

Complexité CM6

Antonio E. Porreca
aeporreca.org/complexite

**Divinations 🌊 et vérifications 🔍,
ou le non déterminisme
dans le monde réel**

Algorithmes non déterministes

```
fonction hamiltonien( $V, E$ )  
   $n := |V|$   
   $perm := \text{tableau}(n)$   
  pour  $i := 0$  à  $n - 1$  faire  
     $perm[i] := \text{devine}(0, \dots, n - 1)$   
  pour chaque  $v \in V$  faire  
    si  $perm$  ne contient pas  $v$  exactement 1 fois alors  
      rejeter  
  pour  $i := 0$  à  $n - 1$  faire  
    si  $(perm[i], perm[(i + 1) \bmod n]) \notin E$  alors  
      rejeter  
  accepter  
fin
```

Algorithmes non déterministes

fonction hamiltonien(V, E)

$n := |V|$

$perm := \text{tableau}(n)$

pour $i := 0$ à $n - 1$ **faire**

$perm[i] := \text{devine}(0, \dots, n - 1)$

pour chaque $v \in V$ **faire**

si $perm$ ne contient pas v exactement 1 fois **alors**
rejeter

pour $i := 0$ à $n - 1$ **faire**

si $(perm[i], perm[(i + 1) \bmod n]) \notin E$ **alors**
rejeter

accepter

fin

perm est-elle une permutation ?

Algorithmes non déterministes

fonction hamiltonien(V, E)

$n := |V|$

$perm := \text{tableau}(n)$

pour $i := 0$ à $n - 1$ **faire**

$perm[i] := \text{devine}(0, \dots, n - 1)$

pour chaque $v \in V$ **faire**

si $perm$ ne contient pas v exactement 1 fois **alors**
rejeter

pour $i := 0$ à $n - 1$ **faire**

si $(perm[i], perm[(i + 1) \bmod n]) \notin E$ **alors**
rejeter

accepter

fin

perm est-elle une permutation ?

perm est-il un cycle dans le graphe ?

Algorithmes non déterministes

fonction hamiltonien(V, E)

$n := |V|$

$perm := \text{tableau}(n)$

pour $i := 0$ à $n - 1$ **faire**

$perm[i] := \text{devine}(0, \dots, n - 1)$

$O(n \log n)$ bits devinés

pour chaque $v \in V$ **faire**

si $perm$ ne contient pas v exactement 1 fois **alors**
rejeter

$perm$ est-elle une permutation ?

pour $i := 0$ à $n - 1$ **faire**

si $(perm[i], perm[(i + 1) \bmod n]) \notin E$ **alors**
rejeter

$perm$ est-il un cycle dans le graphe ?

accepter

fin

Simulation du non déterminisme dans le monde réel*

```
from nondeterminism import *

@nondeterministic
def hamiltonian(vertices, edges):
    n = len(vertices)
    perm = []
    for i in range(n):
        v = guess(vertices)
        perm.append(v)
    for v in vertices:
        if perm.count(v) != 1:
            reject()
    for i in range(n):
        if (perm[i], perm[(i+1)%n]) not in edges:
            reject()
    accept()
```

* github.com/aeporreca/nondeterminism

Simulation du non déterminisme dans le monde réel*

```
from nondeterminism import *
```

```
@nondeterministic
```

```
def hamiltonian(vertices, edges):
```

```
    n = len(vertices)
```

```
    perm = []
```

```
    for i in range(n):
```

```
        v = guess()
```

```
        perm.append(v)
```

```
    for v in vertices:
```

```
        if perm.count(v) > 1:
```

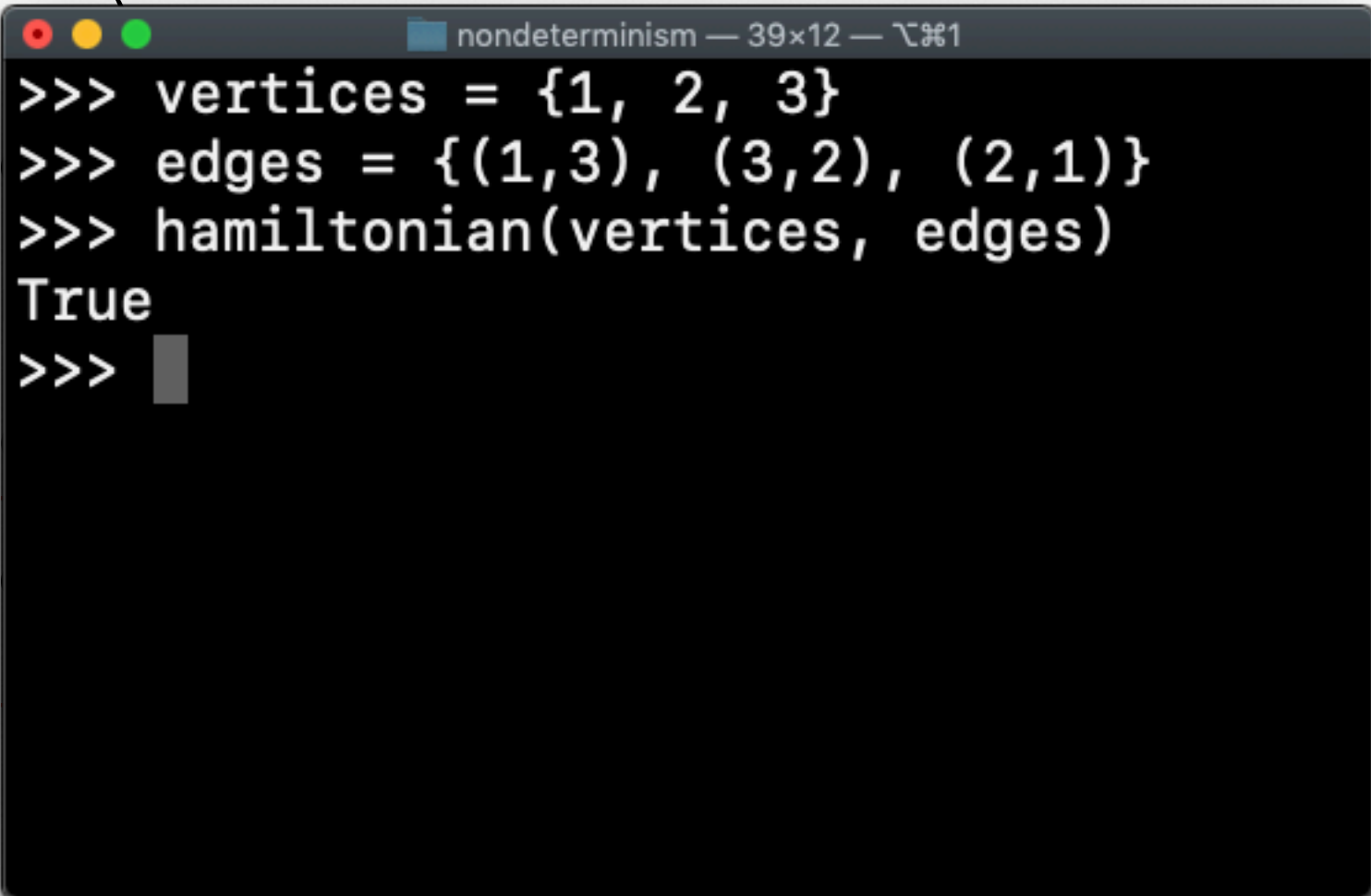
```
            reject()
```

```
    for i in range(n):
```

```
        if (perm[i] - perm[i-1]) % n != 1 and
```

```
            reject()
```

```
    accept()
```



```
nondeterminism — 39x12 — 100%
>>> vertices = {1, 2, 3}
>>> edges = {(1,3), (3,2), (2,1)}
>>> hamiltonian(vertices, edges)
True
>>>
```

* github.com/aeporreca/nondeterminism

Proposition 2-AO (p. 56)

Caractérisation existentielle de NP

Les deux conditions suivantes sont **équivalentes** :

- Le langage L est reconnu en temps polynomiale par une machine de Turing **non déterministe** N
- Il existe une machine de Turing **déterministe** M qui fonctionne en temps polynomial et un polynôme $p(n)$ tels que

$$x \in L \text{ ssi } \exists y \in \{0,1\}^{p(n)} M(x, y) \text{ accepte}$$

Le mot y qui justifie l'appartenance de x à L est appelé **preuve** ou **certificat**



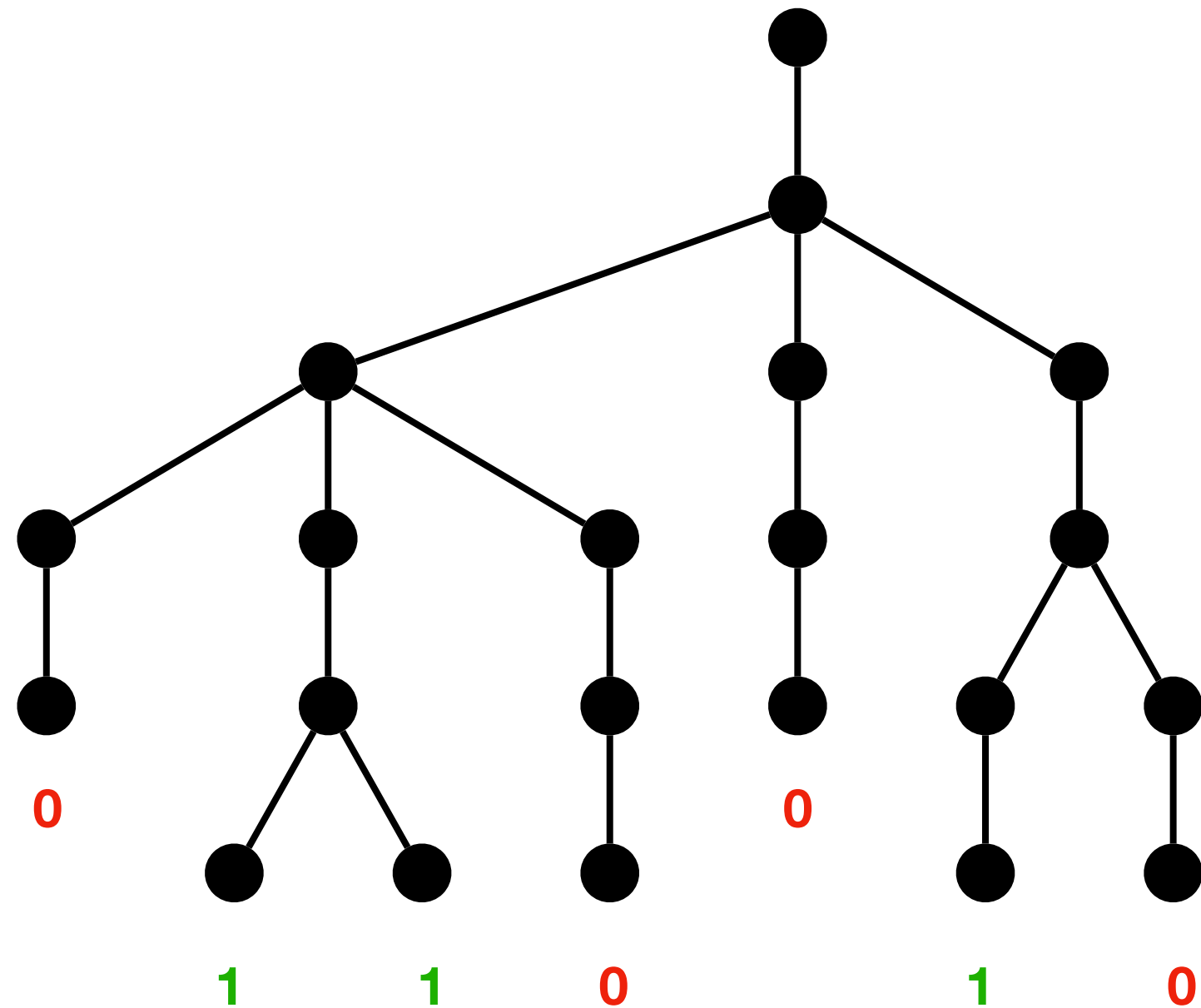
Idée de la démonstration

Le certificat y correspond au **chemin acceptant**
pour l'entrée x dans la machine non déterministe N
qui reconnaît le langage L

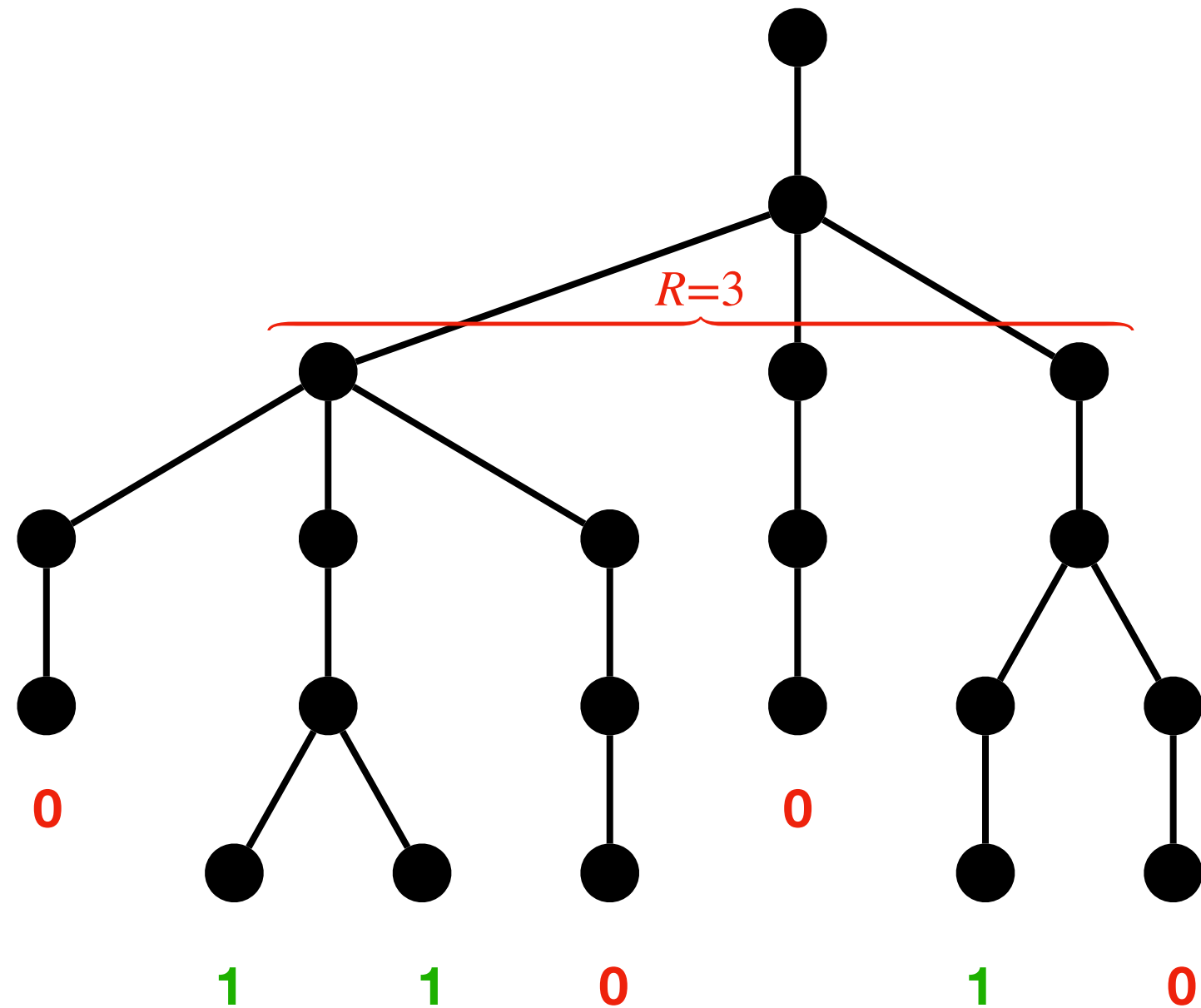
Demonstration

- Supposons que N reconnait L en temps polynomial $q(n)$
- À chaque étape, N choisit entre un nombre $\leq R$ de transitions possibles (R est constant, il ne dépend pas de l'entrée x)
- Chaque choix peut être décrite avec $\lceil \log R \rceil$ bits

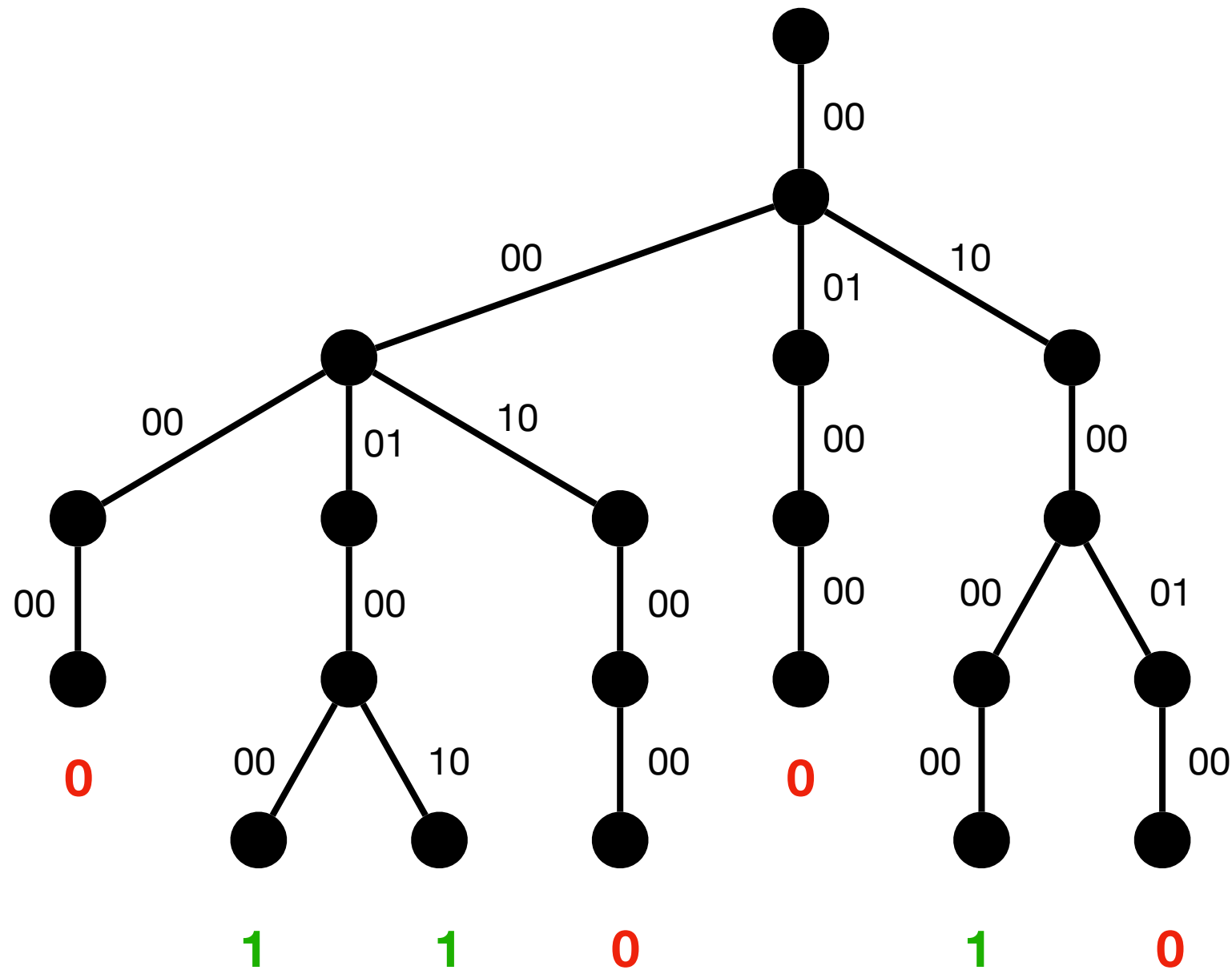
Arbre de calcul de N sur x



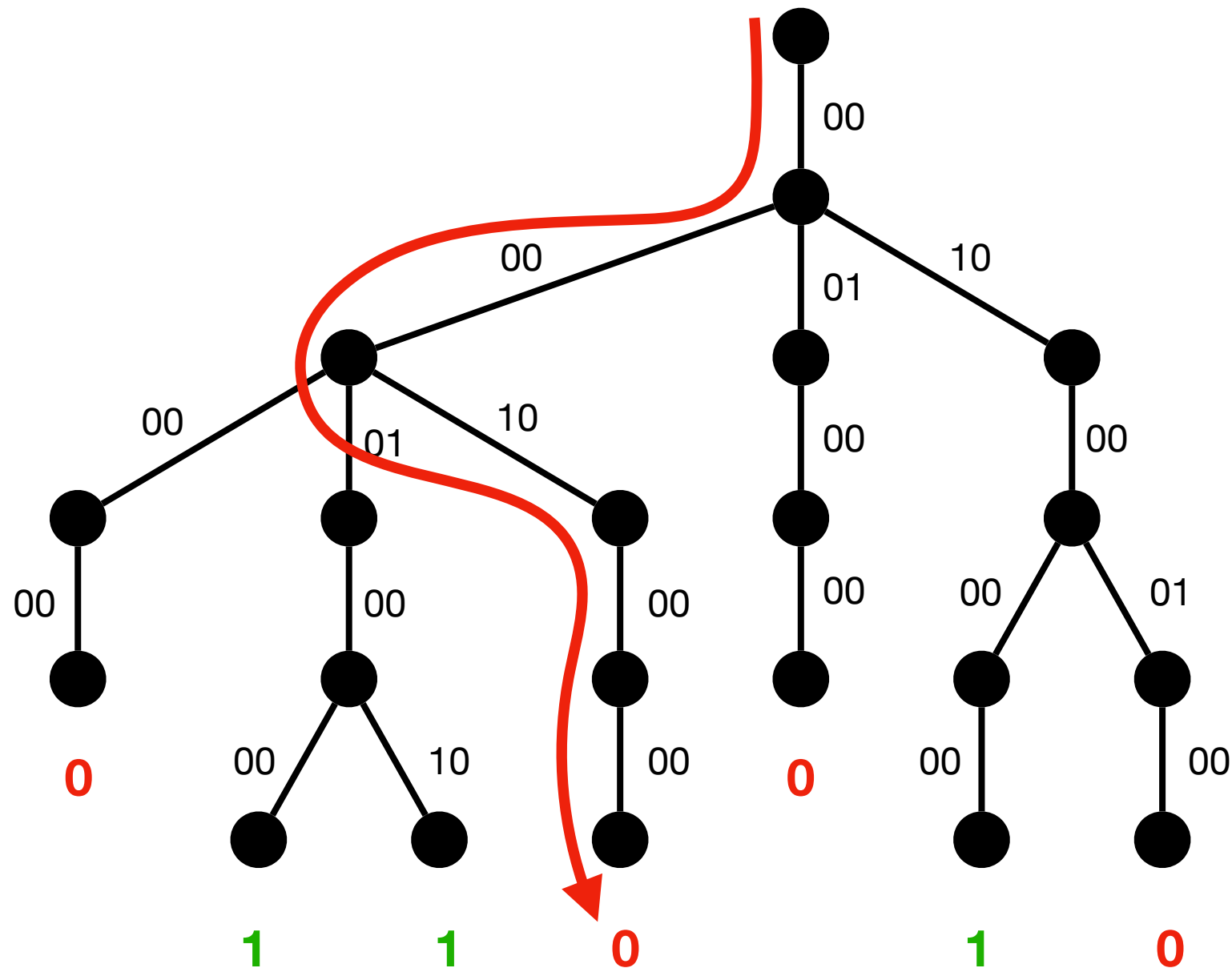
Arbre de calcul de N sur x



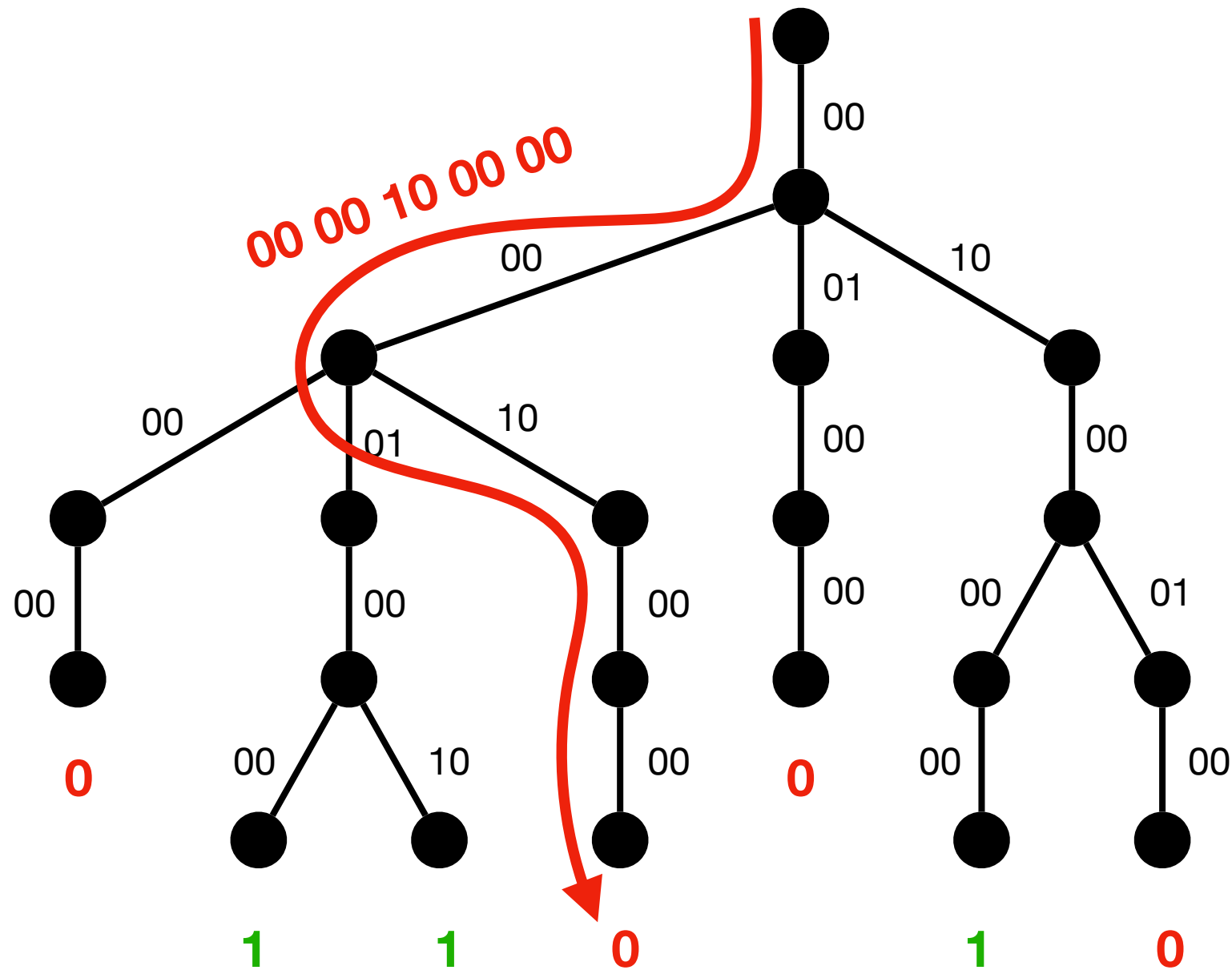
Arbre de calcul de N sur x



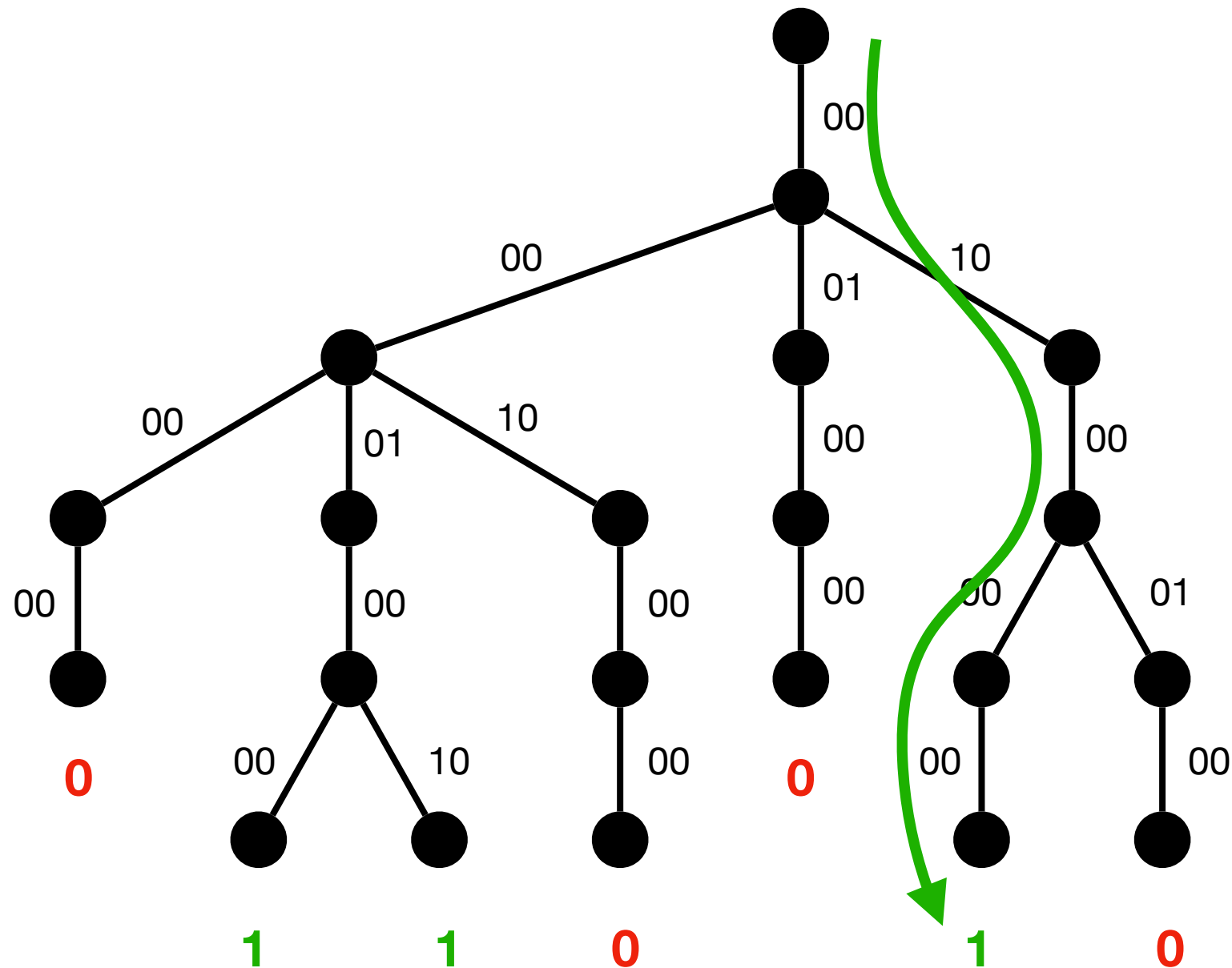
Arbre de calcul de N sur x



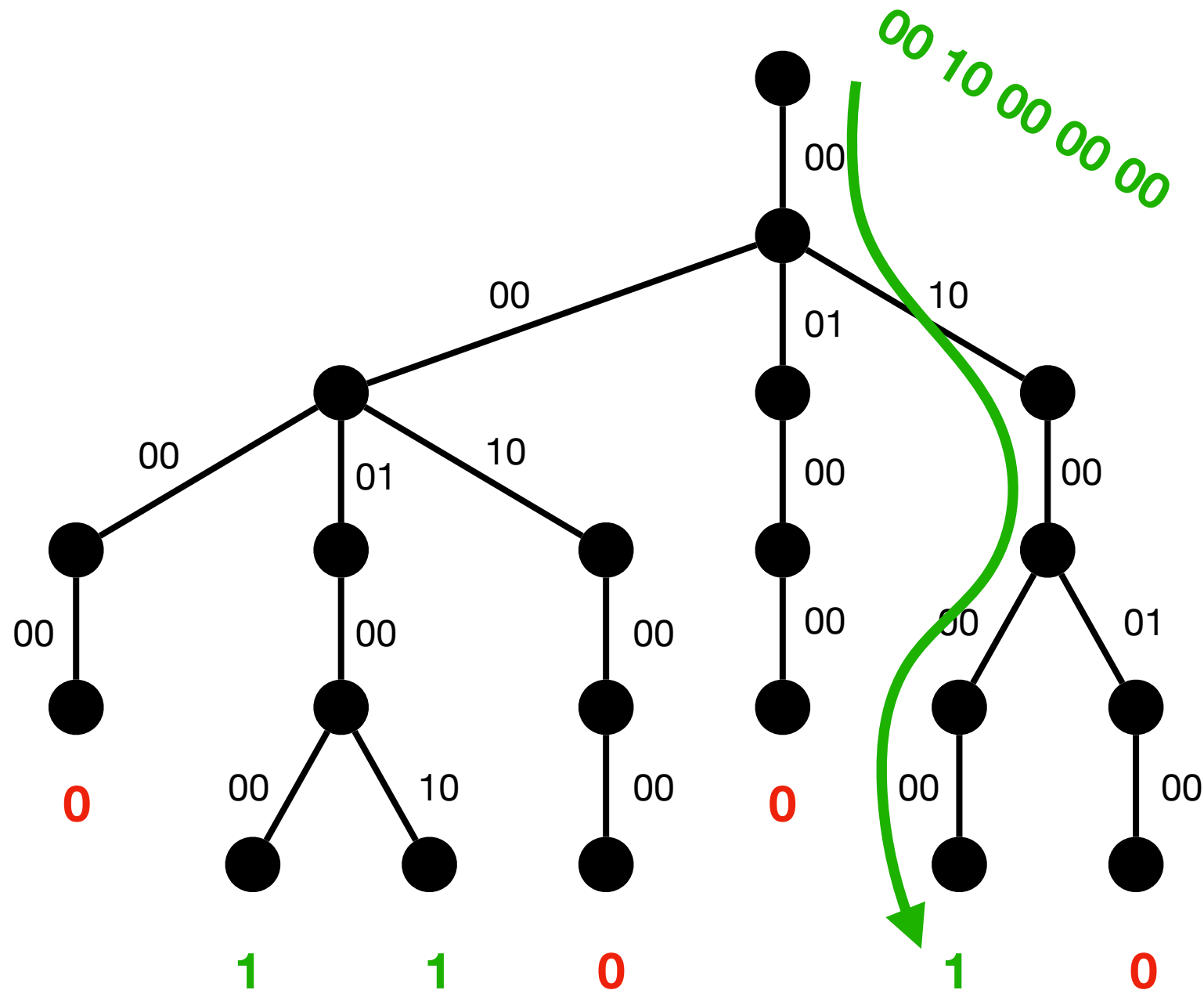
Arbre de calcul de N sur x



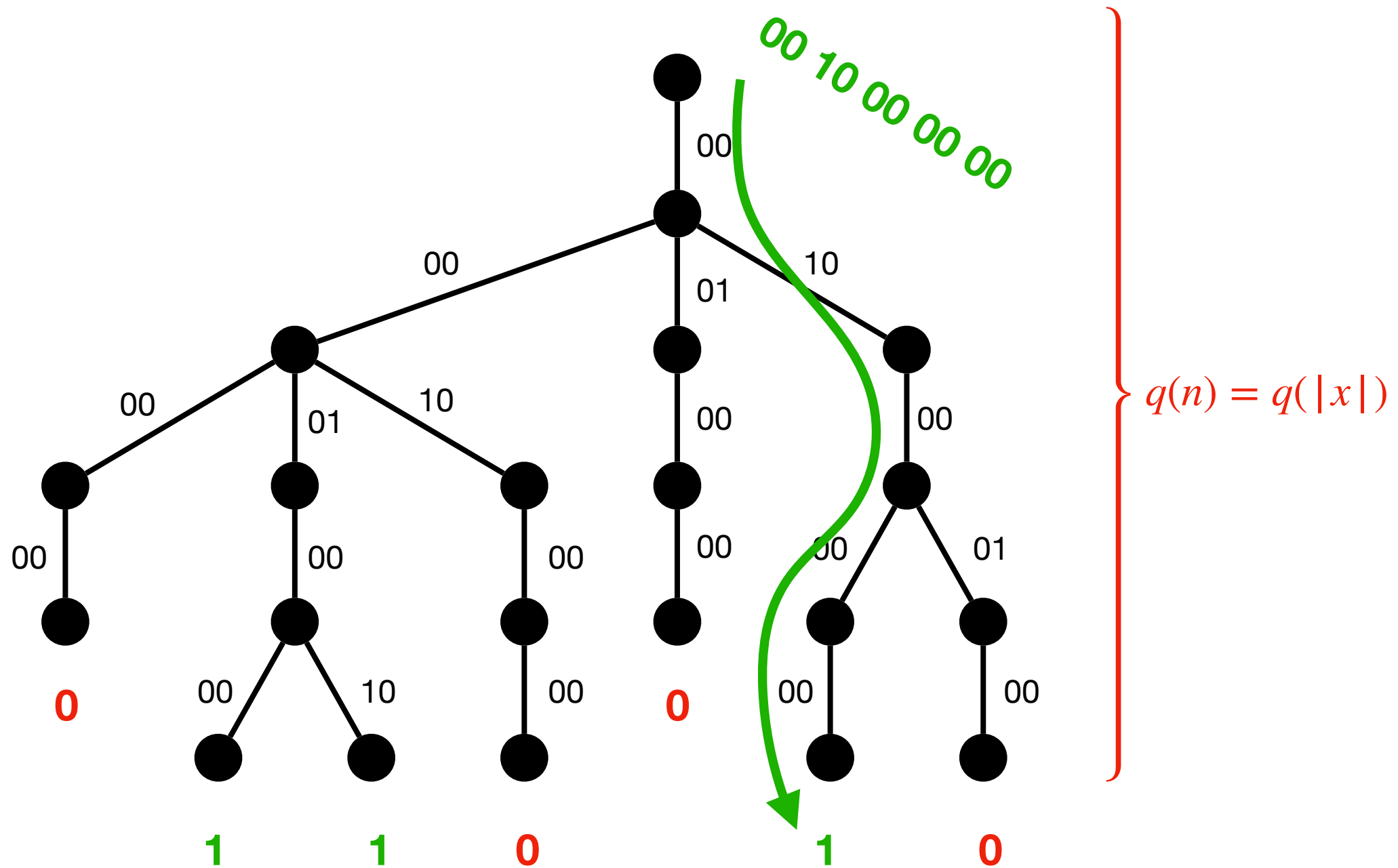
Arbre de calcul de N sur x



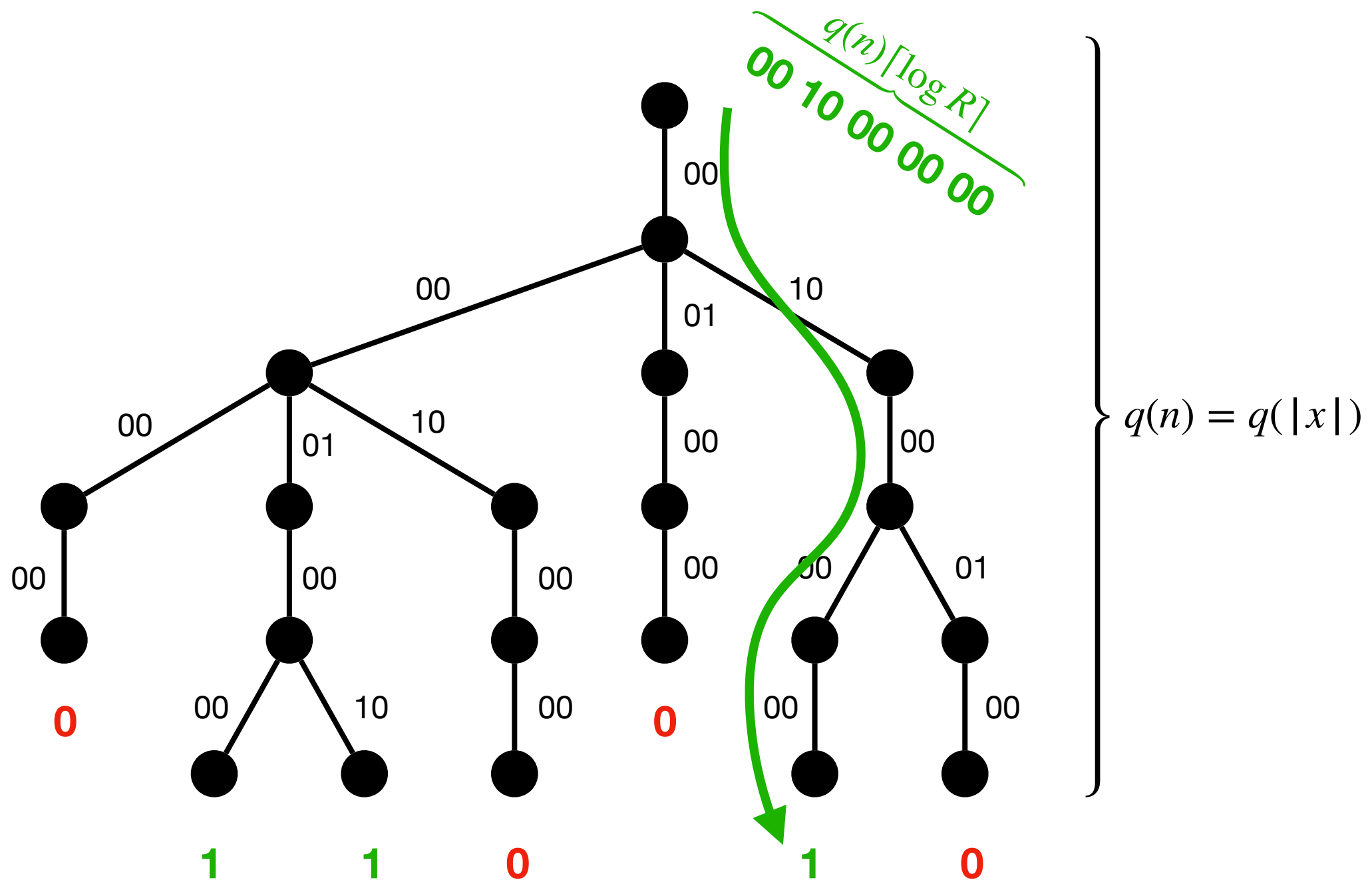
Arbre de calcul de N sur x



Arbre de calcul de N sur x

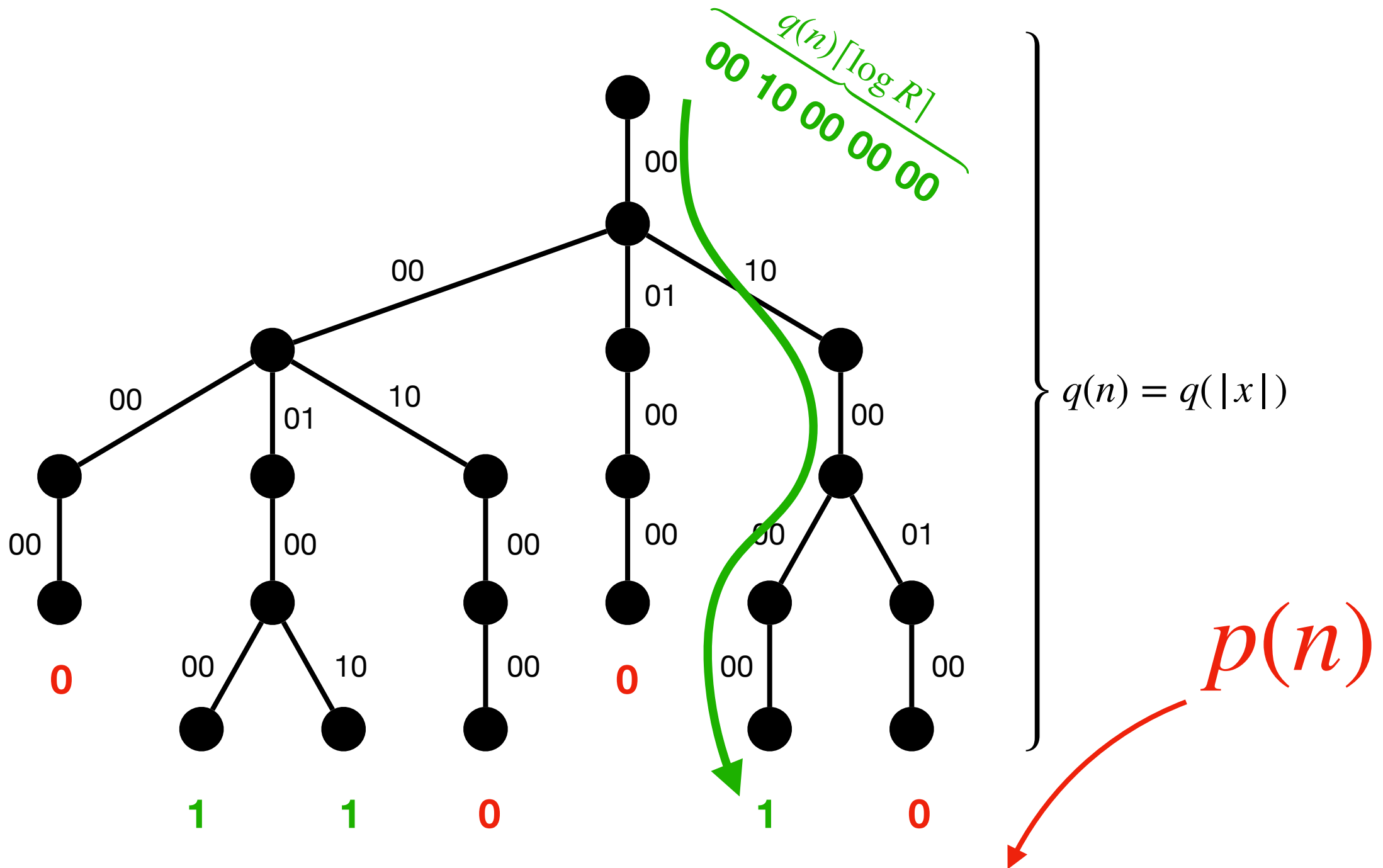


Arbre de calcul de N sur x



Chaque chemin y est décrit par $q(n) \lceil \log R \rceil$ bits

Arbre de calcul de N sur x



Chaque chemin y est décrit par $q(n) \lceil \log R \rceil$ bits

Machine déterministe $M(x, y)$

- **Simuler** la machine non déterministe N sur l'entrée x
- À chaque étape simulée, **choisir la transition indiqué par y**
- Comme on connaît le chemin à simuler, on peut faire ça en **temps polynomial** (déterministe)
- La machine M accepte **ssi la machine N a un chemin acceptant**

Première implication 🍌

- Le langage L est reconnu en temps polynomiale par une machine de Turing **non déterministe** N



- Il existe une machine de Turing **déterministe** M qui fonctionne en temps polynomial et un polynôme $p(n)$ tels que

$$x \in L \text{ ssi } \exists y \in \{0,1\}^{p(n)} M(x, y) \text{ accepte}$$

Deuxième implication 🤔

- Le langage L est reconnu en temps polynomiale par une machine de Turing **non déterministe** N



- Il existe une machine de Turing **déterministe** M qui fonctionne en temps polynomial et un polynôme $p(n)$ tels que

$$x \in L \text{ ssi } \exists y \in \{0,1\}^{p(n)} M(x, y) \text{ accepte}$$

Machine non déterministe $N(x)$

- **Deviner** un certificat $y \in \{0,1\}^{p(n)}$
- **Simuler** $M(x, y)$ et accepter ssi cette machine accepte
- Si M fonctionne et temps $q(n)$, sa simulation prend temps

$$\text{divination} \xrightarrow{p(n)} + \xrightarrow{q(n + p(n))} \text{simulation de } M \text{ sur } (x, y) \text{ vu que } |(x, y)| = n + p(n)$$

Deuxième implication 🍑

- Le langage L est reconnu en temps polynomiale par une machine de Turing **non déterministe** N



- Il existe une machine de Turing **déterministe** M qui fonctionne en temps polynomial et un polynôme $p(n)$ tels que

$$x \in L \text{ ssi } \exists y \in \{0,1\}^{p(n)} M(x, y) \text{ accepte}$$

Algorithmes non déterministes

```
fonction hamiltonien( $V, E$ )  
   $n := |V|$   
   $perm := \text{tableau}(n)$   
  pour  $i := 0$  à  $n - 1$  faire  
     $perm[i] := \text{devine}(0, \dots, n - 1)$   
  si  $perm$  contient des sommets répétés alors  
    rejeter  
  si  $perm$  ne contient pas tous les sommets alors  
    rejeter  
  pour  $i := 0$  à  $n - 1$  faire  
    si  $(perm[i], perm[(i + 1) \bmod n]) \notin E$  alors  
      rejeter  
  accepter  
fin
```

Vérificateurs déterministes



```
fonction vérificateur-hamiltonien( $V$ ,  $E$ , perm)  
   $n := |V|$   
  si perm contient des sommets répétés alors  
    rejeter  
  si perm ne contient pas tous les sommets alors  
    rejeter  
  pour  $i := 0$  à  $n - 1$  faire  
    si ( $perm[i], perm[(i + 1) \bmod n]$ )  $\notin E$  alors  
      rejeter  
  accepter  
fin
```

P vs NP

- **P** est la classe des problèmes faciles à résoudre
- **NP** est la classe des problèmes avec des solutions faciles à vérifier
- Facile à résoudre implique facile à vérifier, donc **$P \subseteq NP$**
- On pense que facile à vérifier n'implique nécessairement pas facile à résoudre, donc **$P \neq NP$**
- Mais ça reste un problème ouvert, d'où le 1000000 \$

Proposition 2-BA (p. 62)

Caractérisation universelle de **coNP**

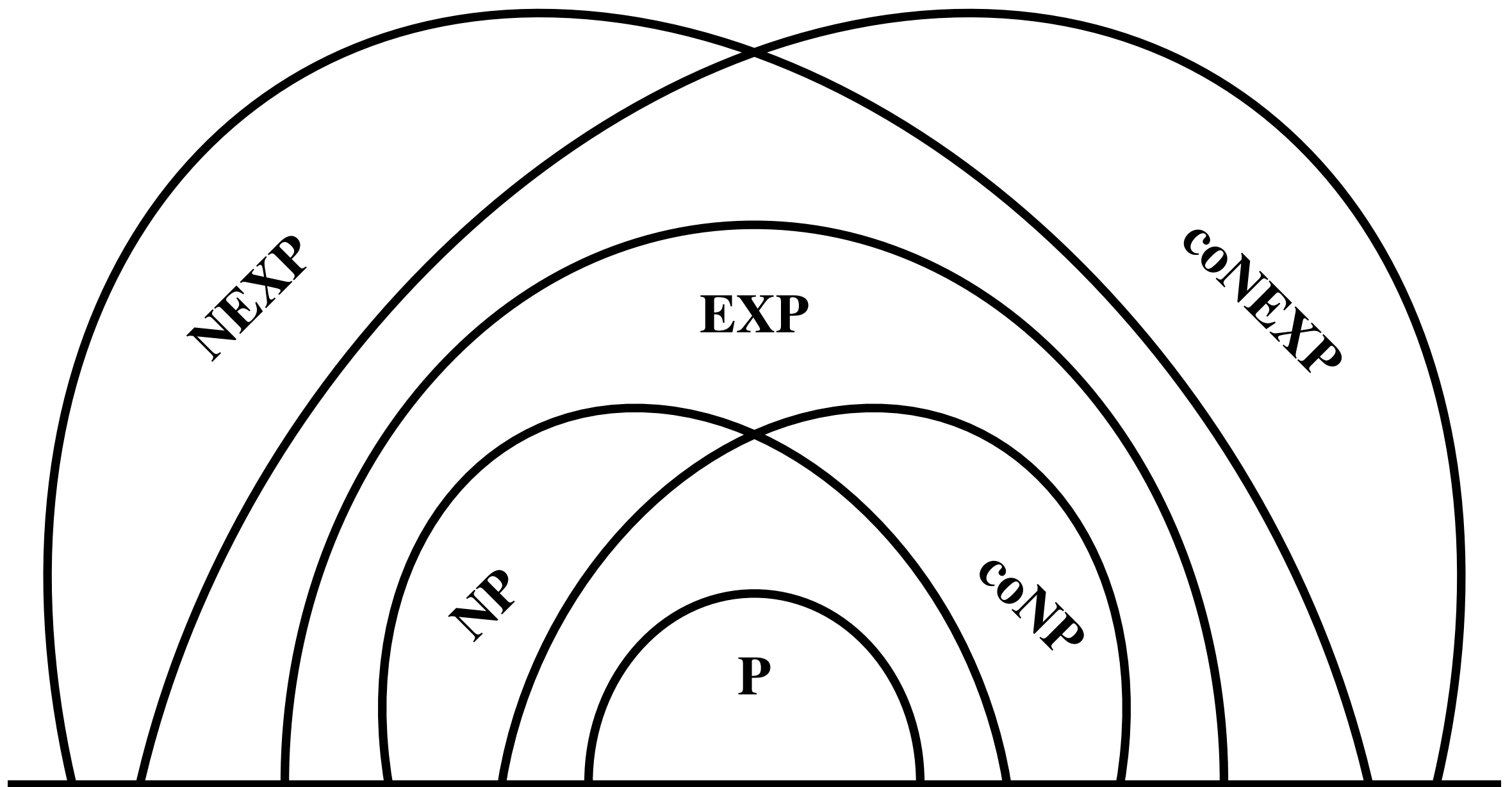
Un langage L appartient à **coNP** ssi il existe une machine de Turing **déterministe** M qui fonctionne en temps polynomial et un polynôme $p(n)$ tels que

$$x \in L \text{ ssi } \forall y \in \{0,1\}^{p(n)} M(x, y) \text{ accepte}$$

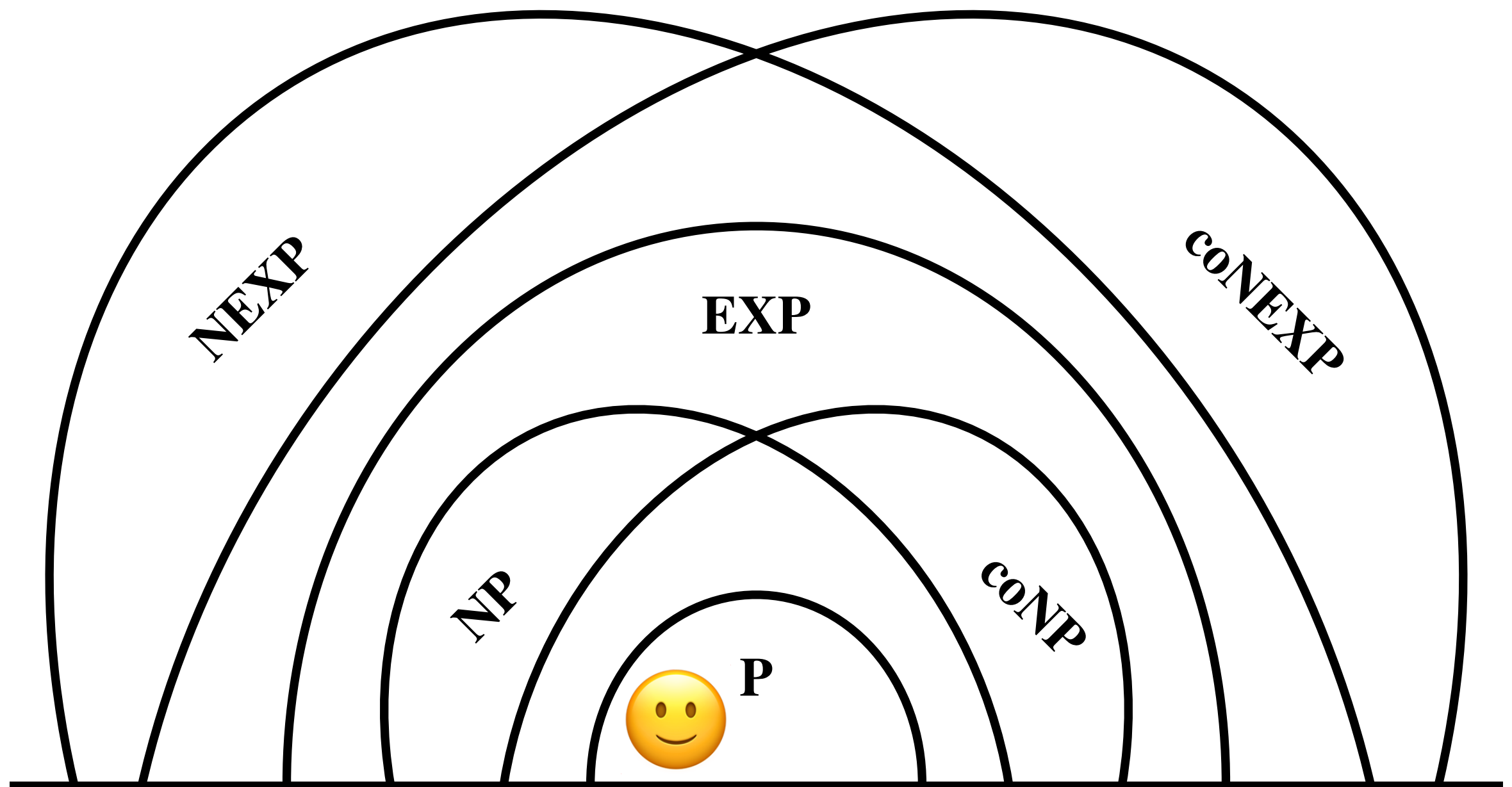


**Réductions,
ou comment classer les
problèmes par difficulté**

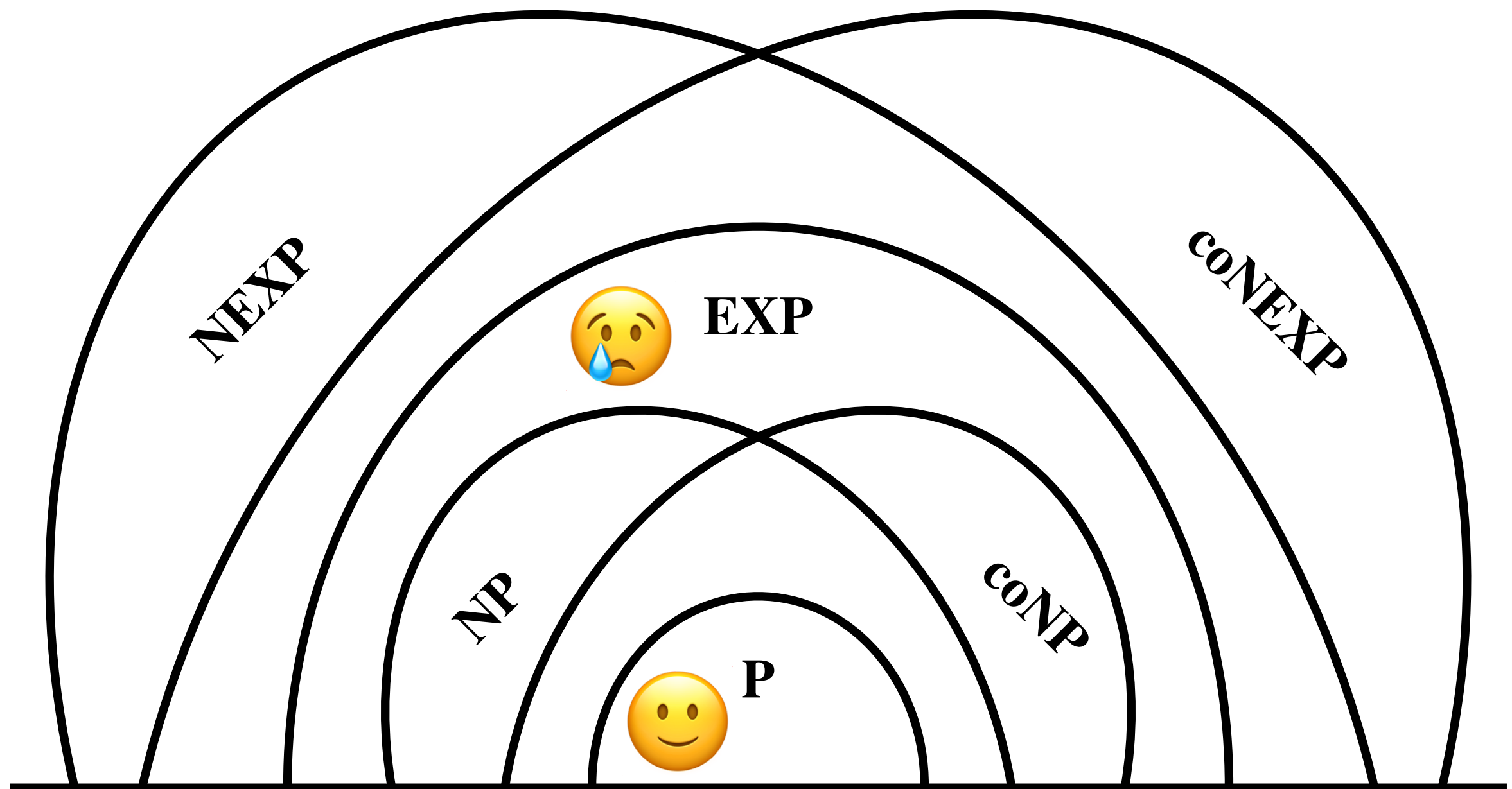
Hiérarchie des classes de complexité



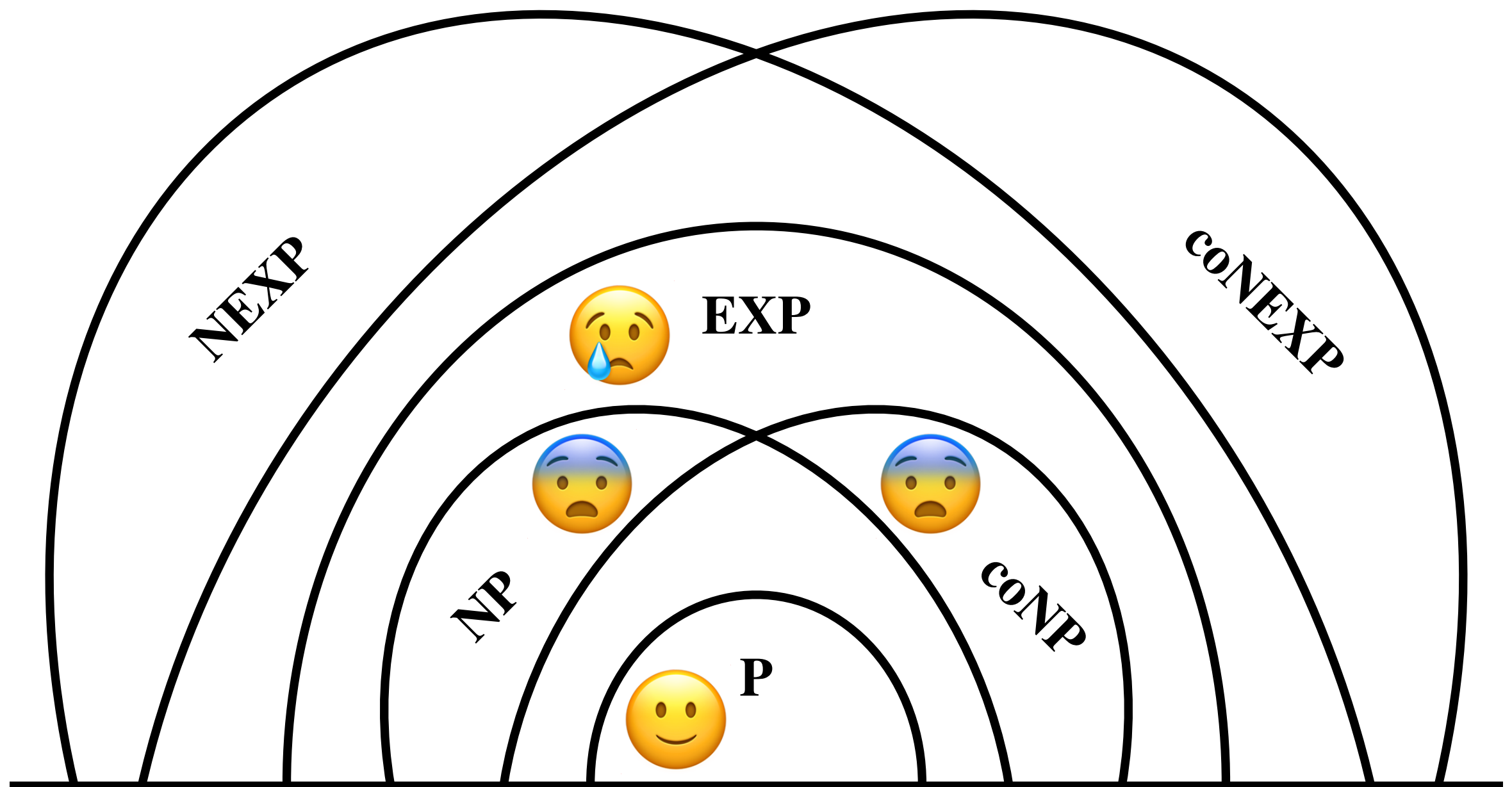
Hiérarchie des classes de complexité



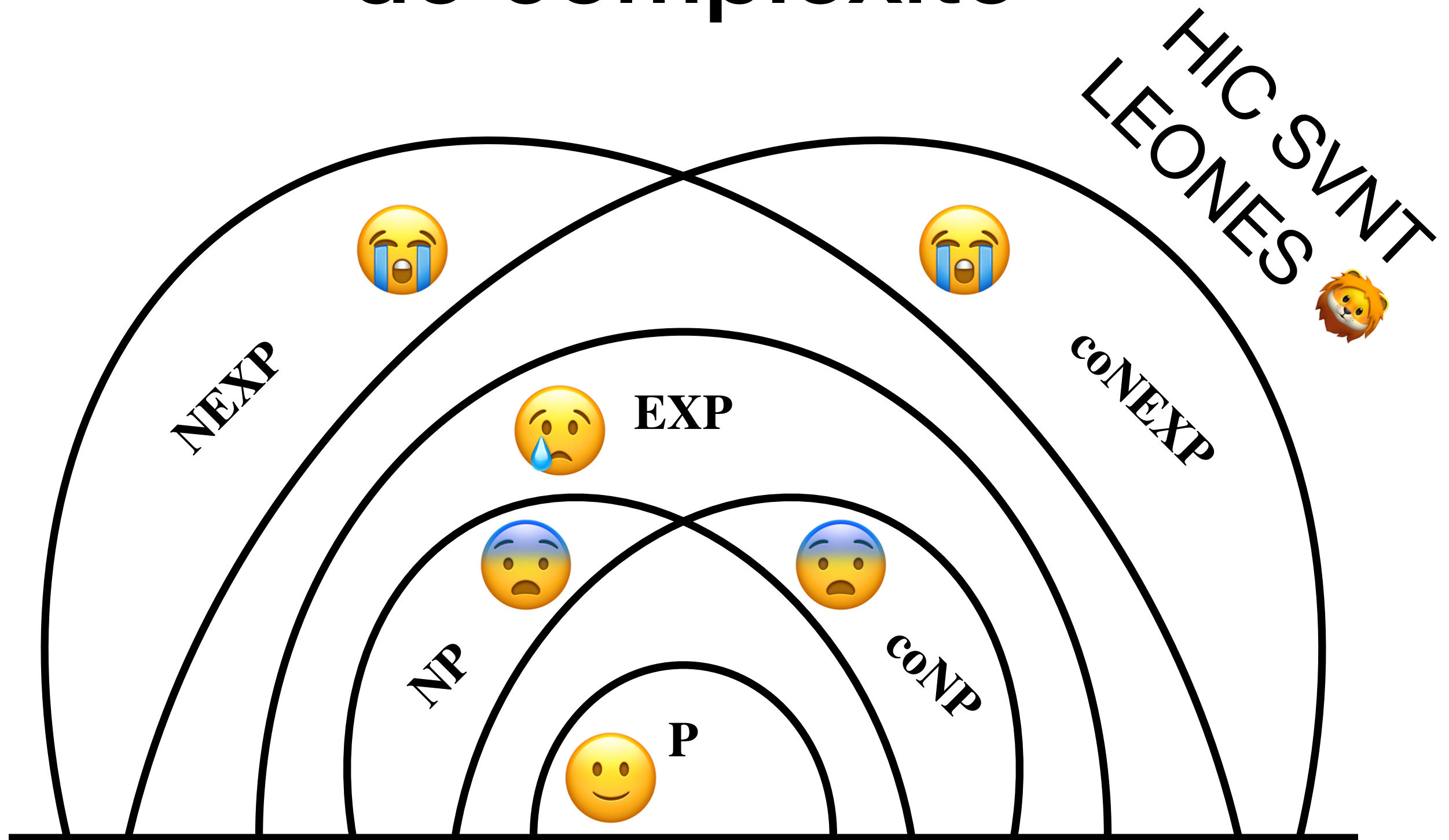
Hiérarchie des classes de complexité



Hiérarchie des classes de complexité



Hiérarchie des classes de complexité



Ordre de difficulté des problèmes

- On veut **classer les problèmes** (et pas les algorithmes !)
en ordre de difficulté
- C'est quoi la **difficulté** ?
- On voudrait que ce soit liée **de quelque façon**
au temps de calcul

Ordre de difficulté des problèmes

- Par exemple, $L_1 \leq L_2$ si le meilleur algorithme (ou machine de Turing) pour L_1 est plus rapide du meilleur algorithme pour L_2
- Problème : souvent on ne sait pas quel est l'algorithme meilleur pour un problème !

Alternative : réductions

- On dit que L_1 est plus facile de L_2 si on peut reconnaître facilement L_1 quand on a un moyen de reconnaître L_2
- Si c'est le cas, résoudre L_2 nous permet de résoudre L_1 , donc L_1 n'est pas plus difficile que L_2
- Dit autrement, pour résoudre L_1 on se ramène à L_2 ...
- ...ou, plus formellement, L_1 se réduit à L_2

Définition 3-A (p. 64)

Réductions (many-one) polynomiales

- Une **réduction (many-one) en temps polynomial** d'un problème L_1 (sur l'alphabet Σ_1) à un problème L_2 (sur l'alphabet Σ_2) est une fonction $f: \Sigma_1^* \rightarrow \Sigma_2^*$ calculable en temps polynomial telle que

$$\forall x \in \Sigma_1^* \quad x \in L_1 \iff f(x) \in L_2$$

- Si une telle f existe, on dit que **L_1 se réduit à L_2** (via f) et on notera **$L_1 \leq_m^P L_2$** (ou parfois, en bref, **$L_1 \leq L_2$**)

Lemme 3-G (p. 66)

\leq_m^P (pré-)ordonne les langages

- \leq est réflexive : $L_1 \leq L_1$ via l'identité $f: \Sigma_1^* \rightarrow \Sigma_1^*$, qui évidemment est calculable en temps polynomial
- \leq est transitive : soit $L_1 \leq L_2$ via $f: \Sigma_1^* \rightarrow \Sigma_2^*$ et $L_2 \leq L_3$ via $g: \Sigma_2^* \rightarrow \Sigma_3^*$
- alors $x \in L_1 \iff f(x) \in L_2$ et $f(x) \in L_2 \iff g(f(x)) \in L_3$, donc $x \in L_1 \iff g(f(x)) \in L_3$
- si f est calculable en temps polynomial $p(n)$ alors $f(x) \leq p(|x|) = p(n)$
- si g est calculable en temps polynomial $q(n)$, alors la fonction composée $g \circ f$ est calculable en temps polynomial $O(p(n) + q(p(n)))$

Definition 3-H (p. 66)

Problèmes équivalents

- Si $L_1 \leq_m^P L_2$ et $L_2 \leq_m^P L_1$ alors on écrit $L_1 \equiv_m^P L_2$
(parfois, en bref, $L_1 \equiv L_2$)
- On dit que les problèmes L_1 et L_2 sont **équivalents**
pour les reductions (many-one) polynomiales
- Par exemple, on a toujours $L_1 \equiv L_1$

Résoudre L_1 avec une réduction

x

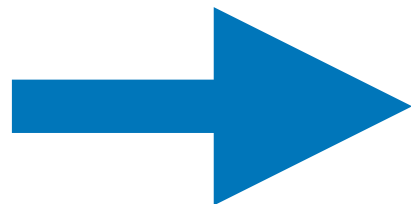


Résoudre L_1 avec une réduction

x



Résoudre L_1 avec une réduction



x



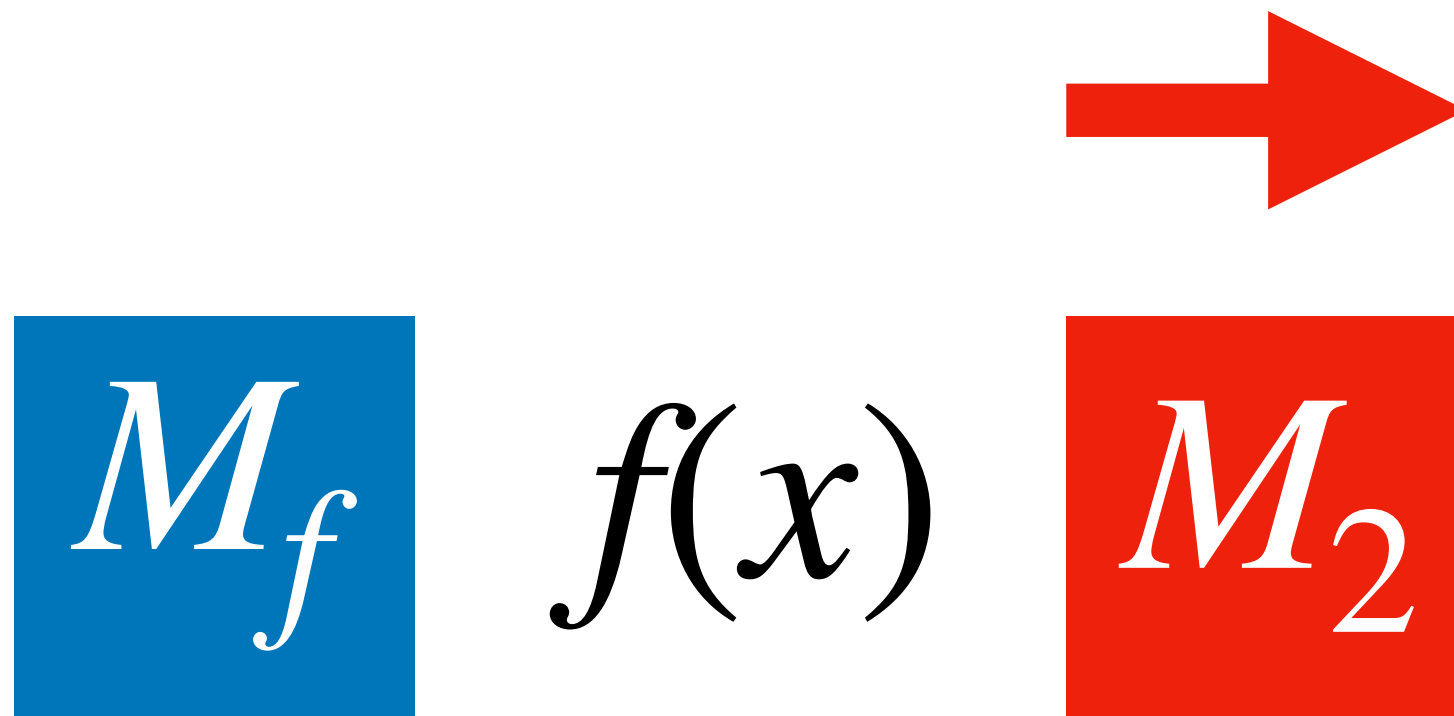
Résoudre L_1 avec une réduction



$f(x)$



Résoudre L_1 avec une réduction



Résoudre L_1 avec une réduction



oui

Résoudre L_1 avec une réduction



oui

non

Résoudre L_1 avec une réduction



oui

non

$$\forall x \in \Sigma_1^* \quad x \in L_1 \iff f(x) \in L_2$$

Résoudre L_1 avec une réduction



Résoudre L_1 avec une réduction


$$M_f$$
$$M_1$$
$$M_2$$

Résoudre L_1 avec une réduction

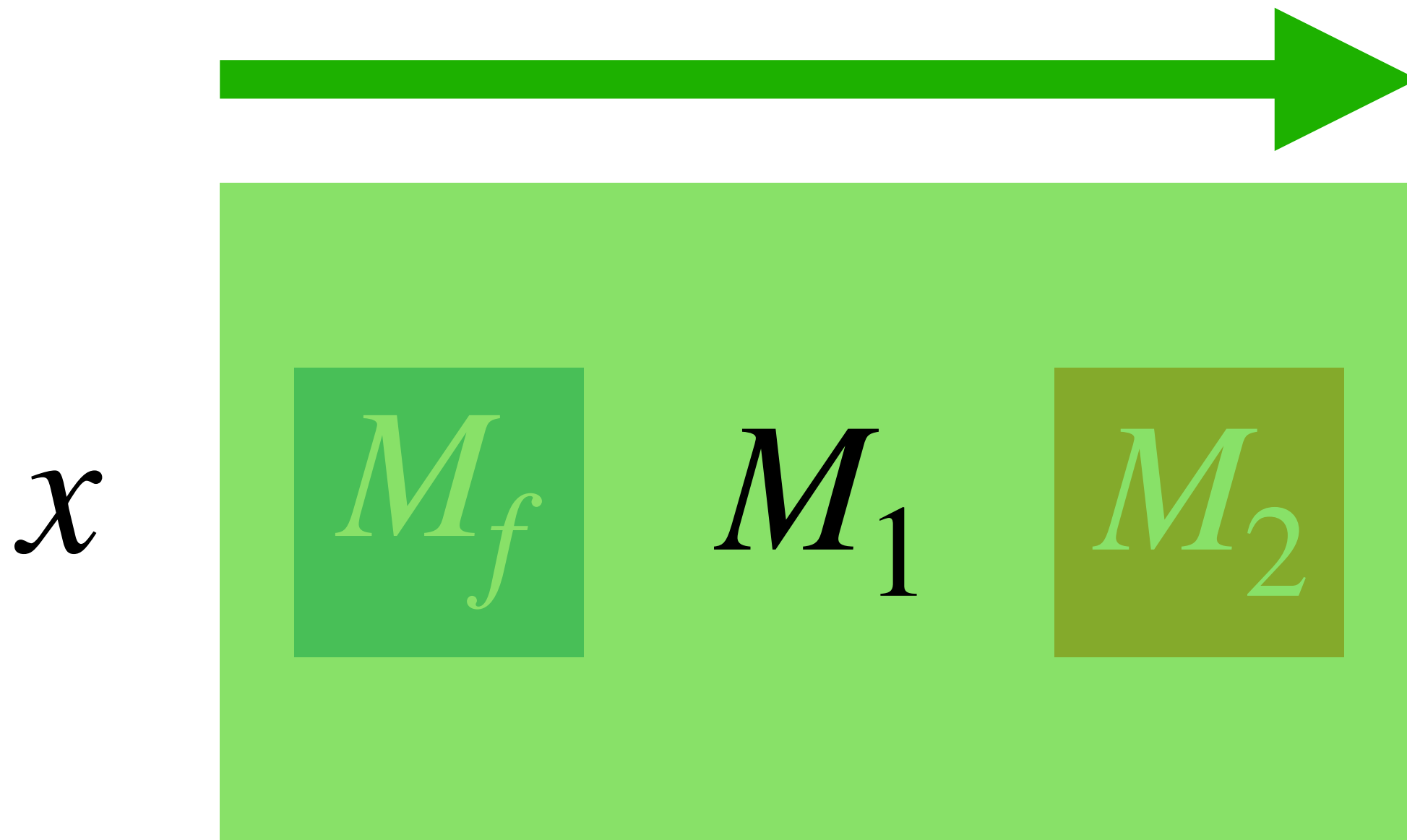
x



M_1



Résoudre L_1 avec une réduction



Résoudre L_1 avec une réduction

M_f

M_1

M_2

oui

Résoudre L_1 avec une réduction


$$M_f$$
$$M_1$$
$$M_2$$

oui

non

Proposition 3-C (p. 64)

P est close pour \leq

Si $L_2 \in \mathbf{P}$ et $L_1 \leq L_2$, alors $L_1 \in \mathbf{P}$

- Soit f la fonction de réduction de L_1 à L_2 en temps polynomial $p(n)$, et soit M_2 une machine déterministe qui reconnaît L_2 en temps polynomial $q(n)$
- Alors le suivant est un algorithme déterministe pour L_1 :
calculer $y = f(x)$ et retourner le résultat de $M_2(y)$
- Ça prend temps $p(n)$ pour calculer f et $q(p(n))$ pour simuler M_2 sur $y = f(x)$, donc temps polynomial, donc $L_1 \in \mathbf{P}$

Proposition 3-C (p. 64)

P est close pour \leq

Ça veut dire que si L_2 est efficacement résoluble et $L_1 \leq L_2$, c'est-à-dire que L_1 est plus simple que L_2 , alors L_1 est aussi efficacement résoluble, conformément à l'intuition

Proposition 3-C (p. 64)

NP est close pour \leq

Si $L_2 \in \mathbf{NP}$ et $L_1 \leq L_2$, alors $L_1 \in \mathbf{NP}$

- Soit f la fonction de réduction de L_1 à L_2 en temps polynomial $p(n)$, et soit M_2 une machine **non déterministe** qui reconnaît L_2 en temps polynomial $q(n)$
- Alors le suivant est un algorithme **non déterministe** pour L_1 :
calculer $y = f(x)$ et retourner le résultat de $M_2(y)$
- Ça prend temps $p(n)$ pour calculer f et $q(p(n))$ pour simuler M_2 sur $y = f(x)$, donc temps polynomial, donc $L_1 \in \mathbf{NP}$

Proposition 3-C (p. 64)

NP est close pour \leq

Ça veut dire que si L_2 est efficacement **verifiable** et $L_1 \leq L_2$, c'est-à-dire que L_1 est plus simple que L_2 , alors L_1 est aussi efficacement **verifiable**, conformément à l'intuition

Exemple 3-1 (p. 66)

Une réduction

Problème CLIQUE

- **Entrée** : un graphe non orienté G et un entier k
- **Question** : G a-t-il une clique de taille k ?

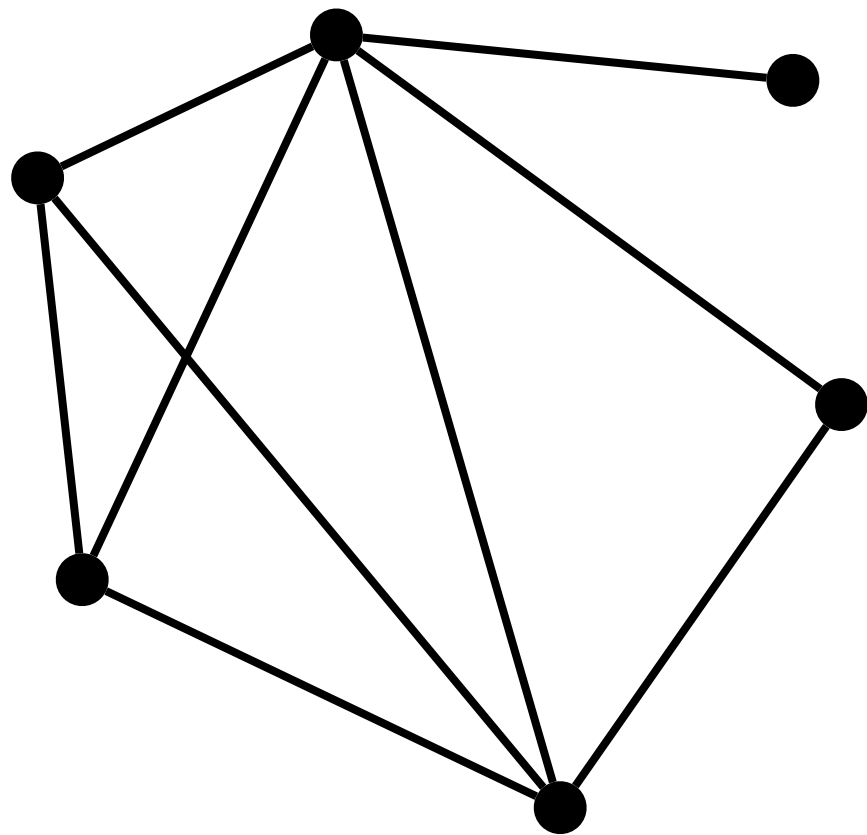
Problème

ENSEMBLE-INDÉPENDANT

- **Entrée** : un graphe non orienté G et un entier k
- **Question** : existe-t-il un ensemble de k sommets indépendants dans G , c'est-à-dire tous non reliés deux à deux ?

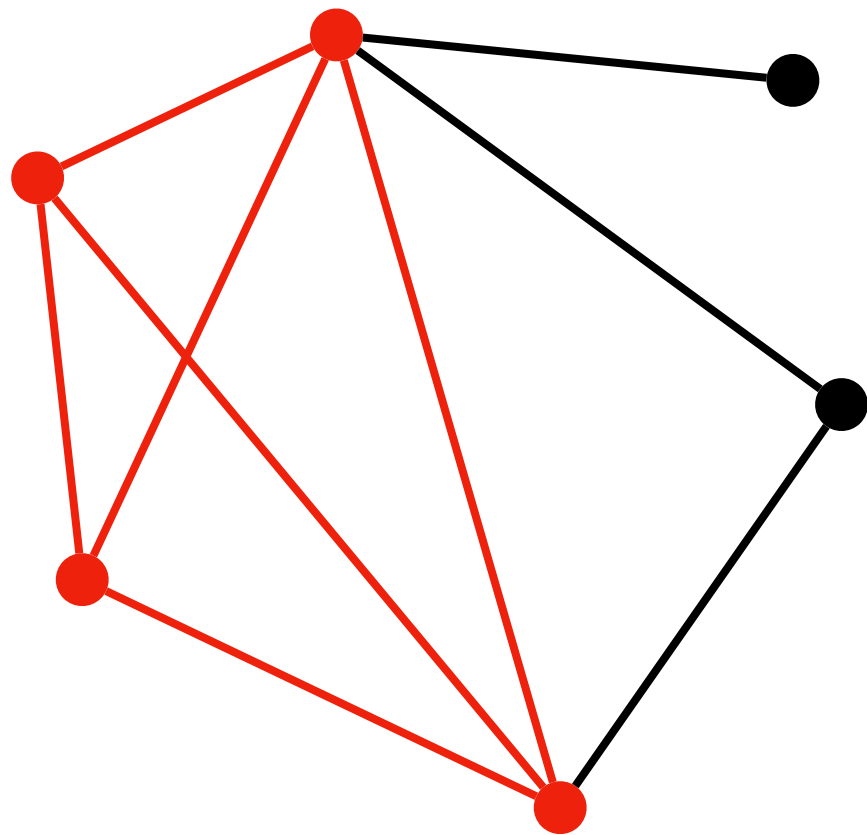
CLIQUE vs ENS-INDÉP

$$G = (V, E)$$



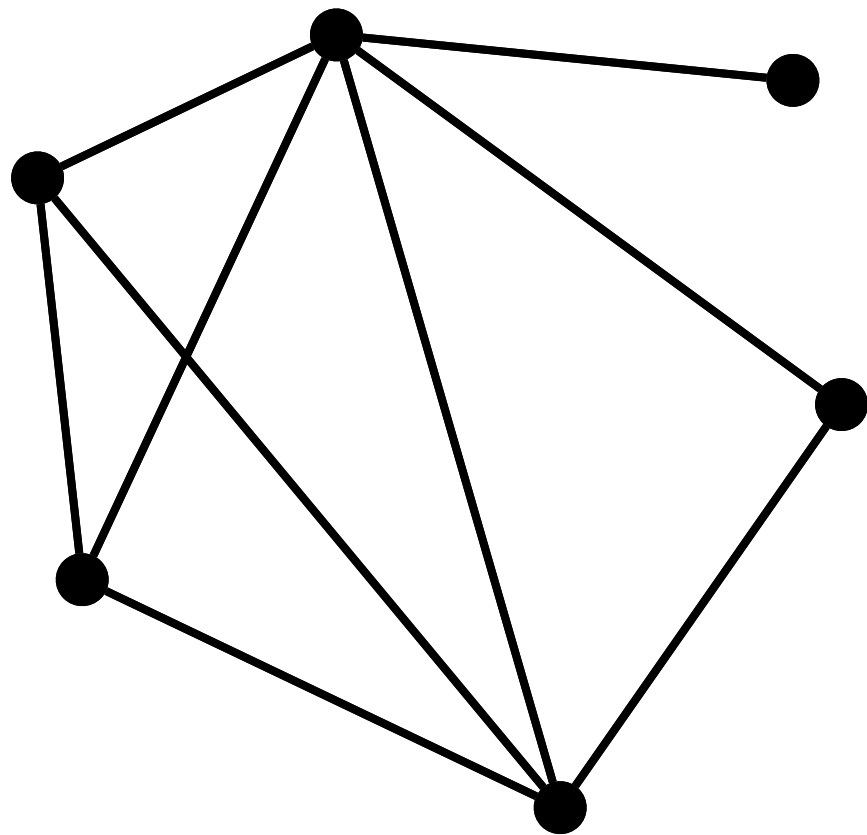
CLIQUE vs ENS-INDÉP

$$G = (V, E)$$



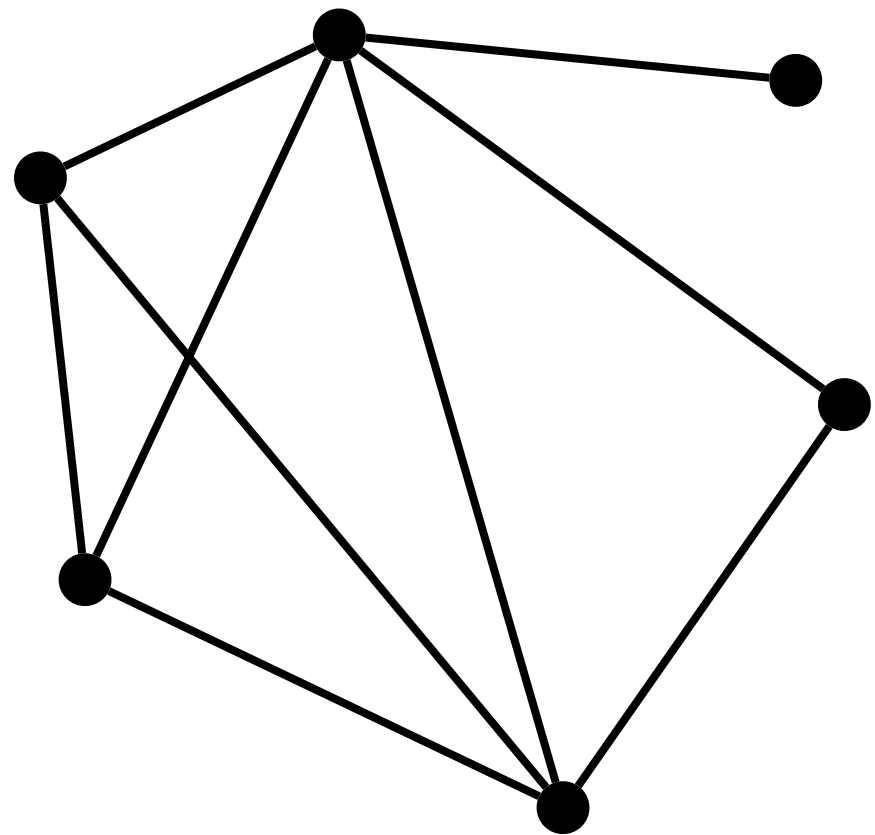
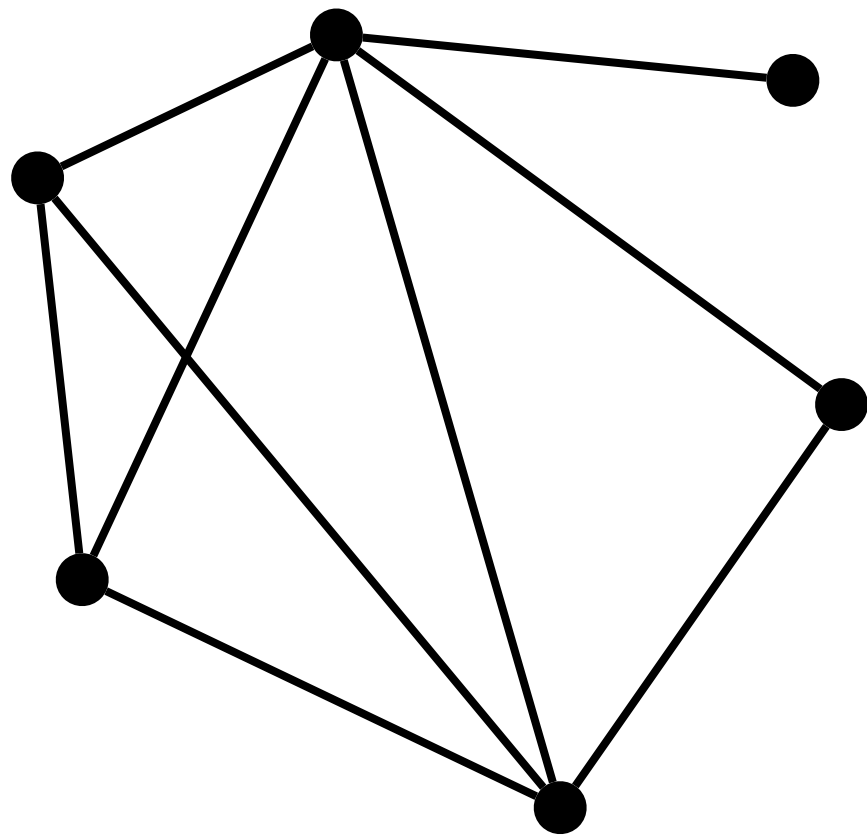
CLIQUE vs ENS-INDÉP

$$G = (V, E)$$



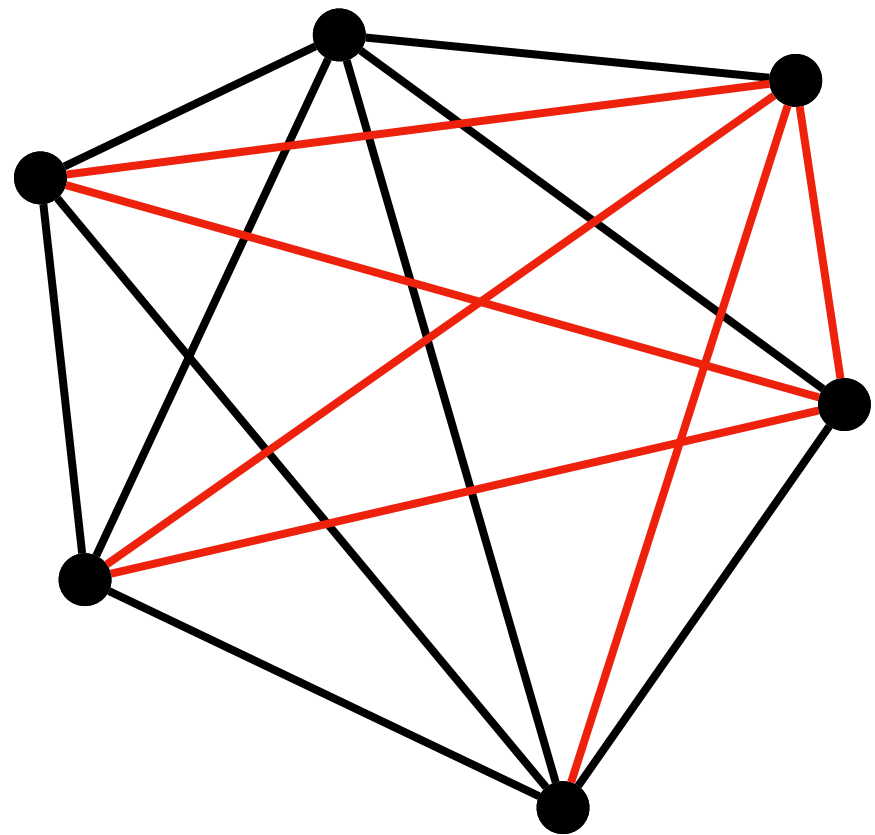
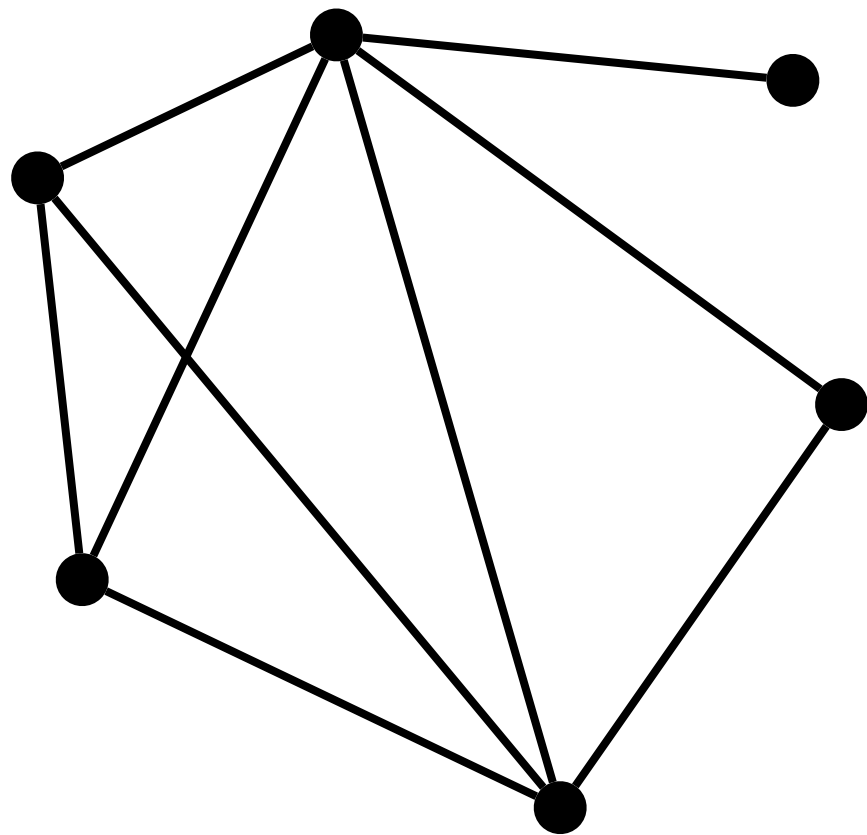
CLIQUE vs ENS-INDÉP

$$G = (V, E)$$



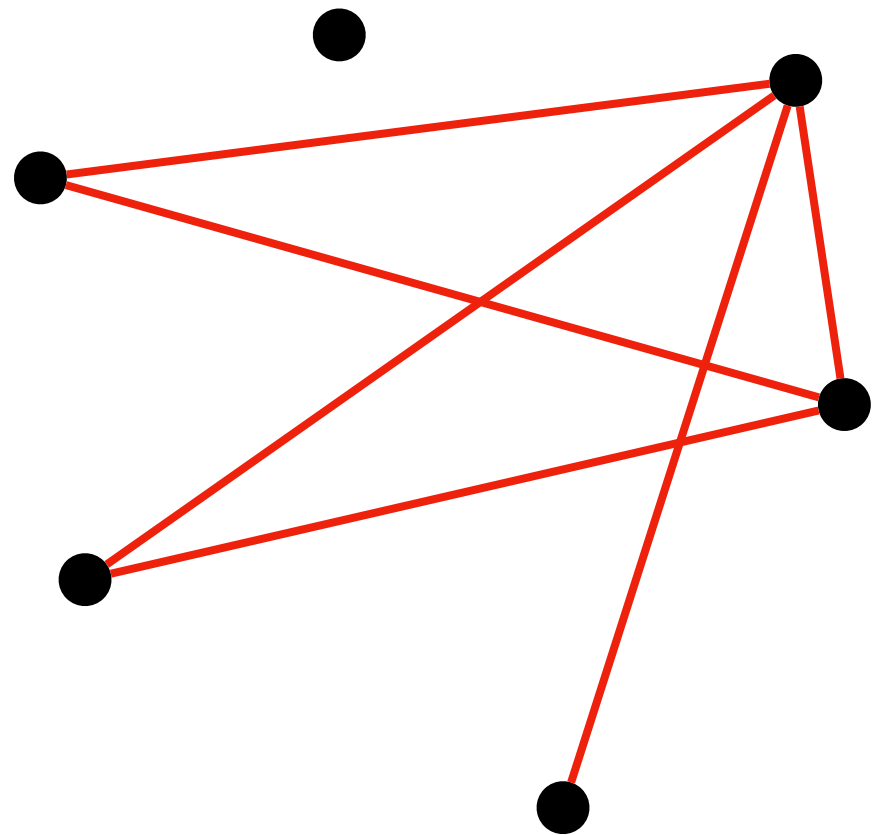
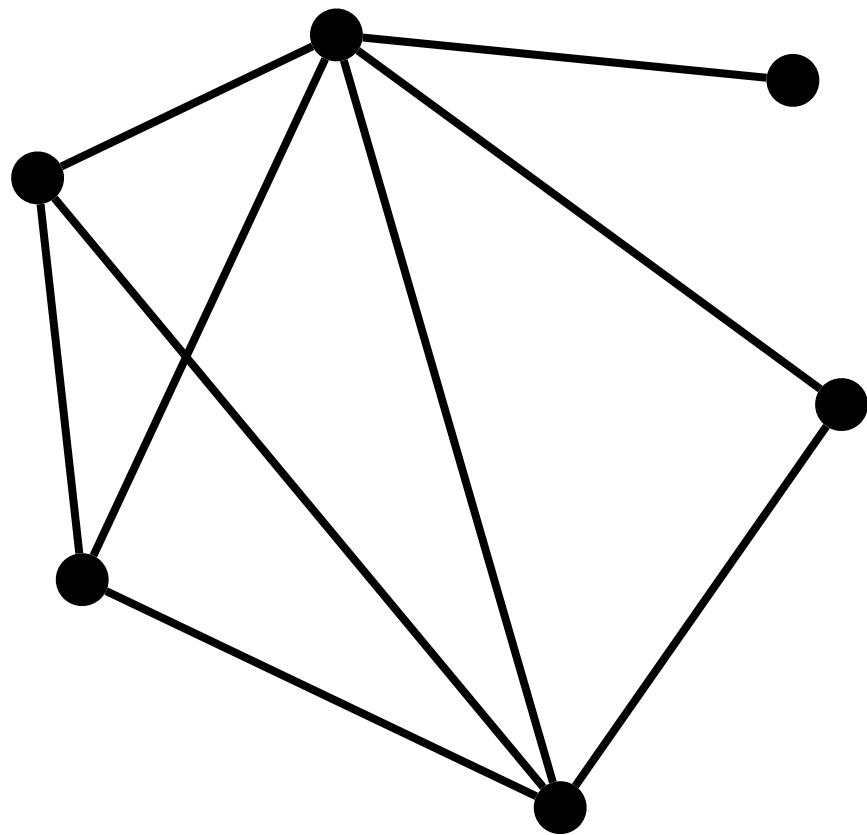
CLIQUE vs ENS-INDÉP

$$G = (V, E)$$



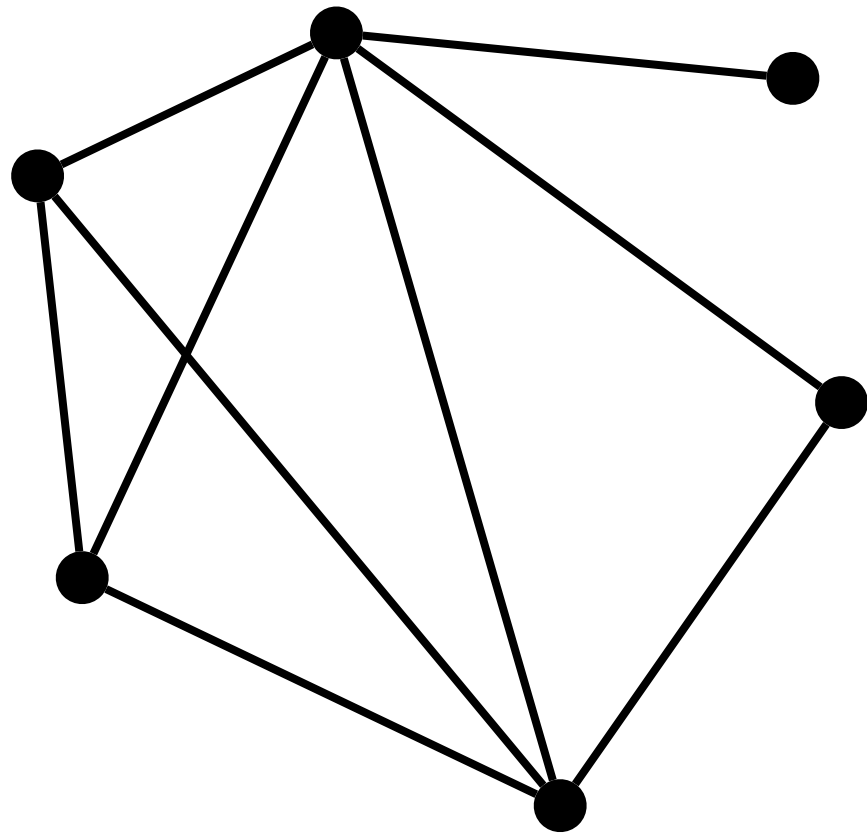
CLIQUE vs ENS-INDÉP

$$G = (V, E)$$

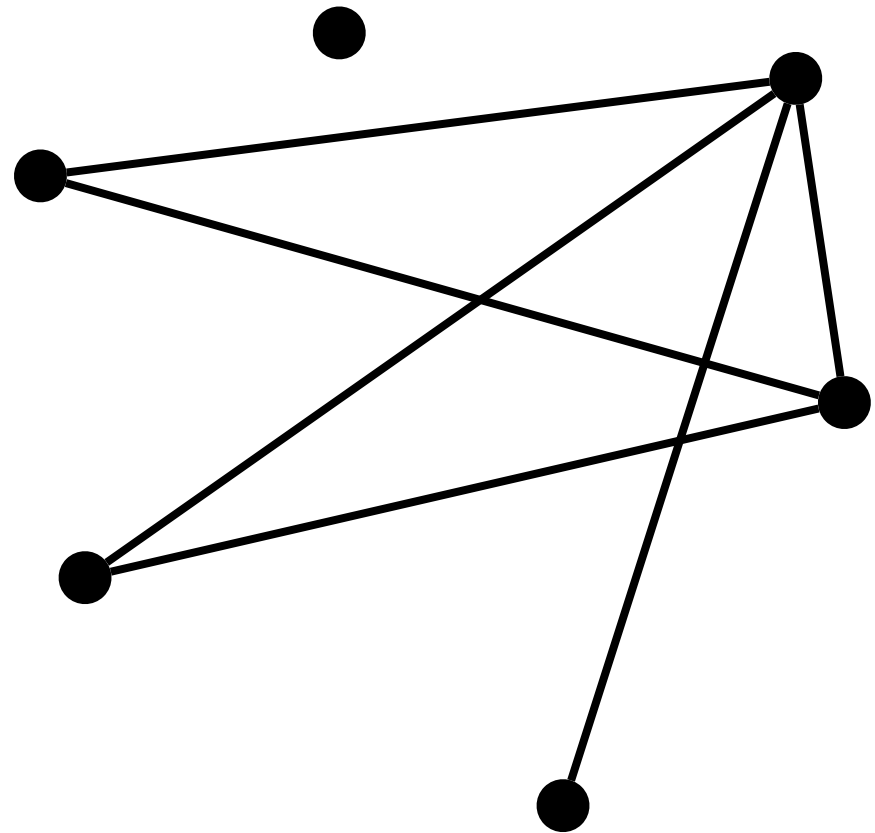


CLIQUE vs ENS-INDÉP

$$G = (V, E)$$

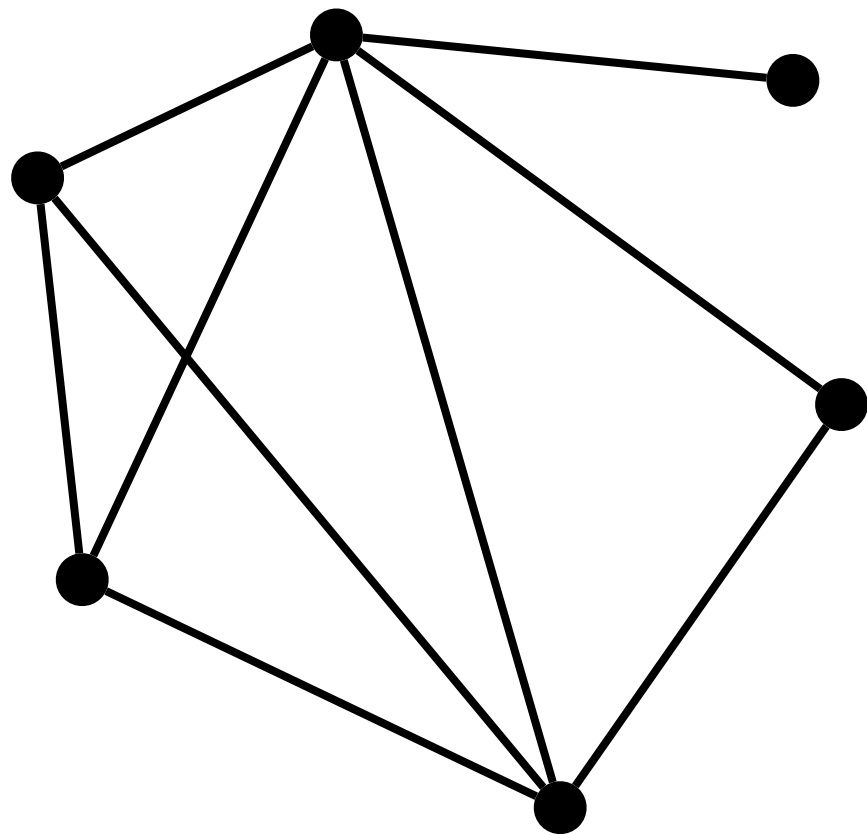


$$\overline{G} = (V, V^2 - E)$$



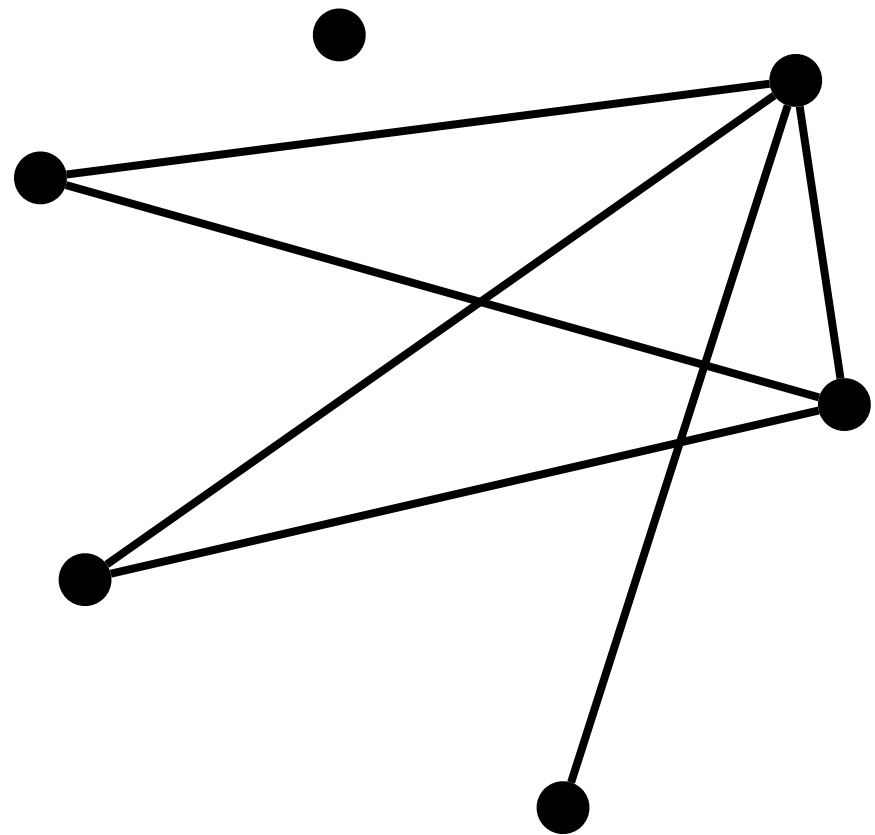
CLIQUE vs ENS-INDÉP

$$G = (V, E)$$



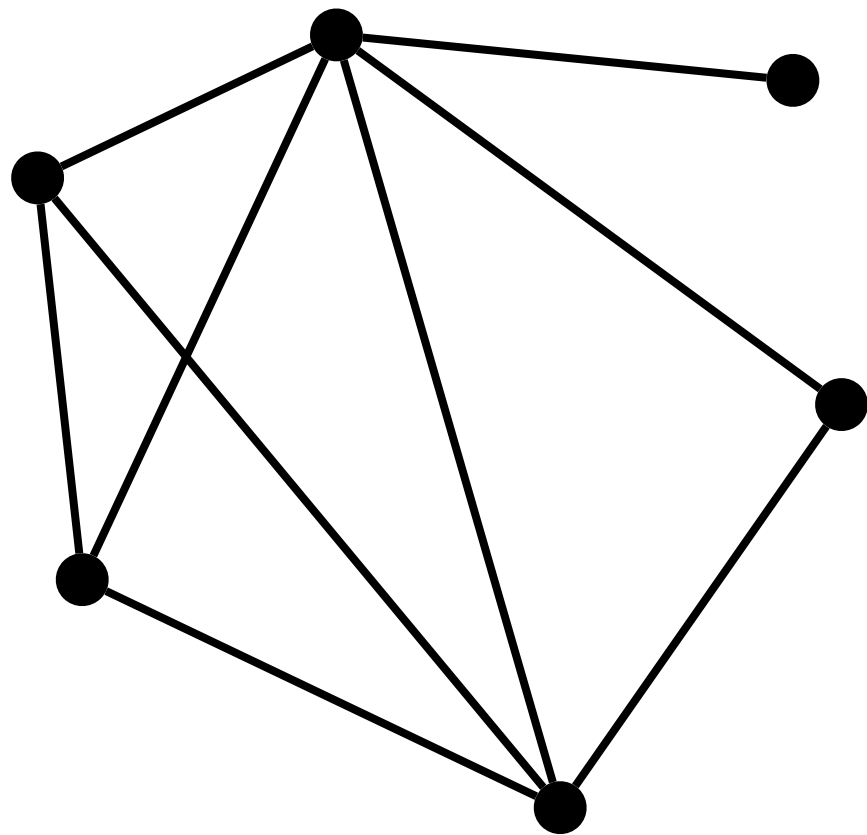
a une clique de taille k

$$\overline{G} = (V, V^2 - E)$$

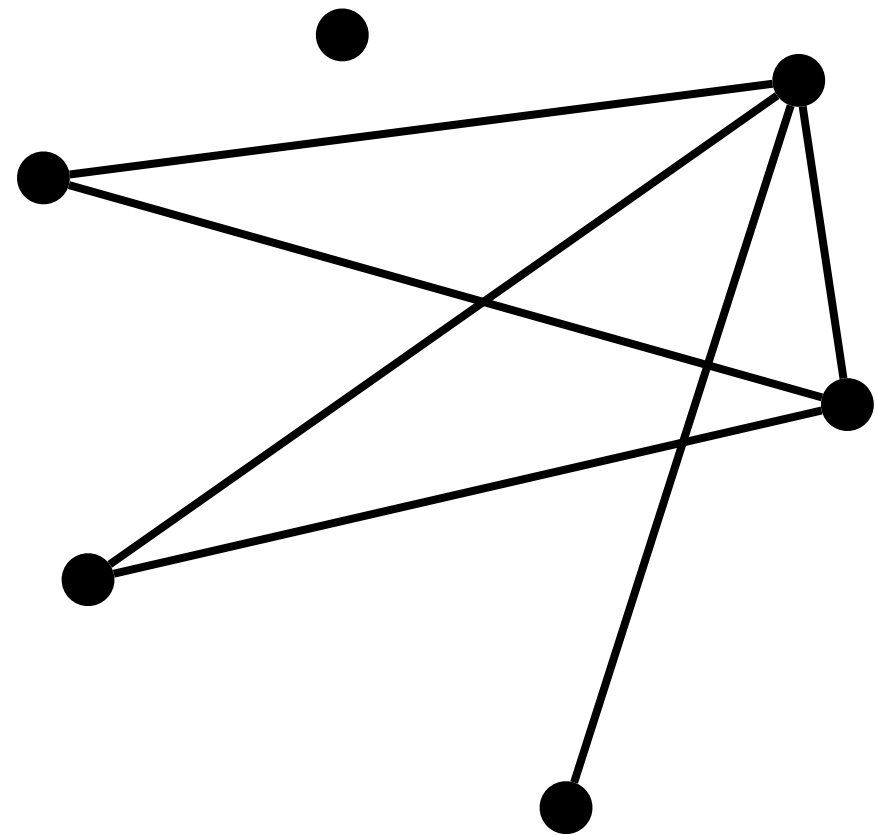


CLIQUE vs ENS-INDÉP

$$G = (V, E)$$



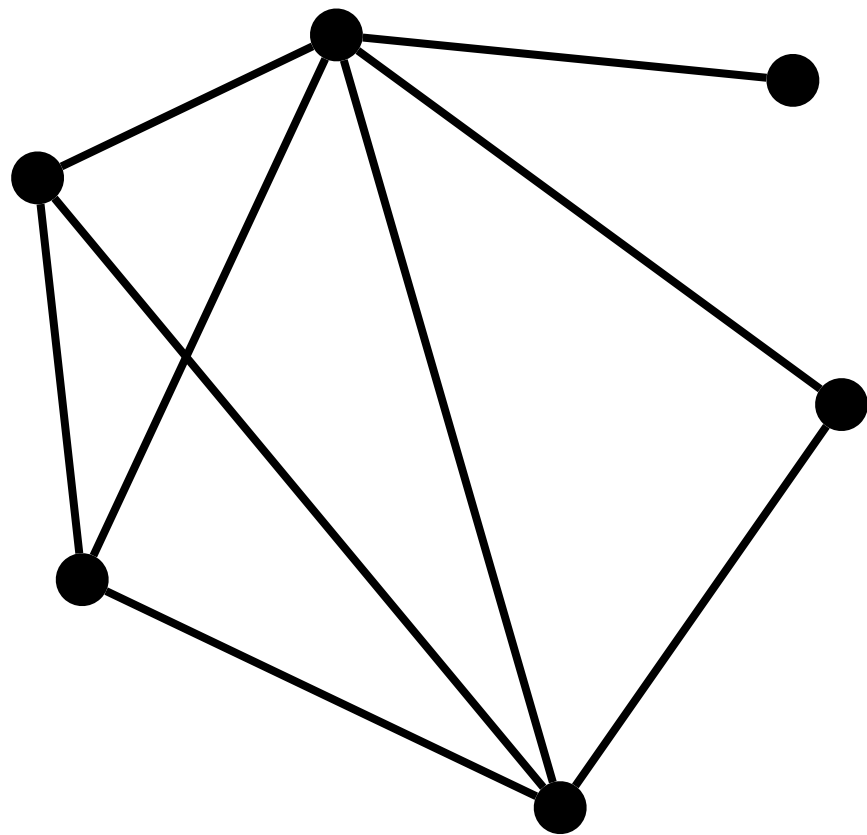
$$\overline{G} = (V, V^2 - E)$$



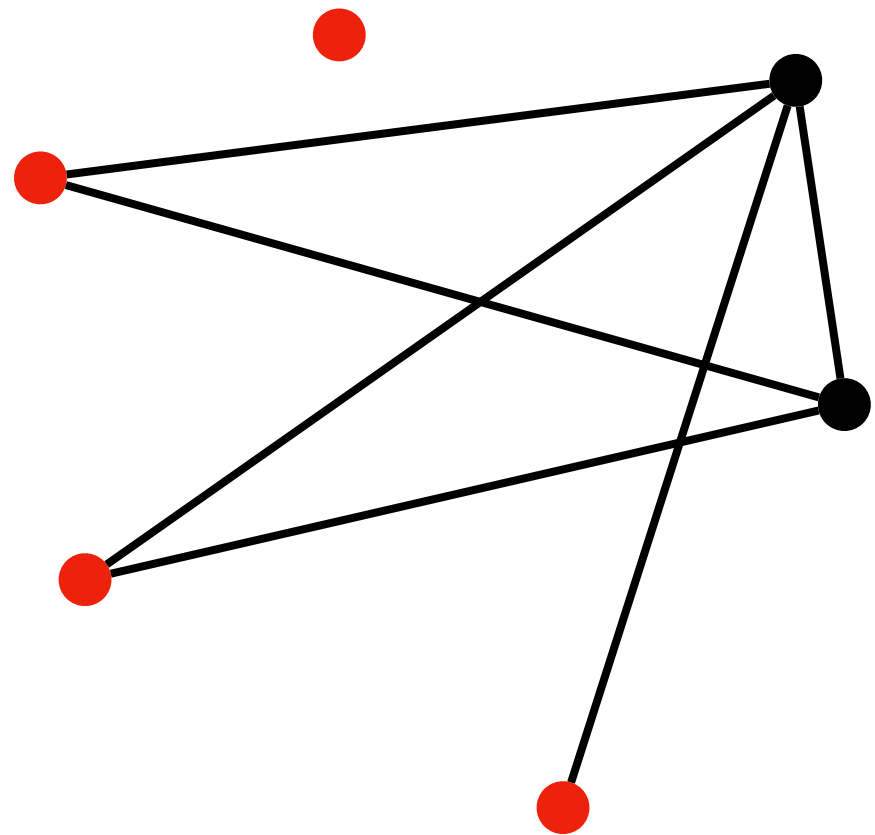
a une clique de taille $k \iff$ a un ens. indép. de taille k

CLIQUE vs ENS-INDÉP

$$G = (V, E)$$



$$\overline{G} = (V, V^2 - E)$$



a une clique de taille $k \iff$ a un ens. indép. de taille k

CLIQUE \leq ENS-INDÉP

- Supposons que CLIQUE et ENS-INDÉP sont définis sur le même alphabet Σ (raisonnable, c'est toujours des graphes !)
- La fonction $f: \Sigma^* \rightarrow \Sigma^*$ définie par $f(V, E) = (V, V^2 - E)$ est calculable en temps polynomial
- (V, E) a une clique de taille k ssi $f(V, E)$ a un ensemble indépendant de taille k
- Donc $(V, E) \in \text{CLIQUE}$ iff $f(V, E) \in \text{ENS-INDÉP}$

Mais aussi

ENS-INDÉP \leq CLIQUE !

- On utilise la même fonction $f: \Sigma^* \rightarrow \Sigma^*$ définie par $f(V, E) = (V, V^2 - E)$, toujours calculable en temps polynomial*
- (V, E) a un ensemble indépendant de taille k ssi $f(V, E)$ a une clique de taille k
- Donc $(V, E) \in \text{ENS-INDÉP}$ iff $f(V, E) \in \text{CLIQUE}$

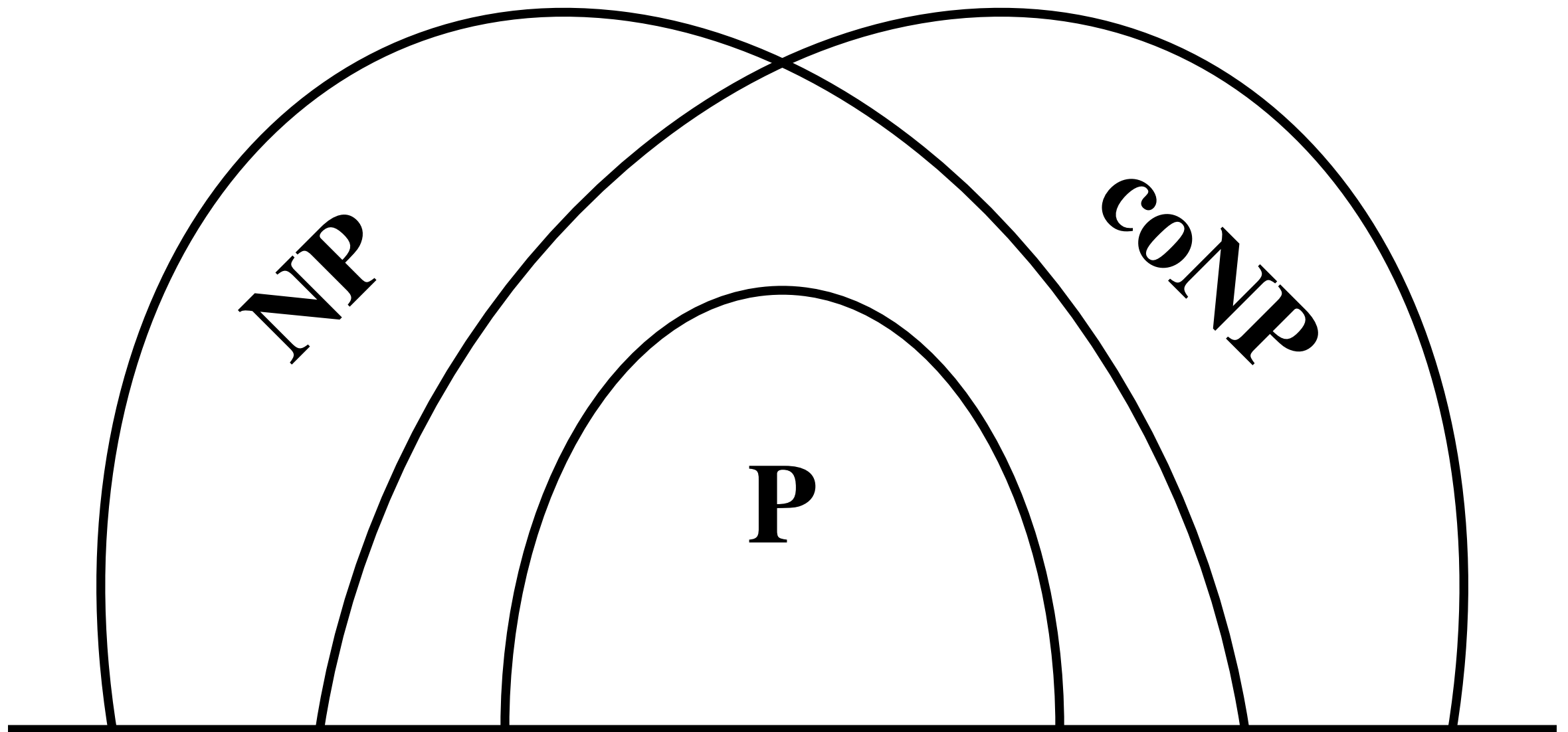
* C'est un cas exceptionnel ! Normalement il faut changer de fonction

ENS-INDÉP \equiv CLIQUE

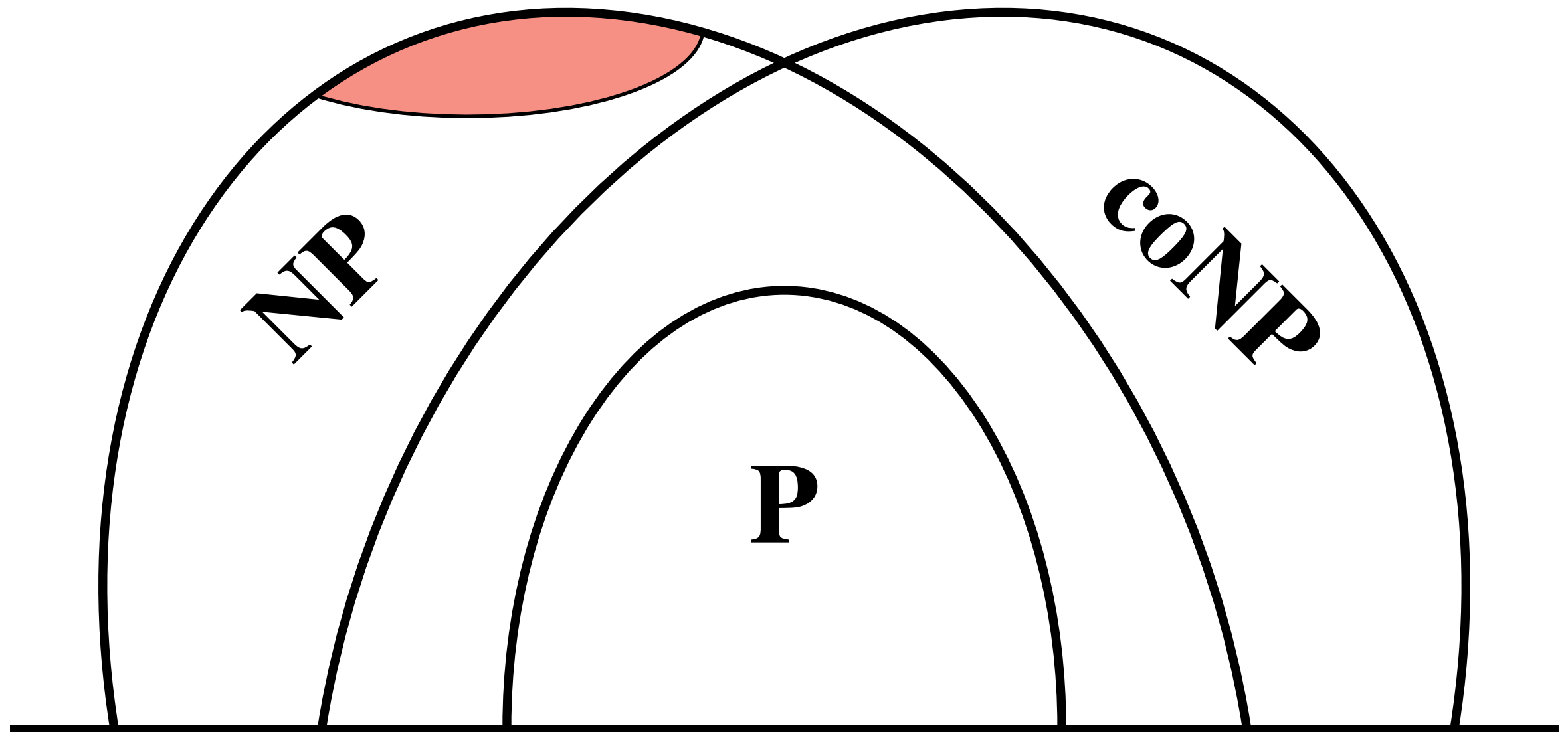
- On sait que CLIQUE \in **NP** (exercice du TD5)
- Puisque ENS-INDÉP \leq CLIQUE, on a ENS-INDÉP \in **NP**, parce que **NP** est clos pour \leq
- Si on découvrait que CLIQUE \in **P**, on aurait aussi ENS-INDÉP \in **P**, parce que **P** est aussi clos par \leq
- Même chose si on découvrait que ENS-INDÉP \in **P** : ça impliquerait CLIQUE \in **P**

**Complétude,
ou les problèmes les plus
difficiles du monde**

Hiérarchie du « possible »

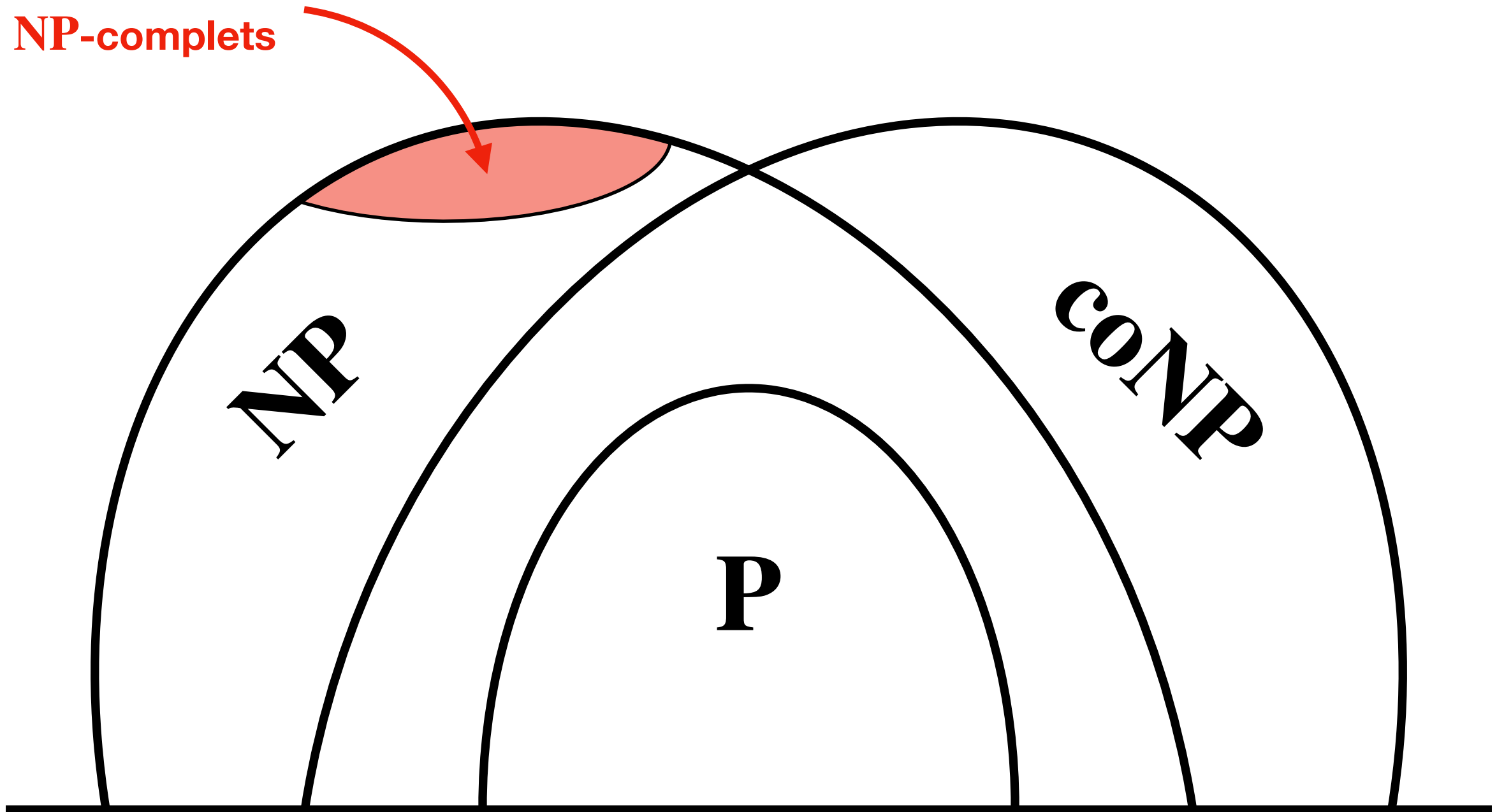


Hiérarchie du « possible »

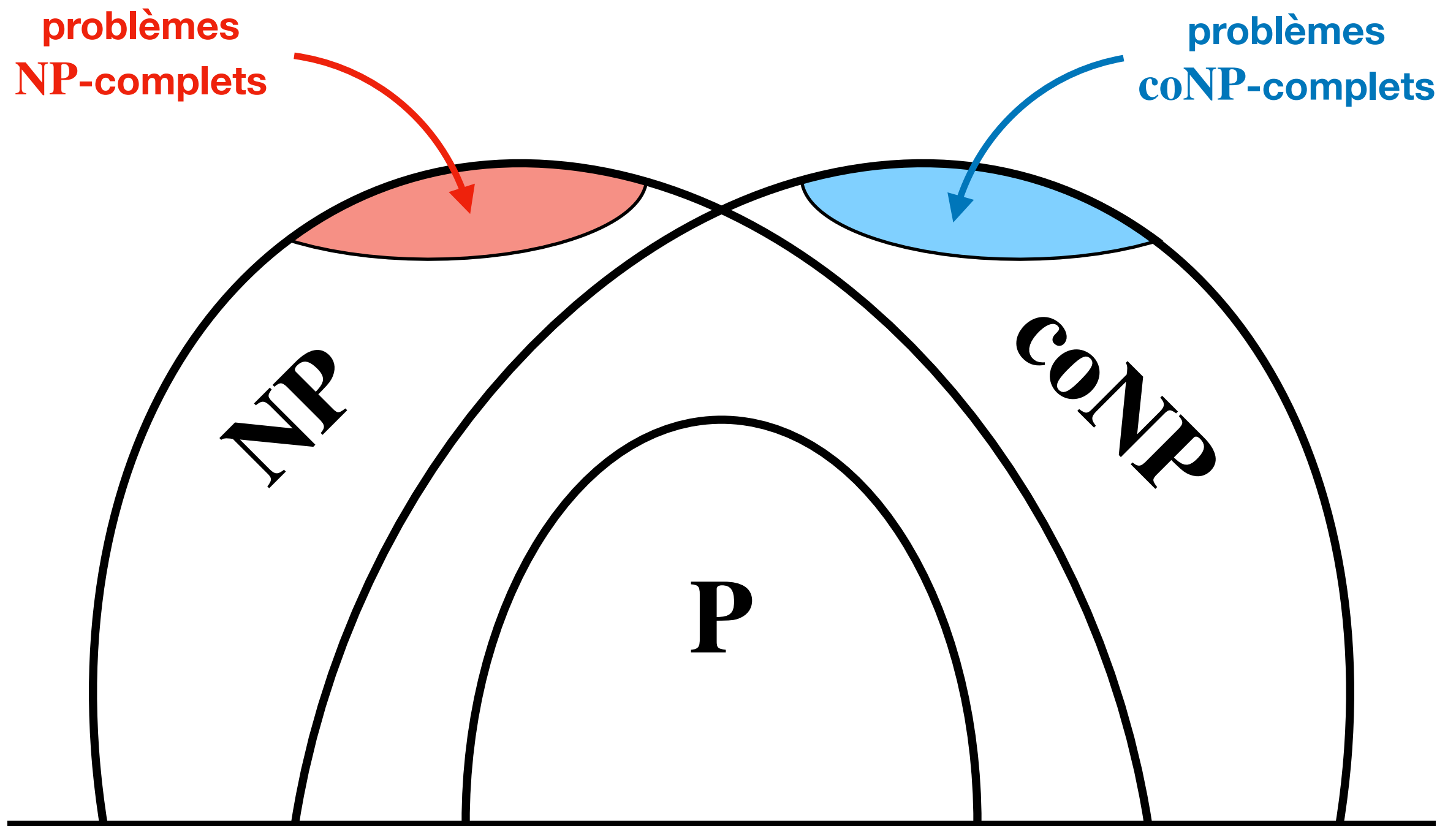


Hiérarchie du « possible »

problèmes
NP-complets



Hiérarchie du « possible »



Definition 3-J (p. 67)

Difficulté et complétude

Soit L un problème et \mathcal{C} une classe de complexité

- On dit que L est \mathcal{C} -difficile (ou \mathcal{C} -dur) si pour tout problème $L' \in \mathcal{C}$ on a $L' \leq L$
- On dit que L est \mathcal{C} -complet s'il est \mathcal{C} -difficile et en plus on a $L \in \mathcal{C}$

Difficulté et complétude

- Les définitions sont très fortes : il faut que **tous** les problèmes de \mathcal{C} se réduisent à L !
- À priori c'est n'est même pas évident qu'il existent des problèmes durs ou complets pour une classe...

P a (beaucoup de) problèmes complets

- Tout problème $L \in \mathbf{P}$ non trivial (c'est-à-dire, $L \neq \emptyset$ et $L \neq \Sigma^*$) est **P-complet** pour les reductions en temps polynomial
- Ça veut dire que cette notion de complétude n'est pas très intéressant pour **P**...

Demonstration

- Soit $L \in \mathbf{P}$ avec $L \neq \emptyset$ et $L \neq \Sigma^*$
- Alors il existe $a \in L$ et aussi $b \notin L$
- Soit $L' \in \mathbf{P}$ et soit $f(x) = \begin{cases} a & \text{si } x \in L' \\ b & \text{si } x \notin L' \end{cases}$
- Comme $L' \in \mathbf{P}$, la fonction f est calculable en temps polynomial
- Mais aussi $f(x) = a \in L$ ssi $x \in L'$, donc $L' \leq L$