Laboratório 3: Variações de Algoritmos de Ordenação

Caio Graça Gomes

24 de Maio de 2020

1 Introdução

Neste laboratório, foram implementadas variações de três algoritmos distintos de ordenação com o objetivo de comparar seus desempenhos, seja analisando o tempo despendido na ordenação, ou seja analisando o espaço de memória ocupado.

2 *Merge Sort*

Para a implementação do algoritmo de *Merge Sort*, utilizou-se duas variações deste: a implementação recursiva e a interativa.

2.1 Recursivo

A principal ideia da implementação recursiva, é dividir o vetor na metade, chamar a função recursiva para cada uma das metades, e depois unir essas metades em um único vetor. Assim, a implementação em $C++$ foi:

```
void mymergesort_recursive(std::vector<int> &v, SortStats &stats) {
    std::vector<int> ordered_vector(v.size());
    merge_recursive(0, v.size()-1, ordered_vector, v, stats);
}

void merge_recursive(int begin, int end, std::vector<int> &ov,
     std::vector<int> &v, SortStats &stats){
```

```
    if(begin<end){
        int middle = (begin+end)/2;
        merge_recursive(begin, middle, ov, v, stats);
        merge_recursive(middle+1, end, ov, v, stats);
        merge(begin, middle, end, ov, v);
    }
}
```

Cuja função merge é dada por:

```
void merge(int begin, int middle, int end, std::vector<int> &ov,
     std::vector<int> &v){
    int iterator = begin, iterator1 = begin, iterator2 = middle+1;
    while(iterator<=end){
        if(iterator1>middle) ov[iterator++] = v[iterator2++];
        else if(iterator2>end) ov[iterator++] = v[iterator1++];
        else{
            if(v[iterator1]<=v[iterator2]) ov[iterator++] =
                v[iterator1++];
            else ov[iterator++] = v[iterator2++];
        }
    }
    for(int j = begin; j <= end; ++j) v[j] = ov[j];
}
```

Sendo assim, submetendo o algoritmo a testes com vetores aleatórios de variados tamanhos, obtém-se o seguinte gráfico do tempo gasto em função do tamanho do vetor ordenado:

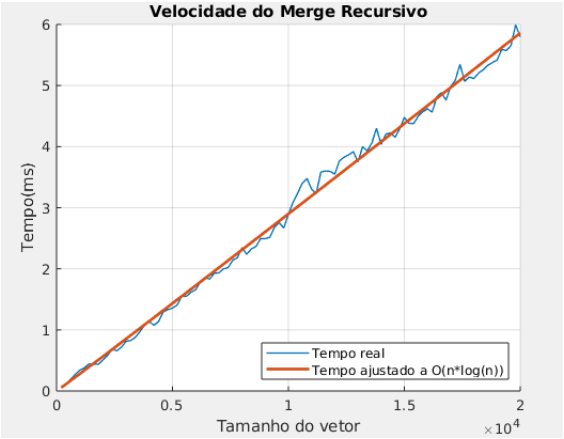

Figure 1: Tempo despendido do *Merge Sort* recursivo

2.2 Iterativo

A ideia do *Merge Sort* iterativo é a mesma do recursivo, porém utilizando uma linha de raciocínio *bottom up* ao invés de *top down*. Ordenando vetores de tamanho pequeno e dobrando o tamanho até que se chegue ao tamanho total do vetor que deve ser ordenado. Assim, tem-se:

```
void mymergesort_iterative(std::vector<int> &v, SortStats &stats) {
    int begin = 0, end = v.size() - 1;
    std::vector<int> ordered_vector(v.size());
    int from, mid, to;
    for (int s = 1; s <= end - begin; s = (s << 1))
    {
            for (int i = begin; i < end; i += (s << 1))
            {
                    from = i;
                    mid = i + s - 1;
                    to = std::min(i + (s << 1) - 1, end);
                    merge(from, mid, to, ordered_vector, v);
            }
    }
}
```

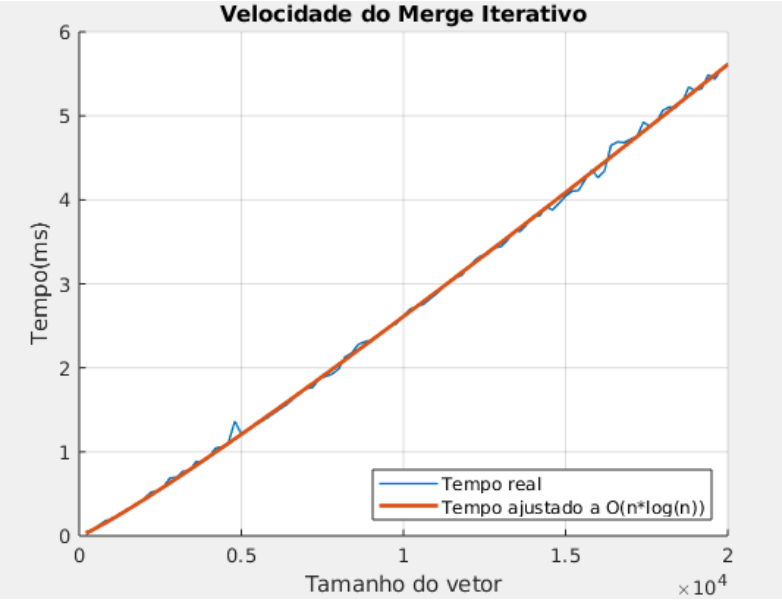

Figure 2: Desempenho temporal do *Merge Sort* iterativo

Pode-se ainda, comparar seus desempenhos:

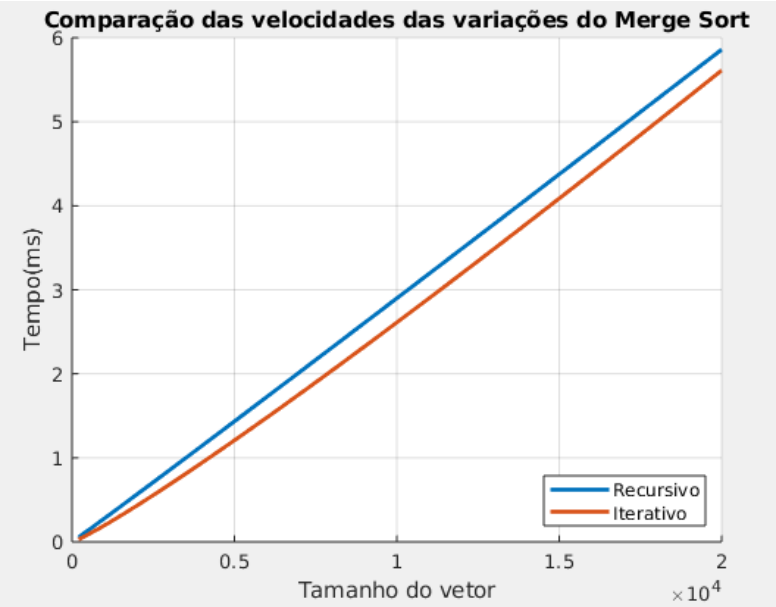

Figure 3: Comparação do desempenho temporal das duas variações do *Merge Sort*

Evidenciando que a versão iterativa é sutilmente mais rápida, além de não montar uma pilha de recursão, sendo portanto, mais recomendável.

3 *Quicksort*

A ideia principal do *Quicksort* é tomar um elemento do vetor, que será chamado de pivô, e fazer comparações em torno dele, de modo que todos os elementos menores fiquem à sua esquerda, e todos os maiores à direita. Após isso, chamar a função recursivamente em cada um dos subvetores gerados (elementos à esquerda e elementos à direita).
No entanto, fazer duas chamadas recursivas não é algo realmente necessário, basta gerar um laço, escolher o pivô, fazer as comparações necessárias, chamar recursivamente nos elementos à esquerda do pivô, e mudar o início do vetor para o elemento após o pivô, até que este não exista.
Evidentemente, a maneira como é escolhida este pivô, influencia no desempenho do algoritmo de ordenação. Com isto, serão utilizados dois métodos: Escolhendo o primeiro elemento do vetor a ser ordenado como pivô, e escolhendo o elemento do começo, do meio, e do fim, e tomando a mediana deles como pivô. Este segundo método será favorável para os casos em que o vetor está praticamente ordenado.

3.1 Duas chamadas recursivas com primeiro elemento como pivô

```
void myquicksort_fixedPivot(std::vector<int> &v, SortStats &stats) {
    quicksort_2recursion(v, 0, v.size()-1, stats, first_element);
}

void quicksort_2recursion(std::vector<int> &v, int begin, int end,
     SortStats &stats, int pivot_function (std::vector<int>&, int,
     int)){
    stats.recursive_calls++;
    stats.custom1++;
    if(begin<end){
        int pivot = partition(v, begin, end, pivot_function);
        quicksort_2recursion(v, begin, pivot - 1, stats,
            pivot_function);
        quicksort_2recursion(v, pivot + 1, end, stats,
            pivot_function);
    }
}

int partition(std::vector<int> &v, int begin, int end, int
     pivot_function(std::vector<int>&, int, int)){
    int pivot = v[pivot_function(v, begin, end)];
    int i = begin - 1;
    for(int j = begin; j < end; ++j){
        if(v[j] <= pivot){++i; troca(v, i, j);}
    }
    troca(v, i + 1, end);
    return i + 1;
}

int first_element(std::vector<int> &v, int begin, int end){troca(v,
     begin, end); return end;}
```

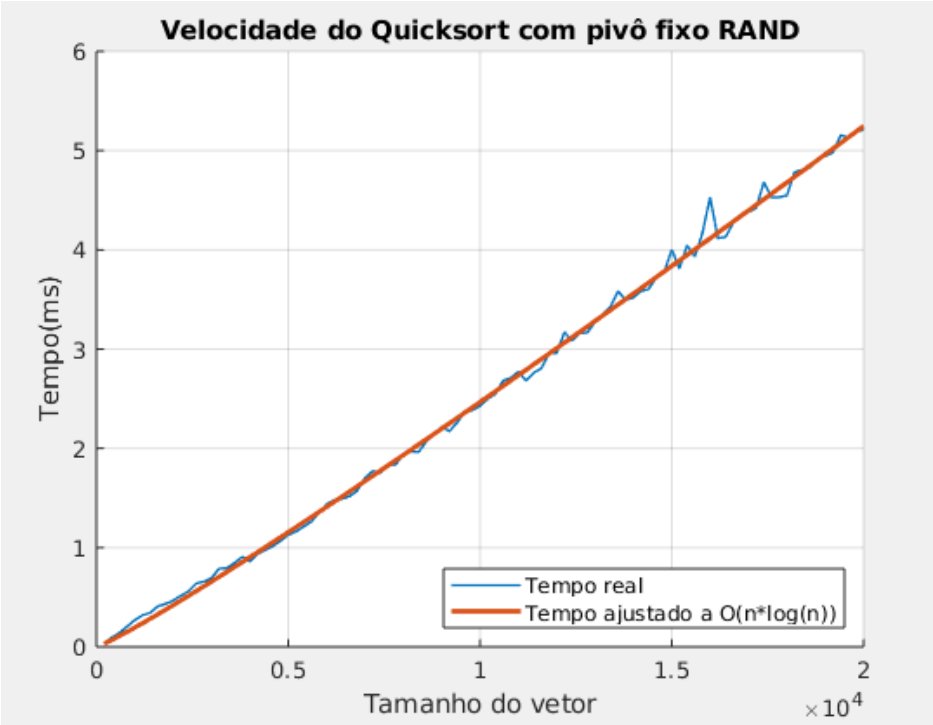

Figure 4: Velocidade do *Quicksort* com duas recursões e primeiro elemento pivô

3.2 Duas chamadas recursivas utilizando mediana de três

```
void myquicksort_2recursion_median0f3(std::vector<int> &v, SortStats
    &stats) {
    quicksort_2recursion(v, 0, v.size()-1, stats, median3);
}

int median3(std::vector<int> &v, int begin, int end){
    int pivot1 = begin, pivot2 = (begin+end)/2, pivot3 = end;
    if((v[pivot1] >= v[pivot2] && v[pivot1] <=
        v[pivot3])||(v[pivot1] >= v[pivot3] && v[pivot1] <=
        v[pivot2])) troca(v, pivot1, end);
    else if((v[pivot2] >= v[pivot1] && v[pivot2] <=
        v[pivot3])||(v[pivot2] >= v[pivot3] && v[pivot2] <=
        v[pivot1])) troca(v, pivot2, end);
    else troca(v, pivot3, end);
    return end;
}
```

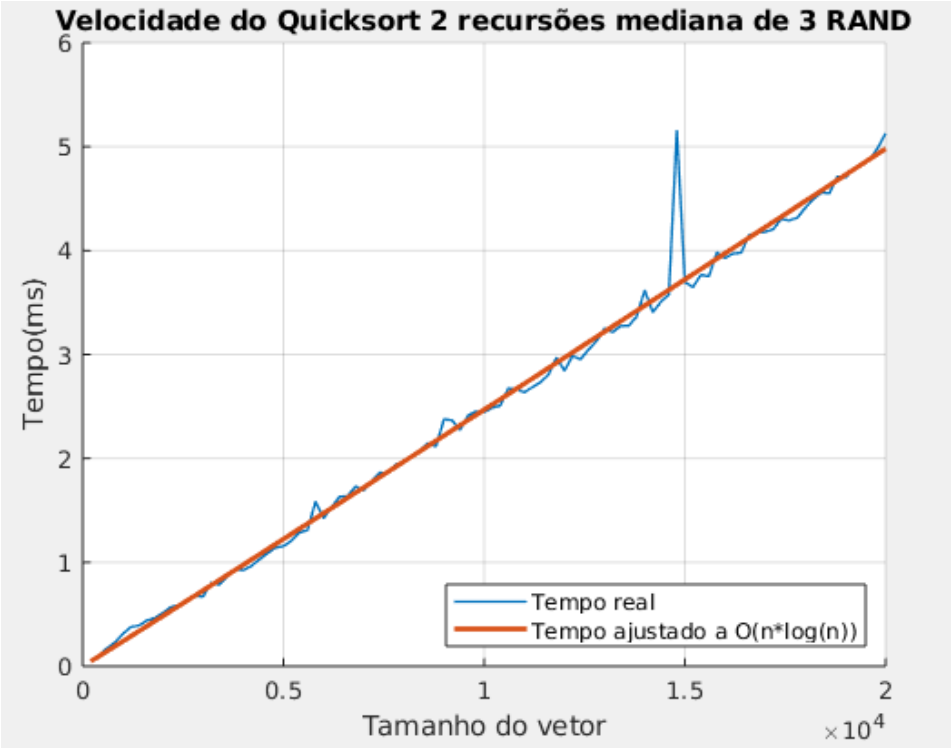

Figure 5: Velocidade do *Quicksort* com duas recursões e mediana de três

3.3 Uma chamada recursiva utilizando mediana de três

```
void myquicksort_1recursion_medianOf3(std::vector<int> &v, SortStats
    &stats) {
    quicksort_1recursion(v, 0, v.size()-1, stats, median3);
}

void quicksort_1recursion(std::vector<int> &v, int begin, int end,
    SortStats &stats, int pivot_function (std::vector<int>&, int,
    int)){
    int pivot;
    while(begin<end){
        pivot = partition(v, begin, end, pivot_function);
        quicksort_1recursion(v, begin, pivot - 1, stats,
            pivot_function);
        begin = pivot + 1;
    }
}
```

Figure 6: Velocidade do *Quicksort* com uma recursão e mediana de três

E ainda, pode-se comparar os desempenhos das variações do *Quicksort*:

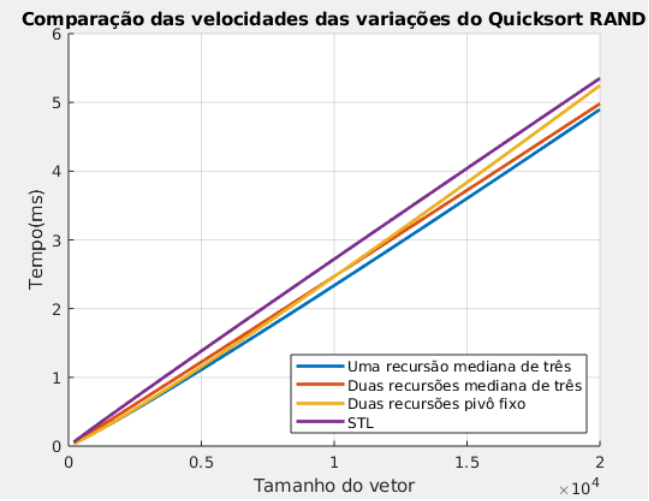

Figure 7: Comparação da velocidade das três variações do *Quicksort* em teste com vetores aleatórios

Quando submetidos a testes aleatórios, a implementação da biblioteca padrão *STL* é a mais lenta, pois é muito mais geral e possui mais acessos a escopos diferentes, enquanto os *Quicksorts* com mediana de três são os mais rápidos com desempenho semelhante.

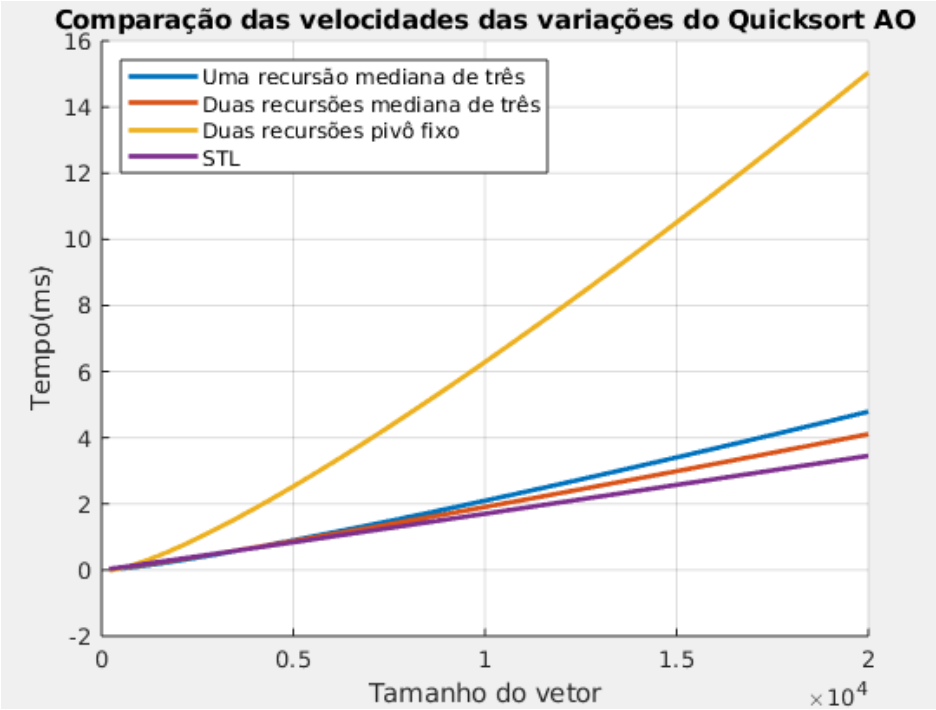

Figure 8: Comparação da velocidade das três variações do *Quicksort* em teste com vetores quase ordenados

No entanto, quando submetidos a testes com vetores quase ordenados, percebe-se que o *Quicksort* com pivô fixo é consideravelmente mais lento que os demais. Além disso, o *Quicksort* com duas recursões foi mais rápido que o com uma recursão, enquanto o algoritmo de ordenação da biblioteca STL foi o mais rápido. Portanto, em casos que se sabe que os vetores sempre serão aproximadamente aleatórios, é mais indicável utilizar a implementação do *Quicksort* com mediana de três em relação aos demais. Porém, quando não se tem conhecimento disto, é preferível usar a implementação da STL.

4 *Radix Sort*

A principal ideia do *Radix Sort* é ordenar os números baseando-se nos dígitos dos elementos no vetor em uma determinada base. Deve-se implementar um algoritmo que ordene dos dígitos menos significativos para os mais significativos de modo a manter a ordem dos elementos empatados.
Para implementação, utilizou-se base hexadecimal e um vetor de filas para armazenar os números que possuem determinado dígito. Note que usar filas será vantajoso pois é possível fazer remoções e consultas no seu começo e inserções no seu final em $O(1)$.

```
#include<queue>

void myradixsort(std::vector<int> &v, SortStats &stats) {
    std::vector<std::queue<int>> rows(16);
    int selector = 0xF;
    for(int i = 0; (1 << (i << 2)) < v.size(); ++i){
        for(int j = 0; j < v.size(); ++j) rows[(v[j] & selector)>>(i
            << 2)].push(v[j]);
        int counter = 0;
        for(int j = 0; j < 16; ++j){
            while(!rows[j].empty()){
                v[counter] = rows[j].front();
                rows[j].pop();
                counter++;
            }
        }
        selector = selector << 4;
    }
}
```

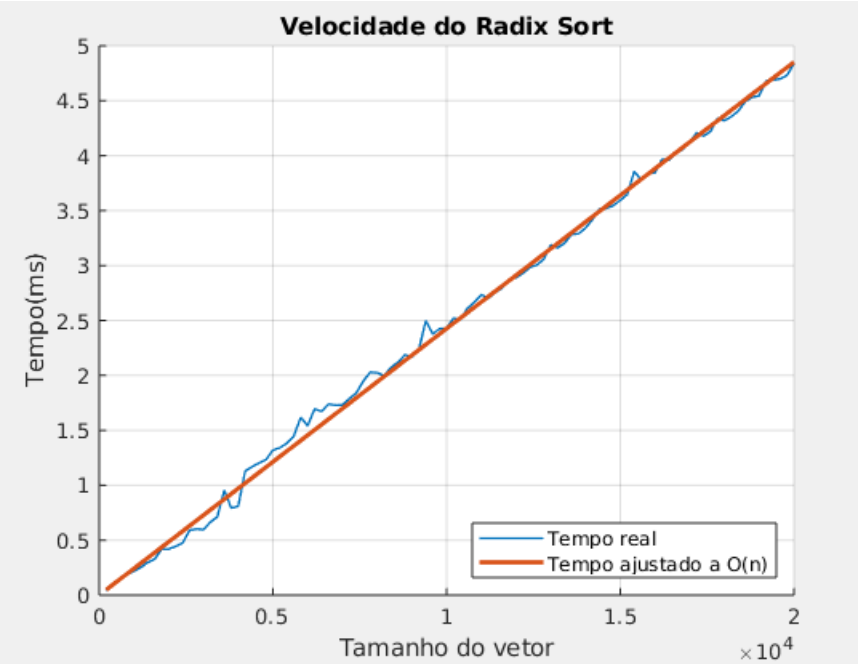

Figure 9: Velocidade do *Radix Sort* em vetores de variados tamanhos

5 Comparação dos algoritmos

5.1 Número de chamadas recursivas

Os algoritmos *Quicksort* e *Merge Sort* recursivo apresentam recursões em suas implementações, isto pode ser indesejado, pois um número elevado de chamadas recursivas pode ser prejudicial para um bom desempenho. Assim:

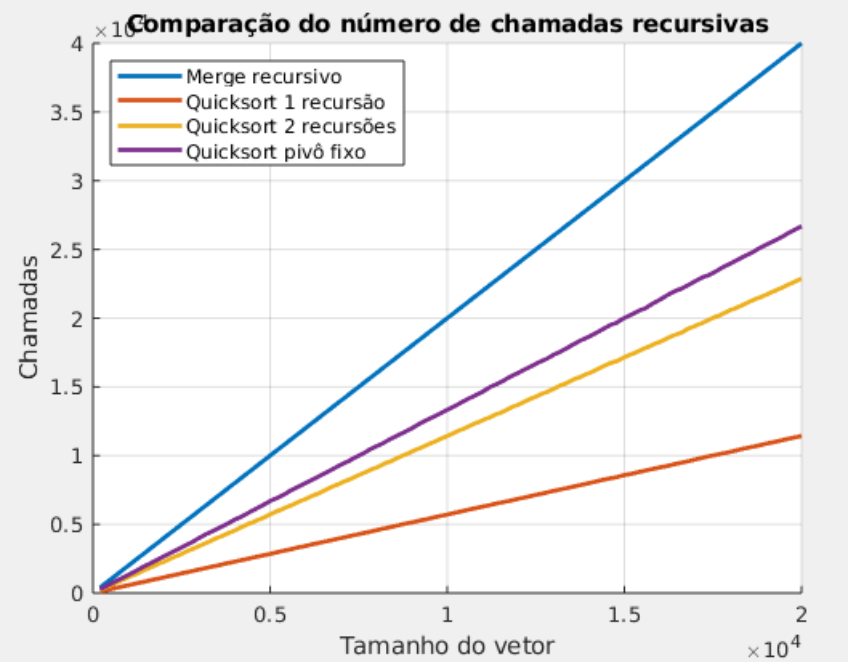

Figure 10: Comparação do número de chamadas recursivas

Como o *Merge Sort* recursivo sempre divide o vetor ao meio e chama recursivamente em cada metade, é esperado que o número de chamadas seja o dobro do tamanho do vetor. O *Quicksort* com duas recursões apresenta o dobro de chamadas que o com uma recursão e o *Quicksort* com pivô fixo faz mais chamadas que os outros pois há uma maior probabilidade de o pivô ser um elemento mais distante da mediana do vetor, desbalanceando as chamadas recursivas.

5.2 Tamanho da pilha de recursão

Os algoritmos *Merge Sort* recursivo, *Quicksort* com 2 recursões e *Quicksort* com pivô fixo apresentam profundidade de pilha de recursão $O(log(n))$. Uma pilha de recursão com alta profundidade é muito indesejado pois pode causar um estouro da pilha de recursão.

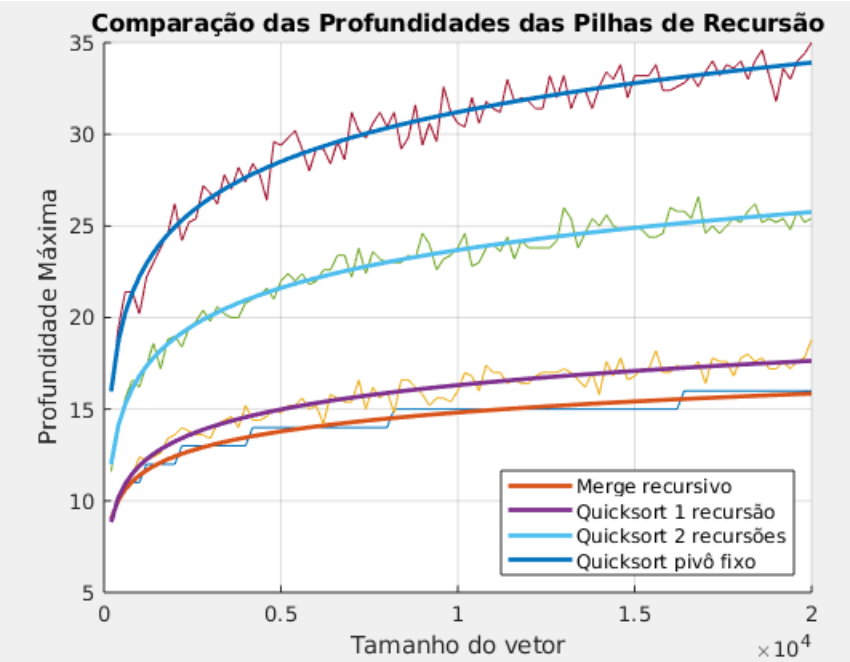

Figure 11: Comparação do tamanho máximo da pilha de recursão

O *Merge Sort* possui a menor profundidade de pilha de recursão, enquanto o *Quicksort* com pivô fixo apresenta a maior, pelos mesmos motivos do número de chamadas recursivas.

5.3 Tempo

Por fim, a análise dos tempos despendidos na ordenação por todos os algoritmos mencionados (Figura 12):
É evidente que os *Merge Sorts* foram os mais lentos para a ordenação de vetores aleatórios, sendo eles menos recomendados pois, além disso, ocupam um espaço de memória $O(n)$ devido à alocação de um vetor auxiliar.
As variações do *Quicksort* com método de mediana de três foram os algoritmos que obtiveram melhor desempenho temporal, no entanto a implementação com uma recursão pode ser preferível pois alcança profundidade menores na pilha de recursão e realiza menos chamadas recursivas, vide figuras 10 e 11.
O *Quicksort* com pivô fixo teve desempenho semelhante ao algoritmo da STL, mas foi inferior às outras versões do *Quicksort*, sendo ele menos desejável. Já a implementação do STL, embora tenho tido desempenho inferior às outras variações do *Quicksort*, teve melhor desempenho em vetores quase ordenados, conforme Figura 8.
Portanto, o algoritmo mais preferível para ordenação depende das entradas es-

peradas e das limitações de *hardware* de um projeto. Para vetores completamente aleatórios é preferível utilizar *Quicksort* com uma recursão e método de mediana de três, mas se o algoritmo é recorrentemente acionado e deve ordenar vetores que estão aproximadamente ordenados, então a implementação da biblioteca padrão *STL* é preferível.

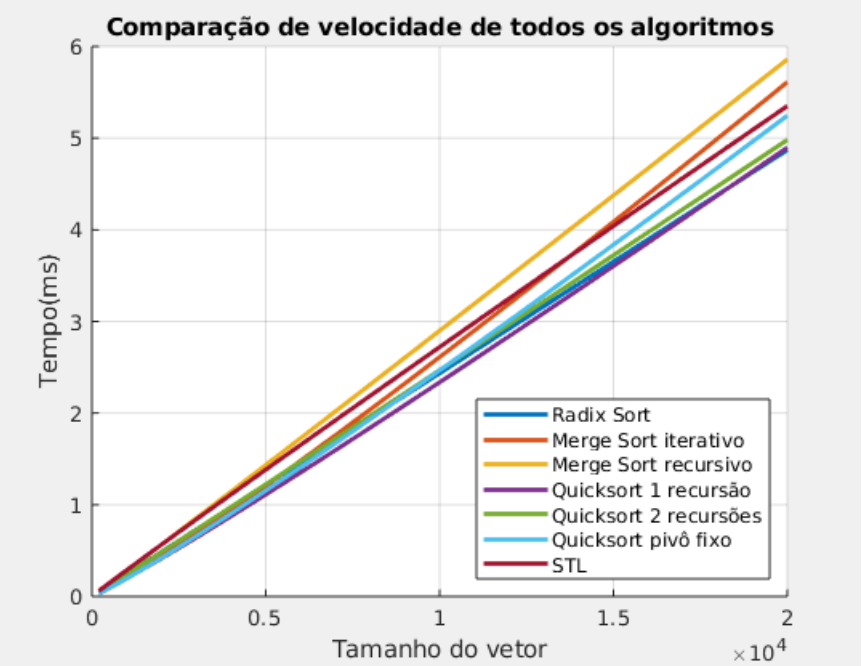

Figure 12: Comparação da velocidade de todos os algoritmos implementados mais o da biblioteca *STL*

6 Conclusões

Conforme a explicação das figuras 3, 7, 8, 10, 11 e 12, pode-se concluir que:

1. Figura 12: Na comparação da melhor versão dos algoritmos *Merge Sort*, *Quicksort* e *Radix Sort*, conclui-se que, quando os elementos do vetor são inteiros e limitados por um número não muito grande, *Radix Sort* apresenta um desempenho superior ao do *Merge Sort* e semelhante ao do *Quicksort*. Mas como nem sempre este é o caso, geralmente deve-se preferir o *Quicksort* aos demais, além de que este não realiza uma alocação de memória extra $O(n)$.

2. Figura 3: A versão iterativa do *Merge Sort* é sutilmente superior à versão recursiva no quesito tempo. Mas como a versão iterativa não faz chamadas

recursivas nem gera uma pilha de recursão de profundidade potencialmente elevada, deve-se sempre preferir a versão iterativa à recursiva.

3. Figuras 7, 8, 10 e 11: O desempenho temporal do *Quicksort* com uma chamada recursiva é semelhante ao com duas chamadas recursivas, no entanto, o número de chamadas recursivas e a profundidade máxima da pilha de recursão do *Quicksort* com uma chamada é consideravelmente menor do que o com duas. Portanto deve-se preferir o *Quicksort* com apenas uma chamada recursiva.

4. Figura 8: A velocidade de ordenação do *Quicksort* com método de pivô fixo é consideravelmente menor que a do *Quicksort* com método de mediana de três. Portanto, deve-se preferir o uso do método mediana de três.

"Pode estar demorando muito por culpa da implementação de quicksort da libc, já que alguns dos nossos dados já vem quase ordenados, fica quadrático". Certamente é possível, mas é plausível ou esperado que uma biblioteca importante, antiga, muito utilizada e bem testada, implemente quicksort de forma a causar o problema citado? Em outras palavras, você, como chefe de um projeto, aprovaria um quicksort implementado assim em um projeto real, considerando o custo-benefício? Justifique.

Não creio que seja plausível que uma biblioteca tão amplamente usada deixe escapar esse defeito, deve haver algum outro motivo subjacente para o problema e, portanto, também por uma questão de custo benefício, aprovaria o Quicksort da libc.

7 Referências

1 T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, MIT Press, 2009