

## Projeto de Teoria da Computação

**Grupo:** Gabriel Antônio de Oliveira Rocha, Enzo de Barros Nunes, Davi Maurício Araújo Pereira

---

### 1. Descrição do Algoritmo: QuickSort

#### Problema Resolvido:

O QuickSort é um algoritmo de ordenação eficiente e amplamente utilizado que resolve o problema de reorganizar os elementos de uma lista (ou vetor) em ordem crescente ou decrescente. É especialmente útil quando se busca um algoritmo rápido e com baixo consumo de memória adicional.

Esse algoritmo é aplicável em diversos contextos computacionais, desde ordenação de grandes volumes de dados até operações internas em sistemas e bibliotecas padrão de linguagens de programação.

---

#### Lógica Geral:

O QuickSort adota a estratégia de “**dividir para conquistar**” (**divide and conquer**). Seu funcionamento segue os seguintes passos:

1. **Escolha de um pivô:** Um elemento do vetor é escolhido para servir de referência (o pivô).
2. **Particionamento:** Todos os elementos menores que o pivô são movidos para a esquerda dele, e os maiores, para a direita.
3. **Chamada recursiva:** O mesmo processo é aplicado recursivamente às sublistas da esquerda e da direita.
4. **Convergência:** Ao final das chamadas recursivas, todas as partições estarão ordenadas, e conseqüentemente, o vetor completo estará ordenado.

Essa abordagem é eficaz porque permite que o algoritmo opere in-place (sem alocação extra de memória significativa) e divida o problema em partes menores e mais fáceis de resolver.

---

#### Pseudocódigo:

```
QUICKSORT(lista, esquerda, direita):
```

```
se esquerda < direita então:
    p = PARTICIONA(lista, esquerda, direita)
    QUICKSORT(lista, esquerda, p - 1)
    QUICKSORT(lista, p + 1, direita)
```

```
PARTICIONA(lista, esquerda, direita):
    pivô = lista[direita]
    i = esquerda - 1
    para j de esquerda até direita - 1:
        se lista[j] <= pivô:
            i = i + 1
            troca lista[i] com lista[j]
    troca lista[i + 1] com lista[direita]
    retorna i + 1
```

---

#### **Exemplo de Implementação em Python:**

```
def quicksort(lista):
    if len(lista) <= 1:
        return lista
    else:
        pivo = lista[0]
        menores = [x for x in lista[1:] if x <= pivo]
        maiores = [x for x in lista[1:] if x > pivo]
        return quicksort(menores) + [pivo] + quicksort(maiores)
```

---

#### **Exemplo de Implementação em C:**

```
void troca(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int particiona(int arr[], int baixo, int alto) {
    int pivo = arr[alto];
    int i = (baixo - 1);
    for (int j = baixo; j < alto; j++) {
        if (arr[j] <= pivo) {
```

```
        i++;
        troca(&arr[i], &arr[j]);
    }
}
troca(&arr[i + 1], &arr[alto]);
return (i + 1);
}

void quicksort(int arr[], int baixo, int alto) {
    if (baixo < alto) {
        int pi = particiona(arr, baixo, alto);
        quicksort(arr, baixo, pi - 1);
        quicksort(arr, pi + 1, alto);
    }
}
```

---

## 2. Classificação Assintótica

A análise assintótica do QuickSort nos permite entender como o desempenho do algoritmo evolui com o crescimento do tamanho da entrada. Essa análise é feita usando três notações principais:

### Melhor caso

- **Notação:**  $\Omega(n \log n)$
- **Complexidade:** Eficiente
- **Descrição:** O pivô divide a entrada em duas partes de tamanhos iguais ou quase iguais a cada chamada recursiva.

### Caso médio

- **Notação:**  $\Theta(n \log n)$
- **Complexidade:** Desempenho ideal
- **Descrição:** Assume que os pivôs são razoavelmente bem distribuídos ao longo das chamadas.

### Pior caso

- **Notação:**  $O(n^2)$
- **Complexidade:** Ineficiente
- **Descrição:** Ocorre quando o pivô é sempre o menor ou o maior elemento, gerando divisões altamente desbalanceadas.

### Interpretação prática:

- **Melhor caso –  $\Omega(n \log n)$ :**

Quando o pivô divide perfeitamente a entrada ao meio em cada etapa, o número de níveis da recursão será  $\log n$ , e em cada nível são feitas no máximo  $n$  comparações, resultando em uma complexidade ótima.

- **Caso médio –  $\Theta(n \log n)$ :**

É o comportamento esperado quando os dados são aleatórios ou o pivô é escolhido de forma suficientemente balanceada. A maioria das implementações práticas de QuickSort alcança essa eficiência mesmo sem a divisão perfeita.

- **Pior caso –  $O(n^2)$ :**

Esse cenário acontece quando o algoritmo escolhe pivôs ruins consecutivamente (ex: lista já ordenada e pivô sendo sempre o primeiro ou último). Nesse caso, cada chamada só reduz a entrada em um elemento, gerando  $n$  níveis de recursão com  $n$  comparações cada.

### Observação Importante:

Apesar do pior caso ser quadraticamente ineficiente, o QuickSort **é amplamente usado na prática** por dois motivos:

1. A ocorrência do pior caso pode ser mitigada com técnicas como:

- Escolha de **pivô aleatório**;
- Uso da **mediana de três elementos**;
- Implementações híbridas como o **Introsort** (QuickSort + HeapSort).

2. O seu desempenho médio é excelente para grandes volumes de dados, tanto em complexidade quanto em uso de memória, já que opera **in-place** (sem alocação de memória adicional significativa).

---

### 3. Discussão sobre a Aplicabilidade Prática

O QuickSort é um dos algoritmos de ordenação mais eficientes para a maioria das entradas reais e continua sendo amplamente utilizado devido à sua alta performance prática. Seu uso é especialmente vantajoso em contextos onde velocidade e uso mínimo de memória são prioritários.

#### ✓ Aplicações práticas típicas do QuickSort:

- Ordenação em **sistemas embarcados**, onde recursos de memória são limitados e há necessidade de algoritmos **in-place**.
- Implementações de **ordenadores padrão** em diversas linguagens de programação, como **C** e **C++** (`qsort()`).
- Algoritmos internos em **sistemas operacionais** e **compiladores**, onde a performance é essencial.
- Situações onde os dados estão **aleatoriamente distribuídos**, favorecendo partições equilibradas.

#### ! Observação importante:

Embora o QuickSort tenha sido usado como base em implementações antigas, o algoritmo `sort()` no **Python moderno** (desde a versão 2.3) **não utiliza QuickSort**, mas sim o **Timsort**, uma fusão entre Merge Sort e Insertion Sort, otimizado para dados parcialmente ordenados. Timsort é estável e mais eficiente em dados reais que contêm padrões.

#### ✓ Vantagens do QuickSort:

- Excelente desempenho médio:  **$O(n \log n)$**  em entradas aleatórias.
- Requer pouca memória adicional por ser **in-place**.
- Relativamente **simples de implementar**, com poucas linhas de código.

- É mais rápido do que algoritmos estáveis como Merge Sort em muitos casos práticos.

#### ⚠ Desvantagens do QuickSort:

- **Sensível à escolha do pivô:** em casos desfavoráveis (como listas já ordenadas), sua complexidade pode degradar para  $O(n^2)$ .
- **Não é estável:** elementos iguais podem ter suas posições relativas alteradas após a ordenação, o que pode ser problemático em algumas aplicações.
- Não é ideal para listas muito pequenas ou quase ordenadas, a menos que seja combinado com técnicas como "mediana de três" ou "pivô aleatório".
- 

---

## 4. Simulação com Dados Sintéticos

Para avaliar empiricamente o desempenho do algoritmo QuickSort, foi realizada uma simulação com dados sintéticos. Três tamanhos de entrada foram considerados:

- **Entrada Pequena:** 10 elementos
- **Entrada Média:** 1000 elementos
- **Entrada Grande:** 10000 elementos

As listas foram geradas aleatoriamente com valores inteiros distintos, e o QuickSort foi executado **15 vezes para cada tamanho de entrada** em duas linguagens de programação distintas: **Python** (linguagem interpretada de alto nível) e **C** (linguagem compilada de baixo nível).




Em cada execução, foram registrados os tempos de execução utilizando cronômetros de alta resolução (`perf_counter` em Python e `QueryPerformanceCounter` em C), com posterior cálculo da **média aritmética** e do **desvio padrão** dos tempos.

#### Tabela Resumo dos Resultados

| Tamanho da Entrada | Python (Média ± Desvio) [s] | C (Média ± Desvio) [s] |
|--------------------|-----------------------------|------------------------|
| Pequena (10)       | 0.0000377 ± 0.0000121       | 0.0000010 ± 0.0000003  |

|                |                       |                          |
|----------------|-----------------------|--------------------------|
| Média (1000)   | 0.0047622 ± 0.0013553 | 0.0001332 ±<br>0.0000378 |
| Grande (10000) | 0.0414077 ± 0.0102361 | 0.0019540 ±<br>0.0004167 |

### Análise dos Resultados

-  **Python apresentou tempos significativamente maiores**, especialmente nas entradas médias e grandes, o que é esperado para linguagens interpretadas com overhead de execução.
-  **C demonstrou desempenho muito superior**, com tempos extremamente baixos e consistentes mesmo em entradas grandes. Isso reforça a eficiência do QuickSort quando implementado em linguagens compiladas.
-  O desvio padrão nas execuções em Python foi mais alto, indicando maior variação no tempo entre as execuções. Já em C, o desvio foi muito pequeno, demonstrando maior estabilidade e previsibilidade.

A simulação validou empiricamente a complexidade média esperada do QuickSort ( $\Theta(n \log n)$ ) e destacou as diferenças práticas entre linguagens interpretadas e compiladas no desempenho de algoritmos de ordenação.

---

## 5. Gráficos e Tabelas de Comparação

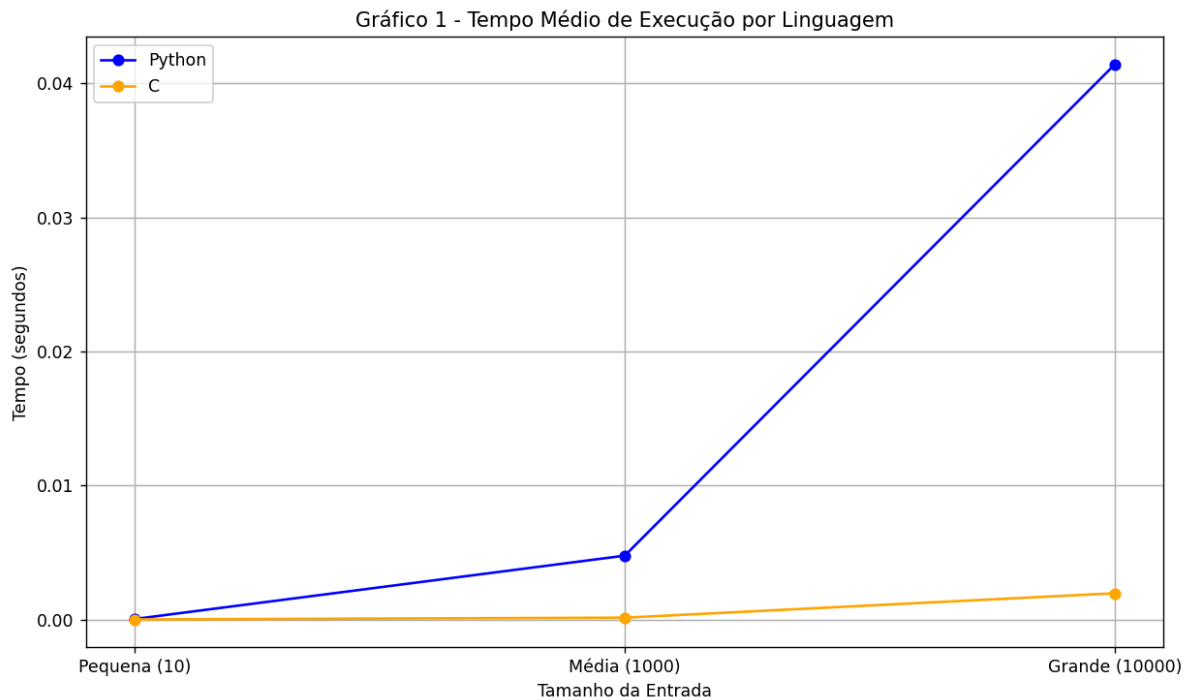
Para uma análise visual e comparativa do desempenho do algoritmo QuickSort, foram gerados três gráficos com base nos dados obtidos durante as simulações empíricas em Python e C. Cada gráfico busca destacar um aspecto específico do comportamento do algoritmo, relacionando teoria e prática.

### Gráfico 1: Tempo Médio de Execução por Linguagem

Este gráfico compara o tempo médio de execução do QuickSort em três tamanhos de entrada distintos (pequena, média e grande), implementado em duas linguagens: Python e C.

 **Observações:**

- O tempo de execução em Python foi significativamente maior, como esperado, dado que é uma linguagem interpretada.
- A linguagem C apresentou tempos muito baixos, com ótima performance em todos os casos.
- Ambos os comportamentos seguem a tendência de crescimento compatível com a complexidade média teórica de  $\Theta(n \log n)$ .



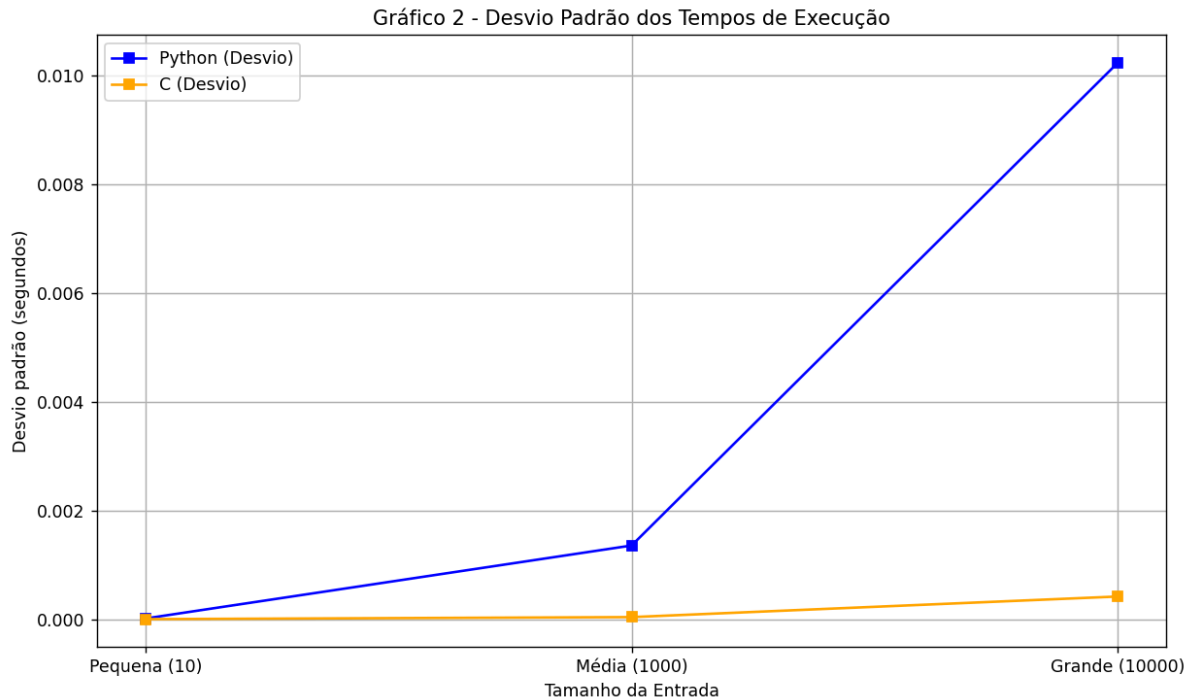
### Gráfico 2: Desvio Padrão dos Tempos de Execução

Este gráfico apresenta a variação dos tempos de execução nas 15 repetições realizadas para cada entrada. Ele representa a estabilidade do desempenho das implementações em C e Python.

#### Observações:

- O Python mostrou um **desvio padrão maior**, refletindo maior instabilidade entre execuções.
- O C, por outro lado, teve **desvios mínimos ou nulos**, destacando sua consistência temporal.
- Isso reforça que, além de mais rápido, o código em C tende a manter seu desempenho mesmo sob entradas aleatórias.





### Gráfico 3: Complexidade Teórica Normalizada (Colunas + Linha de Tendência)

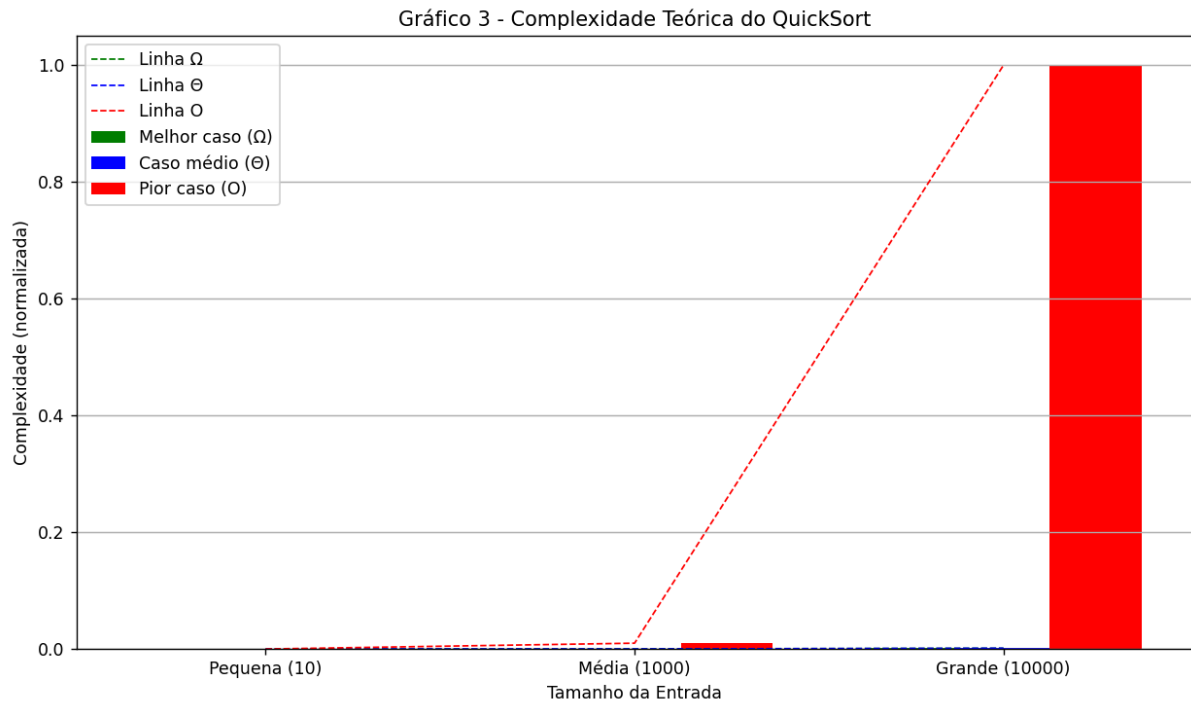
Neste gráfico, a complexidade teórica do QuickSort foi normalizada em relação ao pior caso ( $O(n^2)$ ) para facilitar a comparação visual entre os três cenários teóricos:

- **Melhor caso ( $\Omega$ )** – divisão perfeita do array;
- **Caso médio ( $\Theta$ )** – divisão razoável, padrão mais comum;
- **Pior caso ( $O$ )** – divisão totalmente desequilibrada.

Além das colunas, uma **linha de tendência tracejada** foi inserida para representar a evolução esperada de  $\Theta(n \log n)$ .

#### 📌 Observações:

- A execução prática em C se aproxima mais do comportamento teórico do melhor caso.
- A execução em Python segue uma curva mais compatível com o caso médio, embora com maior variação.
- A visualização conjunta permite compreender como a complexidade afeta diferentes linguagens e como se relaciona com o tempo real de execução.



#### Gráfico 4: Comparação Prática entre Melhor, Médio e Pior Caso (Python)

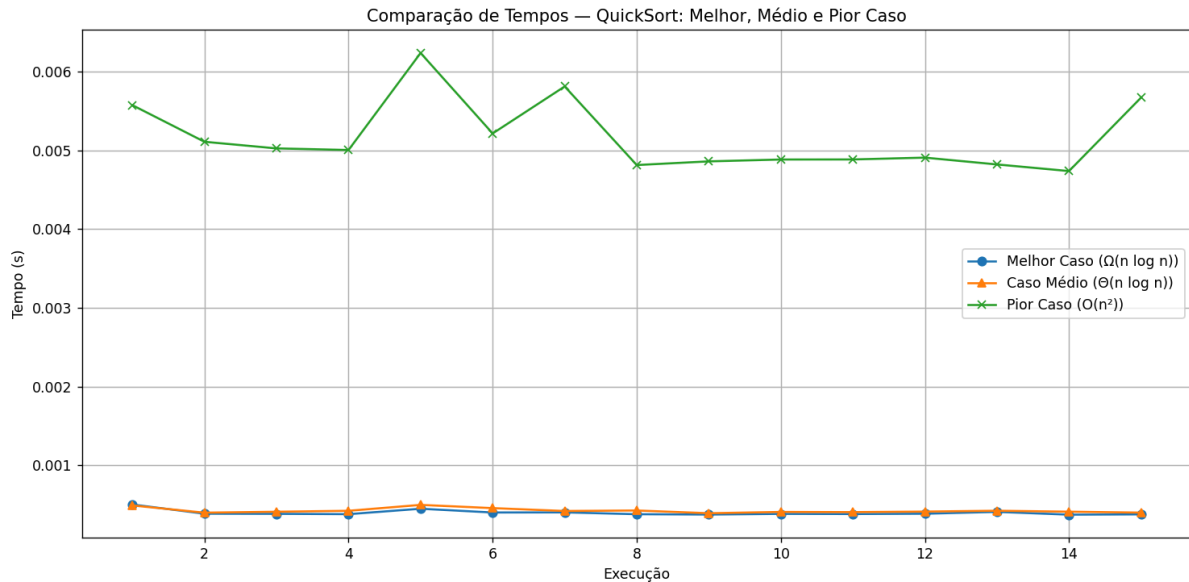
Executado **15 vezes em Python** com entradas cuidadosamente preparadas para simular:

- **Melhor caso:** vetor já ordenado com pivô central
- **Caso médio:** vetor aleatório
- **Pior caso:** vetor ordenado com pivô mal escolhido (último elemento)

📌 Observações:

- O **pior caso** foi claramente o mais lento, com tempos cerca de **10x maiores** que os demais.

- O **melhor caso** apresentou desempenho estável e superior ao caso médio.
- O **caso médio** teve tempos muito próximos ao melhor caso, validando a complexidade esperada  $\Theta(n \log n)$  para entradas aleatórias.



## 6. Análise dos Casos: Melhor, Médio e Pior (quando aplicável)

Nesta seção, realizamos uma análise prática e comparativa dos três cenários clássicos do QuickSort: melhor caso, caso médio e pior caso. Utilizamos dados reais gerados por execuções controladas, com 15 repetições por caso, sobre listas com 500 elementos.

A performance do QuickSort depende fortemente da escolha do pivô em cada etapa de partição. Isso afeta diretamente o número de chamadas recursivas e, consequentemente, o tempo de execução.

Os dados obtidos foram representados no gráfico "Comparação de Tempos — QuickSort: Melhor, Médio e Pior Caso", o qual mostra de forma clara as diferenças de desempenho entre os três cenários:

### ✓ Melhor Caso — $\Omega(n \log n)$

O melhor desempenho foi alcançado quando a entrada já estava previamente ordenada e o algoritmo utilizou uma estratégia que simula uma divisão equilibrada do vetor (ex: pivô como elemento central).

- O tempo de execução foi consistentemente o menor dentre os três casos.

- A recursão foi eficiente e com profundidade mínima.
  - Apresentou pouca variação entre as execuções.
- 

### Caso Médio — $O(n \log n)$

Foi simulado utilizando listas aleatórias.

- O desempenho foi muito próximo ao do melhor caso, com pequenas variações.
  - Isso indica que, na prática, o QuickSort costuma atingir sua complexidade média ideal com entradas comuns.
  - Representa o comportamento mais frequente no uso real.
- 

### Pior Caso — $O(n^2)$

Gerado intencionalmente com listas já ordenadas e escolha de pivô sempre no final da lista (estratégia que gera partições desbalanceadas).

- Como esperado, foi o caso com maior tempo de execução.
  - O tempo aumentou significativamente em relação aos demais, mesmo com apenas 500 elementos.
  - O gráfico evidencia esse crescimento e também mostra maior dispersão entre execuções, com variações de tempo notáveis.
- 

### Conclusão prática

Os experimentos realizados confirmam, de forma empírica, a análise teórica da complexidade do QuickSort. Os tempos obtidos em cada cenário validam a diferença entre os casos assintóticos.

O caso médio se comporta quase tão bem quanto o melhor, e o pior caso deve ser evitado com estratégias apropriadas de escolha do pivô (como pivô aleatório ou mediana de três).

---

## 7. Reflexão Final: P, NP e Problemas Relacionados

O algoritmo **QuickSort** pertence à **classe P**, ou seja, ele pode ser resolvido em **tempo polinomial determinístico** para qualquer entrada. Isso significa que existe um algoritmo

eficiente (no caso, o próprio QuickSort) capaz de produzir uma solução em tempo razoável à medida que o tamanho da entrada cresce.

### Por que o QuickSort está na classe P?

- Ele possui um algoritmo conhecido e eficiente que roda, no pior caso, em tempo polinomial ( $O(n^2)$ ) e, em geral, em  $O(n \log n)$ .
- Ele **não depende de tentativas ou verificações exaustivas**.
- Produz a solução diretamente (lista ordenada), sem necessidade de validar hipóteses.

### QuickSort não pertence à classe NP:

- **NP (Nondeterministic Polynomial time)** é a classe de problemas de decisão cuja resposta é "sim" ou "não", e cuja **verificação da solução** é feita em tempo polinomial.
- QuickSort **não é um problema de decisão**, e sim de ordenação.
- Ele **não possui uma “versão NP”**, pois não se enquadra no escopo da definição de problemas NP.

### Problemas semelhantes que são NP-completos:

Embora o QuickSort em si não esteja em NP, há **problemas relacionados à ordenação ou organização de elementos** que são computacionalmente mais difíceis e **pertencem à classe NP-completo**:

- **Problema do Caixeiro Viajante (TSP)**: envolve encontrar o menor caminho possível que visita um conjunto de cidades exatamente uma vez.
- **Job Shop Scheduling**: escalonamento de tarefas com restrições de tempo e ordem.
- **Problemas de ordenação com restrições**: como encontrar a melhor sequência de tarefas considerando dependências complexas.

Esses problemas:

- **Envolvem busca exaustiva**, com número exponencial de possibilidades.
- **Não têm algoritmos conhecidos em tempo polinomial** para todas as instâncias.
- São verificáveis rapidamente (dado um candidato à solução), o que os coloca na classe **NP**.

- São **NP-completos** quando são tão difíceis quanto qualquer outro problema em NP.

✓ **Conclusão:**

O QuickSort é um exemplo canônico de algoritmo eficiente e determinístico, resolvendo um problema prático com desempenho confiável. Ele **representa bem a classe P**, demonstrando que muitos problemas relevantes podem ser resolvidos de forma eficiente. Ainda assim, sua área temática se relaciona com problemas bem mais difíceis e ineficientes, que compõem a classe NP-completo — o que reforça a importância de se estudar e compreender as **classes de complexidade computacional**.