



UNIVERSIDADE FEDERAL DO TOCANTINS
CIÊNCIAS DA COMPUTAÇÃO - PROJETO E ANÁLISE DE ALGORITMOS
ALUNOS: JHONATA BATISTA, MARCOS ANTONIO, PLACIDO AQUINO, TATIANE
SHIBATA, SILVIO OTAVIO.

Relatório - Algoritmo de Ordenação

O documento a seguir tem como objetivo relatar e analisar o desempenho de algoritmos de ordenação em cenários simulados com vetores de diferentes tamanhos. A fim de buscarmos uma melhor solução para o problema a ser resolvido no futuro.

No decorrer do documento apresentaremos os resultados dos testes, que ocorreram em vetores de 100 a 10.000.000 de elementos inteiros positivos. Os algoritmos foram desenvolvidos utilizando a linguagem de programação Python.

Os seguintes algoritmos foram usados:

1. BubbleSort	2
2. SelectSort	5
3. InsertSort	10
4. MergeSort	13
5. QuickSort	17
6. BucketSort	21
7. HeapSort	24
8. CountingSort	25
9. Conclusão	27

1. BubbleSort

Complexidade pior caso: $O(n^2)$

Complexidade caso médio: $\theta(n^2)$

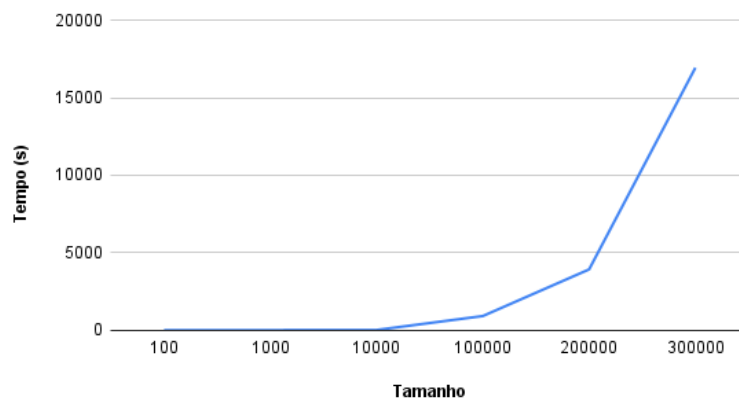
Complexidade melhor caso: $\Omega(n)$

Algoritmo:

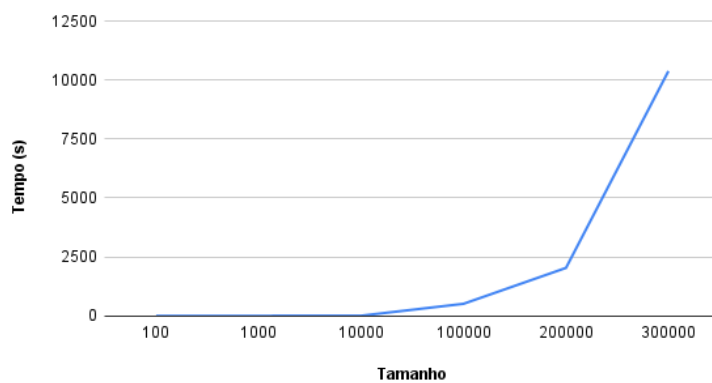
```
def bubbleSort(array):  
    trocas, comp = 0, 0  
    for i in range(len(array)-1):  
        comp += 1  
        for j in range(len(array)-1, i, -1):  
            comp += 1  
            if array[j] < array[j-1]:  
                aux = array[j]  
                array[j] = array[j-1]  
                array[j-1] = aux  
                trocas += 1  
    return [array, comp, trocas]
```

Resultados da ordenação:

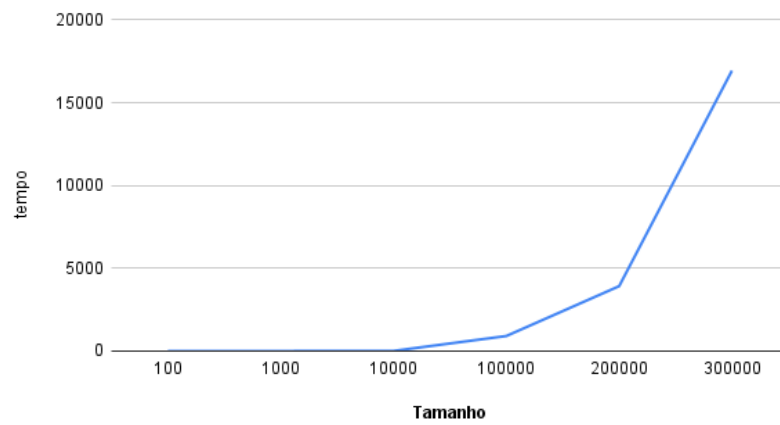
BubbleSort - Aleatorio



BubbleSort - Ordenado



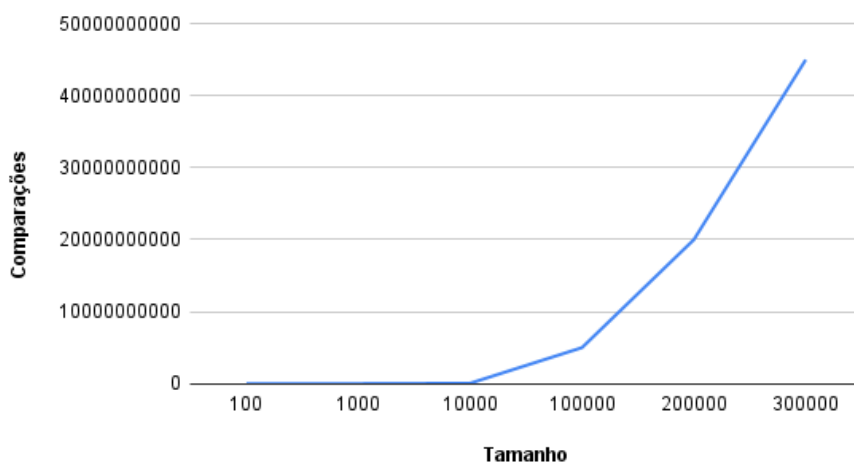
BubbleSort - Decrescente



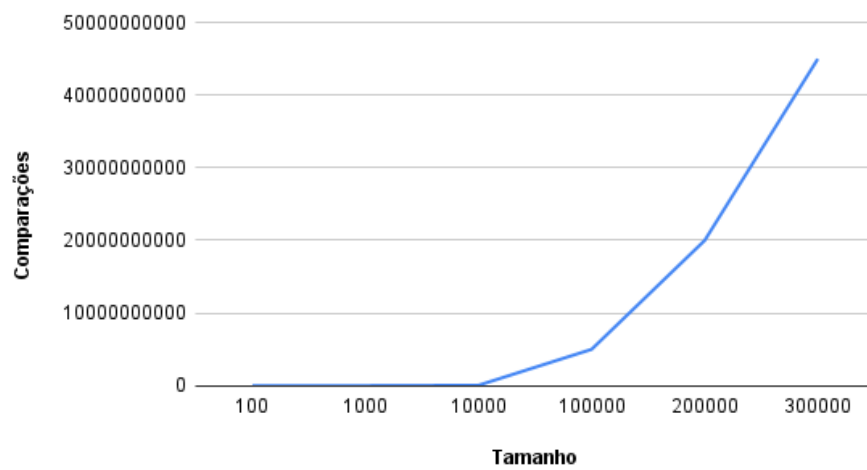
[08]

Número de comparações:

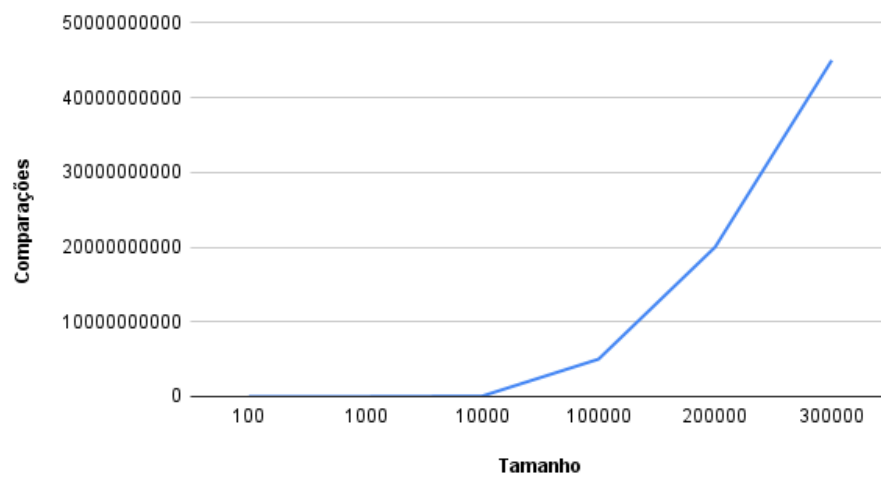
Comparações versus tamanho - Ordenado



Comparações versus tamanho - Aleatório



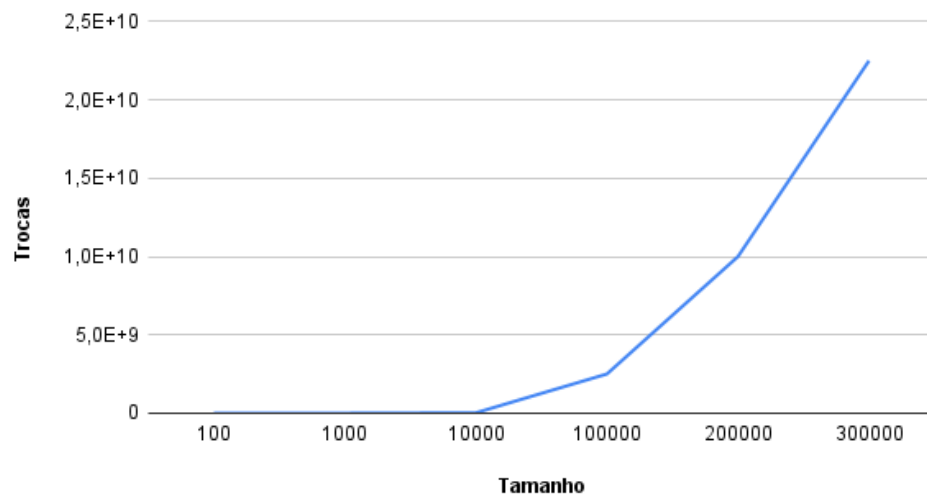
Comparações versus tamanho - Decrescente



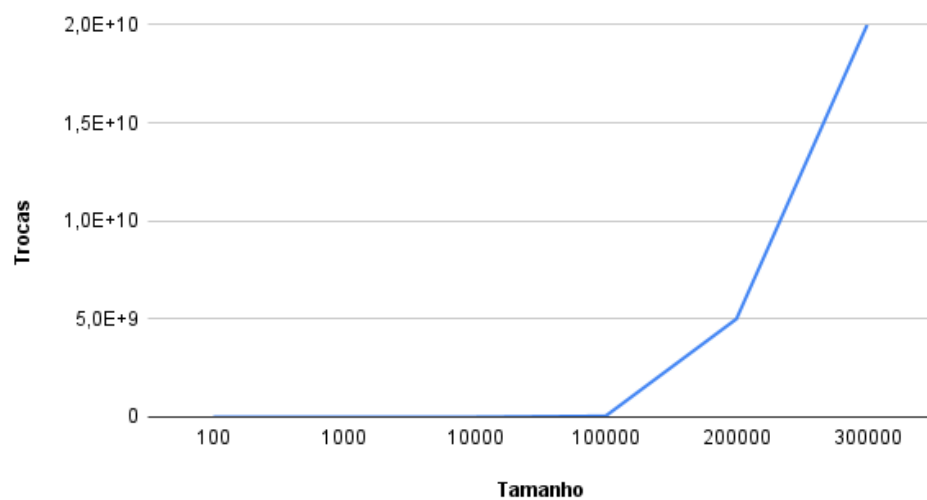
Número de trocas:

Com vetores ordenados, o algoritmo não apresentou nenhuma troca.

Trocas versus tamanho - Aleatório



Trocas versus tamanho - Decrescente



Conclusão:

Nos testes o bubbleSort não conseguia ordenar na casa dos 400.000 elementos. Além de que o algoritmo era bastante lento na quando a quantidade de elementos é maior que 100.000.

2. SelectSort

Complexidade pior caso: $O(n^2)$

Complexidade caso médio: $\theta(n^2)$

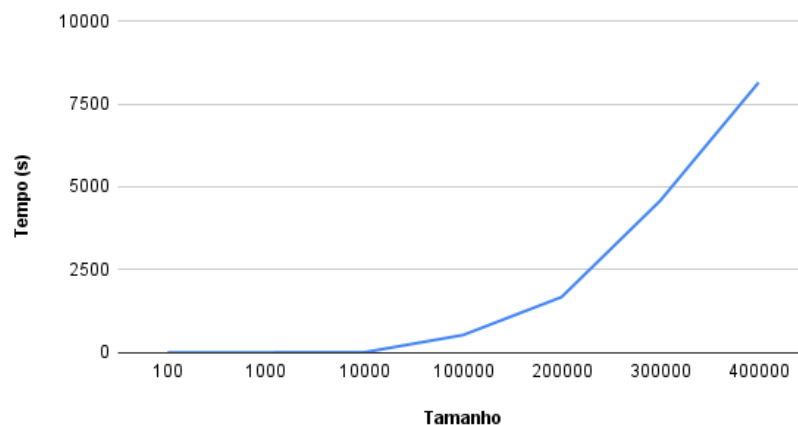
Complexidade melhor caso: $\Omega(n^2)$

Algoritmo:

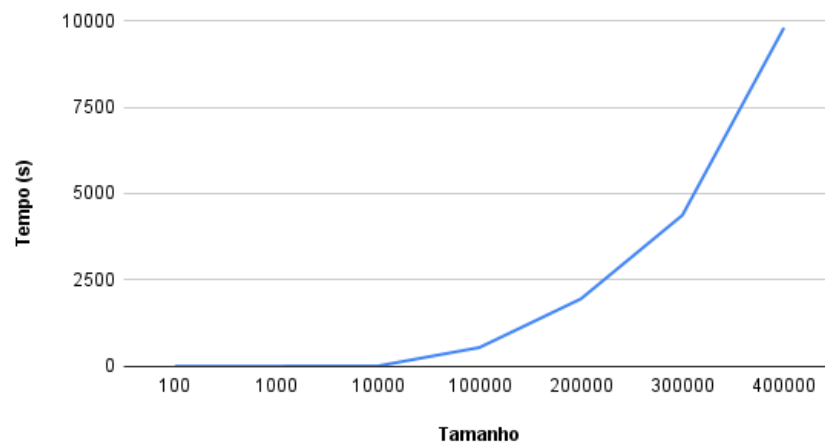
```
def selectionSort(array):  
    trocas, comp = 0, 0  
    for i in range(len(array)):  
        menor = i  
        for j in range(i+1, len(array)):  
            comp += 1  
            if array[j] < array[menor]:  
                menor = j  
  
        if array[i] != array[menor]:  
            temp = array[i]  
            array[i] = array[menor]  
            array[menor] = temp  
            trocas += 1  
  
    return [array, comp, trocas]
```

Resultados da ordenação:

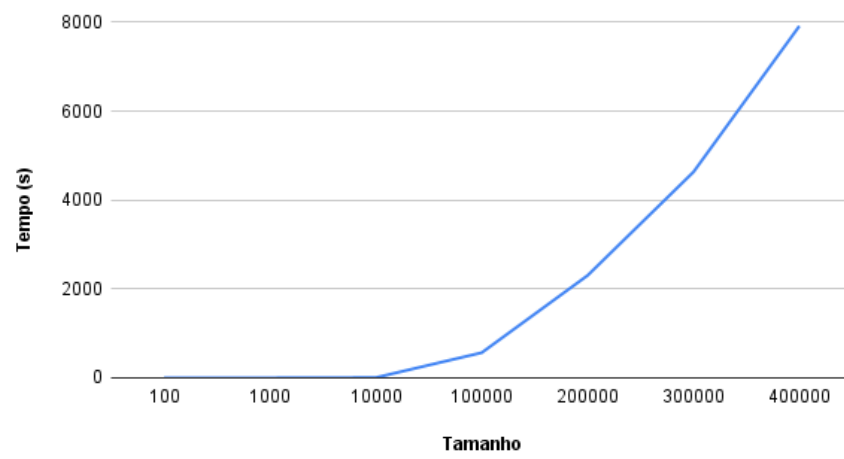
SelectionSort - Ordenado



SelectionSort - Aleatorio

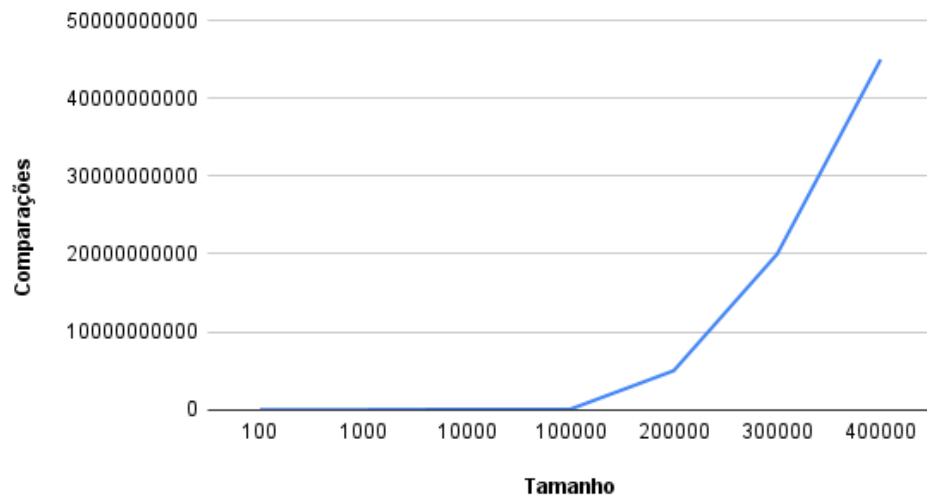


SelectionSort - Decrescente

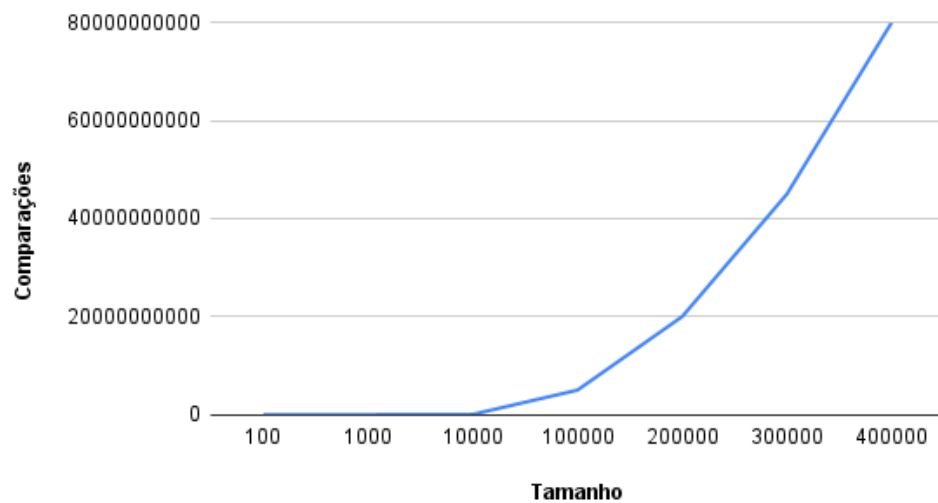


Número de Comparações:

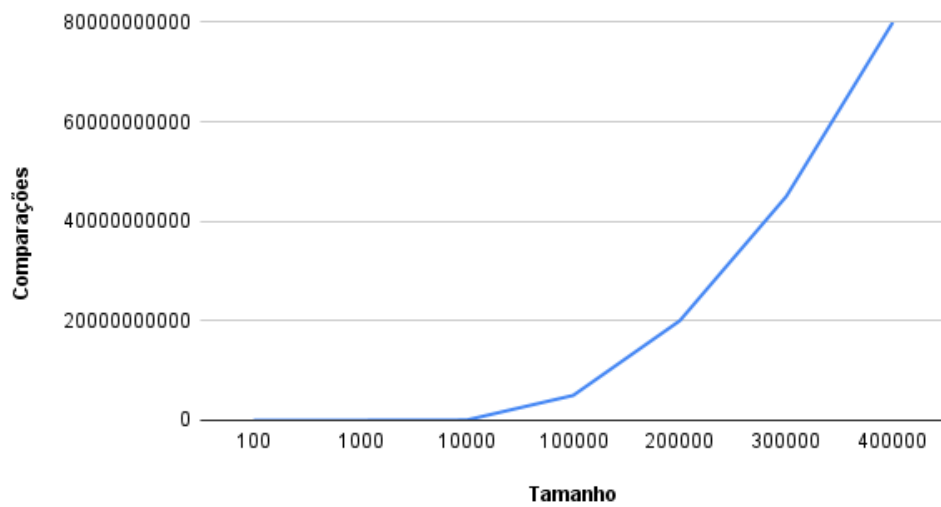
Comparações versus tamanho - Ordenado



Comparações versus tamanho - Aleatório



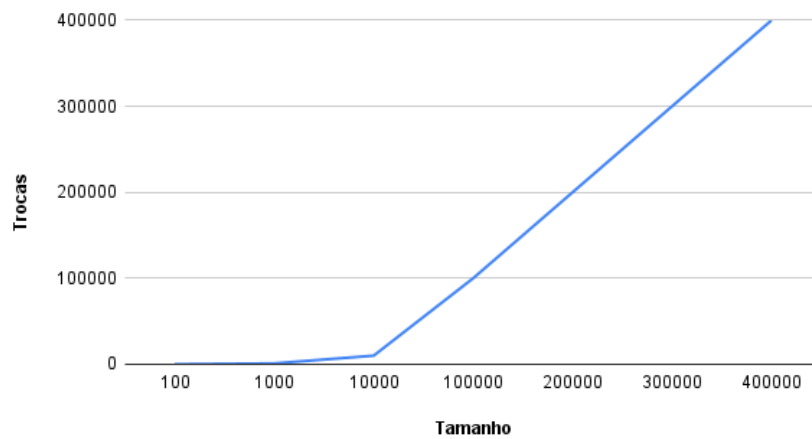
Comparações versus tamanho - Decrescente



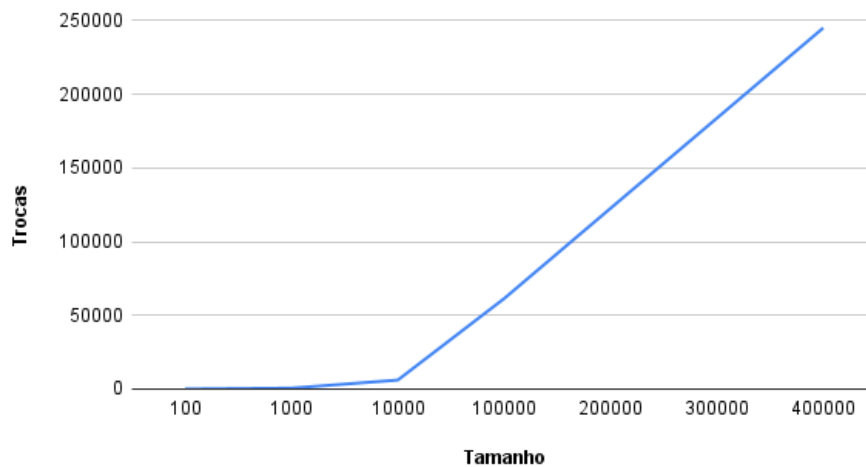
Número de Trocas:

Como o vetor é ordenado, o algoritmo não faz nenhuma troca.

Trocas versus tamanho - Aleatório



Trocas versus tamanho - Decrescente



Conclusão:

O selectSort assim como o bubbleSort não trabalha bem com vetores muito grandes. O algoritmo não conseguiu ordenar vetores na casa dos 500.000.

3. InsertSort

Complexidade pior caso: $O(n^2)$

Complexidade caso médio: $\theta(n^2)$

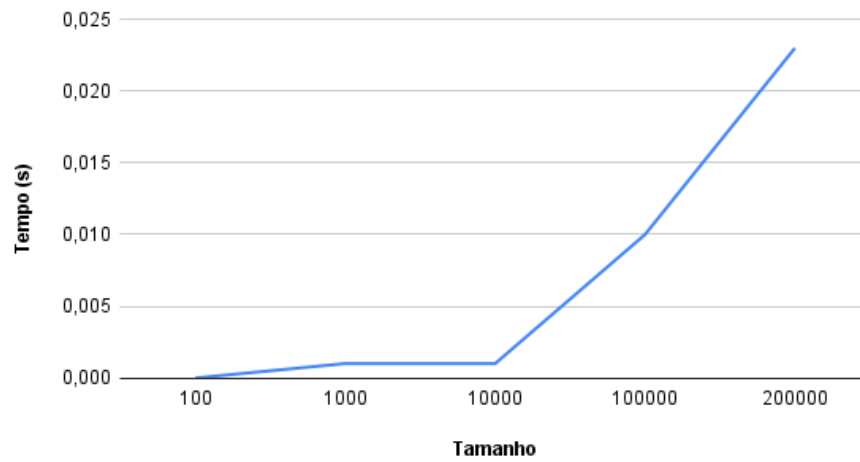
Complexidade melhor caso: $\Omega(n)$

Algoritmo:

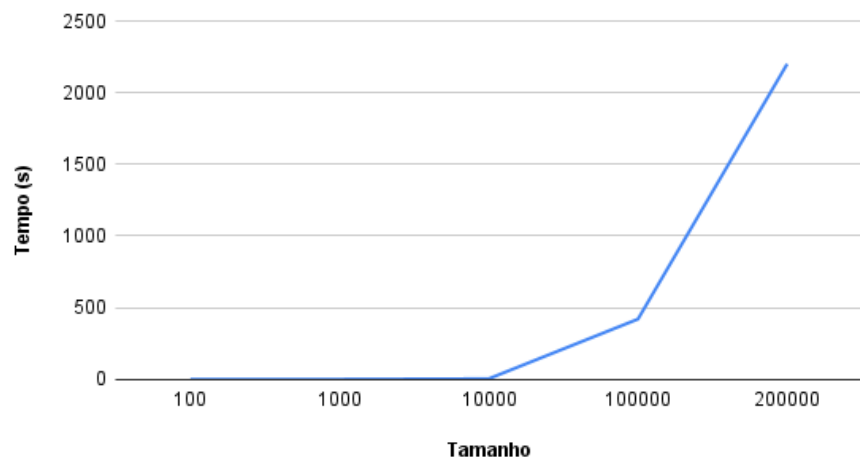
```
def insertionSort(array):  
    trocas, comp = 0, 0  
    for i in range(len(array)):  
        key = array[i]  
        j = i  
        comp += 1  
        while j > 0 and key < array[j - 1]:  
            array[j] = array[j - 1]  
            j -= 1  
            trocas += 1  
  
        array[j] = key  
  
    return [array, comp-1, trocas]
```

Resultados:

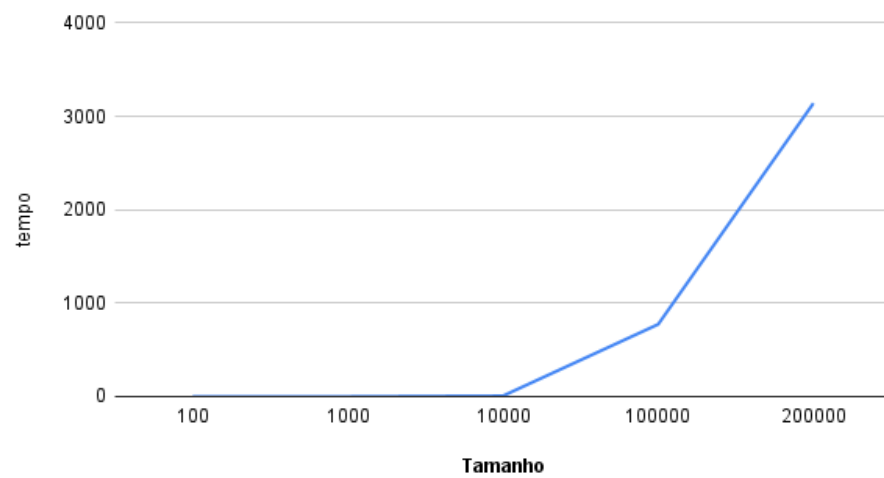
InsertionSort - Ordenado



InsertionSort - Aleatorio

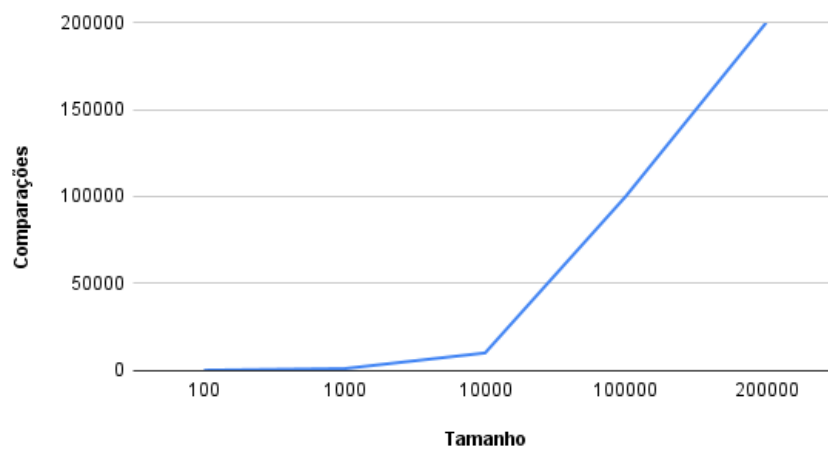


InsertionSort - Decrescente

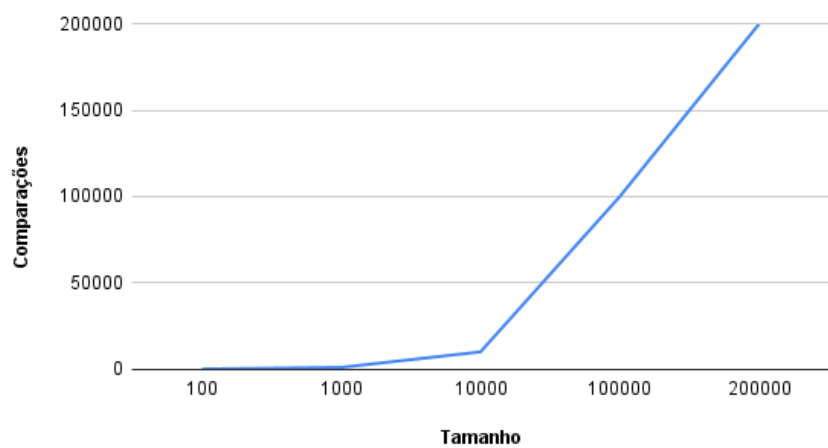


Número de comparações:

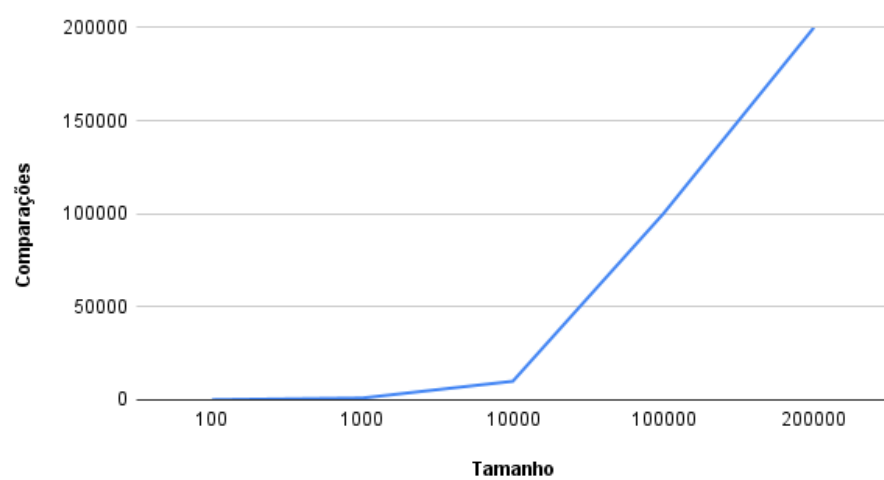
Comparações versus tamanho - Ordenado



Comparações versus tamanho - Aleatório



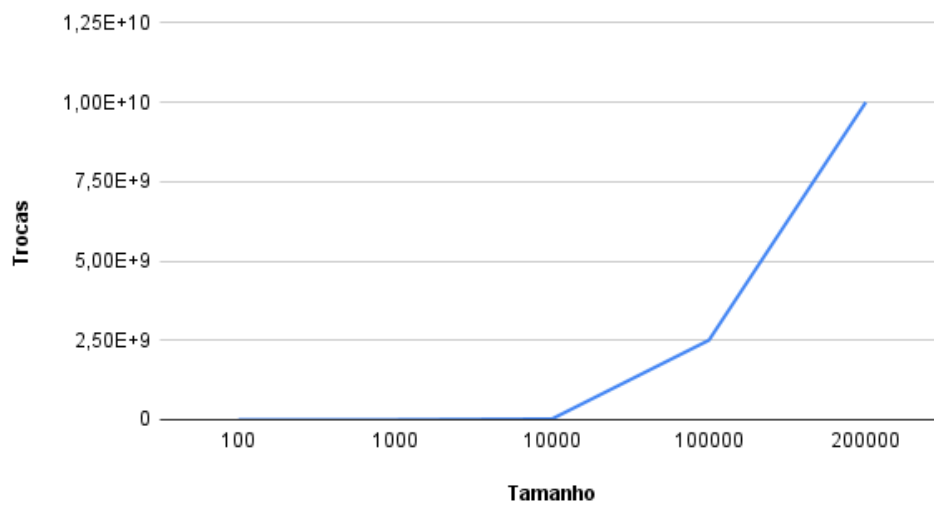
Comparações versus tamanho - Decrescente



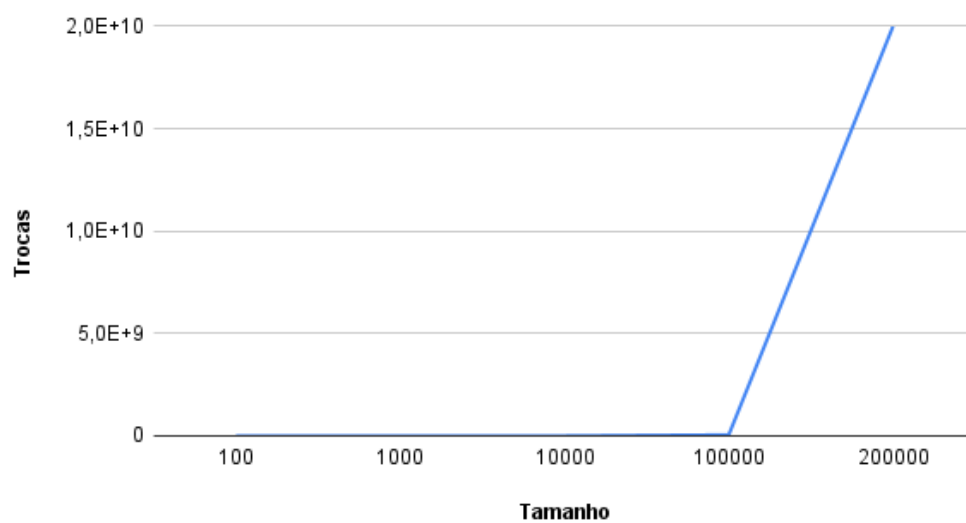
Número de trocas:

Como o vetor é ordenado, o algoritmo não faz nenhuma troca.

Trocas versus tamanho - Aleatório



Trocas versus tamanho - Decrescente

**Conclusão:**

Assim como o BubbleSort e o SelectSort, o InsertSort não é uma boa escolha para vetores muito grandes.


4. MergeSort

Complexidade pior caso: $O(n \log n)$

Complexidade caso médio: $\theta(n \log n)$

Complexidade melhor caso: $\Omega(n \log n)$

Algoritmo:



```
from numpy import arange

def mergeSort(array):
    trocas, comp = 0, 0
    if len(array) > 1:
        mid = len(array)//2

        arrayLeft = array[:mid]
        arrayRight = array[mid:]

        mergeSort(arrayLeft)
        mergeSort(arrayRight)

    i, j, k = 0, 0, 0

    while i < len(arrayLeft) and j < len(arrayRight):
        if arrayLeft[i] < arrayRight[j]:
            array[k] = arrayLeft[i]
            i += 1
        else:
            array[k] = arrayRight[j]
            j += 1
        k += 1
        comp += 1
        trocas += 1

    while i < len(arrayLeft):
        array[k] = arrayLeft[i]
        i += 1
        k += 1

        comp += 1
        trocas += 1

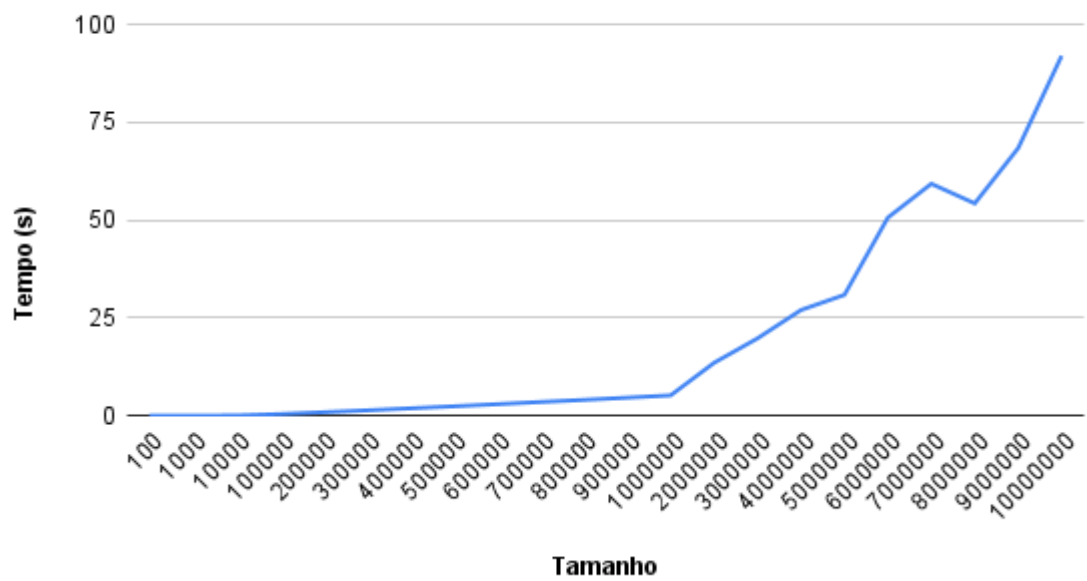
    while j < len(arrayRight):
        array[k] = arrayRight[j]
        j += 1
        k += 1

        comp += 1
        trocas += 1

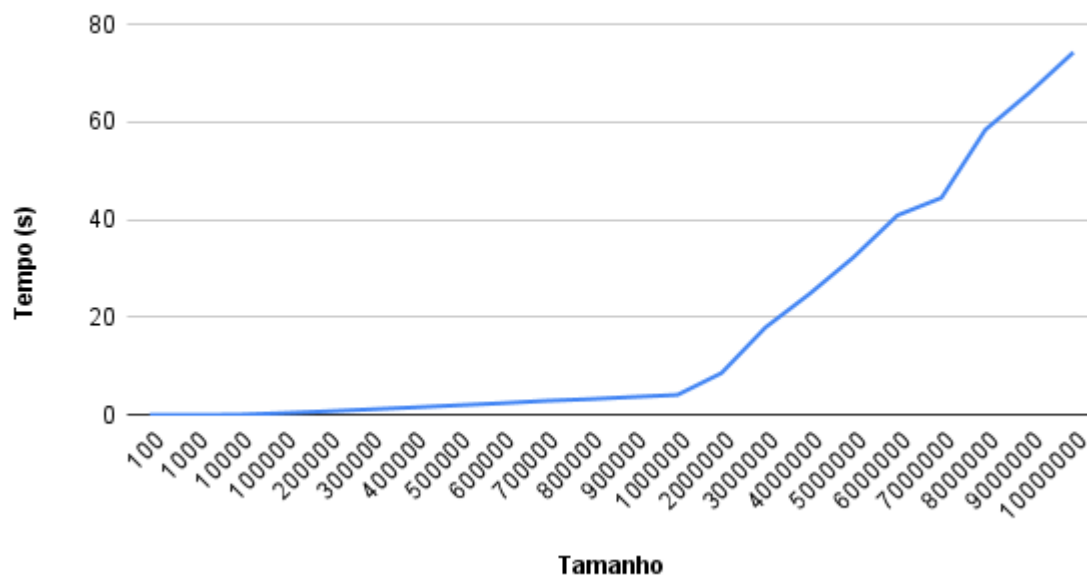
    return [array, comp-1, trocas]
```

Resultados:

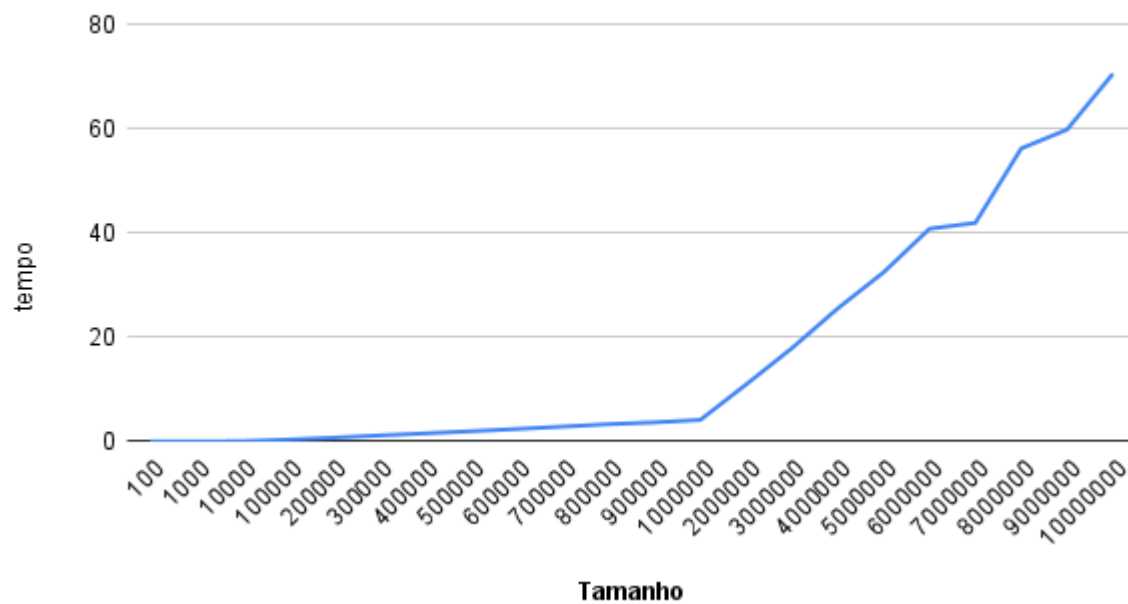
MergeSort - Aleatorio



MergeSort - Ordenado

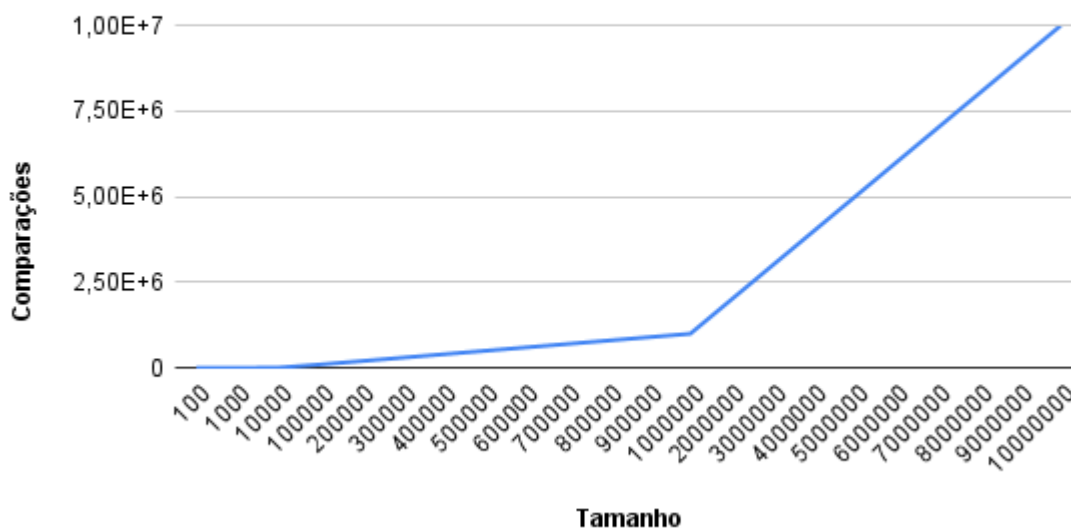


MergeSort - Decrescente

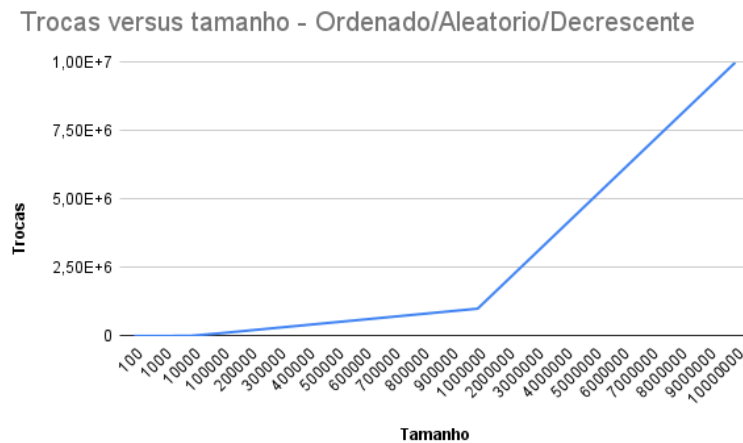


Número de comparações:

Comparações versus tamanho - Ordenado/Aleatório/Decrescente



Número de trocas:



Conclusão:

O mergeSort foi um dos algoritmos de ordenação que apresentaram o melhor resultado.

5. QuickSort

Complexidade pior caso: $O(n^2)$

Complexidade caso médio: $\theta(n^2)$

Complexidade melhor caso: $\Omega(n^2)$

Algoritmo:

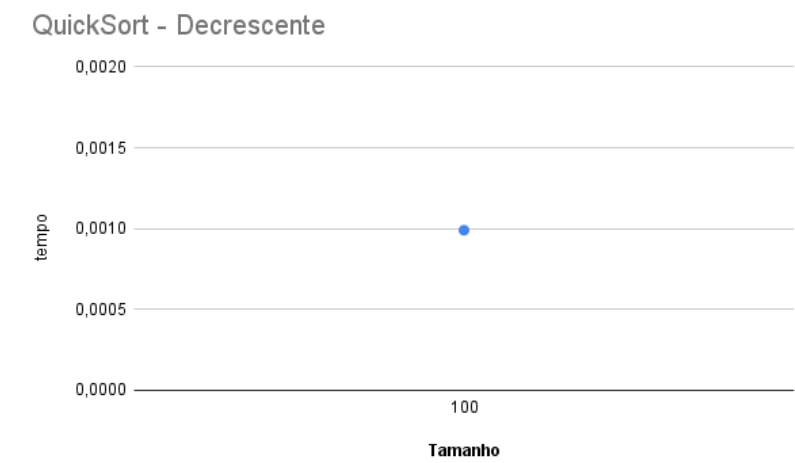
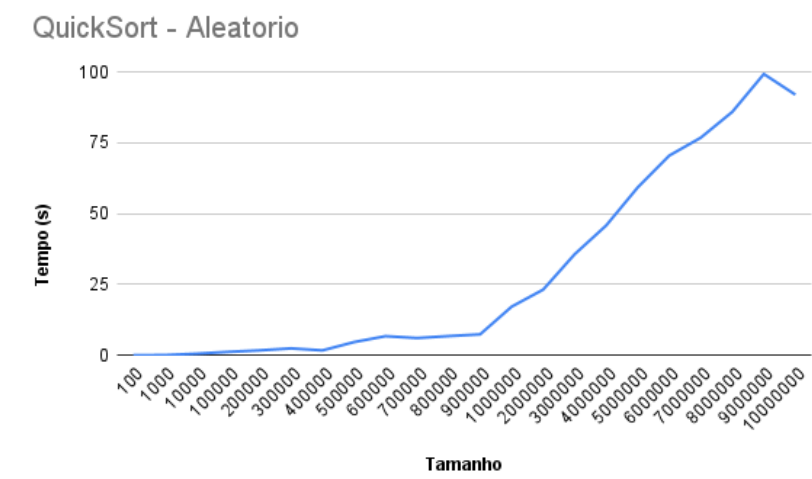
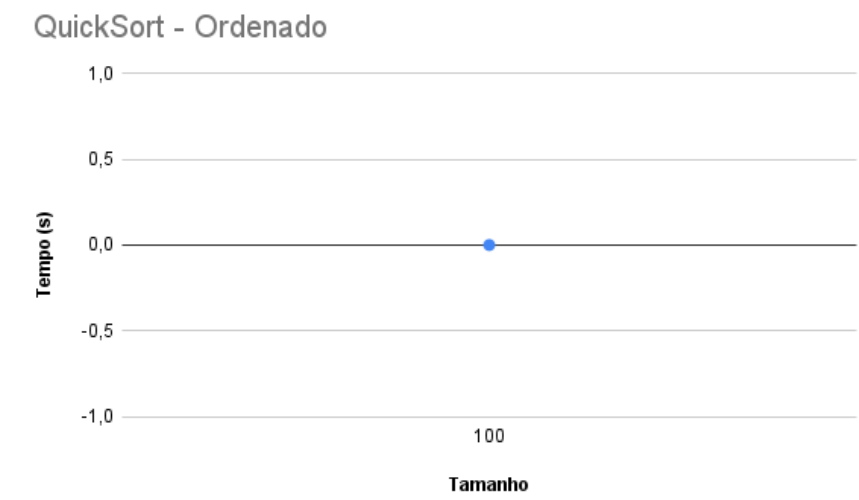
```
def particion(array, init, end):
    trocas, comp = 0, 0
    pivo = array[end-1]
    for i in range(init, end):
        comp += 1
        if array[i] > pivo:
            end += 1
        else:
            end += 1
            init += 1
            array[i], array[init - 1] = array[init - 1], array[i]
            trocas += 1

    return [init - 1, comp, trocas]

def quickSort(array, init = 0, end=None):
    trocas, comp = 0, 0
    end = end if end is not None else len(array)
    if init < end:
        output = particion(array, init, end)
        part = output[0]
        comp += output[1]
        trocas += output[2]
        quickSort(array, init, part)
        quickSort(array, part+1, end)

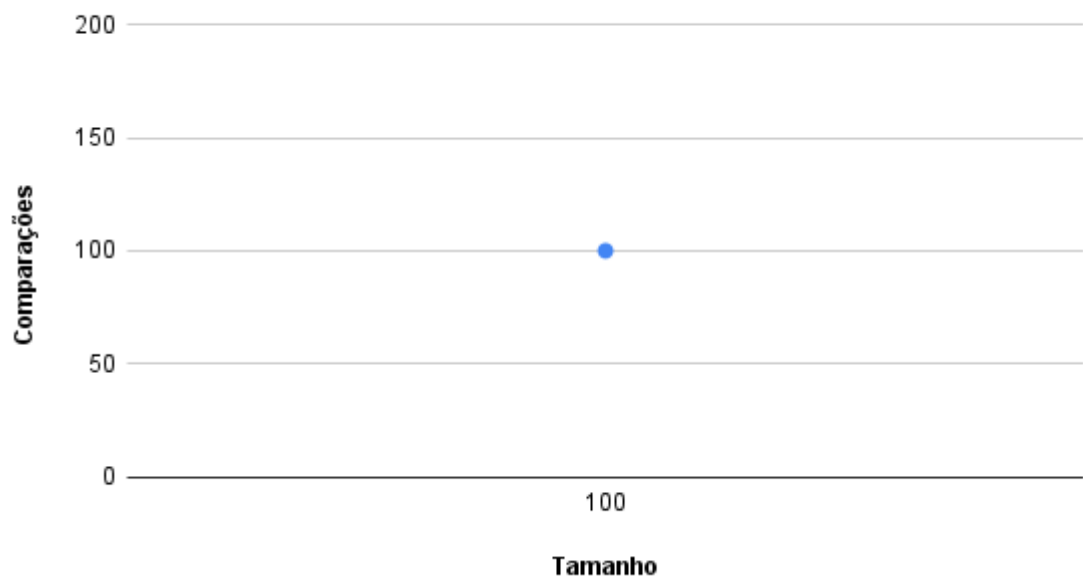
    return [array, comp, trocas]
```

Resultados:

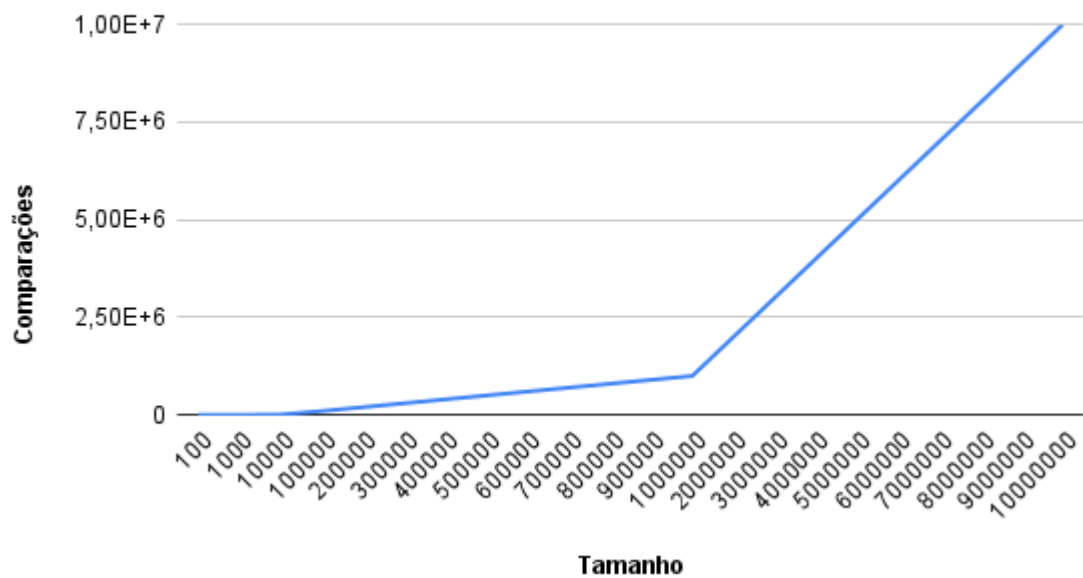


Número de comparações:

Comparações versus tamanho - Ordenado/Decrescente

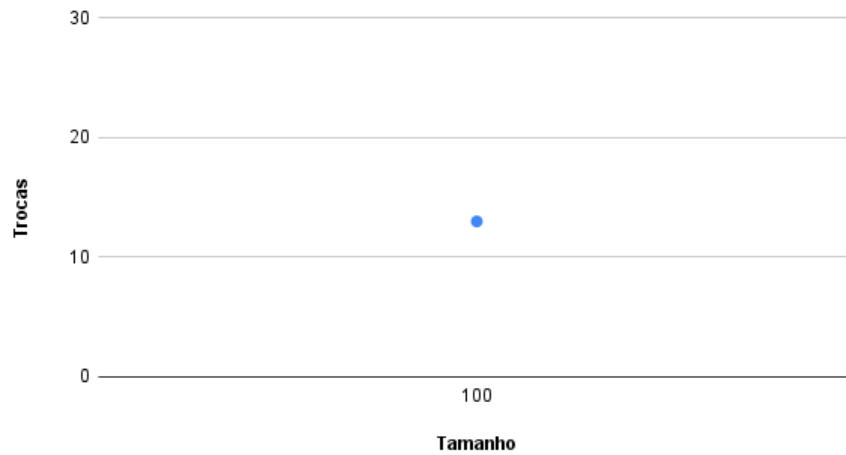


Comparações versus tamanho - Aleatório

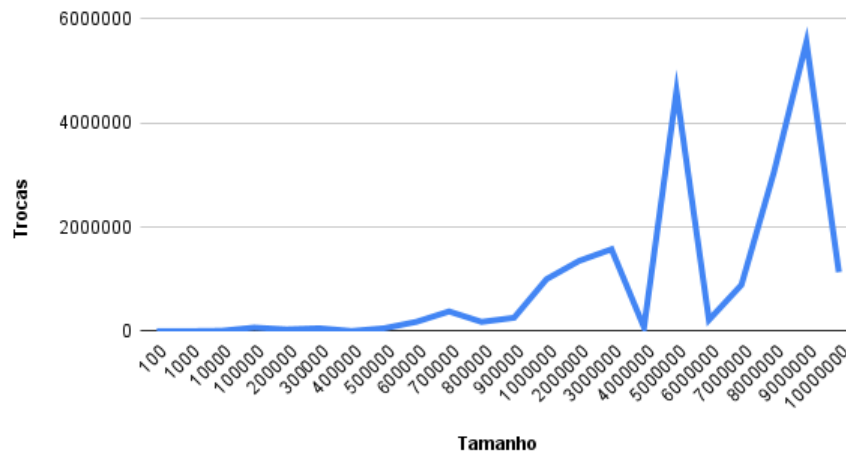


Número de trocas:

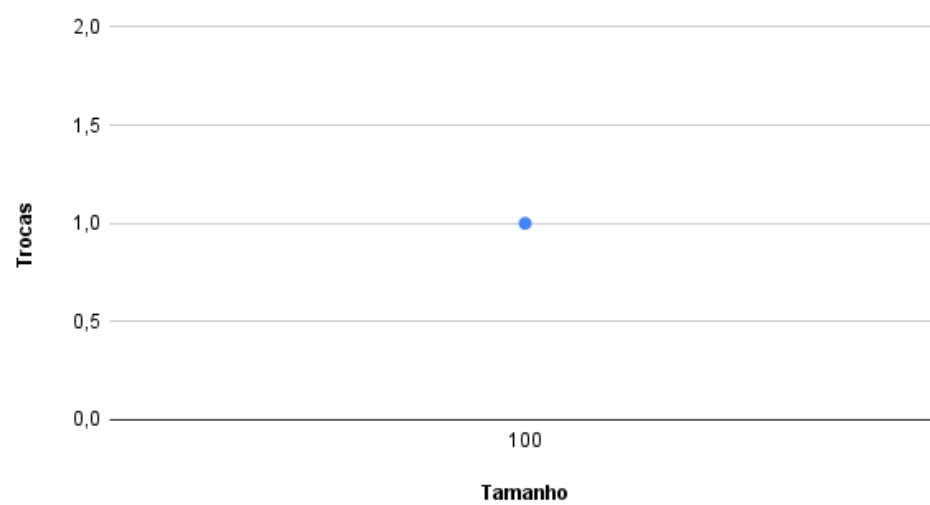
Trocas versus tamanho - Ordenado



Trocas versus tamanho - Aleatório



Trocas versus tamanho - Decrescente



Conclusão:

Assim como o MergeSort, e QuickSort é uma boa escolha para vetores muito grandes, apesar de que para vetores ordenados e em ordem decrescente, o algoritmo não conseguiu rodar com vetores maiores ou iguais a 1000.

6. BucketSort

Bucket sort, ou bin sort, é um algoritmo de ordenação que funciona dividindo um vetor em um número finito de recipientes. Cada recipiente é então ordenado individualmente, seja usando um algoritmo de ordenação diferente.

Complexidade pior caso: $O(n^2)$

Complexidade caso médio: $\theta(n)$

Complexidade melhor caso: $\Omega(n)$

Algoritmo:

```
from Algoritmos.insertionSort import insertionSort

def bucketSort(array):
    comp, trocas = 0, 0

    maxValue = max(array)
    size = maxValue/len(array)

    bucketList = [[] for i in range(len(array))]

    for i in range(len(array)):
        j = int(array[i]/size)
        comp += 1
        if j != len(array):
            bucketList[j].append(array[i])
        else:
            bucketList[len(array)-1].append(array[i])

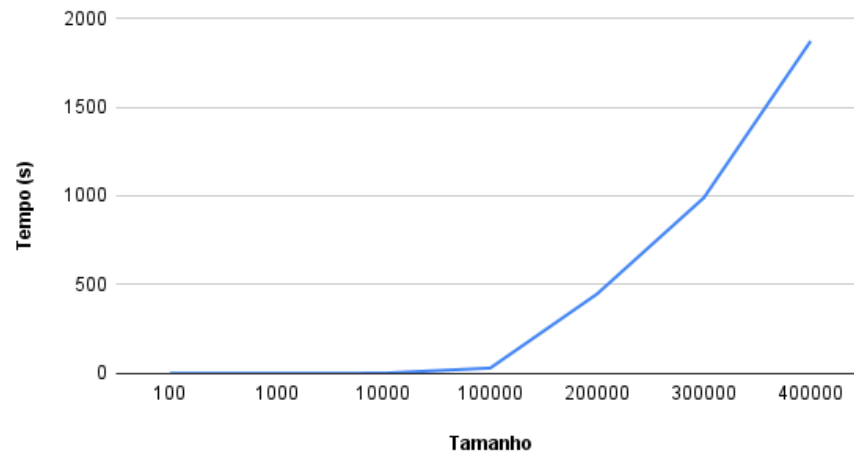
    for i in range(len(array)):
        output = insertionSort(bucketList[i])
        bucketList[i] = output[0]
        comp += output[1]
        trocas += output[2]

    outputArray = []
    for i in range(len(array)):
        outputArray = outputArray + bucketList[i]

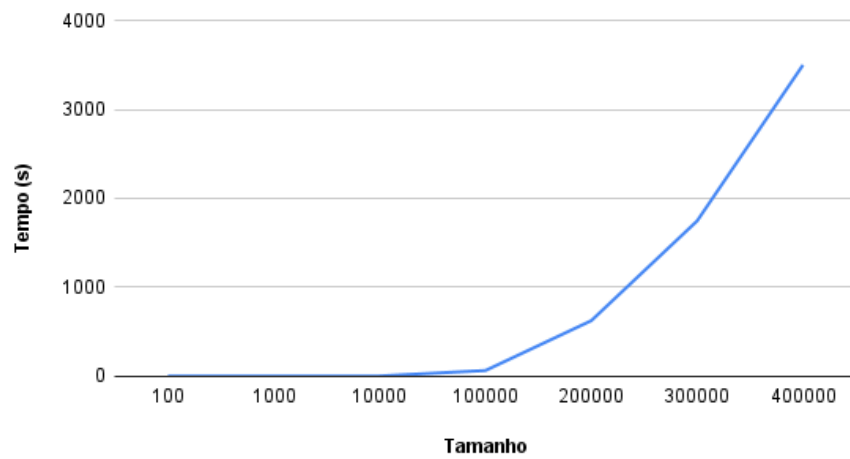
    return [outputArray, comp, trocas]
```

Resultados:

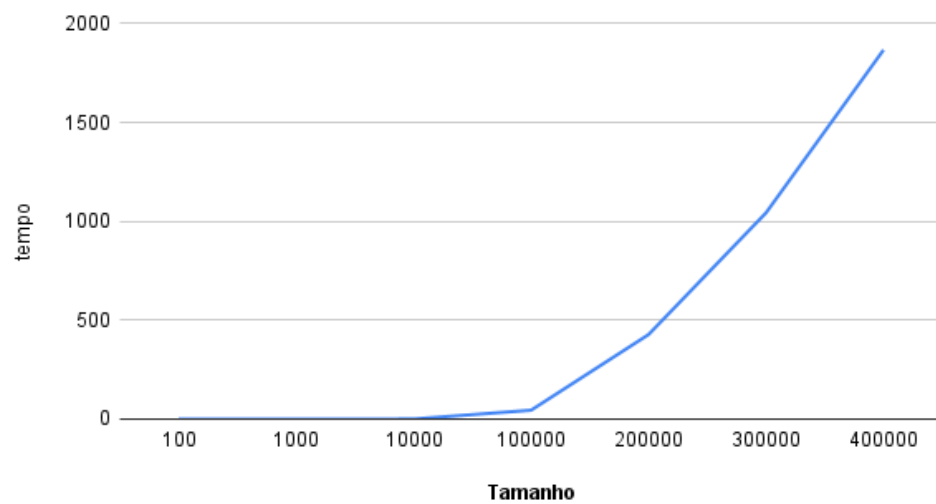
BucketSort - Ordenado



BucketSort - Aleatorio

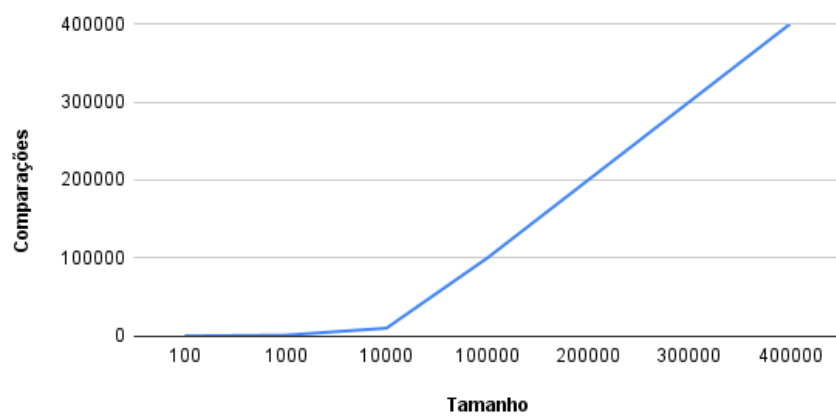


BucketSort - Decrescente



Número de Comparações:

Comparações versus tamanho - Ordenado/Aleatorio/Decrescente



7. HeapSort

O algoritmo heapsort é um algoritmo de ordenação generalista, e faz parte da família de algoritmos de ordenação por seleção.

O heapsort utiliza uma estrutura de dados chamada heap, para ordenar os elementos à medida que os insere na estrutura. Assim, ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da heap, na ordem desejada. A heap pode ser representada como uma árvore (uma árvore binária com propriedades especiais) ou como um vetor.

Complexidade pior caso: $O(n^2)$

Complexidade caso médio: $\theta(n^2)$

Complexidade melhor caso: $\Omega(n^2)$

Algoritmo:

```
def heapify(array, length, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < length and array[largest] < array[left]:
        largest = left

    if right < length and array[largest] < array[right]:
        largest = right

    if largest != i:
        array[i], array[largest] = array[largest], array[i]
        heapify(array, length, largest)

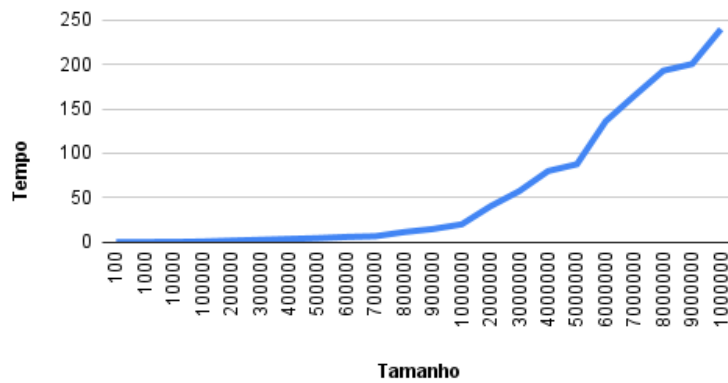
def heapSort(array):
    for i in range(len(array)//2 - 1, -1, -1):
        heapify(array, len(array), i)

    for i in range(len(array)-1, 0, -1):
        array[i], array[0] = array[0], array[i]
        heapify(array, i, 0)

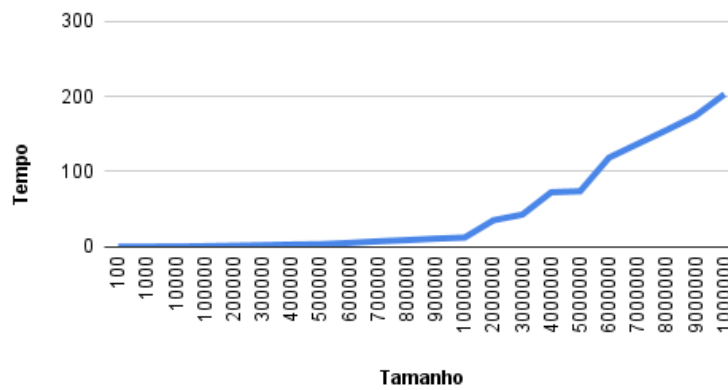
    return [array, 0, 0]
```


Resultados:

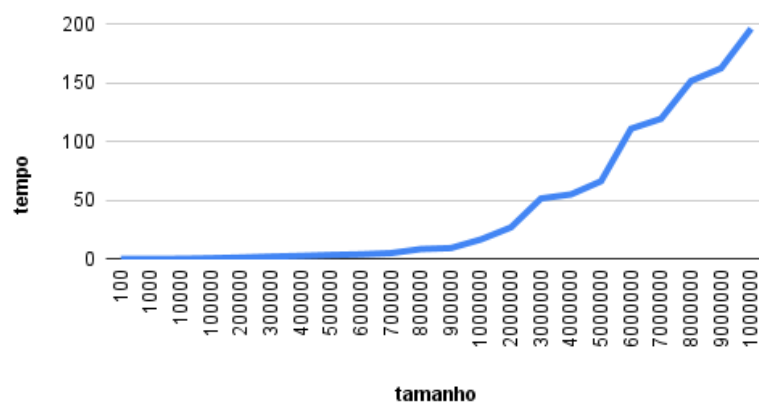
HeapSort - Aleatorio



HeapSort - Ordenado



HeapSort - Decrescente



Número de Comparações: 0

Número de Trocas: 0

8. CountingSort

Counting sort é um algoritmo de ordenação estável cuja complexidade é $O(n^2)$. As chaves podem tomar valores entre 0 e $M-1$. A ideia básica do counting sort é determinar, para cada entrada x , o número de elementos menor que x . Essa informação pode ser usada para colocar o elemento x diretamente em sua posição no array de saída. Por exemplo, se há 17 elementos menores que x , então x pertence a posição 18. Esse esquema deve ser ligeiramente modificado quando houver vários elementos com o mesmo valor, uma vez que nós não queremos que sejam colocados na mesma posição.

Complexidade pior caso: $O(n)$

Complexidade caso médio: $\theta(n)$

Complexidade melhor caso: $\Omega(n)$

Algoritmo:

```
import numpy as np

def countingSort(array):
    max = np.max(array) + 1
    count = [0] * max

    for i in array:
        count[i] += 1

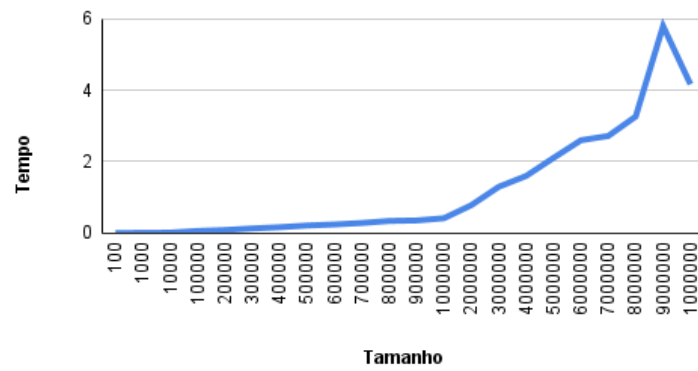
    for i in range(1, max):
        count[i] += count[i-1]

    outputArray = [0] * len(array)
    i = len(array) - 1
    while i >= 0:
        elem = array[i]
        count[elem] -= 1
        newPosition = count[elem]
        outputArray[newPosition] = elem
        i -= 1

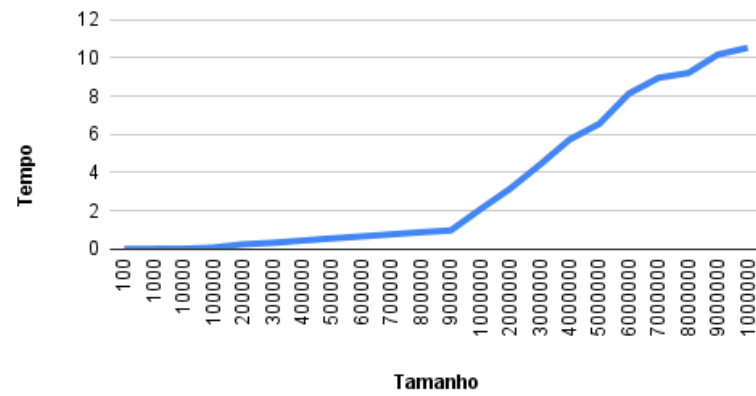
    return [outputArray, ' ', ' ']
```

Resultados:

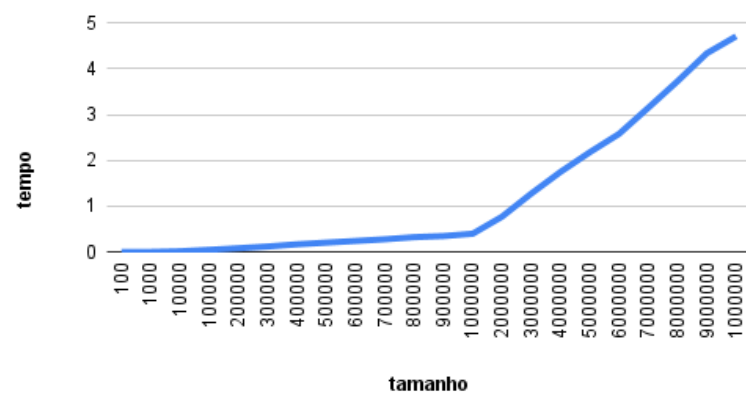
ContingSort - Ordenado



CountingSort - Aleatorio



CountingSort - Decrescente



9. Conclusão

Como podemos ver os algoritmos BubbleSort, SelectSort e InsertSort não são ideais para ordenação de conjuntos grandes, para ordenação de grandes conjuntos podemos citar o MergeSort e o CountingSort, onde o tempo de ordenação de grandes conjuntos é muito pequeno.