

# Análise dos tempos de execução dos algoritmos BubbleSort, InsertionSort, MergeSort, QuickSort, ShellSort

Allan Cordeiro Rocha de Araújo.

<sup>1</sup>Centro de ciência e tecnologia– Universidade Federal de Roraima (UFRR)  
Boa Vista– RR – Brasil

[allanps32008@gmail.com](mailto:allanps32008@gmail.com)

***Resumo.** Será feito uma análise dos algoritmos BubbleSort, MergeSort, InsertionSort, MergeSort, QuickSort e ShellSort, coletados seus tempos de execução e comparado graficamente.*

## 1. Introdução

O objetivo deste artigo é, através de entradas reais, comparar a performance de alguns tipos de algoritmos de ordenação existentes, vê que com diferentes entradas e diferentes tipos de vetor, o comportamento de cada é alterado.

## 2. Definição dos algoritmos

O BubbleSort realiza uma comparação de pares de elementos adjacentes e os troca de lugar se estivessem na ordem errada . Este processo se repete até que mais nenhuma troca seja necessária (elementos já ordenados). A complexidade dele no pior caso é  $O(n^2)$ , no caso médio é  $O(n^2)$  e no pior caso é  $O(n)$ .

O InsertionSort é similar a ordenação de cartas de baralho com as mãos, pega-se uma carta de cada vez e a coloca em seu devido lugar, sempre deixando as cartas da mão em ordem. A complexidade no melhor caso é  $O(n)$ , no caso médio é  $O(n^2)$  e no pior caso é  $O(n^2)$ .

O MergeSort tem com ideia básica: dividir e conquistar, ele divide, recursivamente, o conjunto de dados até que cada subconjunto possua 1 elemento, então ele combina 2 subconjuntos de forma a obter 1 conjunto maior e ordenado, esse processo se repete até que se tenha 1 conjunto ordenado. Sua complexidade no melhor caso é  $O(n \log n)$ , no pior caso é  $O(n \log n)$  e no caso médio é  $O(n \log n)$ .

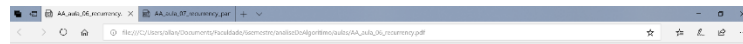
O QuickSort ou Ordenação por troca de partições, a ideia básica é: dividir e conquistar, um elemento é escolhido como pivô arbitrariamente, assim os dados são rearranjados (valores menores colocados antes do pivô e valores maiores colocados depois do pivô). A complexidade dele no pior caso é  $O(n^2)$ , mesmo sendo muito raro de se acontecer, no melhor caso é  $O(n \log n)$  e no caso médio ele não altera a ordem dos elementos.

O ShellSort ele é parecido com a técnica de inserção direta, ele considera o array como vários segmentos a ser ordenado e em cada segmento é aplicado o método da inserção direta, resumindo ele o algoritmo passa várias vezes pela lista dividindo o grupo maior

em menores, nos grupos menores é aplicado o InsertionSort. Sua complexidade no melhor caso é  $O(n \log n)$  e no pior caso é  $O(n^2)$ .

### 3. Cálculo das complexidades

#### BubbleSort



```
BUBBLE-SORT(A, n)
1: for i ← 1 to n do
2:   for j ← i + 1 to n do
3:     if A[j] < A[i] then
4:       troca(A[i], A[j]);
5:     end if
6:   end for
7: end for
```



$$\sum_{i=1}^m \sum_{j=i+1}^m 1 = \sum_{i=1}^m m - (i+1) + 1$$
$$\sum_{i=1}^m m - i = \sum_{i=1}^m m - \sum_{i=1}^m i =$$
$$m(m-1+1) - \frac{m(m+1)}{2} = m^2 - \frac{m^2+m}{2}$$
$$m^2 - \frac{m^2}{2} - \frac{m}{2} = m^2 \left(1 - \frac{1}{2}\right) - \frac{m}{2}$$
$$O(m^2)$$

## InsertionSort

```
FUNÇÃO INSERTION_SORT (A[], tamanho)
    VARIÁVEIS
        i, j, eleito
    PARA I <- 1 ATÉ (tamanho-1) FAÇA
        eleito <- A[i];
        j <- i-1;
        ENQUANTO ((j>=0) E (eleito < A[j])) FAÇA
            A[j+1] := A[j];
# Elemento de lista numerada
            j := j-1;
        FIM_ENQUANTO
        A[j+1] <- eleito;
    FIM_PARA
FIM
```

Handwritten mathematical derivation of the time complexity of Insertion Sort:

$$\sum_{i=2}^{m-1} i-1$$

where  $m = \text{tamanho} - 1$  and  $j+1 = i-1$ .

$$= \sum_{i=2}^m i - \sum_{i=2}^m 1 = \frac{m(m+1)-1(m-2+1)}{2}$$
$$\frac{m^2+m}{2} - 1 - m + 2 - 1 = \frac{m^2+m}{2} - m$$
$$O(m^2)$$

## QuickSort

Algoritmo QuickSort (A, i, f) {

  se (i < f) {

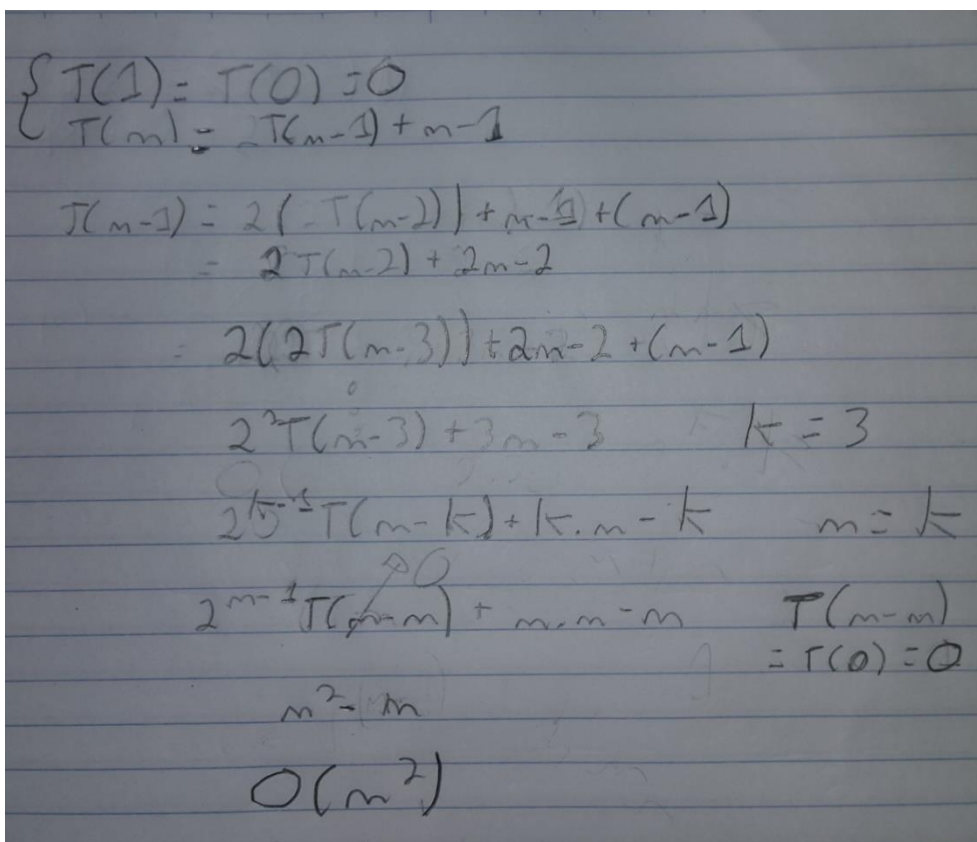
    p := Partição (A, i, f, i);

    QuickSort (A, i, p - 1);

    QuickSort (A, p + 1, f);

  }

}



Handwritten derivation of the time complexity of QuickSort:

$$\begin{aligned} & \begin{cases} T(1) = T(0) = 0 \\ T(n) = 2T(n-1) + n-1 \end{cases} \\ & T(n-1) = 2(T(n-2)) + (n-2) + (n-1) \\ & \quad = 2T(n-2) + 2n-2 \\ & \quad = 2(2T(n-3)) + 2n-2 + (n-1) \\ & \quad = 2^2T(n-3) + 3n-3 \quad k=3 \\ & \quad = 2^{k-1}T(n-k) + k \cdot n - k \quad n=k \\ & \quad = 2^{n-1}T(\overset{0}{n-n}) + n \cdot n - n \quad T(n-n) \\ & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad = T(0) = 0 \\ & \quad n^2 - n \\ & \quad O(n^2) \end{aligned}$$

## ShellSort

```
void shellSort(int *vet, int n) {  
    int i, j, value;  
    int gap = 1;  
    while(gap < n) {  
        gap = 3*gap+1;  
    }  
    while ( gap > 1) {  
        gap /= 3;  
        for(i = gap; i < n; i++) {  
            value = vet[i];  
            j = i;  
            while (j >= gap && value < vet[j - gap]) {  
                vet[j] = vet [j - gap];  
                j = j - gap;  
            }  
            vet [j] = value;  
        }  
    }  
}
```

The image shows a handwritten derivation of the time complexity of ShellSort. It starts with the summation formula for the number of comparisons in each pass, where  $m$  represents the current gap value. The formula is  $\sum_{gap=1}^m j-1 = \sum_{gap=1}^m j - \sum_{gap=1}^m 1$ . This is then simplified to  $\frac{m(m+1)}{2} - (m-1+1)$ . Finally, it is shown that this simplifies to  $\frac{m^2}{2} + \frac{m}{2} - m$ , which is  $O(m^2)$ .

$$\sum_{gap=1}^m j-1 = \sum_{gap=1}^m j - \sum_{gap=1}^m 1$$
$$\frac{m(m+1)}{2} - (m-1+1)$$
$$\frac{m^2}{2} + \frac{m}{2} - m \quad O(m^2)$$

## MergeSort

```
MergeSort (A, i, f) {  
  se (i < f) {  
    m := |(i + f) / 2|;  
    MergeSort (A, i, m);  
    MergeSort (A, m + 1, f);  
    Intercalar (A, i, m, f)  
  }  
}
```

Handwritten mathematical derivation of the MergeSort recurrence relation and its complexity:

$$T(n) = 2T(n/2) + n + 1$$
$$T(1) = 0$$

Pelo método da substituição

$$T(n) = (n-1) + 2 \sum_{i=1}^{n-1} T(i)$$
$$T(1) = 0$$

Portanto pelo método da substituição que

$$T(n) = O(n \log n)$$

#### 4. Análise do tempo de execução

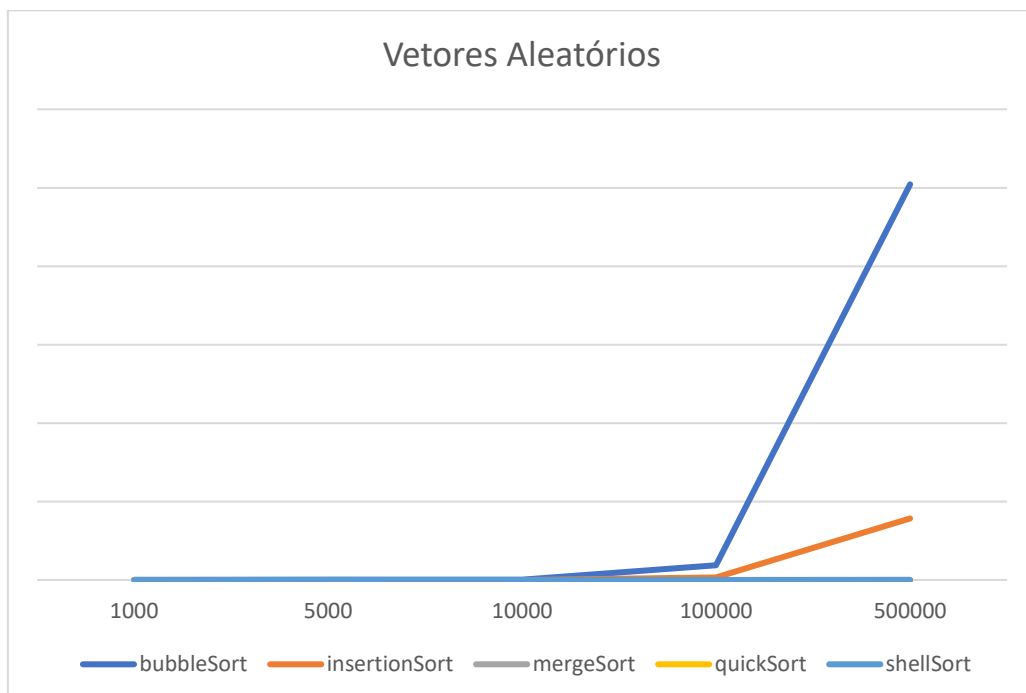
Os testes foram realizados em uma distro do Linux. Intel core i7, geforce GTX 960m, 8gb de RAM, Ubuntu versão 16.04. Todos os testes foram executados 7 vezes e foi retirado a média entre o tempo de execução.

Foi usado a linguagem C++, porém não foi usado qualquer biblioteca que auxiliasse na ordenação ou na criação de vetor, visto isso todos os códigos de ordenação foram implementados na mão. Para o cálculo de tempo foi usado variáveis do tipo `clock_t` e para coleta de tempo a função `clock()`, que usa os ciclos do processador para a obtenção do tempo.

Foi feito uma análise da execução dos algoritmos: BubbleSort, MergeSort, InsertionSort, MergeSort, QuickSort e ShellSort. Foram usados 3 tipos de vetores: ordenado em ordem crescente, ordenado em ordem decrescente e gerado de forma aleatória. Foram usados vários tamanhos de entradas (como será mostrado a seguir). Está plotado em um gráfico feito no Excel tais dados.

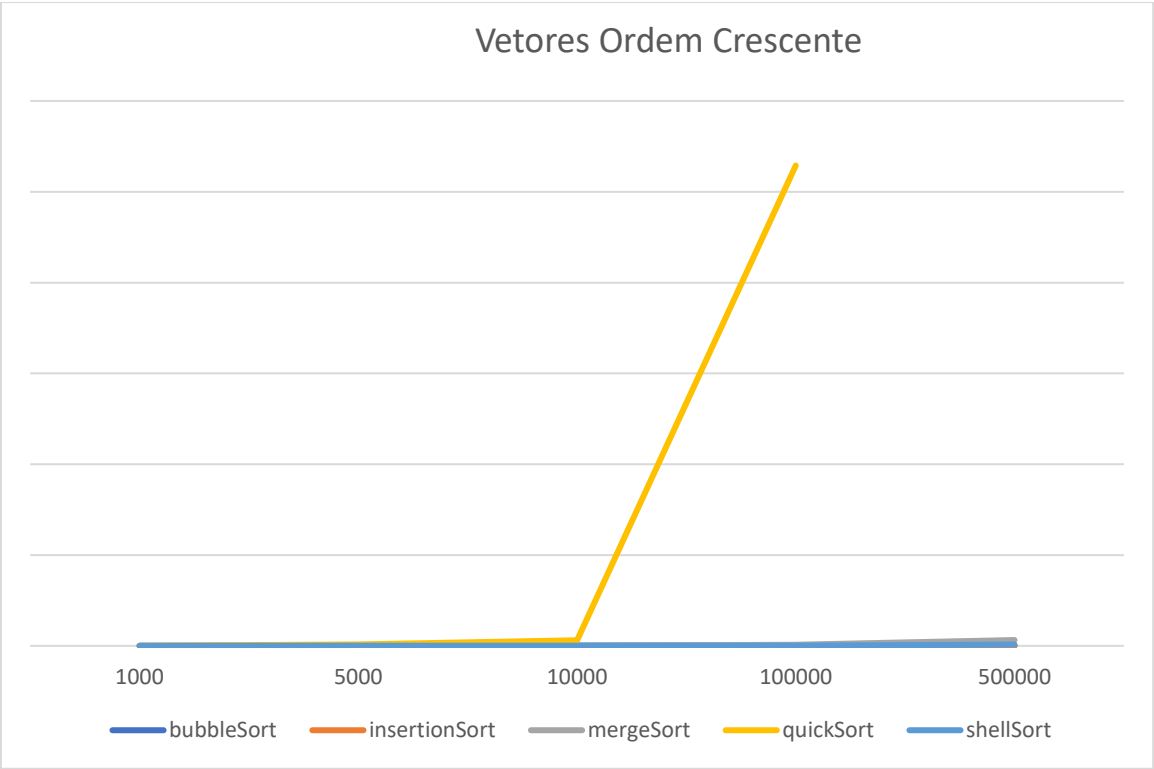
Vetores gerados aleatoriamente.

algoritmos	1000	5000	10000	100000	500000	1000000
bubbleSort	0.002	0,7	0,3	37	1008,74	-----
insertionSort	0.001	0,016	0,08	6,5	156,8	-----
mergeSort	0,0001	0,0015	0,0011	0,0018	0,0102	-----
quickSort	0,001	0,0015	0,002	0,018	0,085	-----
shellSort	0,000012	0,00003	0,015	0,022	0,13	-----



Vetores gerados em ordem crescente

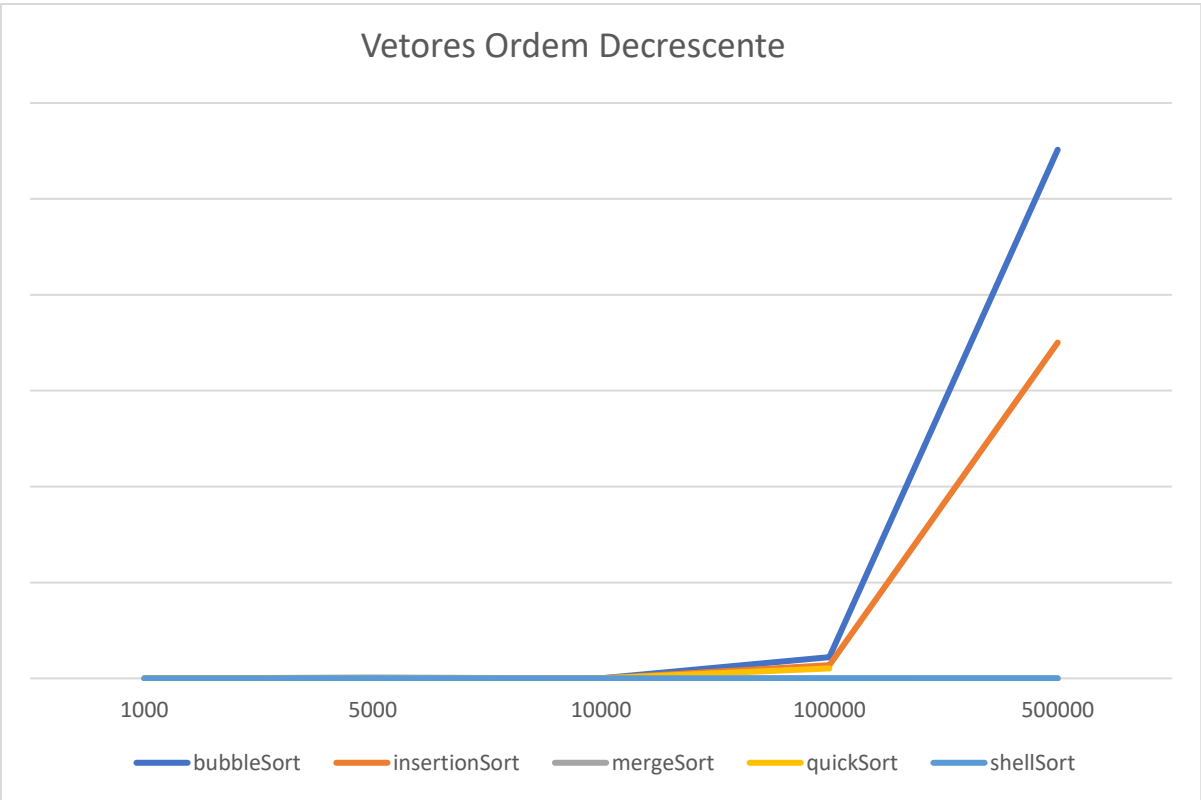
algoritmos	1000	5000	10000	100000	500000	1000000
bubbleSort	0,0000005	0,0000016	0,0000031	0,00375	0,00146	0,002917
insertionSort	0,0000006	0,0000024	0,0000047	0,000447	0,00233	0,00456
mergeSort	0,0002	0,00104	0,0023	0,024	0,127679	0,24634
quickSort	0,00145	0,0322601	0,122372	10,578	-----	-----
shellSort	0,0000028	0,00017	0,000352	0,0044	0,029	0,0581





Vetores gerados em ordem decrescente

algoritmos	1000	5000	10000	100000	500000	1000000
bubbleSort	0,002378	0,66173	0,244918	22,0023	551,228	2161,44
insertionSort	0,001723	0,043113	0,155457	13,8091	350,165	1469,2
mergeSort	0,00022	0,001082	0,00221	0,024	0,125114	0,231683
quickSort	0,001352	0,03255	0,124721	10,4	-----	-----
shellSort	0,000004	0,000245	0,0006	0,0072	0,0436	0,084



## **5. Conclusões**

É bom explicar que nos valores onde existe “----” é porque na execução dos algoritmos para tais valores foi apresentado a saída “Segmentation fault (core dumped)”, a conclusão que se chegou foi o estouro de memória, isso aconteceu principalmente para os algoritmos que usam de recursão e que foi dado uma entrada muito grande, como por exemplo um vetor com 1 milhão de posições.

Tomando algumas conclusões da análise dos dados, percebe-se algumas coisas, como por exemplo o BubbleSort tendo péssimo desempenho para vetores grandes, de uma maneira geral, o único caso em que se saiu bem foi para vetores já ordenados. Algo parecido aconteceu com o InsertionSort, porém este ainda se saiu um pouco melhor para vetores com valores aleatórios. Outro ponto em destaque foi para o QuickSort para vetores em ordem crescente, tendo, disparado, o pior desempenho.

## **Referências**

- "Aula 48 -Ordenação: BubbleSort"; Linguagem C programação descomplicada.  
Disponível em <<https://programacaodescomplicada.wordpress.com/2014/05/03/ed1-aula-48-ordenacao-bubblesort/>>. Acesso em 10 de julho de 2018.
- "Aula 49 -Ordenação: InsertionSort"; Linguagem C programação descomplicada.  
Disponível em <<https://programacaodescomplicada.wordpress.com/2014/05/13/ed1-aula-49-ordenacao-insertionsort/>>. Acesso em 10 de julho de 2018.
- "Aula 51 -Ordenação: MergeSort"; Linguagem C programação descomplicada.  
Disponível em <<https://programacaodescomplicada.wordpress.com/2014/05/19/ed1-aula-51-ordenacao-mergesort/>>. Acesso em 10 de julho de 2018.
- "Aula 52 -Ordenação: QuickSort"; Linguagem C programação descomplicada.  
Disponível em <<https://programacaodescomplicada.wordpress.com/2014/05/23/ed1-aula-52-ordenacao-quicksort/>>. Acesso em 10 de julho de 2018.