

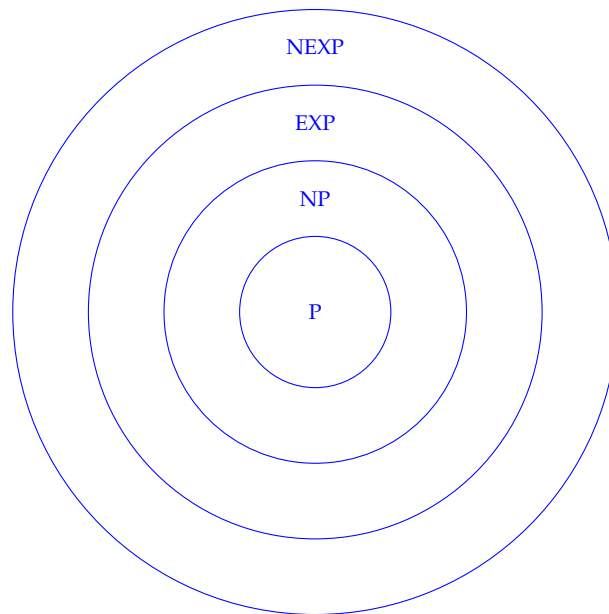
**TD 06 – P, NP et EXP**

---

**Exercice 1.**

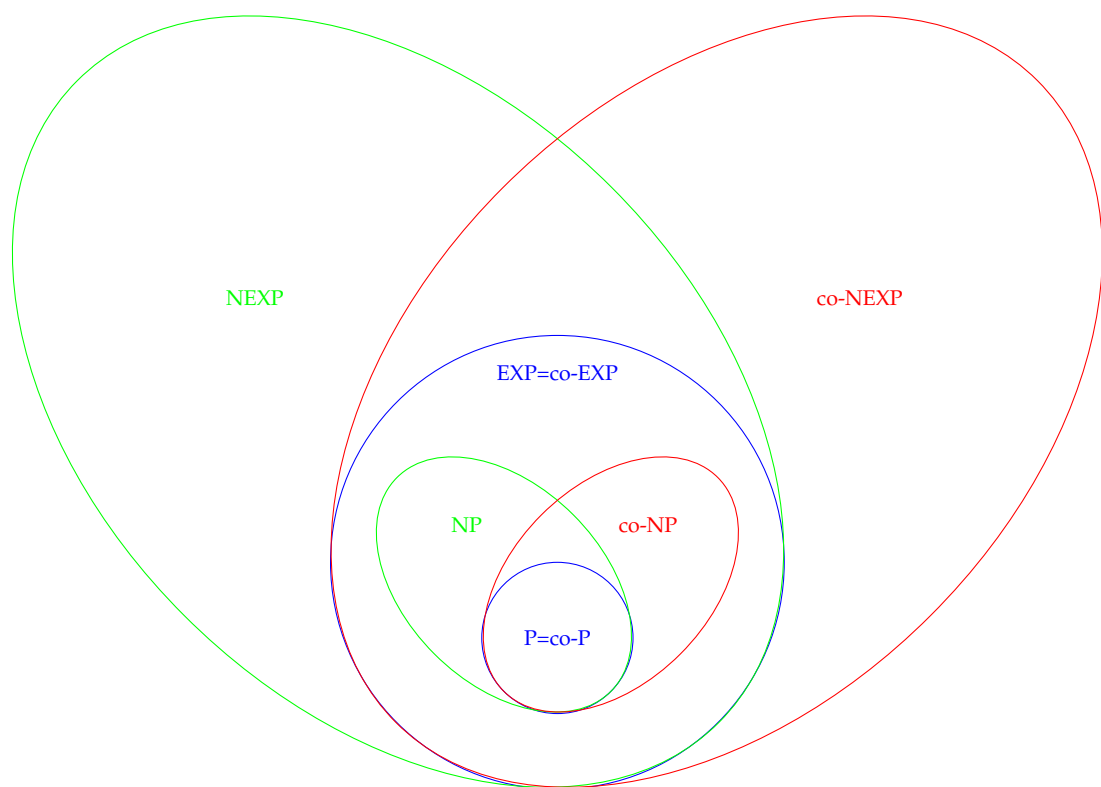
*Inclusion classes de complexité*

1. Dessiner le diagramme de Venn des classes suivantes : P, NP, EXP, NEXP.



2. Y ajouter les classes co-P, co-NP, co-EXP et co-NEXP.





Remarque : on ne sait pas si  $NP \cap coNP = P$  même en supposant que  $P \neq NP$ .

Autre remarque : 50% de mon temps de préparation de TD a été consacré à faire cette \*\*\*\* de figure.)

3. Citer deux inclusions connues pour être strictes.

☞  $P \subset EXP$  et  $NP \subset EXP$

4. Quelle est la question de complexité à 1 million de dollars ? (de la part du Clay Mathematics Institute).

☞  $P \stackrel{?}{=} NP$

5. Est-ce que l'ensemble des problèmes dans  $P$  est dénombrable ?

☞ Oui, d'une façon générale, l'ensemble des problèmes semi-décidables est déjà dénombrable car chaque problème semi-décidable correspond à au moins une machine de Turing et que l'ensemble des machines de Turing est dénombrable. (on peut en effet représenter de façon non ambiguë une machine de Turing  $M$  par son code  $\langle M \rangle$ ).

## Exercice 2.

*Vrai ou Faux*

Pour chacun des énoncés suivants, dire s'il est vrai (donner une preuve) ou faux (donner un contre exemple).

1. Si  $P = NP$  alors  $NP = co-NP$ .

☞ Oui car si  $P = NP$  alors  $co-NP = co-P = P = NP$ .

2. Si  $NP = EXP$  alors  $NP = co-NP$ .

☞ Même argument : si  $EXP = NP$  alors  $co-NP = co-EXP = EXP = NP$ .

3.  $NP \neq NEXP$ .

☞ Oui, par l'argument de la hiérarchie des temps

4.  $P \neq \text{coNEXP}$ .

☞ Oui, par l'argument de la hiérarchie des temps car  $P \subset \text{EXP} \subseteq \text{coNEXP}$ .

5. Si un problème est dans  $P$  alors il n'est pas dans  $\text{EXP}$ .

☞ Non, tout problème dans  $P$  est dans  $\text{EXP}$

### Exercice 3.

Problème  $\text{halt}_{\epsilon,k}$

#### Arrêt en $k$ étapes.

entrée : Une machine de Turing  $M$ , un mot  $w$  et un entier  $k$  encodé en binaire.

question :  $|M(w)| \leq k$ ?

(On suppose que si  $M(w) \uparrow$  alors  $|M(w)| = \inf$ )

Au TD précédent, nous avons montré que ce problème était dans  $\text{EXP}$ . Nous allons maintenant montrer qu'il n'est pas dans  $P$ .

1. Supposons que nous avons une machine  $M_{\text{halt}}$  qui décide le problème de l'arrêt en  $k$  étapes. Autrement dit  $M_{\text{halt}}(\langle M \rangle \# w \# k)$  accepte si et seulement si  $M$  s'arrête sur l'entrée  $w$  en moins de  $k$  étapes.

À partir de cette machine, construire une machine  $M_d$  tel que :

- $M_d(\langle M \rangle / k) \uparrow$  si  $|M(\langle M \rangle / k)| \leq k$  ( $M_d$  boucle à l'infinie si  $M$  s'arrête sur son propre code en moins de  $k$  étapes);
- $M_d(\langle M \rangle / k)$  accepte sinon.

☞ Ne pas hésiter à faire un petit schéma au tableau pour que ce soit plus claire.  $M_d$  doit d'abord transformer son entrée  $\langle M \rangle / k$  sur le ruban en une entrée  $\langle M \rangle \# \langle M \rangle / k \# k$ . Ensuite il rembobine et lance  $M_{\text{halt}}$ . Ensuite soit  $M_{\text{halt}}$  accepte et  $M_d$  rentre dans un état poubelle, soit  $M_{\text{halt}}$  rejette et  $M_d$  entre dans un état acceptant.

2. On considère l'exécution  $M_{\text{halt}}(\langle M_d \rangle \# \langle M_d \rangle / k \# k)$ . Quel est la taille de l'entrée?

☞  $2|\langle M_d \rangle| + 3 + \lceil 2 \log_2(k) \rceil$  car  $k$  est encodé en binaire. Notons que  $2|\langle M_d \rangle| + 3$  est une constante.

3. Maintenant, supposons (par l'absurde) que  $M_{\text{halt}}$  décide en temps polynomial en fonction de la taille de l'entrée. Que peut-on dire sur  $|M_{\text{halt}}(\langle M_d \rangle \# \langle M_d \rangle / k \# k)|$  (asymptotiquement en fonction de  $k$ )?

☞ L'exécution prend un temps  $O(n^c)$  avec  $n$  la taille de l'entrée et  $c$  une constante. On a dit que  $n = 2|\langle M_d \rangle| + 3 + \lceil 2 \log_2(k) \rceil$ .

$2|\langle M_d \rangle| + 3$  et l'arrondi supérieur sont négligeables asymptotiquement. Donc le temps d'exécution est en  $O(\log(k)^c)$  avec  $c$  une constante.

4. Lors de l'exécution de  $M_d(\langle M_d \rangle / k)$ , en combien d'étape de temps  $t_k$  soit l'état acceptant soit l'état poubelle auront-ils été atteints? (asymptotiquement en fonction de  $k$ )?

☞ Avant de rejoindre l'état poubelle ou l'état acceptant, il faut recopier le code de  $M_d$  et  $k$  (mais ça prend un temps constant) et exécuter  $M_{\text{halt}}(\langle M_d \rangle \# \langle M_d \rangle / k \# k)$  (on a dit que c'est en  $O(\log(k)^c)$ ). Donc  $t_k$  est en  $O(\log(k)^c)$ .

5. Peut-on trouver des valeurs de  $k$  pour lesquels  $t_k < k$ ?

☞ Oui, on a dit que  $t_k$  est en  $O(\log(k)^c)$ . Asymptotiquement, c'est négligeable par rapport à  $k$  peu importe la valeur de  $c$ .

6. Est-ce que l'exécution  $M_d(\langle M_d \rangle / k)$  va faire une boucle infinie ou accepter pour de telles valeurs de  $k$ ? Conclure.

☞ Il y a deux possibilités.

- $|M_d(\langle M_d \rangle / k)| = \inf \Leftrightarrow |M_d(\langle M \rangle / k)| \leq k$ . Absurde. Notons que nous n'avons pas utilisé l'hypothèse de temps d'exécution de  $M_{\text{halt}}$  dans ce cas.
- $M_d(\langle M_d \rangle / k)$  accepte. Donc  $|M_d(\langle M_d \rangle / k)| > k$ . Mais on a dit qu'en moins de  $t_k < k$  étapes,  $M_d$  avoir déjà rejoint soit l'état poubelle, soit avoir accepté. Donc s'il n'a pas terminé en moins de  $k$  étapes, c'est que  $M_d(\langle M_d \rangle \# k)$  fait une boucle infinie et donc rejette. Absurde.

Les deux cas sont absurdes. On en conclut que le problème de l'arrêt en  $k$  étapes ne peut pas être résolu en temps  $O(\log(k)^c)$  et donc qu'il n'est pas dans  $P$ . C'est au passage une preuve que  $P \neq EXP$ .

#### Exercice 4.

Carrés magiques

##### DIVISEUR

*entrée* : Deux entiers  $n$  et  $k$  codé en binaire

*question* : Est-ce qu'il existe  $2 \leq p \leq k$  tel que  $p$  divise  $n$  ?

1. Donner un algorithme haut-niveau *déterministe* pour résoudre ce problème. Donner la complexité en temps de l'algorithme et en déduire une classe de complexité du problème DIVISEUR.



```
def diviseur(n,k):
    pour i entre 2 et k:
        si n % i == 0:
            retourner Vrai
    retourner Faux
```

Le temps d'exécution est en  $O(k)$  (en négligeant le temps de la division), la taille de l'entrée étant  $O(\log(k) + \log(n))$ , ça fait que le problème est dans  $EXP$ .

2. Donner un algorithme haut-niveau *non-déterministe* pour résoudre ce problème (vous pouvez utiliser l'instruction *deviner*(1,  $m$ ) pour deviner un entier entre 1 et  $m$ ). Donner la complexité en temps de l'algorithme et en déduire une meilleure classe de complexité de DIVISEUR.



```
def diviseur(n,k):
    i = deviner(1,k)
    retourner n % i == 0
```

Le temps d'exécution est en  $O(1)$  (ou plutôt en  $O(\log(n) + \log(k))$ , le temps de calcul de la division n'est pas négligeable comme il n'y a aucune boucle mais en tout cas ça reste polynomial). Donc le problème est dans  $NP$ .

3. Le chiffrement RSA est un algorithme de cryptographie asymétrique, très utilisé pour échanger des données confidentielles sur Internet.

[La page wikipedia](#) offre une explication détaillée mais le principe est le suivant. Chaque utilisateur a

- une clé privée (que lui seul connaît) : deux entiers premiers  $p$  et  $q$
- une clé publique (que le monde entier peut connaître)  $n = pq$ .

En utilisant la clé publique, on peut chiffrer les messages et avec la clé privée on peut les déchiffrer. Si vous aviez un algorithme polynomial pour résoudre le problème DIVISEUR, que pourriez-vous faire pour casser le chiffrement RSA ?

En connaissant juste la clé publique, on pourrait en déduire la clé privée et ainsi déchiffrer tous les messages qui ne nous sont pas. L'algorithme consiste à jouer au plus ou moins dichotomique pour trouver le plus petit diviseur de  $p$  de  $n$  (et après ça,  $q = n/p$ ).

```
def plus_petit_diviseur(n):
    d = 2
    f = n
    tantque d != f:
        si diviseur(n,(d+f)//2):
            f = (d+f)//2
        sinon:
            d = (d+f)//2+1
    retourner d
```

L'algorithme coupe l'écart entre  $d$  et  $f$  par 2 à chaque étape. Donc il prend un temps  $O(\log(n) * n^c)$  ou  $n^c$  est le temps polynomial pris par l'algorithme qui décide le problème DIVISEUR. On final, si DIVISEUR est dans  $P$  alors on peut casser une clé publique en temps polynomial.

### Exercice 5.

*Carrés magiques (Bonus)*

En mathématiques, un carré magique (normal) d'ordre  $n$  est une grille de taille  $n \times n$  composée des  $n^2$  entiers entre 1 et  $n^2$ . Ces nombres sont disposés de sorte que leurs sommes sur chaque rangée, sur chaque colonne et sur les deux diagonales soient égales. Quelques exemples de carrés magiques (normaux) :

8	1	6
3	5	7
4	9	2

1	14	4	15
8	11	5	10
13	2	16	3
12	7	9	6

16	14	7	30	23
24	17	10	8	31
32	25	18	11	4
5	28	26	19	12
13	6	29	22	20

1	35	4	33	32	6
25	11	9	28	8	30
24	14	18	16	17	22
13	23	19	21	20	15
12	26	27	10	29	7
36	2	34	3	5	31

35	26	17	1	62	53	44
46	37	21	12	3	64	55
57	41	32	23	14	5	66
61	52	43	34	25	16	7
2	63	54	45	36	27	11
13	4	65	56	47	31	22
24	15	6	67	51	42	33

#### CARRE\_MAGIQUE

*entrée* : Une grille  $n \times n$  déjà partiellement remplie.

*question* : Est-ce que cette grille peut être complétée pour obtenir un carré magique ?

- Donner un algorithme haut-niveau *déterministe* pour résoudre ce problème. Donner la complexité en temps de l'algorithme et en déduire une classe de complexité du problème CARRE\_MAGIQUE.

☞ On peut proposer un algorithme de backtracking récursif. On remplit les cases une par une avec des entiers entre 1 et  $n^2$ . À tout moment, si on se rend compte qu'il y a une incohérence (un nombre présent deux fois, une ligne ou une colonne complète dont la somme n'est pas  $(1 + n^2) * n / 2$ ) alors on revient en arrière.

La complexité de l'algorithme est en  $O(N^N)$  avec  $N = n^2$  et l'entrée est en  $O(N \log(N))$ . Le problème est donc dans la classe EXP.

- Donner un algorithme haut-niveau *non-déterministe* pour résoudre ce problème (vous pouvez utiliser l'instruction *deviner*(1,  $m$ ) pour deviner un entier entre 1 et  $m$ ). Donner la complexité en temps de l'algorithme et en déduire une meilleure classe de complexité de CARRE\_MAGIQUE.

☞ Il s'agit simplement de deviner les cases non remplies de la grille et de vérifier que la grille ainsi remplie est cohérente. (La grille remplie est donc le certificat.) Le temps d'exécution non déterministe de notre algorithme est en  $O(N)$  pour deviner les entiers et en  $O(N^2)$  et pour vérifier la cohérence de la grille (voir même en  $O(N)$ ). Donc le problème est dans la classe NP.