Introdução à Complexidade Computacional

Computabilidade e Complexidade

Márcio Nicolau

2025-10-27

Table of contents

List of Figures

Objetivos da Aula

- Mudar o foco de "O que é computável?" para "O que é computável *eficientemente*?".
- Introduzir a complexidade de tempo e espaço como medidas de recursos.
- Utilizar a notação Big-O para analisar o crescimento de algoritmos.
- Apresentar a classe **P** como a formalização de problemas "tratáveis".

Conteúdo

Da Computabilidade à Complexidade: Uma Nova Pergunta

Até agora, nossa principal pergunta foi binária: um problema é decidível ou indecidível? Vimos que problemas como o da parada são fundamentalmente impossíveis de resolver, não importa quanto tempo ou poder computacional tenhamos.

Agora, vamos focar nos problemas que **são decidíveis**. Só porque um problema é solucionável em teoria, não significa que temos uma solução prática para ele.

> **A Mudança de Paradigma**
>
> - **Teoria da Computabilidade**: Estuda os limites do que é *possível* computar. A resposta é sim/não.
> - **Teoria da Complexidade**: Estuda os limites do que é *prático* computar. A resposta é sobre eficiência e recursos (tempo, espaço).

Considere ordenar uma lista de n números. Existem muitos algoritmos para isso:

- **Bubble Sort**: Funciona, mas é lento para listas grandes.
- **Merge Sort**: Funciona e é significativamente mais rápido.

Ambos os algoritmos *decidem* o problema da ordenação. A teoria da complexidade nos dá as ferramentas para analisar *por que* um é melhor que o outro e para classificar problemas com base nos recursos que eles exigem.

Medindo a Complexidade: Tempo e Espaço

Para analisar a eficiência de um algoritmo de forma rigorosa, precisamos de um modelo. Usaremos a Máquina de Turing como nosso modelo formal, mas as ideias se aplicam diretamente a qualquer linguagem de programação.

As duas principais medidas de recursos são:

- **Complexidade de Tempo**: O número de passos que uma Máquina de Turing (ou operações em um programa) leva para parar, em função do tamanho da entrada (n).
- **Complexidade de Espaço**: O número de células da fita que uma Máquina de Turing (ou quantidade de memória em um programa) utiliza, em função do tamanho da entrada (n).

Definição Formal (Tempo): Seja M uma TM decisora. A **complexidade de tempo** de M é a função $f : \mathbb{N} \to \mathbb{N}$, onde $f(n)$ é o número máximo de passos que M usa em qualquer entrada de tamanho n. (Sipser, 2012, p. 277)

Análise Assintótica: A Notação Big-O

Ao analisar algoritmos, não nos importamos com o tempo exato em milissegundos, que depende do hardware. Estamos interessados no **crescimento** do tempo de execução à medida que o tamanho da entrada (n) aumenta. A **notação Big-O** é a linguagem que usamos para descrever esse comportamento assintótico.

- $O(n)$ **- Linear**: O tempo de execução cresce linearmente com a entrada. (Ex: encontrar o maior elemento em uma lista).
- $O(n^2)$ **- Quadrático**: O tempo cresce com o quadrado da entrada. (Ex: Bubble Sort).
- $O(n \log n)$ **- Log-linear**: Crescimento muito eficiente. (Ex: Merge Sort).
- $O(2^n)$ **- Exponencial**: O tempo dobra a cada novo elemento na entrada. Geralmente considerado "intratável" para entradas não triviais.

Exemplo em Python: Comparando Crescimentos

```
import math
```

```python
def O_1(n): # Constante
    return 1

def O_log_n(n): # Logarítmico
    return math.log2(n) if n > 0 else 0

def O_n(n): # Linear
    return n

def O_n_log_n(n): # Log-linear
    return n * math.log2(n) if n > 0 else 0

def O_n2(n): # Quadrático
    return n**2

def O_2_n(n): # Exponencial
    return 2**n

# Vamos ver o número de operações para diferentes tamanhos de n
inputs = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
print(f"{'n':<5} | {'O(log n)':<10} | {'O(n)':<10} | {'O(n log n)':<12} | {'O(n^2)':<10} | {'O(2^n)':<2
print("-" * 75)

for n in inputs:
    print(f"{n:<5} | {O_log_n(n):<10.2f} | {O_n(n):<10} | {O_n_log_n(n):<12.2f} | {O_n2(n):<10} | {O_2_
```

```
n     | O(log n)   | O(n)       | O(n log n)   | O(n^2)     | O(2^n)
---------------------------------------------------------------------------
1     | 0.00       | 1          | 0.00         | 1          | 2
2     | 1.00       | 2          | 2.00         | 4          | 4
3     | 1.58       | 3          | 4.75         | 9          | 8
4     | 2.00       | 4          | 8.00         | 16         | 16
5     | 2.32       | 5          | 11.61        | 25         | 32
6     | 2.58       | 6          | 15.51        | 36         | 64
7     | 2.81       | 7          | 19.65        | 49         | 128
8     | 3.00       | 8          | 24.00        | 64         | 256
9     | 3.17       | 9          | 28.53        | 81         | 512
10    | 3.32       | 10         | 33.22        | 100        | 1,024
11    | 3.46       | 11         | 38.05        | 121        | 2,048
12    | 3.58       | 12         | 43.02        | 144        | 4,096
13    | 3.70       | 13         | 48.11        | 169        | 8,192
14    | 3.81       | 14         | 53.30        | 196        | 16,384
15    | 3.91       | 15         | 58.60        | 225        | 32,768
16    | 4.00       | 16         | 64.00        | 256        | 65,536
17    | 4.09       | 17         | 69.49        | 289        | 131,072
```

```
18    | 4.17        | 18          | 75.06         | 324         | 262,144
19    | 4.25        | 19          | 80.71         | 361         | 524,288
20    | 4.32        | 20          | 86.44         | 400         | 1,048,576
```

Observe o crescimento explosivo de $O(2^n)$. Para $n = 40$, já estamos na casa dos trilhões de operações, tornando o algoritmo impraticável.

A Classe P: Problemas Tratáveis

A primeira e mais importante classe de complexidade é a **Classe P**. Ela representa o conjunto de todos os problemas que podem ser resolvidos "eficientemente".

! Definição: A Classe P

P é a classe de linguagens que são decidíveis por uma Máquina de Turing determinística em **tempo polinomial**.
Ou seja, $L \in P$ se existe um algoritmo que decide L com complexidade de tempo $O(n^k)$ para alguma constante k.

A Tese de Cobham-Edmonds postula que a classe P corresponde à nossa noção intuitiva de problemas "praticamente solucionáveis" ou "tratáveis". Embora um algoritmo $O(n^{100})$ não seja prático, a grande maioria dos problemas polinomiais encontrados na prática tem expoentes pequenos (como 2 ou 3).

Exemplos de Problemas em P:

- **PATH**: Dado um grafo e dois vértices, existe um caminho entre eles? (Resolvível com busca em largura, $O(V + E)$).
- **Ordenação**: Ordenar uma lista de números. (Merge sort, $O(n \log n)$).
- **Teste de Primalidade**: Determinar se um número é primo. (Algoritmo AKS, polinomial).

Diagrama: A Fronteira da Tratabilidade

Exercícios de Verificação

i Atividade Prática

1. **Análise de Algoritmo**: Qual é a complexidade de tempo (em notação Big-O) do seguinte algoritmo Python que verifica se uma lista contém elementos duplicados? Justifique.

```python
def has_duplicates(items):
    for i in range(len(items)):
        for j in range(i + 1, len(items)):
            if items[i] == items[j]:
                return True
    return False
```

2. **Tratável vs. Intratável**: Um engenheiro desenvolveu um novo algoritmo para um problema. A análise de complexidade mostra que ele roda em tempo $O(n^{\log n})$. Este algoritmo é considerado

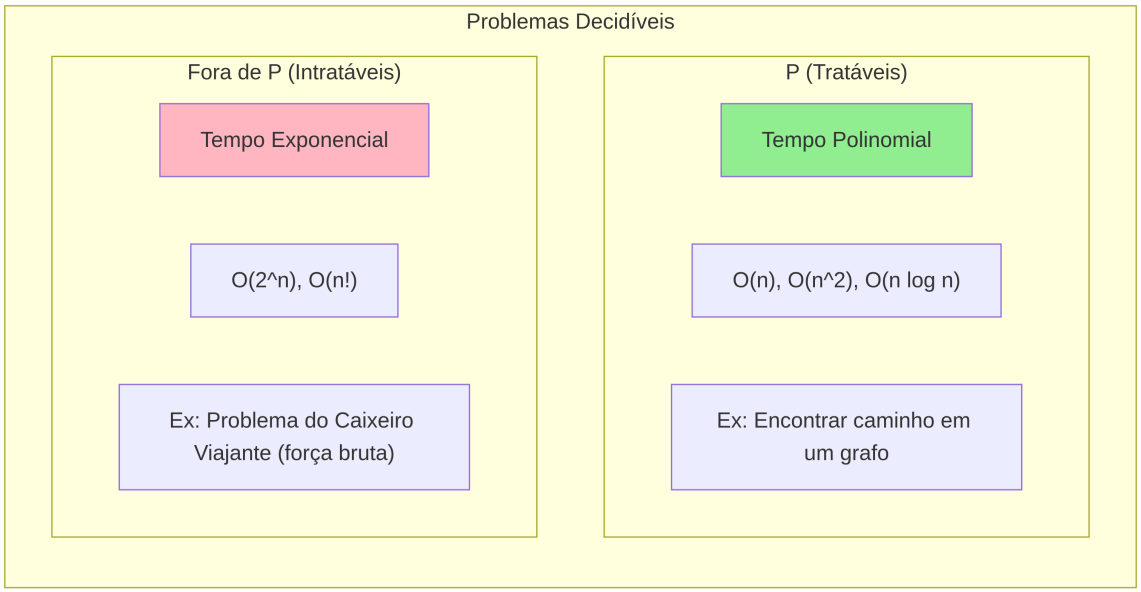

Figure 1: A classe P separa os problemas considerados tratáveis dos intratáveis.

polinomial? O problema está na classe P? Por quê?

3. **Complexidade de Espaço**: Considere o algoritmo da questão 1. Qual é a sua complexidade de **espaço**? Ele aloca nova memória que depende do tamanho da entrada `items`?

Referências Bibliográficas

SIPSER, Michael. **Introdução à Teoria da Computação**. 3. ed. São Paulo, Brasil: Cengage Learning, 2012.