Introdução à teoria da complexidade computacional

Objetivo: Estudar e comparar a dificuldade de problemas

* Problema A é mais difícil que Problema B

quando a complexidade do melhor algoritmo para A é maior que o melhor algoritmo para B.

- Linguagens ⟶ problemas
- Máquinas de Turing ⟶ algoritmos

Problema de otimização (maior caminho) × Problema de decisão (caminho ≥ k)

(formalizados naturalmente como linguagens)

Complexidade de tempo

P: problemas que podem ser resolvidos por algoritmos determinísticos em tempo polinomial

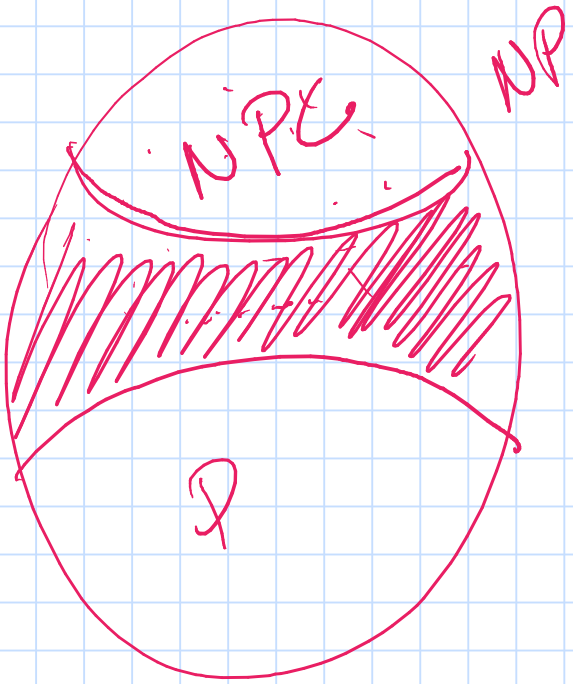
NP: problemas que possuem verificador polinomial

↳ De onde vem o termo?

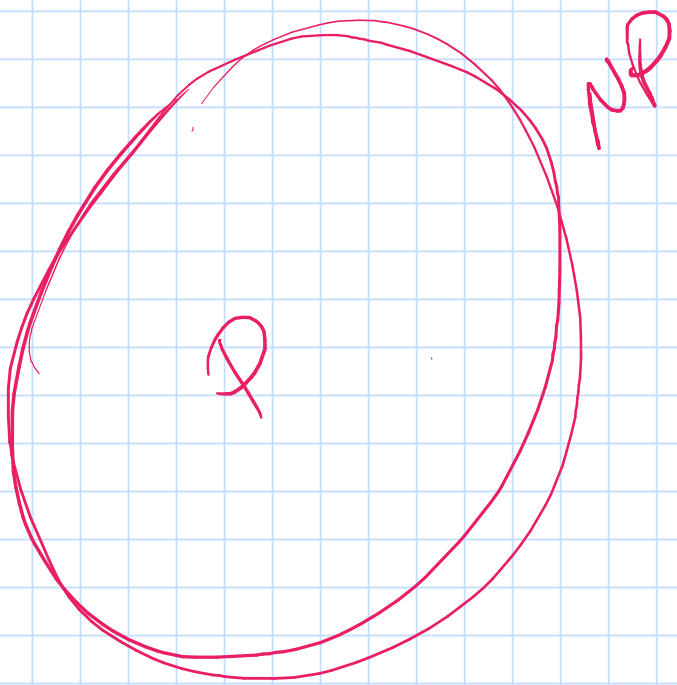
Verificador: Algoritmo que recebe uma instância e uma solução, e verifica se a solução resolve o problema.

P x NP

- $P \subseteq NP$
- Não sabemos se existem ou não algoritmos polinomiais para muitos problemas em NP



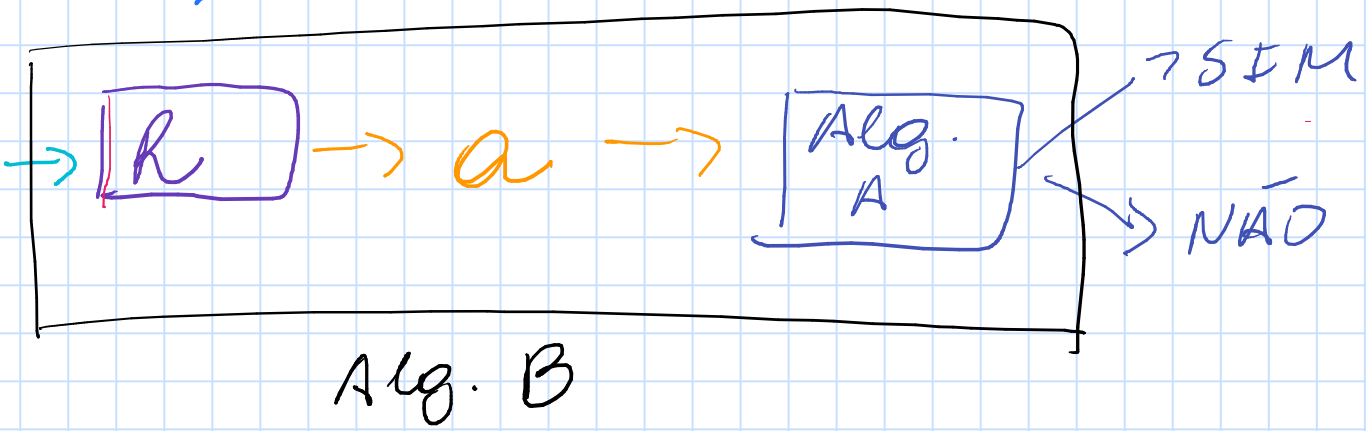
Se $P \neq NP$



NP-completude

↳ Subclasse com os problemas "mais difíceis" de NP (caracterizam a complexidade da classe inteira)

Redução: $B \leq A$



Dizemos que um problema B é redutível ao problema A se existe uma transformação R tal que, para cada instância b de B, produz instância a=R de A que é equivalente a b (possui mesma resposta)

Suponha que R é eficiente e $B \leq A$.

Se A é fácil $\Rightarrow$ B é fácil

Se B é difícil $\Rightarrow$ A é difícil

Um problema de decisão B é NP-completo se:

1. $B \in NP$

2. Todo problema de decisão C em NP é redutível polinomialmente a B

- Cook e Levin mostraram que o problema da satisfabilidade (SAT) é NPC.

SAT: Dada fórmula CNF, verificar se é possível atribuir valores V e F às variáveis de forma que a fórmula é verdadeira.

Teorema: Se B for NPC e B for redutível em tempo polinomial a C, para C em NP, então C é NPC.

B é difícil => A é difícil