
Universidade Federal
de Ouro Preto

# Trabalho Prático- Avaliação Empírica de Algoritmos de Ordenação

**Alunos em Graduação da Universidade Federal de Ouro Preto do curso Ciência da Computação:**

Halliday Gauss Costa dos Santos

Guilherme Augusto de Deus Maciel

**Área:** Projeto de Análise de Algoritmos.

**Professor:** Anderson Almeida Ferreira

**Resumo**

Este trabalho apresenta a análise empírica e comparação dos algoritmos de ordenação InsertionSort, MergeSort e RadixSort utilizando o teste estatístico t, assim como o funcionamento e análise de complexidade dos mesmos através das funções de complexidade, e gráficos alimentados com o tempo médio de execução desses algoritmos sobre diversas instâncias.

## 1 - Introdução:

O objetivo desse trabalho é fazer uma análise empírica e uma comparação estatística dos métodos de ordenação InsertionSort, MergeSort e RadixSort visando determinar qual desses é o melhor algoritmo para uma instância de tamanho “n”, e qual é o comportamento dos mesmos quando o tamanho da instância aumenta.

Primeiramente será apresentado como é o funcionamento dos algoritmos InsertionSort, MergerSort e RadixSort, a implementação e a função de complexidade de cada um no melhor caso, no caso médio e no pior caso. Em seguida será feito a análise das funções de complexidade e serão estabelecidas as notações assintóticas e uma breve comparação entre os algoritmos.

Segundamente serão geradas 20 instâncias distintas de tamanho ‘n’. Os tamanhos que ‘n’ irá assumir são 100, 1.000,10.000, 100.000 e 1.000.000. Portanto, são geradas 100 instâncias as quais serão testadas nos 3 algoritmos implementados. Após o teste de cada instância os tempos de execução são coletados e armazenados em uma tabela por algoritmo, e após completar as tabelas os gráficos relativos a cada tabela serão construídos utilizando o tempo médio com intervalo de 95% de confiança. Uma outra comparação entre os algoritmos será feita de maneira visual, ou seja, através de uma análise do gráfico que reflete o intervalo dos tempos de execução coletados.

Por fim, é feita uma análise estatística através do teste de estatística t pareado com 95% de confiança visando encontrar evidências estatísticas de qual será o melhor método para cada conjunto de 20 instâncias de tamanho ’n’, ou seja, será concluído se um algoritmo é significativamente melhor que o outro para instâncias de tamanho ‘n’.

## 2- Descrição dos Métodos e Análise de Complexidade

### 2.1 – InsertionSort

O InsertionSort é um algoritmo de ordenação encima de um vetor que se baseia em pegar um elemento do vetor por vez e trocar com os anteriores desde que não seja menor que eles.

Primeiramente o algoritmo considera que o primeiro elemento do vetor esteja ordenado, e a partir do segundo elemento é conferido se  o mesmo é menor que o elemento anterior, se for menor ele troca de lugar com esse elemento e repete a troca de elementos com o próximo elemento anterior desde que seja menor ou que já não esteja no início do vetor, caso contrário o elemento a ser ordenado fica nessa posição mesmo e o próximo elemento do vetor(nesse caso o terceiro elemento) passará pelo mesmo processo. Isso se repete até chegar no último elemento do vetor e o posicionar corretamente. A figura abaixo apresenta esse algoritmo na linguagem Python:

```
1  def insertion_sort(lista):
2      for i in range(1, len(lista)):
3          chave = lista[i]
4          k = i
5          while k > 0 and chave < lista[k - 1]:
6              lista[k] = lista[k - 1]
7              k -= 1
8          lista[k] = chave
```

Figura 1. Algoritmo InsertionSort em Python

Percebe-se que se o vetor já estiver ordenado o código dentro do ‘while’ nunca será executado, pois o elemento a ser ordenado, na figura acima chamado de ‘chave’, nunca será menor que o anterior.

Também é possível observar que na linha 2 têm-se um ‘for’ que repete n-1 vezes, e na linha 3, 4 e 8 totalizam 3 instruções de atribuição dentro do escopo do ‘for’. Na linha 6 e 7 têm-se duas instruções de atribuição dentro do escopo do ‘while’ que dependem da posição dos elementos do vetor para ser executadas, considerando que essas duas instruções são executada ‘ti’ vezes, e sabendo que

existem três instruções de comparação no ‘while’ e que o ‘while’ repete sempre uma vez a mais que as instruções dentro de seu escopo, então a linha 5 é executada dentro de uma interação do ‘for’, ‘ti‘+1 vezes.

| Linha | Quantidade de Instruções | Quantas vezes é executada |
|---|---|---|
| 2 | 1 | n-1 |
| 3 | 1 | n-1 |
| 4 | 1 | n-1 |
| 5 | 3 | ti+1 para cada i de 1 a n-1 |
| 6 | 1 | ti para cada i de 1 a n-1 |
| 7 | 1 | ti para cada i de 1 a n-1 |
| 8 | 1 | n-1 |

Figura 2. Quantidade de instruções que são executadas pelo InsertionSort

Fazendo o somatório dessas instruções é obtido:

$$T(n) = 4 \cdot (n-1) + \sum_{i=1}^{n-1} 3 \cdot (ti + 1) + 2 \cdot ti$$

$$T(n) = 7 \cdot n - 7 + 5\sum_{i=1}^{n-1} ti$$

Onde T(n) é a função que calcula a quantidade de instruções que serão realizadas para ordenar um vetor de tamanho ‘n’ utilizando o InsertionSort.

No melhor caso o vetor está ordenado e o escopo do ‘while’ não é executado nenhuma vez, ou seja, ‘ti’ = 0 então T(n) = 7*n -7. Portanto, T(n) é Θ(n) no melhor caso, pois,

1*n <= 7*n -7<= 8*n, para n > 10.

Para calcular o caso médio deve-se calcula o ‘ti’ médio, sabendo que ‘ti’ pode ser executado 0, 1, 2 até i-1 vezes. Fazendo o somatório de (0+1+2+...+i-1) que é igual a (i-1) * i/2 dividindo pela quantidade de termos que é ‘i’ é obtida a média (i-1)/2. Portanto:

$$T(n) = 7 \cdot n - 7 + \frac{5}{4} n \cdot (n-1)$$

$$T(n) = 7 \cdot n - 7 + 5\sum_{i=1}^{n-1} \frac{(i-1)}{2}$$

$$T(n) = \frac{5n^2 + 23n - 28}{4}$$

Sendo assim T(n) é Θ(n^2) no caso médio, pois 1*n^2 <= T(n) <= 28*n^2, para n >5.

O pior caso acontece quando é necessário trocar um elemento com todos os anteriores, ou seja, ‘ti’ = i-1. Então:

$$T(n) = 7n - 7 + 5\sum_{i=1}^{n-1} i - 1$$

$$T(n) = 7n - 7 + 5 \cdot \left( \frac{n(n-1)}{2} - n + 1 \right)$$

$$T(n) = \frac{5n^2 - n - 4}{2}$$

No pior caso T(n) é Θ(n^2) pois 1*n^2 <= T(n) <= 7n^2, para n > 10.

Observando as funções de complexidade do melhor, pior e caso médio é possível perceber que o InsertionSort executa menos operações quando o vetor está ordenado, e no caso médio são executadas menos operações que o pior caso.

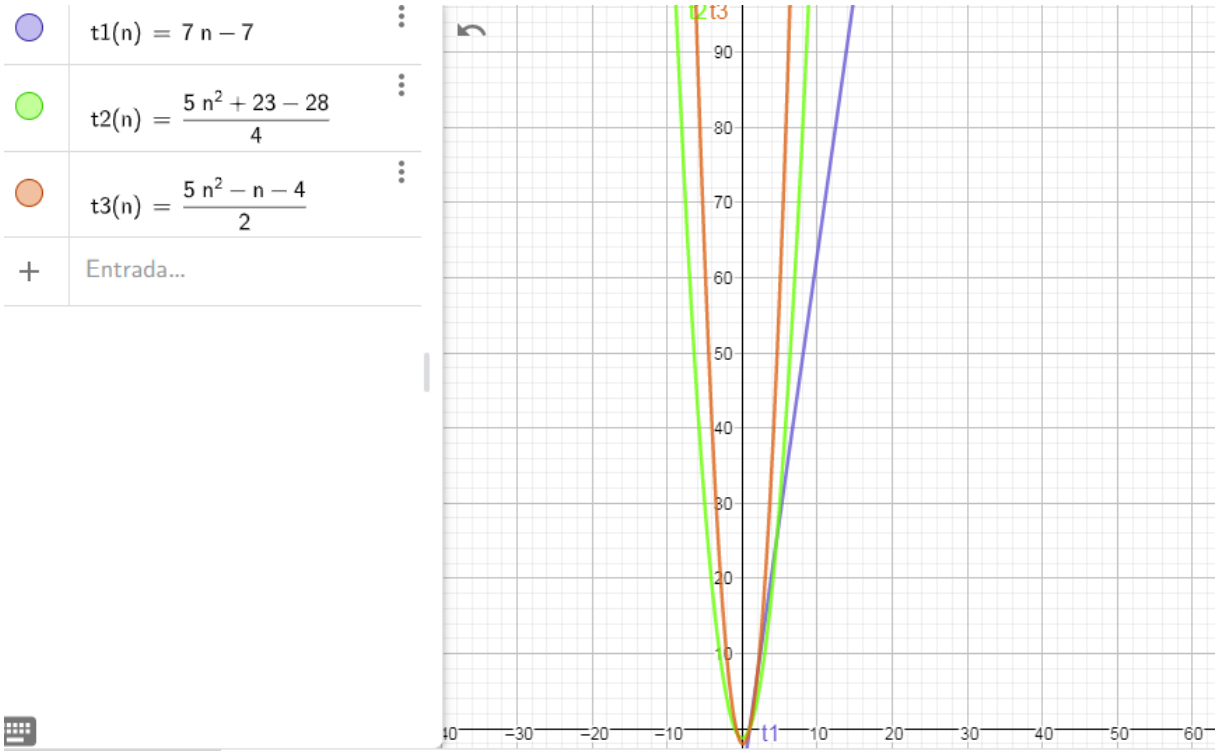



Figura 3. Comparação entre o melhor, pior e caso médio do InsertionSort.

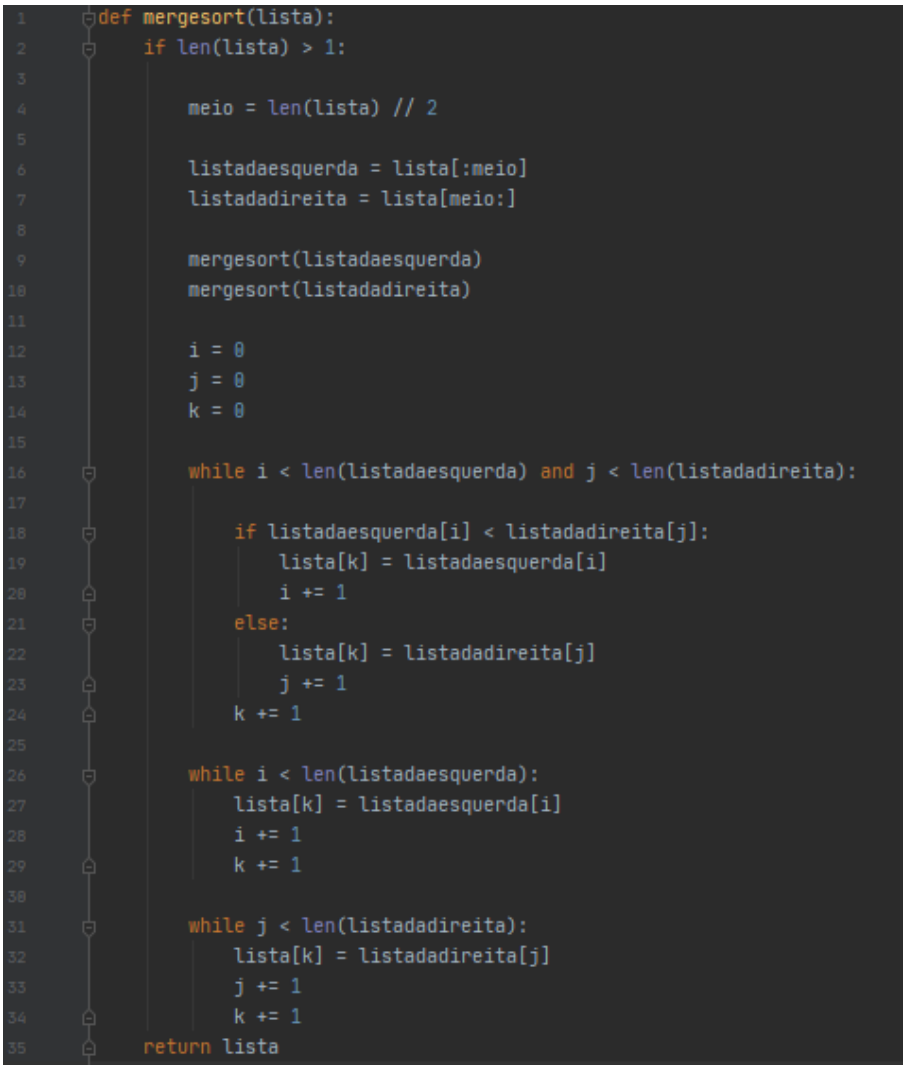
## 2.2 – MergeSort

O MergeSort é um de algoritmo de ordenação por comparação do tipo dividir-para-conquistar. Sua ideia consiste em Dividir (divide o problema em vários subproblemas e resolve esses subproblemas através da recursividade) e Conquistar (após todos os subproblemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos subproblemas).

Os passos do Merge são:

- Dividir: Calcula o ponto médio do vetor. Custo igual a Θ(1). E faz duas chamadas recursivas uma para cada parte do vetor.
- Essa divisão é feita novamente recursivamente para cada chamada recursiva até que sobre um único elemento que por ser único já estar ordenado.
- Conquistar ou Combinar: Recursivamente resolve dois subproblemas, cada um de tamanho n/2. O algoritmo após dividir todo o vetor vai subir recursivamente intercalando os subvetores, ou seja, faz uma combinação entre subvetores que tem custo Θ(n), já que a ordenação desses subvetores é feita comparando os elementos de um subvetor e inserindo os elementos de maneira ordenada em um vetor que terá um tamanho maior. A comparação começa no começo de cada vetor e o menor elemento é inserido no vetor auxiliar e o ponteiro do vetor que teve seu elemento retirado avança em uma posição. Esse procedimento é feito até que tenha um único vetor maior ordenado.

O código em Python abaixo apresenta esse algoritmo:



```
1   def mergesort(lista):
2       if len(lista) > 1:
3
4           meio = len(lista) // 2
5
6           listadaesquerda = lista[:meio]
7           listadadireita = lista[meio:]
8
9           mergesort(listadaesquerda)
10          mergesort(listadadireita)
11
12          i = 0
13          j = 0
14          k = 0
15
16          while i < len(listadaesquerda) and j < len(listadadireita):
17
18              if listadaesquerda[i] < listadadireita[j]:
19                  lista[k] = listadaesquerda[i]
20                  i += 1
21              else:
22                  lista[k] = listadadireita[j]
23                  j += 1
24              k += 1
25
26          while i < len(listadaesquerda):
27              lista[k] = listadaesquerda[i]
28              i += 1
29              k += 1
30
31          while j < len(listadadireita):
32              lista[k] = listadadireita[j]
33              j += 1
34              k += 1
35      return lista
```

Figura 4. Algoritmo MergeSort em Python

Esse código faz exatamente o que foi supracitado, inicialmente, são feitas duas chamadas recursivas, em seguida, serão feitas mais duas chamadas recursiva por recursão até que divida o vetor em 1 elemento (linhas de 2 a 10), e em seguida é feita a intercalação de dois subvetores para cada par de chamada recursiva que tem custo ‘n’ pois percorre ambos subvetores para intercalar (linhas de 16 a 35). Considerando as linhas de 2 a 8 como Θ(1), o custo desse algoritmo a função de complexidade desse algoritmo é:

- Θ(1), se n = 1, ou seja, 1 elemento só tem custo 1 do ‘if’ da linha 2.

- 2*(T/2) + Θ(n), se n>1, pois para cada par de chamada recursiva vai ter uma intercalação de tamanho ‘n’, e o valor de ‘n’ depende da posição na recursão.

Resolvendo a equação de recorrência:

$$T(1) = \theta(1)$$
$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$$

$$T\left(\frac{n}{2}\right) = 2 \cdot T\left(\frac{n}{2^2}\right) + c \cdot \frac{n}{2}$$

$$T\left(\frac{n}{2^2}\right) = 2 \cdot T\left(\frac{n}{2^3}\right) + c \cdot \frac{n}{2^2}$$

$$T(n) = 2 \cdot \left(2 \cdot T\left(\frac{n}{2^2}\right) + c \cdot \frac{n}{2}\right) + c \cdot n$$

---

$$T(n) = 2^2 \cdot T\left(\frac{n}{2^2}\right) + 2 \cdot c \cdot n$$

$$T(n) = 2^2 \cdot \left(2 \cdot T\left(\frac{n}{2^3}\right) + c \cdot \frac{n}{2^2}\right) + 2 \cdot c \cdot n$$

$$T(n) = 2^3 \cdot T\left(\frac{n}{2^3}\right) + 3 \cdot c \cdot n$$

---

$$T(n) = 2^k \cdot T\left(\frac{n}{2^k}\right) + k \cdot c \cdot n$$

Mas como essa divisão de n/2^k acontece até que n/2^k seja 1, ou seja, não tenha mais elementos no vetor para dividir, então k tende a lg(n).

$$T(n) = 2^{lg(n)} \cdot T\left(\frac{n}{2^{lg(n)}}\right) + lg(n) \cdot c \cdot n$$

$$T(n) = n \cdot \theta(1) + c \cdot lg(n) \cdot n$$

$$T(n) = \theta(n) + c \cdot lg(n) \cdot n$$

---

$$T(n) = \theta(n \cdot lg(n))$$

Portanto, o algoritmo MergeSort tem complexidade Θ(n*lg(n)). Cabe ressaltar que independentemente da posição dos elementos dos vetores o MergeSort vai executar os mesmos passos de recursão e intercalação, portanto esse algoritmo tem complexidade Θ(n*lg(n)) no melhor caso, pior caso e no caso médio.

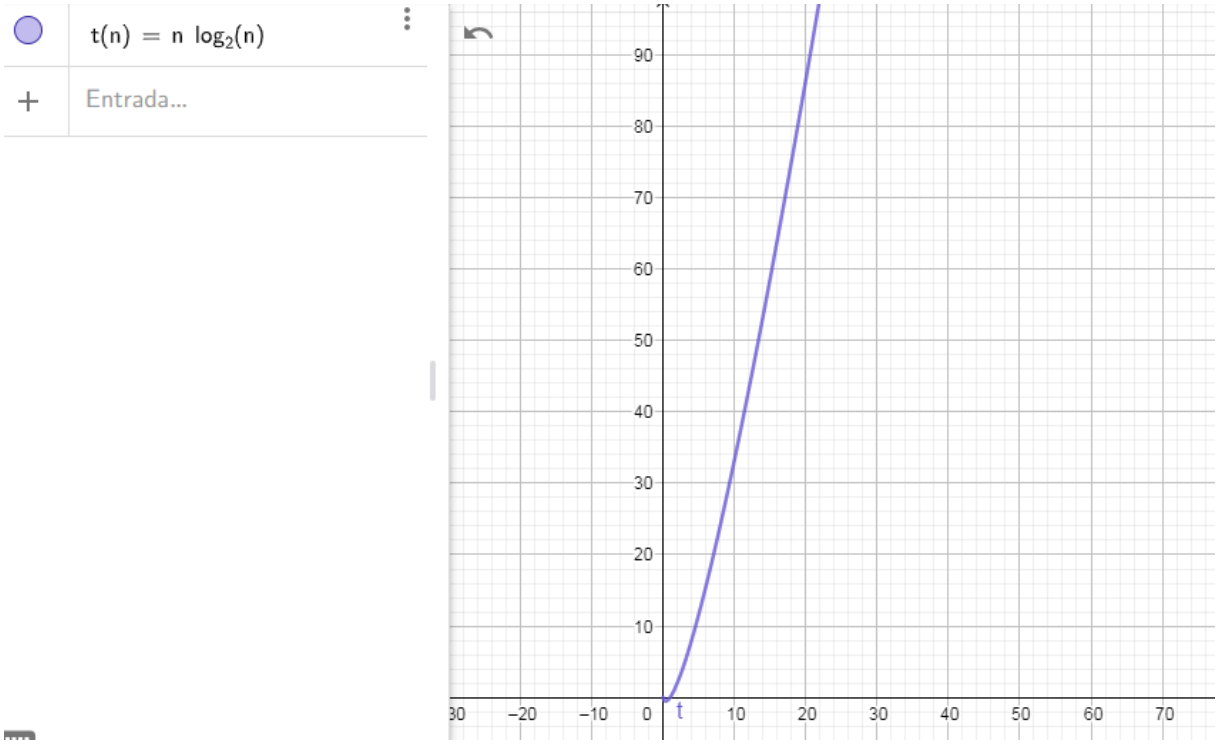



Figura 5. Função de Complexidade do MergeSort.

### 2.3 – RadixSort

O RadixSort é um algoritmo de ordenação que utiliza uma estratégia de ordenação por dígitos ou caracteres de uma palavra ou um número. Em ordenação de números, por exemplo, é pego o último caractere de cada número, ou seja, o dígito menos significativo, em seguida é usado o CoutingSort para ordenar os números, portanto, é criado um vetor com 10 posições e cada número ficará na posição relativa ao seu último dígito, o número 10 por exemplo fica na posição 0, o

número 15 na posição 5, e o número 9 na posição 9, feito isso os números são recolhidos na ordem que ficaram e tudo isso é feito novamente mas com base no próximo caractere, ou seja, o número 10 fica na posição 1 junto com o número 15, e o número 9 como não tem um segundo dígito fica na posição 0. Isso é feito até que tenha analisado todos os caracteres dos números. Cabe ressaltar que o procedimento para ordenar palavras é o mesmo, porém o vetor auxiliar utilizado para fazer a contagem deve ter o tamanho do alfabeto. A figura abaixo apresenta esse algoritmo na linguagem Python:

```python
MAX_CHARS = 28


def radix_sort(lista):
    tamanho_maximo = max([len(palavra) for palavra in lista])

    for pos in range(tamanho_maximo * 1, 1, -1):
        baldes = [list() for y in range(MAX_CHARS)]
        for palavra in lista:
            balde = numero_do_balde(palavra, pos)
            baldes[balde] += [palavra]
        lista = sum(baldes, [])

    return lista


def numero_do_balde(palavra, pos):
    if pos >= len(palavra): return 0
    ch = palavra[pos]
    if ch >= 'A' and ch <= 'Z': return ord(ch) - ord('A') + 1
    if ch >= 'a' and ch <= 'z': return ord(ch) - ord('a') + 1
    return MAX_CHARS - 1
```

Figura 6. Algoritmo RadixSort em Python

É possível perceber que existem dois loops, um ‘for’ mais externo que é repetido de acordo o tamanho da maior palavra ou número a ser ordenado, e um ‘for’ mais interno que é repetido pela quantidade de palavras a serem ordenadas.

Considerando ‘k’ como o tamanho da maior palavra e ‘n’ como a quantidade de palavras, o algoritmo tem complexidade k*n. E é fácil notar que o procedimento de ordenação é o mesmo independente da forma em que os dados a serem ordenados estão no vetor. Portanto a complexidade desse algoritmo é Θ(n*k) no melhor, pior e

caso médio. No entanto o melhor caso, dado uma entrada, seria quando k =1, ou seja, é passado um vetor a ser ordenado com palavras de tamanho 1, e a complexidade nesse caso seria Θ(n), e se o tamanho da maior palavra em um vetor for do tamanho da quantidade de palavras a serem ordenadas esse algoritmo terá a complexidade Θ(n^2), mas isso não interfere muito no resultado, já que esse cenário quase não é visto. O pior caso também pode acontecer quando se tem ordenação de pouco valores.

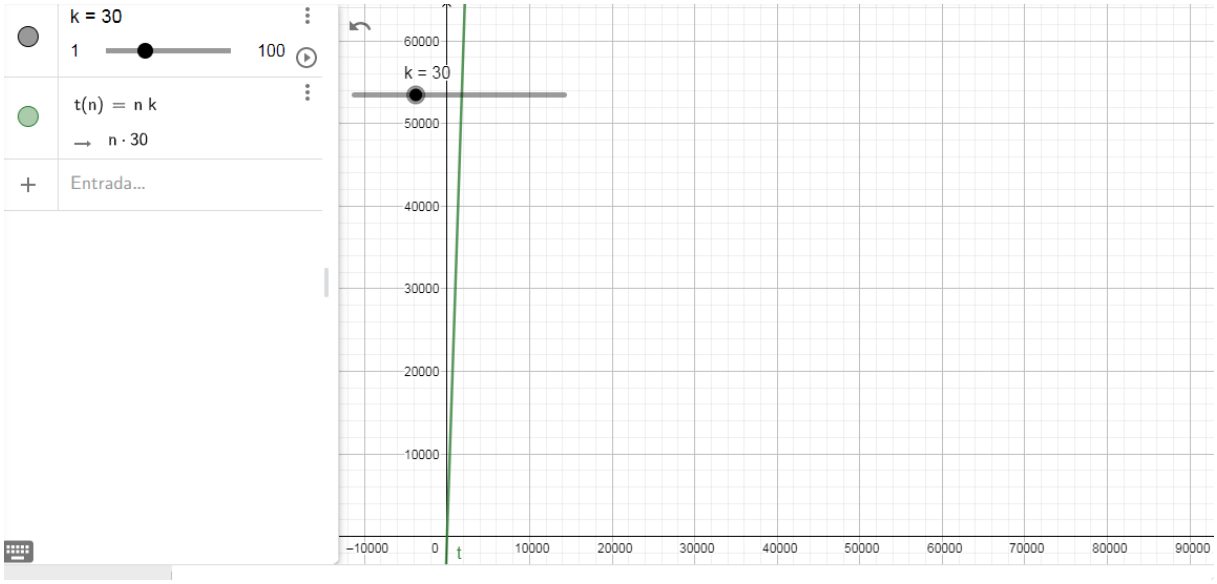



Figura 7. Análise da complexidade do RadixSort através de um controle deslizante.

### 2.3 – Comparação

Observando as funções de complexidade é possível perceber que o RadixSort tem um comportamento um pouco imprevisível porque ‘k’ pode ser variável, mas não tão imprevisível visto que o pior caso é difícil de acontecer, portanto deve ser feita uma análise estatística para saber se o RadixSort é melhor que o MergeSort. Em relação ao InsertionSort, tudo indica que esse algoritmo executará um maior número de instruções, pois sua função de complexidade é a maior e geralmente os dados não estão ordenados.

## 3- Análise Experimental

Os algoritmos foram colocados para ordenar 20 instâncias distintas geradas aleatoriamente para cada valor de ‘n’, sendo que ‘n’ pode valer 100, 1.000, 10.000, 100.000 e 1.000.000, o que totaliza 100 instâncias por algoritmo. Os tempos de execução das instâncias foram coletados e armazenados em tabelas, uma por algoritmo, em seguida foi calculado com 95% de confiança a distribuição da média utilizando a distribuição t para cada valor de ‘n’, e é gerado um gráfico de média para cada algoritmo. Por fim uma análise comparativa é feita.

### 3.1 InsertionSort

Logo abaixo é apresentado uma tabela que contém o tempo de execução do InsertionSort em segundos para as várias instâncias utilizadas no experimento, assim como as médias de cada instância e seus respectivos desvio padrão e intervalos de confiança das médias em 95%:

| | Tamanho das Instâncias | | | | |
|---|---|---|---|---|---|
| Instâncias | 100 | 1000 | 10000 | 100000 | 1000000 |
| | t(s) | t(s) | t(s) | t(s) | t(s) |
| 1 | 0,0010 | 0,1020 | 8,2060 | 916,7140 | 84263,9851 |
| 2 | 0,0000 | 0,0970 | 8,1030 | 910,5780 | 82903,5078 |
| 3 | 0,0010 | 0,1950 | 8,2760 | 894,9730 | 82663,6088 |
| 4 | 0,0000 | 0,0960 | 8,6220 | 908,8570 | 82576,9549 |
| 5 | 0,0010 | 0,0950 | 9,3050 | 997,6330 | 82363,9656 |
| 6 | 0,0000 | 0,0940 | 8,2320 | 914,7620 | 81926,5561 |
| 7 | 0,0010 | 0,0910 | 8,3400 | 853,2014 | 82213,3740 |
| 8 | 0,0010 | 0,0750 | 8,2080 | 831,7764 | 82099,3656 |
| 9 | 0,0010 | 0,0810 | 8,2230 | 828,5472 | 82057,8030 |
| 10 | 0,0000 | 0,0800 | 8,5880 | 805,4436 | 81876,5988 |
| 11 | 0,0010 | 0,0770 | 8,4100 | 849,7632 | 82028,6074 |
| 12 | 0,0000 | 0,0780 | 9,1930 | 881,4468 | 81880,6127 |
| 13 | 0,0010 | 0,0770 | 8,3950 | 826,0200 | 81715,0813 |
| 14 | 0,0010 | 0,0750 | 8,2800 | 799,2348 | 82111,7727 |
| 15 | 0,0010 | 0,0750 | 8,2680 | 816,8628 | 82264,3070 |
| 16 | 0,0010 | 0,0750 | 8,0220 | 800,3424 | 82000,8827 |
| 17 | 0,0020 | 0,0800 | 8,4910 | 826,0200 | 81771,6156 |
| 18 | 0,0010 | 0,0770 | 8,9250 | 829,2180 | 81953,5556 |
| 19 | 0,0010 | 0,0800 | 9,4600 | 897,2964 | 81962,4957 |
| 20 | 0,0020 | 0,0750 | 8,7290 | 805,5216 | 81839,3474 |
| Medias | 0,00085001 | 0,088749969 | 8,513800001 | 859,7105836 | 82223,69989 |
| s | 0,000587184 | 0,026578392 | 0,410731741 | 53,02729436 | 570,6112366 |
| Intervalo | (0.001,0.001) | (0.076,0.101) | (8.322,8.706) | (834.893,884.528) | (81956.646,82490.754) |

Figura 8. Tabela dos tempos de execução do InsertionSort

Pela tabela é possível perceber que à medida que o tamanho da instância aumenta o InsertionSort demora muito para ordenar os dados, e que para instâncias pequenas a ordenação é feita de maneira eficiente. Isso também é possível de observar no gráfico abaixo que contém o intervalo de tempo médio com 95% de confiança para cada tamanho ‘n’ de instância:

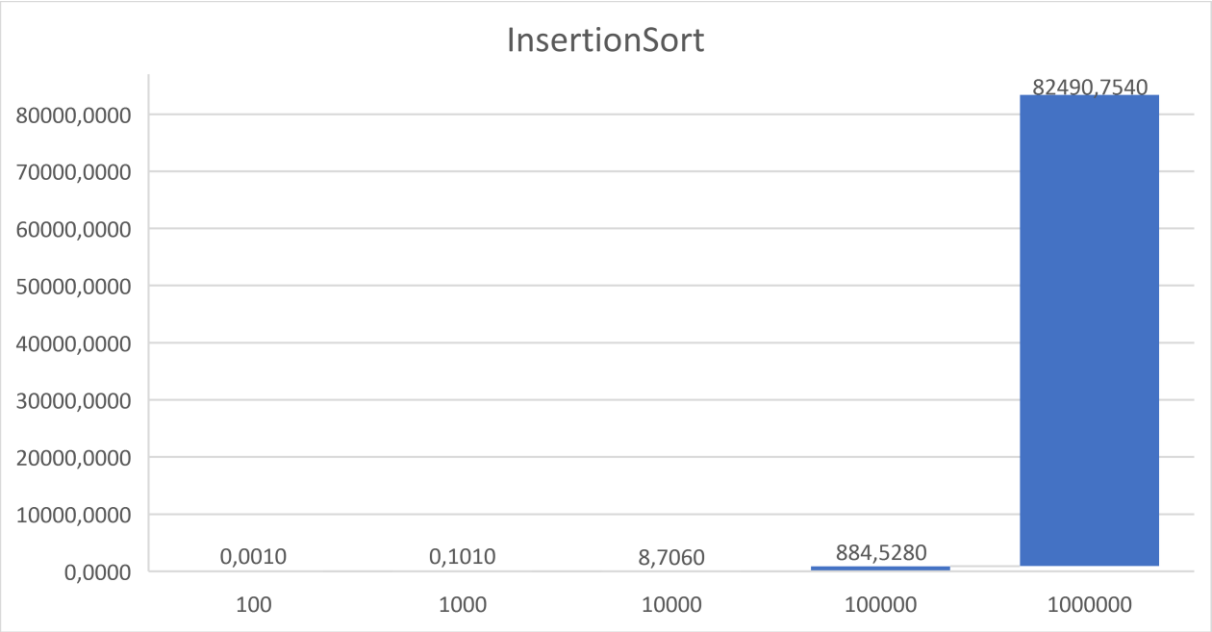



Figura 9. Gráfico do tempo médio de execução do InsertionSort.

Cabe ressaltar que a partir da implementação feita o InsertionSort demora em média várias horas para ordenar um vetor de 1 milhão de elementos.

### 3.2 MergeSort

Adiante é apresentado uma tabela que contém o tempo de execução do MergeSort em segundos para as várias instâncias utilizadas no experimento, assim como as médias de cada instância e seus respectivos desvio padrão e intervalos de confiança da média em 95%:

| | Tamanho das Instâncias | | | | |
|---|---|---|---|---|---|
| **Instâncias** | **100** | **1000** | **10000** | **100000** | **1000000** |
| | **t(s)** | **t(s)** | **t(s)** | **t(s)** | **t(s)** |
| **1** | 0,0000 | 0,0070 | 0,0730 | 1,0560 | 13,3536 |
| **2** | 0,0010 | 0,0070 | 0,0760 | 0,9700 | 13,9252 |
| **3** | 0,0000 | 0,0090 | 0,0880 | 0,9520 | 15,9470 |
| **4** | 0,0010 | 0,0080 | 0,1030 | 1,3140 | 13,7164 |
| **5** | 0,0000 | 0,0050 | 0,0730 | 0,9770 | 13,7072 |
| **6** | 0,0010 | 0,0060 | 0,0730 | 0,9640 | 13,0416 |
| **7** | 0,0000 | 0,0050 | 0,0750 | 0,8580 | 14,4612 |
| **8** | 0,0010 | 0,0050 | 0,0910 | 0,9048 | 13,2288 |
| **9** | 0,0000 | 0,0060 | 0,0740 | 0,8580 | 12,5424 |
| **10** | 0,0010 | 0,0050 | 0,0760 | 0,8580 | 13,4784 |
| **11** | 0,0000 | 0,0060 | 0,0930 | 1,0296 | 12,2460 |
| **12** | 0,0010 | 0,0050 | 0,0720 | 0,8736 | 13,4238 |
| **13** | 0,0000 | 0,0050 | 0,0730 | 0,8736 | 12,6828 |
| **14** | 0,0000 | 0,0060 | 0,0760 | 0,8424 | 12,9636 |
| **15** | 0,0000 | 0,0060 | 0,0730 | 0,8580 | 16,0230 |
| **16** | 0,0010 | 0,0050 | 0,0740 | 0,8580 | 13,4250 |
| **17** | 0,0010 | 0,0050 | 0,0960 | 0,8580 | 14,7734 |
| **18** | 0,0010 | 0,0060 | 0,0930 | 0,8736 | 13,4450 |
| **19** | 0,0010 | 0,0060 | 0,0720 | 0,8580 | 13,4432 |
| **20** | 0,0000 | 0,0050 | 0,0740 | 0,8580 | 14,1656 |
| **Médias** | 0,0005 | 0,005900025 | 0,079900014 | 0,924730015 | 13,69966011 |
| **s** | 0,000512989 | 0,001119138 | 0,009898939 | 0,111796209 | 0,988976218 |
| **Intervalo** | (0, 0.001) | (0.005,0.006) | (0.075,0.085) | (0.872,0.977) | (13.237,14.163) |

Figura 10. Tabela dos tempos de execução do MergeSort

Observando a tabela é possível perceber que à medida que o tamanho da instância aumenta o MergeSort não demora muito tempo para ordenar os dados. É um algoritmo eficiente. Isso também é possível de observar no gráfico abaixo que contém o intervalo de tempo médio com 95% de confiança para cada tamanho ‘n’ de instância:

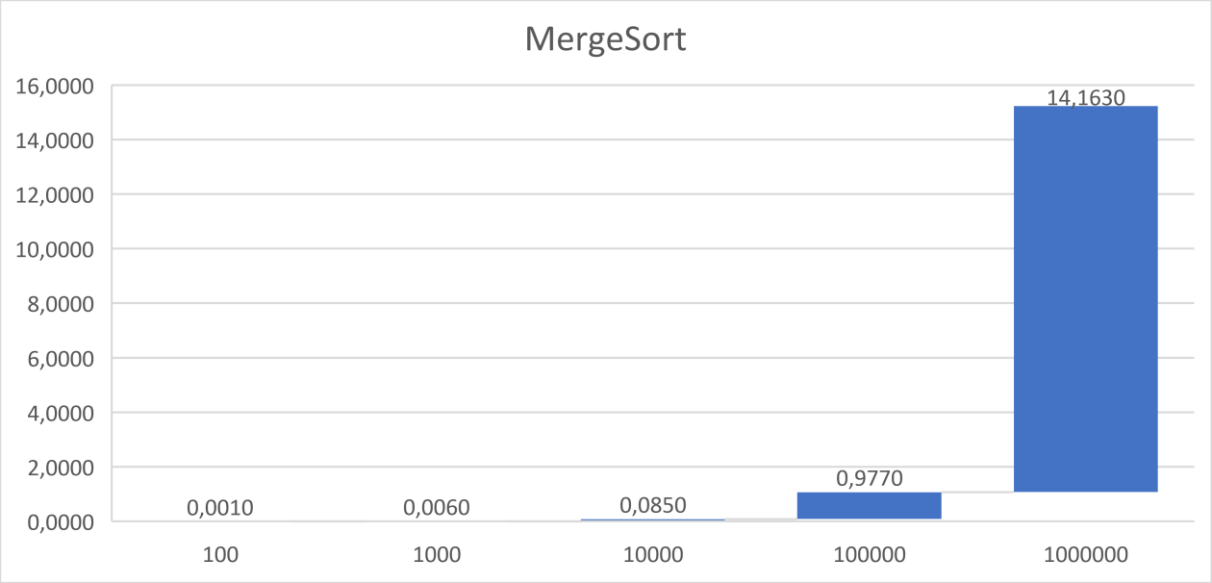



Figura 11. Gráfico do tempo médio de execução do MergeSort

O MergeSort se mostrou um algoritmo eficiente tanto para instâncias pequenas quanto para instâncias grandes, isso acontece porque esse algoritmo realiza somente n*log(n) comparações resultando um poucas iterações para chegar ao resultado esperado.

### 3.3 RadixSort

Abaixo é apresentado uma tabela que contém o tempo de execução do RadixSort em segundos para as várias instâncias utilizadas no experimento, assim como as médias de cada instância e seus respectivos desvio padrão e intervalos de confiança da média em 95%:

| | Tamanho das Instâncias | | | | |
|---|---|---|---|---|---|
| Instâncias | 100 | 1000 | 10000 | 100000 | 1000000 |
| | t(s) | t(s) | t(s) | t(s) | t(s) |
| 1 | 0,0010 | 0,0060 | 0,0440 | 0,6480 | 9,2508 |
| 2 | 0,0000 | 0,0040 | 0,0420 | 0,6230 | 8,8512 |
| 3 | 0,0010 | 0,0050 | 0,0440 | 0,6310 | 9,3212 |
| 4 | 0,0000 | 0,0030 | 0,0590 | 0,8110 | 9,2196 |
| 5 | 0,0010 | 0,0040 | 0,0470 | 0,6340 | 8,6580 |
| 6 | 0,0000 | 0,0040 | 0,0440 | 0,6140 | 9,5004 |
| 7 | 0,0000 | 0,0030 | 0,0440 | 0,5616 | 9,1416 |
| 8 | 0,0000 | 0,0040 | 0,0450 | 0,5616 | 8,4552 |
| 9 | 0,0020 | 0,0030 | 0,0470 | 0,5616 | 8,2212 |
| 10 | 0,0000 | 0,0030 | 0,0450 | 0,5460 | 7,9716 |
| 11 | 0,0010 | 0,0030 | 0,0520 | 0,6084 | 9,0168 |
| 12 | 0,0000 | 0,0030 | 0,0480 | 0,5460 | 9,9996 |
| 13 | 0,0000 | 0,0030 | 0,0590 | 0,6396 | 8,9388 |
| 14 | 0,0010 | 0,0030 | 0,0470 | 0,5772 | 9,0168 |
| 15 | 0,0010 | 0,0030 | 0,0440 | 0,5616 | 9,1728 |
| 16 | 0,0000 | 0,0030 | 0,0430 | 0,5616 | 8,9086 |
| 17 | 0,0000 | 0,0030 | 0,0590 | 0,5616 | 10,0564 |
| 18 | 0,0000 | 0,0030 | 0,0490 | 0,5460 | 8,7868 |
| 19 | 0,0000 | 0,0030 | 0,0450 | 0,5616 | 8,7054 |
| 20 | 0,0000 | 0,0030 | 0,0450 | 0,5616 | 9,9668 |
| Médias | 0,000399995 | 0,003449988 | 0,047599959 | 0,595850003 | 9,057980049 |
| s | 0,000598246 | 0,000825594 | 0,005413478 | 0,061569626 | 0,549746899 |
| Intervalo | (0,0.001) | (0.003,0.004) | (0.045,0.05) | (0.567,0.625) | (8.801,9.315) |

Figura 12. Tabela dos tempos de execução do RadixSort

A partir dos dados coletados acima percebe-se que em poucos segundos o RadixSort ordena os dados e leva em média 9 segundos para ordenar 1 milhão de elementos, portanto é um algoritmo muito eficiente. Isso também é possível de observar no gráfico abaixo que contém o intervalo de tempo médio com 95% de confiança para cada tamanho ‘n’ de instância:

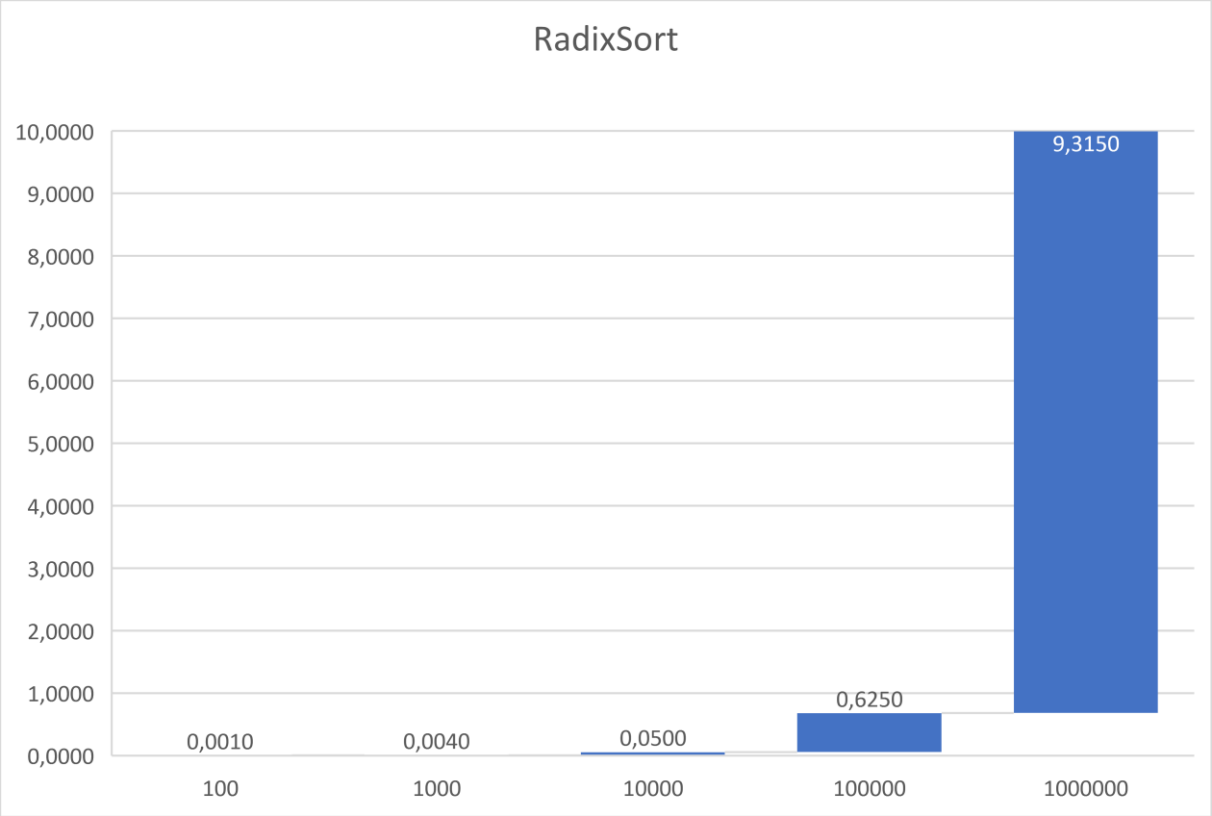



Figura 13. Gráfico do tempo médio de execução do RadixSort

A partir da análise do gráfico acima é notável que o RadixSort se mostrou um algoritmo eficiente tanto para instâncias pequenas quanto para instâncias grandes, isso acontece porque esse algoritmo realiza somente n*k comparações resultando um poucas iterações para chegar ao resultado esperado.

### 3.4- Comparação

Após a execução das diversas instâncias de diversos tamanhos é possível perceber que para instâncias de tamanho menor igual a 10000, não faz muita diferença em qual algoritmo utilizar para ordenar, pois o tempo de execução é bem similar entre os mesmos. Porém o InsertionSort a partir de instâncias de tamanho maior que 10000 mostrou resultados muito inferiores em relação ao MergeSort e RadixSort, demorando minutos ou horas para ordenar um vetor enquanto o MergeSort e o RadixSort ordenam em segundos.

A partir da análise dos resultados obtidos também é possível perceber que para instâncias acima de 10000 elementos o RadixSort se mostrou mais eficiente que o MergerSort, porém a diferença é de apenas alguns segundos. A partir de uma análise estatística dos dados coletados no experimento utilizando o teste t com 95% de confiança será possível concluir com certeza qual é o melhor algoritmo para cada tamanho de instância.

## 5 – Análise Estatística a partir do teste t pareado com 95% de confiança

O Teste Estatístico T, é utilizado para encontrar o algoritmo que obteve estatisticamente o melhor desempenho. Como as mesmas instâncias são usadas para todos os algoritmos para cada valor de ‘n’, é possível realizar o teste T pareado para todos os valores de ‘n’.

**Para N = 100**

**Comparando InsertionSort com MergeSort:**

```
Media A 0.0008500000000000001
Media B 0.0005000000000000001
Desvio Padrao A 0.0005871429486123998
Desvio Padrao B 0.000512989176042577
Desvio Padrao das Diferencas 0.00017434086394791458
Graus de Liberdade: 39.25698611980413
Qual o valor mesmo?2.023
T = 2.023
( -2.691567766631268e-06  :  0.0007026915677666311 )
```

Figura 14. Comparação entre InsertionSort e MergeSort(N=100)

Como a comparação dos algoritmos InsertionSort e MergeSort utilizando o teste estatístico t resultou em um intervalo que contém o valor 0, então houve um empate estatístico.

**Comparando InsertionSort com RadixSort:**

```
Media A 0.0008500000000000001
Media B 0.0004
Desvio Padrao A 0.0005871429486123998
Desvio Padrao B 0.0005982430416161188
Desvio Padrao das Diferencas 0.00018743419898025124
Graus de Liberdade: 39.98527638331884
Qual o valor mesmo?2.021
T = 2.021
( 7.11954838609123e-05  :  0.0008288045161390878 )
```

Figura 15. Comparação entre InsertionSort e RadixSort(N=100)

Como a comparação dos algoritmos InsertionSort e RadixSort utilizando o teste estatístico t resultou em um intervalo que não contém o valor 0, então o algoritmo com menor média de tempo de execução é o melhor para instâncias de tamanho 100, ou seja, o RadixSort é o melhor algoritmo comparado ao InsertionSort em instâncias de tamanho 100.

**Comparando MergeSort com RadixSort:**

```
Media A 0.0005000000000000001
Media B 0.0004
Desvio Padrao A 0.000512989176042577
Desvio Padrao B 0.0005982430416161188
Desvio Padrao das Diferencas 0.00017621756887140218
Graus de Liberdade: 39.044918585064586
Qual o valor mesmo?2.023
T = 2.023
( -0.0002564881418268465  :  0.0004564881418268467 )
```

Figura 16. Comparação entre MergeSort e RadixSort(N=100)

Como a comparação dos algoritmos MergeSort e RadixSort utilizando o teste estatístico t resultou em um intervalo que contém o valor 0, então houve um empate estatístico.

**Para N = 1000**

**Comparando InsertionSort com MergeSort:**

```
Media A 0.08874999999999998
Media B 0.005900000000000002
Desvio Padrao A 0.02657833506329624
Desvio Padrao B 0.0011192102478745306
Desvio Padrao das Diferencas 0.00594836333084904
Graus de Liberdade: 19.074475895981017
T = 2.093
( 0.07040007554853293  :  0.09529992445146702 )
```

Figura 17. Comparação entre InsertionSort e MergeSort(N=1000)

Como a comparação dos algoritmos InsertionSort e MergeSort utilizando o teste estatístico t resultou em um intervalo que não contém o valor 0, então o algoritmo com menor média de tempo de execução é o melhor para instâncias de tamanho 1000, ou seja, o MergeSort é o melhor algoritmo comparado ao InsertionSort em instâncias de tamanho 1000.

**Comparando InsertionSort com RadixSort:**

```
Media A 0.08874999999999998
Media B 0.0034500000000000017
Desvio Padrao A 0.02657833506329624
Desvio Padrao B 0.0008255779474818966
Desvio Padrao das Diferencas 0.0059459628054849565
Graus de Liberdade: 19.04052373898058
T = 2.093
( 0.07285509984811997  :  0.09774490015188 )
```

Figura 18. Comparação entre InsertionSort e RadixSort(N=1000)

Como a comparação dos algoritmos InsertionSort e RadixSort utilizando o teste estatístico t resultou em um intervalo que não contém o valor 0, então o algoritmo com menor média de tempo de execução é o melhor para instâncias de tamanho 1000, ou seja, o RadixSort é o melhor algoritmo comparado ao InsertionSort em instâncias de tamanho 1000.

**Comparando MergeSort com RadixSort:**

```
Media A 0.005900000000000002
Media B 0.0034500000000000017
Desvio Padrao A 0.0011192102478745306
Desvio Padrao B 0.0008255779474818966
Desvio Padrao das Diferencas 0.00031098316082352347
Graus de Liberdade: 36.63257133322209
Qual o valor mesmo?2.026
T = 2.026
( 0.0018199481161715414  :  0.0030800518838284587 )
```

Figura 19. Comparação entre MergeSort e RadixSort(N=1000)

Como a comparação dos algoritmos MergeSort e RadixSort utilizando o teste estatístico t resultou em um intervalo que não contém o valor 0, então o algoritmo com menor média de tempo de execução é o melhor para instâncias de tamanho 1000, ou seja, o RadixSort é o melhor algoritmo comparado ao MergeSort em instâncias de tamanho 1000.

**Para N = 10000**

**Comparando InsertionSort com MergeSort:**

```
Media A 8.5138
Media B 0.0799
Desvio Padrao A 0.4107317731484676
Desvio Padrao B 0.009898963263100361
Desvio Padrao das Diferencas 0.09186908591777988
Graus de Liberdade: 19.024395626115176
T = 2.093
( 8.241618003174086  :  8.626181996825913 )
```

Figura 20. Comparação entre InsertionSort e MergeSort(N=10000)

Como a comparação dos algoritmos InsertionSort e MergeSort utilizando o teste estatístico t resultou em um intervalo que não contém o valor 0, então o algoritmo com menor média de tempo de execução é o melhor para instâncias de tamanho 10000, ou seja, o MergeSort é o melhor algoritmo comparado ao InsertionSort em instâncias de tamanho 10000.

**Comparando InsertionSort com RadixSort:**

```
Media A 8.5138
Media B 0.0476
Desvio Padrao A 0.4107317731484676
Desvio Padrao B 0.005413433583031636
Desvio Padrao das Diferencas 0.09185039323183165
Graus de Liberdade: 19.007295890425336
T = 2.093
( 8.273957126965778  :  8.658442873034224 )
```

Figura 21. Comparação entre InsertionSort e RadixSort(N=10000)

Como a comparação dos algoritmos InsertionSort e RadixSort utilizando o teste estatístico t resultou em um intervalo que não contém o valor 0, então o algoritmo com menor média de tempo de execução é o melhor para instâncias de tamanho 10000, ou seja, o RadixSort é o melhor algoritmo comparado ao InsertionSort em instâncias de tamanho 10000.

**Comparando MergeSort com RadixSort:**

```
Media A 0.0799
Media B 0.0476
Desvio Padrao A 0.009898963263100361
Desvio Padrao B 0.005413433583031636
Desvio Padrao das Diferencas 0.0025228430078198015
Graus de Liberdade: 30.529543814639617
Qual o valor mesmo?2.040
T = 2.04
( 0.0271534002640476  :  0.03744659973595239 )
```

Figura 22. Comparação entre MergeSort e RadixSort(N=10000)

Como a comparação dos algoritmos MergeSort e RadixSort utilizando o teste estatístico t resultou em um intervalo que não contém o valor 0, então o algoritmo com menor média de tempo de execução é o melhor para instâncias de tamanho 10000, ou seja, o RadixSort é o melhor algoritmo comparado ao MergeSort em instâncias de tamanho 10000.

**Para N = 100000**

**Comparando InsertionSort com MergeSort:**

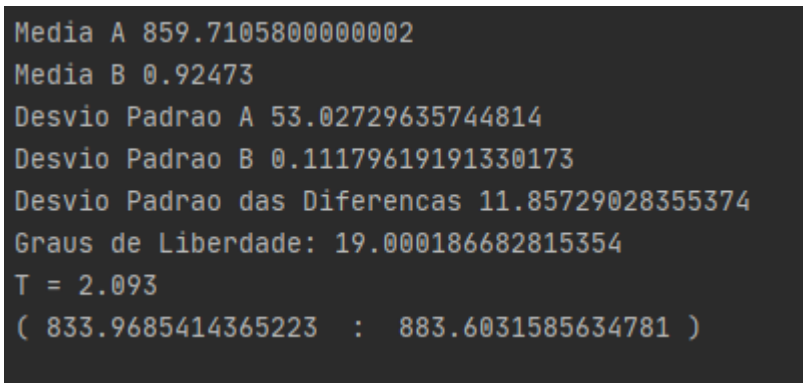



Figura 23. Comparação entre InsertionSort e MergeSort(N=100000)

Como a comparação dos algoritmos InsertionSort e MergeSort utilizando o teste estatístico t resultou em um intervalo que não contém o valor 0, então o algoritmo com menor média de tempo de execução é o melhor para instâncias de tamanho 100000, ou seja, o MergeSort é o melhor algoritmo comparado ao InsertionSort em instâncias de tamanho 100000.

**Comparando InsertionSort com RadixSort:**

```
Media A 859.7105800000002
Media B 0.5958500000000001
Desvio Padrao A 53.02729635744814
Desvio Padrao B 0.0615696396474414
Desvio Padrao das Diferencas 11.857271924437674
Graus de Liberdade: 19.000056621783422
T = 2.093
( 834.2974598621521  :  883.9320001378481 )
```

Figura 24. Comparação entre InsertionSort e RadixSort(N=100000)

Como a comparação dos algoritmos InsertionSort e RadixSort utilizando o teste estatístico t resultou em um intervalo que não contém o valor 0, então o algoritmo com menor média de tempo de execução é o melhor para instâncias de tamanho 100000, ou seja, o RadixSort é o melhor algoritmo comparado ao InsertionSort em instâncias de tamanho 100000.

**Comparando MergeSort com RadixSort:**

```
Media A 0.92473
Media B 0.5958500000000001
Desvio Padrao A 0.11179619191330173
Desvio Padrao B 0.0615696396474414
Desvio Padrao das Diferencas 0.028538753522737798
Graus de Liberdade: 30.665633639542996
Qual o valor mesmo?2.040
T = 2.04
( 0.27066094281361486  :  0.38709905718638504 )
```

Figura 25. Comparação entre MergeSort e RadixSort(N=100000)

Como a comparação dos algoritmos MergeSort e RadixSort utilizando o teste estatístico t resultou em um intervalo que não contém o valor 0, então o algoritmo com menor média de tempo de execução é o melhor para instâncias de tamanho

100000, ou seja, o RadixSort é o melhor algoritmo comparado ao MergeSort em instâncias de tamanho 100000.

**Para N = 1000000**

**Comparando InsertionSort com MergeSort:**

```
Media A 82223.69989000002
Media B 13.69966
Desvio Padrao A 570.6112364849014
Desvio Padrao B 0.9889762023638914
Desvio Padrao das Diferencas 127.5927429904924
Graus de Liberdade: 19.00012616541891
T = 2.093
( 81942.94861892091  :  82477.05184107913 )
```

Figura 26. Comparação entre InsertionSort e MergeSort(N=1000000)

Como a comparação dos algoritmos InsertionSort e MergeSort utilizando o teste estatístico t resultou em um intervalo que não contém o valor 0, então o algoritmo com menor média de tempo de execução é o melhor para instâncias de tamanho 1000000, ou seja, o MergeSort é o melhor algoritmo comparado ao InsertionSort em instâncias de tamanho 1000000.

**Comparando InsertionSort com RadixSort:**

```
Media A 82223.69989000002
Media B 9.05798
Desvio Padrao A 570.6112364849014
Desvio Padrao B 0.5497468466198066
Desvio Padrao das Diferencas 127.59261056668278
Graus de Liberdade: 19.000038984695387
T = 2.093
( 81947.59057608395  :  82481.69324391609 )
```

Figura 27. Comparação entre InsertionSort e RadixSort(N=1000000)

Como a comparação dos algoritmos InsertionSort e RadixSort utilizando o teste estatístico t resultou em um intervalo que não contém o valor 0, então o algoritmo com menor média de tempo de execução é o melhor para instâncias de tamanho 1000000, ou seja, o RadixSort é o melhor algoritmo comparado ao InsertionSort em instâncias de tamanho 1000000.

**Comparando MergeSort com RadixSort:**

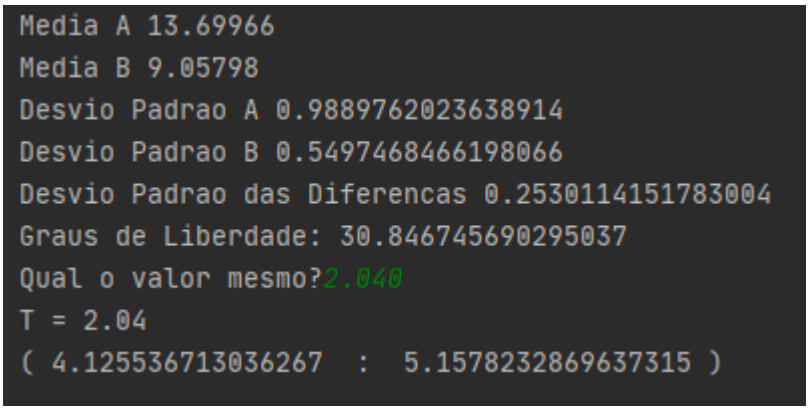


Figura 28. Comparação entre MergeSort e RadixSort(N=1000000)

Como a comparação dos algoritmos MergeSort e RadixSort utilizando o teste estatístico t resultou em um intervalo que não contém o valor 0, então o algoritmo com menor média de tempo de execução é o melhor para instâncias de tamanho 1000000, ou seja, o RadixSort é o melhor algoritmo comparado ao MergeSort em instâncias de tamanho 1000000.

**4- Conclusão**

Analisando os testes estatísticos é possível perceber que para instâncias de tamanho menor ou igual a 100, ou seja, instâncias relativamente pequenas os algoritmos se encontram em um empate estatístico. Porém, o RadixSort obteve um melhor desempenho comparado aos demais algoritmos sendo superior em quase todos os testes, somente comparado ao MergeSort em instâncias de tamanho 100 que não, e o MergeSort, a partir de instâncias de tamanho 100, foi superior somente ao InsertionSort, que por sua vez foi o algoritmo com pior desempenho. Conclui-se

que o algoritmo de ordenação com melhor desempenho dentro os analisados é o RadixSort, em seguida o MergeSort e por último o InsertionSort.

**Referências:**

https://pt.wikipedia.org/wiki/Insertion_sort

https://pt.wikipedia.org/wiki/Merge_sort

https://www.geogebra.org/calculator

https://pt.wikipedia.org/wiki/Radix_sort

https://mathcracker.com/pt/media-calculadora-intervalo-confianca-t#results