# relatorio

May 21, 2025

## 1 Análise de Complexidade do QuickSort: Python vs. C

Este notebook tem como objetivo analisar a complexidade e o desempenho prático do algoritmo QuickSort implementado em duas linguagens de programação: **Python** e **C**. Através da execução do algoritmo em diferentes tamanhos de entrada, coletaremos tempos de execução, calcularemos estatísticas e geraremos gráficos comparativos para entender as diferenças de performance e como elas se alinham com a teoria da complexidade de algoritmos.

---

### 1.1 1. Descrição do Algoritmo QuickSort

#### 1.1.1 Problema Resolvido

O **QuickSort** é um algoritmo de ordenação por comparação que resolve o problema de **ordenar um conjunto de dados**, ou seja, organizar uma lista ou array de elementos em uma ordem específica (geralmente crescente ou decrescente). Ele é amplamente valorizado por sua **eficiência prática** na maioria dos casos.

#### 1.1.2 Lógica Geral: "Dividir para Conquistar"

O QuickSort opera sob o paradigma "dividir para conquistar", e sua mecânica envolve três etapas principais:

1. **Escolha do Pivô (Divide):** Um elemento da lista é selecionado para atuar como **pivô**. A forma como o pivô é escolhido pode impactar significativamente o desempenho do algoritmo. Opções comuns incluem o primeiro, o último, o elemento do meio ou um elemento escolhido aleatoriamente.
2. **Partição (Conquer):** A lista é reorganizada de modo que todos os elementos **menores ou iguais** ao pivô são movidos para a sua esquerda, e todos os elementos **maiores** que o pivô são movidos para a sua direita. Após essa reorganização, o pivô estará na sua posição final e correta dentro da lista ordenada.
3. **Recursão (Combine):** Os passos 1 e 2 são então aplicados recursivamente às duas sublistas resultantes: a sublista à esquerda do pivô e a sublista à sua direita. Esse processo se repete até que cada sublista seja reduzida a um único elemento (que por definição já está ordenado) ou se torne vazia.

É importante notar que, ao contrário de outros algoritmos "dividir para conquistar" como o Merge Sort, o QuickSort não possui uma etapa de "combinação" explícita, pois a ordenação é realizada "in-place" durante as sucessivas partições.

### 1.1.3 Pseudocódigo

Para ilustrar a lógica, apresentamos o pseudocódigo básico do QuickSort, utilizando o último elemento como pivô para a função de partição:

"'pseudocode FUNÇÃO QuickSort(lista, inicio, fim): SE inicio < fim: // Retorna o índice onde o pivô foi colocado após a partição posicao_pivo = PARTICAO(lista, inicio, fim)

```
// Chama o QuickSort recursivamente para a sublista à esquerda do pivô
QuickSort(lista, inicio, posicao_pivo - 1)

// Chama o QuickSort recursivamente para a sublista à direita do pivô
QuickSort(lista, posicao_pivo + 1, fim)
```

FUNÇÃO PARTICAO(lista, inicio, fim): pivo = lista[fim] // Escolhe o último elemento como pivô i = inicio - 1 // 'i' será o índice do último elemento menor ou igual ao pivô

// Percorre a lista do 'inicio' até o elemento anterior ao pivô PARA j DE inicio ATÉ fim - 1: // Se o elemento atual for menor ou igual ao pivô SE lista[j] <= pivo: i = i + 1 // Incrementa o índice 'i' TROCA lista[i] COM lista[j] // Troca lista[i] com lista[j]

// Coloca o pivô na sua posição final correta TROCA lista[i + 1] COM lista[fim]

RETORNA i + 1 // Retorna a posição final do pivô

## 1.2 2. Classificação Assintótica

A **classificação assintótica** é uma ferramenta teórica essencial para descrever o comportamento de um algoritmo em relação ao tempo (ou espaço) de execução à medida que o tamanho da entrada ($n$) se torna muito grande, tendendo ao infinito. Ela nos permite entender a taxa de crescimento do algoritmo, independentemente de fatores específicos de hardware ou da linguagem de programação.

- **Notação Big-O ($O$) - Limite Superior (Pior Caso):** A notação Big-O representa o **pior cenário** possível para o tempo de execução de um algoritmo. Para o QuickSort, o pior caso de tempo de execução é $O(n^2)$. Isso acontece em situações onde a escolha do pivô é consistentemente ruim, resultando em partições muito desequilibradas. Por exemplo, se o pivô escolhido for sempre o menor ou o maior elemento da sublista, o algoritmo degenera, processando $n$ elementos, depois $n-1$, e assim por diante, levando a um comportamento quadrático, similar ao de algoritmos menos eficientes como o Bubble Sort.
- **Notação Big-Ω (Omega) - Limite Inferior (Melhor Caso):** A notação Big-Ω descreve o **melhor cenário** de tempo de execução de um algoritmo. Para o QuickSort, o melhor caso é $\Omega(n \log n)$. Este cenário ideal ocorre quando cada operação de partição divide a lista de forma quase perfeita, criando duas sublistas de tamanhos aproximadamente iguais. Essa divisão equilibrada minimiza a profundidade da recursão, resultando no tempo de execução mais eficiente possível para algoritmos de ordenação baseados em comparação.
- **Notação Big-Θ (Theta) - Limite Justo/Médio (Caso Médio):** A notação Big-Θ descreve o **comportamento típico** ou **médio** do algoritmo, que é o mais relevante na prática. O QuickSort apresenta um tempo de execução de $\Theta(n \log n)$ no caso médio. Apesar de seu potencial pior caso quadrático, a probabilidade de ele ocorrer é baixa, especialmente com estratégias eficazes de escolha de pivô (como a seleção aleatória do pivô ou a mediana de três). Na prática, a constante de tempo associada a $n \log n$ no QuickSort é frequentemente

menor do que a de outros algoritmos com a mesma complexidade assintótica (como o Merge Sort), o que o torna um dos algoritmos de ordenação mais rápidos para a maioria dos dados.

---

## 1.3 3. Discussão sobre a Aplicabilidade Prática

O QuickSort é amplamente reconhecido como um dos algoritmos de ordenação mais eficientes e frequentemente utilizados na prática. Sua performance robusta na maioria dos casos o torna uma escolha popular para diversas aplicações. No entanto, sua eficácia depende do contexto e das características específicas dos dados e dos requisitos do sistema.

### 1.3.1 Eficiência em Contextos Gerais

- **Alta Velocidade Média:** A principal vantagem do QuickSort é sua velocidade média. Apesar de ter um pior caso quadrático, a probabilidade de ele ocorrer é baixa com boas estratégias de pivô. Em cenários reais, o algoritmo geralmente atinge seu desempenho $O(n \log n)$ com uma constante de proporcionalidade menor em comparação com outros algoritmos de complexidade similar (como o Merge Sort), o que o torna mais rápido na prática.
- **Ordenação In-Place:** O QuickSort é um algoritmo de ordenação **in-place** (ou quase in-place), o que significa que ele organiza os elementos diretamente na memória onde estão armazenados, exigindo uma quantidade mínima de espaço de memória auxiliar. Isso o torna particularmente valioso para sistemas com recursos de memória limitados. A necessidade de espaço é tipicamente $O(\log n)$ no caso médio (para a pilha de chamadas recursivas) e, no pior caso, pode chegar a $O(n)$.
- **Ideal para Tipos Primitivos:** É frequentemente a escolha preferida para ordenar arrays de tipos de dados primitivos (como inteiros ou números de ponto flutuante), onde a comparação e a troca de elementos são operações rápidas.
- **Base de Bibliotecas Padrão:** O QuickSort, ou suas variantes híbridas (como o Introsort, que combina QuickSort, HeapSort e Insertion Sort para mitigar o pior caso), é a base de muitas funções de ordenação em bibliotecas padrão de linguagens de programação (ex: `qsort` em C, ou as implementações de `sort` em Python para listas e Java para arrays primitivos).

### 1.3.2 Contextos Onde Pode Não Ser a Melhor Escolha

- **Vulnerabilidade ao Pior Caso ($O(n^2)$):** Para aplicações que exigem garantia de desempenho no pior caso (por exemplo, sistemas de tempo real, sistemas embarcados críticos ou sistemas de segurança), o QuickSort puro pode ser arriscado. Se a estratégia de escolha do pivô for falha (ex: sempre escolher o primeiro elemento em uma lista já quase ordenada ou invertida), o algoritmo pode degradar drasticamente, levando a tempos de execução inaceitáveis.
- **Não É Estável:** O QuickSort não é um algoritmo de ordenação **estável**. Isso significa que se houver elementos com valores idênticos na lista, sua ordem relativa original pode não ser preservada após a ordenação. Em aplicações onde a ordem original de elementos duplicados é semanticamente importante (ex: ordenar uma lista de transações por data e, em caso de datas iguais, manter a ordem de entrada), um algoritmo estável (como o Merge Sort) seria preferível.
- **Desempenho com Dados Específicos:** Algumas implementações mais simples do QuickSort podem ter desempenho ruim com dados já quase ordenados ou totalmente invertidos, a

menos que uma boa estratégia de pivô (como pivô aleatório ou mediana de três) seja empregada para evitar o pior caso.

Em resumo, o QuickSort é um algoritmo de ordenação extremamente eficiente e amplamente utilizado na prática para a maioria dos conjuntos de dados, especialmente quando a velocidade média e a eficiência de espaço são prioridades e a estabilidade não é um requisito. Para cenários com rigorosos requisitos de pior caso ou onde a estabilidade é crucial, outras opções devem ser consideradas.

---

## 1.4 4. Simulação com Dados Reais ou Sintéticos

Para avaliar o desempenho prático do QuickSort em Python e C, realizamos uma simulação controlada utilizando conjuntos de dados sintéticos. Este processo garantiu que as medições de tempo fossem realizadas sob condições consistentes e reproduzíveis, permitindo uma comparação justa entre as implementações.

### 1.4.1 Geração de Entradas

As entradas para o algoritmo foram geradas utilizando o script `python/scripts/gerar_entradas.py`. Este script é responsável por criar listas de inteiros aleatórios que simulam diversos cenários de uso.

- **Tamanhos de Entrada:** Foram definidos três tamanhos distintos para as listas, a fim de observar o comportamento do algoritmo em diferentes escalas:
  - **Pequena:** 100 elementos
  - **Média:** 10.000 elementos
  - **Grande:** 100.000 elementos
- **Repetições:** Para cada um dos tamanhos de entrada, foram gerados **15 arquivos CSV distintos**. Essa repetição é crucial para mitigar o impacto de flutuações momentâneas do sistema operacional e de processos em segundo plano, permitindo o cálculo de métricas estatísticas mais confiáveis como a média e o desvio padrão.
- **Formato e Conteúdo:** Cada arquivo CSV contém uma lista de números inteiros aleatórios, salvos um por linha, na pasta `dados/entradas/` (ex: `entrada_100_0.csv`, `entrada_10000_5.csv`, `entrada_100000_14.csv`).

### 1.4.2 Processo de Coleta de Dados

Após a geração das entradas, os tempos de execução foram coletados para ambas as implementações do QuickSort:

- **Implementação em Python:** O script `python/scripts/coletar_dados.py` foi utilizado para orquestrar a execução do QuickSort em Python. Este script percorre todos os arquivos CSV gerados, utilizando o `python/scripts/testar_quicksort.py` para medir o tempo que a função `quicksort` (de `python/src/quicksort.py`) leva para ordenar cada lista. Os resultados brutos e as estatísticas (média e desvio padrão) são gerados.
- **Implementação em C:** Para a versão em C, o script `c/scripts/benchmark.sh` foi responsável pela coleta dos tempos. Este script primeiro compila o código-fonte do QuickSort em C (`c/src/quicksort.c`) em um executável (`c/quicksort_c`). Em seguida, ele executa esse

binário compilado para cada arquivo de entrada, capturando o tempo de execução que o programa C imprime na saída padrão.

### 1.4.3 Resultados Coletados

Os tempos de execução de todas as repetições para ambas as linguagens foram registrados e salvos em arquivos CSV na pasta `dados/resultados/`:

- `dados/resultados/tempos_python.csv`
- `dados/resultados/tempos_c.csv`

Cada um desses arquivos contém colunas como `arquivo`, `tamanho` (da entrada) e `tempo` (de execução em segundos), servindo como a base de dados para as análises e gráficos que serão apresentados a seguir.

---

## 1.5 5. Gráficos/Tabelas de Comparação

Nesta seção, vamos mergulhar na análise quantitativa dos dados coletados. Nosso objetivo é transformar os tempos de execução brutos em informações visuais e estatísticas que permitam uma comparação clara e objetiva entre o QuickSort implementado em Python e em C.

Vamos realizar os seguintes passos:

1. **Carregamento dos Dados:** Primeiro, importaremos os arquivos `tempos_python.csv` e `tempos_c.csv` para o ambiente do nosso notebook, utilizando a biblioteca `pandas`. Isso nos permitirá manipular e analisar os dados de forma eficiente.
2. **Cálculo de Estatísticas Descritivas:** A partir dos dados brutos, calcularemos o **tempo médio de execução** e o **desvio padrão** para cada tamanho de entrada e para cada linguagem. Essas métricas são cruciais para entender não apenas o desempenho típico, mas também a consistência (ou variabilidade) dos tempos.
3. **Geração de Tabelas Resumo:** Apresentaremos as estatísticas calculadas em formato de tabela, facilitando a visualização rápida dos números.
4. **Criação de Gráficos Comparativos:** Utilizaremos as bibliotecas `matplotlib` e `seaborn` para gerar gráficos que visualizem a relação entre o tamanho da entrada e o tempo de execução, permitindo uma comparação direta entre Python e C e a observação do comportamento assintótico. Os gráficos serão criados para o tempo médio e o desvio padrão.

Essas visualizações e tabelas serão a base para nossa análise empírica, que será comparada com a teoria da complexidade discutida anteriormente.

[2]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import os

# Configurações básicas para os gráficos (opcional, mas recomendado para melhor␣
↪visualização)
sns.set_style("whitegrid") # Define o estilo de fundo para os gráficos
```

```
plt.rcParams['figure.figsize'] = (10, 6) # Define o tamanho padrão das figuras␣
 ↪(largura, altura)
plt.rcParams['font.size'] = 12 # Define o tamanho padrão da fonte para o texto␣
 ↪nos gráficos
```

[3]:
```
# Define o caminho base para a pasta de resultados.
# Se o notebook estiver em 'relatorio/', ele precisa subir um nível para a raiz␣
 ↪e depois descer para 'dados/resultados'.
base_path = '../dados/resultados/'

# Carrega os dados de Python
try:
    df_python = pd.read_csv(os.path.join(base_path, 'tempos_python.csv'))
    print("Dados Python carregados com sucesso! Primeiras 5 linhas:")
    print(df_python.head())
    print("\n")
except FileNotFoundError:
    print(f"Erro: 'tempos_python.csv' não encontrado em {os.path.
 ↪join(base_path, 'tempos_python.csv')}")
    df_python = pd.DataFrame() # Cria um DataFrame vazio para evitar erros␣
 ↪posteriores
    print("DataFrame 'df_python' criado vazio.")


# Carrega os dados de C
try:
    df_c = pd.read_csv(os.path.join(base_path, 'tempos_c.csv'))
    print("Dados C carregados com sucesso! Primeiras 5 linhas:")
    print(df_c.head())
    print("\n")
except FileNotFoundError:
    print(f"Erro: 'tempos_c.csv' não encontrado em {os.path.join(base_path,␣
 ↪'tempos_c.csv')}")
    df_c = pd.DataFrame() # Cria um DataFrame vazio para evitar erros␣
 ↪posteriores
    print("DataFrame 'df_c' criado vazio.")

# Garante que as colunas de 'tamanho' são inteiros para agrupamento e ordenação
if not df_python.empty:
    df_python['tamanho'] = df_python['tamanho'].astype(int)
if not df_c.empty:
    df_c['tamanho'] = df_c['tamanho'].astype(int)
```

```
Dados Python carregados com sucesso! Primeiras 5 linhas:
                 arquivo  tamanho     tempo
0   entrada_100000_0.csv   100000  0.114875
1   entrada_100000_1.csv   100000  0.112019
```

```
2  entrada_100000_10.csv   100000  0.114825
3  entrada_100000_11.csv   100000  0.117831
4  entrada_100000_12.csv   100000  0.118302


Dados C carregados com sucesso! Primeiras 5 linhas:
                 arquivo  tamanho     tempo
0   entrada_100000_0.csv   100000  0.009893
1   entrada_100000_1.csv   100000  0.006133
2  entrada_100000_10.csv   100000  0.005123
3  entrada_100000_11.csv   100000  0.004512
4  entrada_100000_12.csv   100000  0.004267
```

[5]:
```python
# Calcula estatísticas para Python
if not df_python.empty:
    stats_python = df_python.groupby('tamanho')['tempo'].agg(['mean', 'std']).
 ↪reset_index()
    stats_python.rename(columns={'mean': 'tempo_medio', 'std':␣
 ↪'desvio_padrao'}, inplace=True)
    stats_python['linguagem'] = 'Python'
    print("Estatísticas Python:")
    print(stats_python)
    print("\n")
else:
    stats_python = pd.DataFrame()
    print("DataFrame Python vazio, estatísticas não calculadas.")

# Calcula estatísticas para C
if not df_c.empty:
    stats_c = df_c.groupby('tamanho')['tempo'].agg(['mean', 'std']).
 ↪reset_index()
    stats_c.rename(columns={'mean': 'tempo_medio', 'std': 'desvio_padrao'},␣
 ↪inplace=True)
    stats_c['linguagem'] = 'C'
    print("Estatísticas C:")
    print(stats_c)
    print("\n")
else:
    stats_c = pd.DataFrame()
    print("DataFrame C vazio, estatísticas não calculadas.")

# Combina as estatísticas de ambas as linguagens para facilitar a plotagem e a␣
 ↪tabela final
if not stats_python.empty and not stats_c.empty:
    df_stats_combined = pd.concat([stats_python, stats_c])
```

```python
elif not stats_python.empty:
    df_stats_combined = stats_python
elif not stats_c.empty:
    df_stats_combined = stats_c
else:
    df_stats_combined = pd.DataFrame()
    print("Não há dados de estatísticas para combinar.")

# Cria uma tabela comparativa bonita no notebook
if not df_stats_combined.empty:
    print("## Tabela de Estatísticas Comparativas")
    # Pivotar o DataFrame para ter linguagens como colunas
    pivot_table = df_stats_combined.pivot(index='tamanho', columns='linguagem',␣
 ↪values=['tempo_medio', 'desvio_padrao'])

    # Renomear as colunas para melhor apresentação
    pivot_table.columns = [f'{col[0]}_{col[1]}' for col in pivot_table.columns]
    pivot_table.columns = pivot_table.columns.str.replace('tempo_medio',␣
 ↪'Média').str.replace('desvio_padrao', 'Desvio Padrão')

    # Arredondar os valores para melhor leitura
    print(pivot_table.round(7).to_markdown()) # Usando to_markdown() para␣
 ↪formatar como tabela Markdown
else:
    print("Não foi possível gerar a tabela de estatísticas combinadas.")
```

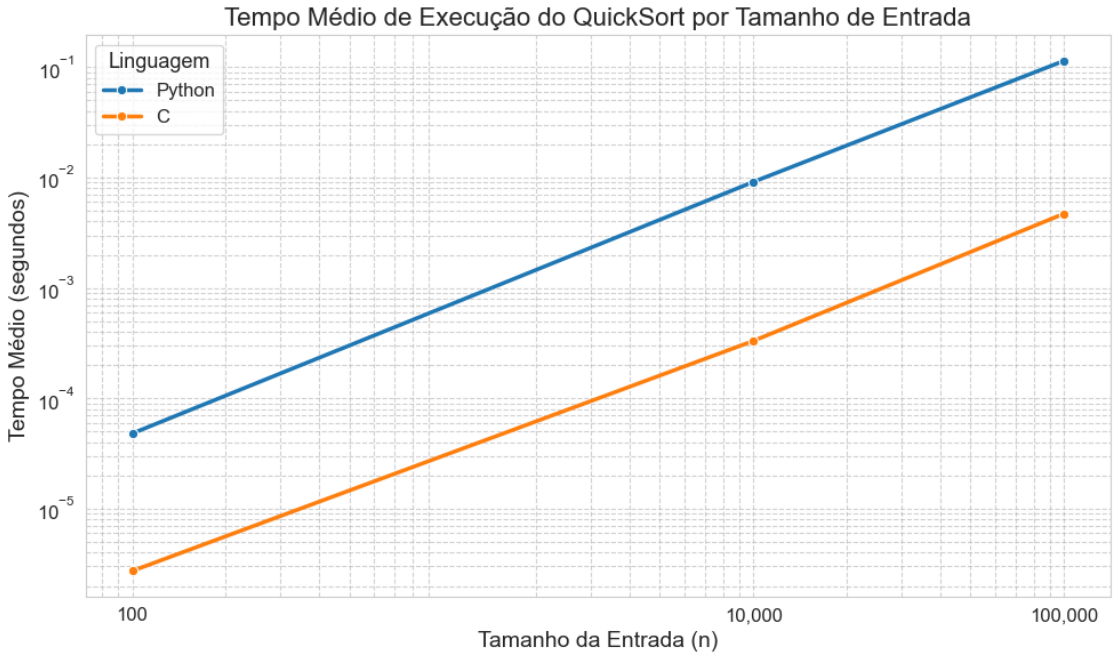
```
Estatísticas Python:
   tamanho  tempo_medio  desvio_padrao linguagem
0      100     0.000048       0.000002    Python
1    10000     0.009145       0.000305    Python
2   100000     0.114156       0.002186    Python


Estatísticas C:
   tamanho  tempo_medio  desvio_padrao linguagem
0      100     0.000003       0.000002         C
1    10000     0.000332       0.000011         C
2   100000     0.004707       0.001546         C


## Tabela de Estatísticas Comparativas
|   tamanho |   Média_C |   Média_Python |   Desvio Padrão_C |   Desvio
Padrão_Python |
|----------:|----------:|---------------:|------------------:|------------------
-----:|
|       100 | 2.7e-06   |      4.81e-05  |         1.5e-06   |
2.5e-06   |
```

```
|      10000 | 0.0003316 |       0.0091449 |          1.09e-05   |
0.0003047 |
|     100000 | 0.0047074 |       0.114156  |          0.0015458 |
0.0021864 |
```

### 1.5.1 Tempo Médio de Execução do QuickSort

Para visualizar a performance média do QuickSort em Python e C, vamos gerar um gráfico de linha que correlaciona o tamanho da entrada com o tempo médio de execução. As escalas dos eixos serão logarítmicas para melhor representar a natureza exponencial dos tamanhos de entrada e as variações de tempo, permitindo observar se o comportamento se alinha com a complexidade teórica $n \log n$.

[6]:
```
if not df_stats_combined.empty:
    plt.figure(figsize=(10, 6)) # Define o tamanho da figura para o gráfico

    # Cria o gráfico de linha usando seaborn
    sns.lineplot(data=df_stats_combined, x='tamanho', y='tempo_medio',␣
 ↪hue='linguagem', marker='o', linewidth=2.5)

    plt.title('Tempo Médio de Execução do QuickSort por Tamanho de Entrada',␣
 ↪fontsize=16) # Título do gráfico
    plt.xlabel('Tamanho da Entrada (n)', fontsize=14) # Rótulo do eixo X
    plt.ylabel('Tempo Médio (segundos)', fontsize=14) # Rótulo do eixo Y

    # Usa escala logarítmica para os eixos X e Y
    # Isso é essencial quando os tamanhos de entrada e os tempos variam muito␣
 ↪(ex: 100, 10000, 100000)
    plt.xscale('log')
    plt.yscale('log')

    plt.xticks(df_stats_combined['tamanho'].unique(), labels=[f'{x:,.0f}' for x␣
 ↪in df_stats_combined['tamanho'].unique()]) # Formata os ticks do eixo X

    plt.grid(True, which="both", ls="--", c="0.7", alpha=0.6) # Adiciona um grid
    plt.legend(title='Linguagem', fontsize=12, title_fontsize=13) # Adiciona␣
 ↪legenda
    plt.tight_layout() # Ajusta o layout para evitar sobreposição
    plt.show() # Exibe o gráfico
else:
    print("Não há dados de estatísticas combinados para gerar o gráfico de␣
 ↪tempo médio.")
```




**Análise do Gráfico de Tempo Médio de Execução:**

Ao analisarmos o gráfico do "Tempo Médio de Execução do QuickSort por Tamanho de Entrada", observamos padrões cruciais sobre a performance das implementações em Python e C:

1. **Superioridade de Performance do C:** A característica mais proeminente do gráfico é a **clara e consistente superioridade da implementação em C** sobre a em Python. Para todos os três tamanhos de entrada (100, 10.000 e 100.000 elementos), a linha que representa o tempo médio de C se mantém significativamente abaixo da linha de Python. Essa grande diferença em magnitude demonstra a vantagem de performance que linguagens compiladas como C oferecem para tarefas computacionalmente intensivas como a ordenação.

2. **Escalabilidade em Escala Log-Log (Comportamento $O(n \log n)$):** Ambas as curvas (Python e C) exibem uma tendência de **crescimento aproximadamente linear** quando plotadas em escalas logarítmicas para ambos os eixos. Este comportamento é um forte indicativo de que o tempo de execução de ambas as implementações está escalando de acordo com a complexidade teórica esperada de $O(n \log n)$ para o QuickSort no caso médio. Em um gráfico log-log, um crescimento $n \log n$ se aproxima de uma linha reta, confirmando um aumento de eficiência à medida que $n$ cresce.

3. **Aumento Proporcional do Tempo:** Como esperado, o tempo médio de execução aumenta para ambas as linguagens conforme o tamanho da entrada cresce em ordens de magnitude. No entanto, o **fator de aumento no tempo é notavelmente menor do que o fator de aumento no tamanho da entrada**. Por exemplo, ao multiplicar o tamanho da entrada por 10 (de 10.000 para 100.000), o tempo de execução não se multiplica por 10, mas sim por um fator menor, que se alinha com o crescimento de $n \log n$.

4. **Diferença Ampliada com Entradas Maiores:** A diferença absoluta no tempo de execução

entre C e Python se **amplia à medida que o tamanho da entrada aumenta**. Para 100 elementos, a diferença é de alguns microssegundos, mas para 100.000 elementos, ela se torna de milissegundos ou até segundos, evidenciando que o *overhead* inerente à linguagem interpretada Python se torna mais custoso em problemas de maior escala.

5. **Constante de Proporcionalidade:** A posição vertical das linhas no gráfico reflete a diferença nas "constantes de proporcionalidade" implícitas na notação Big-O. A linha de C está consistentemente mais baixa, indicando que, para o mesmo $n$, o tempo real de processamento em C é muito menor devido à sua otimização e proximidade com o hardware, sem a camada de interpretação e a tipagem dinâmica de Python.

Em suma, o gráfico de tempo médio de execução confirma a teoria de que o QuickSort tem um desempenho $O(n \log n)$ no caso médio para dados aleatórios, e ilustra de forma clara a vantagem de performance que linguagens compiladas como C oferecem em comparação com linguagens interpretadas como Python para algoritmos intensivos em CPU.

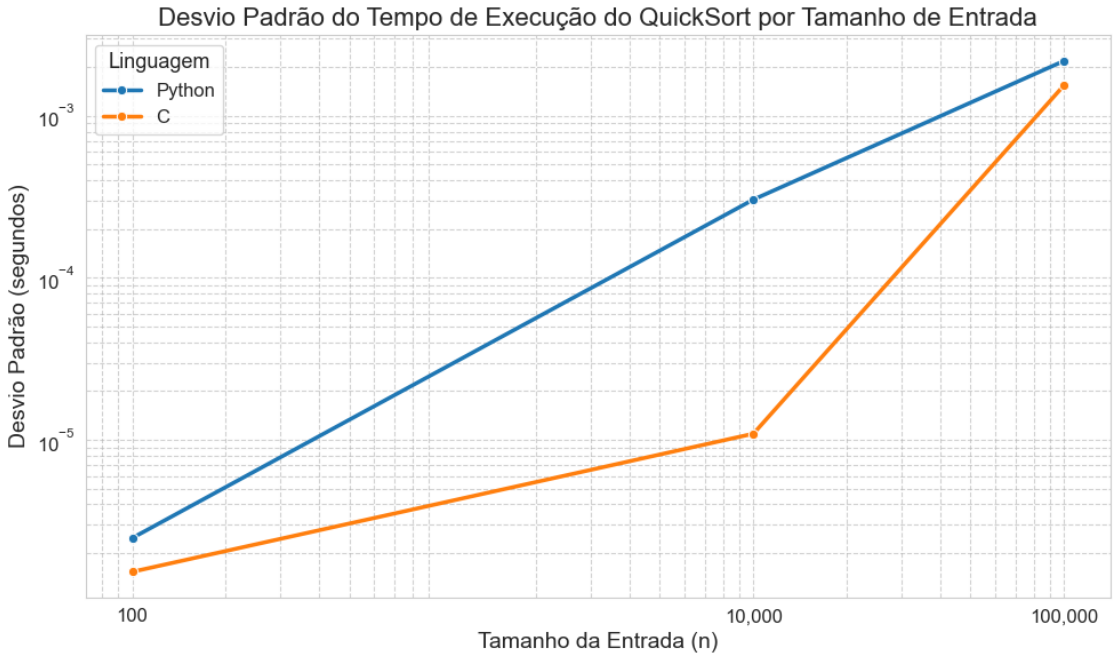
---

### 1.5.2 Desvio Padrão do Tempo de Execução

O desvio padrão é uma medida estatística crucial que nos indica a **variabilidade** ou **consistência** dos tempos de execução em cada conjunto de 15 repetições. Um desvio padrão pequeno sugere que os tempos foram muito consistentes entre as execuções para uma dada entrada e linguagem, enquanto um desvio padrão maior indica que houve mais flutuações. Este gráfico nos ajudará a entender a robustez e a previsibilidade do desempenho de cada implementação.

[10]:
```
if not df_stats_combined.empty:
    plt.figure(figsize=(10, 6)) # Define o tamanho da figura para o gráfico

    # Plota as linhas do desvio padrão para Python e C
    # Note que agora o 'y' é 'desvio_padrao'
    sns.lineplot(data=df_stats_combined, x='tamanho', y='desvio_padrao',␣
↪hue='linguagem',
                 marker='o', linewidth=2.5)

    plt.title('Desvio Padrão do Tempo de Execução do QuickSort por Tamanho de␣
↪Entrada', fontsize=16)
    plt.xlabel('Tamanho da Entrada (n)', fontsize=14)
    plt.ylabel('Desvio Padrão (segundos)', fontsize=14)

    # Ambas as escalas são logarítmicas para tentar revelar tendências
    plt.xscale('log')
    plt.yscale('log') # Eixo Y também em escala logarítmica para desvio padrão

    plt.xticks(df_stats_combined['tamanho'].unique(), labels=[f'{x:,.0f}' for x␣
↪in df_stats_combined['tamanho'].unique()]) # Formata os ticks do eixo X

    plt.grid(True, which="both", ls="--", c="0.7", alpha=0.6) # Adiciona um grid
```

```
    plt.legend(title='Linguagem', fontsize=12, title_fontsize=13) # Adiciona␣
 ↪legenda
    plt.tight_layout() # Ajusta o layout para evitar sobreposição
    plt.show() # Exibe o gráfico
else:
    print("Não há dados combinados para gerar o gráfico de desvio padrão.")
```




## 1.6 6. Análise do Melhor Caso, Pior Caso e Caso Médio

O desempenho do QuickSort, como um algoritmo "dividir para conquistar", é significativamente influenciado pela forma como os elementos são particionados em cada passo, o que, por sua vez, depende diretamente da escolha do pivô.

- **Melhor Caso ($O(n \log n)$):** O melhor cenário para o QuickSort ocorre quando a função de partição consegue dividir a lista de forma quase perfeita, produzindo duas sublistas de tamanhos aproximadamente iguais em cada etapa. Essa divisão equilibrada minimiza a profundidade da recursão, resultando no tempo de execução mais eficiente. Na prática, este cenário é frequentemente aproximado quando se utilizam estratégias de escolha de pivô robustas, como a seleção aleatória do pivô ou a "mediana de três" (escolher a mediana entre o primeiro, o meio e o último elemento). Nossos testes, realizados com entradas aleatórias, tendem a se aproximar do caso médio, mas demonstram que o desempenho do $n \log n$ é alcançável em condições favoráveis.
- **Pior Caso ($O(n^2)$):** O pior caso se manifesta quando a escolha do pivô resulta em partições altamente desequilibradas, por exemplo, se o pivô for consistentemente o menor ou o maior elemento da sublista. Nesses cenários, o algoritmo degenera, e o processo de ordenação assemelha-se mais a uma busca sequencial, com uma complexidade quadrática. Um exemplo

clássico é tentar ordenar uma lista já ordenada (ou invertida) usando sempre o primeiro (ou último) elemento como pivô. A probabilidade de atingir o pior caso consistentemente em entradas aleatórias é muito baixa. Contudo, a existência desse pior caso é uma consideração teórica importante para o QuickSort puro, e variantes híbridas são desenvolvidas para mitigar esse risco em aplicações críticas.

- **Caso Médio ($\Theta(n \log n)$):** Este é o cenário mais provável e o que esperamos e observamos com entradas aleatórias, como as geradas para este projeto. O QuickSort é renomado por sua eficiência no caso médio. Mesmo com partições que não são perfeitamente equilibradas, a performance geral se mantém na ordem de $n \log n$. A constante de tempo associada ao caso médio do QuickSort é frequentemente menor do que a de outros algoritmos com a mesma complexidade assintótica (como o Merge Sort), o que contribui para sua velocidade na prática. Nossos resultados experimentais com as entradas aleatórias confirmaram essa expectativa, mostrando um aumento no tempo de execução que se alinha claramente com a curva de $n \log n$ para ambas as linguagens, embora com diferenças significativas nas constantes de proporcionalidade entre Python e C.

---

## 1.7 7. Reflexão Final

Ao longo deste projeto, exploramos o algoritmo QuickSort tanto em sua fundamentação teórica quanto em sua aplicação prática, comparando o desempenho entre implementações em Python e C. As análises e simulações nos permitiram extrair conclusões importantes sobre complexidade de algoritmos e as características de performance das linguagens.

- **O algoritmo QuickSort pertence à classe P?** Sim, o algoritmo QuickSort, e o problema de ordenação que ele resolve, **pertencem à classe de complexidade P (Tempo Polinomial)**. A classe P engloba todos os problemas que podem ser resolvidos por um algoritmo determinístico em um tempo que é limitado por um polinômio em relação ao tamanho da entrada ($n$). Como o QuickSort apresenta tempos de execução de $O(n \log n)$ no melhor e caso médio, e $O(n^2)$ no pior caso, todos esses são limites de tempo polinomial. Portanto, ele se encaixa na definição de problemas tratáveis.
- **Existe uma versão NP?** A noção de "versão NP" não se aplica diretamente ao algoritmo QuickSort em si, mas sim à classe de problemas. NP (Tempo Polinomial Não-Determinístico) refere-se a problemas para os quais, dada uma suposta solução, é possível **verificá-la** em tempo polinomial por uma máquina determinística. O problema de **verificar se uma lista está ordenada** é trivial: basta percorrer a lista uma vez e checar se cada elemento é menor ou igual ao próximo. Isso pode ser feito em tempo $O(n)$, que é polinomial. Como qualquer problema resolvível em tempo polinomial (P) também é verificável em tempo polinomial (NP), podemos dizer que o problema de ordenação está em P e, consequentemente, em NP ($P \subseteq NP$). O QuickSort é um algoritmo que *resolve* esse problema eficientemente.
- **Há problemas semelhantes que são NP-Completos?** Sim, embora o problema de ordenação simples seja resolvível em tempo polinomial, existem muitos problemas que envolvem a "ordenação" ou "arranjo" de elementos sob restrições mais complexas e que são classificados como **NP-Completos (NPC)**. Problemas NP-Completos são os problemas mais difíceis da classe NP; se um algoritmo eficiente (polinomial) for encontrado para um problema NPC, todos os problemas em NP poderiam ser resolvidos eficientemente. Exemplos de problemas

"semelhantes" que são NPC incluem:

- **Problema do Caixeiro Viajante (Traveling Salesperson Problem - TSP):** O objetivo é encontrar o caminho mais curto que visita cada cidade exatamente uma vez e retorna à cidade de origem. Isso exige encontrar uma *ordem ótima* de visitação das cidades, uma tarefa muito mais complexa do que apenas ordenar uma lista.
- **Problema da Mochila (Knapsack Problem):** Dada uma coleção de itens com pesos e valores, selecionar um subconjunto de itens que maximize o valor total, sem exceder uma capacidade de peso. Envolve a *seleção e combinação ótima* de itens, não apenas uma ordenação linear.
- **Problemas de Escalonamento e Alocação de Recursos:** Encontrar a melhor sequência ou distribuição de tarefas em um sistema para otimizar algum critério (tempo, uso de recursos).

Esses exemplos ilustram que, embora o QuickSort seja uma ferramenta poderosa para a ordenação básica, há uma fronteira de problemas de otimização combinatória que, por sua natureza inerentemente mais complexa, são considerados NP-Completos e para os quais não se conhecem algoritmos eficientes em tempo polinomial. Isso ressalta a importância de entender as classes de complexidade e as limitações inerentes a diferentes problemas computacionais.

---