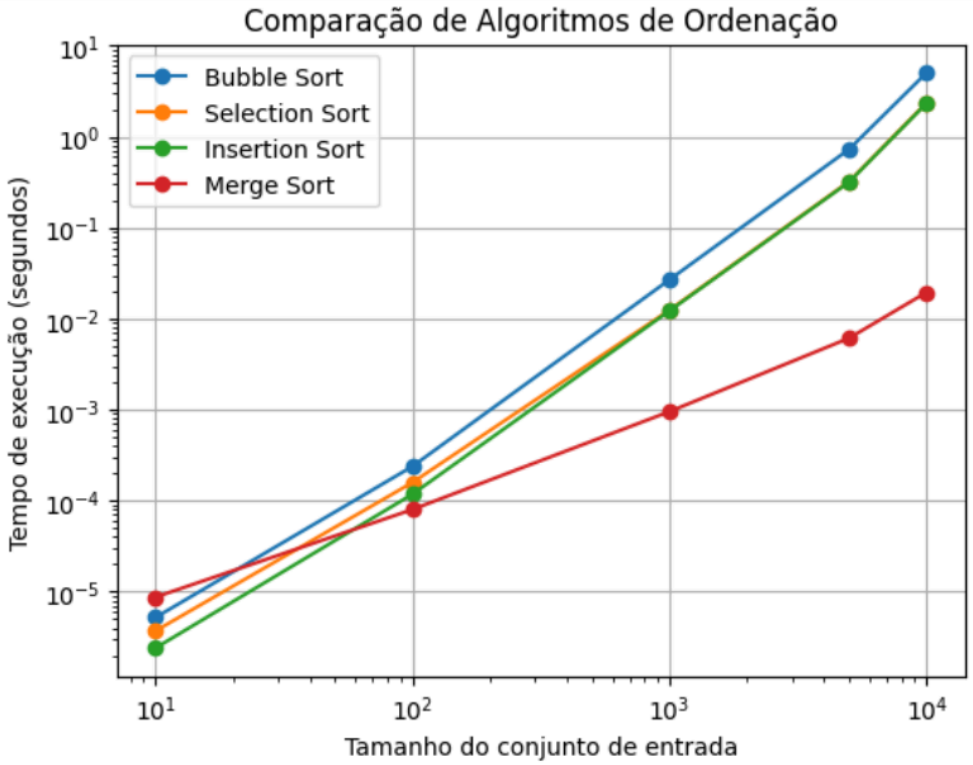
Apresente resultados visuais, como gráficos e tabelas, para demonstrar o desempenho do algoritmo em diferentes cenários.

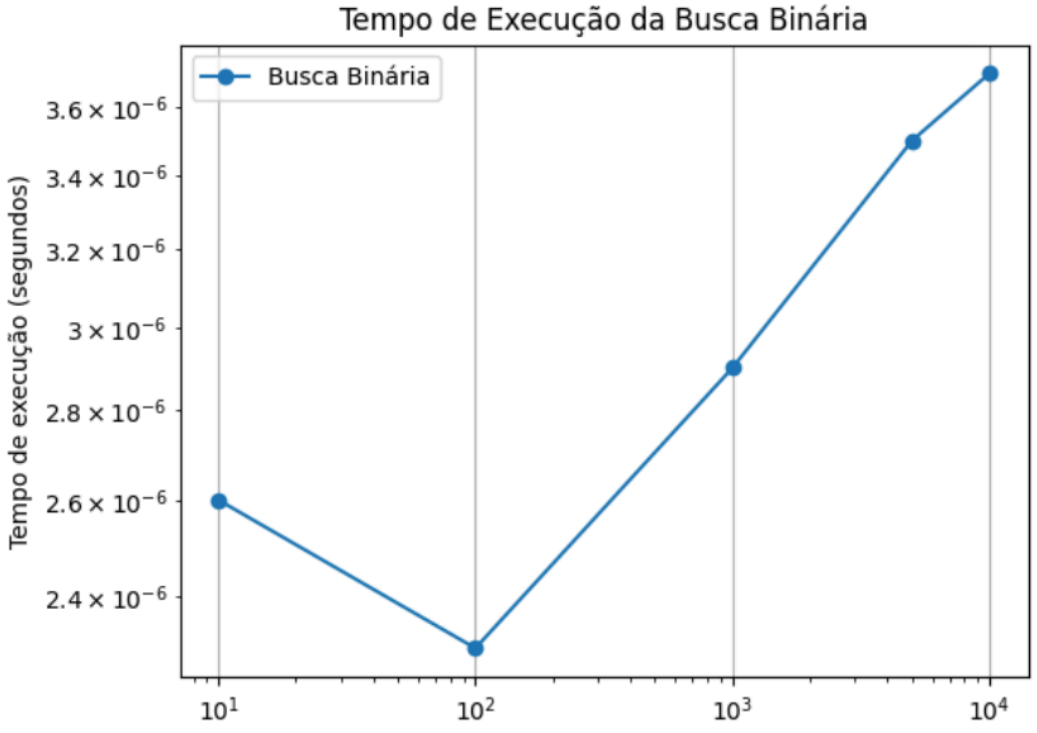
```
import time
import random
import matplotlib.pyplot as plt

def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

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

def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]
        merge_sort(L)
        merge_sort(R)
```

```
        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
    return arr

def binary_search(arr, target):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
def test_sorting_algorithm(algorithm, arr):
    start_time = time.perf_counter_ns()
    sorted_arr = algorithm(arr.copy())
    end_time = time.perf_counter_ns()
    return end_time - start_time, sorted_arr
sizes = [10, 100, 1000, 5000, 10000]
algorithms = {
    "Bubble Sort": bubble_sort,
    "Selection Sort": selection_sort,
    "Insertion Sort": insertion_sort,
    "Merge Sort": merge_sort,
}
```

```
results = {size: {} for size in sizes}

for size in sizes:
    arr = random.sample(range(size * 10), size)
    for name, algorithm in algorithms.items():
        execution_time, _ = test_sorting_algorithm(algorithm, arr)
        if execution_time == 0:              execution_time = 1e-10
        results[size][name] = execution_time / 1e9
binary_search_results = {}

for size in sizes:
    sorted_arr = sorted(random.sample(range(size * 10), size))
    target = sorted_arr[random.randint(0, size - 1)]
    start_time = time.perf_counter_ns()
    index = binary_search(sorted_arr, target)
    end_time = time.perf_counter_ns()
    execution_time = end_time - start_time
    if execution_time == 0:          execution_time = 1e-10
    binary_search_results[size] = execution_time / 1e9
fig, ax = plt.subplots()
for name in algorithms:
    times = [results[size][name] for size in sizes]
    ax.plot(sizes, times, marker='o', label=name)
ax.set_xlabel('Tamanho do conjunto de entrada')
ax.set_ylabel('Tempo de execução (segundos)')
ax.set_title('Comparação de Algoritmos de Ordenação')
ax.legend()
ax.grid(True)
plt.xscale('log')
plt.yscale('log')
plt.show()
fig, ax = plt.subplots()
binary_search_times = [binary_search_results[size] for size in sizes]
ax.plot(sizes, binary_search_times, marker='o', label='Busca Binária')
ax.set_xlabel('Tamanho do conjunto de entrada')
ax.set_ylabel('Tempo de execução (segundos)')
ax.set_title('Tempo de Execução da Busca Binária')
ax.legend()
ax.grid(True)
plt.xscale('log')
plt.yscale('log')
plt.show()
```

Resultado:


Comparação de Algoritmos de Ordenação
Bubble Sort
Selection Sort
Insertion Sort
Merge Sort
Tempo de execução (segundos)
Tamanho do conjunto de entrada


Tempo de Execução da Busca Binária
Busca Binária
Tempo de execução (segundos)