

# Análise Empírica de Algoritmos de Ordenação

Daniel Reis Arruda Sales  
Eliézer Alencar Moreira  
João Victor Ribeiro Santos

## Resumo

Neste trabalho, realizamos uma análise empírica de seis algoritmos de ordenação: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort. Implementamos cada algoritmo e realizamos testes com listas de diferentes tamanhos e distribuições, medindo o tempo de execução, o número de comparações e o número de trocas. Os resultados foram analisados e comparados com as expectativas teóricas.

## Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
1.1	Objetivo do Trabalho . . . . .	3
<b>2</b>	<b>Revisão Teórica</b>	<b>3</b>
2.1	Bubble Sort . . . . .	3
2.2	Selection Sort . . . . .	3
2.3	Insertion Sort . . . . .	4
2.4	Merge Sort . . . . .	4
2.5	Quick Sort . . . . .	4
2.6	Heap Sort . . . . .	5
2.7	Resumo Comparativo . . . . .	5
<b>3</b>	<b>Metodologia</b>	<b>5</b>
3.1	Ambiente de Teste . . . . .	5
3.2	Implementação dos Algoritmos . . . . .	6
3.3	Procedimento de Teste . . . . .	12
3.3.1	Testes de Tempo de Execução . . . . .	12
3.3.2	Testes de Comparação e Troca . . . . .	13

<b>4</b>	<b>Resultados</b>	<b>14</b>
4.1	Gráficos . . . . .	18
4.1.1	Gráficos dos tempos dos algoritmos . . . . .	18
4.1.2	Gráficos das comparações dos algoritmos . . . . .	21
4.1.3	Gráficos das trocas dos algoritmos . . . . .	24
<b>5</b>	<b>Discussão</b>	<b>27</b>
5.1	Análise dos Resultados . . . . .	27
5.2	Heap Sort . . . . .	27
5.3	Insertion Sort . . . . .	27
5.4	Merge Sort . . . . .	27
5.5	Selection Sort . . . . .	27
5.6	Bubble Sort . . . . .	27
5.7	Quick Sort Iterativo . . . . .	28
5.8	Comparação com Expectativas Teóricas . . . . .	28
5.9	Considerações sobre Implementação e Uso de Memória . . . . .	28
5.10	Limitações do Trabalho . . . . .	29
5.11	Sugestões para Estudos Futuros . . . . .	29
<b>6</b>	<b>Conclusão</b>	<b>29</b>
6.1	Recomendações . . . . .	29
<b>7</b>	<b>Referências</b>	<b>30</b>

## 1 Introdução

A ordenação de dados é uma operação fundamental na ciência da computação, crucial para a eficiência de diversos sistemas computacionais. Sua importância se manifesta na otimização de buscas, no processamento de grandes volumes de dados e na melhoria da experiência do usuário em interfaces digitais.

Diversos algoritmos de ordenação foram desenvolvidos ao longo dos anos, cada um com características próprias de desempenho e eficiência. Entre os mais conhecidos estão o Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort. Embora a complexidade teórica desses algoritmos seja bem estabelecida, seu desempenho prático pode variar significativamente dependendo do contexto de aplicação.

## 1.1 Objetivo do Trabalho

O objetivo deste trabalho é realizar uma análise empírica dos seguintes algoritmos de ordenação: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort.

## 2 Revisão Teórica

### 2.1 Bubble Sort

O *Bubble Sort* é um algoritmo simples que compara dois elementos adjacentes e os troca caso estejam na ordem errada. O processo é repetido até que a lista esteja completamente ordenada.

- **Complexidade:**
  - *Melhor caso:*  $O(n)$  (quando a lista já está ordenada)
  - *Caso médio:*  $O(n^2)$
  - *Pior caso:*  $O(n^2)$
- **In-place:** Sim. Utiliza apenas uma quantidade constante de memória adicional.
- **Estável:** Sim. Elementos iguais mantêm suas posições relativas.

### 2.2 Selection Sort

O *Selection Sort* percorre a lista à procura do menor elemento e o coloca na posição correta a cada iteração. Repete esse processo para cada elemento subsequente.

- **Complexidade:**
  - *Melhor caso:*  $O(n^2)$
  - *Caso médio:*  $O(n^2)$
  - *Pior caso:*  $O(n^2)$
- **In-place:** Sim. O algoritmo usa apenas memória adicional constante.
- **Estável:** Não. Elementos iguais podem mudar de posição.

## 2.3 Insertion Sort

O *Insertion Sort* constrói a lista ordenada um elemento de cada vez, inserindo o elemento da lista desordenada na posição correta na lista já ordenada.

- **Complexidade:**
  - *Melhor caso:*  $O(n)$  (quando a lista já está ordenada)
  - *Caso médio:*  $O(n^2)$
  - *Pior caso:*  $O(n^2)$
- **In-place:** Sim. O algoritmo usa memória adicional mínima.
- **Estável:** Sim. Elementos iguais mantêm suas posições relativas.

## 2.4 Merge Sort

O *Merge Sort* é um algoritmo baseado na técnica de divisão e conquista. Ele divide a lista em duas partes, ordena cada parte recursivamente e depois as combina.

- **Complexidade:**
  - *Melhor caso:*  $O(n \log n)$
  - *Caso médio:*  $O(n \log n)$
  - *Pior caso:*  $O(n \log n)$
- **In-place:** Não. O algoritmo necessita de memória adicional para armazenar sublistas temporárias.
- **Estável:** Sim. Elementos iguais mantêm suas posições relativas.

## 2.5 Quick Sort

O *Quick Sort* também usa a técnica de divisão e conquista. Ele escolhe um pivô, rearranja a lista para que os menores elementos fiquem à esquerda e os maiores à direita, e aplica o processo recursivamente.

- **Complexidade:**
  - *Melhor caso:*  $O(n \log n)$
  - *Caso médio:*  $O(n \log n)$

- *Pior caso:*  $O(n^2)$  (quando o pivô é mal escolhido)
- **In-place:** Sim. Apenas pequenas quantidades de memória adicional são utilizadas.
- **Estável:** Não. Elementos iguais podem ser reposicionados.

## 2.6 Heap Sort

O *Heap Sort* transforma a lista em uma estrutura de heap (máximo ou mínimo) e então extrai repetidamente o maior ou menor elemento para formar a lista ordenada.

- **Complexidade:**
  - *Melhor caso:*  $O(n \log n)$
  - *Caso médio:*  $O(n \log n)$
  - *Pior caso:*  $O(n \log n)$
- **In-place:** Sim. O algoritmo utiliza memória adicional constante.
- **Estável:** Não. A estabilidade não é garantida, pois a estrutura de heap pode mudar a posição de elementos iguais.

## 2.7 Resumo Comparativo

Algoritmo	Melhor Caso	Pior Caso	In-place	Estável
Bubble Sort	$O(n)$	$O(n^2)$	Sim	Sim
Selection Sort	$O(n^2)$	$O(n^2)$	Sim	Não
Insertion Sort	$O(n)$	$O(n^2)$	Sim	Sim
Merge Sort	$O(n \log n)$	$O(n \log n)$	Não	Sim
Quick Sort	$O(n \log n)$	$O(n^2)$	Sim	Não
Heap Sort	$O(n \log n)$	$O(n \log n)$	Sim	Não

Tabela 1: Comparação dos Algoritmos de Ordenação

## 3 Metodologia

### 3.1 Ambiente de Teste

Os testes de desempenho dos algoritmos de ordenação foram realizados em um ambiente de hardware e software com as seguintes especificações:

- **Hardware:**

- Processador: AMD Ryzen 5 5500U @ 2.10 GHz
- Memória RAM: 16 GB DDR4
- Disco: SSD de 512 GB
- Modelo do Notebook: Dell Inspiron 15 3525

- **Software:**

- Sistema Operacional: Ubuntu 22.04 (via WSL2)
- Linguagem de Programação: Python 3.10
- Ambiente de Desenvolvimento: Visual Studio Code
- Biblioteca de Medição de Tempo: `timeit`
- Biblioteca para Manipulação de Arrays: `numpy`

## 3.2 Implementação dos Algoritmos

Implementações dos algoritmos utilizados para ordenação, representados por imagens contendo o código de cada um.



```

1  comparacoes = trocas = 0
2
3  def bubblesort(lista):
4      global comparacoes
5      global trocas
6      troca = True
7      while troca:
8          troca = False
9          for i in range(len(lista) - 1):
10             comparacoes += 1
11             if lista[i] > lista[i+1]:
12                 trocas += 1
13                 trocar(lista, i)
14                 troca = True
15
16     return [comparacoes, trocas]
17
18 def trocar(listaa, indice):
19     listaa[indice], listaa[indice + 1] = listaa[indice + 1] , listaa[indice]
20
21
22 if __name__ == '__main__':
23     a = [7,8,5,9,6,3,2,4,1,0]
24
25     b = [35,64,2]
26
27     print(bubblesort(a))
28
29     print(bubblesort(b))

```

Figura 1: Implementação do Algoritmo Bubble Sort

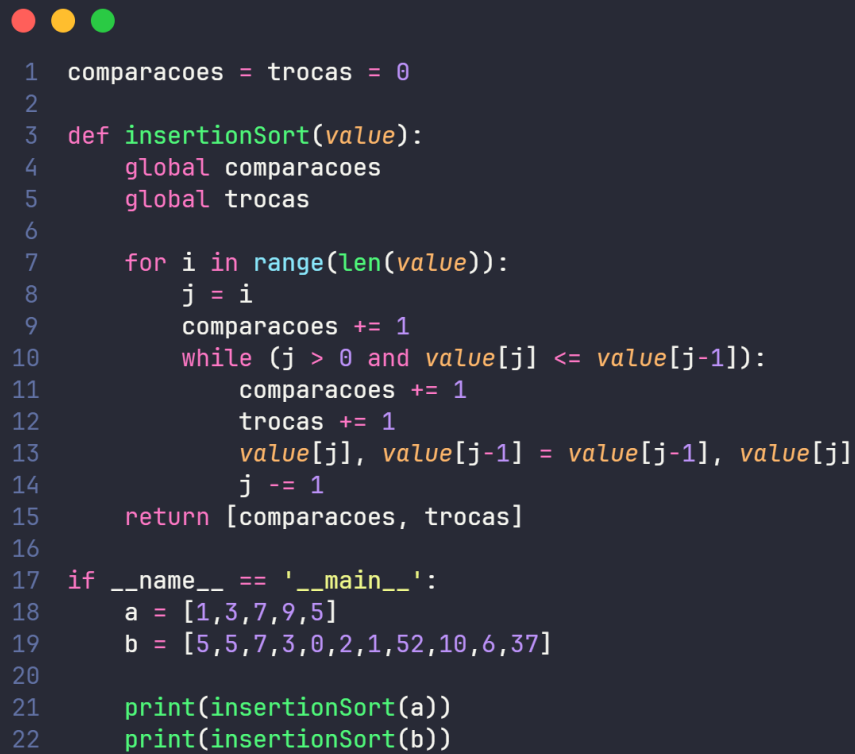
```

1  comparacoes = trocas = 0
2
3  def heapify(arr, n, i):
4      global comparacoes
5      global trocas
6      parent = i
7      left = 2 * i + 1
8      right = 2 * i + 2
9
10     comparacoes += 1
11     # Se o filho na esquerda é maior
12     if left < n and arr[left] > arr[parent]:
13         parent = left
14
15     comparacoes += 1
16     # Se o filho na direita é maior
17     if right < n and arr[right] > arr[parent]:
18         parent = right
19
20     if parent != i:
21         trocas += 1
22         arr[i], arr[parent] = arr[parent], arr[i] # Troca
23         heapify(arr, n, parent)
24
25  def heap_sort(arr):
26     global comparacoes
27     global trocas
28     n = len(arr)
29
30     # Constrói o Max Heap
31     for i in range(n // 2 - 1, -1, -1):
32         heapify(arr, n, i)
33
34     for i in range(n-1, 0, -1):
35         trocas += 1
36         # Move a raiz atual para o final
37         arr[i], arr[0] = arr[0], arr[i]
38
39         heapify(arr, i, 0)
40
41     return [comparacoes, trocas]

```

Figura 2: Implementação do Algoritmo Heap Sort





```
1 comparacoes = trocas = 0
2
3 def insertionSort(value):
4     global comparacoes
5     global trocas
6
7     for i in range(len(value)):
8         j = i
9         comparacoes += 1
10        while (j > 0 and value[j] <= value[j-1]):
11            comparacoes += 1
12            trocas += 1
13            value[j], value[j-1] = value[j-1], value[j]
14            j -= 1
15        return [comparacoes, trocas]
16
17 if __name__ == '__main__':
18     a = [1,3,7,9,5]
19     b = [5,5,7,3,0,2,1,52,10,6,37]
20
21     print(insertionSort(a))
22     print(insertionSort(b))
```

Figura 3: Implementação do Algoritmo Insertion Sort

```

1  comparacoes = trocas = 0
2
3  def mergeSort(array):
4
5      global comparacoes
6      global trocas
7
8      if len(array) > 1:
9          mid = len(array)//2
10
11         L = array[:mid]
12         R = array[mid:]
13         mergeSort(L)
14         mergeSort(R)
15
16         i = j = k = 0
17
18         while i < len(L) and j < len(R):
19             comparacoes += 1
20             if L[i] < R[j]:
21                 array[k] = L[i]
22                 i += 1
23             else:
24                 array[k] = R[j]
25                 j += 1
26
27             k += 1
28
29         while i < len(L):
30             array[k] = L[i]
31             i += 1
32             k += 1
33
34         while j < len(R):
35             array[k] = R[j]
36             j += 1
37             k += 1
38
39         return [comparacoes, trocas]
40
41 def printList(array):
42     for i in range(len(array)):
43         print(array[i], end=" ")
44     print()
45
46
47 if __name__ == '__main__':
48     a = [89,76,12,45,23]
49     printList(a)
50     mergeSort(a)
51     printList(a)

```

Figura 4: Implementação do Algoritmo Merge Sort

```

1  comparacoes = trocas = 0
2
3  def iterative_quickSort(arr):
4      global comparacoes
5      global trocas
6      # Criação de uma pilha simulando o processo recursivo
7      stack = [(0, len(arr) - 1)]
8
9      # Enquanto houver segmentos na pilha para ordenar
10     while stack:
11         start, end = stack.pop()
12
13         comparacoes += 1
14         if start >= end:
15             continue
16
17         # Particionar o array e obter o índice do pivô
18         pivot_index = partition(arr, start, end)
19
20         # Empilhar as duas partes para continuar ordenando
21         stack.append((start, pivot_index - 1)) # Parte esquerda do pivô
22         stack.append((pivot_index + 1, end))   # Parte direita do pivô
23
24     return [comparacoes, trocas]
25
26 def partition(arr, start, end):
27     global comparacoes
28     global trocas
29     # Pivô escolhido como o último elemento
30     pivot = arr[end]
31     i = start - 1
32
33     # Organiza os elementos em torno do pivô
34     for j in range(start, end):
35         comparacoes += 1
36         if arr[j] < pivot:
37             i += 1
38             trocas += 1
39             arr[i], arr[j] = arr[j], arr[i]
40
41     # Coloca o pivô na posição correta
42     trocas += 1
43     arr[i + 1], arr[end] = arr[end], arr[i + 1]
44
45     # Retorna o índice do pivô
46     return i + 1

```

Figura 5: Implementação do Algoritmo Quick Sort



```

1  comparacoes = trocas = 0
2
3  def selectionsort(lista):
4      global comparacoes
5      global trocas
6
7      for i in range(len(lista) - 1):
8          imenor = i
9          for f in range(i+1, len(lista)):
10             comparacoes += 1
11             if lista[f] < lista[imenor]:
12                 imenor = f
13
14             trocas += 1
15             lista[i], lista[imenor] = lista[imenor], lista[i]
16     return [comparacoes, trocas]
17
18 if __name__ == '__main__':
19
20     a = [9,5,6,7,3,1]
21     b = [456,789,123]
22
23     print(selectionsort(a))
24     print(selectionsort(b))

```

Figura 6: Implementação do Algoritmo Selection Sort

### 3.3 Procedimento de Teste

#### 3.3.1 Testes de Tempo de Execução

Para medir o tempo de execução dos algoritmos, utilizamos o seguinte código:

```

1 from algoritmos import *
2 import timeit, pandas as pd
3 import numpy as np
4
5 algoritmos = ["iterative_quickSort", "heap_sort", "insertionSort", "mergeSort", "selectionsort", "bubblesort",]
6 inputs_size = [1_000, 10_000, 50_000, 100_000]
7 inputs_ordenados_asc = []
8 inputs_ordenados_dec = []
9 inputs_desordenados = []
10
11 for size in inputs_size:
12     inputs_ordenados_asc.append(np.arange(size))
13     inputs_ordenados_dec.append(np.arange(size, -1, -1))
14     inputs_desordenados.append(np.random.randint(size, size=size))
15
16 resultados = {}
17 for algoritmo in algoritmos:
18     resultados[algoritmo] = []
19     for j, size in enumerate(inputs_size):
20         # Input ordenado crescente
21         stmt = f"{algoritmo}(np.copy(inputs_ordenados_asc[{j}]))"
22         time = timeit.timeit(stmt=stmt, globals=globals(), number=1)
23         result_asc = (size, 'Crescente', time)
24         print(f"{algoritmo}: {result_asc}")
25
26         # Input ordenado decrescente
27         stmt = f"{algoritmo}(np.copy(inputs_ordenados_dec[{j}]))"
28         time = timeit.timeit(stmt=stmt, globals=globals(), number=1)
29         result_dec = (size, 'Decrescente', time)
30         print(f"{algoritmo}: {result_dec}")
31
32         # Input desordenado
33         stmt = f"{algoritmo}(np.copy(inputs_desordenados[{j}]))"
34         time = timeit.timeit(stmt=stmt, globals=globals(), number=1)
35         result_desordenado = (size, 'Desordenado', time)
36         print(f"{algoritmo}: {result_desordenado}")
37
38         result = [result_asc, result_dec, result_desordenado]
39         resultados[algoritmo].append(result)
40
41 pd.DataFrame(resultados).to_json('tempos.json')

```

Figura 7: Código para medir o tempo de execução dos algoritmos

O código realiza os seguintes passos:

- Define uma lista de algoritmos de ordenação.
- Gera entradas de diferentes tamanhos, incluindo listas ordenadas e desordenadas.
- Mede o tempo de execução de cada algoritmo para as diferentes entradas e armazena os resultados em um arquivo JSON.

### 3.3.2 Testes de Comparação e Troca

O seguinte código é usado para medir o número de comparações e trocas realizadas por cada algoritmo:

```

1 from algoritmos import *
2 import pandas as pd
3 import numpy as np
4
5 algoritmos = [iterative_quickSort, heap_sort, insertionSort, mergeSort, selectionsort, bubblesort]
6 inputs_size = [1_000, 10_000, 50_000, 100_000]
7 inputs_ordenados_asc = []
8 inputs_ordenados_dec = []
9 inputs_desordenados = []
10
11 for size in inputs_size:
12     inputs_ordenados_asc.append(np.arange(size))
13     inputs_ordenados_dec.append(np.arange(size, -1, -1))
14     inputs_desordenados.append(np.random.randint(size, size=size))
15
16 resultados = {}
17 for algoritmo in algoritmos:
18     resultados[algoritmo] = []
19     for j, size in enumerate(inputs_size):
20         # Input ordenado crescente
21         comparacoes, trocas = algoritmo(np.copy(inputs_ordenados_asc[j]))
22         result_asc = ("Crescente", comparacoes, trocas)
23         print(f"{algoritmo.__name__} {size}: {result_asc}")
24
25         # Input ordenado decrescente
26         comparacoes, trocas = algoritmo(np.copy(inputs_ordenados_dec[j]))
27         result_dec = ("Decrescente", comparacoes, trocas)
28         print(f"{algoritmo.__name__} {size}: {result_dec}")
29
30         # Input desordenado
31         comparacoes, trocas = algoritmo(np.copy(inputs_desordenados[j]))
32         result_desordenado = ("Desordenado", comparacoes, trocas)
33         print(f"{algoritmo.__name__} {size}: {result_desordenado}")
34
35         result = [f"Tamanho: {size}", result_asc, result_dec, result_desordenado]
36         resultados[algoritmo].append(result)
37
38 pd.DataFrame(resultados).to_json('comparacoes_trocas.json')
39

```

Figura 8: Código para medir comparações e trocas

Esse código segue uma abordagem semelhante ao anterior, mas foca em contabilizar quantas comparações e trocas cada algoritmo realiza para diferentes entradas.

## 4 Resultados

Os resultados dos tempos de execução em segundos para os algoritmos testados são apresentados na Tabela 2. Os dados foram coletados para diferentes tamanhos de entrada e para diferentes ordens de dados (crescente, decrescente e desordenado).

Tabela 2: Tempos de Execução dos Algoritmos

Algoritmo	Tamanho	Crescente	Decrescente	Desordenado
Heap Sort	1000	0.00888	0.00739	0.00893
	10000	0.12215	0.10439	0.10909
	50000	0.68834	0.63021	0.65801
	100000	1.40688	1.32619	1.38904
Insertion Sort	1000	0.00025	0.29353	0.14528
	10000	0.00224	30.00210	15.07364
	50000	0.01121	753.18931	377.70774
	100000	0.02262	3039.50276	1534.32284
Merge Sort	1000	0.00739	0.00491	0.00463
	10000	0.05255	0.05465	0.06067
	50000	0.29973	0.31259	0.34199
	100000	0.63286	0.65795	0.72052
Selection Sort	1000	0.09591	0.09681	0.09602
	10000	9.86280	9.76944	9.53009
	50000	234.30547	242.74690	240.57817
	100000	954.51418	966.17692	910.94735
Bubble Sort	1000	0.00074	0.39445	0.26839
	10000	0.00247	39.86541	28.90482
	50000	0.01166	988.36041	725.55994
	100000	0.02512	3958.45200	2976.68094
Quick Sort	1000	0.27202	0.17743	0.00435
	10000	28.09641	17.74824	0.06338
	50000	719.73262	477.51905	0.37427
	100000	3062.80170	1978.09924	0.91525

Tabela 3: Comparações dos Algoritmos de Ordenação

Algoritmo	Tamanho	Crescente	Decrescente	Desordenado
Iterative Quick Sort	1000	501499	1004000	1015788
	10000	51030787	101055788	101238789
	50000	1351313788	2601438789	2602425535
	100000	7602575534	12602825535	12605246287
Heap Sort	1000	20416	38176	57274
	10000	331186	575510	834058
	50000	2430666	3879190	5404580
	100000	8806288	11901360	15152122
Insertion Sort	1000	1000	502501	754505
	10000	764505	50779506	75481665
	50000	75531665	1325606666	1947363851
	100000	1947463851	6947613852	9453985336
Merge Sort	1000	4932	9981	17200
	10000	81808	150826	250029
	50000	632541	1034504	1602780
	100000	2417804	3271720	4489115
Selection Sort	1000	499500	1000000	1499500
	10000	51494500	101499500	151494500
	50000	1401469500	2651494500	3901469500
	100000	8901419500	13901469500	18901419500
Bubble Sort	1000	999	1001999	1959041
	10000	1969040	101979040	201189118
	50000	201239117	2701289117	5186289416
	100000	5186389415	15186489415	25151889760



Tabela 4: Trocas dos Algoritmos de Ordenação

Algoritmo	Tamanho	Crescente	Decrescente	Desordenado
Iterative Quick Sort	1000	500499	751499	757929
	10000	50762928	75772928	75879187
	50000	1325904186	1950954186	1951434597
	100000	6951484596	9451584596	9452717168
Heap Sort	1000	9708	18088	27137
	10000	159093	276255	400529
	50000	1173833	1873095	2610790
	100000	4261644	5759180	7334561
Insertion Sort	1000	0	500500	751504
	10000	751504	50756504	75448663
	50000	75448663	1325473663	1947180848
	100000	1947180848	6947230848	9453502332
Merge Sort	1000	0	0	0
	10000	0	0	0
	50000	0	0	0
	100000	0	0	0
Selection Sort	1000	999	1999	2998
	10000	12997	22997	32996
	50000	82995	132995	182994
	100000	282993	382993	482992
Bubble Sort	1000	0	500500	750993
	10000	750993	50755993	75443228
	50000	75443228	1325468228	1947150619
	100000	1947150619	6947200619	9453421814

## 4.1 Gráficos

### 4.1.1 Gráficos dos tempos dos algoritmos

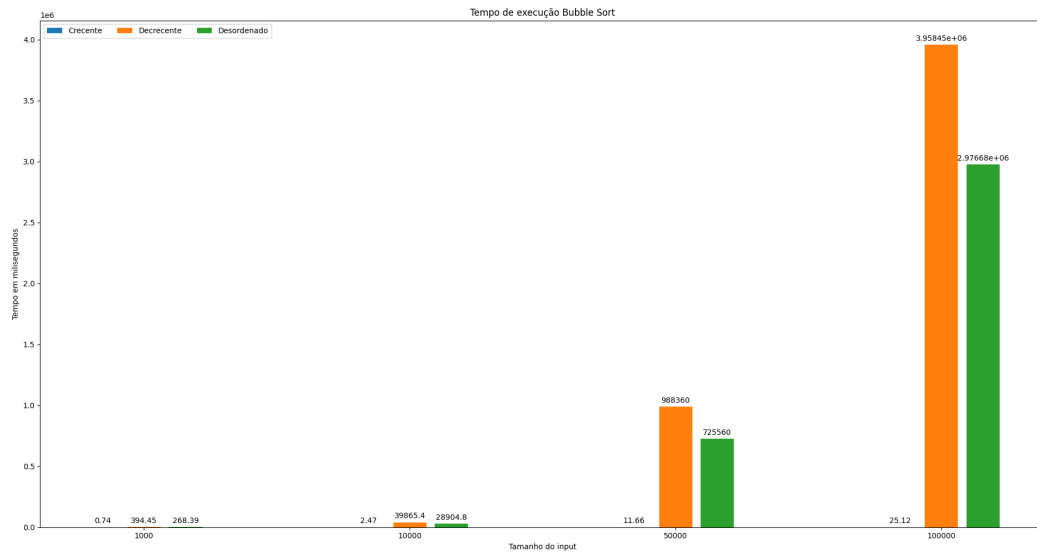


Figura 9: Tempo Bubble Sort

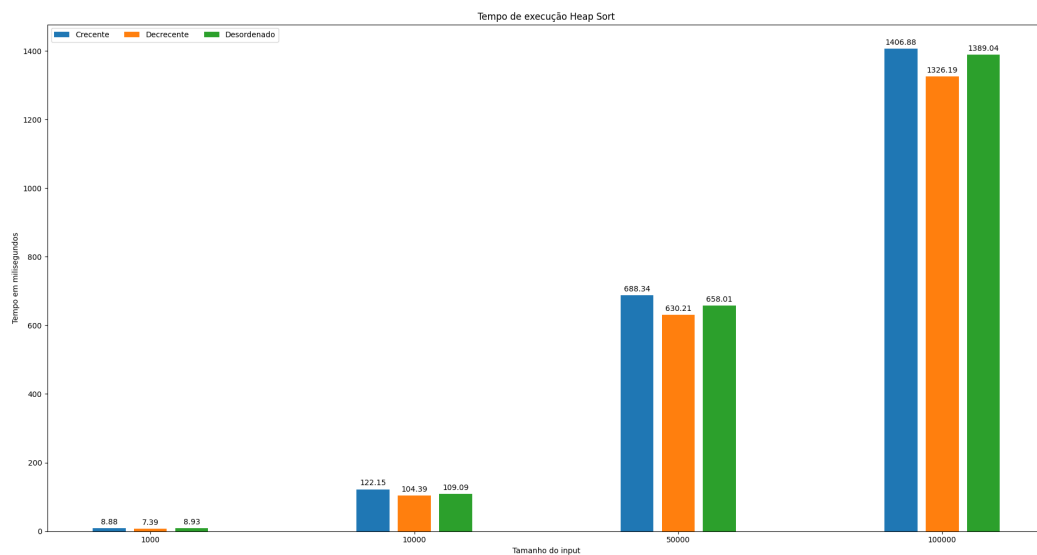


Figura 10: Tempo HeapSort

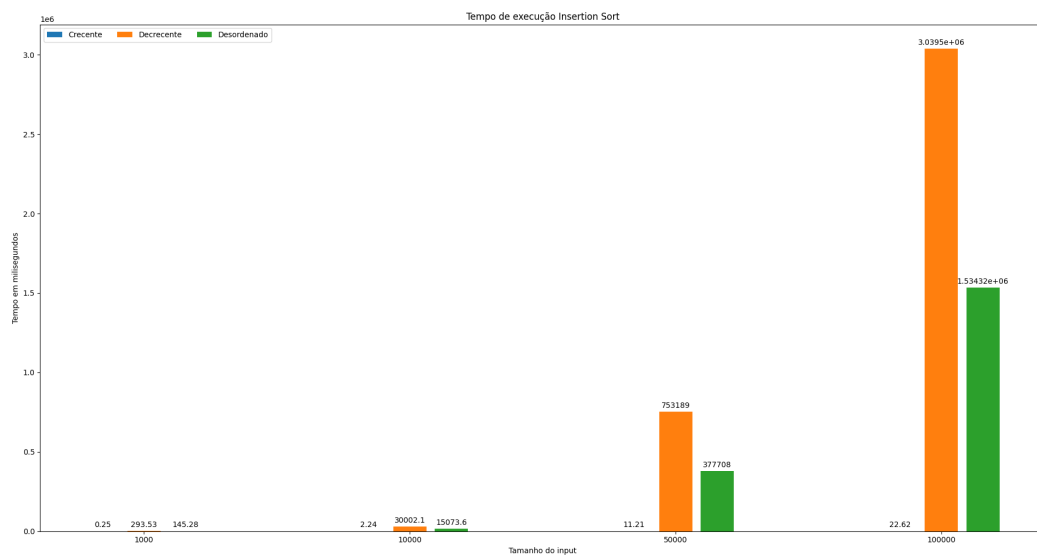


Figura 11: Tempo Insertion Sort

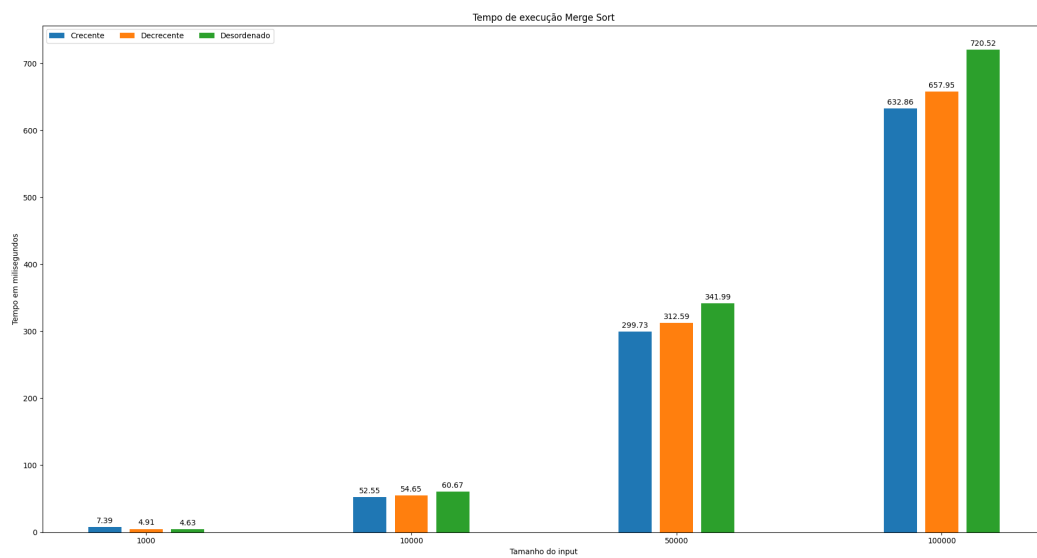


Figura 12: Tempo Merge Sort

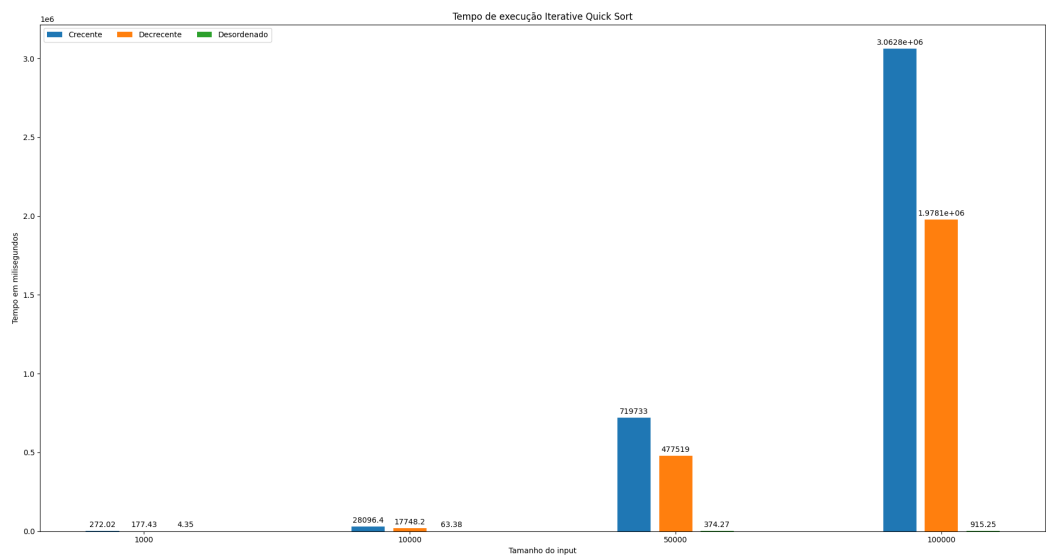


Figura 13: Tempo Quick Sort

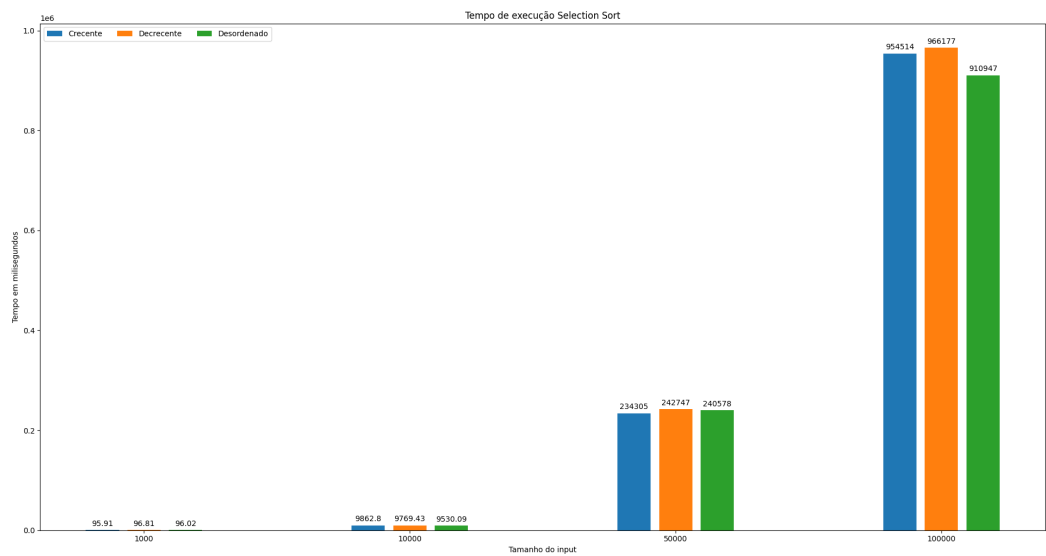


Figura 14: Tempo Selection Sort

## 4.1.2 Gráficos das comparações dos algoritmos

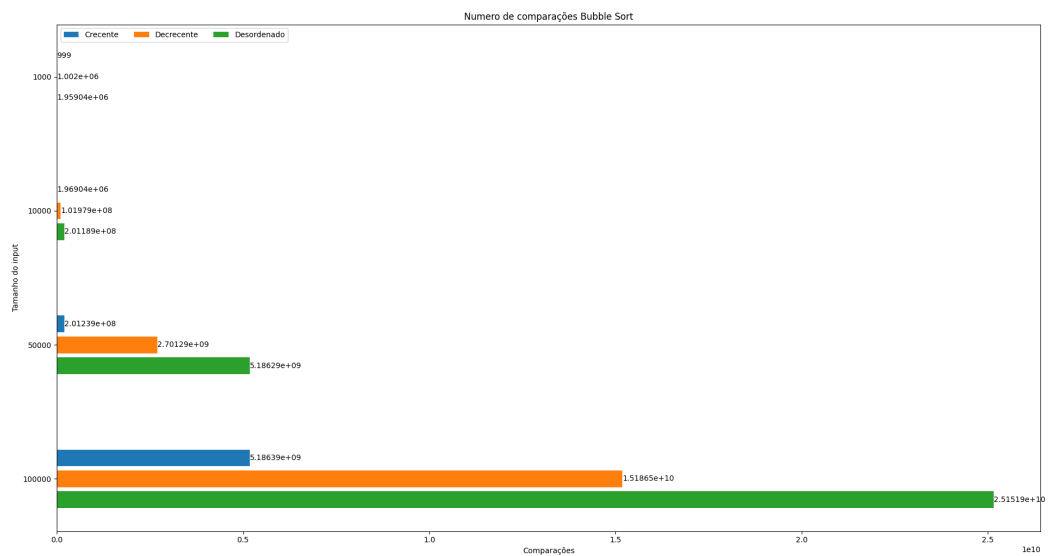


Figura 15: Comparações Bubble Sort

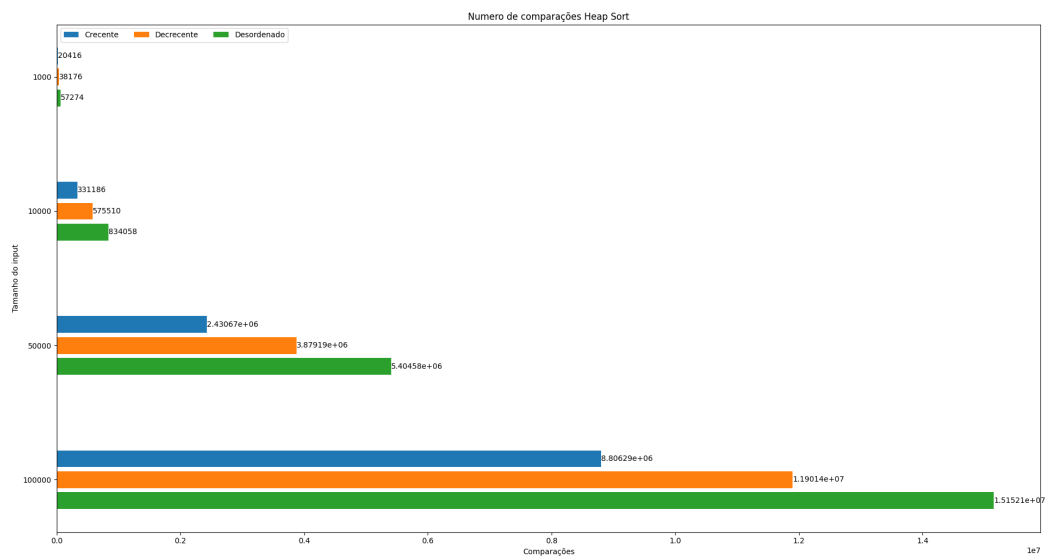


Figura 16: Comparações HeapSort

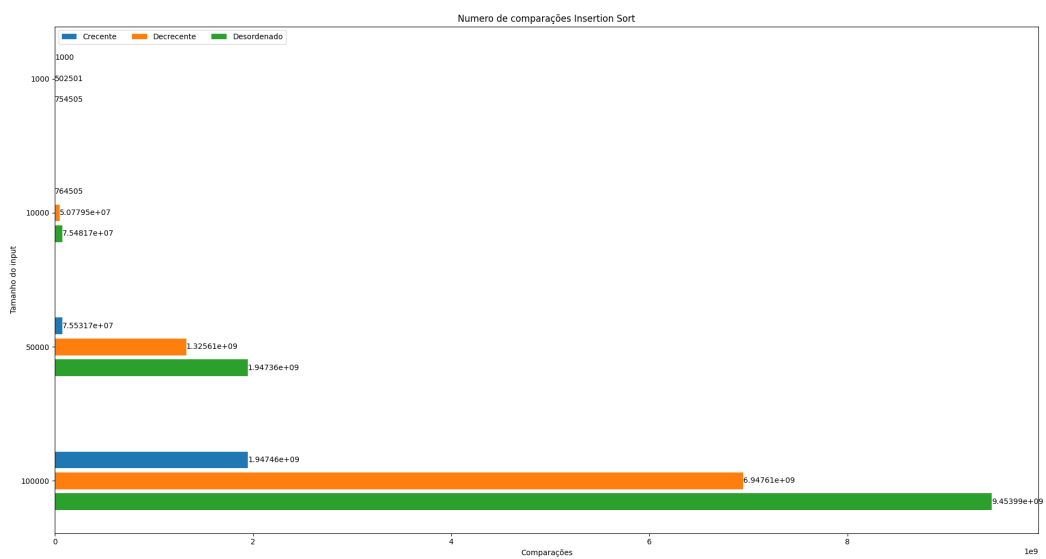


Figura 17: Comparações Insertion Sort

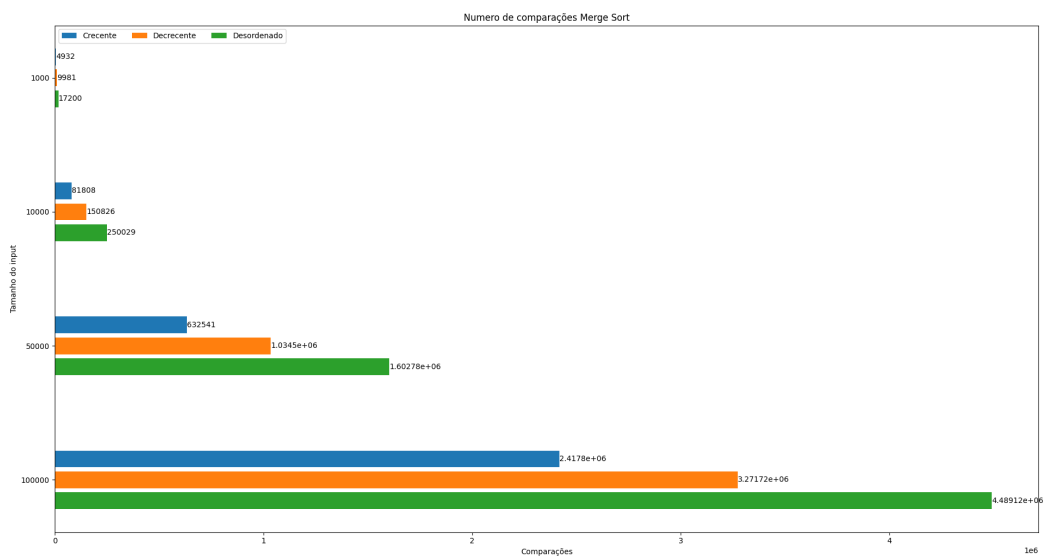


Figura 18: Comparações Merge Sort

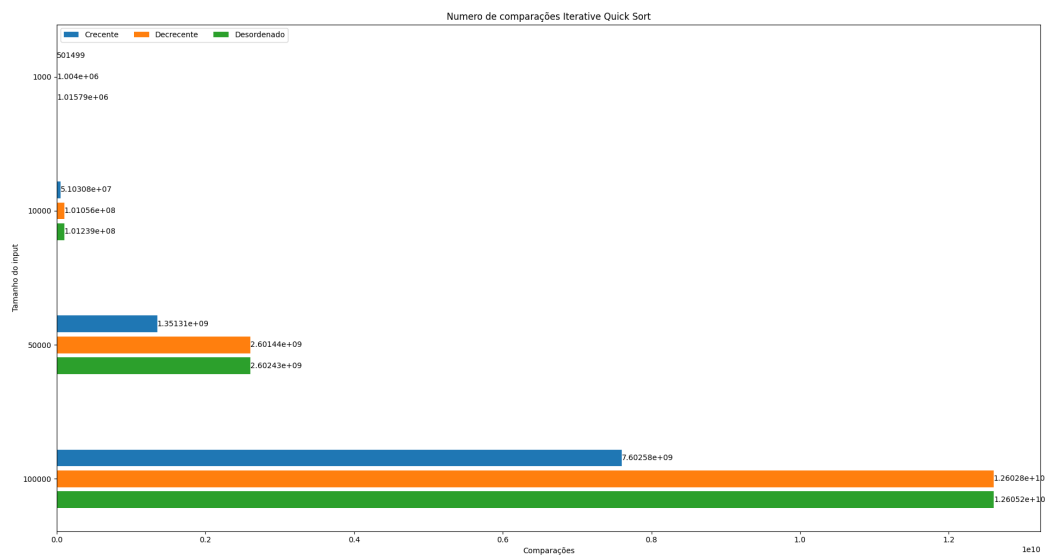


Figura 19: Comparações Quick Sort

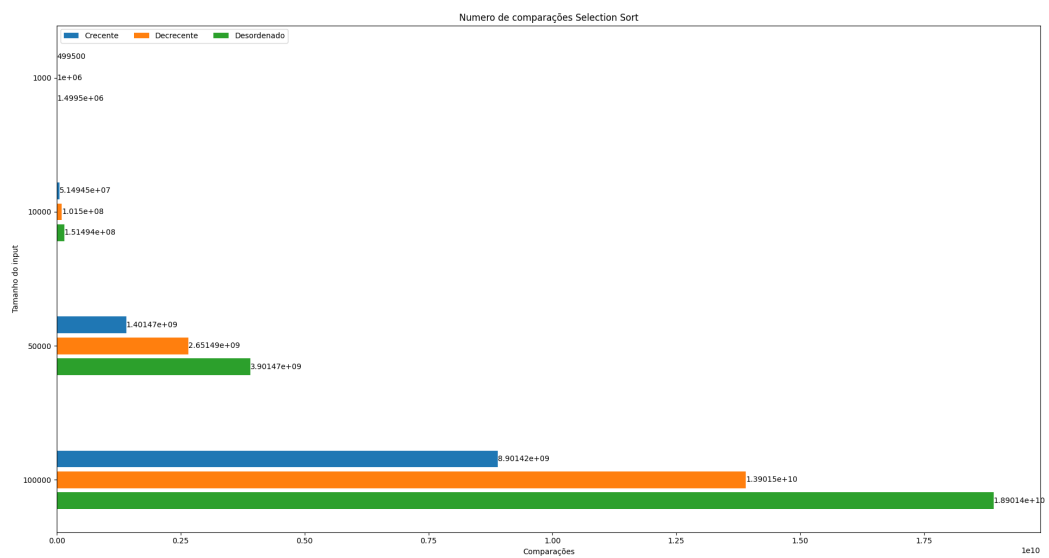


Figura 20: Comparações Selection Sort

### 4.1.3 Gráficos das trocas dos algoritmos

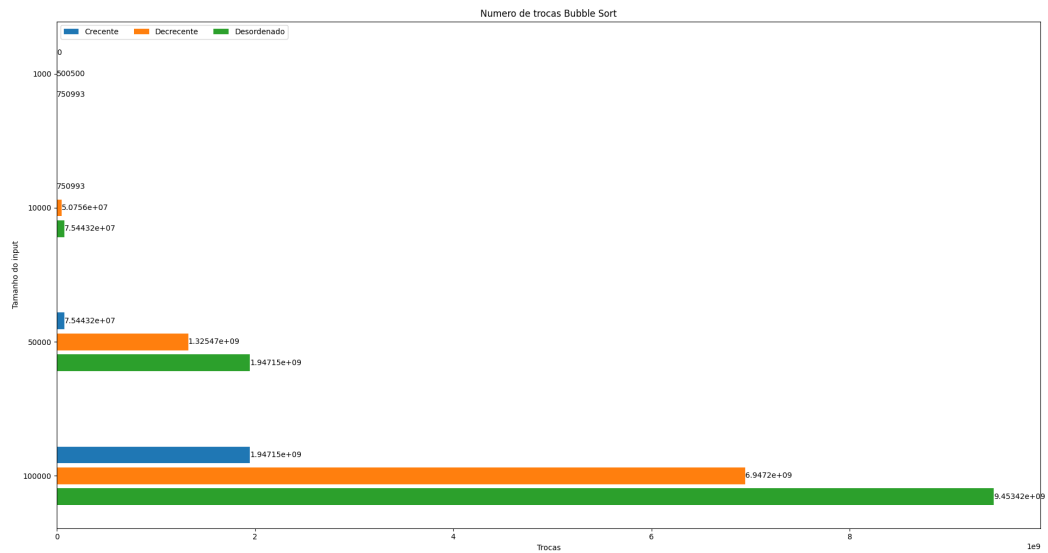


Figura 21: Trocas Bubble Sort

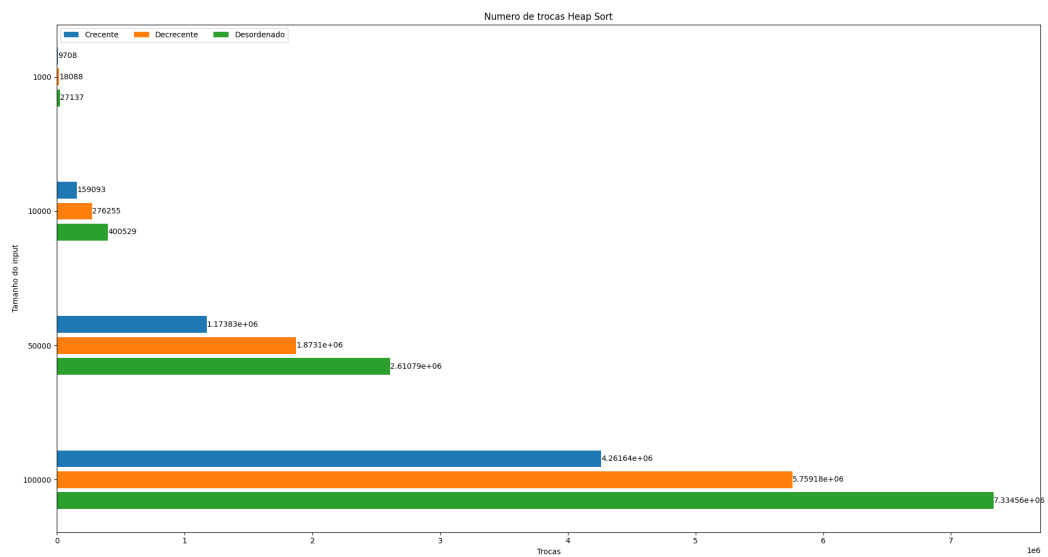


Figura 22: Trocas HeapSort



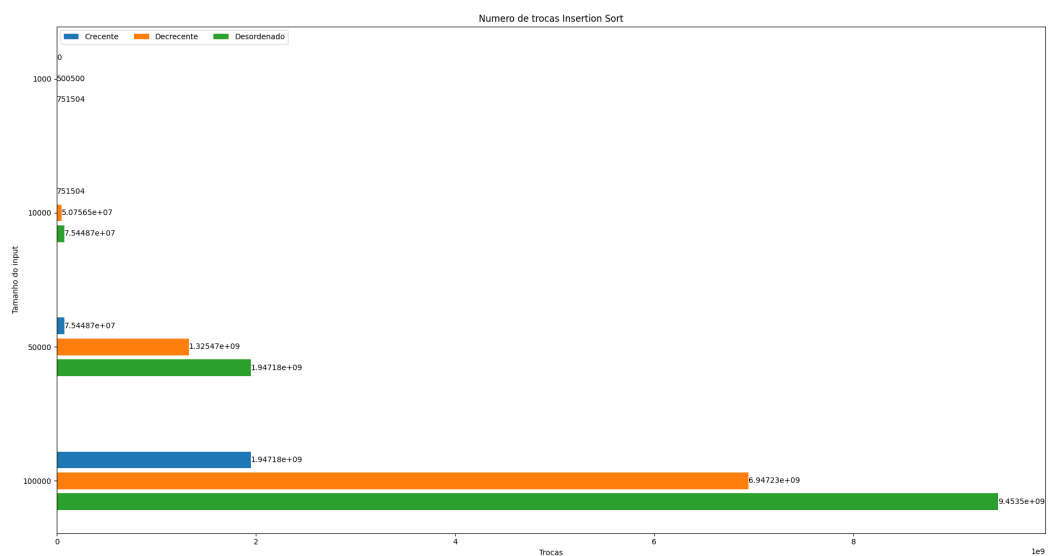


Figura 23: Trocas Insertion Sort

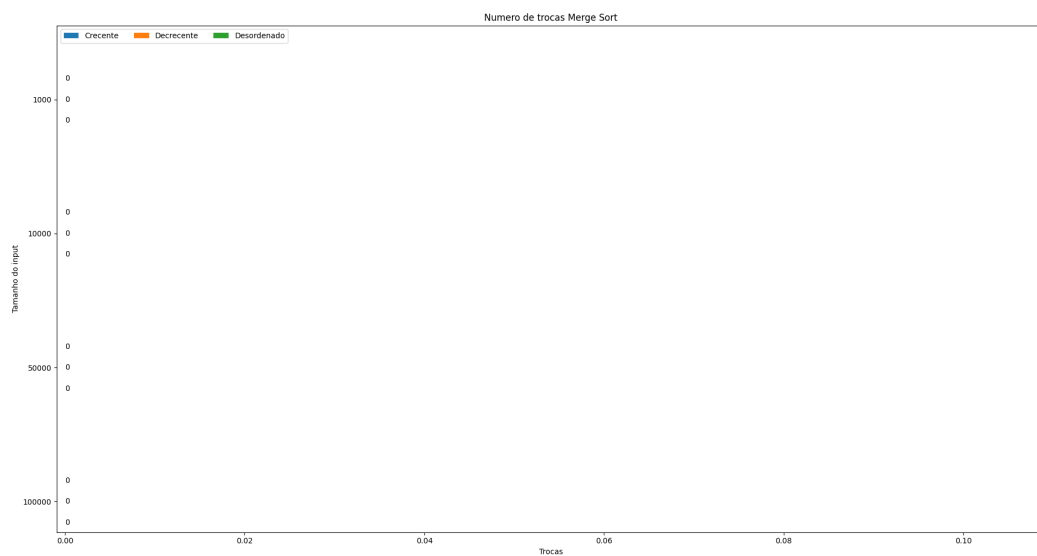


Figura 24: Trocas Merge Sort

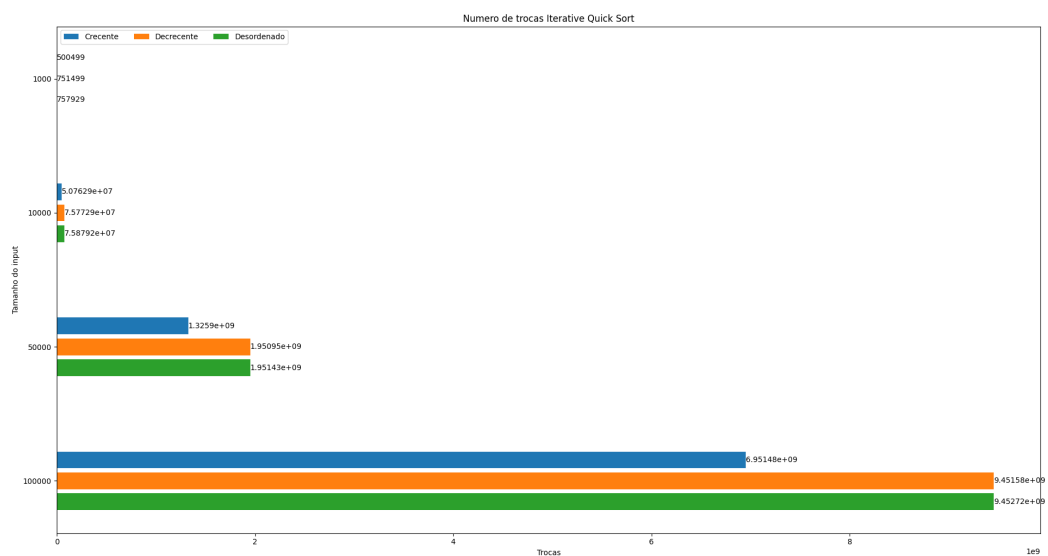


Figura 25: Trocas Quick Sort

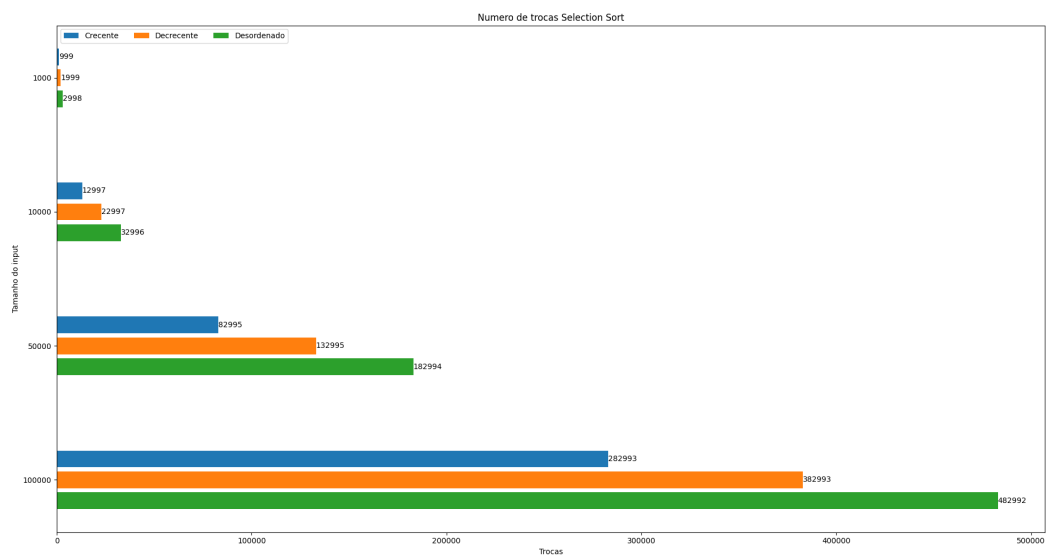


Figura 26: Trocas Selection Sort

## 5 Discussão

### 5.1 Análise dos Resultados

### 5.2 Heap Sort

O Heap Sort apresenta um desempenho consistente, com tempos de execução que aumentam gradualmente com o tamanho dos dados. A diferença de tempo entre as entradas crescentes, decrescentes e desordenadas é mínima, o que é esperado, pois o Heap Sort tem um desempenho médio e pior de  $O(n \log n)$ , independentemente da ordem de entrada.

### 5.3 Insertion Sort

O Insertion Sort apresenta tempos de execução muito baixos para entradas crescentes, que são esperados, pois o algoritmo funciona bem em listas já ordenadas. No entanto, para entradas decrescentes e desordenadas, os tempos de execução aumentam drasticamente, especialmente para tamanhos maiores, refletindo sua complexidade  $O(n^2)$  no pior caso.

### 5.4 Merge Sort

O Merge Sort apresenta um desempenho estável e consistente. Os tempos de execução para diferentes ordens são bastante próximos, o que confirma a expectativa teórica de que o Merge Sort opera em  $O(n \log n)$  em todos os casos. Isso o torna uma escolha sólida para listas de tamanhos maiores.

### 5.5 Selection Sort

Os resultados do Selection Sort confirmam a expectativa teórica de que ele é ineficiente, com tempos de execução elevados que aumentam rapidamente com o tamanho dos dados. Sua complexidade é  $O(n^2)$ , o que se reflete nos altos tempos de execução observados, especialmente para entradas decrescentes e desordenadas.

### 5.6 Bubble Sort

O Bubble Sort, assim como o Selection Sort, demonstra um desempenho insatisfatório para entradas maiores. A complexidade  $O(n^2)$  é evidente em seus tempos de execução, especialmente para listas em ordem decrescente, onde o tempo se eleva a níveis preocupantes.

## 5.7 Quick Sort Iterativo

O Quick Sort Iterativo mostra um desempenho bastante eficiente para listas desordenadas, com tempos baixos comparados aos outros algoritmos. No entanto, para entradas crescentes e decrescentes, os tempos aumentam significativamente, especialmente em listas maiores. A complexidade média do Quick Sort é  $O(n \log n)$ , mas sua complexidade no pior caso pode ser  $O(n^2)$ , o que é visível nas diferenças de desempenho.

## 5.8 Comparação com Expectativas Teóricas

Os resultados obtidos confirmam amplamente as expectativas teóricas em relação aos algoritmos de ordenação. Algoritmos com complexidade  $O(n^2)$ , como Bubble Sort, Insertion Sort e Selection Sort, apresentaram desempenhos inferiores, especialmente em entradas maiores e desordenadas. Em contraste, os algoritmos com complexidade  $O(n \log n)$ , como Merge Sort, Quick Sort e Heap Sort, mantiveram um desempenho estável e eficiente, confirmando as previsões teóricas.

Uma observação interessante foi o desempenho do Quick Sort Iterativo. Embora seja eficiente na maioria dos casos, ele demonstrou vulnerabilidade em entradas ordenadas de forma crescente ou decrescente, confirmando a possibilidade de atingir a complexidade  $O(n^2)$  no pior caso. No entanto, as melhorias oferecidas por sua implementação iterativa se mostraram vantajosas em relação ao uso de memória.

## 5.9 Considerações sobre Implementação e Uso de Memória

Durante a implementação dos algoritmos, verificamos que o Merge Sort, apesar de eficiente em termos de tempo, exige um uso de memória adicional devido à necessidade de criar arrays temporários durante o processo de divisão e fusão dos dados. O Quick Sort Iterativo, por sua vez, mostrou-se mais eficiente em termos de uso de memória, eliminando a necessidade de recursão profunda.

Algoritmos como Heap Sort, que operam de forma in-place, demonstraram um uso mais otimizado da memória, o que o torna uma opção interessante quando o consumo de memória é uma limitação, embora seu tempo de execução seja um pouco maior em comparação ao Quick Sort em cenários ideais.

## 5.10 Limitações do Trabalho

Uma limitação importante deste trabalho foi o foco exclusivo no tempo de execução e no uso de memória. Aspectos como a facilidade de implementação e a adaptabilidade dos algoritmos a diferentes estruturas de dados não foram explorados. Além disso, não realizamos testes em conjuntos de dados muito grandes ou altamente específicos (como dados quase ordenados ou com padrões repetidos), o que poderia trazer insights adicionais sobre o comportamento dos algoritmos.

Outra limitação foi o ambiente de teste utilizado. Diferentes arquiteturas de hardware e configurações de sistema podem influenciar os tempos de execução, especialmente em algoritmos mais complexos como o Quick Sort e o Merge Sort.

## 5.11 Sugestões para Estudos Futuros

Para estudos futuros, sugerimos a análise de algoritmos de ordenação híbridos, como o TimSort, que combina o Merge Sort com o Insertion Sort para otimizar o desempenho em entradas parcialmente ordenadas. Também seria interessante explorar a eficiência dos algoritmos de ordenação em conjuntos de dados ainda maiores e em diferentes plataformas de hardware.

Outro aspecto a ser investigado é o impacto de paralelização dos algoritmos em ambientes de múltiplos núcleos ou distribuídos, especialmente para algoritmos como Quick Sort e Merge Sort, que podem se beneficiar significativamente de abordagens paralelas.

# 6 Conclusão

A análise dos resultados dos algoritmos de ordenação revela diferenças significativas em desempenho, dependendo da ordem e do tamanho dos dados. Algoritmos como Merge Sort e Quick Sort demonstram melhor desempenho em cenários gerais, enquanto Insertion Sort, Selection Sort e Bubble Sort são menos eficientes para listas maiores e desordenadas. A escolha do algoritmo adequado deve considerar as características da entrada e os requisitos de desempenho.

## 6.1 Recomendações

Com base nos resultados observados, recomendamos o uso do Merge Sort ou do Quick Sort para a maioria das aplicações práticas, especialmente quando

o desempenho consistente em todos os cenários é crucial. O Heap Sort pode ser uma boa alternativa quando o uso de memória é uma preocupação. Por outro lado, algoritmos como Insertion Sort e Selection Sort são recomendados apenas para conjuntos de dados pequenos ou já quase ordenados, onde sua simplicidade e eficiência podem ser mais adequadas.

## 7 Referências

1. Cormen, Thomas H., et al. \*Introduction to algorithms\*. MIT press, 2009.
2. Sedgewick, Robert, and Kevin Wayne. \*Algorithms\*. Addison-Wesley Professional, 2011.
3. Knuth, Donald E. \*The art of computer programming: sorting and searching\*. Vol. 3. Addison-Wesley Professional, 1998.
4. Skiena, Steven S. \*The algorithm design manual\*. Springer Nature, 2020.
5. Goodrich, Michael T., and Roberto Tamassia. \*Algorithm design and applications\*. John Wiley & Sons, 2014.
6. McIlroy, M. Douglas. "A killer adversary for quicksort." \*Software-Practice and Experience\*, vol. 29, no. 4, 1999, pp. 341-344.