UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI
CIÊNCIA DA COMPUTAÇÃO

Comparação de Algoritmos de Ordenação

Projeto e Análise de Algoritmos

Bárbara Boechat

São João del-Rei
Fevereiro de 2021

Sumário

1 Introdução

A ordenação é uma maneira de reorganizar um conjunto de objetos a partir de uma ordem ascendente ou descendente e tem como objetivo facilitar a recuperação de itens do conjunto ordenado. Assim para testar os métodos de ordenação é necessário que o computador realize a execução desses métodos para enfim verificar seu desempenho.

Como existem diversos métodos de ordenação neste trabalho serão realizadas comparações de desempenho entre os algoritmos **MergeSort**, **InsertionSort** e **TimSort** considerando o tempo e o número de comparações feitas para ordenar completamente listas de diferentes tamanhos que seguem três diferentes padrões: ordenada ascendente, ordenada descendente e aleatório.

2 InsertionSort

O **InsertionSort** consiste em ordenar os itens inserindo-os na posição corresponde da lista, nessa estratégia um valor 'chave' é comparado com os outros itens até que a posição correta seja encontrada. Essa comparação é feita em direção à esquerda, comparando a 'chave' e o antecessor.

Se o 'item' comparado for menor, a lista deve ser deslocada para a direita, visando "abrir" um novo espaço para colocar a 'chave' na posição correspondente; finalmente ao encontrar um 'item' maior ou não haver mais itens, significa que foi encontrada a posição que este 'item' deve estar, ele ser posicionado corretamente na posição correspondente. Como mostrado no diagrama da figura 1.

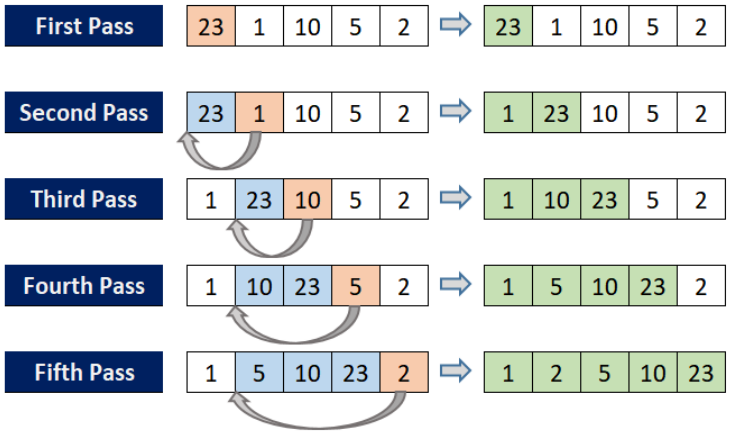

Figura 1: Diagrama InsertionSort

2.1 Complexidade

2.1.1 Melhor Caso

O Insert é mais rápido e eficiente quando utilizado para ordenar listas pequenas, em que o número de comparações é menor e quando a lista já está ordenada, pois assim apenas é necessário percorrer a lista n — 1 vezes, ou seja, todos os seus elementos exceto o primeiro. Neste caso, a ordem de complexidade é O(n).

2.1.2 Caso Médio

Este caso é uma média entre todas as entradas possíveis. Sendo assim, a ordem de complexidade é $O(n^2)$.

2.1.3 Pior Caso

O pior caso ocorre quando os elementos da lista estão em ordem decrescente, pois o laço interno realizará a quantidade máxima de iterações, de forma que a 'chave' sempre será menor o antecessor. Neste caso, a ordem de complexidade é $O(n^2)$.

3 MergeSort

O **MergeSort** é um algoritmo que utiliza do ideal "dividir e conquistar", assim, ele recursivamente dissolve o problema em subproblemas, até que eles se tornem simples o suficiente para serem resolvidos diretamente. As soluções dos subproblemas são então combinadas de modo a gerar a solução completa do problema original. De forma geral, o algoritmo divide a lista em partes iguais, as ordena e as recombina ordenadamente para gerar a solução de fato. O fluxograma na figura 2 exemplifica o processo realizado pelo algoritmo.

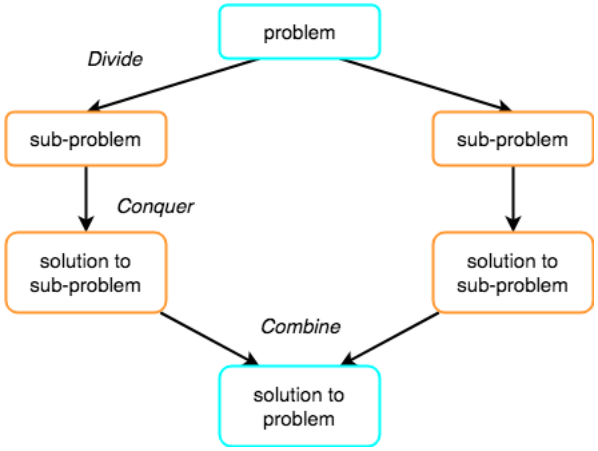

Figura 2: Fluxograma MergeSort

3.1 Complexidade

O Merge é mais rápido e eficiente quando utilizado para ordenar listas grandes ou grandes quantidades de dados. Neste algoritmo o Melhor Caso, Caso Médio e Pior Caso iguais, pois realizá etapas que contribuem para o cálculo da complexidade da seguinte maneira:

- Operação de divisão pela metade da lista principal: Máximo log(n + 1);
- Operação de etapa única para descobrir o meio do subarray: O(1);
- Operação para realizar o merge dos subarrays: O(n).

Portanto, o tempo total se tornará n (log n + 1), o que resulta numa complexidade de tempo de O (n * log n).

4 TimSort

O **TimSort** foi inventado por Tim Peters em 2002 para ser usado na linguagem de programação Python, e tem sido o algoritmo de ordenação padrão de Python desde a versão 2.3. Além disso, também foi escolhido como algoritmo de ordenação padrão do Java por ser bastante eficiente se comparado aos demais algoritmos existentes. Ele é baseado no MergeSort e InsertionSort sendo considerado um algoritmo de ordenação híbrido, uma combinação eficiente de outros algoritmos.

4.1 Divisão de Runs

Este algoritmo aproveita os sub-vetores já ordenados a seu favor, sejam eles ascendentes ou descendentes, primeiro é definido um tamanho de minrun, neste trabalho seu tamanho é 32 para listas de tamanhos maiores ou iguais a 32. Este valor arbitrário serve para garantir que todos os sub-vetores, as runs, tenham um comprimento mínimo.

Durante a divisão das runs, o algoritmo adota um valor de indexação médio na porção analisada da lista original, assim, de acordo a chave deste index, o item atual será adicionado ao lado esquerdo ou direito da run. Após a formação das runs o algoritmo está pronto para prosseguir e utilizar o InsertionSort, que é especialmente eficiente para ordenar vetores pequenos e ou já ordenados.

4.2 InsertionSort com Busca Binária

Como um de seus algoritmos o TimSort utiliza-se do InsertionSort como método de ordenação de suas runs, como dito anteriormente, este algoritmo, tratando-se de listas pequenas e ou já ordenadas, ou seja, em seu melhor caso sua ordem de complexidade é O(n). Contudo, neste trabalho foi adotada a Busca Binária como método de otimização, ela consiste em localizar o ponto médio de um vetor e, caso o item seja maior ou menor que este ponto médio, deve-se escolher a metade apropriada para prosseguir a busca, dividindo essa metade e repetindo o processo recursivamente até que seja encontrado o valor correto, como mostrado na figura 3.

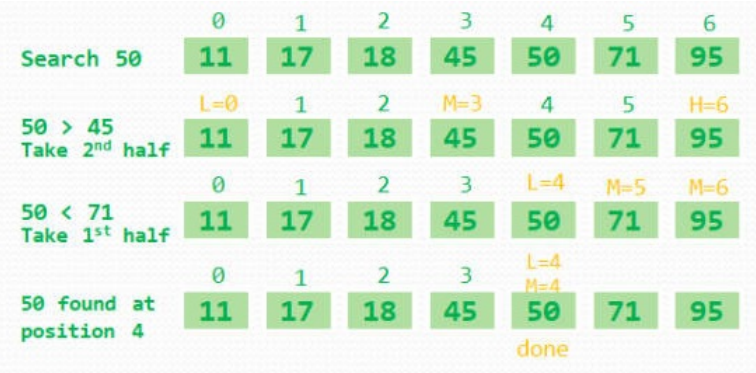

Figura 3: Busca Binária

4.3 Merge das Runs

Após a criação e ordenação das runs, deve-se utilizar do MergeSort para juntá-las, exemplo na figura 4. Na melhor das hipóteses as runs já estarão completamente ordenadas e o algoritmo irá terminar. Neste trabalho a ordem das runs que serão unidas pelo merge não segue o modelo de pilha necessariamente, mas a ordem normal da lista principal. O merge será guiado por valores de indexação que indicam a esquerda, meio e direita de uma lista de tamanho 2 vezes maior que uma run a ser unida, avançando assim na lista principal até seu final. A medida que esse laço avança o tamanho dos valores de indexação irão dobrar, fazendo com que listas cada vez maiores sejam unidas em cada passo.

4.4 Galloping

Figura 4: Merge de duas Runs

4.5 Complexidade

4.5.1 Melhor Caso

O melhor caso é uma lista já ordenada, pois assim apenas é necessário percorrer a lista n — 1 vezes, ou seja, todos os seus elementos exceto o primeiro. Neste caso, a ordem de complexidade Ω(n) análoga a ordem de complexidade do melhor caso do InsertionSort.

4.5.2 Caso Médio e Pior Caso

A análise de complexidade do Timsort é o mesmo que do Mergesort, sendo este caso uma média entre todas as entradas possíveis a ordem de complexidade bem como do pior caso é (nlogn).

5 Resultados

5.1 Método Adotado

Os algoritmos foram comparados sob a ótica do número de comparações feitas para ordenar uma lista e o tempo levado para realizar essa tarefa. Além disso, eles foram sujeitos a entradas de diferentes tamanhos e ordenação, como mostrado na tabela 1.

Tamanho da Entrada	32	64	1024	10K	100K	1M
Ordenação da Entrada	Aleatória	Crescente	Decrescente			

Tabela 1: Tabela de Parâmetros

Para cada tamanho de entrada foram gerados 10 arquivos diferentes, assim cada algoritmo inicialmente ordenou um arquivo aleatório, o resultado ordenado passou a ser a entrada ordenada crescente e a seguir a saída ordenada foi invertida para se tornar a entrada ordenada decrescente. Afim de medir o número de comparações realizadas por cada algoritmo um contador foi adicionado ao código para que fosse contabilizada cada comparação entre diferentes valores da lista, sendo essa comparação verdadeira ou falso foi somado um ponto ao contador. Por fim, todas as execuções geraram resultados que foram processados para gerar os gráficos dos resultados acerca do número de comparações e tempo de execução dos algoritmos InsertionSort, MergeSort e TimSort.

5.2 Análise de Tempo e Comparações

Todos os testes foram executados nas configurações: Processador: WSL2; Intel Core i3-9100F; Card Gráfico: NVIDIA GTX 1660 Super; 16GB RAM. A seguir os resultados:

Na figura 5 para a entrada ordenada ascendente o algoritmo que fez menos comparações foi o InsertionSort mesmo para as maiores entradas, o que já era esperado levando em consideração sua análise de complexidade, pois esse tipo de entrada é o seu melhor caso.

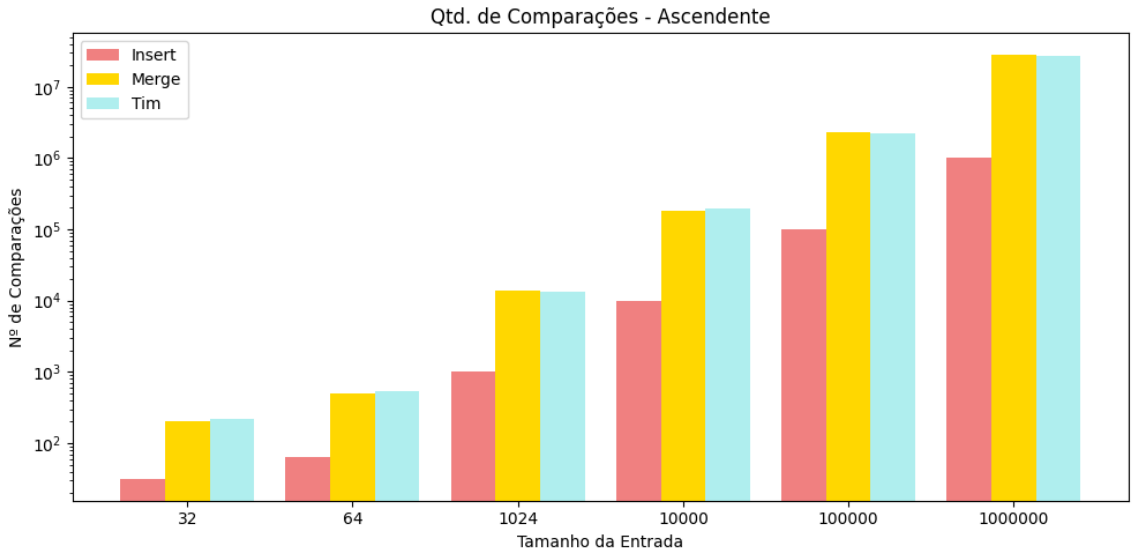

Figura 5: Resultado das Comparações - Entrada Ascendente

O baixo número de comparações realizadas pelo Insert reflete no tempo de execução necessário para ordenar a entrada ascendente, sendo assim mais rápido que os outros dois algoritmos, dentre os quais o Merge demorou mais tempo que seu concorrente Tim mesmo tendo feito menos comparações em alguns casos, como apontado pela figura 6.

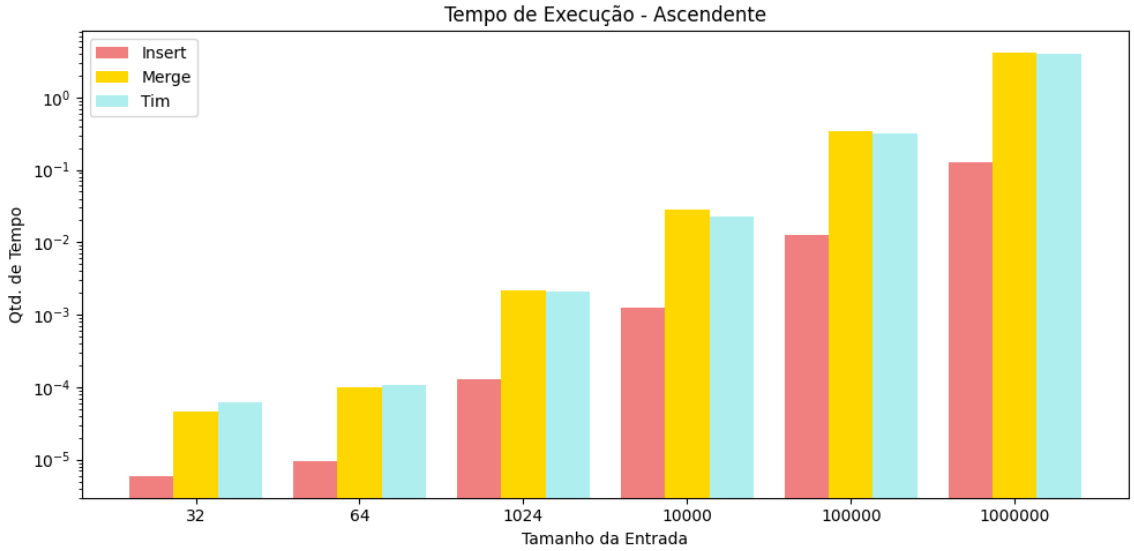

Figura 6: Resultado de Tempo - Entrada Ascendente

Já para a entrada ordenada descendente (figura 7) o InsertionSort realizou o maior número de comparações em relação ao Merge e ao Tim que fizeram um número de comparações bem próximo. Este resultado já era esperado para o Insertion por conta de se tratar do seu pior caso.

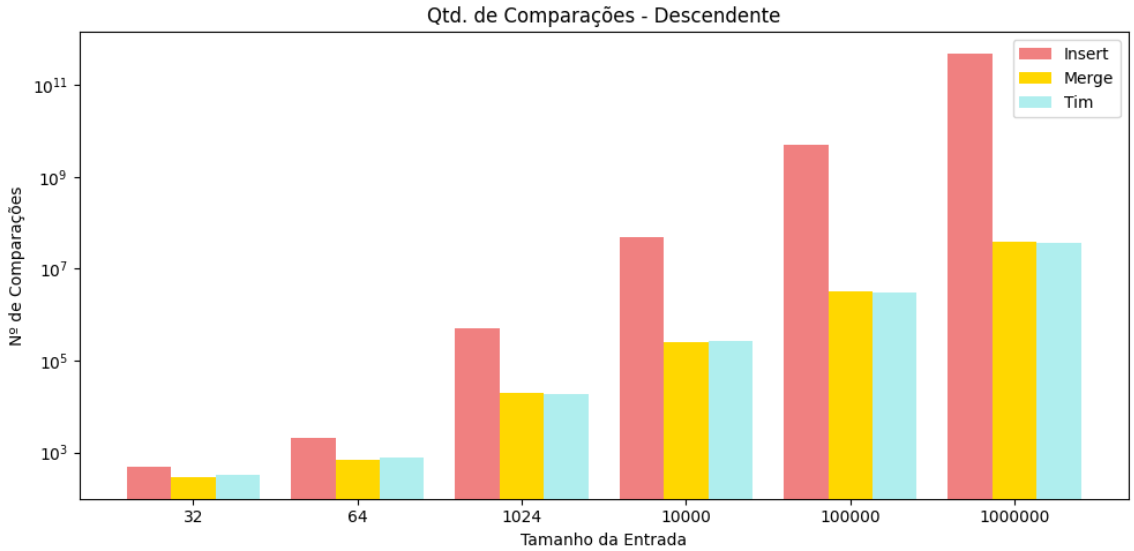

Figura 7: Resultado das Comparações - Entrada Descendente

O número de comparações realizadas pelo Insert em seu pior caso é expressivamente refletido no tempo de execução. Assim, como mostra a figura 8, de maneira análoga ao gráfico de comparações, seu tempo de execução é consideravelmente maior do que para os outros dois algoritmos que mantiveram uma média bem próxima principalmente para as entradas maiores.

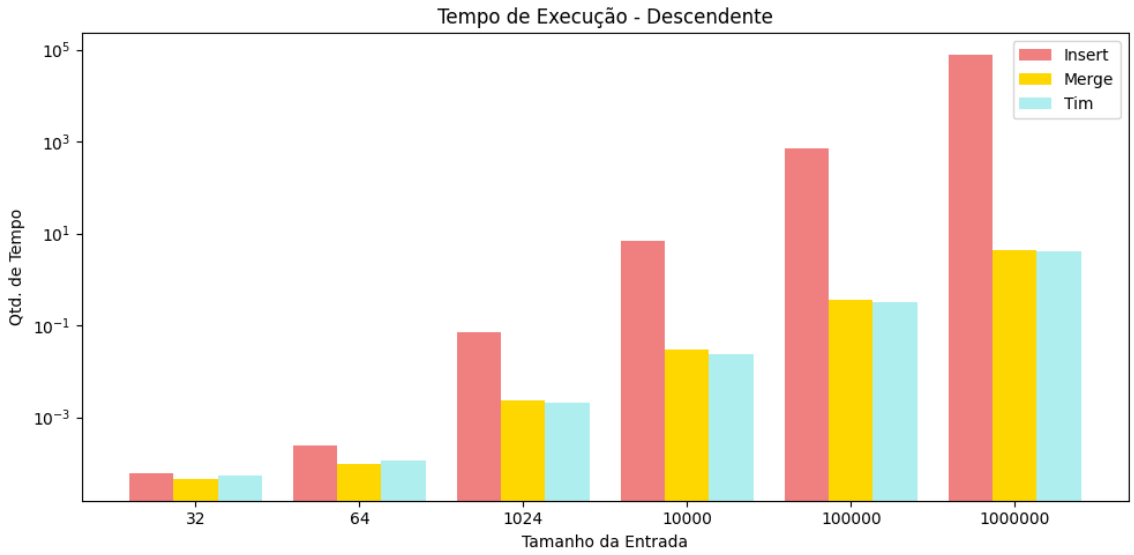

Figura 8: Resultado de Tempo - Entrada Descendente

Para a entrada aleatória o maior número de comparações, como mostra a figura 9, também foi realizado pelo InsertionSort, principalmente para as entradas maiores. Entre Merge e Tim o número de comparações é bem próximo, porém o segundo alcançou um número de comparações superior ao do primeiro especialmente na entrada de tamanho 10.000.

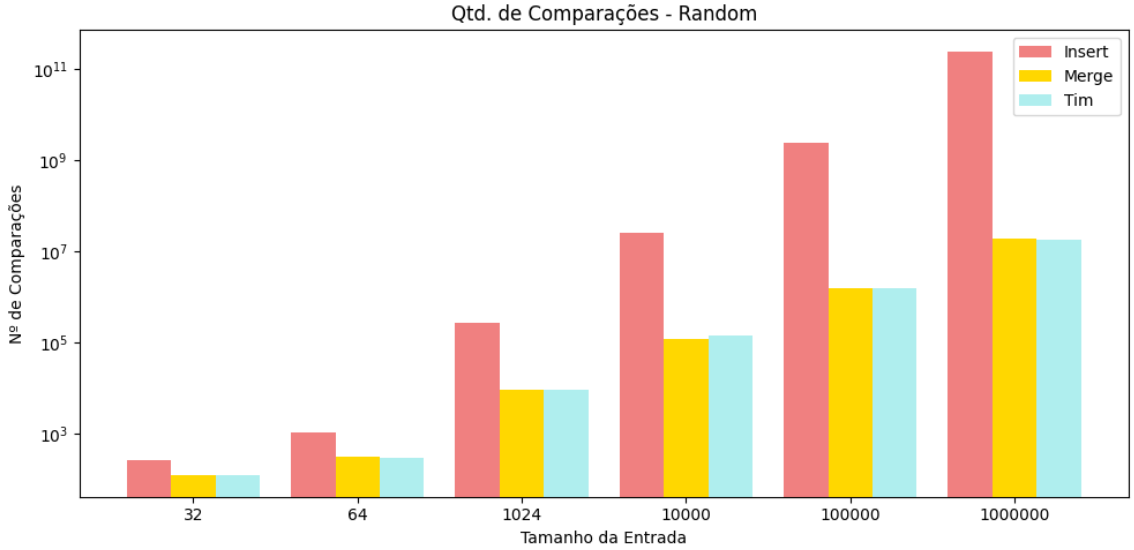

Figura 9: Resultado das Comparações - Entrada Aleatória

O tempo de execução para a entrada aleatória (figura 10) é bem parecido com o que ocorreu para o Insert na entrada decrescente (figura 8) que marcou o maior tempo especialmente para os maiores vetores de números. Porém, contrário as outras entradas, o Tim levou menos tempo que o Merge para entradas pequenas, diferença essa que diminuiu a medida que as entradas aumentaram.

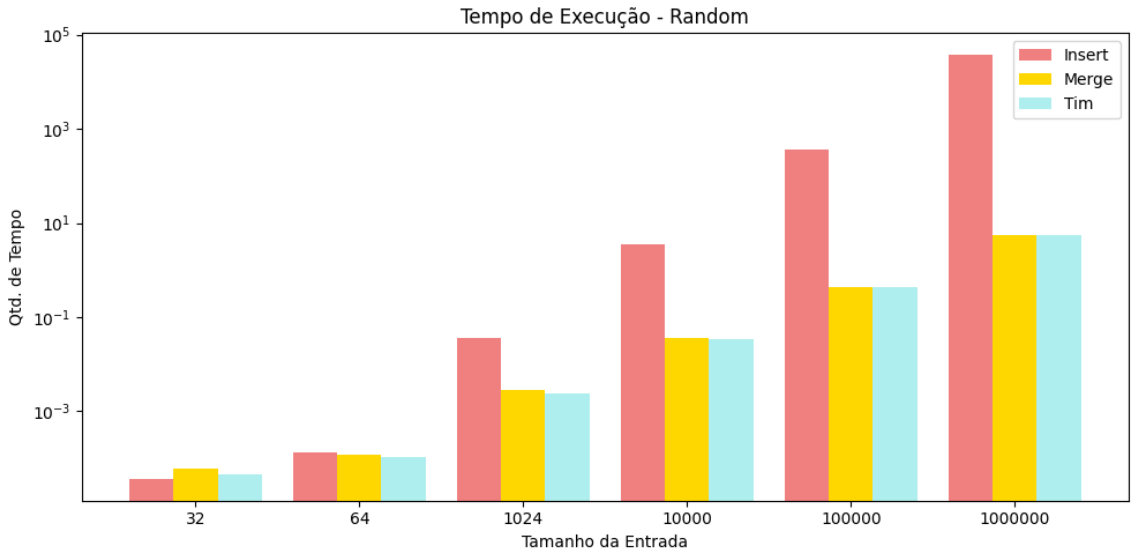

Figura 10: Resultado de Tempo - Entrada Aleatória

6 Conclusão

Defronte os resultados apresentados, é evidente que o algoritmo InsertionSort obteve os piores resultados em relação ao MergeSort e o TimSort para entradas ordenadas decrescente e aleatória de todos os tamanhos. Porém, é preciso destacar que os algoritmos MergeSort e TimSort obtiveram resultados similares para todas as entradas tanto em número de comparações quanto em tempo de execução, salvo alguns tamanhos de entrada em que ora o primeiro obteve melhor desempenho ora o segundo. Assim, é possível concluir que nenhum dos resultados apresentados entrou em conflito com o que já havia sido previamente analisado para esses algoritmos e apenas reforçou o precedente do TimSort e seu bom desempenho.