

Classes de Complexidade de Tempo: P e NP

Computabilidade e Complexidade

Márcio Nicolau

2025-11-03

Table of contents

Objetivos da Aula	1
Conteúdo	1
Revisitando a Classe P	1
A Classe NP: O Poder da Verificação	2
A Grande Questão: P vs. NP	4
Exemplos em Python: Resolver vs. Verificar	5
Propriedades Importantes da Classe NP	11
Exercícios de Verificação	12
Perspectivas e Importância Histórica	12
Referências Bibliográficas	13

List of Figures

1	A relação (acreditada) entre as classes P e NP.	5
2	Hierarquia das classes de complexidade (conjectural).	6

Objetivos da Aula

- Definir formalmente a classe **P** de problemas resolvíveis em tempo polinomial.
- Definir a classe **NP** usando a noção de verificação em tempo polinomial.
- Distinguir claramente entre resolver um problema e verificar uma solução.
- Apresentar a questão **P vs. NP** como o problema em aberto mais importante da ciência da computação.

Conteúdo

Revisitando a Classe P

Na aula anterior, introduzimos a **Classe P** como nossa formalização para problemas “eficientemente solucionáveis” ou “tratáveis”.

Definição Essencial: Classe P

P é a classe de linguagens (problemas de decisão) que podem ser **decididas** por uma Máquina de Turing **determinística** em tempo **polinomial**.

Formalmente:

$$P = \bigcup_{k \geq 1} \text{TIME}(n^k)$$

Em termos simples: se um problema está em P, existe um algoritmo para ele que roda em tempo $O(n^k)$ para alguma constante k , onde n é o tamanho da entrada.

Problemas em P são ótimos de se ter. Encontrar um caminho em um grafo, ordenar uma lista, multiplicar matrizes — todos eles têm algoritmos eficientes que podemos executar em computadores reais.

Exemplos Clássicos de Problemas em P

Vejamos alguns exemplos fundamentais de problemas que estão em P, conforme apresentados em (Sipser, 2012):

1. **PATH** = $\{\langle G, s, t \rangle \mid G \text{ é um grafo direcionado que contém um caminho direcionado de } s \text{ para } t\}$
 - **Algoritmo:** Busca em largura (BFS) ou busca em profundidade (DFS)
 - **Complexidade:** $O(m + n)$ onde m é o número de arestas e n o número de vértices
2. **RELPRIME** = $\{\langle x, y \rangle \mid x \text{ e } y \text{ são inteiros relativamente primos}\}$
 - **Algoritmo:** Algoritmo de Euclides para calcular o MDC
 - **Complexidade:** $O(n)$ onde n é o número de bits na entrada
3. **PRIMES** = $\{\langle p \rangle \mid p \text{ é um número primo}\}$
 - **Algoritmo:** Teste de primalidade AKS (Agrawal-Kayal-Saxena)
 - **Complexidade:** $O((\log n)^{12})$ - polinomial no tamanho da entrada (Sipser, 2012)

Por que tempo polinomial?

A escolha de tempo polinomial como definição de “eficiente” não é arbitrária:

1. **Robustez:** A classe P é invariante sob diferentes modelos de computação razoáveis
2. **Fechamento:** P é fechada sob operações naturais como composição
3. **Correspondência prática:** Na prática, algoritmos polinomiais tendem a ser viáveis, enquanto algoritmos exponenciais não são
4. **Distinção clara:** A diferença entre n^{100} e 2^n é significativa conforme n cresce

A Classe NP: O Poder da Verificação

Agora, vamos considerar um tipo diferente de problema. Imagine que você está tentando resolver um quebra-cabeça Sudoku.

- **Resolver o Sudoku do zero:** Pode ser muito difícil. Você pode precisar tentar várias combinações, voltar atrás (backtracking), e isso pode levar muito tempo.
- **Verificar uma solução completa:** Se alguém lhe entregar um Sudoku preenchido e disser “aqui está a solução”, é extremamente **fácil e rápido** verificar se a solução está correta. Você só precisa checar

cada linha, coluna e bloco para garantir que não há números repetidos.

Essa distinção entre “difícil de resolver, mas fácil de verificar” é a essência da **Classe NP**.

! Definição Formal: Classe NP

NP é a classe de linguagens que possuem **verificadores** em tempo polinomial.

Uma linguagem L está em NP se existe um algoritmo V (o verificador) e uma constante k tal que:

$$w \in L \iff \exists c, |c| \leq |w|^k, \text{ tal que } V(\langle w, c \rangle) \text{ aceita em tempo polinomial.}$$

Onde:

- w é a instância do problema (o tabuleiro Sudoku vazio).
- c é o “certificado” ou “testemunha” (o Sudoku preenchido).
- V é o verificador que, dado o problema e o certificado, confirma a solução rapidamente.

NP significa **Nondeterministic Polynomial Time** (Tempo Polinomial Não-Determinístico). Uma definição alternativa e equivalente é que NP é a classe de problemas que podem ser resolvidos em tempo polinomial por uma Máquina de Turing **Não-Determinística**. A NTM “adivinha” magicamente o certificado correto e depois o verifica.

Definição Alternativa: NP via Máquinas de Turing Não-Determinísticas

i Definição Alternativa de NP

Uma linguagem L está em **NP** se existe uma Máquina de Turing Não-Determinística (MTN) que decide L em tempo polinomial.

Formalmente:

$$\text{NP} = \bigcup_{k \geq 1} \text{NTIME}(n^k)$$

onde $\text{NTIME}(t(n))$ é a classe de linguagens decidíveis por uma MTN em tempo $O(t(n))$.

Teorema (Sipser, 2012): As duas definições de NP são equivalentes: 1. Linguagens com verificadores em tempo polinomial 2. Linguagens decidíveis por MTN em tempo polinomial

Prova (Esboço):

- ($\Rightarrow$) Se L tem verificador V em tempo $O(n^k)$, construímos uma MTN N que:
 1. Não-deterministicamente “adivinha” um certificado c de tamanho $\leq n^k$
 2. Executa V deterministicamente em $\langle w, c \rangle$
 3. Aceita se V aceitaO tempo total é polinomial.
- ($\Leftarrow$) Se L é decidida por MTN N em tempo $O(n^k)$, construímos um verificador V que:
 1. Recebe entrada $\langle w, c \rangle$ onde c codifica as escolhas não-determinísticas de N
 2. Simula N em w seguindo as escolhas especificadas em c

3. Aceita se a simulação aceita

O tempo de verificação é polinomial. $\square$

Exemplos Clássicos de Problemas em NP

Vejamos exemplos detalhados conforme (Sipser, 2012):

1. **HAMPATH** = $\{\langle G, s, t \rangle \mid G \text{ é um grafo direcionado contendo um caminho hamiltoniano de } s \text{ para } t\}$
 - **Certificado:** Uma sequência de vértices $v_1, v_2, \dots, v_n$
 - **Verificador:** Verifica que:
 - $v_1 = s$ e $v_n = t$
 - Todos os vértices são distintos
 - (v_i, v_{i+1}) é uma aresta para todo i
 - **Tempo:** $O(n^2)$ onde n é o número de vértices
2. **CLIQUE** = $\{\langle G, k \rangle \mid G \text{ é um grafo não-direcionado contendo uma } k\text{-clique}\}$
 - **Certificado:** Um subconjunto C de k vértices
 - **Verificador:** Verifica que todo par de vértices em C está conectado por uma aresta
 - **Tempo:** $O(k^2)$
3. **SUBSET-SUM** = $\{\langle S, t \rangle \mid S \text{ é um multiconjunto de inteiros cuja soma de algum subconjunto é } t\}$
 - **Certificado:** Um subconjunto $R \subseteq S$
 - **Verificador:** Soma os elementos de R e verifica se a soma é t
 - **Tempo:** $O(n)$ onde $n = |S|$
4. **SAT** = $\{\langle \phi \rangle \mid \phi \text{ é uma fórmula booleana satisfazível}\}$
 - **Certificado:** Uma atribuição de valores para as variáveis
 - **Verificador:** Avalia ϕ sob a atribuição
 - **Tempo:** $O(n)$ onde n é o tamanho de ϕ

! Observação Fundamental

Observe que, para todos esses problemas:

- Não conhecemos algoritmos polinomiais para **resolvê-los** (encontrar o certificado)
- Mas temos algoritmos polinomiais **triviais** para **verificar** uma solução proposta

Esta é a essência da diferença entre P e NP!

A Grande Questão: P vs. NP

É fácil ver que todo problema em P também está em NP. Se podemos *resolver* um problema em tempo polinomial, podemos certamente *verificar* uma solução em tempo polinomial (basta ignorar o certificado e resolver o problema do zero). Portanto, $P \subseteq NP$.

A questão de um milhão de dólares, literalmente (oferecido pelo Clay Mathematics Institute), é:

P = NP ?

- Se $P = NP$, isso significaria que todo problema para o qual uma solução pode ser verificada rapidamente também pode ser *resolvido* rapidamente. Isso teria consequências revolucionárias para a ciência, engenharia, economia e, especialmente, criptografia (que se baseia na dificuldade de certos problemas em NP).
- Se $P \neq NP$, como a maioria dos cientistas da computação acredita, isso significa que existem problemas (como SAT e HAMPATH) que são fundamentalmente mais difíceis de resolver do que de verificar.

Diagrama: A Relação entre P e NP

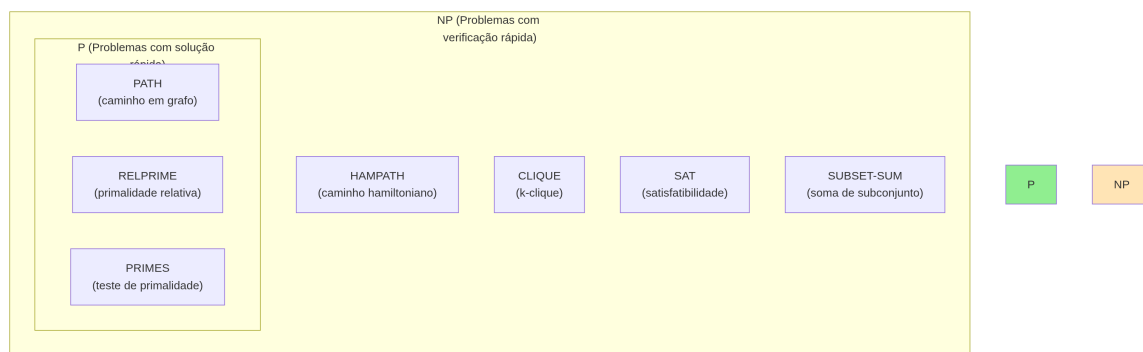


Figure 1: A relação (acreditada) entre as classes P e NP.

Aqui está um diagrama mais completo mostrando também co-NP:

i Interpretação dos Diagramas

- **Verde (P):** Problemas que sabemos resolver eficientemente
- **Amarelo (NP):** Problemas onde podemos verificar soluções eficientemente
- **Azul (co-NP):** Complementos de problemas em NP
- **Rosa (EXPTIME):** Problemas resolvíveis em tempo exponencial

Questões em aberto:

- $P = NP$? (Acredita-se que não)
- $NP = co-NP$? (Acredita-se que não)
- $NP = EXPTIME$? (Acredita-se que sim, mas não provado!)

Exemplos em Python: Resolver vs. Verificar

Vamos ilustrar a diferença P vs. NP com problemas concretos, demonstrando a disparidade entre resolver e verificar.

Exemplo 1: SUBSET-SUM

O problema **SUBSET-SUM** é um exemplo clássico em ([Sipser, 2012](#)):

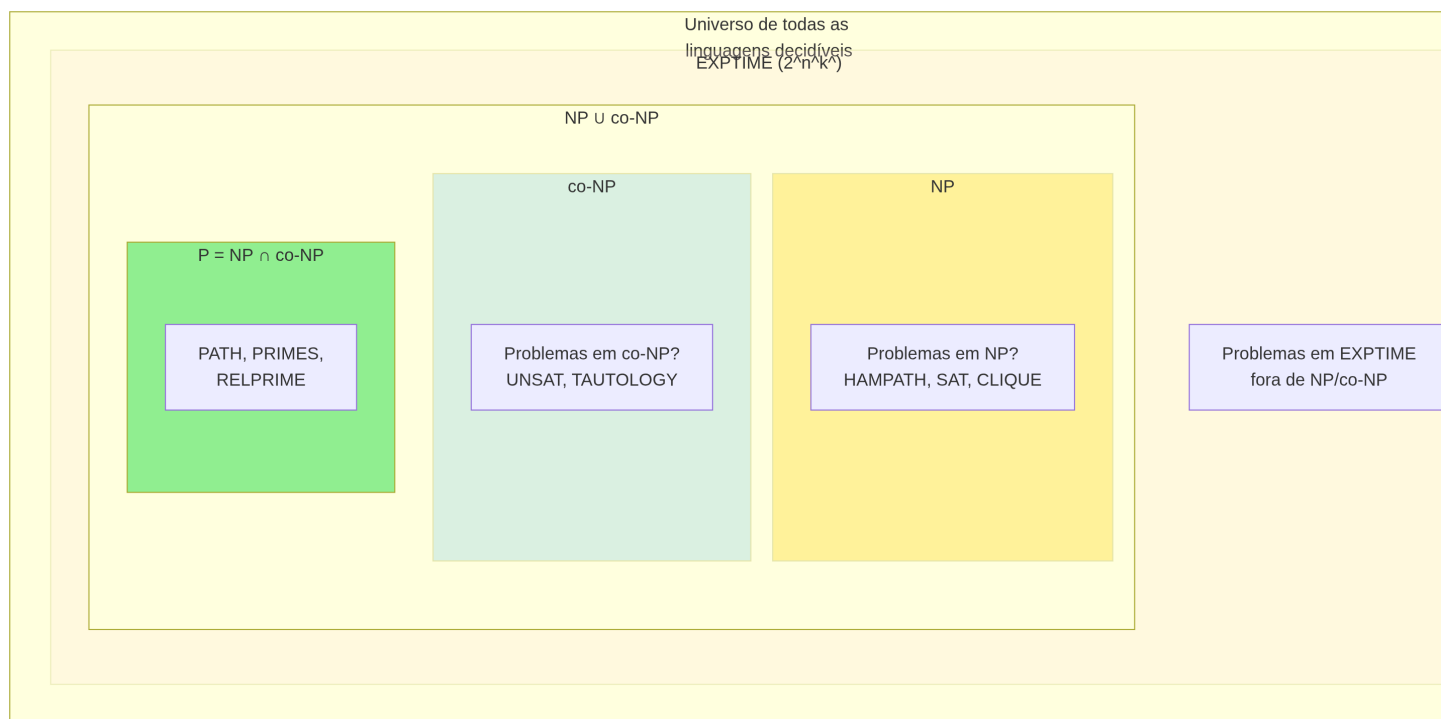


Figure 2: Hierarquia das classes de complexidade (conjectural).

- **Problema:** Dado um conjunto de inteiros S e um alvo t , existe um subconjunto de S cuja soma é exatamente t ?

```
# --- O Verificador (Polinomial) ---
def verificar_subset_sum(conjunto, alvo, certificado):
    """
    Verifica se o 'certificado' (um subconjunto) é uma solução válida.
    Esta operação é RÁPIDA (linear no tamanho do conjunto).
    """
    soma_certificado = sum(certificado)

    # Verifica se a soma está correta e se todos os elementos do certificado
    # realmente pertencem ao conjunto original.
    if soma_certificado != alvo:
        return False

    return all(item in conjunto for item in certificado)

# --- O Resolvedor (Exponencial, força bruta) ---
def resolver_subset_sum_forca_bruta(conjunto, alvo):
    """
    Tenta encontrar uma solução por força bruta.
    Testa TODOS os 2^n subconjuntos possíveis.
    Esta operação é LENTA (exponencial).
    """
    n = len(conjunto)
    # Itera de 1 até 2^n - 1
    for i in range(1, 1 << n):
        soma_atual = 0
        subconjunto_atual = []
        for j in range(n):
            # Usa máscaras de bits para gerar todos os subconjuntos
            if (i >> j) & 1:
                soma_atual += conjunto[j]
                subconjunto_atual.append(conjunto[j])

        if soma_atual == alvo:
            return subconjunto_atual # Encontrou a solução

    return None # Nenhuma solução encontrada

# --- Exemplo de Uso ---
meu_conjunto = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
meu_alvo = 9

print("--- Verificando uma solução (rápido) ---")
```

```

solucao_proposta = [3, 1, 5]
resultado_verificacao = verificar_subset_sum(meu_conjunto, meu_alvo, solucao_proposta)
print(f"A solução {solucao_proposta} é válida? {resultado_verificacao}")

print("\n--- Resolvendo do zero (lento) ---")
solucao_encontrada = resolver_subset_sum_forca_bruta(meu_conjunto, meu_alvo)
print(f"Solução encontrada por força bruta: {solucao_encontrada}")

# Demonstrando a diferença de complexidade
print("\n--- Análise de Complexidade ---")
print(f"Tamanho do conjunto: {len(meu_conjunto)}")
print(f"Número de subconjuntos possíveis:  $2^{\text{len}(\text{meu\_conjunto})} = {2**\text{len}(\text{meu\_conjunto})}$ ")
print(f"Tempo do verificador:  $O(n) = O(\text{len}(\text{meu\_conjunto}))$ ")
print(f"Tempo do resolvidor:  $O(2^n) = O({2**\text{len}(\text{meu\_conjunto})})$ ")

```

```

--- Verificando uma solução (rápido) ---
A solução [3, 1, 5] é válida? True

```

```

--- Resolvendo do zero (lento) ---
Solução encontrada por força bruta: [3, 1, 4, 1]

```

```

--- Análise de Complexidade ---
Tamanho do conjunto: 11
Número de subconjuntos possíveis:  $2^{11} = 2048$ 
Tempo do verificador:  $O(n) = O(11)$ 
Tempo do resolvidor:  $O(2^n) = O(2048)$ 

```

Exemplo 2: HAMPATH (Caminho Hamiltoniano)

Vamos implementar verificador e resolvidor para o problema do caminho hamiltoniano:

```

from itertools import permutations

def verificar_hampath(grafo, inicio, fim, certificado):
    """
    Verifica se 'certificado' é um caminho hamiltoniano válido de inicio até fim.
    Tempo:  $O(n^2)$  onde n é o número de vértices.

    Args:
        grafo: dicionário de adjacências {vértice: [vizinhos]}
        inicio: vértice inicial
        fim: vértice final
        certificado: lista de vértices representando o caminho
    """
    # Verifica se começa e termina nos vértices corretos
    if len(certificado) == 0 or certificado[0] != inicio or certificado[-1] != fim:

```



```

        return False

# Verifica se todos os vértices aparecem exatamente uma vez
if len(certificado) != len(set(certificado)) or len(certificado) != len(grafo):
    return False

# Verifica se há arestas consecutivas
for i in range(len(certificado) - 1):
    if certificado[i+1] not in grafo.get(certificado[i], []):
        return False

return True

def resolver_hamiltoniano_forca_bruta(grafo, inicio, fim):
    """
    Encontra um caminho hamiltoniano por força bruta.
    Tempo: O(n!) - MUITO lento!
    """
    vertices = list(grafo.keys())

    # Remove inicio e fim da lista de vértices intermediários
    vertices_meio = [v for v in vertices if v != inicio and v != fim]

    # Testa todas as permutações dos vértices intermediários
    for perm in permutations(vertices_meio):
        caminho = [inicio] + list(perm) + [fim]

        # Verifica se este caminho é válido
        valido = True
        for i in range(len(caminho) - 1):
            if caminho[i+1] not in grafo.get(caminho[i], []):
                valido = False
                break

        if valido:
            return caminho

    return None

# --- Exemplo de Uso ---
# Grafo exemplo: um ciclo com 5 vértices
grafo_exemplo = {
    'A': ['B', 'E'],
    'B': ['A', 'C'],
    'C': ['B', 'D'],
    'D': ['C', 'E'],
    'E': ['D', 'A'],
}

```

```

        'D': ['C', 'E'],
        'E': ['D', 'A']
    }

    print("=== HAMPATH: Caminho Hamiltoniano ===\n")
    print(f"Grafo: {grafo_exemplo}\n")

    # Testando o verificador
    certificado_valido = ['A', 'B', 'C', 'D', 'E']
    certificado_invalido = ['A', 'C', 'B', 'D', 'E']

    print("--- Verificação (rápida) ---")
    print(f"Caminho {certificado_valido}: {verificar_hamopath(grafo_exemplo, 'A', 'E', certificado_valido)}")
    print(f"Caminho {certificado_invalido}: {verificar_hamopath(grafo_exemplo, 'A', 'E', certificado_invalido)}")

    print("\n--- Resolução por força bruta (lenta) ---")
    solucao = resolver_hamopath_forca_bruta(grafo_exemplo, 'A', 'E')
    print(f"Caminho encontrado: {solucao}")

    print("\n--- Análise de Complexidade ---")
    n = len(grafo_exemplo)
    print(f"Número de vértices: {n}")
    print(f"Permutações a testar: (n-2)! = {n-2}!   {eval(f'__import__(\"math\").factorial({n-2})')}")
    print(f"Tempo do verificador:  $O(n^2) = O({n**2})$ ")
    print(f"Tempo do resolvedor:  $O(n!) =$  crescimento super-exponencial")

```

=== HAMPATH: Caminho Hamiltoniano ===

Grafo: {'A': ['B', 'E'], 'B': ['A', 'C'], 'C': ['B', 'D'], 'D': ['C', 'E'], 'E': ['D', 'A']}

--- Verificação (rápida) ---

Caminho ['A', 'B', 'C', 'D', 'E']: True

Caminho ['A', 'C', 'B', 'D', 'E']: False

--- Resolução por força bruta (lenta) ---

Caminho encontrado: ['A', 'B', 'C', 'D', 'E']

--- Análise de Complexidade ---

Número de vértices: 5

Permutações a testar: (n-2)! = 3! = 6

Tempo do verificador: $O(n^2) = O(25)$

Tempo do resolvedor: $O(n!) =$ crescimento super-exponencial

⚠ Explosão Combinatória

Observe que para HAMPATH com apenas 10 vértices:

- O verificador executa em ~ 100 operações (10^2)
- O resolvidor força-bruta executa em ~ 40.320 operações ($8!$)
- Com 20 vértices, seria $18! \approx 6.4 \times 10^{15}$ operações!

Esta é a **explosão combinatória** que torna esses problemas intratáveis na prática.

O código demonstra que a **verificação** é trivial e rápida, enquanto a **resolução** por força bruta é muito lenta. A questão P vs. NP, para estes problemas, é: será que existe um algoritmo *inteligente* que rode em tempo polinomial? (Até hoje, ninguém encontrou um).

Propriedades Importantes da Classe NP

Antes dos exercícios, vejamos algumas propriedades fundamentais de NP conforme (Sipser, 2012):

i Teorema: $P \subseteq NP$

Teorema: $P \subseteq NP$

Prova: Seja $L \in P$. Então existe uma MT determinística M que decide L em tempo polinomial $p(n)$.

Construimos um verificador V para L da seguinte forma:

- V recebe entrada $\langle w, c \rangle$ (onde c é um certificado qualquer)
- V **ignora** completamente c
- V simula M em w e aceita se M aceita

Como M roda em tempo $p(n)$, o verificador V também roda em tempo polinomial. Portanto, $L \in NP$.

□

💡 Fechamento de NP

A classe NP é fechada sob as seguintes operações:

1. **União:** Se $L_1, L_2 \in NP$, então $L_1 \cup L_2 \in NP$
2. **Concatenação:** Se $L_1, L_2 \in NP$, então $L_1 \circ L_2 \in NP$
3. **Estrela de Kleene:** Se $L \in NP$, então $L^* \in NP$

Ideia: Os certificados podem ser combinados de forma apropriada. Por exemplo, para união, o certificado indica qual linguagem aceita a entrada e fornece o certificado correspondente.

! Co-NP

Define-se também a classe **co-NP**:

$$\text{co-NP} = \{L \mid \bar{L} \in \text{NP}\}$$

Ou seja, co-NP contém os complementos das linguagens em NP.

Exemplo: $\overline{\text{SAT}}$ = “esta fórmula é insatisfazível?” está em co-NP.

Questão aberta: $NP = co-NP$? (Acredita-se que não)

Observação: Se $P = NP$, então $NP = co-NP$ (pois P é fechado sob complemento).

Exercícios de Verificação

Atividade Prática

1. **Classificação:** O problema do **Caixeiro Viajante (TSP)** pergunta: “Dado um conjunto de cidades e distâncias entre elas, existe um tour que visita todas as cidades e retorna à origem com um custo total menor que k ?”. Explique por que este problema está na classe **NP**. Qual seria o certificado?
2. **P ou NP?:** Considere o problema de **Multiplicação de Matrizes**: “Dadas duas matrizes A e B , o produto delas é C ?”. Este problema está em P ou apenas em NP ? Justifique.
3. **Implicações:** Suponha que um pesquisador anuncie amanhã que provou que $P = NP$. Qual seria a implicação mais significativa para a segurança na internet, que depende de algoritmos como o RSA (cuja segurança se baseia na dificuldade de fatorar grandes números)?
4. **Verificadores:** Considere o problema **COMPOSTOS** = $\{\langle n \rangle \mid n \text{ é um número composto (não-primo)}\}$. Mostre que este problema está em NP construindo um verificador apropriado. Qual seria o certificado?
5. **Co-NP:** O problema $\overline{HAMPATH}$ = “o grafo G NÃO possui caminho hamiltoniano de s para t ” está em $co-NP$. Por que é difícil mostrar que está em NP ? O que seria um certificado para provar que não existe caminho hamiltoniano?
6. **Fechamento:** Use a propriedade de fechamento sob união para mostrar que o problema “existe um caminho hamiltoniano OU uma 3-clique em G ?” está em NP .

Perspectivas e Importância Histórica

A questão P vs. NP é considerada por muitos como o problema mais importante da ciência da computação, e um dos problemas mais importantes de toda a matemática.

O Prêmio do Milênio

Em 2000, o Clay Mathematics Institute designou P vs. NP como um dos sete [Problemas do Prêmio do Milênio](#), oferecendo US\$ 1.000.000 pela primeira solução correta.

Até hoje, o problema permanece sem solução. Apenas um dos sete problemas (a Conjectura de Poincaré) foi resolvido desde então.

Por que P vs. NP é tão importante?

1. **Implicações práticas:** Milhares de problemas importantes em otimização, planejamento, design de circuitos, bioinformática, etc., são NP -completos (próxima aula). Se $P = NP$, todos eles teriam soluções eficientes.

2. **Fundamentos da criptografia:** A segurança de praticamente toda a criptografia moderna depende da suposição de que $P \neq NP$.
3. **Limites da computação:** A resposta determinará limites fundamentais sobre o que computadores podem fazer eficientemente.
4. **Filosofia da matemática:** Relaciona-se com questões profundas sobre provas matemáticas. Se $P = NP$, existiriam “atalhos” para encontrar provas de teoremas (Sipser, 2012).

Evidências e Crenças

Por que a maioria dos cientistas acredita que $P \neq NP$?

1. **Décadas de tentativas:** Milhares de pesquisadores tentaram encontrar algoritmos polinomiais para problemas NP-completos por mais de 50 anos, sem sucesso.
2. **Separações em outros modelos:** Existem separações análogas em outros modelos de computação.
3. **Intuição:** Parece “improvável” que verificar seja tão fácil quanto encontrar.

Mas não temos prova! Todas as tentativas de provar $P \neq NP$ esbarraram em barreiras técnicas fundamentais, como:

- Barreiras de relativização
- Barreiras de algebrização
- Barreiras de circuitos booleanos

Curiosidade Histórica

O problema P vs. NP foi formalmente formulado por Stephen Cook em 1971 e, independentemente, por Leonid Levin na URSS. Suas contribuições fundamentais levaram à descoberta dos problemas NP-completos (que estudaremos na próxima aula).

A carta de Kurt Gödel para John von Neumann em 1956 já discutia essencialmente a mesma questão, mostrando que a ideia é ainda mais antiga! (Sipser, 2012)

Próximos Passos

Na próxima aula, estudaremos os **problemas NP-completos**: problemas em NP que são, em certo sentido, os “mais difíceis” de NP . Veremos que:

- Se algum problema NP-completo estiver em P , então $P = NP$
- Se algum problema NP-completo não estiver em P , então $P \neq NP$

Isso torna os problemas NP-completos centrais para resolver a questão P vs. NP !

Referências Bibliográficas

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