UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

CENTRO DE ENSINO SUPERIOR DO SERIDÓ

BACHARELADO EM SISTEMAS DE INFORMAÇÃO.

FRANCIMAR ALEXANDRE DE OLIVEIRA DANTAS

ATIVIDADE PRÁTICA I

CAICÓ - RN

2025

FRANCIMAR ALEXANDRE DE OLIVEIRA DANTAS

ATIVIDADE PRÁTICA I:

Trabalho apresentado como parte dos requisitos para a obtenção de nota na disciplina Estrutura de Dados, do curso de Bacharelado em Sistemas de Informação, da Universidade Federal do Rio Grande do Norte.

Orientador(a): Dr. JOAO PAULO DE SOUZA MEDEIROS.

.

CAICÓ - RN

2025

RESUMO

Relatório escrito com objetivo de analisar, os algoritmos de ordenação, insertion-sort, merge-sort, e observar seus respectivos tempos de execução. Essa analise será feita através de gráficos, e por meio de analise assintótica. **Palavras-chave**:

Ordenação, Algoritmos, Complexidade, Análise de Desempenho, Estrutura de Dados.

ABSTRACT

Written report with the objective of analyzing the insert-sort-merge-sort sorting algorithms, and observing their respective execution times. This analysis will be done through graphs, and through asymptotic analysis. **Keywords**: Sorting Algorithms, Computational Complexity, Performance Analysis, Data Structures, Efficiency.

LISTA DE FIGURAS

SUMÁRIO

1 Insertion-Sort

1 Descrição do Algoritmo

Insere cada elemento na posição correta em uma parte já ordenada do vetor. Vantagem: Simples e eficiente para pequenos conjuntos ou quase ordenados.

2 algoritmo utilizado

```
def insertion_sort(A):
    for i in range(1, len(A)):
        chave = A[i]
        j = i
        while j > 0 and chave < A[j - 1]:
            A[j] = A[j - 1]
            j -= 1
        A[j] = chave
    return A
```

3 Análise de Complexidade

Melhor Caso – O(n)

O melhor caso ocorre quando o vetor já está ordenado. Nesse cenário, a condição do `while` nunca é satisfeita, pois nenhum elemento é menor que os anteriores. Assim, o algoritmo realiza apenas as comparações do `for`, sem executar o `while`.

$$T(n) = c \cdot n \Rightarrow \boxed{O(n)}$$

Pior Caso – O(n²)

O pior caso ocorre quando o vetor está ordenado em ordem decrescente. Cada novo elemento precisa ser comparado e trocado com todos os anteriores, fazendo com que o número de comparações e trocas aumente quadraticamente.

$$T(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \Rightarrow \boxed{O(n^2)}$$

Caso Médio – O(n²)

No caso médio, assumimos que os elementos estão dispostos aleatoriamente. Em média, o elemento atual será comparado com metade dos anteriores. Ainda assim, o número total de operações cresce quadraticamente.

$$T(n) \approx \frac{n^2}{4} \Rightarrow \boxed{O(n^2)}$$

4 Resumo da Tabela de Complexidade

Caso	Comparações	Trocas	Complexidade
Melhor	$n-1$	0	$O(n)$
Pior	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	$O(n^2)$
Médio	$\approx \frac{n^2}{4}$	$\approx \frac{n^2}{4}$	$O(n^2)$

5 gráficos insertion

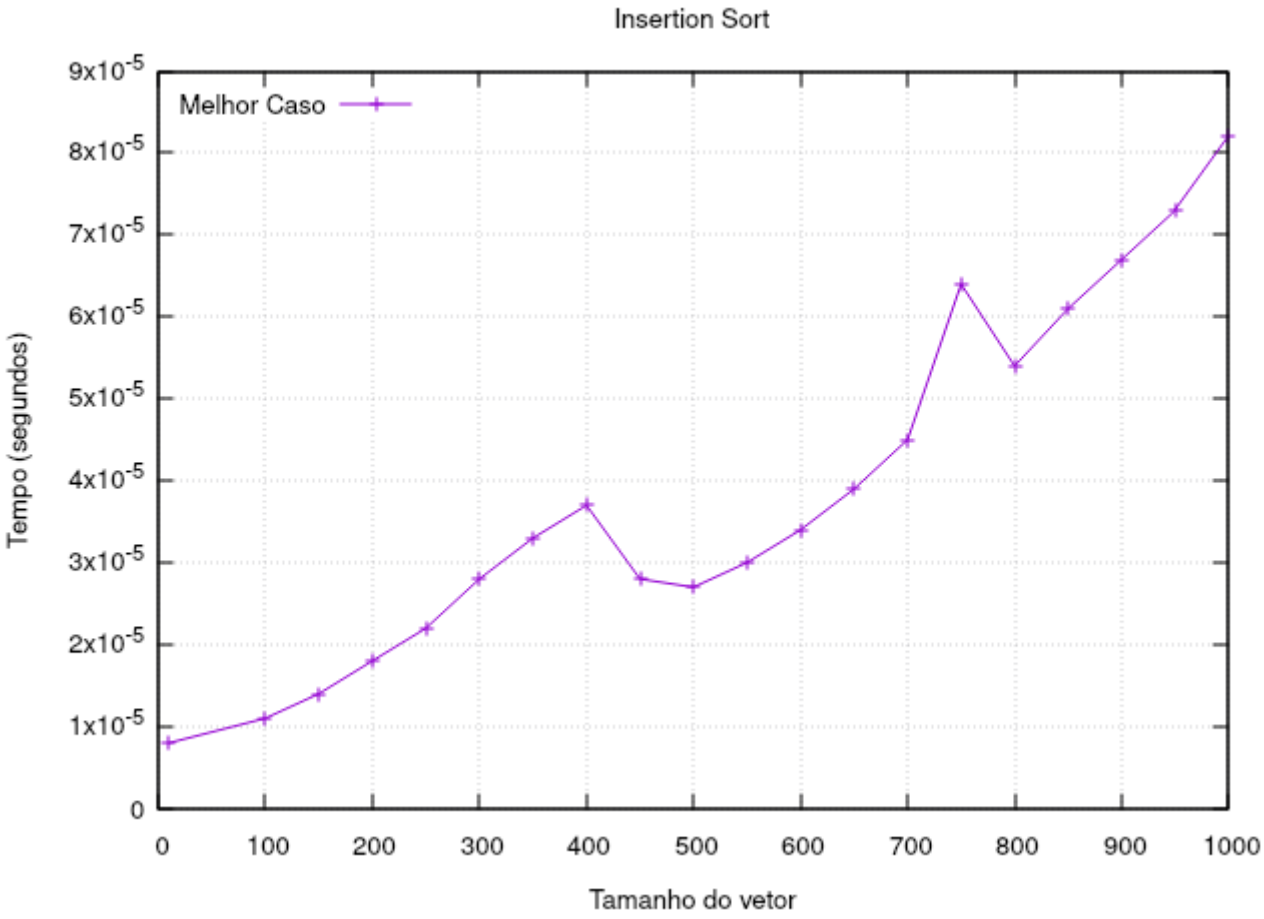

Figura 1 – Gráfico do melhor caso insertion sort

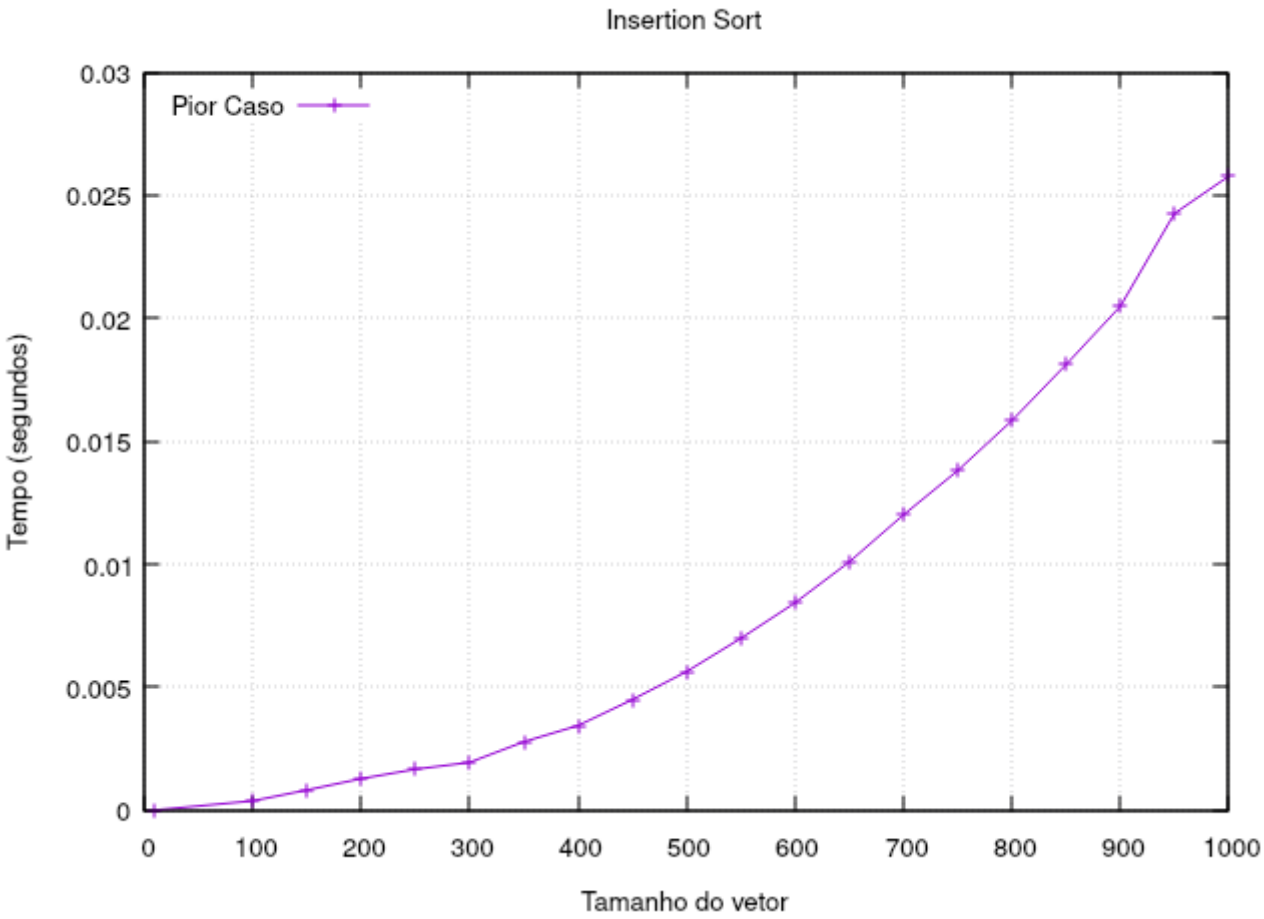

Figura 2 – Gráfico do pior caso insertion sort

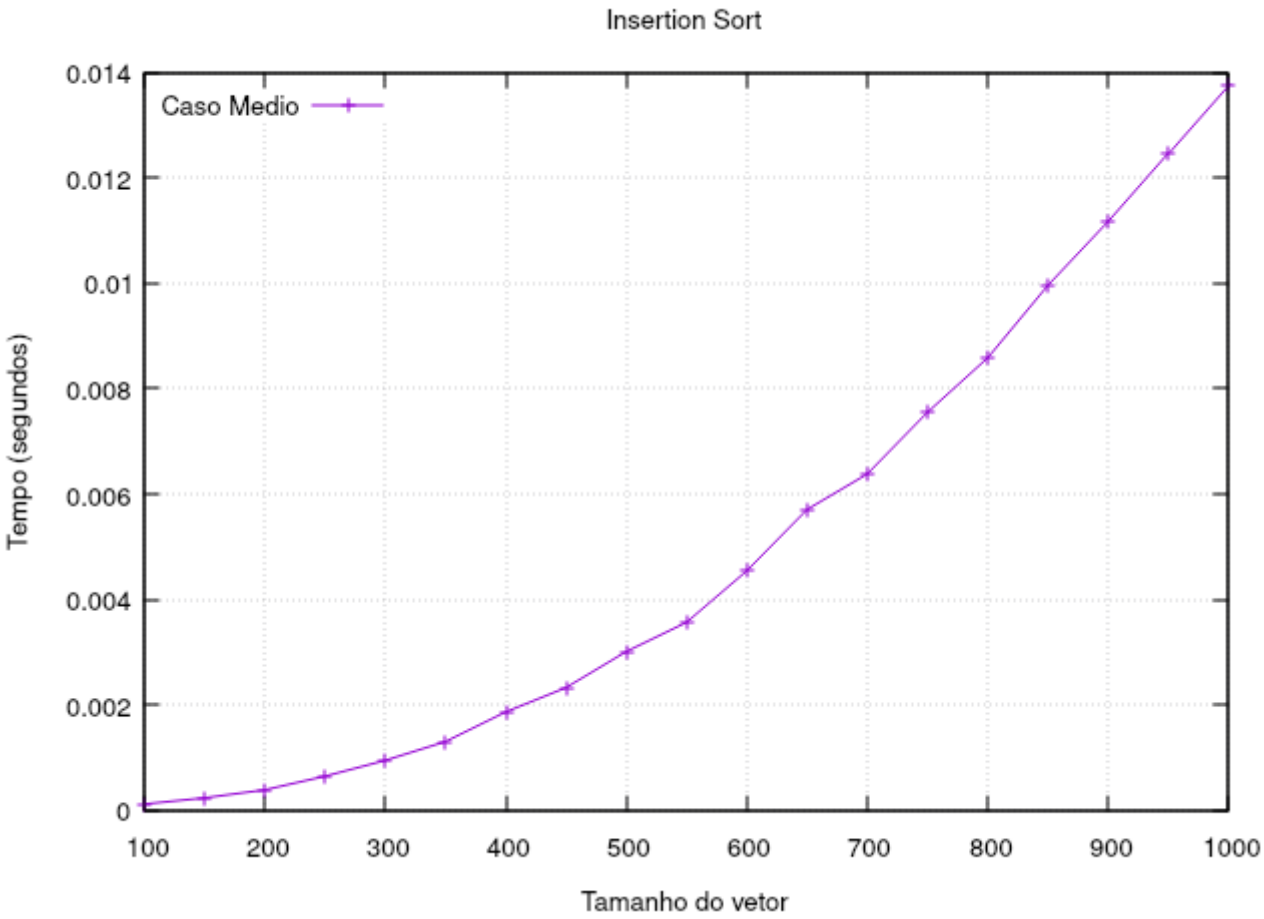

Figura 3 – Gráfico do caso médio insertion sort

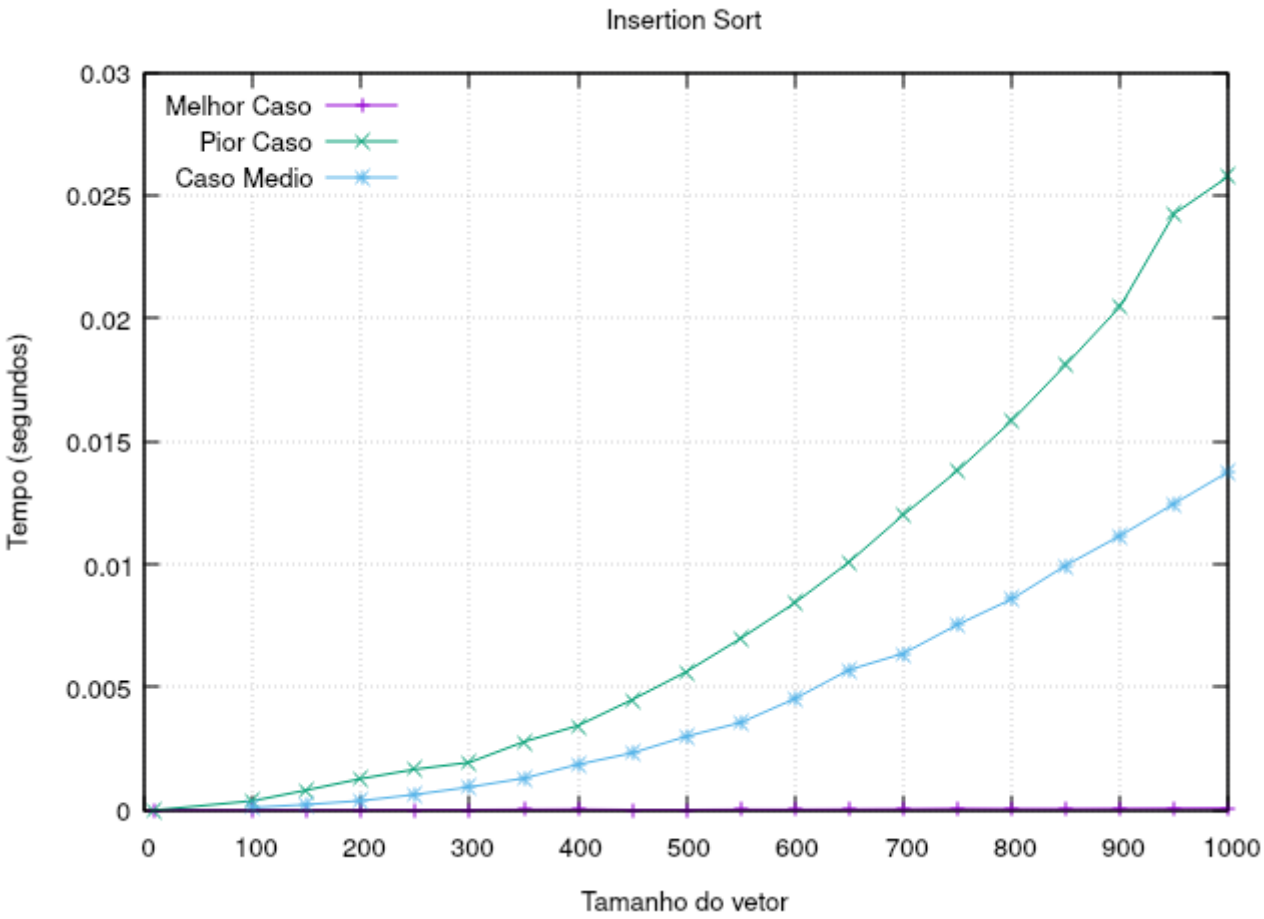

Figura 4 – Gráfico insertion sort - comparação dos pior melhor e médio caso

2 Selection Sort

6 Descrição do Algoritmo

O **Selection Sort** percorre o vetor procurando o menor elemento e o coloca na primeira posição. Em seguida, repete esse processo para a segunda posição, e assim por diante. Em cada iteração, uma seleção é feita e uma troca ocorre ao final da varredura.

7 Algoritmo Utilizado

```
def selection_sort(A):
    n = len(A)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if A[j] < A[min_idx]:
                min_idx = j
        A[i], A[min_idx] = A[min_idx], A[i]
    return A
```

8 Análise de Complexidade

Melhor Caso – O(n²)

Mesmo que o vetor já esteja ordenado, o algoritmo ainda realiza todas as comparações possíveis, pois não há verificação de ordem. No entanto, o número de trocas é mínimo, pois o menor elemento já está na posição correta em cada iteração.

$$T(n) = \sum_{i=0}^{n-1}(n - i - 1) = \frac{n(n-1)}{2} \Rightarrow \boxed{O(n^2)}$$

Pior Caso – O(n²)

O número de comparações e o comportamento do algoritmo são os mesmos em qualquer caso, já que ele sempre percorre o vetor para encontrar o menor valor restante. O número de trocas pode variar, mas as comparações permanecem constantes.

$$T(n) = \frac{n(n-1)}{2} \Rightarrow \boxed{O(n^2)}$$

Caso Médio – O(n²)

Como o algoritmo não se beneficia da ordenação parcial dos elementos, o caso médio também realiza o mesmo número de comparações, independentemente da distribuição dos valores no vetor.

$$T(n) = \frac{n(n-1)}{2} \Rightarrow \boxed{O(n^2)}$$

9 Resumo da Tabela de Complexidade

Caso	Comparações	Trocas	Complexidade
Melhor	$\frac{n(n-1)}{2}$	≈ 0	$O(n^2)$
Pior	$\frac{n(n-1)}{2}$	$\approx n$	$O(n^2)$
Médio	$\frac{n(n-1)}{2}$	$\approx n$	$O(n^2)$

10 Gráficos selection sort

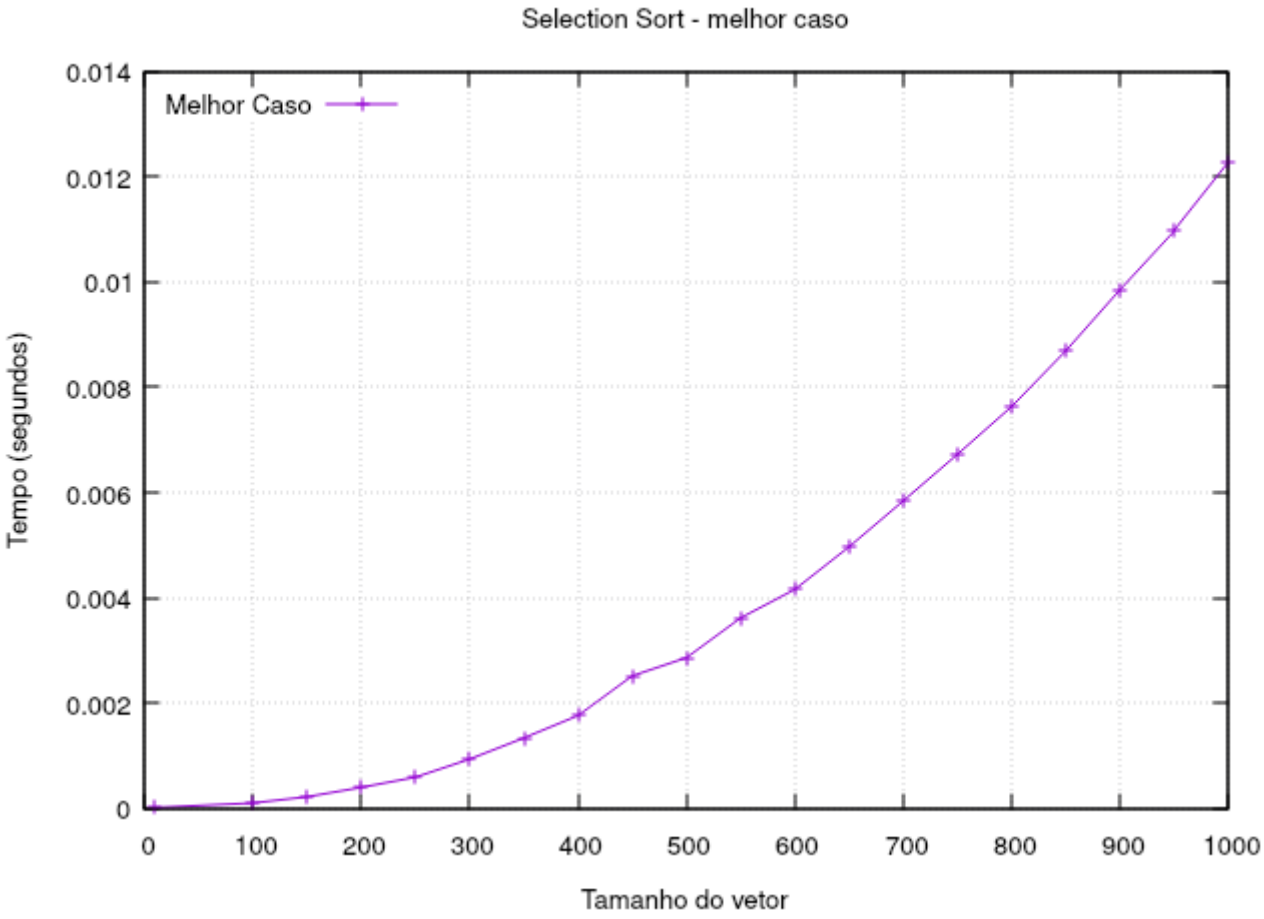

Figura 5 – Gráfico do melhor caso selection sort

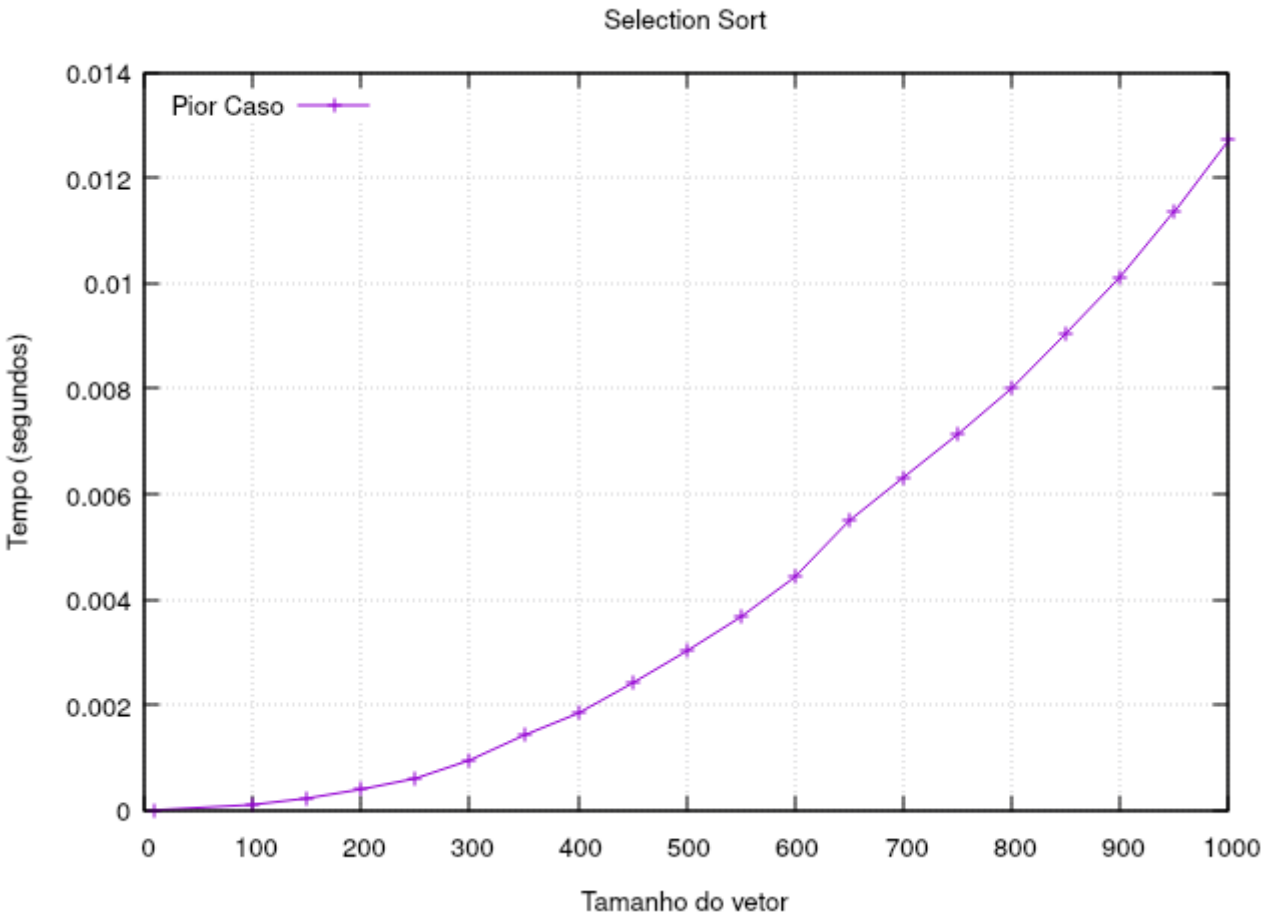

Figura 6 – Gráfico do pior caso selection sort

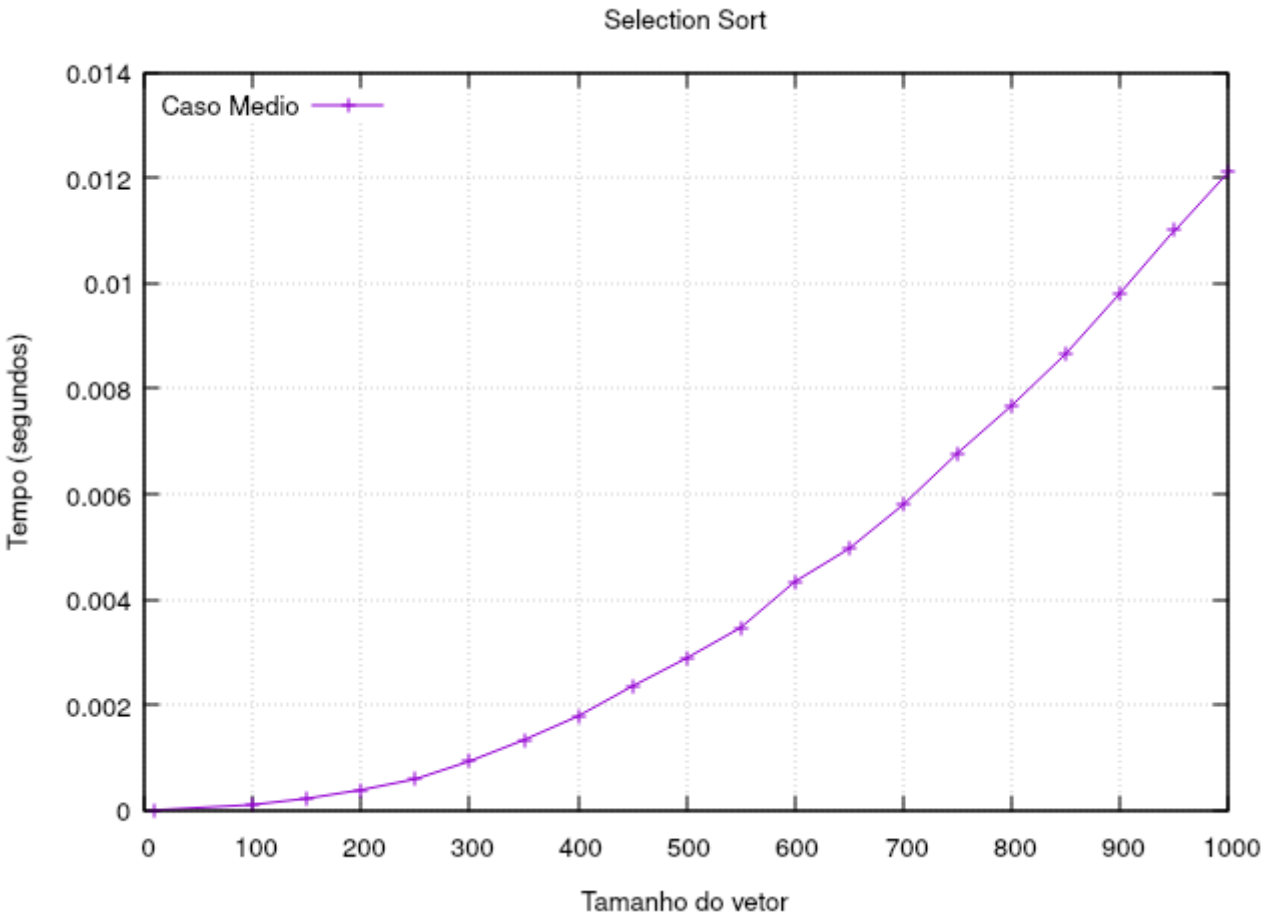

Figura 7 – Gráfico do caso médio selection sort

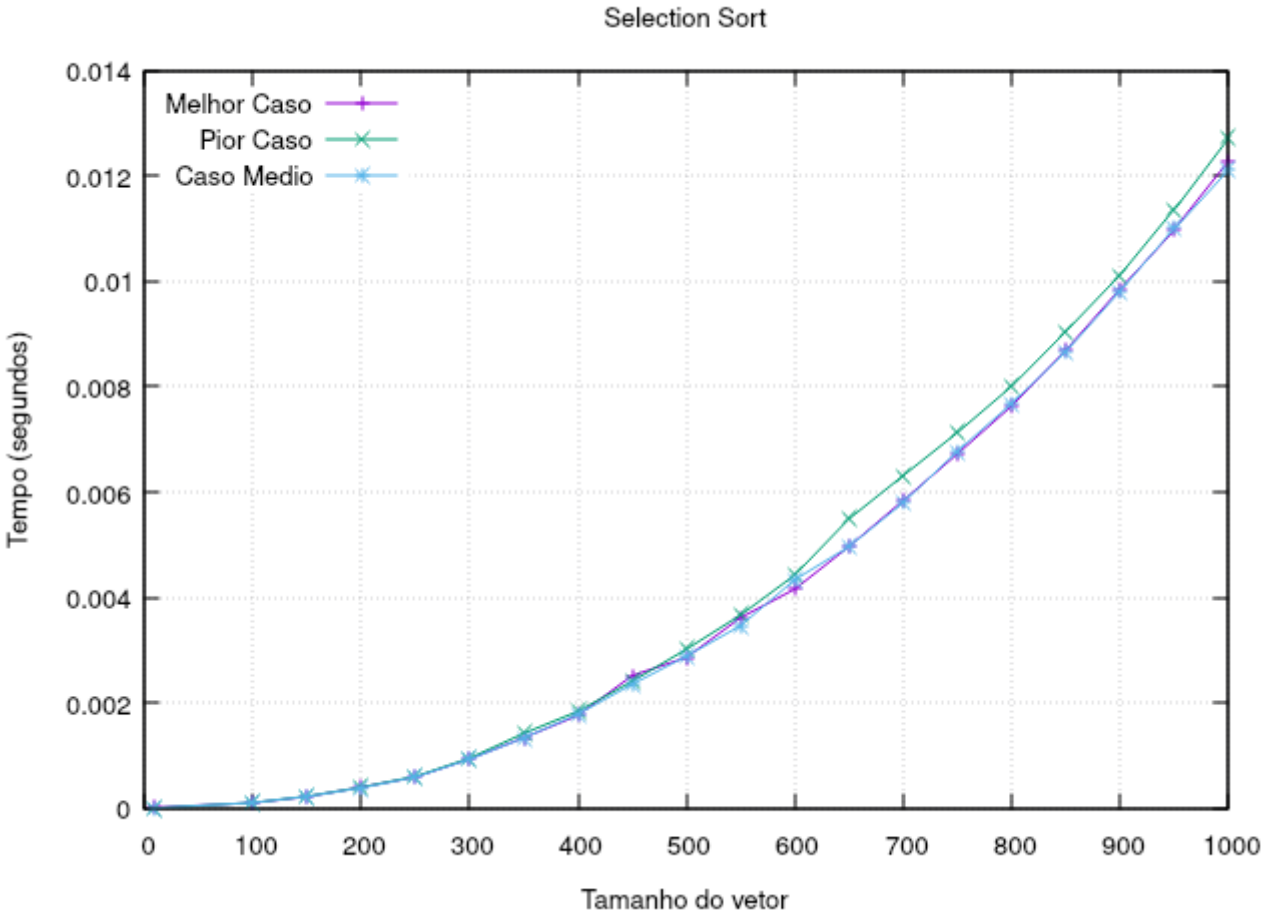

Figura 8 – Gráfico comparação do melhor, pior e médio caso selection

3 Merge Sort

11 Descrição do Algoritmo

O Merge Sort divide o vetor em duas metades, ordena recursivamente cada metade e depois junta (merge) as duas partes de forma ordenada. É eficiente e estável, especialmente para grandes volumes de dados.

12 Algoritmo Utilizado

```
def merge_sort(A):
    if len(A) <= 1:
        return A
    meio = len(A) // 2
    esquerda = merge_sort(A[:meio])
    direita = merge_sort(A[meio:])
    return merge(esquerda, direita)

def merge(esq, dir):
    resultado = []
    i = j = 0
    while i < len(esq) and j < len(dir):
        if esq[i] <= dir[j]:
            resultado.append(esq[i])
            i += 1
        else:
            resultado.append(dir[j])
            j += 1
    resultado.extend(esq[i:])
    resultado.extend(dir[j:])
    return resultado
```

13 Análise de Complexidade

Melhor Caso – O(n log n)

Mesmo com os dados ordenados, o Merge Sort sempre divide o vetor e realiza fusões. Por isso, o tempo de execução permanece em $O(n \log n)$.

$$T(n) = 2T(n/2) + n \Rightarrow \boxed{O(n \log n)}$$

Pior Caso – O(n log n)

Independente da ordem dos dados, o processo de divisão e mesclagem ocorre sempre, resultando em um comportamento estável e previsível.

$$T(n) = 2T(n/2) + n \Rightarrow \boxed{O(n \log n)}$$

Caso Médio – O(n log n)

Como o algoritmo sempre realiza o mesmo número de divisões e junções, o tempo médio de execução também é $O(n \log n)$.

$$T(n) = 2T(n/2) + n \Rightarrow \boxed{O(n \log n)}$$

14 Resumo da Tabela de Complexidade

Caso	Comparações	Trocas (ou fusões)	Complexidade
Melhor	$n \log n$	n	$O(n \log n)$
Pior	$n \log n$	n	$O(n \log n)$
Médio	$n \log n$	n	$O(n \log n)$

15 Gráficos merge sort

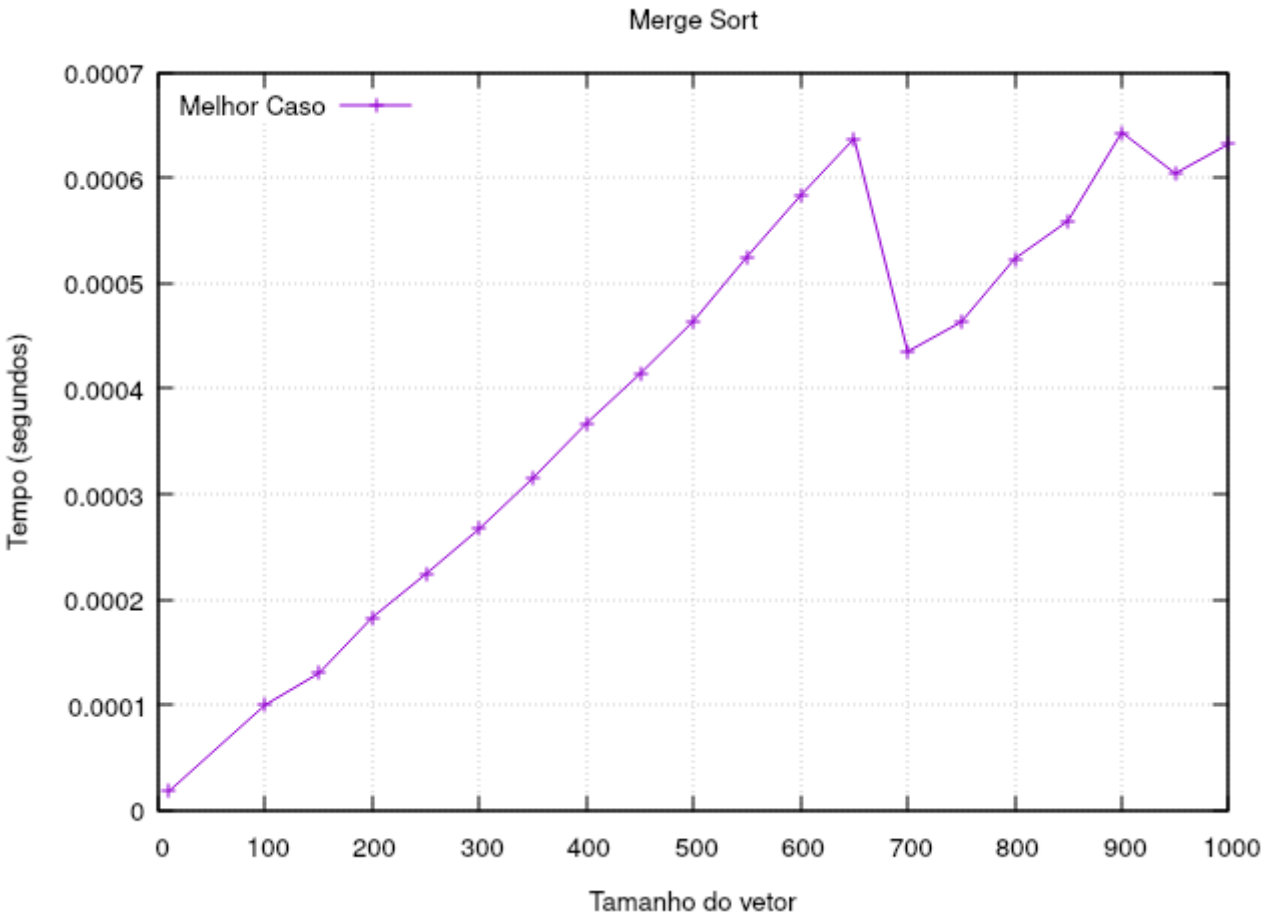

Figura 9 – Gráfico do melhor caso Merge Sort

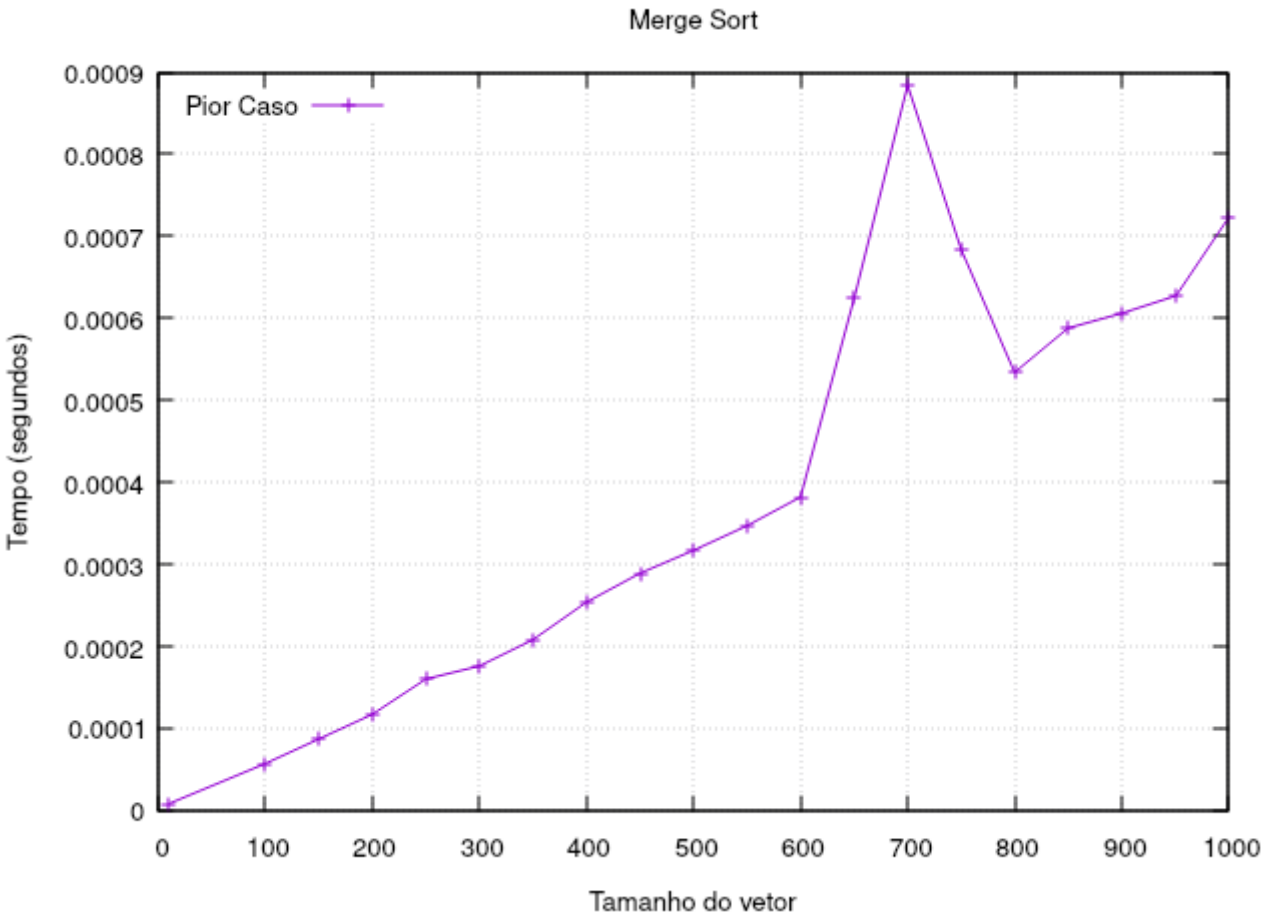

Figura 10 – Gráfico do pior caso Merge Sort

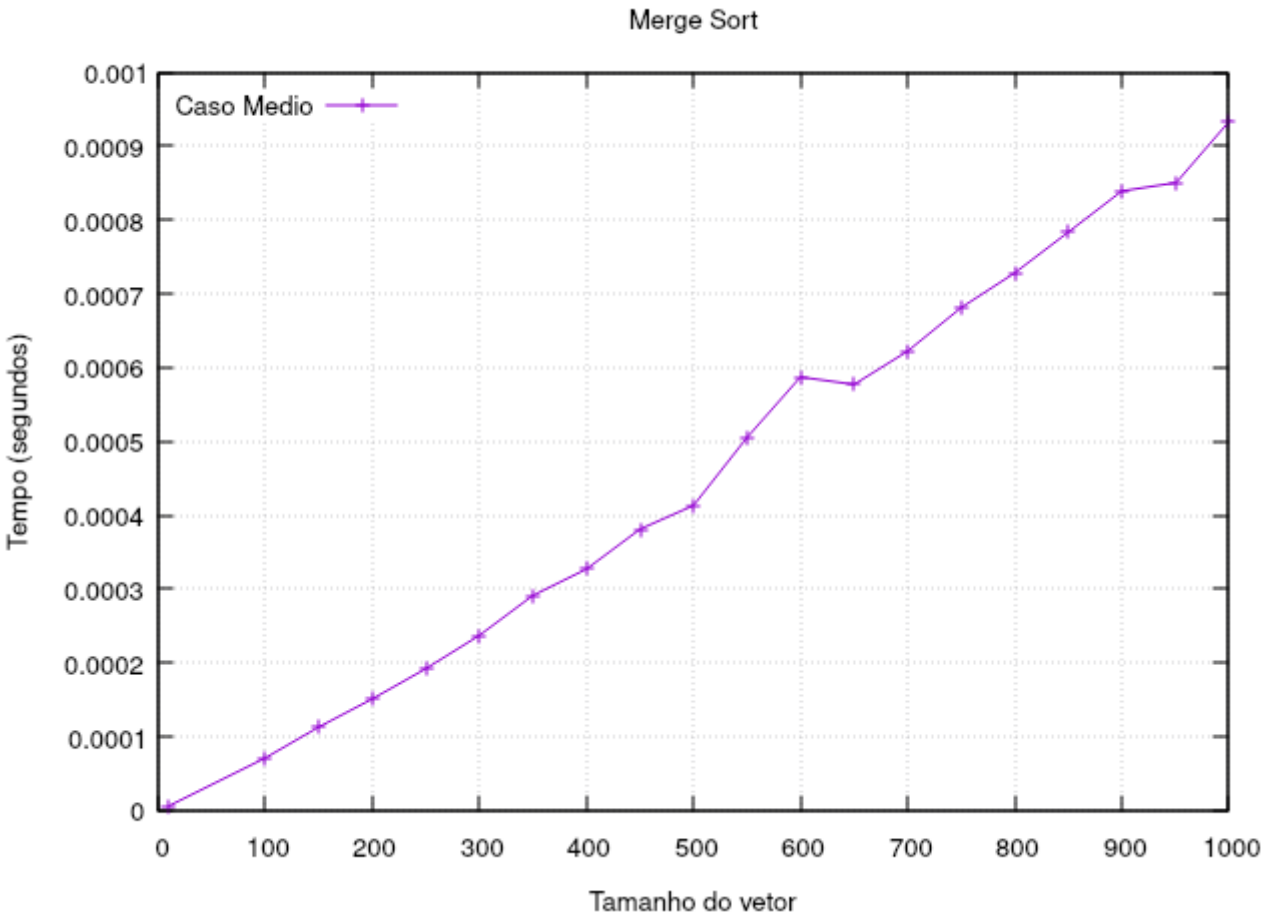

Figura 11 – Gráfico do caso médio Merge Sort

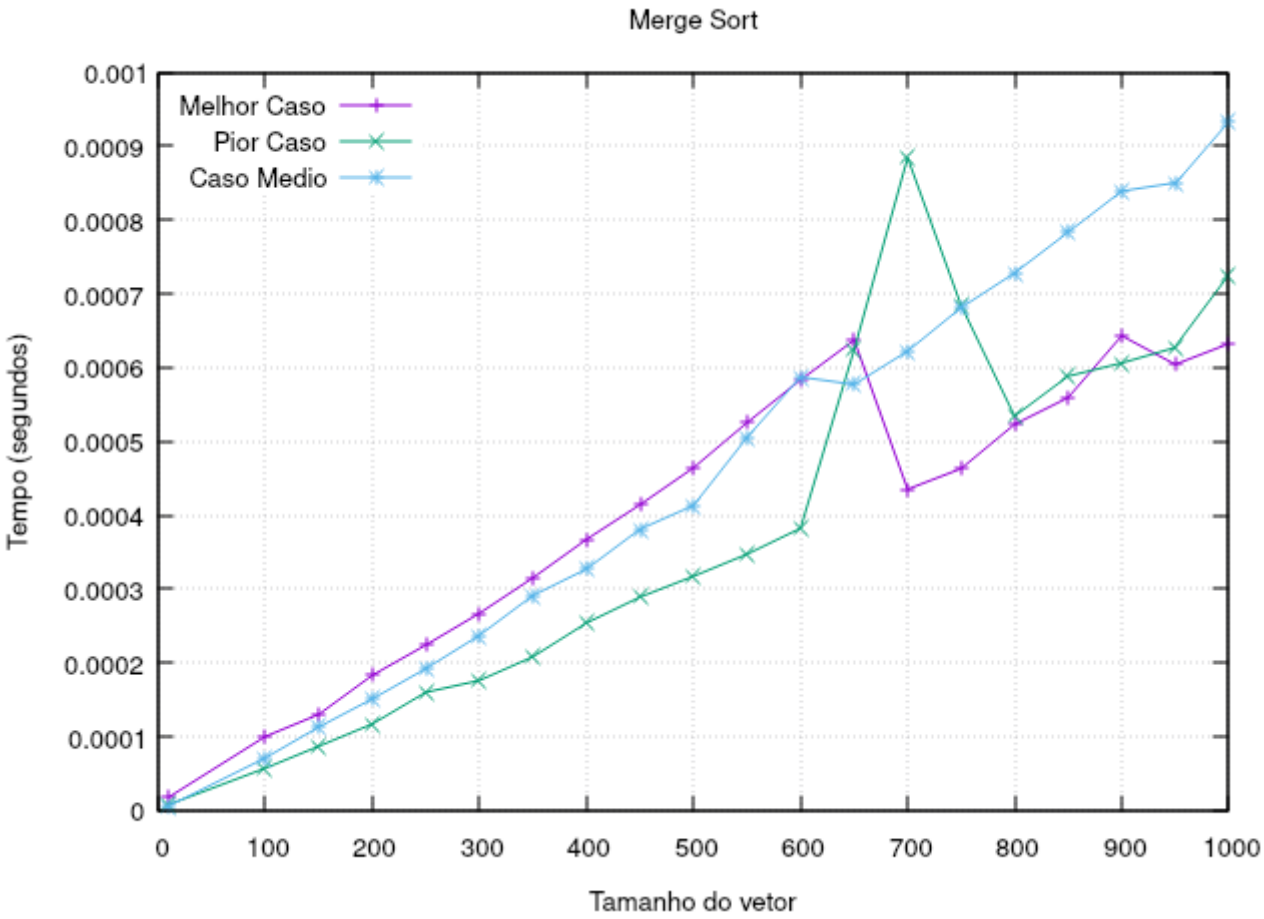

Figura 12 – Gráfico comparação melhor pior e médio caso merge sort

4 Quick Sort

16 Descrição do Algoritmo

O Quick Sort escolhe um elemento como pivô, particiona o vetor em dois subgrupos (menores e maiores que o pivô) e aplica a ordenação recursivamente. É muito rápido para dados aleatórios, mas sensível à escolha do pivô.

17 Algoritmo Utilizado

```
def quick_sort(A):
    if len(A) <= 1:
        return A
    pivot = A[-1]
    menores = [x for x in A[:-1] if x <= pivot]
    maiores = [x for x in A[:-1] if x > pivot]
    return quick_sort(menores) + [pivot] + quick_sort(maiores)
```

18 Análise de Complexidade

Melhor Caso – O(n log n)

Ocorre quando o pivô divide o vetor de forma equilibrada. O número de chamadas recursivas e comparações é mínimo.

$$T(n) = 2T(n/2) + n \Rightarrow \boxed{O(n \log n)}$$

Pior Caso – O(n²)

Acontece quando o pivô é sempre o maior ou menor elemento, criando partições altamente desequilibradas.

$$T(n) = T(n-1) + n \Rightarrow \boxed{O(n^2)}$$

Caso Médio – O(n log n)

Assumindo uma distribuição aleatória dos pivôs, o desempenho médio tende a ser próximo do ótimo.

$$T(n) \approx 1.39n \log n \Rightarrow \boxed{O(n \log n)}$$

19 Resumo da Tabela de Complexidade

Caso	Comparações	Trocas	Complexidade
Melhor	$n \log n$	n	$O(n \log n)$
Pior	n^2	n^2	$O(n^2)$
Médio	$\approx 1.39n \log n$	$\approx n \log n$	$O(n \log n)$

20 Gráficos Quick sort

5 Counting Sort (Distribution Sort)

21 Descrição do Algoritmo

Counting Sort conta quantas vezes cada valor aparece e, com base nisso, determina sua posição final. É muito rápido para intervalos pequenos de inteiros.

22 Algoritmo Utilizado

```
def counting_sort(A):
    if not A:
        return []
    max_val = max(A)
    count = [0] * (max_val + 1)
    for num in A:
        count[num] += 1
    resultado = []
    for i, c in enumerate(count):
        resultado.extend([i] * c)
    return resultado
```

23 Análise de Complexidade

Melhor Caso – O(n + k)

Funciona em tempo linear desde que o maior valor (k) não seja muito maior que n.

$$T(n) = n + k \Rightarrow \boxed{O(n+k)}$$

Pior Caso – O(n + k)

Mesmo no pior cenário (dados dispersos), o tempo de execução continua linear.

$$T(n) = n + k \Rightarrow \boxed{O(n+k)}$$

Caso Médio – O(n + k)

Com dados uniformemente distribuídos, o algoritmo mantém sua eficiência.

$$T(n) = n + k \Rightarrow \boxed{O(n+k)}$$

24 Resumo da Tabela de Complexidade

Caso	Contagens	Escritas	Complexidade
Melhor	$n + k$	n	$O(n + k)$
Pior	$n + k$	n	$O(n + k)$
Médio	$n + k$	n	$O(n + k)$

25 gráfico distribution sort

6 Gráfico comparando todos os algoritmos

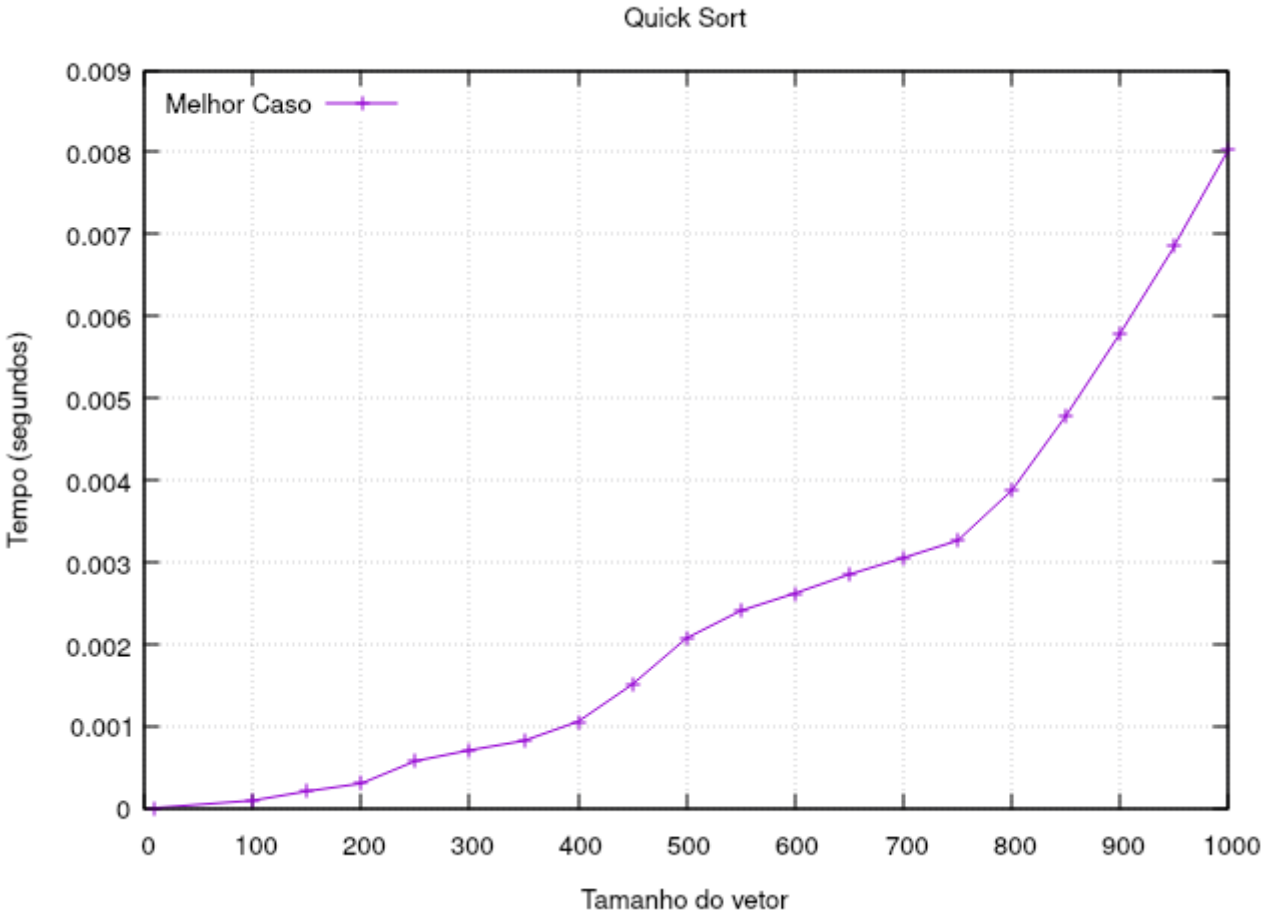

Figura 13 – Gráfico do melhor caso Quick Sort

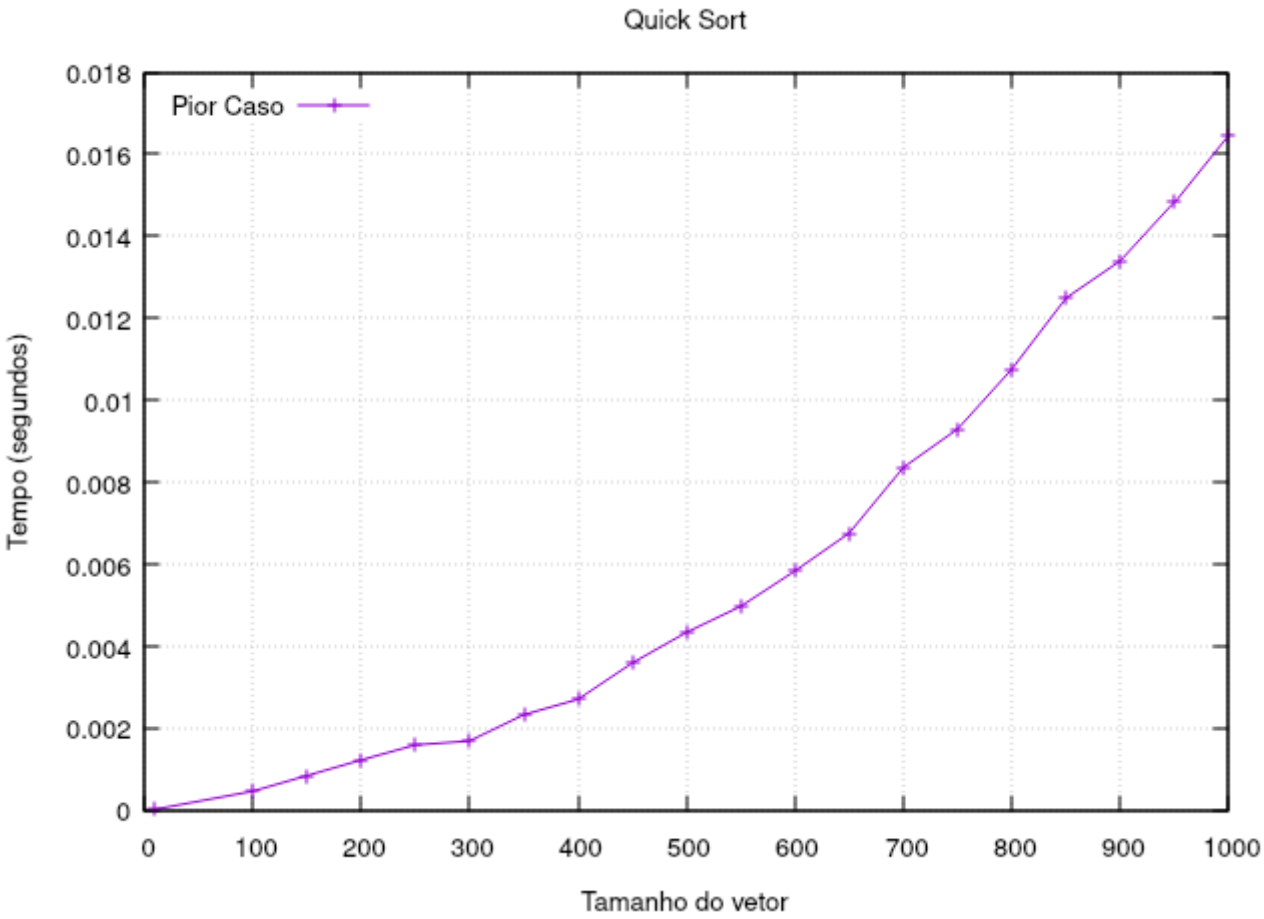

Figura 14 – Gráfico do pior caso Quick Sort

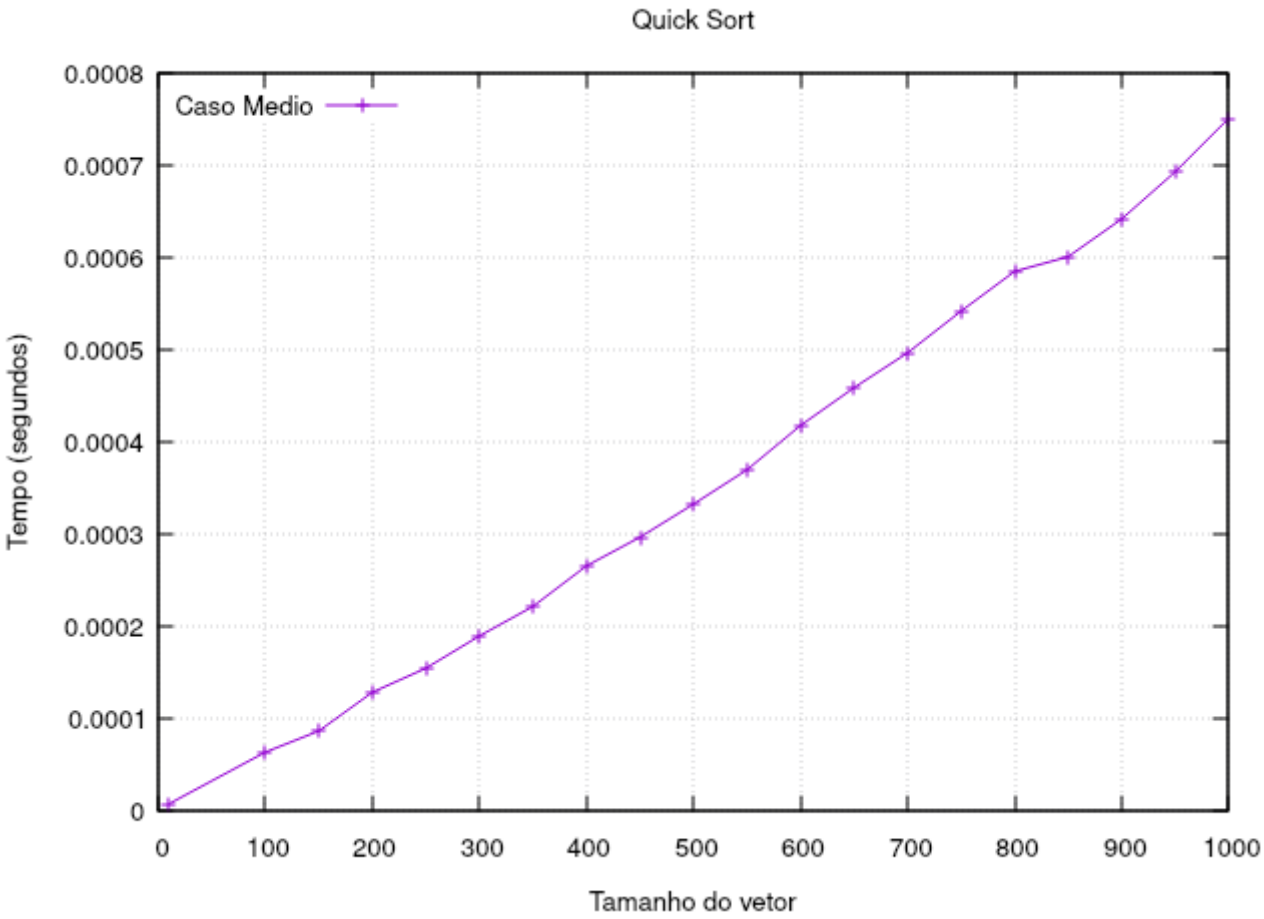

Figura 15 – Gráfico do caso médio Quick Sort

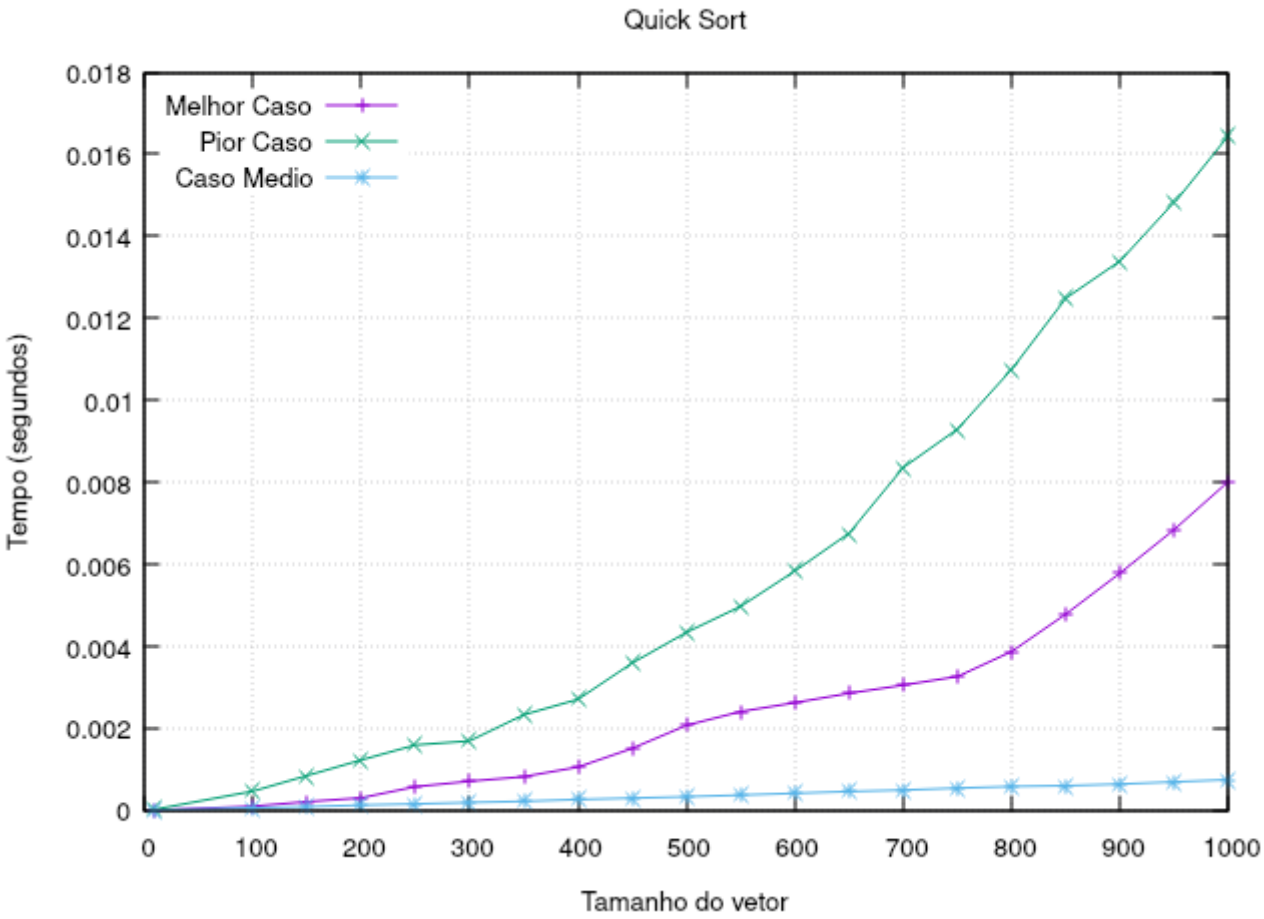

Figura 16 – Gráfico comparação melhor, pior e médio caso

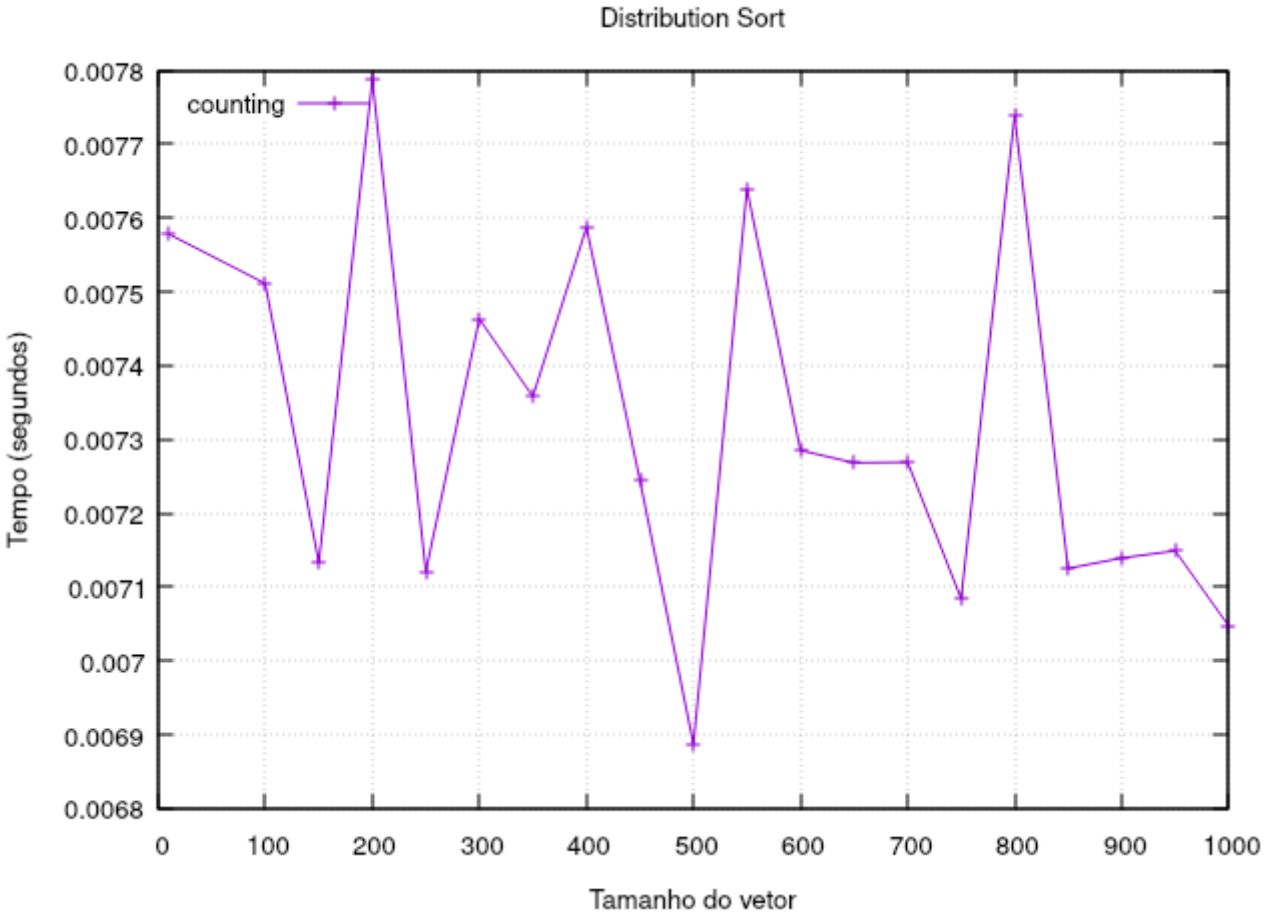

Figura 17 – Gráfico do distribution sort(Counting Sort)

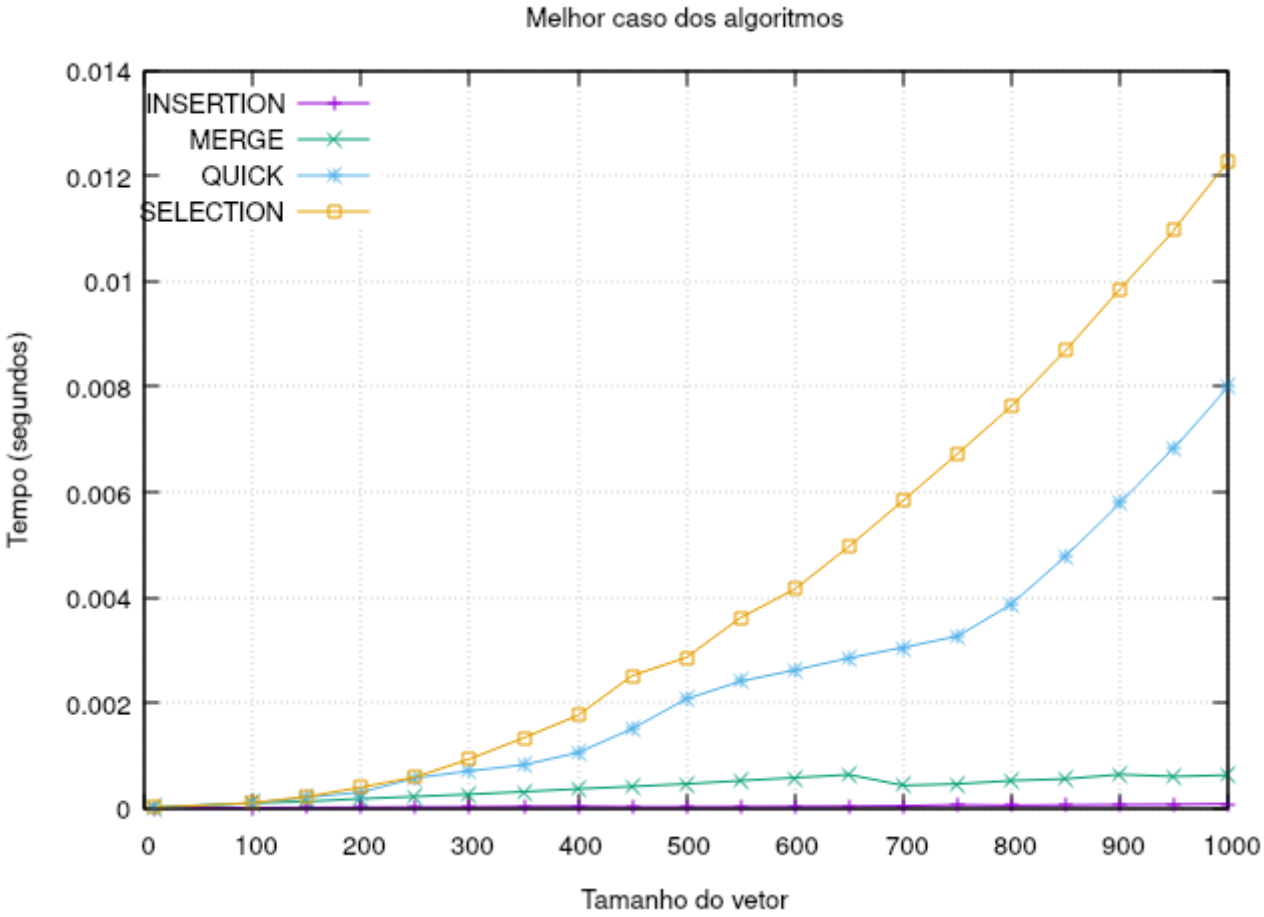

Figura 18 – Gráfico do melhor caso

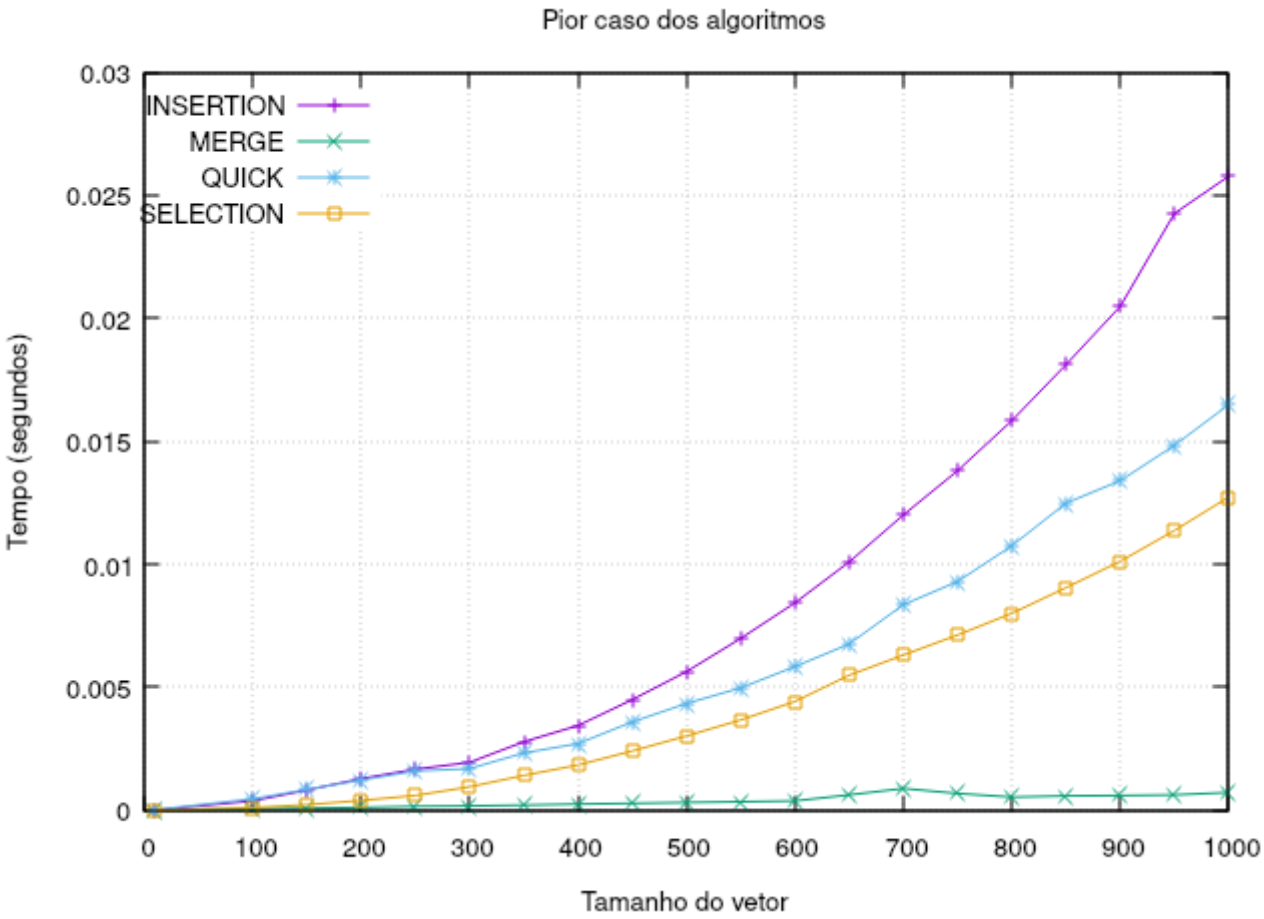

Figura 19 – Gráfico do pior caso

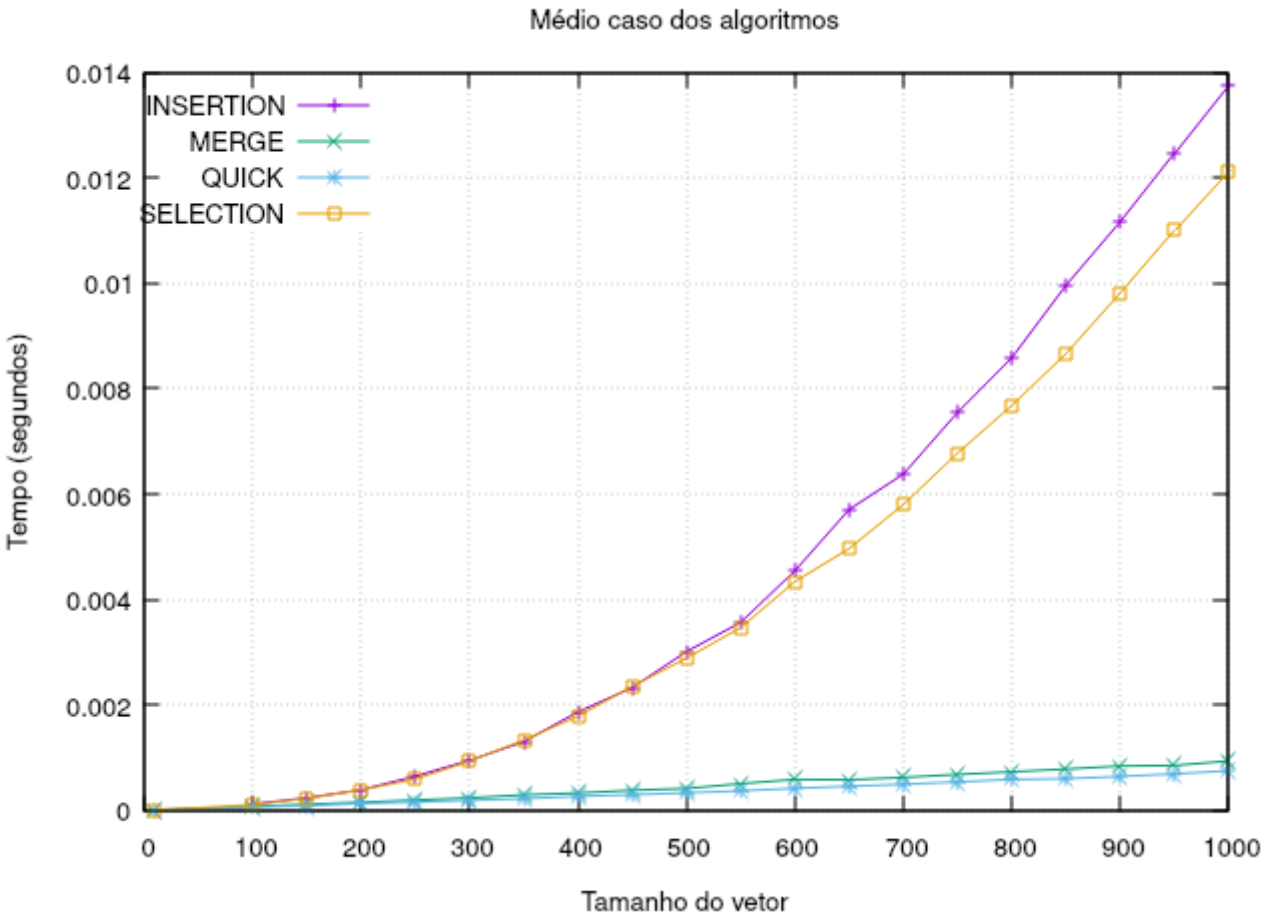

Figura 20 – Gráfico do caso médio