Algoritmos e Estrutura de Dados

Aula 17 – Estrutura de Dados: Complexidade e Completude

Prof. Tiago A. E. Ferreira

Introdução

- Complexidade, definição genérica:
  - Complexidade é a escola filosófica que vê o mundo como um todo indissociável e propõe uma abordagem multidisciplinar para a construção do conhecimento. Contrapõe-se à *causalidade* por abordar os fenômenos como totalidade orgânica.

Complexidade Computacional

- Teoria da complexidade computacional é a parte da teoria da computação que estuda os recursos necessários durante o cálculo para resolver um problema.
- A teoria da complexidade difere da teoria da computabilidade a qual se ocupa de possibilidade de se solucionar um problema com algoritmos sem levar em conta os recursos necessários para ele.

Completude

- Completude tem a ver com a completa descrição ou abrangência de um modelo, geralmente abstrato, com o problema/universo que ele envolve.
- Ex: o programa:
  - x=input(digite um número)
  - print x+1
- É completo para os inteiros, pois dado qualquer inteiro, a operação será executada.

Completude

- Demonstrado por Kurt Gödel, o *teorema da incompletude*, garante que todos os cálculos lógicos de operações simples sobre os inteiros não podem ser simultaneamente *completos* e *consistentes*.

Decidibilidade

- Um problema édito **decidível** se há uma máquina de Turing (computador) que possa computar, independente do tempo que leve. Caso contrário, é chamado de **indecidível**.
- Exemplo: dizer se um número é primo ou não é trivialmente decidível, mas é exponencial no tamanho da entrada.

Computabilidade

- A Teoria da Computabilidade estuda:
  - Questões de decibilidade
  - Modelos de computação
  - Capacidade computacional.

Sumarizando

- A teoria da complexidade está preocupara com resolver problemas difíceis de maneira eficiente.
- A teoria da computabilidade está preocupada com a forma com que as computações são feitas (modelo abstrato de computação).
- O foco da aula de hoje é **complexidade**.

Definições

- Algoritmo polinomial é do tipo:
  - n^K, onde n é o tamanho da entrada e k é constante.
- Um algoritmo exponencial é do tipo:
  - K^{cn}, onde n é o tamanho da entrada e k e c são constantes maiores que zero.
- Problemas polinomiais são ditos **tratáveis** e exponenciais **intratáveis** (difíceis).

Exemplos de Custos

Função de Custo	n=10	n=20	n=30	n=40	n=50	n=60
n	10^{-5} s	2.10^{-5} s	3.10^{-5} s	4.10^{-5} s	5.10^{-5} s	6.10^{-5} s
n^2	10^{-4} s	4.10^{-4} s	9.10^{-4} s	16.10^{-4} s	25.10^{-4} s	36.10^{-4} s
n^3	10^{-3} s	8.10^{-3} s	27.10^{-3} s	64.10^{-3} s	125.10^{-3} s	216.10^{-3} s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	10^{-3} s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc.
3^n	59.10^{-3} s	58 min	6,5 anos	3855 séc.	10^8 séc.	10^{13} séc.

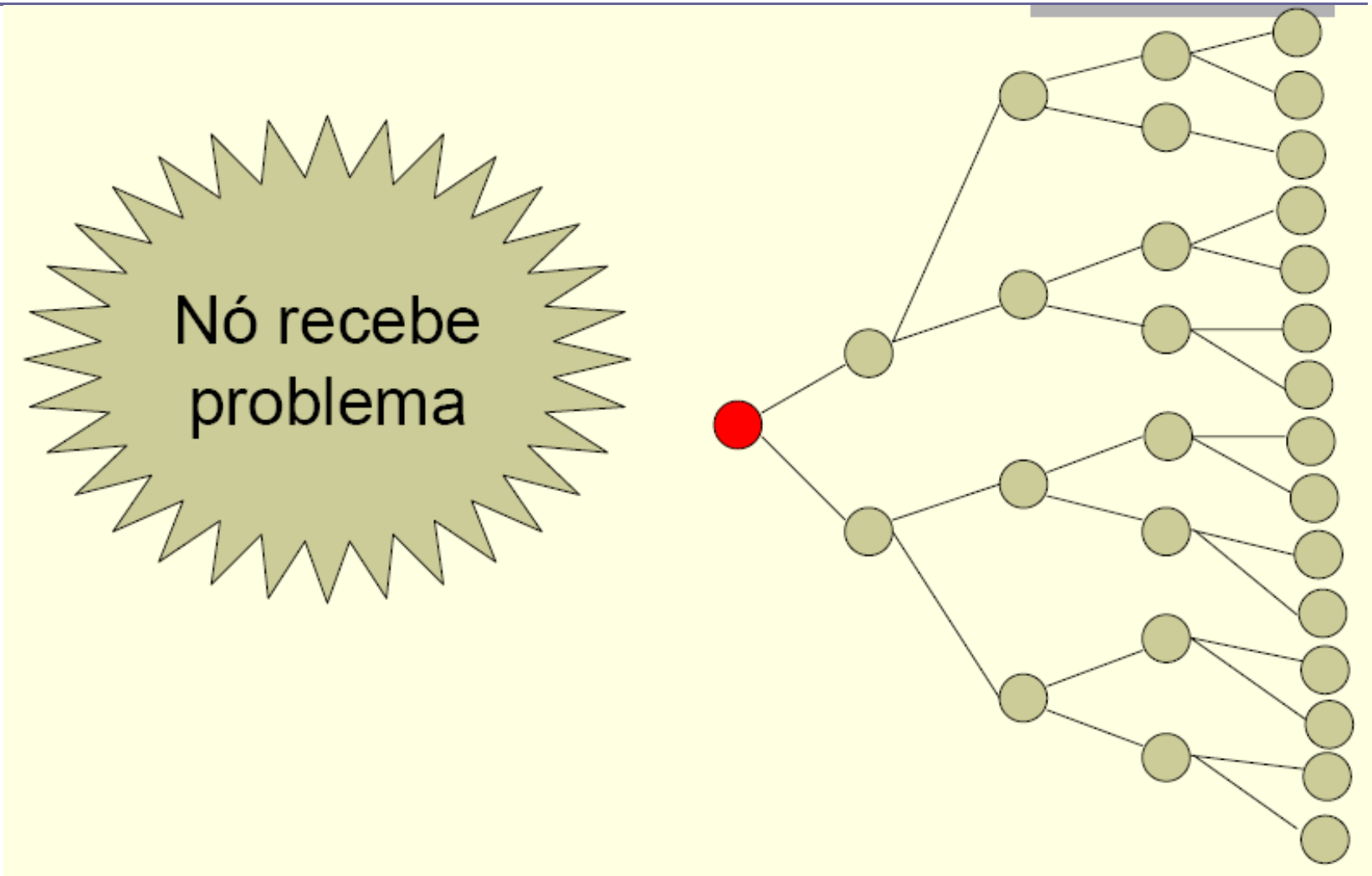
Algoritmos Determinísticos

- São algoritmos que apresentam comportamento previsível. Dada uma determinada entrada, o algoritmo apresenta sempre a mesma saída.
- Máquinas determinísticas são aquelas que só podem estar em um estado por vez.
  - Um estado define o que a máquina está fazendo em um determinado tempo.

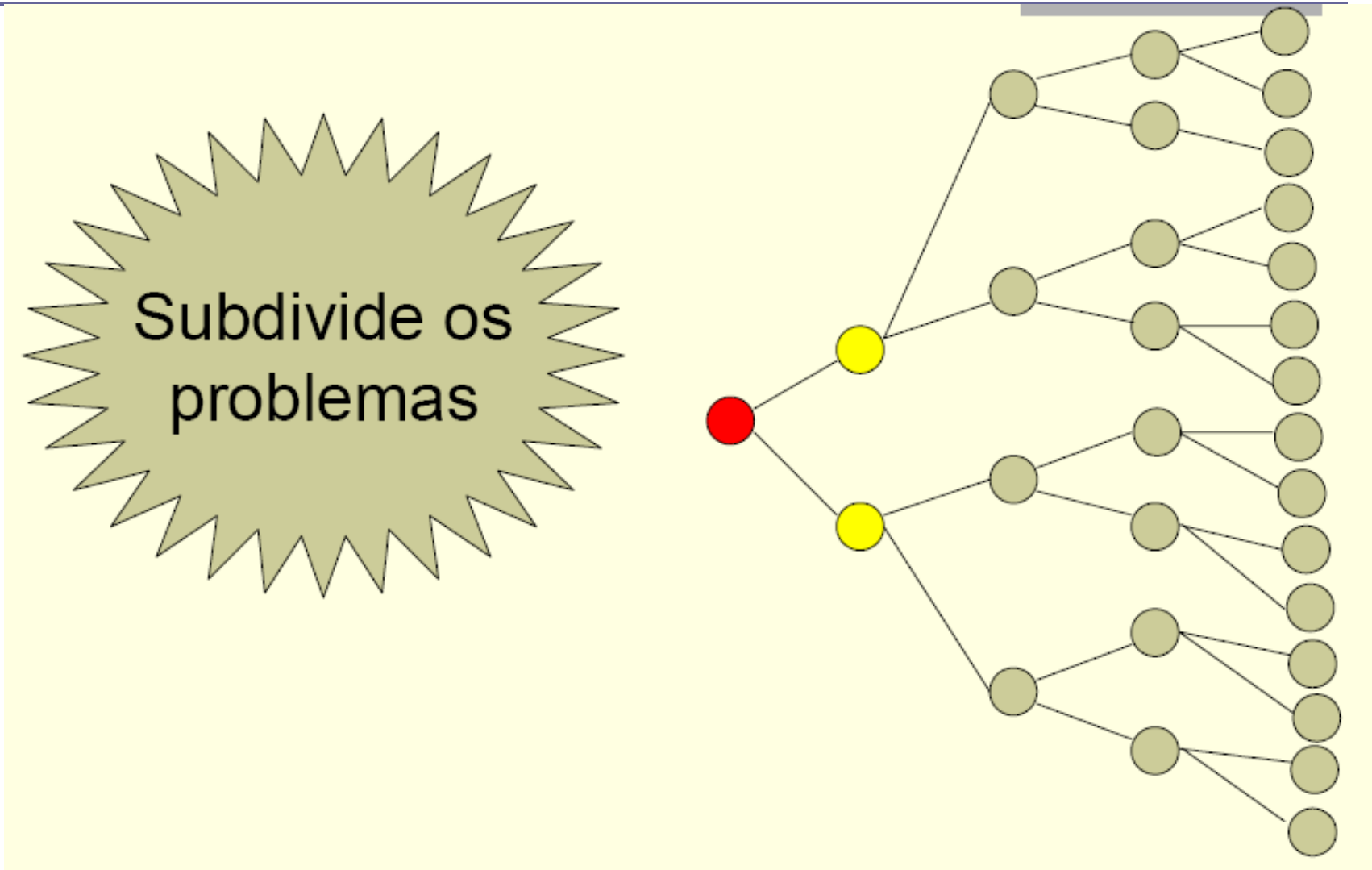
Algoritmos Não-Determinísticos

- São algoritmos que apresentam comportamento guiado por uma certa distribuição estatística. Dada uma determinada entrada, o algoritmo apresenta uma saída com uma dada probabilidade.

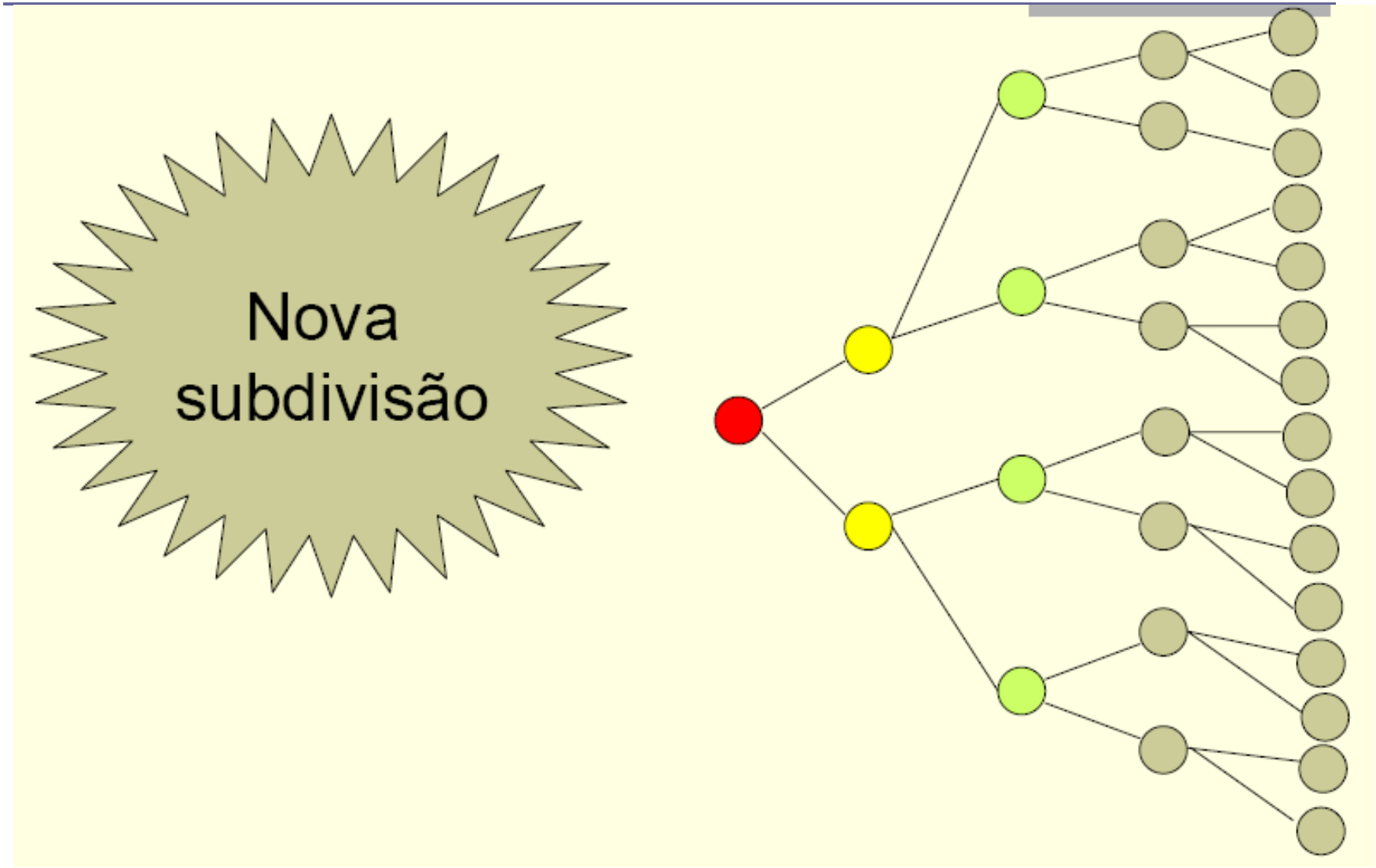
Algoritmos Não-Determinísticos



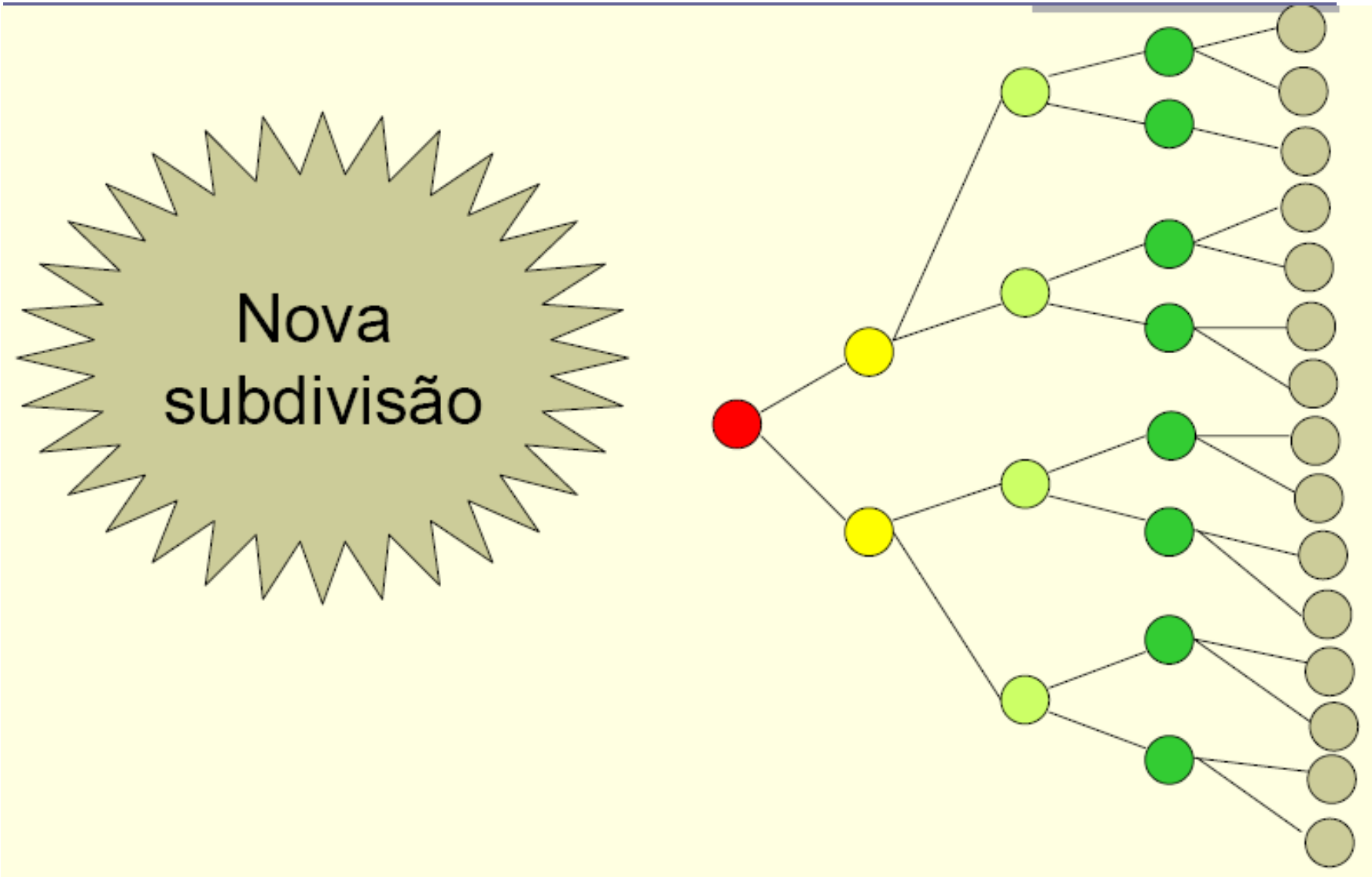
Algoritmo Não-Determinístico



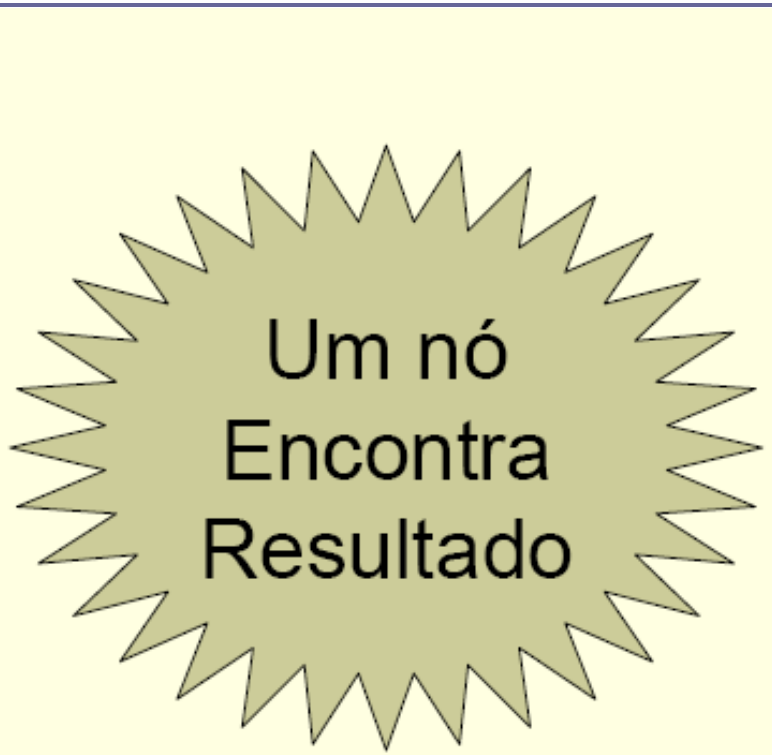
Algoritmo Não-Determinístico

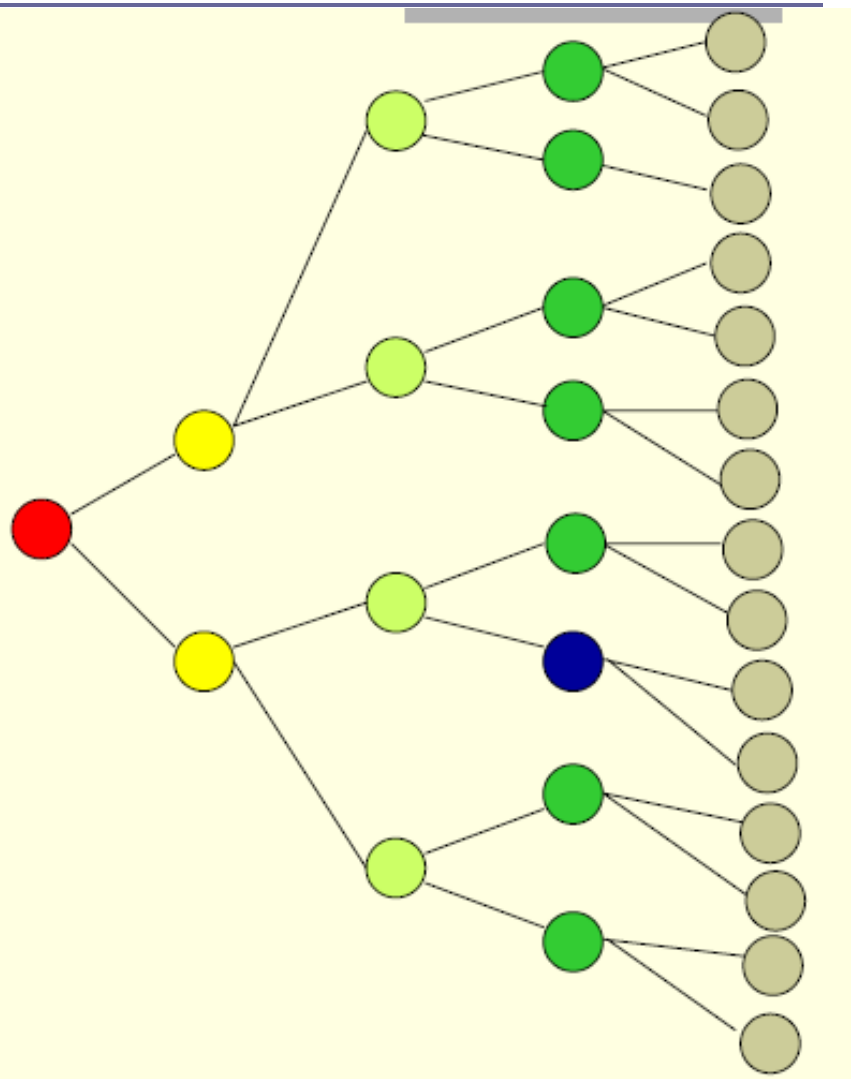


Algoritmos Não-Determinísticos

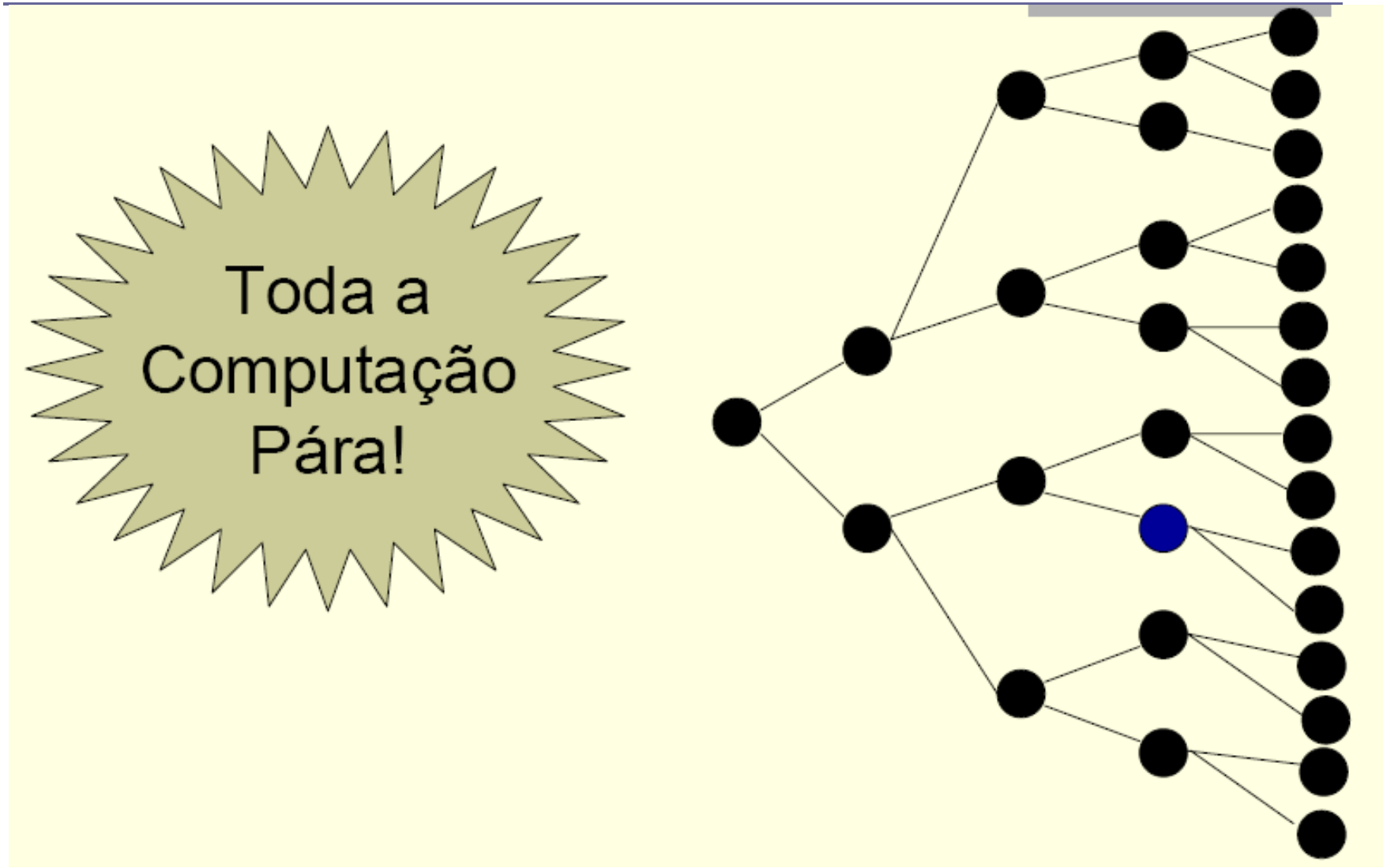


Algoritmos Não-Determinísticos





Algoritmos Não-Determinísticos



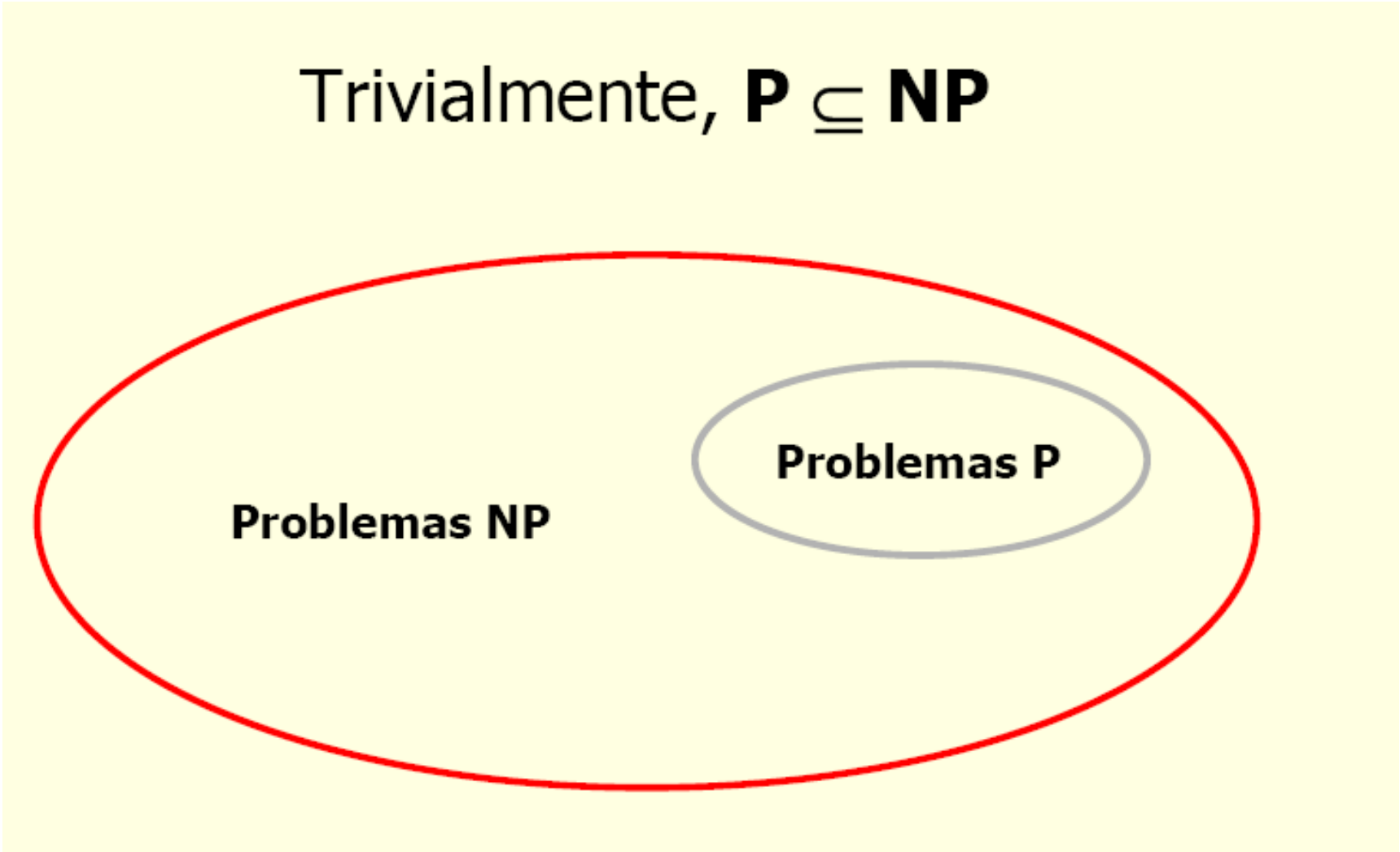
Algoritmos Não-Determinísticos

- Em outras palavras:
  - Problema inicial com resolução em $\mathbf{2^n}$ passos em máquina determinística (pior caso)
  - Em uma máquina não-determinística leva tempo n.
- Intrigante:
  - Será que existe algum problema que apenas uma máquina não-determinística resolva?
  - **Não.** Uma máquina não-determinística calcula apenas mais rápido, porém é equivalente a máquinas determinísticas.

Classe de Problemas

- Problemas P
  - É a classe dos problemas resolvíveis em tempo **Polinomial** em máquina determinística.
- Problemas NP
  - É a classe dos problemas resolvíveis em tempo **Polinomial em máquina Não-determinística**.

Classe de Problemas



Classe de Problemas

- Problemas NPC ou NP-Completos
  - É a classe dos problemas que são Polinomiais em máquinas Não-Determinísticas e são Completos no sentido que, para cada um deles, *todos* os problemas de **NP** podem ser reduzidos a ele em tempo polinomial (em máquina determinística).

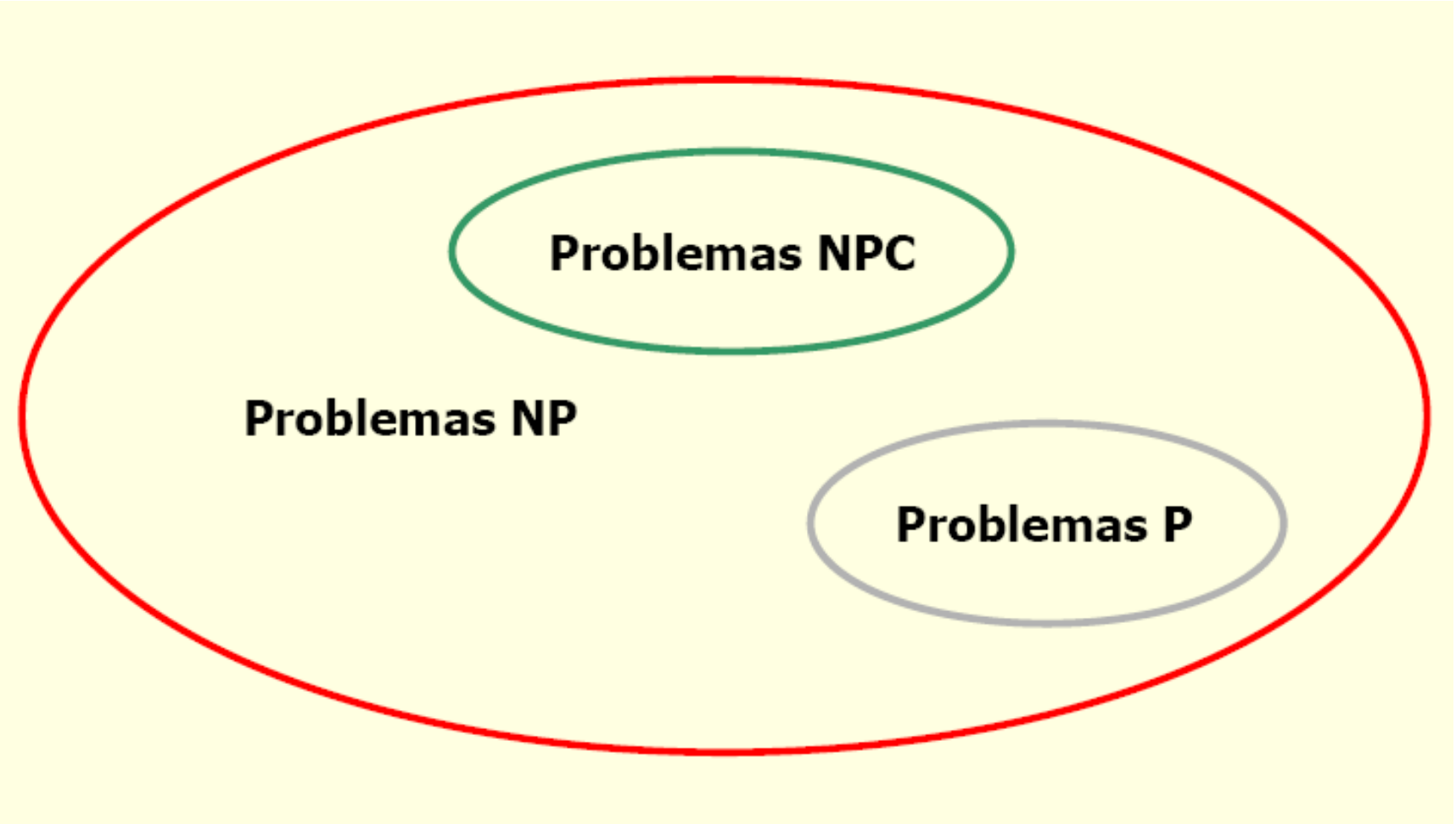
NP-Completo

- Formalmente, um problema é NP-Completo se:
  - O problema é NP
  - É NP-Difícil
- NP-Difícil é um problema que pode ser **reduzido** a outro problema NP, em outras palavras, se tem mesma complexidade de um problema NP.

NP-Completo

- Problemas NPC ou NP-Completos
  - Se *um* problema de **NPC** puder ser resolvido em tempo polinomial em máquina determinística, então todos os problemas de **NP** o poderão, e **NP = P;**
  - Se *um* problema de **NPC** for *provadamente* intratável, então todos os problemas em **NPC** também o serão. Daí: (**NPC** intercepta **P**) se somente se (**NP = P**). A maioria crê que **NPC** é disjunto de **P**, portanto **NP≠P**.
- Este é um dos problemas em aberto que pode mudar todo o entendimento da computabilidade na prática, se for provado que NP = P

Acredita-se



Classe de Problemas

- Como provar que um problema A é NPC ?
  - Mostrar que A pertence a NP;
  - Selecionar um problema A', conhecido, de NPC;
  - Construir uma transformação f:A'→A;
  - Mostrar que f é uma transformação polinomial.

Problemas NP-Completos

- TSP - *TheTravelingSalesmanProblem*(PCV -**O Problema do Caixeiro Viajante**):
  - Um vendedor tem que visitar cada uma de n cidades, terminando naquela por onde começou;
  - Cada arco tem um custo associado;
  - O problema, na versão de decisão, pergunta: “Há uma turnê de custo menor ou a igual a um valor, dado?”;
  - A versão de otimização pergunta: “Qual o menor custo possível para uma turnê, e qual é ela?”

Problemas NP-Completos

- HC – *The Hamilton Circuit* (= *Cycle*)
- **O Circuito Hamiltoniano**:
  - Dado um grafo orientado, determine se há um ciclo que visite cada nó exatamente uma vez, terminando no nó inicial.

Problemas NP-Completos

- ZOK (*The Zero-One Knapsack*)
- (**O Problema da Mochila**, cada item sendo totalmente tomado ou não tomado):
  - Uma mochila tem capacidade de M kilogramas;
  - Num cofre, há N itens, cada um com seu peso (p_i) e seu valor (v_i);
  - Escolha que itens levar na mochila, de modo que o total de peso seja menor ou igual a M, e o total dos valores seja máximo.

Problemas NP-Completos

- Boolean Satisfiability Problem (SAT)
- **Satisfatibilidade de expressão booleana**:
  - Dada uma expressão proposicional na forma normal conjuntiva, determine se há algum conjunto de atribuições “true” e “false” aos símbolos proposicionais, que torne o valor da expressão igual a “true”.
- Foi o primeiro problema NP-Completo proposto por Stephen Cookem em 1971. Foi demonstrado através de máquina de Turing.

Técnicas de trabalhar com algoritmos NP-Completos

- Não faz sentido dizer que um problema é NP e simplesmente esquecê-lo.
- Existem técnicas de solução que vão tentar achar uma solução otimizada de um problema NP.
- Essas técnicas trabalham na questão de comprometer algo para achar uma solução:
  - Optimalidade
  - Robustez
  - Eficiência
  - Completude das Soluções

Técnicas de trabalhar com algoritmos NP-Completos

- Algumas dessas técnicas:
  - Algoritmos de aproximação (solução pode ser ou não ótima)
    - Comprometem a “optamibilidade”.
  - Algoritmos que tratam problemas NPC em tempo polinomial na média.
    - Comprometem a “eficiência”.
  - Algoritmos que funcionam para casos especiais da entrada.
    - Comprometem a “completude”.
  - Algoritmos exponenciais que trabalham bem para pequenas entradas (backtracking).