Rapid Sort: Vantagens, Desvantagens e Comparações com outros algoritmos

Carolina Vasques Moreira, Gabriel Félix Ramos, Geovany Carlos Mendes

[1]Instituto de Matemática e Computação – Universidade Federal de Itajubá (UNIFEI)
Caixa Postal 50 – 37500 903 – Itajubá – MG – Brazil

{carol.vasques4, gabriel.felixramos, geovany.c.mendes}@gmail.com

Abstract. *The purpose of this document is to analyze the Rapid Sort algorithm for different input types, compared to Selection Sort, Insertion Sort, Merge Sort and Quick Sort algorithms. A brief description of the algorithms will be given along with information of complexity and stability. The comparison will be done by using three performance metrics: key comparison number, number of register copied and total time spent in sorting.*

Resumo. *Este documento tem como objetivo analisar a eficiência do algoritmo de ordenação Rapid Sort para diferentes tipos de entrada, comparando-o com os algoritmos Selection Sort, Insertion Sort, Merge Sort e Quick Sort. Será dada uma breve descrição dos algoritmos citados, juntamente com informações de complexidade e estabilidade. A comparação será feita por meio de três métricas: número de comparação de chaves, número de cópias de registros realizadas e tempo total gasto para ordenação.*

1. Inrodução

A ordenação consiste na organização de uma coleção de elementos em uma ordem determinada, seja ela crescente ou decrescente. [Singh and Sarmah 2015] Para descobrir qual algoritmo de ordenação é melhor para nosso conjunto de dados e cenário, é necessário comparar esses algoritmos. Para tal, a complexidade desses algoritmos precisa ser considerada. Existem dois tipos de complexidade:

Complexidade de **espaço**: Fornece a quantidade total de memória necessária para executar o algoritmo. Quando falamos em complexidade de espaço, precisamos voltar um pouco no tempo e nos lembrar do quão caro era ter uma memória grande e disponível para seu algoritmo trabalhar. Hoje em dia, a memória se tornou mais barata, portanto mais acessível, deixando a complexidade de espaço "um pouco de lado", tornando a complexidade de tempo a mais importante e também a mais custosa de se otimizar.

Complexidade de **tempo**: Fornece o tempo total necessário para executar o algoritmo. Dessa forma, quanto menor a complexidade, maior a eficiência do algoritmo. Um dos desafios para a comparação de complexidade de tempo se refere à dificuldade em comparar o tempo exato de execução de um algoritmo, pois este depende totalmente do processador e linguagem utilizada, e mesmo que o processador e linguagem sejam iguais, ainda existe a variável de utilização da CPU. Por este motivo, os algoritmos podem ser comparados utilizando a função que determina a taxa de crescimento do algoritmo, expressa em termos do tamanho da entrada n.

Existem várias aplicações de algoritmos de ordenação em todos os campos da ciência da computação, com diferentes complexidades, sendo que o mais eficiente conhecido possui complexidade assintótica *O(nlogn)*. [Singh and Sarmah 2015] A partir disto, iremos estudar a complexidade e eficiência do algoritmo Rapid Sort para diferentes entradas, comparado-o com outros algoritmos existentes.

2. Algoritmos Utilizados

Serão apresentados os algoritmos utilizados neste experimento: Rapid Sort, Selection Sort, Insertion Sort, Merge Sort e Quick Sort. Cada algoritmo terá seu método de funcionamento explicitado, bem como a complexidade de tempo para cada tipo de entrada (desde o melhor caso até o pior caso) e a estabilidade associada a ele, ou seja, se o algoritmo é capaz de "manter a ordem relativa dos itens com chaves iguais." [Ziviani et al. 2004]

2.1. Rapid Sort

O Rapid Sort é um método de ordenação muito similar ao Insertion Sort (subseção 2.3). O algoritmo consiste em selecionar o menor elemento e o maior elemento do vetor e movê-los para a extremidade esquerda e a extremidade direita, respectivamente. Estas posições são demarcadas como lowerbound e upperbound (o equivalente em inglês para "extremidade menor" e "extremidade maior"). Em seguida, os demais elementos do vetor são então posicionados próximos ao lowerbound ou ao upperbound: é realizado um cálculo de distância para determinar se o valor está mais próximo numericamente do mínimo ou do máximo; o elemento é então colocado em uma das extremidades e o marcador referente àquele extremo é atualizado. [Singh and Sarmah 2015]

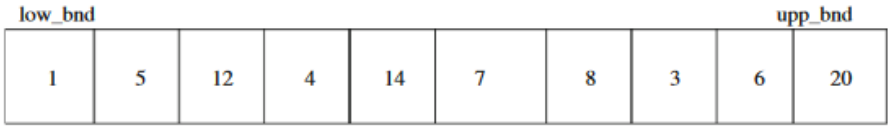

Figura 1. Exemplo de vetor para ordenação pelo Rapid Sort. [Singh and Sarmah 2015]

Por exemplo, para o vetor representado na Figura 1: o vetor é percorrido em ordem a fim de se posicionar os elementos no lowerbound ou no upperbound, começando pelo 5. A distância do 5 ao mínimo, que corresponde ao elemento 1, é igual a 4; já a distância do 5 ao máximo, que é o elemento 20, é igual a 15. Portanto, o elemento está mais próximo do mínimo, e permanece na posição 1 do vetor. Ele é comparado com os demais elementos à esquerda para determinar sua posição correta (se algum elemento à esquerda for maior que o elemento que acaba de chegar, então é preciso trocá-los de lugar para que a ordem crescente seja respeitada), e então o marcador lowerbound passa a ser o novo elemento (lowerbound + 1).

Da mesma forma, o próximo elemento do vetor (12) está mais próximo do máximo. Assim, ele será trocado de lugar com o elemento na posição 8 do vetor, a fim de ser posicionado próximo ao upperbound. Ele é então comparado com os elementos à sua direita para que sua posição correta seja encontrada, e então upperbound será atualizado

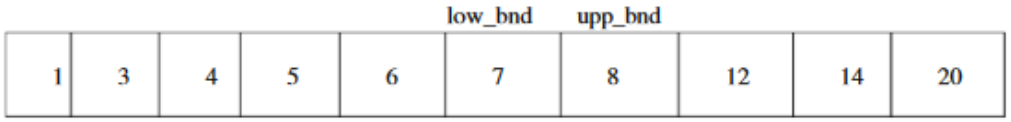

Figura 2. Condição de parada do Rapid Sort. [Singh and Sarmah 2015]

para demarcar o novo elemento (upperbound - 1). O algoritmo continua sua execução até não haver nenhum elemento desordenado ($|lower$bound - upperbound$|$ = 1).

Como analisado, para cada elemento posicionado em uma extremidade, sua posição exata no vetor precisa ser encontrada. Para isto, pode ser utilizada uma busca linear de ordem O(n) ou então uma busca binária de ordem O(logn). Para posicionar o elemento, trocas sucessivas entre elementos vizinhos podem ser necessárias. Para que isto não ocorra, a solução é utilizar listas encadeadas para que seja possível percorrer os elementos e posicioná-los de forma mais eficiente. Desta forma, a complexidade para posicionamento passa a ser O(1). [Singh and Sarmah 2015]

```
6  void rapidSort(int *v,int tam,double *TeC){
7      //variáveis
8      int i,aux,pos=0;
9      //marcadores iniciados no começo e no fim do vetor
10     int lowBound = 0;
11     int upBound = tam-1;
12     //coloca o maior elemento no final
13     for(i=1;i<tam;i++){
14         //incrementa comparações
15         TeC[1]+=1.0;
16         if(v[i]>v[pos])
17         {
18             //posição do maior elemento
19             pos = i;
20         }
21     }
22     aux = v[pos];
23     v[pos] = v[tam-1];
24     v[tam-1] = aux;
25     pos=0;
26     //coloca o menor elemento no começo
27     for(i=1;i<tam;i++){
28         //incrementa comparações
29         TeC[1]+=1.0;
30         if(v[i]<v[pos])
31         {
32             //posição do menor elemento
33             pos = i;
34         }
35     }
```

Figura 3. Implementação em C do Rapid Sort - Parte 1.

Analisando o algoritmo como um todo, sua complexidade tanto para o melhor caso (que consiste em inverter um vetor com elementos com valores próximos) quanto para o pior caso é igual a **O(n)**. O Rapid Sort não é um método estável, uma vez que trocas entre elementos podem reverter a ordem relativa original de elementos com chaves iguais.

```
36      aux = v[pos];
37      v[pos] = v[0];
38      v[0] = aux;
39      while((upBound-lowBound)>1){
40          //verifica se o valor está mais próximo do maior ou do menor
41          if(abs(v[lowBound]-v[lowBound+1])<abs(v[upBound]-v[lowBound+1])){
42              //incrementa comparações
43              TeC[1]+=1.0;
44              if(v[lowBound+1]<v[lowBound]){
45                  //incrementa comparações
46                  TeC[1]+=1.0;
47                  aux = v[lowBound+1];
48                  i = lowBound;
49                  while ((i>=0)&&(aux<v[i])){
50                      v[i+1] = v[i];
51                      i--;
52                      //incrementa trocas
53                      TeC[0]+=1.0;
```

Figura 4. Implementação em C do Rapid Sort - Parte 2.

```
54                  }
55                  v[i+1] = aux;
56              }
57              lowBound++;
58          }
59          else{
60              //troca o lowBound+1 com o upBound-1
61              aux = v[lowBound+1];
62              v[lowBound+1] = v[upBound-1];
63              v[upBound-1] = aux;
64              //incrementa trocas
65              TeC[0]+=1.0;
66              //acha a posição correta do elemento na parte com maiores elementos
67              if(v[upBound-1]>v[lowBound]){
68                  //incrementa comparações
69                  TeC[1]+=1.0;
70                  aux = v[upBound-1];
71                  i = upBound;
```

Figura 5. Implementação em C do Rapid Sort - Parte 3.

```
72                  while ((i<=tam)&&(aux>v[i])){
73                      v[i-1] = v[i];
74                      i++;
75                      //incrementa trocas
76                      TeC[0]+=1.0;
77                  }
78                  v[i-1] = aux;
79                  //incrementa trocas
80                  TeC[0]+=1.0;
81              }
82              upBound--;
83          }
84
85      }
86
87  }
```

Figura 6. Implementação em C do Rapid Sort - Parte 4.

2.2. Selection Sort

O algoritmo de ordenação por seleção (ou Selection Sort) é um dos mais simples que existem. Ele consiste em realizar os seguintes passos: “selecione o menor item do vetor

e a seguir troque-o com o item que está na primeira posição do vetor. Repita estas duas operações com os *n - 1* itens restantes, depois com os *n - 2* itens, até que reste apenas um elemento."[Ziviani et al. 2004]

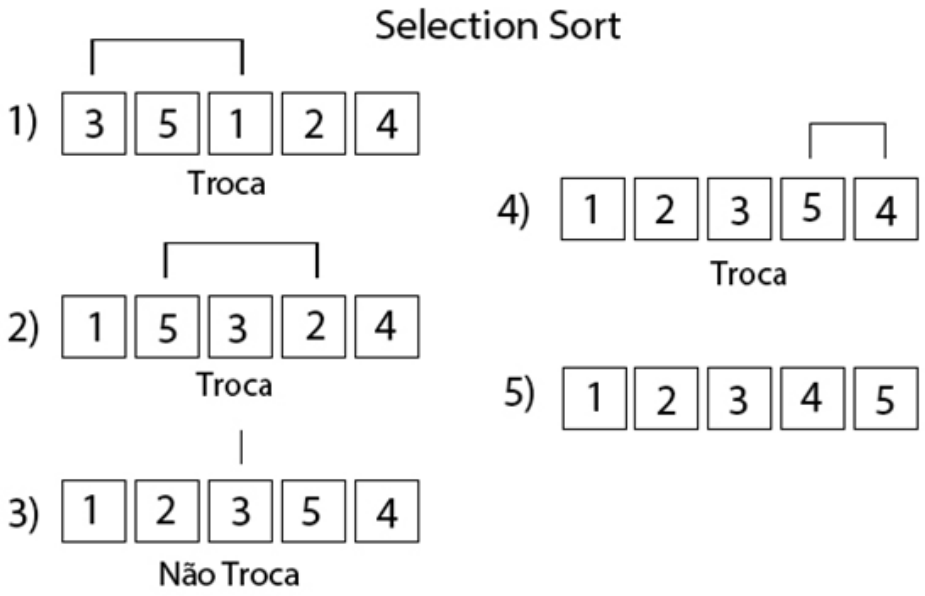

Figura 7. Funcionamento da ordenação por seleção. [de Almeida H. 2013]

A complexidade do Selection Sort é igual a $O(n^2)$ para todos os casos. Segundo Ziviani (2004), esta é uma das principais desvantagens do algoritmo, pois, mesmo para vetores inteiramente ou parcialmente ordenados, o método de seleção ainda apresenta complexidade alta. O Selection Sort não é estável, uma vez que nem sempre mantém a ordem dos elementos com chaves iguais - o que também pode ser considerado uma desvantagem, em contraste com sua simplicidade de implementação.

2.3. Insertion Sort

No método de ordenação por inserção, divide-se o vetor em duas partes, ordenada e desordenada, por meio de um marcador que sinaliza o primeiro elemento da parte desordenada. O item que se encontra no marcador é então colocado em seu lugar apropriado no vetor "movendo-se itens com chaves maiores para a direita e então inserindo o item na posição deixada vazia. Neste processo de alternar comparações e movimentos de registros existem duas condições distintas que podem causar a terminação do processo: (i) um item com chave menor que o item em consideração é encontrado; (ii) o final da seqüência destino é atingido à esquerda." [Ziviani et al. 2004]

O algoritmo de ordenação por inserção tem complexidade igual a *O(n)* no melhor caso e $O(n^2)$ no caso médio e no pior caso, sendo um algoritmo ótimo para casos em que o vetor está quase ordenado. O Insertion Sort é um método estável, sendo capaz de manter elementos com chaves iguais na mesma ordem relativa em que se encontravam ao início da ordenação.

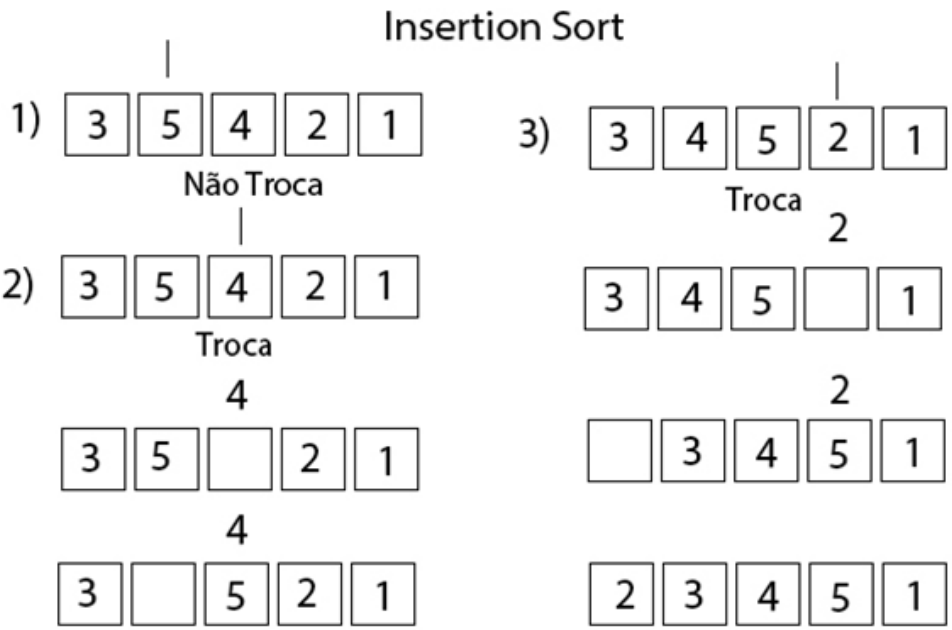

Figura 8. Funcionamento da ordenação por inserção. [de Almeida H. 2013]

2.4. Merge Sort

O Merge Sort é um algoritmo baseado no paradigma de divisão e conquista, o qual "consiste em dividir o problema a ser resolvido em partes menores, encontrar soluções para as partes, e então combinar as soluções obtidas em uma solução global." [Moreira and Viana 2011]

"A operação de divisão é imediata: o vetor é dividido em dois vetores com metade do tamanho do original, que são ordenados recursivamente. O trabalho do algoritmo está concentrado na conquista: a intercalação dos dois subvetores ordenados." [Beder 2018] O Merge Sort utiliza um vetor auxiliar para realizar a intercalação dos elementos, os quais são copiados para o vetor original ao serem ordenados.

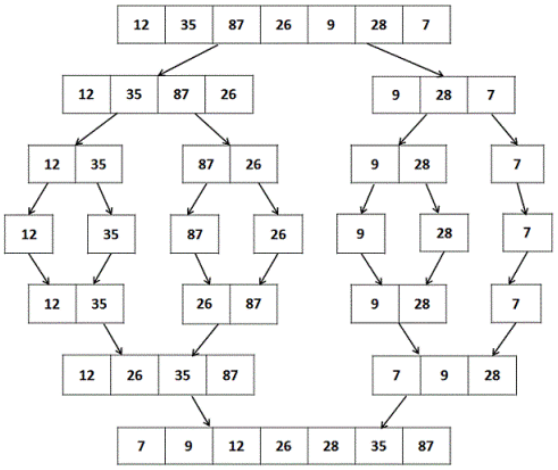

Figura 9. Funcionamento do Merge Sort. [Roberts 2017]

Ao contrário do que as representações do funcionamento do algoritmo possam indicar, o Merge Sort não é paralelo: o método não ordena todas as seções de forma simultânea, pelo contrário; cada um dos "lados" do vetor é ordenado em um momento da execução, de forma sucessiva, de acordo com a ordem da pilha de recursão.

O Merge Sort é estável e sua complexidade é igual a *O(nlogn)* em todos os casos, o que o caracteriza como um dos melhores algoritmos de ordenação. Entretanto, a necessidade de alocar espaço para o vetor auxiliar se configura como uma desvantagem deste método.

2.5. Quick Sort

Assim como o Merge Sort, o Quick Sort também utiliza a técnica de divisão e conquista. "É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações, sendo provavelmente mais utilizado do que qualquer outro algoritmo. O algoritmo foi inventado por C. A. R. Hoare em 1960, quando visitava a Universidade de Moscou como estudante." [Ziviani et al. 2004]

O algoritmo consiste em escolher um elemento como pivô e em seguida particionar o vetor de forma que os elementos menores que o pivô estejam do lado esquerdo e os elementos maiores que o pivô estejam do lado direito. Em seguida, esta ordenação é então realizada novamente, porém, desta vez para os elementos do lado esquerdo e para os elementos do lado direito do vetor, até que restarem vetores de tamanho unitário.

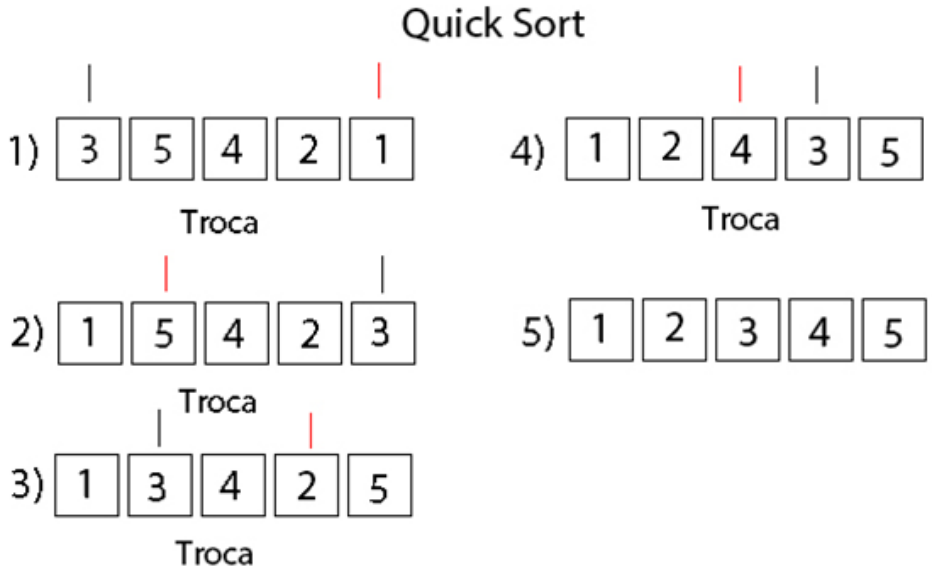

Figura 10. Funcionamento do Quick Sort. [de Almeida H. 2013]

O Quick Sort possui complexidade igual a *O(nlogn)*, exceto em seu pior caso, que é quando a partição realizada não divide o vetor corretamente - por exemplo, quando o pivô escolhido é o menor elemento. Neste caso, a complexidade sobe para $O(n^2)$. Assim como o Selection Sort, o Quick Sort não é estável; apesar disso, diferente-

mente do Merge Sort, o método necessita apenas de uma pequena pilha como memória auxiliar.[Ziviani et al. 2004]

3. Ambiente de Teste

Os testes foram realizados em um computador Lenovo Ideapad 330 com as seguintes configurações: processador Intel Core i5-8250U 1.60GHz, memória de 8GB, disco rígido com 1TB de armazenamento e utilização do sistema operacional Windows 10.

Os algoritmos - juntamente com as funções auxiliares - foram implementados na linguagem C e executados com o auxílio da IDE NetBeans versão 8.2. O código original foi alterado para contabilização das trocas e comparações realizadas. Os arquivos de entrada possuíam elementos ordenados de três formas: crescente, decrescente e dispostos de maneira aleatória. Foram feitos testes com arquivos de cinco tamanhos distintos: 10000, 12500, 15000, 20000 e 30000 elementos.

4. Resultados e Discussão

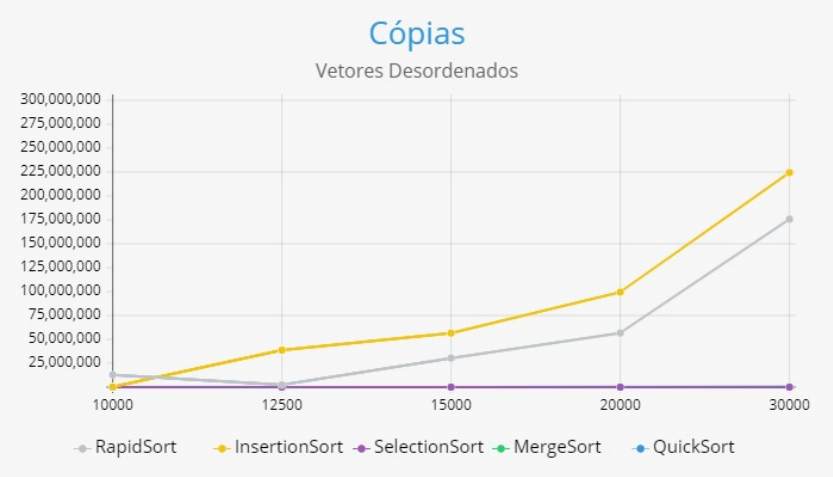

Figura 11. Número de cópias de registro - entrada desordenada.

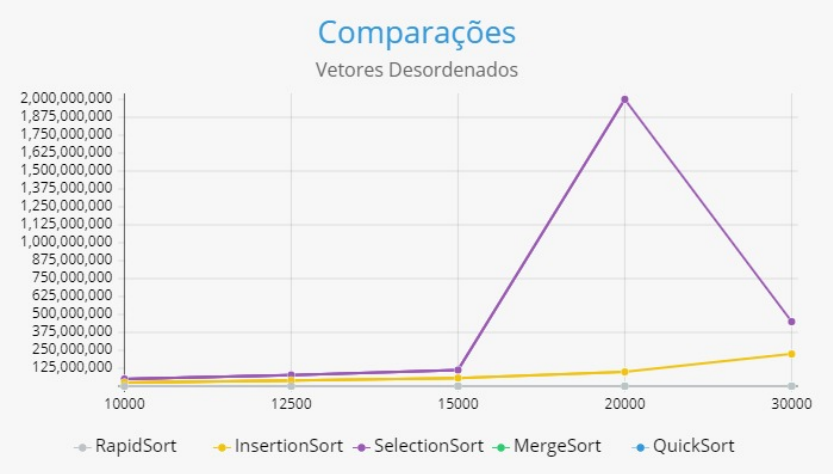

Figura 12. Número de comparações realizadas - entrada desordenada.

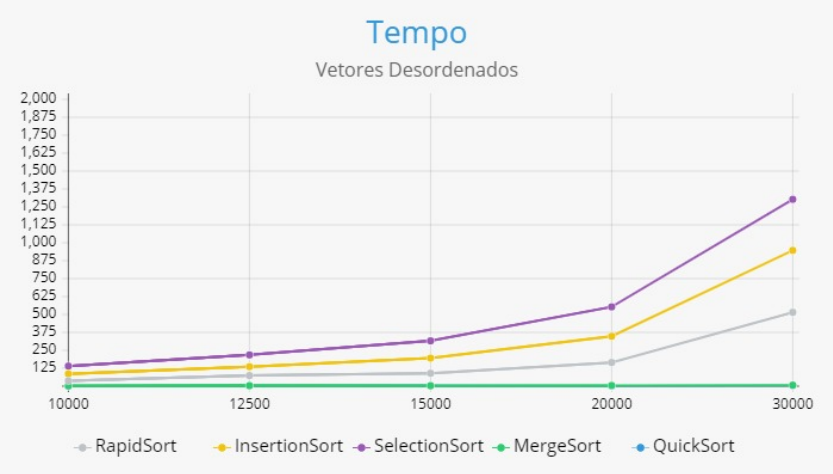

Figura 13. Tempo de execução - entrada desordenada.

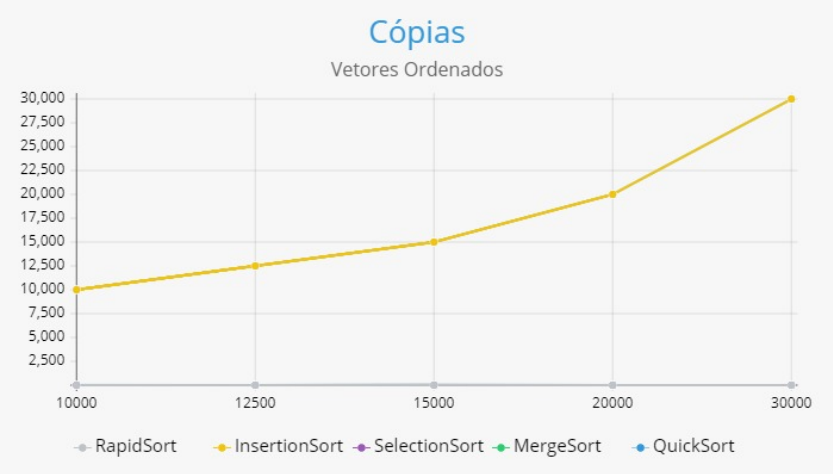

Figura 14. Número de cópias de registro - entrada crescente.

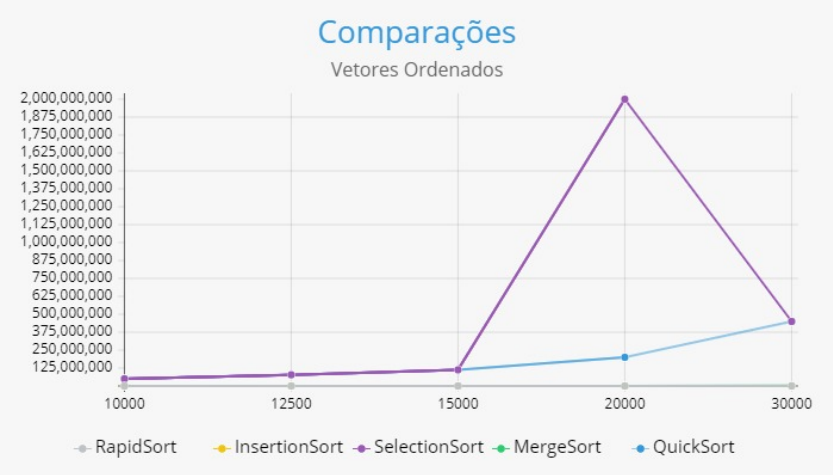

Figura 15. Número de comparações realizadas - entrada crescente.

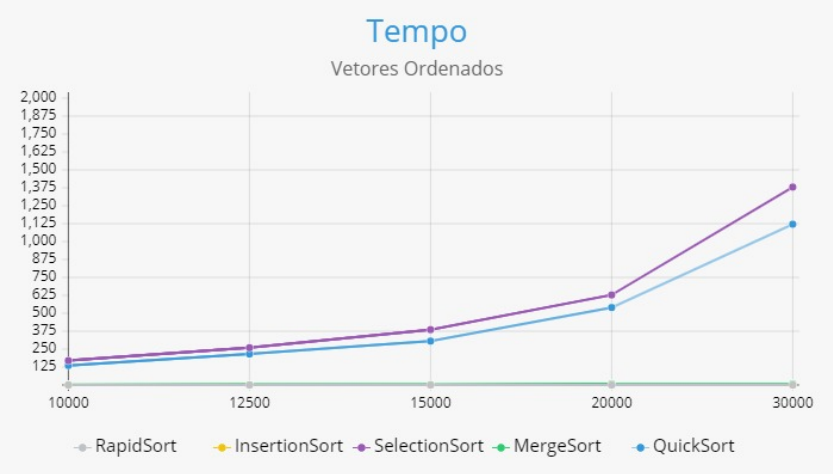

Figura 16. Tempo de execução - entrada crescente.

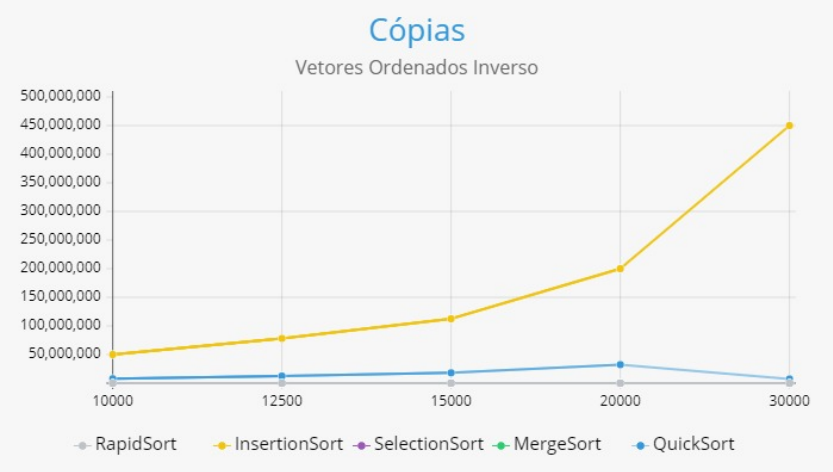

Figura 17. Número de cópias de registro - entrada decrescente.

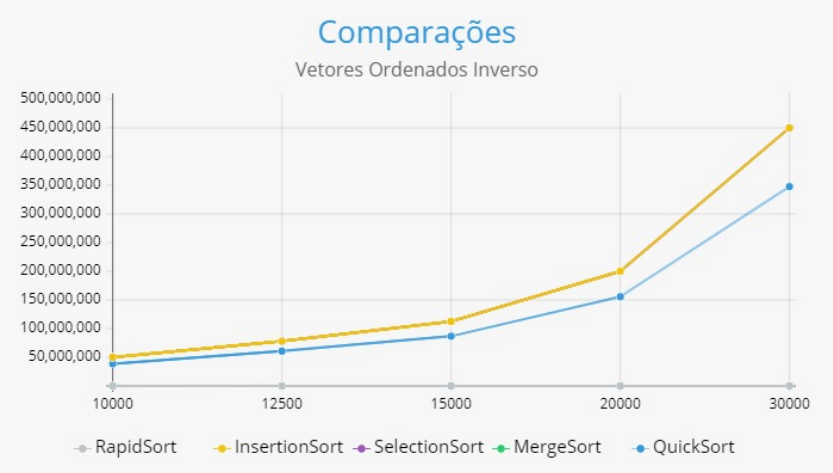

Figura 18. Número de comparações realizadas - entrada decrescente.

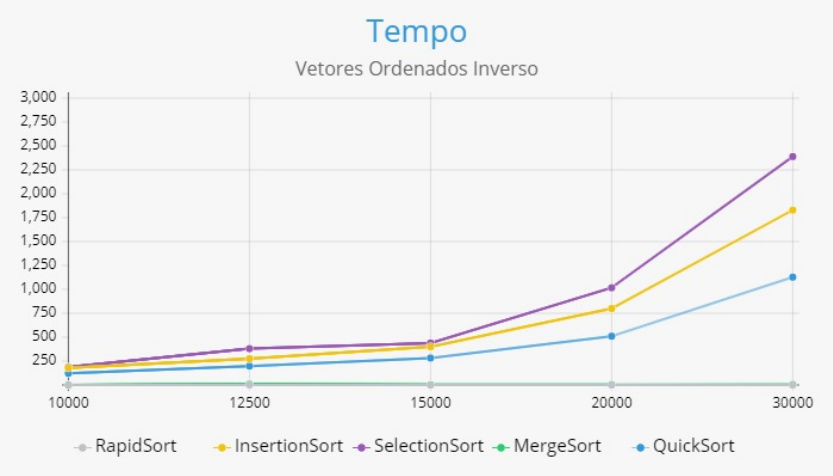

Figura 19. Tempo de execução - entrada decrescente.

RapidSort - Desordenado			
Tamanho	Cópias	Comparações	Tempo de execução
10000	12955470	35720	37,96
12500	2475518	47029	74,03
15000	30392832	54684	90,13
20000	56701153	70435	165,69
30000	175740409	93743	515,1

RapidSort - Ordenado			
Tamanho	Cópias	Comparações	Tempo de execução
10000	13	29996	0
12500	2	37496	0
15000	20	44996	0,5
20000	6	5996	0,5
30000	7	89996	0,53

RapidSort - Ordenado inverso			
Tamanho	Cópias	Comparações	Tempo de execução
10000	9997	29997	0
12500	12502	37496	1,06
15000	15008	44996	0,53
20000	19998	5996	0,53
30000	29996	89996	0

Figura 20. Resultados numéricos do Rapid Sort.

SelectionSort - Desordenado			
Tamanho	Cópias	Comparações	Tempo de execução
10000	9992	49994993	140,59
12500	12490	78118741	218,69
15000	14986	112492487	316,66
20000	19992	1999989993	553,2
30000	29990	449984991	1303,16

InsertionSort - Desordenado			
Tamanho	Cópias	Comparações	Tempo de execução
10000	248050	24795046	86,96
12500	38699955	38687456	136,5
15000	56537248	56522249	196,43
20000	99496155	99476156	348,36
30000	224446776	224416777	947,46

SelectionSort - Ordenado			
Tamanho	Cópias	Comparações	Tempo de execução
10000	0	49985001	170,8
12500	0	78106251	260,43
15000	0	112477501	386,5
20000	0	1999700001	628,73
30000	0	449955001	1380,66

InsertionSort - Ordenado			
Tamanho	Cópias	Comparações	Tempo de execução
10000	9999	0	0
12500	12499	0	0
15000	14999	0	0
20000	19999	0	0
30000	29999	0	0

SelectionSort - Ordenado inverso			
Tamanho	Cópias	Comparações	Tempo de execução
10000	6131	49991132	188,06
12500	7645	78113896	380,19
15000	9197	112486698	439,6
20000	12249	199982250	1017,63
30000	18351	449973352	2389,73

InsertionSort - Ordenado inverso			
Tamanho	Cópias	Comparações	Tempo de execução
10000	49999846	49989847	179,69
12500	78125022	78112523	275,6
15000	112499886	112484887	400,53
20000	200000202	199980203	800,56
30000	450000104	449970105	1829,76

Figura 21. Resultados numéricos do Selection e do Insertion Sort.

QuickSort - Desordenado			
Tamanho	Cópias	Comparações	Tempo de execução
10000	81911	157452	1,53
12500	91225	195453	2,06
15000	112658	236951	4,66
20000	158121	336715	1,56
30000	220240	515793	4,16

MergeSort - Desordenado			
Tamanho	Cópias	Comparações	Tempo de execução
10000	59012	120493	2,63
12500	76215	154584	4,13
15000	93592	189450	3,73
20000	128185	260815	5,2
30000	201884	408584	7,73

QuickSort - Ordenado			
Tamanho	Cópias	Comparações	Tempo de execução
10000	9999	49995000	136,4
12500	12499	78118750	216,19
15000	14999	112492500	306,76
20000	19999	199990000	540,73
30000	29999	449985000	1121,9

MergeSort - Ordenado			
Tamanho	Cópias	Comparações	Tempo de execução
10000	0	69008	1,6
12500	0	87988	2,6
15000	0	106364	3,1
20000	0	148016	7,26
30000	0	2277728	7,29

QuickSort - Ordenado inverso			
Tamanho	Cópias	Comparações	Tempo de execução
10000	7783868	38668477	122,9
12500	12408188	60759774	198,03
15000	18079574	86757126	282,76
20000	32380983	155590893	510,93
30000	7349594	347584773	1128,66

MergeSort - Ordenado inverso			
Tamanho	Cópias	Comparações	Tempo de execução
10000	62749	67256	0
12500	81003	86534	9,9
15000	99213	105733	3,13
20000	135721	144356	4,16
30000	213502	226482	6,83

Figura 22. Resultados numéricos do Quick e do Merge Sort.

Através da análise dos resultados, é possível observar que os algoritmos que utilizam recursão (Merge Sort e Quick Sort) se mostraram mais eficientes. Entretanto, durante a execução do Quick Sort, ocorreu estouro da pilha de chamadas diversas vezes. Em testes iniciais, o Merge Sort se mostrou muito eficiente mesmo para entradas de ordem 10^6; porém, os resultados destas entradas não foram considerados devido à limitação enfrentada pelo Quick Sort (o que resultou na diminuição do tamanho máximo da entrada a ser utilizada nos testes).

5. Conclusão

Após o experimento, foi possível concluir que o Merge Sort e o Quick Sort são os algoritmos mais indicados para ordenar vetores maiores. Os métodos de ordenação mais simples, Selection Sort e Insertion Sort, podem ser utilizados para ordenar vetores pequenos, uma vez que são fáceis de implementar e não há necessidade de investir em algoritmos recursivos (que consomem bastante memória). Já o Rapid Sort se mostrou muito eficiente em todas as métricas avaliadas quando realizamos a ordenação de vetores decrescentes.

Referências

Beder, D. M. (2018). Algoritmos de ordenação: Mergesort.

de Almeida H., B. (2013). *Algoritmos de ordenação: análise e comparação*. DevMedia, https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261.

Moreira, F. V. C. and Viana, G. V. R. (2011). Técnicas de divisão e conquista e de programação dinâmica para a resolução de problemas de otimização. *Revista Científica da Faculdade Lourenço Filho*, 8(1).

Roberts, E. R. (2017). Merge sort. https://medium.com/@ethan.reid.roberts/merge-sort-11732041ff58.

Singh, H. R. and Sarmah, M. (2015). Comparing rapid sort with some existing sorting algorithms. In *Proceedings of Fourth International Conference on Soft Computing for Problem Solving*, pages 609–618. Springer.

Ziviani, N. et al. (2004). *Projeto de algoritmos: com implementações em Pascal e C*, volume 2. Thomson Luton.