhar e desempilhar as chamadas da função. A cada partição, ele faz recursão só no sub-vetor menor e trata o maior dentro de um laço, evitando metade das transições de contexto que a versão tradicional de duas recursões exige. Esse corte no número de chamadas recursivas explica por que, nos seus testes, o tempo de execução do algoritmo de uma recursão ficou consistentemente abaixo do obtido com duas recursões.

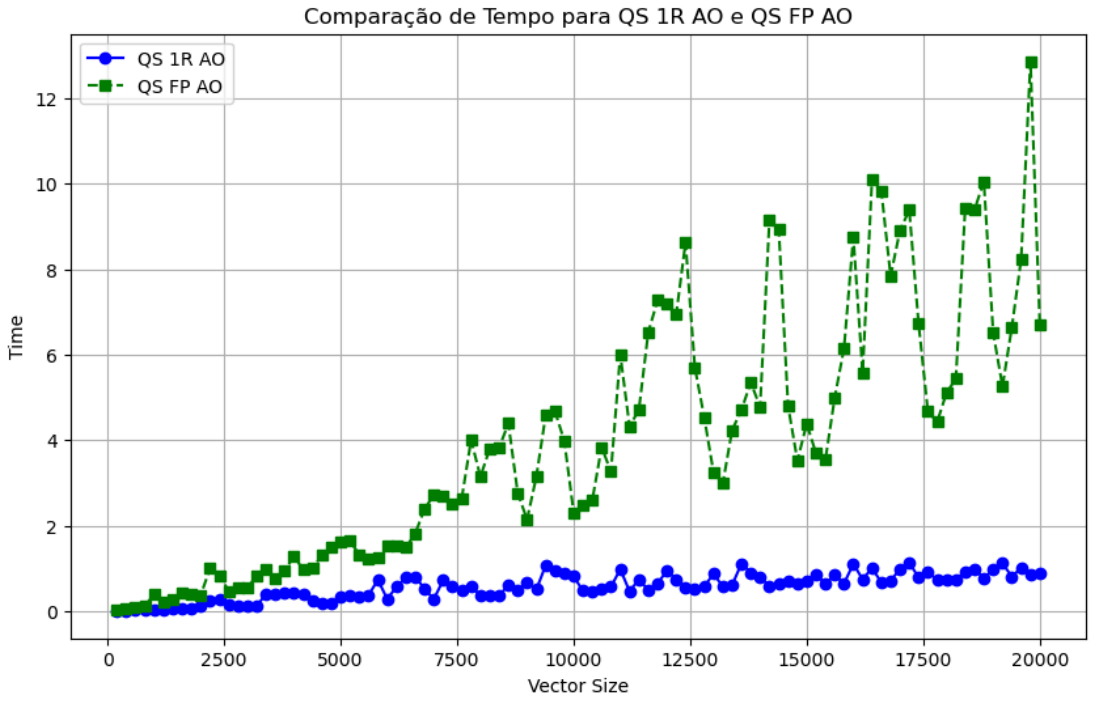
Além disso, a profundidade da árvore de recursão também explica o porquê o QuickSort com uma recursão ter um desempenho melhor. Em termos do tamanho da árvore de recursão, embora ambas mantenham complexidade assintótica de O(log n) em casos médios, o fator constante é bem menor para a estratégia de uma recursão, tornando-a mais segura contra *stack overflow* e melhor em termos de memória. Em resumo, as duas implementações partilham a mesma ordem de grandeza em espaço — O(log n) — e de tempo — O(n log n) —, mas a tail-recursion demanda aproximadamente metade da pilha e, por isso, entrega tempos menores e uso de memória mais eficiente.

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

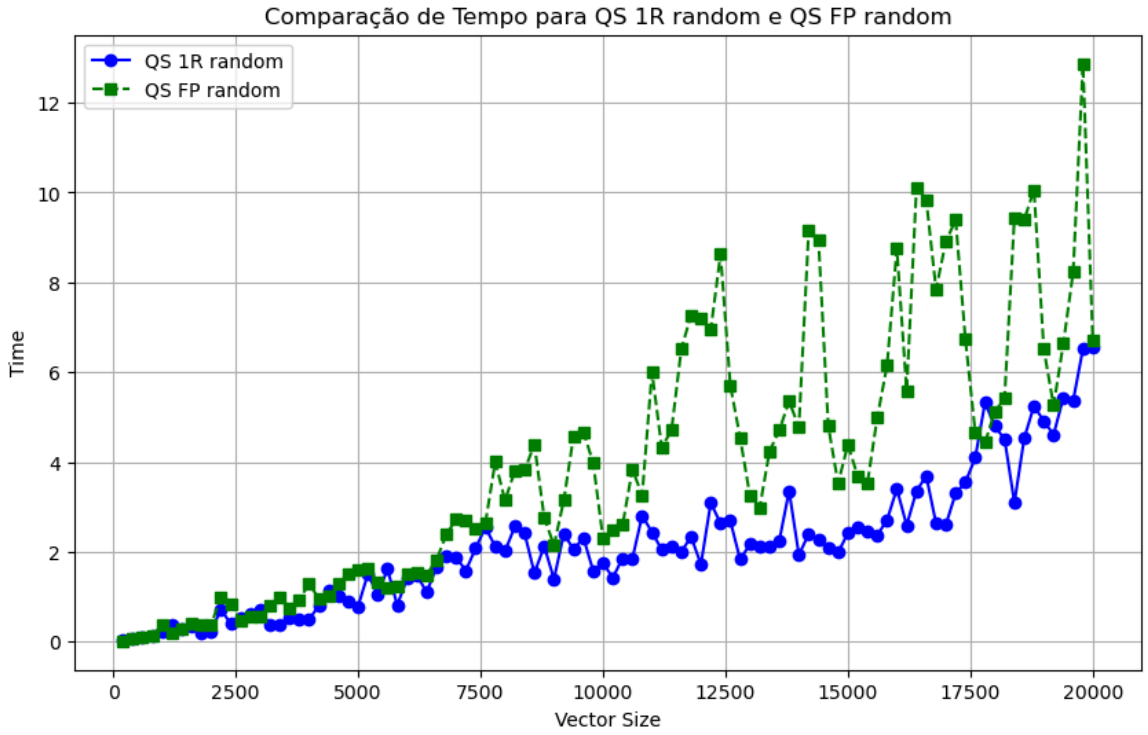
Considerar o caso de entrada aleatória **E** (AND, &&, both, ambos) o caso quase-ordenado.

O caso quase-ordenado é chamado de AO (Almost Ordered) no código.

O gráfico a seguir se refere ao tempo para ordenar vetores quase ordenados utilizando o Quicksort com uma chamada recursiva mas com estratégias de escolhas de pivot diferentes: um escolhe a mediana entre os elementos do ínicio, meio e fim (denotado como ‘QS 1R’ no gráfico) e o outro escolhe sempre o primeiro elemento como vetor (denotado como ‘QS FP’ no gráfico).



O segundo gráfico repete-se a análise, mas com um vetor gerado randomicamente.



Quando o vetor está quase ordenado, escolher sempre o primeiro elemento como pivô é garantir que o QuickSort opere no seu pior caso O(n^2): o primeiro item tende a ser o menor, gerando partições extremamente desbalanceadas (um sub-vetor com n-1 elementos e outro vazio). O resultado é a explosão de partições lineares, tempo de execução próximo de O(n²). Já o algoritmo que elege a mediana entre início, meio e fim, força o pivô a cair perto do centro do intervalo mesmo quando o vetor já está quase ordenado. Isso garante partições muito mais equilibradas, mantém a complexidade prática em O(n log n) e explica por que a curva azul teve valores muito menores para o tempo. O pequeno custo extra de comparar três valores para achar a mediana é constante e rapidamente compensado pela drástica redução de chamadas recursivas e trocas desnecessárias.

Nos vetores aleatórios, a probabilidade de complexidade média para as duas estratégias é O(n log n). Mesmo assim, a estratégia da mediana continua mais eficiente. Esse ganho adicional vem de filtrar pivôs ocasionalmente ruins, reduzindo a variância no tempo total.

Relatório: Q5 Questão sobre caso real:

Atenção: A pergunta fornece um contexto, não importam outras situações.

Caso real de opinião de engenheiro: "*Pode estar demorando muito por culpa da implementação de quicksort da libc [aqui poderia ser outra famosa lib de propósito geral], como alguns dos nossos dados já vem mais ou menos ordenados, fica perto de quadrático*". Certamente é possível, mas é plausível ou esperado que uma biblioteca importante, antiga, muito utilizada e bem testada, e de propósito geral, implemente quicksort da forma citada? Em outras palavras, você, como chefe de equipe de desenvolvimento, aprovaria um quicksort quadrático p/ dados quase ordenados em um projeto real de desenvolvimento de uma lib de propósito geral, **considerando o custo-benefício das alternativas concretas de implementação**? O custo-benefício deve considerar a dificuldade adicional de implementação e teste (será implementado e testado de qualquer forma), o custo computacional em processamento e memória, e os potenciais ganhos e perdas nos casos mais plausíveis para uma lib de propósito geral. Justifique.

Não é esperado que uma biblioteca amplamente testada implemente um Quicksort que tenha um caso O(n^2) para um vetor quase ordenado, já que com apenas uma mudança simples na escolha da estratégia do pivot melhora bastante a performance do algoritmo. Se, de fato, o problema fosse a implementação do QuickSort, eu não aprovaria o uso do quicksort quadrático, uma vez que o processamento de dados é um dos aspectos fundamentais de organização de uma empresa e ele deve ser feito de forma minimamente eficiente. Grandes empresas geram enormes fluxos de dados, o que demoraria muito para ser ordenado usando um Quicksort quadrático e teria um custo enorme de tempo, sendo mais vantajoso implementar uma outra versão do algoritmo, mesmo que gaste mais financeiramente no início.