Aluna: Catherine M. Cavalcanti de Souza

Tema: Tree Sort

1. Introdução teórica

O Tree Sort é um algoritmo de ordenação baseado em árvores binárias de busca (BST).

Ele funciona inserindo todos os elementos da lista em uma BST e depois realizando um percurso em ordem (in-order traversal), que naturalmente retorna os elementos em ordem crescente.

Para ordem decrescente, basta fazer o percurso em ordem reversa (right → root → left).

2. Funcionamento

1. **Inserção**: cada elemento da lista é inserido na árvore binária de busca.
    - Elementos menores vão para a subárvore esquerda.
    - Elementos maiores vão para a subárvore direita.
2. **Percurso**: após inserir todos os elementos, percorremos a árvore em ordem.
    - Esse percurso visita os nós em ordem crescente (left → root → right).
3. **Resultado**: a lista ordenada é obtida a partir desse percorrimento.
4. **Resultado decrescente**: o percorrimento na árvore é feito de maneira inversa.
    - Percorre primeiro a subárvore direita (right → root → left) para a ordem decrescente.

CÓDIGO ⬇

```python
import time # Importa o módulo 'time' para medir o desempenho de execução.
import random # Importa 'random' para gerar dados de teste aleatórios.
import matplotlib.pyplot as plt # Importa 'matplotlib.pyplot' para criação de gráficos.

class Node:
    def __init__(self, valor): # Define o valor e os ponteiros para os filhos
        self.valor = valor # Inicializa o valor do nó
        self.esquerda = None # Inicializa o filho da esquerda como vazio
        self.direita = None # Inicializa o filho da direita como vazio


    def inserir(self, valor): # Método para inserir um novo valor na árvore
        if valor < self.valor: # Se o valor inserido for menor que o valor atual, deve ir para a esquerda
            if self.esquerda is None: # Se não houver filho à esquerda, cria um novo nó
                self.esquerda = Node(valor)
            else: # Caso já exista, chama recursivamente o inserir no filho esquerdo
                self.esquerda.inserir(valor)
        else:
            if self.direita is None: # Se ao valor for maior ou igual, deve ir para a direita
                self.direita = Node(valor)
            else: # Caso já exista, chama recursivamente o inserir no filho direito
                self.direita.inserir(valor)

    def em_ordem(self, resultado): # Método para percorrer a árvore em ordem
        if self.esquerda: # Se existir filho à esquerda, percorre primeiro
            self.esquerda.em_ordem(resultado)
        resultado.append(self.valor) # Adiciona o valor atual à lista de resultados
        if self.direita: # Se existir filho à direita, percorre depois
            self.direita.em_ordem(resultado)

    def tree_sort(arr): # Método estático para executar o Tree Sort em ordem crescente
      if not arr: # Se a lista estiver vazia, retorna lista vazia
          return []
      root = Node(arr[0]) # Cria a raiz da árvore com o primeiro elemento da lista
      for valor in arr[1:]: # Insere os demais elementos na árvore
          root.inserir(valor)
      resultado = [] # Cria uma lista para armazenar o resultado ordenado

      root.em_ordem(resultado) # Percorre a árvore em ordem para obter os elementos ordenados
      return resultado # Retorna a lista final em ordem crescente

    def tree_sort_desc(arr): # Método estático para executar o Tree Sort em ordem decrescente
        if not arr: # Se a lista estiver vazia, retorna lista vazia
            return []
        root = Node(arr[0]) # Cria a raiz da árvore com o primeiro elemento da lista
        for valor in arr[1:]: # Insere os demais elementos na árvore
            root.inserir(valor)
        resultado = [] # Cria uma lista para armazenar o resultado ordenado
        root.em_ordem(resultado) # Percorre a árvore em ordem para obter os elementos ordenados
        return resultado[::-1] # Retorna a lista em ordem decrescente

```

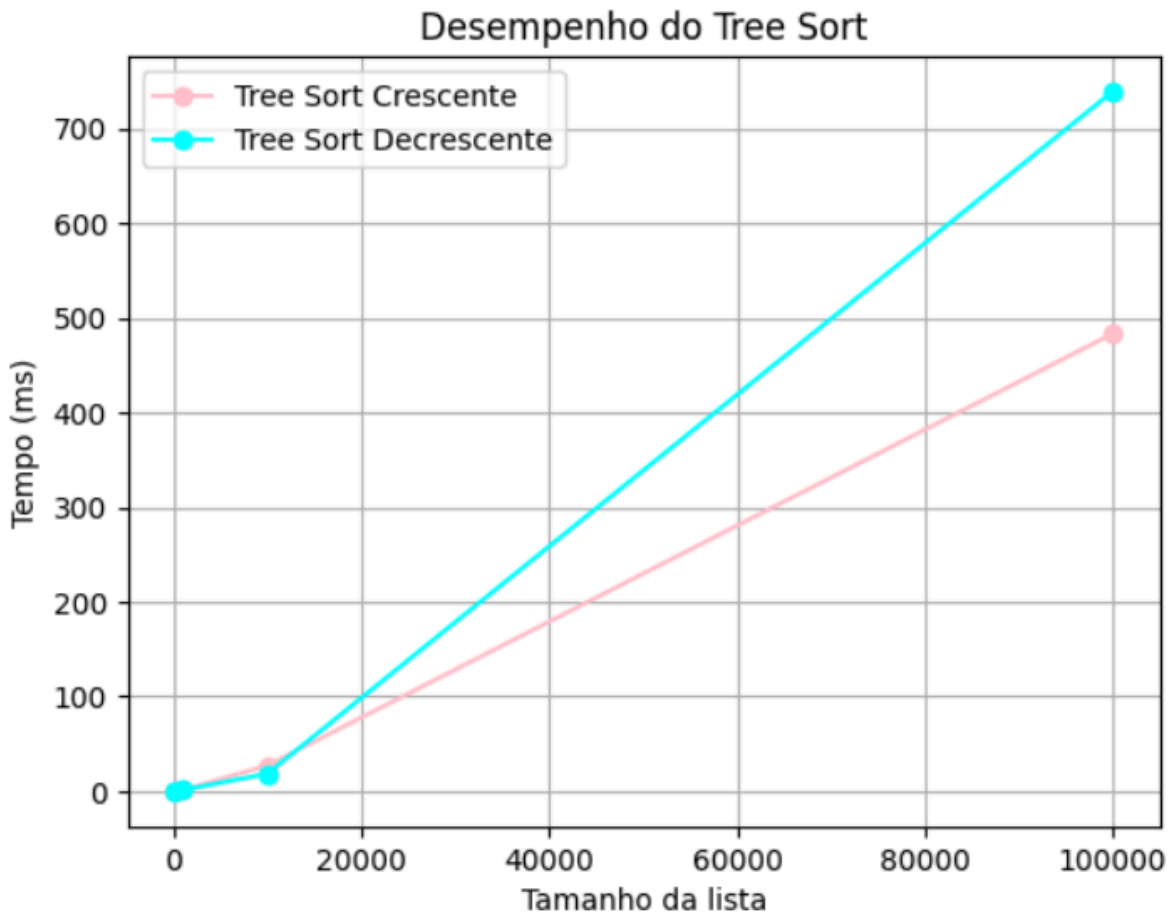
```python

# Demonstração simples

lista = [12, 10, 16, 9, 15, 11, 13, 14]

# Ordenação crescente usando tree_sort
print("Crescente:", Node.tree_sort(lista))

# Ordenação decrescente usando tree_sort_descS
print("Decrescente:", Node.tree_sort_desc(lista))


def gerar_lista(tamanho): # Função para gerar as listas (100, 1000, 10000, 100000)
    """Gera uma lista aleatória com o tamanho pedido."""
    return random.sample(range(tamanho * 10), tamanho)

def medir_tempo(funcao, lista): # Função para medir o tempo que a ordenação leva para executar (em milissegundos)
    inicio = time.time()
    funcao(lista)
    fim = time.time()
    return (fim - inicio) * 1000

def testar_metodos(): # Função para testar a ordenação cres e desc
    tamanhos = [100, 1000, 10000, 100000]
    resultados_cresc = []
    resultados_desc = []

    for t in tamanhos:
        lista = gerar_lista(t)

        # Teste para crescente
        tempo_c = medir_tempo(Node.tree_sort, lista)
        resultados_cresc.append(tempo_c)

        # Teste para decrescente
        tempo_d = medir_tempo(Node.tree_sort_desc, lista)
        resultados_desc.append(tempo_d)

        print(f"\nTamanho {t}:")
        print(f"  Crescente: {tempo_c:.4f} ms")
        print(f"  Decrescente: {tempo_d:.4f} ms")

    return tamanhos, resultados_cresc, resultados_desc

def gerar_grafico(): # Função para gerar o gráfico com base nos testes
    tamanhos, cresc, desc = testar_metodos()

    plt.plot(tamanhos, cresc, marker="o", label="Tree Sort Crescente", color="pink")
    plt.plot(tamanhos, desc, marker="o", label="Tree Sort Decrescente", color="cyan")

    plt.xlabel("Tamanho da lista")
    plt.ylabel("Tempo (ms)")
    plt.title("Desempenho do Tree Sort")
    plt.legend()
    plt.grid(True)
    plt.show()
```

COMPLEXIDADE

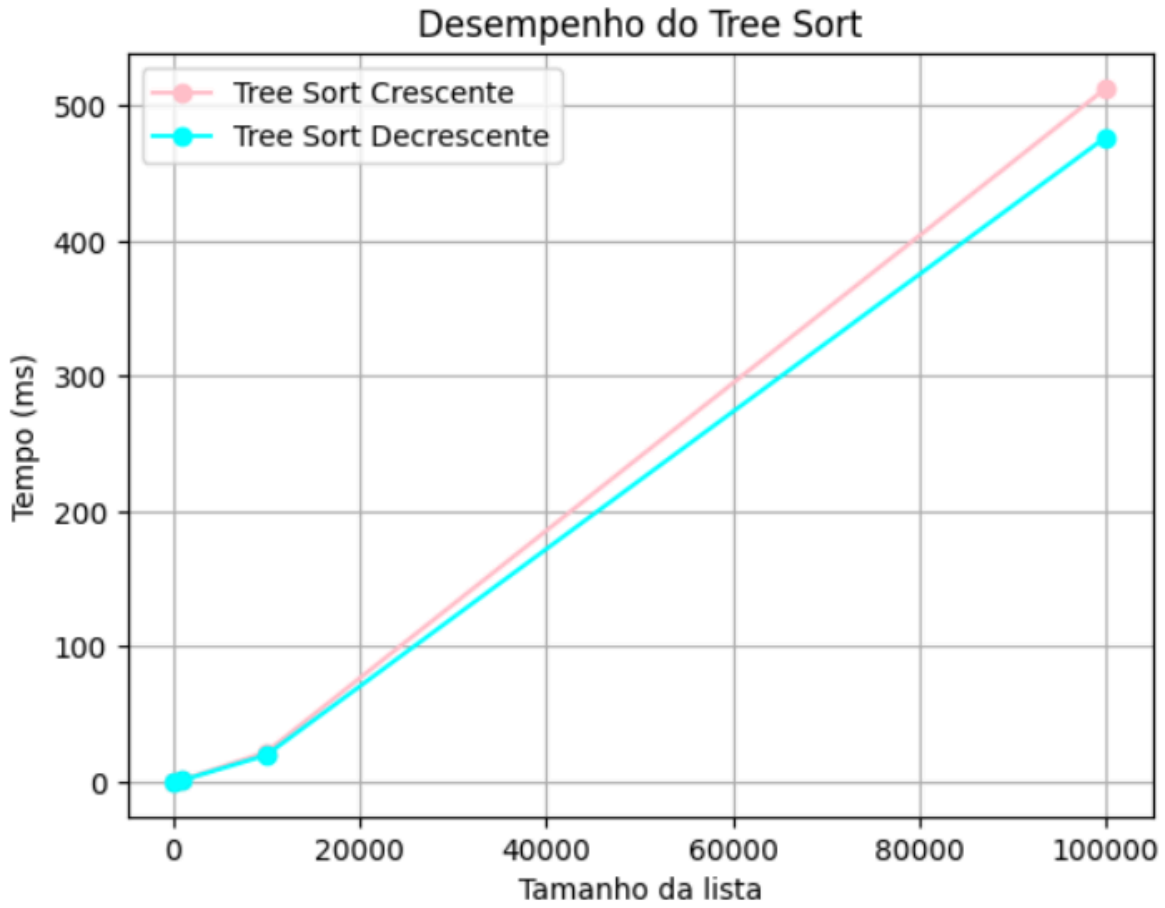
- **Melhor caso**: O(n log n) — quando a árvore fica balanceada.
- **Caso médio**: O(n log n).
- **Pior caso**: O(n²) — ocorre quando a árvore fica degenerada (por exemplo, se os dados já estão ordenados e não há balanceamento).

TABELA DE TESTES

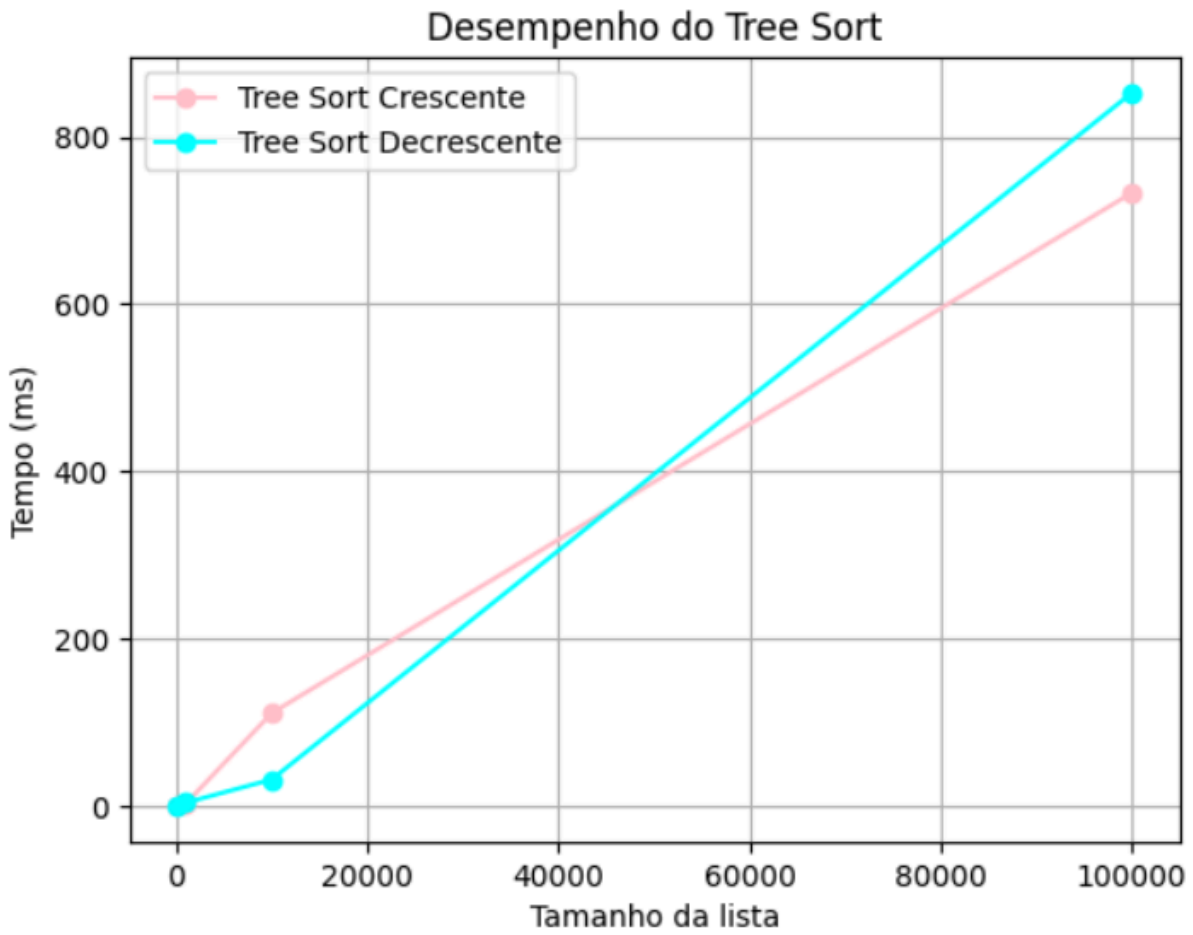
Situação em que a desc custou + tempo	100	1.000	10.000	100.000
Crescente	0.1466 ms	1.5316 ms	27.1986 ms	483.1939 ms
Decrescente	0.0899 ms	1.3180 ms	18.5730 ms	739.0559 ms



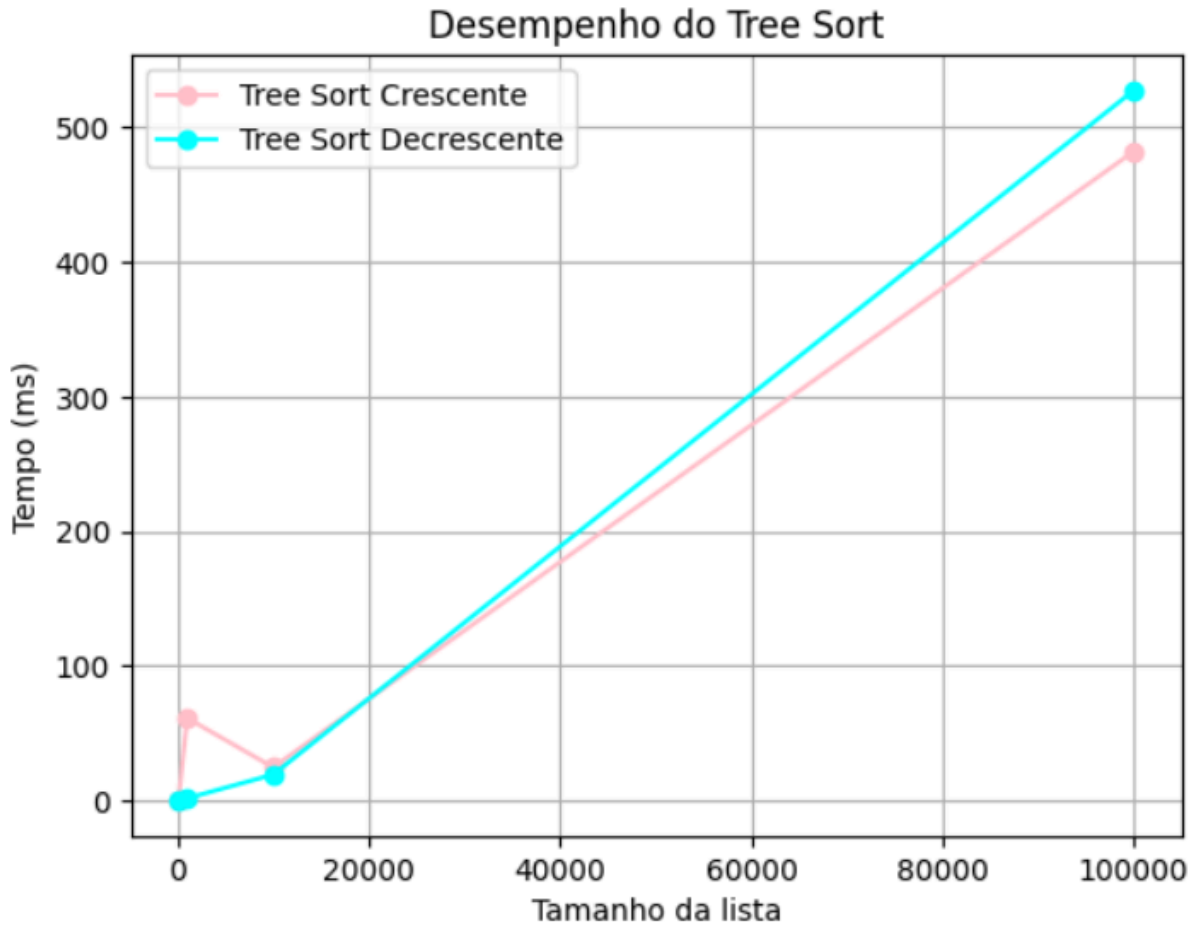
Situação em que a cresc custou + tempo	100	1.000	10.000	100.000
Crescente	0.1101 ms	1.4489 ms	21.9772 ms	512.8317 ms
Decrescente	0.0849 ms	1.1785 ms	19.6905 ms	476.7008 ms



Situação em que houve variações	100	1.000	10.000	100.000
Crescente	0.1888 ms	2.6259 ms	110.8451 ms	732.9566 ms
Decrescente	0.1626 ms	3.7749 ms	31.5170 ms	852.3810 ms



Situação em que houve variações e picos	100	1.000	10.000	100.000
Crescente	0.1137 ms	61.0645 ms	24.4641 ms	482.1954 ms
Decrescente	0.0992 ms	1.4341 ms	19.0535 ms	527.5319 ms



3. Discussão

COMPORTAMENTO GERAL

Nos testes, o tempo de execução aumenta conforme o tamanho da lista cresce, como esperado para algoritmos de ordenação. Porém, com tempos distintos entre a ordenação crescente e decrescente.

Normalmente, a ordenação crescente tende a ser ligeiramente mais rápida, mas aconteceram muitas variações em que a decrescente foi mais rápida. Ao que parece, a diferença entre crescente e decrescente não está na lógica do Tree Sort, e sim na operação de inversão (`[::-1]`) no caso decrescente, que é extremamente rápida e não altera a estrutura da árvore.

COMPLEXIDADE TEÓRICA

A complexidade do Tree Sort depende da forma da árvore:

Melhor caso O(n log n):

Acontece quando os valores são inseridos de forma que a árvore fique relativamente equilibrada (valores aleatórios). Dessa maneira a árvore não fica profunda demais, as inserções são mais eficientes e a travessia em ordem é linear e rápida.

Pior caso O(n²):

Ocorre quando a árvore fica totalmente desbalanceada, parecida com uma lista encadeada ([1, 2, 3, 4, 5] ou [5, 4, 3, 2, 1]).

Caso Médio O(n log n)

O caso médio considera uma lista com elementos distribuídos de forma aleatória. No caso do Tree Sort, isso geralmente leva a uma árvore que fica com profundidade próxima ao log n, resultando em inserções mais eficientes,
número reduzido de comparações e percurso final otimizado.

Os testes confirmaram isso, pois os tempos subiram suavemente entre as entradas 100 → 1.000 → 10.000, em um padrão parecido com n log n.

Crescimento quanto maior o volume de dados

Mesmo sendo O(n log n) na média, o Tree Sort usa recursão na inserção, cria muitos objetos (nós da árvore), faz alocações constantes de memória e executa diversas comparações a cada inserção.

Por isso, quando passamos de 10 mil para 100 mil elementos:

- O log n cresce pouco, mas
- O "n" cresce 10 vezes

Então, o tempo total cresce de forma perceptível.

Comparando crescente e decrescente

A diferença entre crescente e decrescente no algoritmo não é computacionalmente relevante, pois a árvore é construída da mesma forma nos dois casos e percurso é o mesmo. A única diferença é o [::-1].

Então, ao que parece, a diferença de tempo observada entre crescente e decrescente é apenas variação natural da execução, não do algoritmo em si.

4. Conclusão

Com base nos testes, podemos dá para concluir que:

- O Tree Sort se comportou com eficiência no caso médio em que foi testado, apresentando tempos compatíveis com O(n log n).
- A construção da árvore é responsável pela maior parte do custo total.
- A ordenação decrescente não é mais lenta, a diferença se deve apenas à etapa final de inversão da lista.
- Para listas muito grandes, o algoritmo se torna significativamente mais lento devido à criação intensiva de nós e chamadas recursivas.
- Se a entrada estivesse pré-ordenada, o tempo seria muito maior (pior caso = $O(n^2)$), o que não foi o caso dos testes.