**Universidade Federal de Itajubá**

# Relatório de Análise de Algoritmos

**Trabalho submetido ao Prof. João Paulo Leite.**

**Aluno:** Felipe dos Santos
**Matrícula**: 2019002970

**Itajubá**
**03 de Maio de 2021**

## INTRODUÇÃO

Algoritmos são um conjunto de passos para resolver um determinado problema. Uma receita de bolo é o exemplo mais comum que, de fato, representa o conceito de um algoritmo. Os passos de um algoritmo são compostos por operações, essas operações podem ser simples ou complexas como, por exemplo, uma equação de segundo grau comum e uma equação exponencial. Ao desenvolver um algoritmo, existem diversas maneiras de se obter a solução, porém algumas soluções são mais eficientes que as outras. Isso é relativo a diversas variáveis como, por exemplo, um algoritmo pode ser ruim para determinada arquitetura e organização e bom para outra. No caso deste trabalho, o foco dos testes foram os processadores de uso geral e as linguagens de desenvolvimento de alto nível, x86 e C + +. Os algoritmos estudados foram o Bubble Sort, Selection Sort, Insertion Sort, Shell Sort e Merge Sort. Ambos os algoritmos são de ordenação e foram empregados para ordenar palavras, utilizando o tipo de dados String da linguagem C + +.

**OBJETIVO**

O objetivo do relatório foi coletar o tempo de execução dos algoritmos Bubble Sort, Selection Sort, Insertion Sort, Shell Sort e Merge Sort, e realizar a análise dos dados para identificar a eficiência dos algoritmos de acordo com o tamanho da entrada.

## CÓDIGO-FONTE

```
12  void bubbleSort(string *vetor, int tam)
13  {
14      string aux;
15      for(int i = 0; i < tam; i++)
16      {
17          for (int j = 0; j < tam - 1 - i; j++)
18          {
19              if (vetor[j+1] < (vetor[j]))
20              {
21                  aux = vetor[j];
22                  vetor[j] = vetor[j+1];
23                  vetor[j+1] = aux;
24              }
25
26          }
27      }
28  }
```

**Figura 00: Função Bubble Sort.**

```
30  void selectionSort(string *vetor, int tam)
31  {
32      int min;
33      string aux;
34      for (int i = 0; i < (tam-1); i++)
35      {
36          min = i;
37          for (int j = (i+1); j < tam; j++)
38          {
39              if (vetor[j] < vetor[min])
40                  min = j;
41          }
42          aux = vetor[i];
43          vetor[i] = vetor[min];
44          vetor[min] = aux;
45      }
46  }
```

**Figura 01: Função Selection Sort.**

```
48  void insertionSort(string *vetor, int tam)
49  {
50      int j;
51      string aux;
52
53      for(int i = 1; i < tam; i++){
54          aux = vetor[i];
55          for (j = i-1; ((j>=0) && vetor[j] > aux); j--)
56          {
57              vetor[j+1] = vetor[j];
58          }
59          vetor[j+1] = aux;
60      }
61  }
```

**Figura 02: Função Insertion Sort.**

```
63  void print(string *word)
64  {
65      for(int i = 0; i < N; i++)
66          cout << word[i] << endl;
67  }
68
69  void ler_arquivo(string *vet)
70  {
71      ifstream arquivo("aurelio40000.txt");
72      if(arquivo)
73          {
74              for(int i=0; i < N; i++)
75              {
76                  arquivo >> vet[i];
77              }
78              arquivo.close();
79          }
80  }
```

**Figura 03: Função para imprimir e abrir o arquivo .txt.**

```
82  void shellSort(string *vet, int tam)
83  {
84      string aux;
85      int j, h;
86      h = tam/2;
87      while (h >= 1)
88      {
89          for(int i = 1; i < tam; i++){
90              aux = vet[i];
91              for (j = i-h; ((j >=0 ) && vet[j] > aux); j = j - h)
92                  vet[j+h] = vet[j];
93          vet[j+h] = aux;
94          }
95      h = h/2;
96      }
97
98  }
```

**Figura 04: Função Shell Sort.**

```
100  void mergeIntercala(string *vet, string *aux, int ini, int meio, int fim)
101  {
102      int atual, fimEsq, n;
103      atual = ini;
104      fimEsq = meio-1;
105      n = fim - ini + 1;
106      while ((ini <= fimEsq) && (meio <= fim))
107      {
108          if(vet[ini] <= vet[meio])
109          {
110              aux[atual++] = vet[ini++];
111          }
112          else
113              aux[atual++] = vet[meio++];
114      }
115      while (ini <= fimEsq)
116      {
117          aux[atual++] = vet[ini++];
118      }
119      while (meio <= fim)
120      {
121          aux[atual++] = vet[meio++];
122      }
123
124      for(int i = 0; i < n; i++)
125      {
126          vet[fim] = aux[fim];
127          fim--;
128      }
```

**Figura 05: Função utilizada dentro do Merge Sort 1.**

```
132  void mergeDivide(string *vet, string *aux, int ini, int fim)
133  {
134      int meio;
135      if(fim > ini){
136          meio = (ini+fim)/2;
137          mergeDivide(vet, aux, ini, meio);
138          mergeDivide(vet, aux, meio+1, fim);
139          mergeIntercala(vet, aux, ini, meio+1, fim);
140      }
141  }
142
143  void mergeSort(string *vet, int tam)
144  {
145      string aux[tam];
146      mergeDivide(vet, aux, 0, tam-1);
147  }
```

**Figura 06: Função utilizada dentro do Merge Sort 2 e Merge Sort.**

```
149  int main(){
150
151      //string vet[N];
152      clock_t tini, tfim, tms;
153      ler_arquivo(teste);
154      tini = clock();
155      mergeSort(teste, N); //escolher a função desejada
156      tfim = clock();
157      tms = ((tfim - tini)*1000/CLOCKS_PER_SEC);
158      //print(vet);
159      cout << "Tempo total: " << tms << "ms" << "\n";
160
161      return 0;
162  }
```

**Figura 07: Função main com estrutura para coleta de tempo de execução.**

## TESTES E RESULTADOS

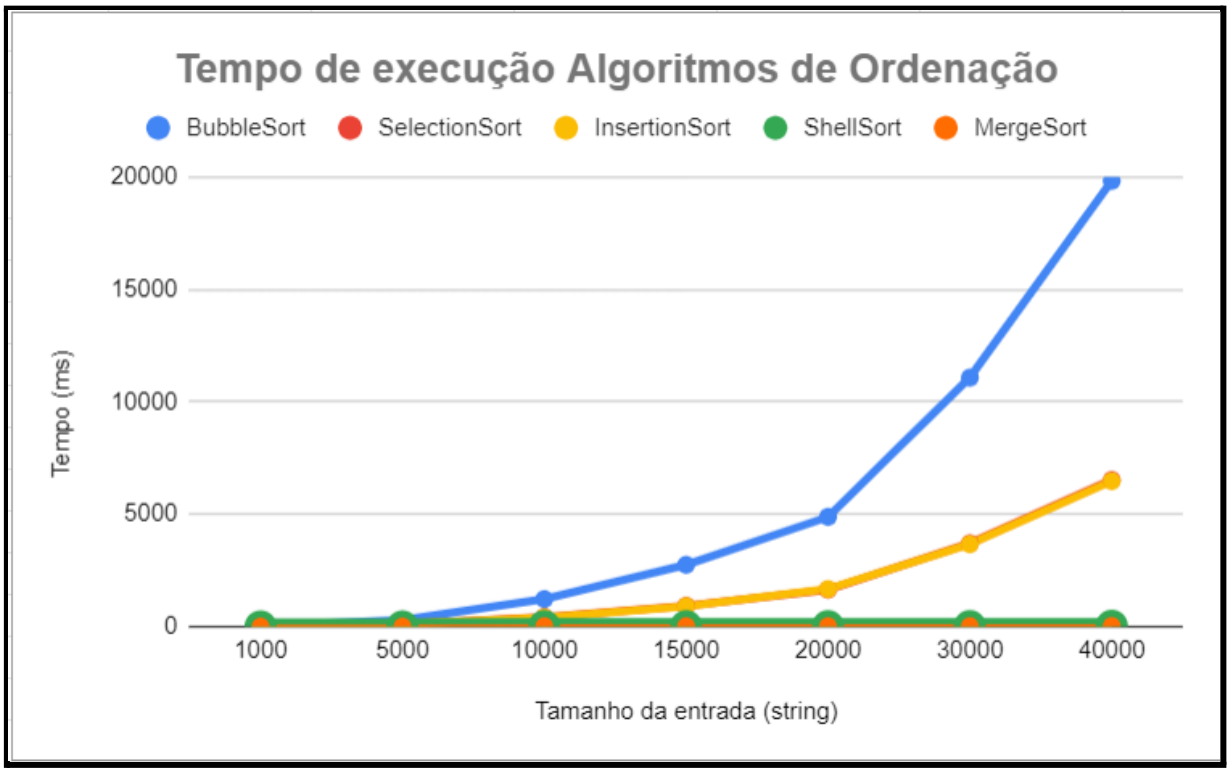


**Figura 08: Gráfico do tempo de execução dos 5 algoritmos.**

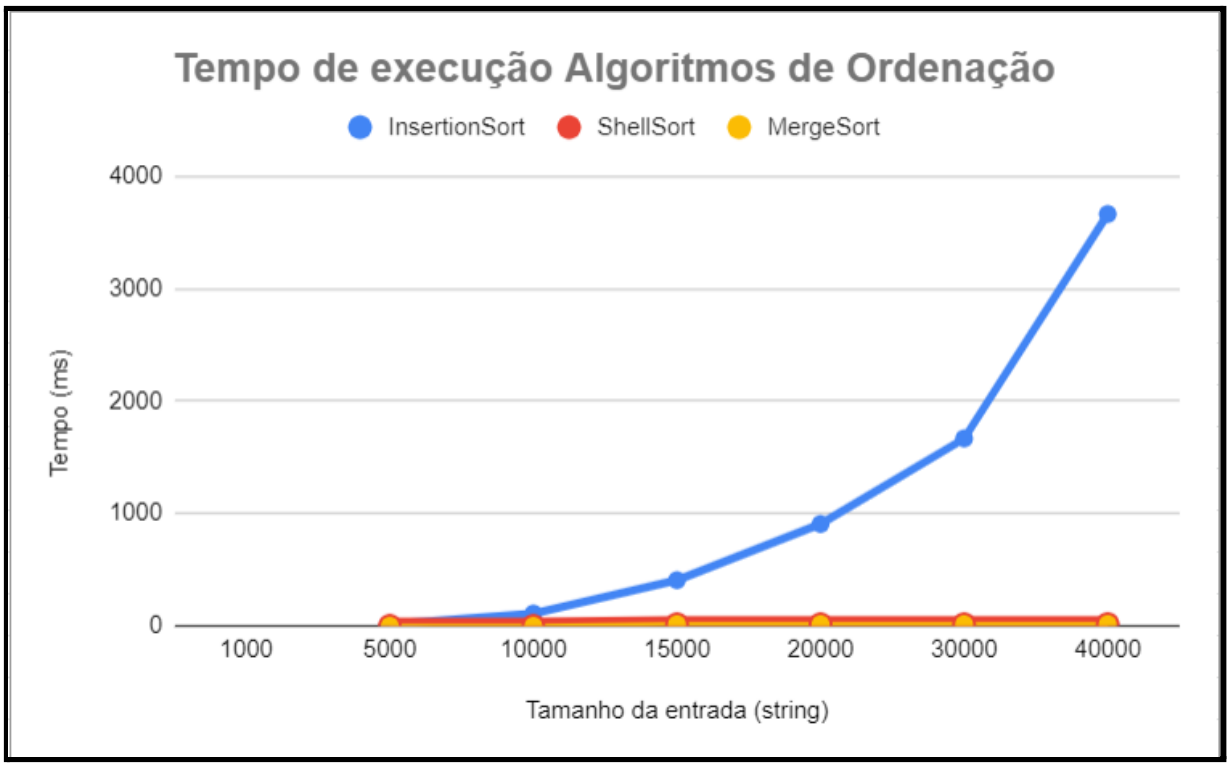


**Figura 09: Gráfico do tempo de execução dos 3 algoritmos.**

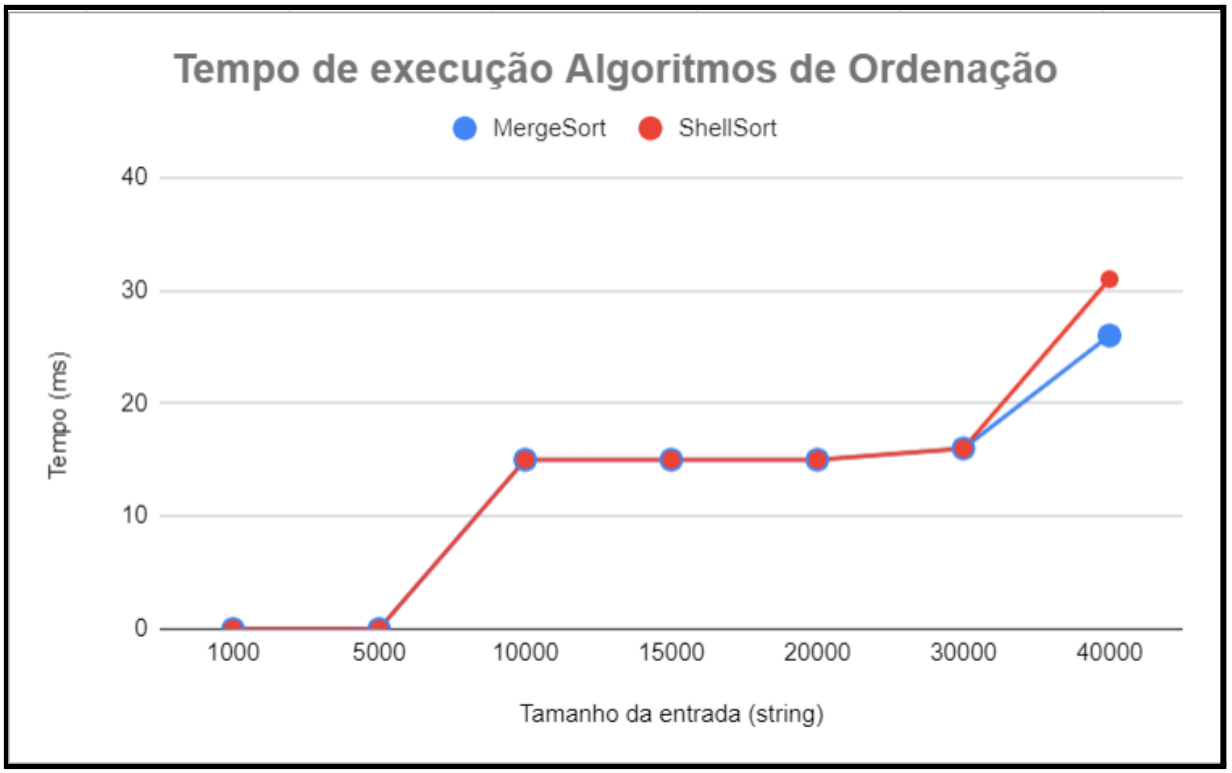


**Figura 10: Gráfico do tempo de execução dos 2 algoritmos eficientes.**

| BubbleSort(ms | Tamanho da entrada | SelectionSort(n | Tamanho da entrada | InsertionSort(m | Tamanho da entrada |
|---|---|---|---|---|---|
| 15 | 1000 | 15 | 1000 | 15 | 1000 |
| 296 | 5000 | 109 | 5000 | 109 | 5000 |
| 1222 | 10000 | 421 | 10000 | 406 | 10000 |
| 2746 | 15000 | 921 | 15000 | 906 | 15000 |
| 4880 | 20000 | 1641 | 20000 | 1667 | 20000 |
| 11072 | 30000 | 3699 | 30000 | 3664 | 30000 |
| 19825 | 40000 | 6518 | 40000 | 6469 | 40000 |

**Figura 11: Tabela com os dados dos 3 algoritmos de ordenação.**

| ShellSort(ms) | Tamanho da entrada | MergeSort(ms) | Tamanho da entrada |
|---|---|---|---|
| 0 | 1000 | 0 | 1000 |
| 0 | 5000 | 0 | 5000 |
| 15 | 10000 | 15 | 10000 |
| 15 | 15000 | 15 | 15000 |
| 15 | 20000 | 15 | 20000 |
| 16 | 30000 | 16 | 30000 |
| 31 | 40000 | 26 | 40000 |

**Figura 12: Tabela com os dados dos 2 algoritmos de ordenação eficientes.**

**DISCUSSÃO**

A realização dos algoritmos foi realizada na linguagem C + + e as bibliotecas utilizadas foram iostream, string, fstream e ctime. Sob a perspectiva da análise do pior caso de execução dos algoritmos, o Bubble Sort, Insertion Sort e Selection Sort são algoritmos O ($n^2$), isto é, possuem a linha de tendência semelhante a um polinômio de segundo grau. Quanto a esses algoritmos, ambos resolvem o problema para entradas limitadas, como a ordenação de arquivos locais em um sistema local. Porém, caso a aplicação fosse direcionada, por exemplo, para um banco de dados de uma empresa multinacional, a entrada de dados seria consideravelmente grande, nesse caso os algoritmos Shell Sort e Merge Sort são mais apropriados. O Shell Sort apresentou resultados consideravelmente mais eficientes com relação aos três primeiros algoritmos citados, porém ainda não é o mais eficiente. Sua linha de tendência se assemelha a um polinômio de grau menor que 2 e maior que 1 e o seu pior caso descreve um algoritmo O($n^{1,x}$), com 1 < x < 2. Já o algoritmo Merge Sort apresentou maior eficiência para entradas maiores que 30.000,00 com relação ao algoritmo Shell Sort. Sua linha de tendência se assemelha a uma curva logarítmica, sendo um algoritmo 0($nlog(n)$).

A partir das tabelas, foram gerados gráficos para o tamanho da entrada em função do tempo. É importante ressaltar que os valores do tempo de execução serão diferentes em outros ambientes, sendo a comparação válida apenas para todos os casos sendo executados sob as mesmas condições.

**CONCLUSÃO**

Ainda que para determinados casos ambos os algoritmos resolvam o problema, a eficiência é um ponto que deve ser levado em consideração em algoritmos. A eficiência de um algoritmo está relacionada à quantidade de processamento que essa solução necessita para resolver o problema. Quanto maior a necessidade de processamento, maior o uso de energia elétrica.

A partir da análise, o algoritmo Merge Sort demonstrou ser a solução com o menor tempo de execução. O Merge Sort é um algoritmo recursivo e iterativo. Para entradas consideravelmente grandes, o Merge Sort é o algoritmo mais recomendável sob a perspectiva de eficiência.