

# Introdução a Problemas P (Complexidade de Problemas e Tempo Polinomial)

Pedro Ximenes

Universidade Católica de Pernambuco (UNICAP)

Análise de Algoritmos – 2026.1

# Roteiro da Aula

- 1 Revisão da Aula Anterior
- 2 Introdução – Complexidade de Problemas
- 3 Tipos de Problemas Computacionais
- 4 Algoritmos Polinomiais
- 5 A Classe P
- 6 Exemplos de Problemas em P
- 7 O Que NÃO Está (Aparentemente) em P
- 8 Propriedades da Classe P
- 9 Olhando Adiante – A Classe NP
- 10 Exercícios Resolvidos
- 11 Exercícios Propostos
- 12 Gabarito dos Exercícios Propostos
- 13 Resumo e Conclusão

# Revisão – Aula 6: Complexidade em Estruturas de Dados

## O que vimos

- 1 A **escolha da ED** afeta diretamente a complexidade do algoritmo.
- 2 **ArrayList**: acesso  $O(1)$ , inserção/remoção arbitrária  $O(n)$ .
- 3 **LinkedList**:  $O(1)$  nas pontas,  $O(n)$  para acesso por índice.
- 4 **Pilha (LIFO) e Fila (FIFO)**: todas as operações em  $O(1)$ .
- 5 **BST**:  $O(\log n)$  se balanceada,  $O(n)$  se degenerada.
- 6 **Heap**:  $O(\log n)$  inserção/remoção,  $O(1)$  para consultar o máximo.
- 7 **Tabela Hash**:  $O(1)$  médio,  $O(n)$  no pior caso.

## Ponte para a Aula 7

Até aqui medimos a complexidade de um **algoritmo**. Nesta aula daremos um passo além: mediremos a complexidade de **problemas**.

# Revisão – Complexidades que Já Conhecemos

| Complexidade  | Exemplo                     | Comportamento |
|---------------|-----------------------------|---------------|
| $O(1)$        | Acesso em array             | Constante     |
| $O(\log n)$   | Busca binária, Heap         | Logarítmico   |
| $O(n)$        | Busca linear                | Linear        |
| $O(n \log n)$ | Merge Sort                  | Quase linear  |
| $O(n^2)$      | Bubble/Selection Sort       | Quadrático    |
| $O(n^3)$      | Mult. de matrizes (naive)   | Cúbico        |
| $O(2^n)$      | Subconjuntos de um conjunto | Exponencial   |
| $O(n!)$       | Permutações (TSP ingênuo)   | Fatorial      |

## Observação Importante

Todas as complexidades da forma  $O(n^k)$  (onde  $k$  é uma constante) compartilham uma propriedade especial: são **polinomiais**. Esta aula é sobre **essa fronteira**.

# Até Agora: Complexidade de Algoritmos

## O que estamos medindo desde a Aula 1?

Medimos quanto tempo um **algoritmo específico** leva, em função do tamanho da entrada  $n$ .

- Bubble Sort:  $O(n^2)$ .
- Merge Sort:  $O(n \log n)$ .
- Busca binária:  $O(\log n)$ .

## Exemplo

Para o **problema de ordenação**:

- O algoritmo *Bubble Sort* leva  $O(n^2)$ .
- O algoritmo *Merge Sort* leva  $O(n \log n)$ .

O problema é o mesmo (ordenar). Os algoritmos é que são diferentes.

# Nova Pergunta: Complexidade do Problema

## Pergunta Fundamental

Será que, para qualquer problema, sempre existe um algoritmo “razoavelmente rápido”? E o que significa “razoavelmente rápido”?

## Complexidade de um Problema

É o **menor** tempo possível gasto pelo **melhor** algoritmo que resolve o problema.

- Não é o algoritmo que conhecemos *hoje*.
- É o que poderia existir em *qualquer lugar e em qualquer época*.
- É uma propriedade **intrínseca** do problema.

## Intuição

Para a ordenação (baseada em comparações), a complexidade é  $\Theta(n \log n)$ . Nenhum algoritmo baseado em comparações pode fazer melhor. Essa é a **complexidade do problema**.

# Por que isso importa?

## Motivação Prática

Imagine que seu chefe te pede um algoritmo “rápido” para um problema X.

- Você tenta. Tenta de novo. Tenta por semanas.
- Todos os algoritmos que você inventa são lentos (exponenciais).

## Duas Possibilidades

- 1 Você é incompetente: o algoritmo rápido existe, mas você não o encontra. → Continuar tentando.
- 2 O algoritmo rápido **não existe**: o problema é intrinsecamente difícil. → Buscar outras estratégias (aproximação, heurísticas).

## A Teoria Ajuda

A teoria da complexidade nos ajuda a **provar** em qual das duas situações estamos. Nesta aula começamos dando nome às coisas: o que é um problema “fácil”?

# Três Tipos de Problemas Computacionais

Para falar de classes de complexidade precisamos organizar os problemas

Vamos dividir os problemas que encontramos em três categorias:

- 1 **Problemas de Otimização** – pedem o mínimo ou o máximo de alguma coisa.
- 2 **Problemas de Busca** – pedem um objeto que satisfaça uma propriedade.
- 3 **Problemas de Decisão** – a resposta é apenas “sim” ou “não”.

Para a teoria da complexidade

Restringiremos nossa atenção aos **problemas de decisão**.  
Veremos em breve que isso não é uma perda real.

# Problemas de Otimização

## Definição

Um problema de **otimização** pede um objeto que **minimize** ou **maximize** alguma quantidade.

## Exemplos Clássicos

- **Caminho Mínimo**: dado um grafo com pesos e dois vértices  $r$  e  $s$ , encontrar um caminho de  $r$  a  $s$  de **menor peso**.
- **Árvore Geradora Mínima (MST)**: dado um grafo com pesos, encontrar uma árvore geradora de **peso mínimo**.
- **Mochila Ótima**: dados itens com pesos  $p_i$  e valores  $v_i$  e capacidade  $c$ , encontrar o subconjunto de valor **máximo** que cabe na mochila.
- **Caixeiro Viajante (TSP)**: encontrar o ciclo de **menor custo** que visita todas as cidades.

## Característica

A saída é um **objeto** (um caminho, uma árvore, um conjunto) que otimiza um critério.

# Problemas de Busca

## Definição

Um problema de **busca** pede um objeto que satisfaça certas propriedades. Não envolve minimização/maximização, apenas um **limiar**  $k$ .

## Exemplos

- **Caminho Curto:** dado um grafo e um número  $k$ , encontrar um caminho de  $r$  a  $s$  de peso  $\leq k$  (ou constatar que não existe).
- **Subset-Sum:** dados naturais  $p_1, \dots, p_n$  e  $c$ , encontrar um subconjunto  $K$  tal que  $\sum_{i \in K} p_i = c$ .
- **Circuito Hamiltoniano:** dado um grafo, encontrar um circuito que passa por todos os vértices *uma única vez*.
- **Fatoração:** dado um natural  $n$ , encontrar  $p$  com  $1 < p < n$  que divida  $n$ .

## Relação com Otimização

Cada busca é **mais fácil** (ou igual) que a otimização correspondente: se sabemos resolver a otimização, usamos a solução e comparamos com  $k$ .

# Problemas de Decisão

## Definição

Um problema de **decisão** é um problema cuja resposta é apenas “sim” ou “não”.

## Exemplos

- **Quadrado-Perfeito:** dado  $n \in \mathbb{N}$ , decidir se  $\sqrt{n}$  é natural.
- **Eq-Grau-2:** dados inteiros  $a, b, c$ , decidir se existe inteiro  $x$  com  $ax^2 + bx + c = 0$ .
- **Caminho-Curto:** dados um grafo, dois vértices e  $k$ , decidir se existe caminho de peso  $\leq k$ .
- **Subset-Sum:** dados  $p_1, \dots, p_n$  e  $c$ , decidir se existe subconjunto com soma igual a  $c$ .
- **Circuito-Hamiltoniano:** dado um grafo, decidir se existe circuito que passa por todos os vértices.

## Por que usar só decisão?

Decisão é a forma **mais simples**. Se um problema de decisão é “difícil”, os correspondentes problemas de busca e otimização também são.

# Conversão: Otimização $\rightarrow$ Decisão

## Ideia

Todo problema de otimização pode ser reformulado como decisão adicionando um limiar  $k$ .

## Exemplos de Conversão

- **Otim. SHORTEST-PATH:** qual o caminho mínimo entre  $u$  e  $v$ ?
- **Decisão PATH:** existe caminho entre  $u$  e  $v$  com  $\leq k$  arestas?
- **Otim.:** qual é o maior conjunto independente em  $G$ ?
- **Decisão INDEPENDENT-SET:**  $G$  tem conjunto independente com  $\geq k$  vértices?

## Relação Importante

Decisão  $\preceq$  Busca  $\preceq$  Otimização

Isto é: **decisão não é mais difícil** que busca, que não é mais difícil que otimização. Se a decisão é difícil, a otimização também é.

# O Problema do Caixeiro Viajante (TSP)

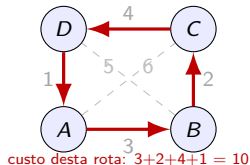
## Problema do caixeiro viajante (*Traveling Salesperson Problem*)

Dado um conjunto de **idades** e as **distâncias** (ou custos) entre **cada par** de cidades, deseja-se encontrar um **ciclo** que:

- visita **cada cidade exatamente uma vez** (ciclo hamiltoniano);
- e tem **custo total mínimo** (soma das distâncias das arestas usadas), voltando à cidade de partida.

### Em uma frase

Encontrar a **rota mais barata** que passa por todas as cidades e retorna ao início, sem repetir nenhuma parada.



# Visualizando as Três Formas – Exemplo TSP

## Caixeiro Viajante em 3 formas

**Instância:** Um grafo com pesos nas arestas representando cidades e distâncias.

- **Otimização:** Qual é o ciclo hamiltoniano de **menor peso**?
- **Busca:** Existe ciclo hamiltoniano com peso  $\leq k$ ? Encontre-o.
- **Decisão:** Existe ciclo hamiltoniano com peso  $\leq k$ ? (**sim/não**)

## Observação

Responder à versão de **decisão** parece mais simples, mas para muitos problemas famosos (como o TSP) não se conhece algoritmo eficiente nem mesmo para a decisão!

# O que é um Algoritmo Polinomial?

## Definição

Um algoritmo é **polinomial** se seu tempo de execução no pior caso é limitado por uma função polinomial do tamanho  $n$  da entrada:

$$T(n) = O(n^k) \quad \text{para alguma constante } k \geq 0.$$

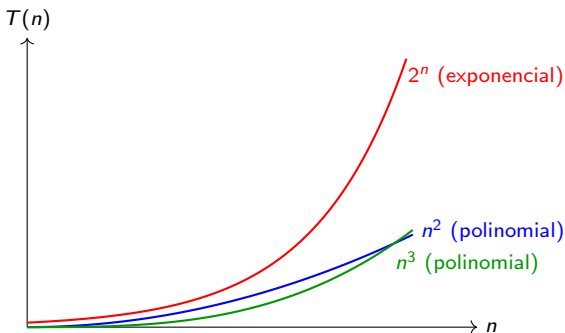
## Exemplos de Funções Polinomiais

- $T(n) = 5n \quad (k = 1) \Rightarrow$  polinomial.
- $T(n) = 100n^4 + 300n^2 + 5000 \quad (k = 4) \Rightarrow$  polinomial.
- $T(n) = 200n^9 \log n \Rightarrow$  polinomial, pois  $n^9 \log n \in O(n^{10})$ .
- $T(n) = n^2 \sqrt{n} = n^{2,5} \Rightarrow$  polinomial, pois  $n^{2,5} \in O(n^3)$ .

## Não Polinomiais

$T(n) = 2^n$ ,  $T(n) = n!$ ,  $T(n) = 1,1^n \Rightarrow$  **exponenciais**, não são polinomiais.

# Polinomial vs Exponencial – Intuição Visual



## Observação

Polinômios (mesmo de grau alto) crescem *much more slowly* que funções exponenciais. A partir de um certo  $n$ , qualquer exponencial ultrapassa qualquer polinômio.

# Crescimento Comparativo – Exemplo Numérico

## Comparativo para $n = 50$ (1 GHz)

| $T(n)$ | Valor                      | Tempo de execução                            |
|--------|----------------------------|----------------------------------------------|
| $n^2$  | 2 500                      | $2,5\mu s$                                   |
| $n^3$  | 125 000                    | $125\mu s$                                   |
| $n^5$  | $\approx 3 \times 10^8$    | $\approx 0,3$ s                              |
| $2^n$  | $\approx 10^{15}$          | $\approx 13$ dias                            |
| $n!$   | $\approx 3 \times 10^{64}$ | $\approx$ idade do universo $\times 10^{40}$ |

## Moral

Mesmo com um supercomputador rodando a 1 GHz, problemas exponenciais rapidamente se tornam inviáveis. Já polinômios com expoente pequeno rodam quase instantaneamente mesmo para  $n$  grande.

# Por que Adotamos “Polinomial = Tratável” ?

## Três Argumentos (Cormen, Capítulo 34)

- 1 **Empírico:** embora  $\Theta(n^{100})$  pareça “lento”, na prática os algoritmos polinomiais úteis têm expoente baixo ( $k = 1, 2, 3$ ). Tão logo se descobre um algoritmo polinomial, geralmente surgem versões mais eficientes.
- 2 **Independência do Modelo:** um problema que é polinomial em um modelo (RAM, Turing, paralelo com  $p$  polinomial) é polinomial em todos os outros modelos razoáveis.
- 3 **Fechamento:** a classe dos polinômios é fechada sob soma, multiplicação e composição. Compor algoritmos polinomiais resulta em algoritmo polinomial.

## Exemplo de Fechamento

Se  $A$  roda em  $O(n^2)$  e  $B$  em  $O(n^3)$ , rodar “ $A$  depois  $B$ ” continua em  $O(n^3)$ , que é polinomial.

# Pseudopolinomial

## Definição Informal

Um algoritmo é **pseudopolinomial** se seu tempo é polinomial no **valor** de algum parâmetro numérico, mas não no tamanho (em bits) da entrada.

## Exemplo – Mochila Booleana (Prog. Dinâmica)

- Complexidade:  $O(n \cdot W)$ , onde  $W$  é a capacidade da mochila.
- Parece polinomial em  $n$  e  $W$ . Mas o tamanho **em bits** de  $W$  é  $\log_2 W$ .
- Se  $W = 2^n$ , o algoritmo é  $O(n \cdot 2^n)$  em função do tamanho da entrada.

## Pseudopolinomiais não são polinomiais

Mas funcionam bem na prática quando os valores dos parâmetros são pequenos.

# Problema Polinomial

## Definição

Um **problema computacional** é **polinomial** se existe um algoritmo polinomial que o resolve.

## Observação Crucial

**Basta existir um algoritmo polinomial!** Não importa se é fácil achar. A propriedade é intrínseca ao problema.

## Exemplos

- Ordenação: polinomial ( $O(n \log n)$ ).
- Busca binária: polinomial ( $O(\log n) \subseteq O(n)$ ).
- Multiplicação de matrizes: polinomial ( $O(n^3)$ ).
- MST (Kruskal): polinomial ( $O(m \log n)$ ).
- Caminho mínimo (Dijkstra): polinomial ( $O(m + n \log n)$ ).

# A Classe P

## Definição Formal

A **classe P** é o conjunto de todos os **problemas de decisão** que podem ser resolvidos por um algoritmo em tempo polinomial.

$$P = \{ \text{decisão resolvível em } O(n^k) \text{ para alguma constante } k \}$$

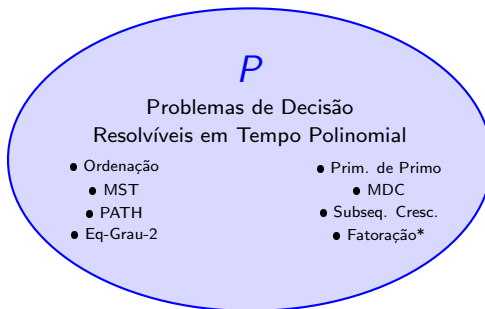
## Interpretação

Problemas em  $P$  são considerados **tratáveis** ou **fáceis** – para eles temos esperança de obter resposta em tempo razoável, mesmo para entradas grandes.

## Atenção à Notação

$P$  refere-se apenas a **problemas de decisão**. Otimização e busca têm versões equivalentes em  $P$ , mas formalmente, a classe  $P$  trata de decisão.

# Visualizando a Classe P



## \* Detalhe

A versão de **decisão** da fatoração (" $n$  é composto?") está em  $P$  (AKS 2002/2004). A **busca** dos fatores provavelmente não está em  $P$  – e é nisso que se apoia a criptografia RSA.

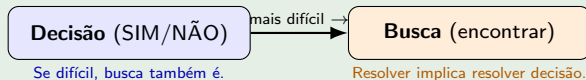
# Decisão vs Busca – Uma Distinção Fundamental

## Ideia Central

Para o mesmo problema, existem dois níveis de dificuldade:

- **Decisão:** “Existe solução?” → resposta é apenas **SIM** ou **NÃO**.
- **Busca:** “Qual é a solução?” → é preciso **encontrar** o objeto.

## Intuição



## Consequência

Se provarmos que a **decisão** é difícil, a **busca** correspondente também é. Por isso a teoria foca em decisão.

# Fatoração: Decisão vs Busca

## Versão de Decisão

**Pergunta:**  $n$  é composto?

Resposta: apenas **SIM** ou **NÃO**.

**Exemplo:**

- $n = 221$
- $221 = 13 \times 17$  (tem fator)
- Resposta: **SIM**, é composto.

**Dificuldade:** resolvível em tempo polinomial pelo algoritmo AKS (2002).

## Versão de Busca

**Pergunta:** quais são os fatores de  $n$ ?

Precisa **encontrar**  $p$  e  $q$  tais que  
 $n = p \times q$ .

**Exemplo:**

- $n = 221$
- Resposta: 13 e 17
- pois  $221 = 13 \times 17$

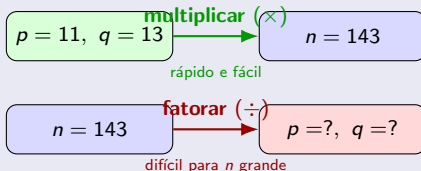
**Dificuldade:** nenhum algoritmo eficiente é conhecido para  $n$  grandes.

## Resumo

Saber que  $n$  é composto (**decisão**) é muito mais fácil do que achar os fatores (**busca**).

# Por que a Fatoração é Importante? – RSA

## A Assimetria que Sustenta a Criptografia



## Ideia do RSA (simplificada)

- 1 Escolhe-se dois primos  $p$  e  $q$  e calcula-se  $n = p \times q$  (fácil).
- 2 O valor  $n$  é tornado **público**;  $p$  e  $q$  ficam **secretos**.
- 3 A segurança depende de que fatorar  $n$  seja **inviável** para  $n$  muito grande.

## Conexão com a Classe P

A versão de **decisão** da primalidade está em  $P$  (AKS, 2002).

A versão de **busca** (encontrar os fatores) provavelmente **não está em P** – e é nisso que o RSA se apoia.

# Exemplo 1 – Eq-Grau-2 (Enunciado)

## Problema de Decisão

**Eq-Grau-2:** Dados inteiros  $a, b, c$ , decidir se existe um inteiro  $x \in \mathbb{Z}$  tal que:

$$ax^2 + bx + c = 0.$$

## Entrada e Saída

- **Entrada:** três inteiros  $a, b, c$ .
- **Saída:** **SIM** se  $\exists x \in \mathbb{Z}$  com  $ax^2 + bx + c = 0$ ; **NÃO** caso contrário.

## Pergunta Central

Existe um algoritmo *eficiente (polinomial)* que decide Eq-Grau-2?

# Exemplo 1 – Eq-Grau-2 (Algoritmo)

## Algoritmo Polinomial – Discriminante

- 1 Calcular o discriminante  $\Delta = b^2 - 4ac$ .
- 2 Se  $\Delta < 0$ : não existe raiz real  $\Rightarrow$  responder **NÃO**.
- 3 Calcular  $r = \lfloor \sqrt{\Delta} \rfloor$ ; verificar se  $r^2 = \Delta$  (raiz exata inteira).
- 4 Se  $r^2 \neq \Delta$ : raízes irracionais  $\Rightarrow$  responder **NÃO**.
- 5 Verificar se  $(-b + r) \bmod (2a) = 0$  ou  $(-b - r) \bmod (2a) = 0$ .
- 6 Responder **SIM** ou **NÃO** conforme o passo 5.

## Custo e Conclusão

Poucas operações aritméticas ( $+$ ,  $\times$ ,  $\div$ ,  $\sqrt{\phantom{x}}$ ) sobre  $a$ ,  $b$ ,  $c$  – quantidade constante de operações, independente do tamanho da entrada.

$$\text{Eq-Grau-2} \in P$$

## Exemplo 2 – PATH: Caminho entre Vértices (Enunciado)

### Problema de Decisão

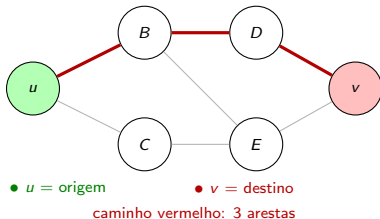
**PATH:** Dados um grafo  $G=(V, E)$ , dois vértices  $u, v \in V$  e um inteiro  $k \geq 0$ , decidir se existe caminho de  $u$  a  $v$  com **no máximo  $k$  arestas**.

### Instância

$|V|=6$ ,  $|E|=7$ ,  $k=3$ .

Origem:  $u$ , Destino:  $v$ .

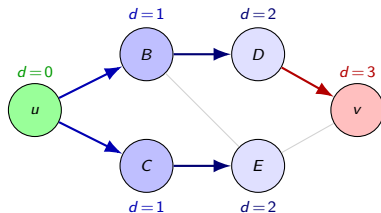
**Pergunta:** existe caminho  $u \rightarrow v$  com  $\leq 3$  arestas?



## Exemplo 2 – PATH: Caminho entre Vértices (Algoritmo)

### Algoritmo: BFS

- 1 Verificar  $G$  válido e  $u, v \in V$ .
- 2 Executar **BFS** a partir de  $u$ , calculando  $d(u, w)$  para todo  $w \in V$ .
- 3 Se  $d(u, v) \leq k$ : responder **SIM**.
- 4 Caso contrário: responder **NÃO**.



### Custo e Resultado

BFS:  $O(|V| + |E|)$  – linear.

Exemplo:  $d(u, v) = 3 \leq k = 3 \Rightarrow$  **SIM**

PATH  $\in P$

# Exemplo 3 – Verificar Palíndromo (Enunciado)

## Problema de Decisão

**PALÍNDROMO:** Dado um array  $A$  com  $n$  elementos, decidir se  $A$  é um **palíndromo**, ou seja, se  $A[i] = A[n-1-i]$  para todo  $i$ .

## Instâncias

| Array $A$       | Palíndromo? |
|-----------------|-------------|
| [1, 2, 3, 2, 1] | SIM         |
| [5, 5, 5]       | SIM         |
| [2, 3, 1, 4]    | NÃO         |
| [7, 8, 7]       | SIM         |

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 2 | 1 |
|---|---|---|---|---|

[1, 2, 3, 2, 1] **SIM**

|   |   |   |   |
|---|---|---|---|
| 2 | 3 | 1 | 4 |
|---|---|---|---|

[2, 3, 1, 4] **NÃO**

# Exemplo 3 – Verificar Palíndromo (Algoritmo)

## Algoritmo: Comparação Linear

- 1 Para  $i \leftarrow 0$  até  $\lfloor n/2 \rfloor - 1$
- 2   Se  $A[i] \neq A[n-1-i]$
- 3     responder **NÃO**
- 4 Responder **SIM**

## Custo e Resultado

$O(n)$  – verificação em tempo linear.

PALÍNDROMO  $\in P$

Exemplo:  $A = [1, 2, 3, 2, 1]$ ,  $n = 5$

| $i$                                    | $A[i]$ | $A[n-1-i]$ | OK  |
|----------------------------------------|--------|------------|-----|
| 0                                      | 1      | 1          | Sim |
| 1                                      | 2      | 2          | Sim |
| Pare ( $i < \lfloor 5/2 \rfloor = 2$ ) |        |            |     |

Todas as comparações OK  $\Rightarrow$  **SIM**

# Exemplo 4 – Árvore Geradora Mínima (Enunciado)

## O que é uma Árvore Geradora?

Uma **árvore geradora** (spanning tree) de um grafo  $G=(V, E)$  é um subgrafo que:

- Contém **todos** os vértices de  $V$
- Possui exatamente  $|V|-1$  arestas
- É conectado e acíclico (i.e., é uma árvore)

Se o grafo tem pesos nas arestas, a **árvore geradora mínima (MST)** é aquela com menor peso total.

## Problema de Decisão

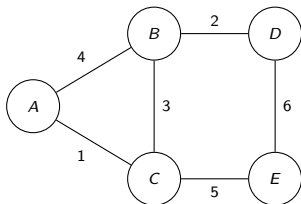
**ÁRVORE-GERADORA-LEVE:** Dado um grafo conectado  $G=(V, E)$  com pesos nas arestas e um número  $k$ , decidir se existe uma árvore geradora de **peso total**  $\leq k$ .

## Instância

Grafo com 5 vértices, 6 arestas.

Limiar:  $k=12$ .

**Pergunta:** existe árvore geradora de peso  $\leq 12$ ?



# Exemplo 4 – Árvore Geradora Mínima (Algoritmo)

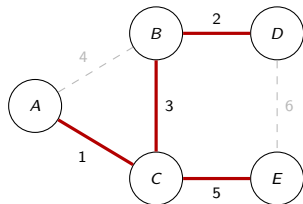
## Algoritmo: Kruskal

- 1 Ordenar arestas por peso crescente.
- 2 Adicionar cada aresta que não forma ciclo.
- 3 Calcular o peso total da MST.
- 4 **SIM** se peso  $\leq k$ , senão **NÃO**.

Kruskal no exemplo ( $k=12$ ):

- A–C: 1 (add)
- B–D: 2 (add)
- B–C: 3 (add)
- A–B: 4 (ciclo, pula)
- C–E: 5 (add)

Peso MST:  $1+2+3+5=11 \leq 12 \Rightarrow$  **SIM**



MST (vermelho): peso = 11

Tracejado: arestas rejeitadas

## Custo e Resultado

$O(|E| \log |V|)$  com heap binária.

ÁRVORE-GERADORA-LEVE  $\in P$

# Exemplo 5 – IS-SORTED: Verificação de Ordenação (Enunciado)

## Problema de Decisão

**IS-SORTED:** Dado um vetor  $A[0..n-1]$  de inteiros, decidir se está em **ordem não decrescente**:

$$A[0] \leq A[1] \leq \dots \leq A[n-1].$$

## Exemplos

| Vetor           | Resposta                             |
|-----------------|--------------------------------------|
| [1, 3, 5, 7, 9] | <b>SIM</b> (estritamente crescente)  |
| [2, 2, 4, 8]    | <b>SIM</b> (não decrescente)         |
| [5, 3, 6, 1]    | <b>NÃO</b> ( $A[1] = 3 < A[0] = 5$ ) |
| [10, 9, 8, 7]   | <b>NÃO</b> (ordem decrescente)       |

## Pergunta

Existe um algoritmo polinomial para IS-SORTED?

# Exemplo 5 – IS-SORTED: Verificação de Ordenação (Algoritmo)

## Algoritmo: Varredura Linear

```
algoritmo isSorted(A, n)
  para i de 1 ate n-1 faca
    se A[i-1] > A[i] entao
      retorne FALSE      // violacao encontrada
    fim se
  fim para
  retorne TRUE           // nenhuma violacao
fim algoritmo
```

**Custo:**  $O(n)$  – uma única varredura linear.

IS-SORTED  $\in P$

## Observação

Ordenar o vetor (Merge Sort) leva  $O(n \log n) \Rightarrow$  o problema de ordenação também está em  $P$ .

# Exemplo 6 – PRIMO: Teste de Primalidade (Enunciado)

## Problema de Decisão

**PRIMO:** Dado um número natural  $n$ , decidir se  $n$  é primo.

## Exemplos

| $n$ | Primo?     | Justificativa           |
|-----|------------|-------------------------|
| 2   | <b>SIM</b> | Menor primo             |
| 7   | <b>SIM</b> | Divisores apenas 1 e 7  |
| 9   | <b>NÃO</b> | $9 = 3 \times 3$        |
| 97  | <b>SIM</b> | Sem divisor em $[2, 9]$ |
| 100 | <b>NÃO</b> | $100 = 2^2 \times 5^2$  |

## Cuidado com a Intuição

Testar todos os divisores até  $\sqrt{n}$  parece rápido, mas é **lento** para valores grandes de  $n$  – veremos isso com mais detalhe adiante.

# Exemplo 6 – PRIMO: Teste de Primalidade (Algoritmo)

## Solução Ingênua (Lenta)

Testar todos os divisores de 2 até  $\sqrt{n}$ : custo  $O(\sqrt{n})$  – **não é polinomial eficiente** para entradas grandes.

## Algoritmo AKS – Descoberta Histórica (2002)

**Agrawal, Kayal e Saxena** (IIT Kanpur, Índia) publicaram o primeiro algoritmo **determinístico e eficiente** para primalidade:

- Funciona em tempo polinomial para qualquer entrada.
- Antes de 2002, só existiam algoritmos *probabilísticos* rápidos (ex.: Miller–Rabin).

PRIMO  $\in P$

# Exemplo 7 – DIVISOR-COMUM-GRANDE (Enunciado)

## Problema de Decisão

**DIVISOR-COMUM-GRANDE:** Dados naturais  $m$ ,  $n$  e  $k$ , decidir se existe divisor comum  $d \geq k$  de  $m$  e  $n$  (i.e.,  $d \mid m$ ,  $d \mid n$  e  $d \geq k$ ).

## Exemplos

| $m$ | $n$ | $k$ | Resposta                                         |
|-----|-----|-----|--------------------------------------------------|
| 12  | 8   | 3   | <b>SIM</b> ( $\text{mdc}(12, 8) = 4 \geq 3$ )    |
| 15  | 25  | 10  | <b>NÃO</b> ( $\text{mdc}(15, 25) = 5 < 10$ )     |
| 36  | 48  | 12  | <b>SIM</b> ( $\text{mdc}(36, 48) = 12 \geq 12$ ) |

## Ideia-Chave

Basta calcular  $d = \text{mdc}(m, n)$  e verificar se  $d \geq k$ . O maior divisor comum é o único candidato relevante.

# Exemplo 7 – DIVISOR-COMUM-GRANDE (Algoritmo)

## Algoritmo de Euclides

```
algoritmo mdc(m, n)
  enquanto n ≠ 0 faça
    r ← m mod n
    m ← n
    n ← r
  fim enquanto
  retorne m
fim algoritmo

algoritmo divisorComumGrande(m, n, k)
  d ← mdc(m, n)
  retorne (d ≥ k)
fim algoritmo
```

**Iterações de Euclides:**  $O(\log(\min(m, n)))$  – proporcional ao número de bits.

## Custo e Conclusão

Custo total:  $O(\log^2(\min(m, n)))$  no modelo de bits (incluindo custo das operações de módulo).

DIVISOR-COMUM-GRANDE  $\in P$

## Exemplo 8 – BUSCA-LINEAR (Enunciado)

### Problema de Decisão

**BUSCA-LINEAR:** Dado um array  $A[0..n-1]$  e um valor  $x$ , decidir se  $x$  aparece em  $A$ .

### Exemplos

| Array $A$       | Valor $x$ | Resposta               |
|-----------------|-----------|------------------------|
| [3, 1, 4, 1, 5] | 4         | <b>SIM</b> (posição 2) |
| [3, 1, 4, 1, 5] | 7         | <b>NÃO</b>             |
| [9, 9, 9]       | 9         | <b>SIM</b>             |
| [] (vazio)      | 1         | <b>NÃO</b>             |

### Aplicação

Fundamental em bases de dados, dicionários, compiladores e sistemas de busca.

# Exemplo 8 – BUSCA-LINEAR (Algoritmo)

## Algoritmo Simples

- Para  $i \leftarrow 0$  até  $n - 1$
- Se  $A[i] = x$ : retorne **SIM**
- Retorne **NÃO** (se não achou)

**Pior caso:** percorrer todo o array  $\Rightarrow O(n)$ .

## Custo e Conclusão

No pior caso, fazemos  $n$  comparações:  $O(n)$  – linear em  $n$ .

BUSCA-LINEAR  $\in P$

# Exemplo 9 – FIBONACCI-N (Enunciado)

## Problema de Decisão

**FIBONACCI-N:** Dado um número natural  $n$ , calcular o  $n$ -ésimo número de Fibonacci.

## Definição Clássica

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2) \text{ para } n \geq 2$$

## Exemplos

| $n$ | $F(n)$ |
|-----|--------|
| 0   | 0      |
| 1   | 1      |
| 5   | 5      |
| 10  | 55     |
| 20  | 6765   |

# Exemplo 9 – FIBONACCI-N (Algoritmo Polinomial)

## Abordagem Iterativa (Eficiente)

- Se  $n = 0$ : retorne 0
- Se  $n = 1$ : retorne 1
- Inicialize  $a \leftarrow 0$ ,  $b \leftarrow 1$
- Para  $i \leftarrow 2$  até  $n$ : atualizar  $a \leftarrow b$ ,  $b \leftarrow a + b$
- Retorne  $b$

**Custo:**  $O(n)$  – loop de  $n - 1$  iterações, cada uma em tempo constante.

## Contraste

A abordagem **recursiva ingênua**  $F(n) = F(n-1) + F(n-2)$  é exponencial  $O(2^n)$  e inviável para  $n$  grande.

FIBONACCI-N (iterativo)  $\in P$

# Exemplo 10 – MAIOR-ELEMENTO (Enunciado)

## Problema de Decisão

**MAIOR-ELEMENTO:** Dado um array  $A[0..n-1]$  de inteiros e um valor  $k$ , decidir se o **máximo** de  $A$  é  $\geq k$ .

## Exemplos

| Array $A$    | Limiar $k$ | Resposta                          |
|--------------|------------|-----------------------------------|
| [2, 8, 3, 5] | 7          | <b>SIM</b> ( $\max = 8 \geq 7$ )  |
| [2, 8, 3, 5] | 9          | <b>NÃO</b> ( $\max = 8 < 9$ )     |
| [1, 1, 1]    | 1          | <b>SIM</b> ( $\max = 1 \geq 1$ )  |
| [10, 5, 3]   | 0          | <b>SIM</b> ( $\max = 10 \geq 0$ ) |

# Exemplo 10 – MAIOR-ELEMENTO (Algoritmo)

## Algoritmo de Varredura

- Inicialize máximo  $\leftarrow A[0]$
- Para  $i \leftarrow 1$  até  $n - 1$
- Se  $A[i] > \text{máximo}$ : atualizar máximo  $\leftarrow A[i]$
- Retorne (máximo  $\geq k$ )

**Pior caso:** percorrer todo o array  $\Rightarrow O(n)$ .

## Custo e Conclusão

Examinamos cada um dos  $n$  elementos uma única vez. Custo:  $O(n)$  – linear.

MAIOR-ELEMENTO  $\in P$

# Problemas Cuja Pertença a P é Desconhecida

## O Drama da Computação

Existem **centenas** de problemas para os quais:

- **Ninguém** conseguiu encontrar um algoritmo polinomial.
- **Ninguém** conseguiu provar que não existe.

## Exemplos Famosos (Provavelmente fora de P)

- **Circuito Hamiltoniano:** existe ciclo que passa por todos os vértices?
- **Caixeiro Viajante (TSP):** existe rota de peso  $\leq k$  por todas as cidades?
- **Subset-Sum:** existe subconjunto com soma exatamente  $c$ ?
- **Mochila Booleana:** existe subconjunto cuja soma de pesos  $\leq c$  e valor  $\geq d$ ?
- **Coloração de Grafos ( $k \geq 3$ ):**  $G$  pode ser colorido com  $k$  cores?
- **SAT:** uma fórmula booleana é satisfazível?

## Suspeita

Todos esses problemas estão em uma classe maior chamada **NP** (próxima aula).  
Acredita-se fortemente que  $P \neq NP$ .

# Problemas Indecidíveis – Vão Muito Além

## Um Nível Acima da “Dificuldade”

Existem problemas para os quais **não existe** algoritmo nenhum (nem exponencial!) que os resolva em tempo finito.

## Exemplo – Problema da Parada (Halting Problem)

“Dado um programa  $P$  e uma entrada  $x$ , decidir se  $P$  termina em  $x$ .”

Alan Turing provou (1936): **não existe** algoritmo que resolva esse problema para todos os programas.

## Outro Exemplo – Equações Diofantinas

“Dada uma equação polinomial com coeficientes inteiros, existem soluções inteiras?”

Hilbert (10º problema, 1900) → Matiyasevich (1970): **indecidível**.

## Hierarquia Intuitiva

$$P \subsetneq NP \subsetneq \text{Exponencial} \subsetneq \dots \subsetneq \text{Indecidíveis}$$

# O Problema da Parada (Halting Problem) – Detalhamento

## Definição Formal

**HALTING:** Dado um programa  $P$  e uma entrada  $x$ , decidir se a execução de  $P(x)$  termina (ou entra em loop infinito).

## Por que é indecidível?

**Prova de Turing (1936):** Suponha que existe uma máquina de Turing  $H$  que resolve HALTING. Então:

- $H(P, x)$  retorna **SIM** se  $P(x)$  termina, **NÃO** se não termina.
- Construa um programa  $D$  que use  $H$  assim: se  $H$  diz que  $D(D)$  termina,  $D$  entra em loop; se  $H$  diz que não termina,  $D$  para.
- Pergunta:  $D(D)$  termina? Se sim,  $D$  entra em loop (contradição). Se não,  $D$  termina (contradição).
- Logo,  $H$  não pode existir!

## Conclusão

Nenhum programa pode resolver HALTING para **todos** os pares  $(P, x)$ . É um limite fundamental da computação.

# Propriedades de Fechamento de P

## Fechamento

A classe  $P$  é **fechada** sob várias operações:

- **Complemento:** se  $L \in P$ , então  $\bar{L} \in P$ .
- **União:** se  $L_1, L_2 \in P$ , então  $L_1 \cup L_2 \in P$ .
- **Interseção:** se  $L_1, L_2 \in P$ , então  $L_1 \cap L_2 \in P$ .
- **Concatenação:** se  $L_1, L_2 \in P$ , então  $L_1 \cdot L_2 \in P$ .
- **Composição:** compor algoritmos polinomiais dá algoritmo polinomial.

## Por que isso é importante?

Significa que podemos combinar problemas fáceis e o resultado continua fácil. A tratabilidade é **modular**.

## Atenção

Um número *constante* de chamadas a sub-rotinas polinomiais + trabalho adicional polinomial = algoritmo polinomial. Mas um número *polinomial* de chamadas pode resultar em algoritmo **exponencial** (Cormen 34.1-5).

# Exemplo de Fechamento – Composição

## Exemplo Concreto

Temos dois algoritmos polinomiais:

- $A$ : detecta se um grafo é conexo ( $O(|V| + |E|)$ ).
- $B$ : calcula a MST ( $O(|E| \log |V|)$ ).

**Algoritmo composto**  $C$ : “dado um grafo com pesos e um número  $k$ , decidir se o grafo é conexo E tem MST de peso  $\leq k$ .”

- 1 Executa  $A$ :  $O(|V| + |E|)$ .
- 2 Se não é conexo: responde **NÃO**.
- 3 Executa  $B$ :  $O(|E| \log |V|)$ .
- 4 Compara com  $k$ .

**Total:**  $O(|E| \log |V|)$  – continua polinomial!  $\Rightarrow C \in P$ .

# Além de P – A Classe NP (Prévia)

## Motivação

Alguns problemas não sabemos resolver em tempo polinomial, mas sabemos **verificar rapidamente** se uma solução candidata está correta.

## Exemplo: Circuito Hamiltoniano

**Resolver:** Encontrar um ciclo que passe por todos os vértices? → Difícil (não se conhece polinomial).

**Verificar:** Um amigo lhe dá uma lista de vértices. É um ciclo hamiltoniano?

- Verificar que a lista contém cada vértice exatamente uma vez.  $O(n)$
- Verificar que arestas consecutivas existem no grafo.  $O(n)$

Total:  $O(n^2)$  – **polinomial**.

## Preview

**NP** = problemas de decisão cuja resposta “SIM” pode ser **verificada** em tempo polinomial, dada uma “dica” (certificado).

# P vs NP – A Grande Pergunta

## A Pergunta do Milênio

$$P \stackrel{?}{=} NP$$

## Intuição

- **P**: problemas que podemos **resolver** rapidamente.
- **NP**: problemas cuja solução podemos **verificar** rapidamente.

Verificar é intuitivamente mais fácil que resolver. Logo, provavelmente,  $P \neq NP$ .

## Clay Mathematics Institute

Um dos 7 Problemas do Milênio. Quem resolver ganha **US\$ 1 milhão**. Desde 1971 (Cook/Levin), ninguém provou nem  $P = NP$ , nem  $P \neq NP$ .

## Certeza

$P \subseteq NP$  (todo problema que resolvemos, sabemos verificar). Continuaremos esse assunto na próxima aula!

# Exercício Resolvido 1 – Classificar Complexidades

## Enunciado

Classifique cada complexidade abaixo como **polinomial** ou **não polinomial**. Justifique.

①  $T(n) = 7n^3 + 5n$

②  $T(n) = 100 \cdot 2^n$

③  $T(n) = n \log n$

④  $T(n) = n^{0,5} \cdot n^{1,5}$

⑤  $T(n) = n^{\log n}$

⑥  $T(n) = 2^{\log n}$

# Exercício 1 – Solução (Passo a Passo)

- ❶  $T(n) = 7n^3 + 5n$ : Polinômio de grau 3  $\Rightarrow O(n^3)$ . **Polinomial.**
- ❷  $T(n) = 100 \cdot 2^n$ : Exponencial, já que  $2^n$  cresce mais rápido que qualquer  $n^k$ . **Não polinomial.**
- ❸  $T(n) = n \log n$ :  $\log n < n$  para  $n$  grande, então  $n \log n < n \cdot n = n^2 \in O(n^2)$ . **Polinomial.**
- ❹  $T(n) = n^{0,5} \cdot n^{1,5} = n^2$ : Claramente polinomial. **Polinomial.**
- ❺  $T(n) = n^{\log n}$ : O expoente *cresce* com  $n$  (não é constante). Não se encaixa em  $O(n^k)$ . **Não polinomial.**  
(De fato,  $n^{\log n} = 2^{(\log n)^2}$ , “quase-polinomial”.)
- ❻  $T(n) = 2^{\log n} = n$ : Identidade matemática ( $2^{\log_2 n} = n$ ). **Polinomial.**

## Resumo

Polinomial  $\iff T(n) \leq c \cdot n^k$  com  $k$  **constante**. Se o expoente varia com  $n$ , não é polinomial.

# Exercício Resolvido 2 – Eq-Grau-2 $\in P$

## Enunciado

Mostre, dando o algoritmo e analisando a complexidade, que o problema **Eq-Grau-2** pertence a  $P$ :

“Dados  $a, b, c \in \mathbb{Z}$ , existe  $x \in \mathbb{Z}$  com  $ax^2 + bx + c = 0$ ?”

# Exercício 2 – Solução

## Algoritmo:

```

algoritmo eqGrau2(a, b, c)
  se a = 0 entao
    // Equacao linear bx + c = 0
    se b = 0 entao
      retorne (c = 0)      // 0 = 0 vs k = 0
    fim se
    retorne (c mod b = 0)  // x = -c/b inteiro?
  fim se
  delta ← b*b - 4*a*c
  se delta < 0 entao
    retorne FALSE          // sem raizes reais
  fim se
  r ← raizQuadradaInteira(delta)
  se r*r ≠ delta entao
    retorne FALSE          // raiz nao inteira
  fim se
  // x = (-b +- r) / (2a) deve ser inteiro
  retorne ((-b + r) mod (2a) = 0) ou ((-b - r) mod (2a) = 0)
fim algoritmo

```

**Análise:** Todas as operações são aritméticas com números de  $O(\log |a| + \log |b| + \log |c|)$  bits. Cada operação leva tempo polinomial no tamanho da entrada.

$$\text{Eq-Grau-2} \in P$$

# Exercício Resolvido 3 – Identificar Problemas em P

## Enunciado

Para cada problema abaixo, indique se está em  $P$  (sim/não/desconhecido). Justifique.

- ① Ordenar um vetor de  $n$  inteiros.
- ② Decidir se um grafo é conexo.
- ③ Decidir se um grafo tem um ciclo hamiltoniano.
- ④ Encontrar o menor valor em um vetor.
- ⑤ Calcular o  $n$ -ésimo número de Fibonacci (iterativamente).
- ⑥ Decidir se uma fórmula booleana com  $n$  variáveis é satisfazível (SAT).

# Exercício 3 – Solução

❶ **Ordenação: Está em P.**

Merge Sort resolve em  $O(n \log n) \subseteq O(n^2)$ .

❷ **Conexidade do grafo: Está em P.**

DFS/BFS em  $O(|V| + |E|)$  determina a conexidade.

❸ **Ciclo Hamiltoniano: Desconhecido / Provavelmente NÃO.**

É NP-completo. Nenhum algoritmo polinomial é conhecido.

❹ **Mínimo de vetor: Está em P.**

Varredura linear em  $O(n)$ .

❺ **Fibonacci iterativo: Está em P.**

$O(n)$  operações aritméticas (embora os números cresçam, o custo é polinomial no tamanho da saída).

❻ **SAT: Desconhecido / Provavelmente NÃO.**

É NP-completo (Teorema de Cook-Levin, 1971). Testa ingenuamente  $2^n$  atribuições.

# Exercício Resolvido 4 – Tamanho da Entrada

## Enunciado

Considere o seguinte algoritmo para decidir se  $n$  é primo:

```
algoritmo primoIngenuo(n)
  para i de 2 ate n-1 faca
    se n mod i = 0 entao
      retorne FALSE
    fim se
  fim para
  retorne TRUE
fim algoritmo
```

Esse algoritmo é polinomial? Explique cuidadosamente.

## Exercício 4 – Solução

**Passo 1 – Contar operações.** O laço executa  $n - 2$  iterações, cada uma com uma operação de módulo. Custo:  $\Theta(n)$ .

**Passo 2 – Pensar no tamanho da entrada.**

$n$  é representado em binário com  $m = \lceil \log_2 n \rceil$  bits. Logo  $n = \Theta(2^m)$ .

**Passo 3 – Reescrever o custo em função de  $m$ .**

$$T(m) = \Theta(n) = \Theta(2^m).$$

### Conclusão

O algoritmo é **EXPONENCIAL** em  $m$  (tamanho da entrada em bits), embora pareça linear em  $n$ . **Não** é polinomial.

### Observação Importante

Isto **não** significa que o problema *PRIMO* não pertença a P! Só significa que este *algoritmo específico* não é polinomial. De fato, PRIMO está em P (AKS, 2002).

## Exercício Resolvido 5 – Composição de Algoritmos

### Enunciado

Seja  $A$  um algoritmo em  $O(n^2)$  e  $B$  em  $O(n^3)$ .

Considere o algoritmo  $C$  que primeiro executa  $A$  e depois, sobre a saída de  $A$ , executa  $B$ .

Suponha que a saída de  $A$  tem tamanho no máximo  $|saida\_A| \leq c \cdot n^2$  para alguma constante  $c$ . Qual a complexidade de  $C$ ? Ele é polinomial?

## Exercício 5 – Solução

**Passo 1 – Custo de  $A$ .**  $A$  roda em  $O(n^2)$ .

**Passo 2 – Tamanho da saída de  $A$ .** Por hipótese:  $|\text{saida}_A| \leq c \cdot n^2$ .

**Passo 3 – Custo de  $B$ .**  $B$  roda em  $O(m^3)$ , onde  $m$  = tamanho da entrada de  $B$  = tamanho da saída de  $A$ .

$$T_B = O((c \cdot n^2)^3) = O(n^6).$$

**Passo 4 – Custo total de  $C$ .**

$$T_C = T_A + T_B = O(n^2) + O(n^6) = O(n^6).$$

### Resposta

$C$  é  $O(n^6)$  – polinomial.

Isso ilustra o **fechamento de  $P$  sob composição**.

# Exercício Resolvido 6 – Reformulação Otim. → Decisão

## Enunciado

Escreva a versão de decisão do seguinte problema de otimização:

**COBERTURA-MÍNIMA:** Dado um grafo  $G$ , encontre o menor conjunto  $C \subseteq V$  tal que toda aresta de  $G$  tem pelo menos uma ponta em  $C$ .

# Exercício 6 – Solução

## Versão de Decisão:

### COBERTURA-PEQUENA

Dado um grafo  $G$  e um número natural  $k$ , decidir se existe um conjunto  $C \subseteq V$ , com  $|C| \leq k$ , tal que toda aresta de  $G$  tem pelo menos uma ponta em  $C$ .

## Como a decisão resolve a otimização?

- 1 Com um algoritmo polinomial para COBERTURA-PEQUENA, usamos **busca binária** em  $k \in [1, |V|]$ .
- 2 Achar o menor  $k^*$  para o qual a resposta é SIM.
- 3 Isso dá  $O(\log |V|)$  chamadas à decisão, cada uma polinomial.

### Conclusão

COBERTURA-PEQUENA  $\in P \Rightarrow$  COBERTURA-MÍNIMA resolvível em tempo polinomial.

**Obs:** ninguém sabe se COBERTURA-PEQUENA  $\in P$  – ela é NP-completa.

# Exercício Resolvido 7 – Certificar Pertinência em P

## Enunciado

Considere o seguinte problema: **MAIOR-PAR**: Dado um vetor  $A$  de  $n$  inteiros positivos e um valor  $s$ , decidir se existem  $i \neq j$  com  $A[i] + A[j] = s$ .

Mostre que MAIOR-PAR  $\in P$ , dando um algoritmo polinomial.

# Exercício 7 – Solução

## Abordagem 1 – Dois Loops (ingênua).

```
algoritmo maiorPar_v1(A, n, s)
  para i de 0 ate n-2 faca
    para j de i+1 ate n-1 faca
      se A[i] + A[j] = s entao retorne TRUE
    fim para
  fim para
  retorne FALSE
fim algoritmo
```

**Custo:**  $O(n^2)$  – polinomial.

## Abordagem 2 – Tabela Hash (mais eficiente).

```
algoritmo maiorPar_v2(A, n, s)
  H ← conjunto vazio
  para i de 0 ate n-1 faca
    se (s - A[i]) pertence a H entao retorne TRUE
    H.adicionar(A[i])
  fim para
  retorne FALSE
fim algoritmo
```

**Custo:**  $O(n)$  médio – polinomial.

MAIOR-PAR  $\in P$

# Exercício Proposto 1

## Classificação de Complexidades

Para cada função abaixo, classifique como **polinomial** ou **não polinomial**. Justifique cada resposta.

①  $T(n) = 2n^{10} + 50n^7 + 100n$

②  $T(n) = n \cdot 2^n$

③  $T(n) = 2^{100} \cdot n$

④  $T(n) = n^{\sqrt{n}}$

⑤  $T(n) = (\log n)^{100}$

⑥  $T(n) = \sqrt{n^3}$

*Dica:* Pergunte-se se o expoente de  $n$  é uma *constante* (polinomial) ou varia com  $n$  (não polinomial).

## Exercício Proposto 2

### Problemas em P

Para cada problema abaixo, **(a)** reformule-o como problema de decisão e **(b)** diga se ele está em P, dando o algoritmo polinomial (se houver).

- ❶ Dado um vetor, encontrar o maior elemento.
- ❷ Dado um grafo, verificar se é bipartido.
- ❸ Dada uma matriz  $n \times n$ , calcular o determinante.
- ❹ Dado um grafo com pesos, encontrar um caminho hamiltoniano de peso mínimo.
- ❺ Dado um grafo e um número  $k$ , decidir se o grafo pode ser colorido com  $k = 2$  cores.

## Exercício Proposto 3

### Tamanho da Entrada

Considere o seguinte algoritmo para calcular  $2^n$ :

```
resultado ← 1
para  $i$  de 1 até  $n$  faça
    resultado ←  $2 \cdot \text{resultado}$ 
retorne resultado
```

- 1 Esse algoritmo é polinomial em  $n$ ?
- 2 Esse algoritmo é polinomial no **tamanho da entrada** (em bits)?
- 3 Explique a diferença.

*Dica:* Lembre-se que a entrada  $n$  é codificada em  $\lceil \log_2 n \rceil$  bits. E pense também no **tamanho da saída**.

# Exercício Proposto 4

## Composição de Algoritmos

Três algoritmos polinomiais compõem um pipeline:

- $A$ : entrada de tamanho  $n$ , saída de tamanho  $O(n)$ , custo  $O(n \log n)$ .
- $B$ : entrada de tamanho  $m$ , saída de tamanho  $O(m^2)$ , custo  $O(m^2)$ .
- $C$ : entrada de tamanho  $k$ , saída de tamanho  $O(k)$ , custo  $O(k^3)$ .

Execute  $A$ , depois  $B$  na saída de  $A$ , depois  $C$  na saída de  $B$ .

- 1 Qual a complexidade total em função de  $n$ ?
- 2 O pipeline ainda é polinomial?

## Exercício Proposto 5

### Reformulação Otimização $\rightarrow$ Decisão

Para cada problema de otimização, escreva a versão de decisão correspondente:

- 1 Encontrar o caminho mais longo entre dois vértices de um grafo.
- 2 Encontrar o maior conjunto independente em um grafo.
- 3 Encontrar a maior subsequência comum de duas strings.
- 4 Encontrar o menor número de moedas para totalizar um valor  $v$ .

*Dica:* Introduza um parâmetro  $k$  que serve de limiar e troque “encontrar o máximo/mínimo” por “existe solução com valor  $\geq k$  ou  $\leq k$ ”.

## Exercício Proposto 6

### Certificar $\in P$ – Problema Novo

Considere o problema:

**TRÊS-SOMAM-ZERO:** Dado um vetor  $A$  com  $n$  inteiros, decidir se existem três índices distintos  $i, j, k$  tais que  $A[i] + A[j] + A[k] = 0$ .

- 1 Dê um algoritmo para este problema.
- 2 Analise a complexidade.
- 3 Conclua se **TRÊS-SOMAM-ZERO**  $\in P$ .

*Dica:* Três loops aninhados funcionam. Pense também em uma abordagem com ordenação + dois ponteiros.

# Exercício Proposto 7

## Discussão Conceitual

Discuta (em no máximo 5 linhas) cada afirmação abaixo, dizendo se é **verdadeira** ou **falsa** e por quê:

- ① “Se um problema está em P, então ele pode ser resolvido rapidamente para qualquer entrada.”
- ② “Todo problema de decisão está em P ou não é computável.”
- ③ “Se um algoritmo tem complexidade  $O(n^{100})$ , ele é prático.”
- ④ “Se um algoritmo é exponencial, o problema *não* está em P.”
- ⑤ “Todo problema polinomial de busca pode ser transformado em um problema polinomial de decisão.”

# Gabarito – Exercício Proposto 1

- 1  $T(n) = 2n^{10} + 50n^7 + 100n \Rightarrow O(n^{10})$ . **Polinomial.**
- 2  $T(n) = n \cdot 2^n \Rightarrow$  domina a parte exponencial. **Não polinomial.**
- 3  $T(n) = 2^{100} \cdot n = O(n)$  – constante absurda mas fixa. **Polinomial.**  
( $2^{100}$  é uma constante – enorme, mas constante.)
- 4  $T(n) = n^{\sqrt{n}} \Rightarrow$  expoente cresce com  $n$ . **Não polinomial.**  
(É “subexponencial” ou “quase-exponencial”.)
- 5  $T(n) = (\log n)^{100} \in O(n^\epsilon)$  para qualquer  $\epsilon > 0$ . **Polinomial.**  
(Na verdade *polilogarítmico*, ainda mais rápido que polinomial!)
- 6  $T(n) = \sqrt{n^3} = n^{1,5} \in O(n^2)$ . **Polinomial.**

## Regra de Ouro

$T(n)$  é polinomial  $\iff$  existe constante  $k$  tal que  $T(n) \in O(n^k)$ .

# Gabarito – Exercício Proposto 2

## 1 Maior elemento:

- Decisão: “existe elemento  $\geq k$ ?”
- Algoritmo: varredura linear. Custo  $O(n)$ . **Está em P.**

## 2 Grafo bipartido:

- Decisão: “ $G$  é bipartido?”
- Algoritmo: BFS/DFS colorindo com 2 cores.  $O(|V| + |E|)$ . **Está em P.**

## 3 Determinante:

- Decisão: “ $\det(M) = 0$ ?” (ou  $\det(M) \leq k$ ?)
- Eliminação Gaussiana:  $O(n^3)$ . **Está em P.**

## 4 Caminho Hamiltoniano de peso mínimo (TSP):

- Decisão: “existe caminho hamiltoniano de peso  $\leq k$ ?”
- **Não se sabe se está em P.** É NP-completo.

## 5 2-Coloração:

- Decisão: “ $G$  pode ser colorido com 2 cores?” (equivale a ser bipartido).
- $O(|V| + |E|)$ . **Está em P.**

## Gabarito – Exercício Proposto 3

**(1) Polinomial em  $n$ ?** Sim: o laço faz  $n$  iterações, cada com uma multiplicação.

**(2) Polinomial no tamanho da entrada?**

O tamanho da entrada (em bits) é  $m = \lceil \log_2 n \rceil$ , logo  $n = 2^m$ . O algoritmo faz  $n = 2^m$  iterações  $\Rightarrow$  **exponencial** em  $m$ .

**(3) Explicação.**

Existem **duas medidas de tamanho** em jogo:

- Tamanho da **entrada**:  $\log_2 n$  bits.
- Tamanho da **saída**:  $2^n$  tem  $n$  bits. **O próprio resultado é exponencial no tamanho da entrada!**

Nenhum algoritmo pode produzir uma saída exponencial em tempo polinomial. Portanto, o problema (“calcular  $2^n$ ”) intrinsecamente exige tempo exponencial em função do tamanho da entrada.

### Moral

O *tamanho* correto da entrada é sempre medido em **bits**, e o *tamanho da saída* também importa.

## Gabarito – Exercício Proposto 4

**Passo 1 – Custo de  $A$ :**  $O(n \log n)$ . Saída tem tamanho  $O(n)$ .

**Passo 2 – Entrada de  $B$ :**  $m = O(n)$ . Custo de  $B$ :  $O(m^2) = O(n^2)$ .  
Saída de  $B$ :  $O(m^2) = O(n^2)$ .

**Passo 3 – Entrada de  $C$ :**  $k = O(n^2)$ . Custo de  $C$ :  $O(k^3) = O((n^2)^3) = O(n^6)$ .

**Passo 4 – Custo total:**

$$T(n) = O(n \log n) + O(n^2) + O(n^6) = O(n^6).$$

### Resposta

$O(n^6)$  – **polinomial**.

Mais uma vez: composições de algoritmos polinomiais são polinomiais, contanto que as saídas intermediárias tenham tamanho polinomial.

# Gabarito – Exercício Proposto 5

## 1 Caminho mais longo:

- Decisão: “dado  $G$ ,  $u$ ,  $v$  e  $k$ , existe um caminho *simples* de  $u$  a  $v$  com  $\geq k$  arestas?”

## 2 Maior conjunto independente:

- Decisão: “dado  $G$  e  $k$ , existe conjunto independente com  $\geq k$  vértices?”

## 3 Maior subsequência comum (LCS):

- Decisão: “dadas strings  $X$ ,  $Y$  e  $k$ , existe subsequência comum a  $X$  e  $Y$  com  $\geq k$  caracteres?”

## 4 Menor número de moedas:

- Decisão: “dados  $v$ , moedas  $c_1, \dots, c_n$  e  $k$ , existe forma de totalizar  $v$  com  $\leq k$  moedas?”

## Padrão

Toda otimização vira decisão introduzindo um limiar  $k$  e trocando “encontre o máximo/mínimo” por “existe solução com valor  $\geq / \leq k$ ?”.

# Gabarito – Exercício Proposto 6

## Algoritmo 1 – Força Bruta ( $O(n^3)$ ).

```
algoritmo tresSomamZero_v1(A, n)
  para i de 0 ate n-3 faca
    para j de i+1 ate n-2 faca
      para k de j+1 ate n-1 faca
        se A[i] + A[j] + A[k] = 0 entao
          retorne TRUE
        fim se
      fim para
    fim para
  fim para
  retorne FALSE
fim algoritmo
```

Custo:  $O(n^3)$  – polinomial.

## Algoritmo 2 – Ordenação + Dois Ponteiros ( $O(n^2)$ ).

- 1 Ordenar A:  $O(n \log n)$ .
- 2 Para cada  $i$ , procurar  $j < k$  com  $A[j] + A[k] = -A[i]$  usando dois ponteiros em  $O(n)$ .
- 3 Total:  $O(n \cdot n) = O(n^2)$ .

TRÊS-SOMAM-ZERO  $\in P$

# Gabarito – Exercício Proposto 7

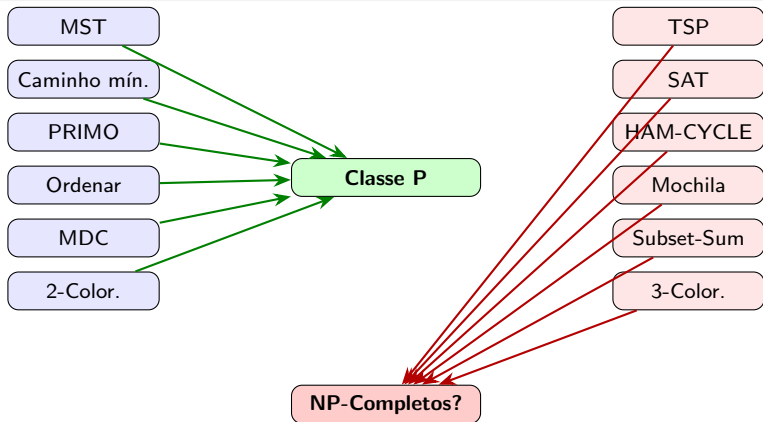
- 1 “Se um problema está em P, então pode ser resolvido rapidamente para qualquer entrada.”  
**Parcialmente falsa.** Está em P = existe algoritmo polinomial. Mas “rapidamente” depende do expoente e da constante.  $O(n^{100})$  é polinomial, mas inviável na prática.
- 2 “Todo problema de decisão está em P ou não é computável.”  
**Falsa.** Existem problemas *computáveis mas fora de P*: por exemplo, todos os problemas exponenciais ( $EXP \supsetneq P$ ) ou NP-completos, se  $P \neq NP$ .
- 3 “Se um algoritmo tem complexidade  $O(n^{100})$ , ele é prático.”  
**Falsa na prática.**  $n^{100}$  explode para qualquer  $n$  maior que 2. Embora seja polinomial em teoria, é inviável computacionalmente.
- 4 “Se um algoritmo é exponencial, o problema não está em P.”  
**Falsa.** A classificação em P é sobre *existência* de algoritmo polinomial. Um problema pode ter um algoritmo exponencial e *também* um algoritmo polinomial. Ex.: primalidade.
- 5 “Todo problema polinomial de busca pode ser transformado em um problema polinomial de decisão.”  
**Verdadeira (e a recíproca também vale, com cuidado).** Se temos um algoritmo que resolve o problema de busca, basta comparar a solução com o limiar  $k$  em tempo polinomial – o resultado é um algoritmo de decisão polinomial.

# Resumo da Aula 7

## O que aprendemos

- 1 **Complexidade de problemas** vs complexidade de algoritmos: uma é intrínseca, a outra depende da implementação.
- 2 Três tipos de problemas: **otimização**, **busca** e **decisão**. Decisão é a forma canônica para teoria da complexidade.
- 3 **Algoritmo polinomial**:  $T(n) = O(n^k)$  com  $k$  constante.
- 4 **Classe P**: problemas de decisão resolvíveis em tempo polinomial.
- 5 Polinomial = tratável; exponencial = intratável.
- 6 O tamanho da entrada é medido em **bits** – cuidado com pseudopolinomiais.
- 7  $P$  é fechada sob complemento, união, interseção, concatenação e composição.
- 8 Muitos problemas em P: MST, caminho mínimo, primalidade, MDC, 2-coloração, ordenação.
- 9 Muitos problemas famosos **aparentemente não** em P: TSP, SAT, mochila, hamiltoniano.

# Mapa Mental – Problemas → Classes



Vindo na próxima aula

Estudaremos formalmente a classe **NP** e a famosa questão **P vs NP**.

# Tabela Resumo – Conceitos Centrais

| Conceito              | Significado                                |
|-----------------------|--------------------------------------------|
| Algoritmo polinomial  | $T(n) = O(n^k)$ , $k$ constante            |
| Algoritmo exponencial | $T(n) = \Omega(a^n)$ , $a > 1$             |
| Problema tratável     | Tem algoritmo polinomial                   |
| Problema intratável   | Não tem algoritmo polinomial               |
| Problema de decisão   | Resposta SIM/NÃO                           |
| Classe P              | Decisão + polinomial                       |
| Classe NP (preview)   | Decisão + verificável em polinomial        |
| Pseudopolinomial      | Polinomial no valor, não no tamanho (bits) |

## Mensagem Principal

A classe P formaliza nossa intuição de “problema fácil”. Ela dá nome a uma fronteira fundamental entre o *tratável* e o *intratável*. Muitos problemas cotidianos (ordenação, caminhos mínimos, fluxo) estão em P. Outros, aparentemente simples de enunciar, não estão – e sobre eles se ergue o maior problema em aberto da computação:  $P \stackrel{?}{=} NP$ .

# Próxima Aula

## Aula 8 – A Classe NP e Verificação Polinomial

- Algoritmos de **verificação** e o conceito de **certificado**.
- A classe **NP**: definição formal.
- Por que  $P \subseteq NP$ .
- A questão  $P \stackrel{?}{=} NP$  e suas consequências.
- Problemas **NP-Completo**s: o que são e por que importam.
- Redução polinomial:  $X \preceq Y$ .
- Exemplos: SAT, Circuito Hamiltoniano, 3-Coloração.

## Dúvidas?

# Referências



T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Algoritmos: Teoria e Prática*. 3ª ed, Elsevier, 2012. Capítulo 34 – Problemas NP-Completo, Seções 34.1 e 34.2.



P. Feofiloff. *Minicurso de Análise de Algoritmos*. IME-USP, 2021.



P. Feofiloff. *Análise de Algoritmos – Complexidade e problemas NP-completos*. Notas de aula disponíveis em [ime.usp.br/~pf/analise\\_de\\_algoritmos/](http://ime.usp.br/~pf/analise_de_algoritmos/).



M. Sipser. *Introduction to the Theory of Computation*. 3rd ed, Cengage, 2013. Capítulo 7.



J. Kleinberg, É. Tardos. *Algorithm Design*. Pearson, 2005. Capítulo 8.