Relatório de CTC-12 / 2023

Laboratório 03 - Algoritmos de ordenação

Aluno

Marcel Versiani e Silva

Turma COMP.25

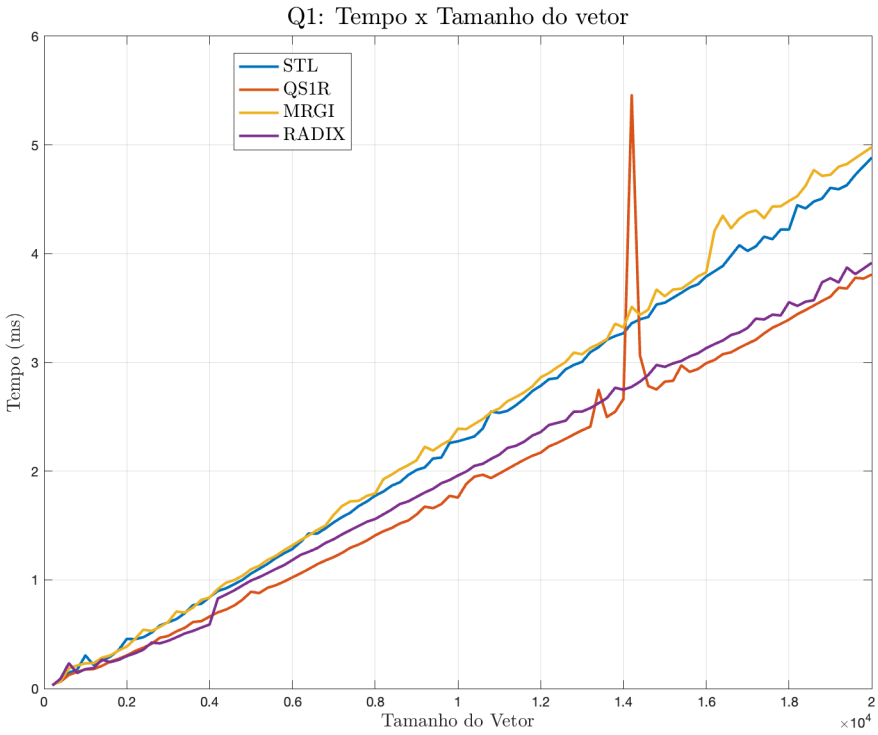
Professor

Luiz Gustavo Bizarro Mirisola

Instituto Tecnológico de Aeronáutica - ITA

1

QuickSort X MergeSort X RadixSort X std::sort.



Os algoritmos analisados no gráfico acima são o da STL, o QuickSort com uma recursão e mediana de 3 como pivô (QS1R), o MergeSort Iterativo (MRGI) e o RadixSort (RADIX).

Pode-se perceber que o RadixSort e o QuickSort foram mais rápidos que o padrão da STL (apesar do QS1R possui alguns *outliers* numa pequena faixa de vetores), enquanto que o MergeSort se saiu ligeiramente pior.

O Radix, apesar de possui complexidade no tempo $\Theta(n)$ em relação ao $O(nlogn)$ médio do QS1R, se saiu um pouco pior que este por ter constantes de multiplicativas mais altas, o que era de esperar pela literatura, dada principalmente pela manipulação de filas durante o código.

Não é de se estranhar que a STL obteve um dos piores desempenhos, por se tratar de um código robusto e generalista, que deve ser bom o suficiente para a maioria dos tipos de dados a serem usados.

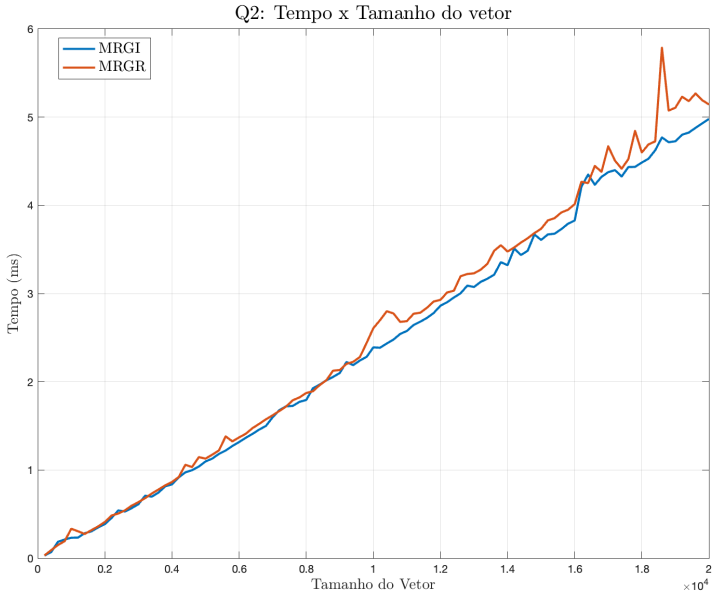
A complexidade $\Theta(nlogn)$ do MergeSort induz à possibilidade de ser mais rápido que o QuickSort, mas este utiliza de operações computacionalmente rápidas ao seu favor, principalmente quando usado em memória primária e também não utiliza vetores auxiliares. O MergeSort, no entanto, se sai melhor em memórias secundárias, que não é este caso, além de manipular vetor auxiliar.

Ressalta-se que apesar do QS1R ser uma versão com apenas uma chamada recursiva, este ainda possui a limitação da pilha de recursão gerada à complexidade de espaço de $O(logn)$ assim como o MergeSort Iterativo, o que deve ser levado em conta em termos de memória do projeto, já que a versão com duas recursões do QuickSort é ainda pior com complexidade de espaço $O(n)$.

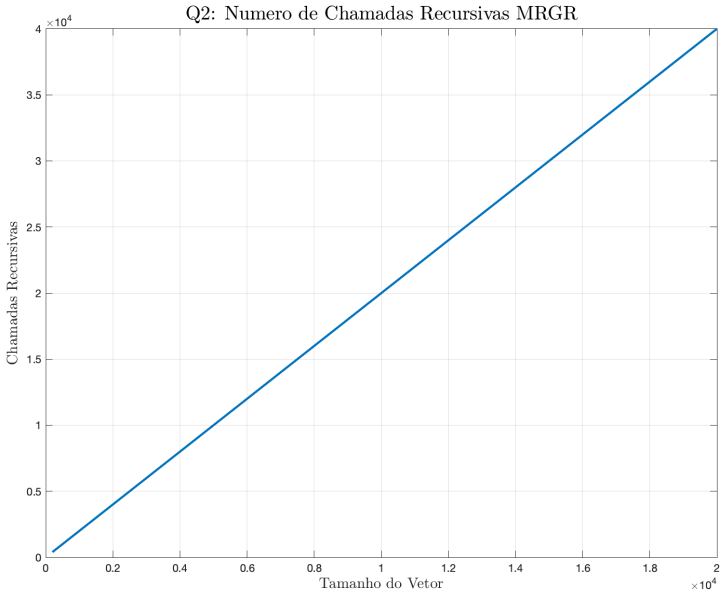
2

MergeSort: Recursivo X Iterativo.

Os algoritmos presentes no gráfico abaixo são o MergeSort Recursivo (MRGR) e o MergeSort Iterativo (MRGI). Observa-se que o algoritmo iterativo se saiu ligeiramente melhor que o recursivo em termos de tempo, pois como não precisa realizar chamadas recursivas, economiza-se alocações e desalocações na pilha de memória e várias chamadas de função, o que diminui as constantes multiplicativas na complexidade de tempo. Ademais, utilizar o MRGI elimina as alocações de vetores adicionais na memória, com isso, há vantagens claras em termos de consumo de memória ao se utilizar o MergeSort Iterativo.



O gráfico abaixo mostra a quantidade de chamadas recursivas realizadas no MRGR, enquanto que o MRGI não possui esse problema com a pilha de execução e com maiores gastos de memória.

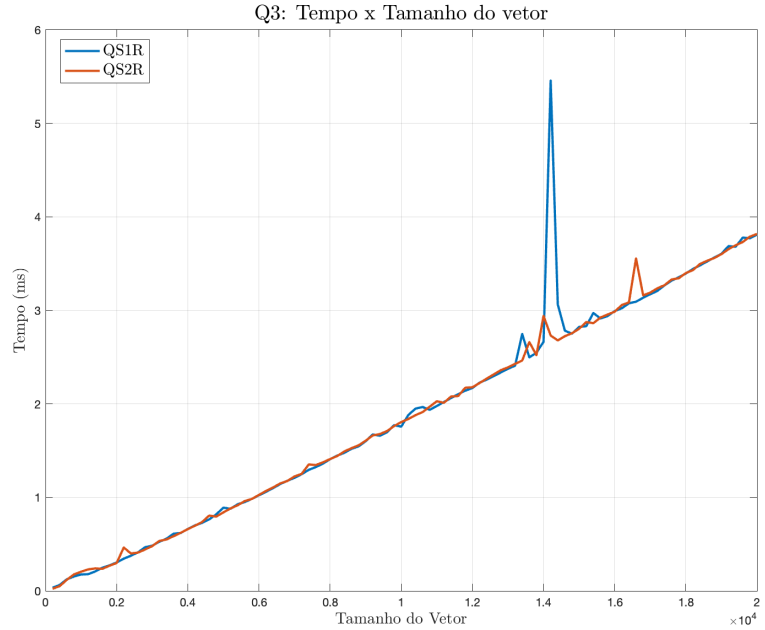


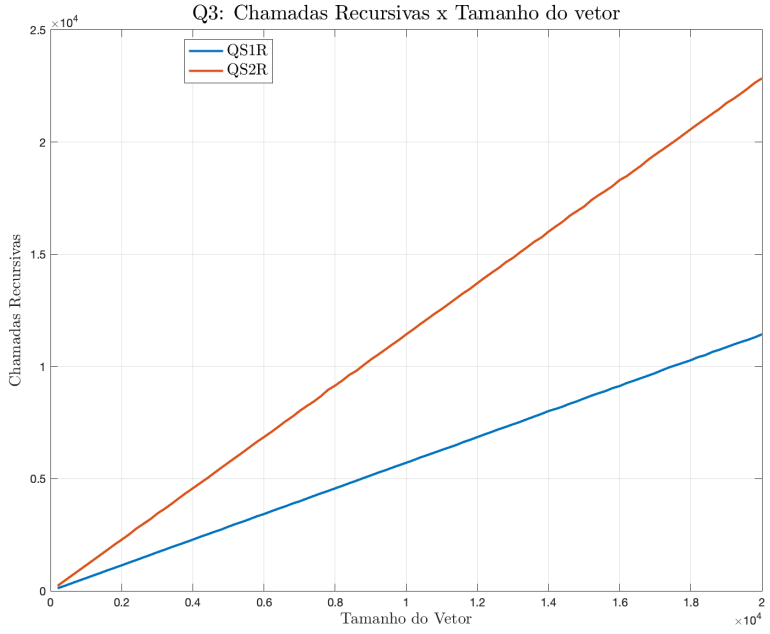
3

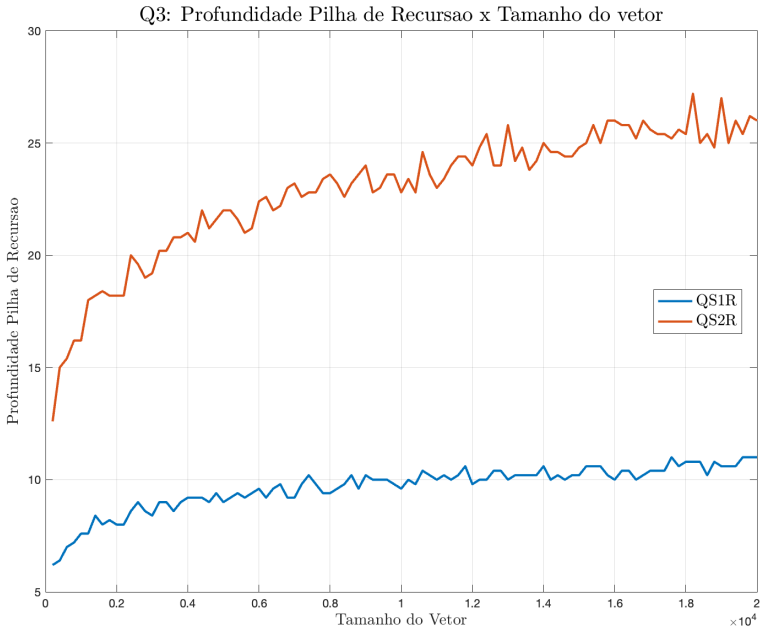
QuickSort com 1 recursão X QuickSort com 2 recursões.

Os algoritmos presentes nos gráficos abaixo são o QuickSort com 1 recursão (QS1R) e o QuickSort com 2 recursões (QS2R), ambos utilizando o pivô como a mediana de 3. Comparando-se os tempo de execução, ambos apresentam desempenhos praticamente iguais, com exceção dos *outliers* do algoritmo com uma recursão presentes numa pequena faixa de vetores.

Contudo, há uma alocação de memória mais eficiente na pilha de execução do QS1R, que tem complexidade de espaço $O(logn)$, se comparado à do QS2R com complexidade $O(n)$. Em relação ao número de chamadas recursivas, a árvore do QS1R é mais de três vezes menor do que a do QS2R, uma clara vantagem do uso do QS1R em termos de consumo de memória primária.







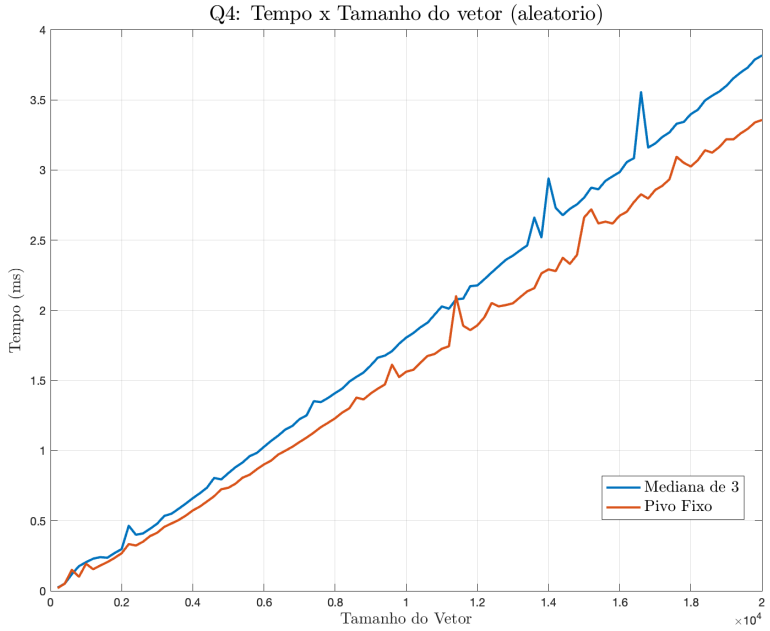
4

QuickSort com mediana de 3 X Quicksort com pivô fixo para vetores quase ordenados.

Os algoritmos presentes nos gráficos abaixo são o QuickSort com mediana de 3 e o QuickSort com pivô fixo, ambos realizando duas chamadas recursivas. Para dados providos de vetores randomizados, nota-se um desempenho um pouco melhor em questão de tempo quando usa-se a mediana de 3, esta que reduz a probabilidade do algoritmo apresentar um tempo de execução quadrático para o pior caso ao evitar a escolha de um pivô ruim.

Para vetores quase ordenados, a diferença de desempenho se torna bastante evidente. O pivô fixo apresenta um desempenho pior nesse caso sempre $\Theta(n^2)$, pois realiza um grande número de trocas sem eficiência e com isso é consideravelmente mais lento que o QuickSort com mediana de 3, que consegue evitar a escolha de um pivô ruim ao obter o médio dentre três opções de pivôs contidos no vetor. Com isso, a escolha de usar a mediana de 3 prova-se uma boa mudança por ter custo computacional baixo e melhorar o algoritmo, evitando os piores casos.

Q4: Tempo x Tamanho do vetor (aleatorio)



Q4: Tempo x Tamanho do vetor (quase ordenado)