



Universidade Federal do Rio Grande do Norte – UFRN  
Centro de Ensino Superior do Seridó – CERES  
Departamento de Computação e Tecnologia – DCT  
Bacharelado em Sistemas de Informação – BSI

# Análise de tempo de execução de algoritmos de ordenação

Lucas Mateus da Silva

Orientador: Prof. Dr. João Paulo de Souza Medeiros

**Relatório Técnico** apresentado ao Curso de Bacharelado em Sistemas de Informação como parte dos requisitos para aprovação na disciplina de Estrutura de dados.

Caicó, RN, 28 de maio de 2023

# Resumo

Trabalho desenvolvido com o objetivo de averiguar os tempos de execução de algoritmos de ordenação: merge-sort, insertion-sort, quick-sort e selection-sort. Também fazendo a análise assintótica e visualização gráfica. Todos os algoritmos mencionados neste documento podem ser encontrados no repositório mencionado ao final deste documento.

**Keywords:** Algoritmos; Ordenação; Complexidade.

# Sumário

<b>Lista de Figuras</b>	<b>3</b>
<b>1 Insertion-sort</b>	<b>4</b>
1.1 O algoritmo . . . . .	4
1.2 Complexidade . . . . .	4
1.2.1 melhor caso . . . . .	4
1.2.2 pior caso . . . . .	4
1.2.3 Médio caso . . . . .	5
1.3 Implementação . . . . .	5
<b>2 Selection-sort</b>	<b>8</b>
2.1 O algoritmo . . . . .	8
2.2 Complexidade . . . . .	8
2.3 Implementação . . . . .	8
<b>3 Quick-sort</b>	<b>10</b>
3.1 O algoritmo . . . . .	10
3.2 Complexidade . . . . .	10
3.2.1 melhor caso . . . . .	10
3.2.2 pior caso . . . . .	11
3.2.3 Médio caso . . . . .	11
3.3 Implementação . . . . .	11
<b>4 Merge-sort</b>	<b>13</b>
4.1 O algoritmo . . . . .	13
4.2 Complexidade . . . . .	14
4.3 Implementação . . . . .	14
<b>5 Counting-sort</b>	<b>17</b>
5.1 O algoritmo . . . . .	17
5.2 Complexidade . . . . .	17
5.3 Implementação . . . . .	18
<b>6 Conclusões</b>	<b>21</b>
6.1 Links Externos . . . . .	21

## Lista de Figuras

1.1	melhor caso do insertion . . . . .	5
1.2	pior caso do insertion . . . . .	6
1.3	insertion-sort adaptado com python . . . . .	6
1.4	tempos de execução do insertion-sort . . . . .	7
2.1	selection-sort adaptado com python . . . . .	9
2.2	tempo de execução do selection-sort . . . . .	9
3.1	quick-sort adaptado com python . . . . .	11
3.2	tempos de execução do quick-sort . . . . .	12
4.1	calculo de tempo de execução do merge . . . . .	15
4.2	merge-sort adaptado com python . . . . .	16
4.3	tempo de execução do merge-sort . . . . .	16
5.1	calculo de tempo de execução do counting sort . . . . .	18
5.2	counting-sort adaptado com python . . . . .	19
5.3	tempo de execução do counting-sort . . . . .	20
6.1	Comparação entre todos os algoritmos . . . . .	21
6.2	Comparação entre quick, merge e distribution . . . . .	22

# 1. Insertion-sort

O algoritmo Insertion-sort é um método para ordenação de vetores que percorre o vetor a partir do segundo elemento e compara cada elemento com os anteriores, realizando trocas quando necessário para encontrar a posição correta. Esse processo é repetido para cada elemento subsequente, assim ordenando o vetor. Embora simples de implementar, o Insertion-sort não é tão eficiente quanto outros algoritmos mais avançados, tendo uma complexidade de tempo  $O(n^2)$  no pior caso e no caso médio, onde "n" é o número de elementos no array. No entanto, é útil quando o array já está parcialmente ordenado, pois nesse caso ele pode executar de forma mais rápida com complexidade de  $O(n)$ .

## 1.1 O algoritmo

```
algoritmo insertion-sort( $v, n$ )  
  para cada  $e$  de 2 até  $n$  faça  
     $i \leftarrow e$   
    enquanto  $i > 1$  e  $v[i-1] > v[i]$  faça  
      swap( $v[i-1], v[i]$ )  
       $i \leftarrow i - 1$   
    fim enquanto  
  fim para
```

## 1.2 Complexidade

### 1.2.1 melhor caso

O melhor caso para o tempo de execução desse algoritmo ocorre quando o vetor já está ordenado, logo o algoritmo só percorrerá o vetor inteiro sem nunca entrar dentro da segunda estrutura de repetição. Com esse caso o tempo de execução é linear e a complexidade é  $O(n)$ . (visto em [1.1](#))

### 1.2.2 pior caso

O pior caso em tempo de execução para esse algoritmo ocorre quando o vetor está ordenado de maneira decrescente, logo o algoritmo terá de executar todas as trocas. Passando sempre pelas duas estruturas de repetição "n" vezes, o algoritmo irá ter um tempo de execução quadrático com complexidade  $O(n^2)$ . (visto em [1.2](#))

$$T_b(n) = (n-1)c_1 + (n-1)c_2 + (n-1)c_3$$

$$T_b(n) = nc_1 + nc_2 + nc_3 + (-c_2 - c_3)$$

$$T_b(n) = an - b$$

$$O(n) = \text{linear}$$

Figura 1.1: melhor caso do insertion

### 1.2.3 Médio caso

O tempo de execução esperado ocorre quando o vetor tem valores aleatórios, podendo ser parcialmente ordenados ou não. Para esses casos o tempo de execução esperado é quadrático com complexidade  $O(n^2)$ .

## 1.3 Implementação

O algoritmo foi implementado usando a linguagem python como visto na figura 1.3, e seus resultados averiguados na figura 1.4 mostram que apesar de também ser quadrático, no pior caso, nomeado "p-insertion", o algoritmo irá ser pior como visto nos cálculos da figura 1.2. Além disso o tempo de execução no médio caso seguiu o esperado e foi linear.

$$\begin{aligned}
 T_W(n) &= nC_3 + (n-1)C_2 + C_3 \left( \sum_{i=2}^n i \right) + (C_4 + C_5) \left( \sum_{i=1}^{n-1} i \right) \\
 T_W(n) &= nC_3 + (n-1)C_2 + C_3 \left( \frac{n^2 + n - 2}{2} \right) + (C_4 + C_5) \left( \frac{n^2 - n}{2} \right) \\
 T_W(n) &= n(C_3 + C_2) + (C_3 - C_2) + n \frac{C_3}{2} + \frac{nC_3}{2} - C_3 + n^2 \frac{(C_3 + C_5)}{2} - \\
 &\quad n \left( \frac{C_4 + C_5}{2} \right) \\
 T_W(n) &= n^2 \left( \frac{C_3 + C_4 + C_5}{2} \right) + n \left( C_3 + C_2 + \frac{C_3}{2} - \frac{C_4 + C_5}{2} \right) + C_2 \\
 &\quad \underline{\hspace{10em}} \\
 n^2 a + nb + c &\rightarrow \text{quadrática } O(n^2)
 \end{aligned}$$

CS Digitalizado com CamScanner

Figura 1.2: pior caso do insertion

```

1 def insertionSort(vetor: list[int], n: int):
2     for e in range(1, n):
3         i: int = e
4         while (i > 0) and vetor[i-1] > vetor[i]:
5             vetor[i], vetor[i-1] = vetor[i-1], vetor[i]
6             i -= 1

```

Figura 1.3: insertion-sort adaptado com python

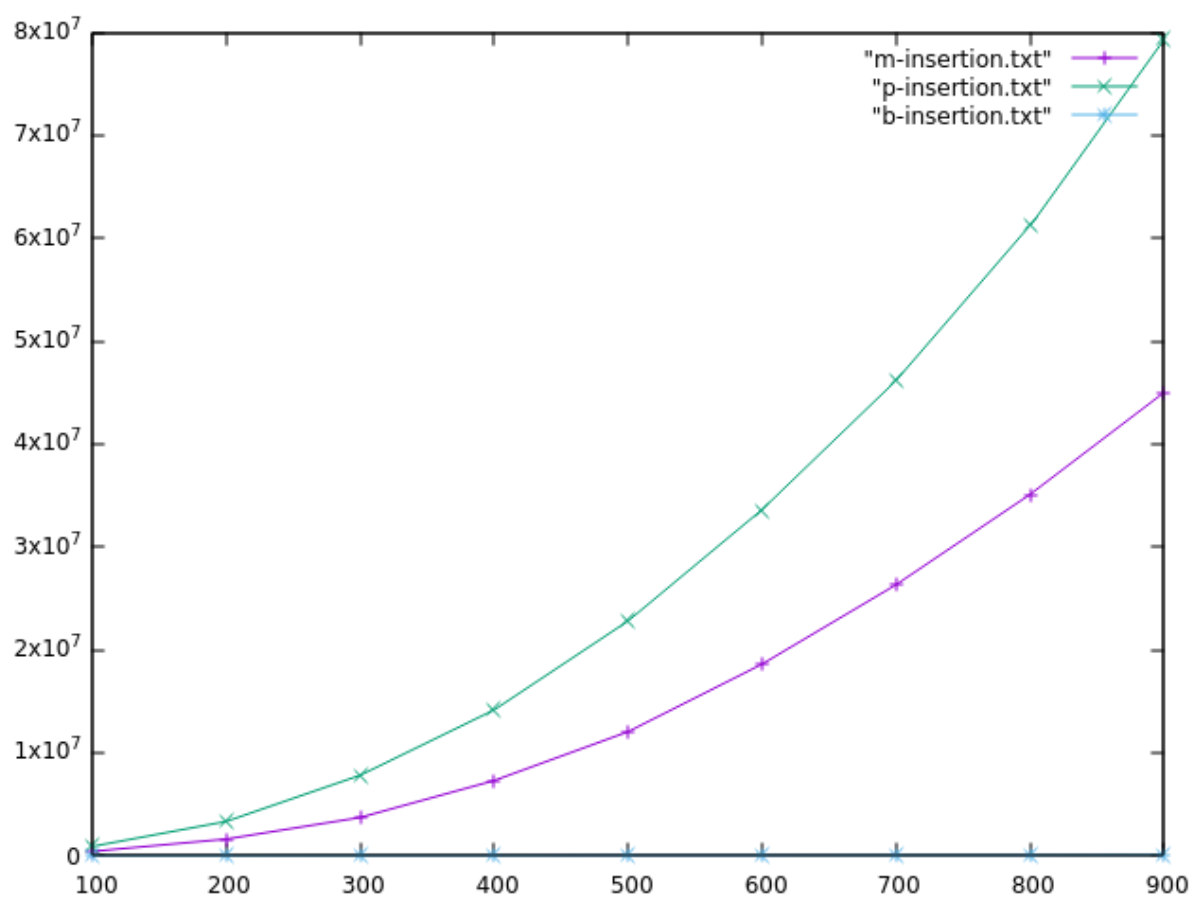


Figura 1.4: tempos de execução do insertion-sort



## 2. Selection-sort

A ordenação por seleção ou selection-sort consiste em selecionar o menor item e colocar na primeira posição, selecionar o segundo menor item e colocar na segunda posição, seguindo estes passos até que reste um único elemento. O selection-sort tem uma complexidade de tempo  $O(n^2)$  em todos os casos, onde "n" é o número de elementos no array, e pode ser menos eficiente para listas grandes. No entanto, é um algoritmo in-place, não exigindo espaço adicional, o que pode ser vantajoso em situações com restrição de espaço.

### 2.1 O algoritmo

```
algoritmo selection-sort( $v, n$ )  
  para cada  $i$  de 1 até  $(n - 1)$  faça  
     $m \leftarrow i$   
    para cada  $j$  de  $(i + 1)$  até  $n$  faça  
      se  $v[m] > v[j]$  então  
         $m \leftarrow j$   
      fim se  
    fim para  
    swap( $v[m], v[i]$ )  
  fim para
```

### 2.2 Complexidade

Esse algoritmo não possui melhor nem pior caso, o tempo de execução do selection-sort só alterado pelo tamanho do vetor, afetará quantas vezes cada estrutura de repetição irá ser executada. Não foi anexado os calculos desse algoritmo mas, como existem duas estruturas de repetição aninhadas que tem suas repetições associadas ao número de itens do vetor, o tempo será quadrático, com complexidade  $O(n^2)$ .

### 2.3 Implementação

O algoritmo foi implementado usando a linguagem python, e seus resultados mostram que quanto maior o vetor, maior é o crescimento do tempo de execução que cresce de maneira quadrática como visto na figura [2.2](#).

```
1 def selectionSort(vetor: list[int], n: int):  
2  
3     for i in range(0, (n - 1)):  
4         m = i  
5  
6         for j in range(i+1, n):  
7  
8             if vetor[m] > vetor[j]:  
9                 m = j  
10  
11         vetor[i], vetor[m] = vetor[m], vetor[i]
```

Figura 2.1: selection-sort adaptado com python

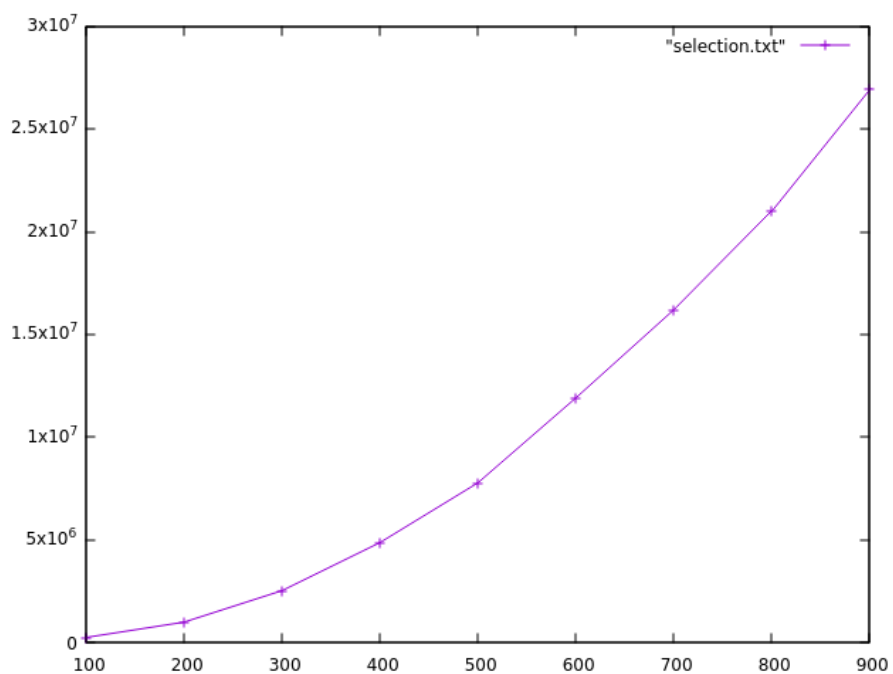


Figura 2.2: tempo de execução do selection-sort

## 3. Quick-sort

O algoritmo quick-sort assim como o nome diz é considerado um dos mais rápidos para uma ampla variedade de situações. É um método eficiente de classificação que utiliza a estratégia "dividir e conquistar" em que ideia básica é dividir o problema de ordenar um conjunto com  $n$  itens em dois problemas menores, e esses problemas menores são ordenados independentemente e os resultados são combinados para produzir a solução final. O algoritmo divide seu array de entrada em dois sub-arrays a partir de um pivô, para em seguida realizar o mesmo procedimento nos dois arrays menores até um array unitário. Possui complexidade  $O(n^2)$  no pior caso e  $O(n \log n)$  no melhor e médio caso.

### 3.1 O algoritmo

```

algoritmo quick-sort( $v, s, e$ )
  se  $s < e$  então
     $p \leftarrow \text{partition}(v, s, e)$ 
    quick-sort( $v, s, (p - 1)$ )
    quick-sort( $v, (p + 1), e$ )
  fim se
algoritmo partition( $v, s, e$ )
   $d \leftarrow s - 1$ 
  para cada  $i$  de  $s$  até  $(e - 1)$  faça
    se  $v[i] \leq v[e]$  então
       $d \leftarrow d + 1$ 
      swap( $v[d], v[i]$ )
    fim se
  fim para
  swap( $v[d + 1], v[e]$ )
  retorne  $(d + 1)$ 

```

### 3.2 Complexidade

#### 3.2.1 melhor caso

O melhor caso do do quick-sort ocorre quando o pivô escolhido sempre é o elemento central do vetor ordenado. Seguindo essa lógica do melhor caso o algoritmo terá complexidade  $O(n \cdot \log n)$ .

### 3.2.2 pior caso

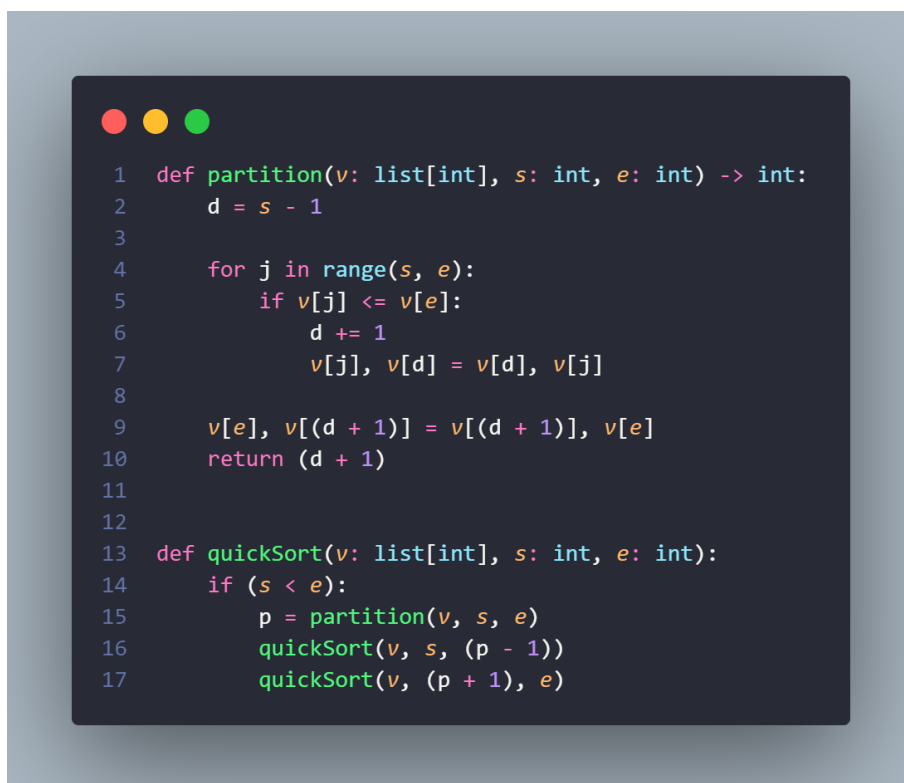
Assim como no algoritmo insertion-sort, o pior caso do quick-sort ocorre quando o vetor está ordenado de maneira decrescente, pois assim o pivô escolhido pode ser sempre o menor ou o maior elemento do vetor. Isso faz com que o algoritmo divida o vetor em uma parte com todos os elementos, exceto o pivô, e outra parte vazia. Esse comportamento leva a recursões excessivas e desnecessárias, o que impacta negativamente o desempenho.

### 3.2.3 Médio caso

O tempo de execução esperado tem tempo mais próximo do pior caso, tendo complexidade  $O(n \cdot \log n)$ .

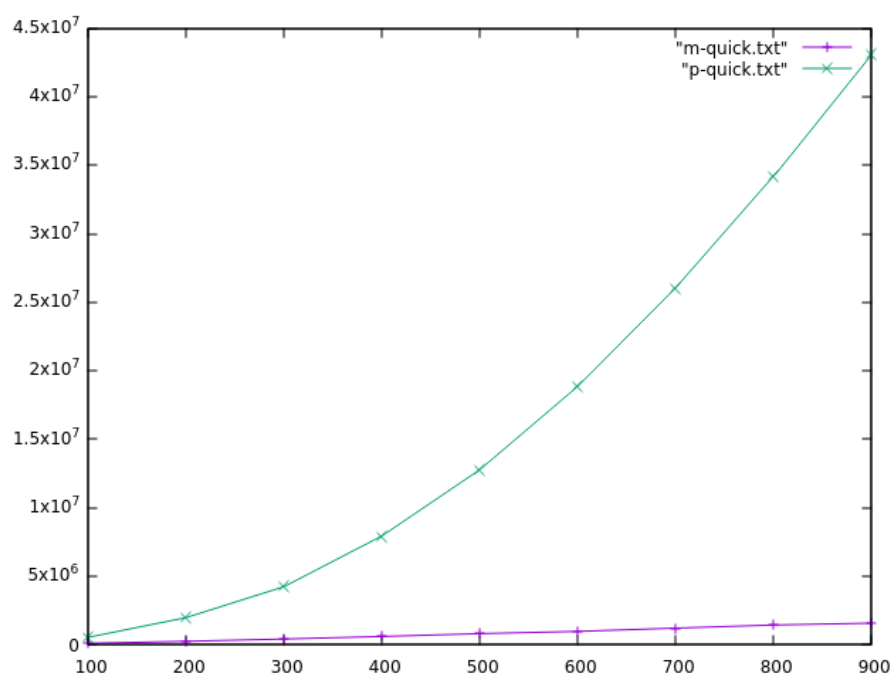
## 3.3 Implementação

O algoritmo foi implementado usando a linguagem python, e seus resultados, vistos na figura 3.2 mostram que no pior caso, nomeado "p-quick", o algoritmo irá ter crescimento quadrático. Ao caso médio seguiu com crescimento logaritmico.



```
1 def partition(v: list[int], s: int, e: int) -> int:
2     d = s - 1
3
4     for j in range(s, e):
5         if v[j] <= v[e]:
6             d += 1
7             v[j], v[d] = v[d], v[j]
8
9     v[e], v[(d + 1)] = v[(d + 1)], v[e]
10    return (d + 1)
11
12
13 def quickSort(v: list[int], s: int, e: int):
14     if (s < e):
15         p = partition(v, s, e)
16         quickSort(v, s, (p - 1))
17         quickSort(v, (p + 1), e)
```

Figura 3.1: quick-sort adaptado com python



**Figura 3.2:** tempos de execução do quick-sort

## 4. Merge-sort

O algoritmo merge-sort é outro que também segue a estratégia de dividir para conquistar com complexidade  $O(n \log n)$  para todos os casos. Inicialmente, o array é dividida ao meio até que cada sub-array contenha apenas um elemento. Em seguida, o algoritmo mescla repetidamente os sub-arrays, comparando os elementos em pares e criando um novo array ordenado. Esse processo de mesclagem é realizado de forma recursiva, formando sub-arrays maiores e ordenados a cada etapa. Por fim, os dois sub-arrays finais são mesclados em um único array ordenado.

### 4.1 O algoritmo

```

algoritmo merge-sort( $v, s, e$ )
  se  $s < e$  então
     $m \leftarrow (s + e) / 2$ 
     $merge\_sort(v, s, m)$ 
     $merge\_sort(v, (m + 1), e)$ 
     $merge(v, s, m, e)$ 
  fim se
algoritmo merge( $v, s, m, e$ )
   $p \leftarrow s$ 
   $q \leftarrow m + 1$ 
   $w \leftarrow$  uma lista vazia de tamanho  $e - s + 1$ 
  para cada  $i$  de 1 até  $(e - s + 1)$  faça
    se  $(q > e)$  ou  $(p \leq m)$  e  $(v[p] < v[q])$  então
       $w[i] \leftarrow v[p]$ 
       $p \leftarrow p + 1$ 
    else
       $w[i] \leftarrow v[q]$ 
       $q \leftarrow q + 1$ 
    fim se
  fim para
  para cada  $i$  de 1 até  $(e - s + 1)$  faça
     $v[s + i - 1] \leftarrow w[i]$ 
  fim para

```

## 4.2 Complexidade

O tempo de execução esperado ocorre quando o vetor tem valores aleatorios, podendo ser parcialmente ordenados ou não. Para esses casos o tempo de execução esperado é polilogaritmico com complexidade  $O(n \cdot \log n)$ . (visto em [4.1](#))

## 4.3 Implementação

O algoritmo foi implementado usando a linguagem python, e seus resultados, vistos na figura [4.3](#), apontam que apesar não parecerem se adequar ao esperado, pela escala em que está sendo exibido, o crescimento seguiu exatamente como o esperado, podendo ser melhor visualizado em outra escala na figura [6.2](#).

$$T(1) = C_3 \rightarrow \text{Caso base}$$

$$T(n) = C_1 + C_2 + C_3 + C_4 + C_5 + T(n/2) + T(n/2) + T^n(n)$$

$$T(n) = a + 2T(n/2) + T^M(n)$$

$$* T(n/2) = a + 2T(n/4) + T^M(n/2)$$

$$T(n) = 3a + 4T(n/4) + T^M(n/2) + T^M(n)$$

$$T(n) = (x-1)a + nT(n/x) + \sum_{i=0}^{(\log_2 x) - 1} 2^i T^M(n/2^i) *$$

IGUALANDO AO CASO BASE

$$T(n) = (n-1)a + nT(n/n) + \sum_{i=0}^{\log_2 n} 2^i T^M(n/2^i)$$

$$T(n) = (n-1)a + nc_1 + bn \cdot \log_2 n + c \cdot 2^{\log_2 n} - 1$$

$$T(n) = (n-1)a + c_1 + b \log_2 n + c(n-1)$$

$$T(n) = b \log_2 n + n(a + c_1 + c) - a - c$$

$$O(n \cdot \log n)$$

$$* \text{ Merge: } \text{limite} \Rightarrow b \frac{n}{2^i} + c$$

Figura 4.1: cálculo de tempo de execução do merge



```
1 def merge(v: list[int], s: int, m: int, e: int) -> list[int]:
2     p = s
3     q = m + 1
4     w: list[int] = []
5     for i in range(0, (e - s + 1)):
6         if (q > e) or ((p <= m) and (v[p] < v[q])):
7             w.append(v[p])
8             p += 1
9         else:
10            w.append(v[q])
11            q += 1
12
13    for i in range(0, (e - s + 1)):
14        v[(s + i)] = w[i]
15    return v
16
17
18 def mergeSort(v: list[int], s: int, e: int) -> list[int]:
19     if (s < e):
20         m: int = (s + e) // 2
21         mergeSort(v, s, m)
22         mergeSort(v, (m + 1), e)
23         merge(v, s, m, e)
24     return v
```

Figura 4.2: merge-sort adaptado com python

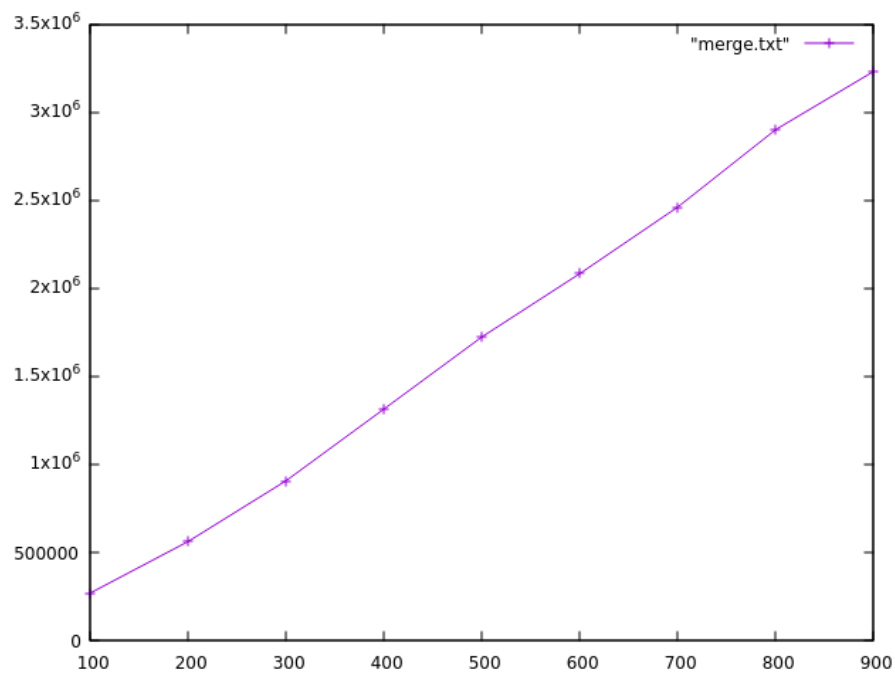


Figura 4.3: tempo de execução do merge-sort

## 5. Counting-sort

O Counting-sort é um algoritmo de ordenação eficiente projetado especificamente para ordenar uma lista de números inteiros não negativos. Ao contrário de outros algoritmos de ordenação que comparam os elementos entre si, esse algoritmo aproveita a propriedade dos números inteiros não negativos para realizar a ordenação de maneira linear. O Counting Sort é um algoritmo eficiente com uma complexidade de tempo de  $O(n + k)$  em todos os casos, onde "n" é o tamanho do array original e "k" é o maior valor na array. No entanto, para listas que contenham valores muito altos ele pode exigir muito espaço na memória para os arrays auxiliares.

### 5.1 O algoritmo

**algoritmo** counting-sort( $v, n$ )

$b \leftarrow \max(v)$

$c \leftarrow$  um array de tamanho  $(b + 1)$  preenchido com zeros

$w \leftarrow$  um array de tamanho  $n$  preenchido com zeros

**para**  $i \leftarrow 0$  **to**  $(n - 1)$  **faça**

$c[v[i]] \leftarrow c[v[i]] + 1$

**fim para**

**para**  $i \leftarrow 1$  **to**  $b$  **faça**

$c[i] \leftarrow c[i] + c[i - 1]$

**fim para**

**para**  $j \leftarrow (n - 1)$  **downto**  $0$  **faça**

$w[c[v[j]] - 1] \leftarrow v[j]$

$c[v[j]] \leftarrow c[v[j]] - 1$

**fim para**

**para**  $i \leftarrow 0$  **to**  $(n - 1)$  **faça**

$v[i] \leftarrow w[i]$

**fim para**

### 5.2 Complexidade

Esse algoritmo não possui melhor nem pior caso, sempre será linear, o tempo de execução do counting-sort só alterado pelo tamanho do vetor, afetará quantas vezes cada loop for será executada, então para vetores muito grandes e que contenham valores muito grandes, o tempo será cada vez pior. (visto em [5.1](#))

max  $\rightarrow$  busca orden em busca da máx = linear

$$T(n) = C_1 + an + b + C_2 + C_3 + nC_4 + (n-1)C_5 +$$

$$kC_6 + (k-1)C_7 + nC_8 + (n-1)(C_9 + C_{10}) + nC_{11} + (n-1)C_{12}$$

$$T(n) = n(C_4 + C_5 + C_8 + C_9 + C_{10} + C_{11} + C_{12}) + k(C_6 + C_7)$$

$$+ (C_1 + C_2 + C_3 - C_4 - C_7 - C_9 - C_{10} - C_{12}) + an + b$$

$$T(n) = nd + kg + h + an + b$$

$$T(n) = \underline{ng + kg + d}$$

linear de 2 variáveis  $\Rightarrow O(n+k)$

CS Digitalizado com CamScanner

**Figura 5.1:** cálculo de tempo de execução do counting sort

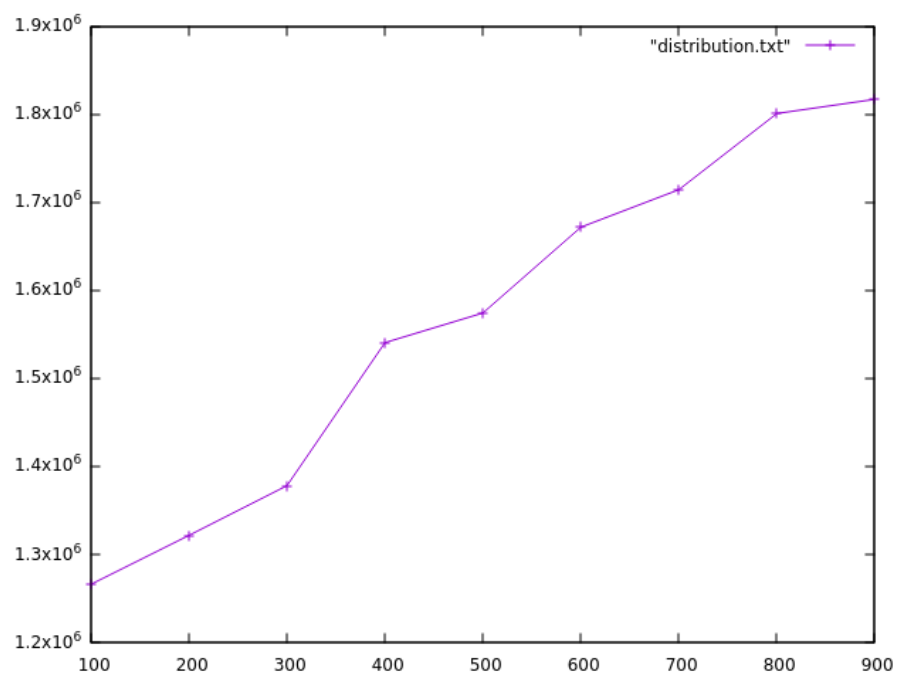
### 5.3 Implementação

O algoritmo foi implementado usando a linguagem python como visto em na figura [5.2](#), e seu tempo permanece linear podendo ser afetado também pelo tamanho do vetor auxiliar, explicando as variações vistas na figura [5.3](#).



```
1  def countingSort(v: list[int], n):
2      b = max(v)
3      c = [0] * (b + 1)
4      w = [0] * n
5
6      for i in range(n):
7          c[v[i]] += 1
8
9      for i in range(1, b + 1):
10         c[i] += c[i - 1]
11
12     for j in range(n - 1, -1, -1):
13         w[c[v[j]] - 1] = v[j]
14         c[v[j]] -= 1
15
16     for i in range(n):
17         v[i] = w[i]
```

**Figura 5.2:** counting-sort adaptado com python



**Figura 5.3:** tempo de execução do counting-sort

## 6. Conclusões

Como visto no gráfico da figura 6.1 o insertion-sort teve o pior resultado dentre todos, comparando o caso médio com vetores aleatórios, apesar de assim com o selection-sort ele ter complexidade quadrática, a estratégia do selection se torna um pouco mais eficiente vetores muito grandes. O merge-sort que tem a mesma complexidade que o quick se saiu levemente pior, já apesar de seguir estratégias parecidas o merge cria vetores auxiliares e tem de junta-los novamente, o que demanda um pouco a mais de tempo. Por ultimo distribution-counting-sort teve tempo e execução muito próximo ao merge e ao quick, porém deve se usado com cautela pois pode haver um uso excessivo de memória para vetores com valores que contenham valores muito altos.

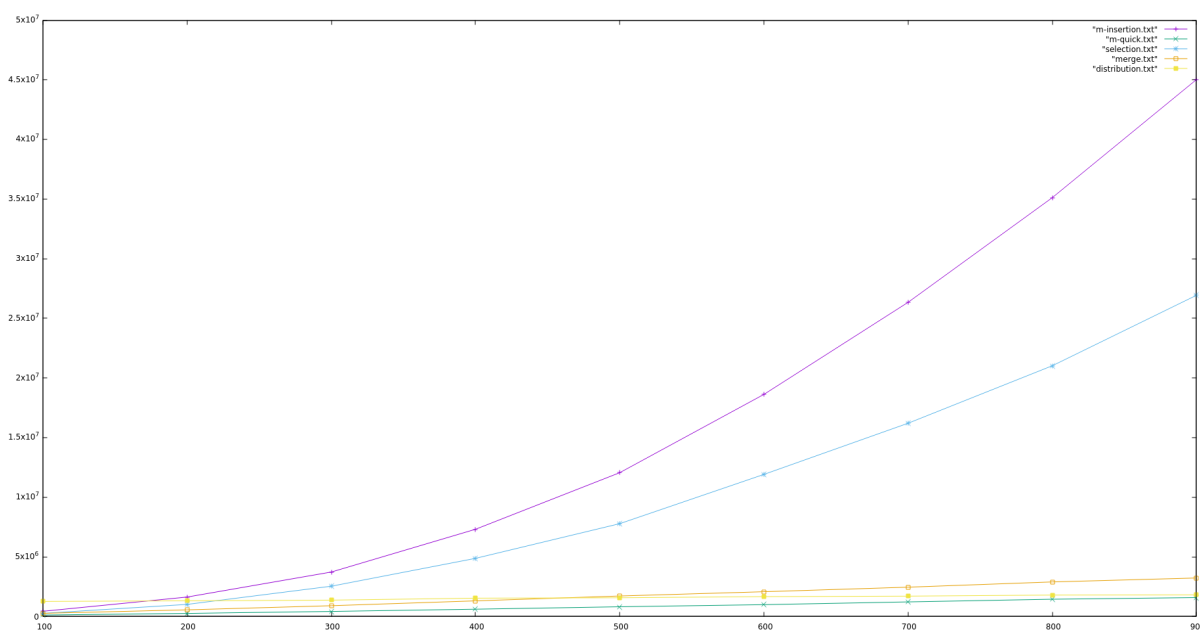
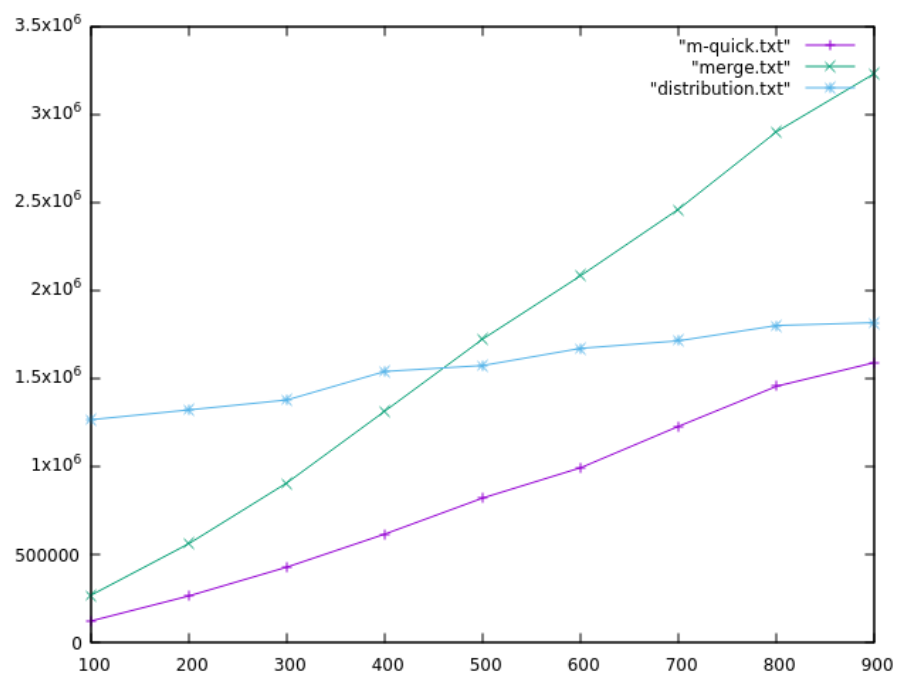


Figura 6.1: Comparação entre todos os algoritmos

### 6.1 Links Externos

Os algoritmos que foram usados para os testes contidos nesse documento podem ser encontrados no seguinte repositório remoto no github: [data-structure](#)



**Figura 6.2:** Comparação entre quick, merge e distribution