Université Pierre-et-Marie-Curie

Master d'informatique

4I019 - Complexité, Algorithmes Randomisés et Approchés

Fanny Pascual
Damien Vergnaud

Notes de cours

Écrit par B.T. Luong

Table des matières

Première partie

Introduction à la théorie du calcul et à la complexité des problème, Méthodes arborescentes, et Algorithmes d'approximation

1 Introduction à la théorie du calcul

1.1 Problème, algorithme

Problème de décision :
- D : ensemble des instances
- D^+ : ensemble des instances dont la réponse est "oui"

Étant donné I $\in D$, est-ce que I $\in D^+$?

3 types de problèmes :
- Problème indécidables : Il n'existe pas d'algorithme les résolvant
- Problème facile : Il existe un algorithme efficace les résolvant
- Problème difficile : on conjecture qu'$\nexists$ d'algorithme efficace les résolvant

Problèmes indécidables :
- Problème de l'arrêt
- Comparaison 2 algorithmes

1.2 Formalisation

1.2.1 Problème et langage

Une machine code des données avec un alphabet Σ.
Ordinateur $\Sigma = \{0, 1\}$.
Un mot : suite infinie de symboles de Σ.
Exemple : 01100.
Un langage : ensemble de mots.
Exemple : $\Sigma = \{0, 1, \#\}$
L = {w#w | w est un mot sur {0, 1}}
010#010 $\in$ L
010#010 $\notin$ L
Soit L un langage :
Instance : un mot x
Question : x $\in$ L ?

1.2.2 Machine de Turing

a Description de haut niveau
- Une mémoire : un ruban infini découpé en cases.
- Une tête de lecture/écriture
- Des règles de fonctionnement qui indiquent ce que fait la tête de lecture

Une MT possède un nombre fini d'état dont :
- L'état initial

— L'état d'acceptation : q accepter
— L'état rejet : q rejet

Instruction

Si je suis dans un état E_i et que je lis un symbole α alors :

1. J'écris un symbole β
2. Je me déplace d'une case (soit à gauche ou à droite)
3. Je passe dans l'état E_j

Description de haut niveau d'une MT

— Se déplacer de la 1^{ere} position du mot à gauche du # à la 1^{ere} position du mot à droite du # et vérifier que ces mots contiennent le même symbole. Si ce n'est pas les cas → Rejeter.
— Barrer les symboles dès qu'ils sont vérifiés.
— Recommencer avec la position suivante tant que tous les symboles à gauche de # n'ont pas été barrés.
— Quand tous les symboles à gauche du # ont été barrés, regarder s'il existe des symboles à droites du #. Si c'est le cas rejeter, sinon accepter.

b Définition formelle d'une MT (bas niveau)

Une MT est un 7-uplet :

1. Q : L'ensemble des états.
2. Σ : L'alphabet d'entrée (qui contient pas les symbole ␣)
3. Γ : L'alphabet du ruban avec ␣$\in \Gamma$, et $\Sigma \in \Gamma$
4. δ = Q x Γ → Q x Γ x {G, D} : La fonction de transition
5. $q_0 \in$ Q : L'état initial
6. $q_{accepter} \in$ Q
7. $q_{rejet} \in$ Q ($q_{accepter} \neq q_{rejet}$)

Exemple :
E ={$\#x_1\#x_2...\#x_n$ est un mot | $x_i \in \{0, 1\}$ et $x_i \neq x_j$ $\forall$ i ≠ j}

c Langage/Problème décidable

Soit M une MT
— M accepte un mot w si l'exécution de M sur w conduit à $q_{accepter}$.
— M rejette un mot w si l'exécution de M sur w conduit à q_{rejet}.

Langage connu par M = {w | M accepte w}
Un décideur : MT qui termine pour tout mot w
Un langage est décidé par une MT s'il existe une MT que le décide.
Problème indécidable : ∄ MT qui le décide.

Problème de Hilbert

D = {p | p est un polynôme ayant une racine entière}
p(x) = $3x^2$ + 2x - 1 = 0
M : "Évaluer p(x) avec x prenant successivement les valeurs 0, 1, -1, 2, -2, ...
Si p(x) = 0 pour une de ces valeurs, alors accepter.
Plusieurs variables : Polynôme de Matijasevic.

d Machine de Turing universelle

MT qui simule n'importe quelle autre MT.
Reconnaît le langage {<M,w> | M est une MT qui accepte w} sur l'entrée (M,w).
MTU =

1. Simule M sur l'entrée w
2. Si M entre dans un état d'acceptation, accepter.
   Si M entre dans un état de rejet, rejeter.

Machine de Turing non déterministe
À chaque itération, la machine ne peut avoir plusieurs actions possibles.
δ = Q x $\Gamma \longmapsto$ P (Q x Γ x {G, D})
Une MTND accepte le mot w $\Leftrightarrow$ Il existe une branche que mène à l'état d'acceptation.

f La thèse de Church Turing (1930 - 1936)
Notion intuitive d'algorithme = MT.
Un problème peut être résolu par un algorithme ssi il peut être décidé par une MT.

2 Complexité de problèmes

2.1 Complexité temporelle définie par les machines de Turing

Soit M une MTD qui est un décideur.
La complexité temporelle de M est une fonction f : $\mathbb{N} \longmapsto \mathbb{N}$ où f(n) est le nombre maximum d'étapes que n utilise pour une entrée de talle n.

2.2 Complexité d'un problème définie par les machines de Turing

Soit f : $\mathbb{N} \longmapsto \mathbb{R}^+$ une fonction.
La classe de complexité TIME(f(n)) = l'ensemble de tous les langages qui sont décidables par une *MTD* qui s'exécute en temps f(n).

2.3 La classe P

P = $\bigcup_{k \in \mathbb{N}}$ TIME(n^k)
Ensemble de tous les langages décidables en temps polynomial par une MTD.

2.4 Temps d'exécutions d'une machine de Turing non déterministe

Soit M une MTND qui est un décideur.
Le temps d'exécution de M est une fonction f : $\mathbb{N} \longmapsto \mathbb{N}$ où f(n) = nombre maximum d'étapes que M utilise pour n'importe quelle entrée de taille n.
Soit f : $\mathbb{N} \longmapsto \mathbb{R}^+$. La classe de complexité NTIME(f(n)) = ensemble de tous les langages décidables par une MTND qui s'exécute en f(n).

2.5 La classe NP

NP (non déterministe polynomial) = $\bigcup_{k \in \mathbb{N}}$ NTIME(n^k)
Ensemble des langages décidés en temps polynomiale par une MTND.

Proposition :
Un langage L sur un alphabet Σ est dans NP si et seulement s'il existe un polynôme p et une MTND M telle que tout mot x de taille n : x $\in$ L $\Leftrightarrow$ $\exists$y (certificat "indice") $\in \Sigma^{p(n)}$, M accepte l'entrée (x,y).

2.6 Exemple de problème dans NP

CHAÎNE HAMILTONIENNE : CH

- **Entrée :** G.
- **Question :** G possède-t-il une chaîne hamiltonienne ie une chaîne qui passe une fois et une seule fois par chaque sommet de G ?
- **Certificat :** Chaîne hamiltonienne.

CLIQUE

- **Entrée :** Un graphe non orienté G et un entier k.
- **Question :** G possède-t-il une clique de taille k ? (un ensemble de k sommets 2 à 2 adjacents)
- **Certificat :** Une clique de taille k.

PARTITION

- **Entrée :** S ensemble de n entiers positifs.
- **Question :** Peut-on partitionner S en 2 sous-ensembles S_1 et S_2 tels que la somme des entiers dans S_1 = la somme des entiers dans S_2 ?
- **Certificat :** S_1 et S_2.

SATISFIABILITÉ (SAT)

- **Entrée :** Une formule booléenne ϕ.
- **Question :** ϕ est-elle satisfiable ?
- **Certificat :** Valeurs de vérité (V ou F) aux variables.

2.7 P $\vee$ NP

P $\leftarrow$ problèmes dont on peut rapidement trouver leur solution.
NP $\leftarrow$ problèmes dont on peut rapidement vérifier qu'une solution est bien une solution.
P $\subset$ NP
Question ouverte : P $\neq$ NP ?

2.8 Problème NP-complet

Un problème NP-complet si sa complexité est liée à celle de la classe NP.
Un algorithme polynomial résolvant un problème NP-complet permet de résoudre en temps polynomial tout problème de NP.

Théorème : SAT $\in$ P $\Leftrightarrow$ P = NP

2.9 Réduction polynomiale

Le problème A peut être réduit en temps polynomial au problème B ce qui s'écrit A $\leq_p$ B s'il existe une fonction f exécutable en temps polynomial telle que pour toute instance w de A :

La réponse au problème A sur l'instance w est OUI.
$\Leftrightarrow$ La réponse au problème B sur l'instance f(w) est OUI.
f est une réduction polynomiale de A vers B.

Théorème : Si A $\leq_p$ B et B $\in$ P alors A $\in$ P.

Définition : Soit A un problème de décision.
A est NP-complet si :

1. A $\in$ NP
2. Chaque problème B de NP peut être réduit en temps polynomial à A

2.10 Comment montrer qu'un problème est NP-complet

Théorème : Si B est NP-complet et B $\leq_p$ A avec A $\in$ NP, alors A est NP-complet.
Pour montrer qu'un problème est NP-complet :

1. On montre que A $\in$ NP (certificat)
2. On montre qu'il existe un problème NP-complet B tel que B $\leq_p$ A

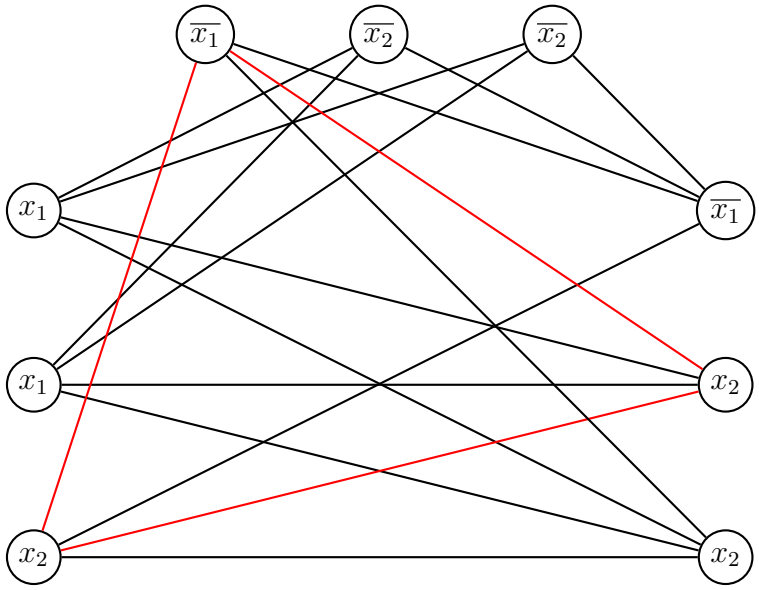
Résumé :

1. Énoncer clairement le problème A dont on veut montrer qu'il est NP-complet.
2. On montre que A $\in$ NP (certificat).
3. Énoncer clairement un problème B dont on sait qu'il est NP-complet.
4. On décrit une réduction polynomiale f qui transforme B en A.
5. On prouve que f transforme B en A : $\forall$ entrée w de B, on obtient une entrée f(w) de A tel que : la solution au problème B sur l'instance w est OUI $\Leftrightarrow$ la solution au problème A sur l'instance f(w) est OUI.

Exemple : On montre que CLIQUE est NP-complet

1. Problème CLIQUE :
   Entrée : Un graphe C et un entier k
   Question : Est-ce qu'il existe une clique de taille k dans G ?
2. CLIQUE $\in$ NP
   Certificat : clique de taille k
3. Problème 3-SAT :
   Une formule booléenne est en forme normale conjonctive (FNC) si elle constitue d'un ensemble de clauses (litéraux reliés par des $\vee$) reliées par des $\wedge$.
   Exemple : $(x_1 \vee x_2) \wedge (\overline{x_2} \vee x_3 \vee \overline{x_4})$
   Une formule booléenne 3FNC est une formule en FNC telle que chaque clause a 3 littéraux.
   Exemple : $(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee x_4 \vee \overline{x_2})$
   Problème 3-SAT :
   Entrée : Une formule booléenne 3FNC ϕ
   Question : ϕ est-elle satisfiable ?
4. On réduit 3-SAT à CLIQUE :
   Étant une instance de 3-SAT à m clauses, on construit l'instance de CLIQUE suivant :
   — k = m
   — les sommets de G sont organisés en triples t_1, ..., t_m
   Chaque triplet correspond à une clause de ϕ :

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



Les arêtes de G lient tous les sommets sauf ceux qui sont dans un des 2 cas suivant :
— Il n'y a pas d'arête entre 2 sommets d'un même triplet.
— Il n'y a pas d'arête entre 2 sommets étiquetés par des littéraux opposés.

5. Propriété : Il existe dans G(ϕ) une clique de taille k $\Leftrightarrow$ ϕ est satisfiable.
($\Leftarrow$) Supposons que ϕ est satisfiable. Donc on a au moins 1 littéral VRAI dans chaque clause. Dans chaque triplet de G on sélectionne un littéral qui a la valeur VRAI dans ϕ. Les sommets sélectionnés forment une clique de taille k :
— k sommets sélectionnés
— chaque paire de sommets est reliée par une arête (pas de 2 sommets est d'un triplet, pas de 2 sommets correspondant à des littéraux opposés)
($\Rightarrow$) Supposons qu'il existe une clique de taille k dans G(ϕ). Montrons que ϕ est satisfiable.
— On n'a pas 2 sommets dans un même triplet.
— On associe la valeur VRAI à chaque littéral associé à un sommet de la clique.
ϕ est satisfiable car :
— On n'a pas affecté la valeur VRAI à des littéraux opposés.
— On a au moins un littéral VRAI par clause.

3 Algorithmes approchés à garantie de performance

Problème d'optimisation
— **Entrée :** I
— **Fonction objective f :** $A_I \longmapsto \mathbb{R}$
⋆ Trouver un élément $\overline{x}$ de A_i tel que f($\overline{x}$) $\leq$ f(x) $\forall$ x $\in A_I$. (problème de minimisation)
⋆ Trouver un élément $\overline{x}$ de A_i tel que f($\overline{x}$) $\geq$ f(x) $\forall$ x $\in A_I$. (problème de maximisation)

Problème de décision associé
— **Entrée :** I, borne B
— **Fonction objective f :** $A_I \longmapsto \mathbb{R}$
⋆ Est-ce qu'il existe un élément x $\in A_I$ tel que f(x) $\leq$ B ? (problème de minimisation)
⋆ Est-ce qu'il existe un élément x $\in A_I$ tel que f(x) $\geq$ B ? (problème de maximisation)

Un problème d'optimisation est NP-difficile si le problème de décision associé est NP-complet.

Un algorithme est α-approché s'il retourne pour chaque instance du problème une solution dont le coût est :

— au plus α OPT pour un problème de minimisation
— au moins α OPT pour un problème de maximisation
où OPT la valeur d'une solution optimale et α le rapport d'approximation.

Exemple : MAX SAT
Entrée : Une forme FNC ϕ.
But : Maximiser le problème de clauses satisfiables dans ϕ.
Exemple : $(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_2 \vee x_1) \wedge \overline{x_2} \wedge (\overline{x_2} \vee \overline{x_1})$
Algorithme :

1. Affecter toutes les variables à VRAI $\Rightarrow m_1$ clauses satisfiables.
2. Affecter toutes les variables à FAUX $\Rightarrow m_2$ clauses satisfiables.

Retourner l'affectation qui satisfait le plus de clauses : #Clauses satisfaites = max $\{m_1, m_2\}$

$$m_1 + m_2 \geq \text{m} \Rightarrow \max\{m_1, m_2\} \geq \frac{m}{2} \geq \frac{OPT}{2}$$

Or OPT $\leq$ m
Cet algorithme est $\frac{1}{2}$-approché.

3 catégories de problème d'optimisation

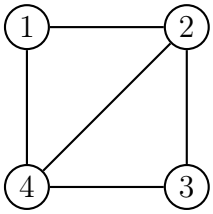
1. Les problèmes non-approximables
   $\nexists$ d'algorithme polynomial α-approché, $\forall\alpha$ constante, si P $\neq$ NP.
2. Les problèmes approximables
   Il existe une constante α telle qu'il existe algorithme polynomial α-approché.
3. Les problèmes totalement approximables
   Un schéma d'approximation est pour un problème P est un algorithme qui prend en entrée :
   — une instance I de P
   — un nombre $\epsilon > 0$ (précision souhaitée)
   et qui retourne une solution SA dont le coût est tel que :
   — coût $(S_A) \geq (1 - \epsilon)$ OPT si P est un problème de maximisation.
   — coût $(S_A) \leq (1 + \epsilon)$ OPT si P est un problème de minimisation.

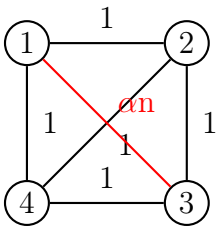
Exemple :

Problème VOYAGEUR DE COMMERCE (VC)

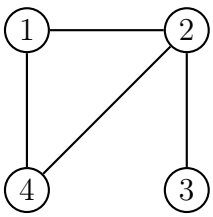
Entrée : G un graphe complet G = S, A dont les arêtes sont valuées par un coût ≥ 0.
Question : Trouver un cycle hamiltonien de coût minimum.
Théorème : $\forall$ constante α, $\nexists$ d'algorithme α-approché polynomial pour le problème VC sauf si P = NP.
Preuve :
Idée : Réduction CYCLE HAMILTONIENNE $\Rightarrow$ VC
Soit A un algorithme polynomial α-approché pour VC.
On suppose P $\neq$ NP.
On transforme G : entrée VC $\Rightarrow$ G' : entrée CH, telle que :
— Si G' admet un cycle hamiltonien alors le coût optimal du VC dans G est n.
— Sinon le coût optimal du VC dans G $> \alpha$n.
Réduction :
G :
— même que G'
— arête a un coût 1 ssi {u, v} appartient à G'
— sinon {u, v} a un coût αn

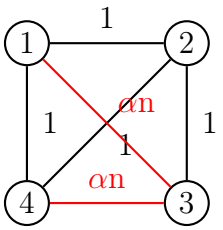
Supposons qu'il existe un algorithme polynomial α-approché pour le problème du VC.
Réduction :
CYCLE HAMILTONIEN (NP-'complet)
Entrée : Graphe G) (S, A)
VOYAGEUR DE COMMERCE
Entrée C' = (S', A) aec S' = S
Pondération $\forall\{u, v\} \in A'$
— Si $\{u, v\} \in A$ alors coût($\{u, v\}$) = 1
— Sinon coût($\{u, v\}$) = αn
$\mathcal{A}$ permet de détecter un cycle hamiltonien dans G.
— Si G contient un cycle hamiltonien $\Rightarrow$ OPT = n

G : 

$\Rightarrow$ G' : 

$\Rightarrow \mathcal{A}$ retourne une solution de coût $\leq \alpha n$.
— Si G ne contient pas de cycle hamiltonien, la solution optimale du VC utilise au moins 1 arête de coût αn $\Rightarrow$ OPT $> \alpha n$

G : 

$\Rightarrow$ G' : 

$\Rightarrow \mathcal{A}$ répond en temps polynomial de problème du cycle hamiltonien.

Donc P = NP $\Rightarrow$ Contradiction.

4 Méthodes arborescents exactes et approchées

Voyageur de commerce dans un graphe métrique

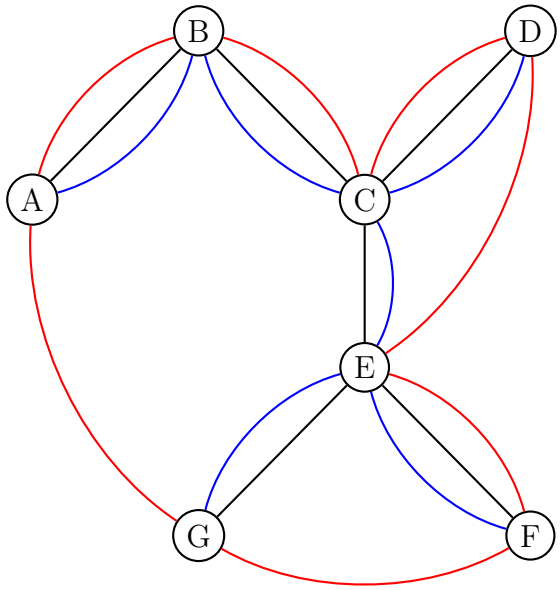
Algorithme 2-approché :
Soit G = (S, A)
Inégalité triangulaire : $\forall$ triplet (a, b, c) $\in S^3$

$$\text{coût}(\{a, b\}) \leq \text{coût}(\{b, c\}) + \text{coût}(\{c, a\})$$

Algorithme

1. Calculer un arbre courant de coût minimum, T de G.
   coût(T) $\leq$ OPT
2. Dupliquer chaque arête de T pour obtenir un graphe eulérien T'.
   coût(T') = 2coût(T) $\leq$ 2OPT
3. Calculer un cycle eulérien de T'.
   coût(C) = coût(T') $\leq$ 2OPT
4. Retourner $\mathcal{C}$ le cycle qui passe par tous les sommets de G.
   coût($\mathcal{C}$) $\leq$ coût(C) (inégalité triangulaire)
   $\Rightarrow$ Donc coût($\mathcal{C}$) $\leq$ 2OPT

L'algorithme est 2-approché.



— T
— T'
— $\mathcal{C}$

C = (A, B, C, D, C, E, F, E, G, E, C, B, A)
$\mathcal{C}$ = (A, B, C, D, E, F, G, A)

Soit OPT la valeur d'une solution optimale du VC.
On a coût(T) $\leq$ OPT.
Tour optimal de coût OPT.
On lui retire une arête. On obtient O' de coût O' $\leq$ OPT et coût O'$\geq$ coût(T).

Problème du Sac à dos

Schéma d'approximation algorithme (1 - ϵ)-approché.
Polynomial en la taille d'entrée et $\frac{1}{\epsilon}$.
Algorithme pseudo-polynomial dont le temps d'exécution est, pour tout instance I, bornée par un polynôme de taille de I dans laquelle les nombres sont codés en unaire.
Soit OPT la valeur des objets dans une solution optimale, V_{max} la valeur maximale d'un objet de S.
On a n$V_{max} \geq$ OPT.
Soit i $\in \{1, ..., n\}$ et v $\in \{0, 1, ..., nV_{max}\}$.
Soit S_{iv} un sous-ensemble de $\{a_1, ..., a_i\}$ de valeur v et de poids minimum.
P(i, v) : poids cumulé des objets de S_{iv} (P(i, v) = $+\infty$ si S_{iv} n'existe pas).

On a : P(1, 0) = 0
P(1, v) = poids(a_1) si v = val(a_1)
$\forall$ v $\neq$ 0 ∞ sinon

Exemple : Sac à dos
— Capacité B = 6
— Objets B = $\{a_1, ..., a_6\}$

B	a_1	a_2	a_3	a_4
Poids	2	7	1	2
Val	1	4	2	2

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\{a_1\}$	0	2	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
$\{a_1, a_2\}$	0	2	∞	∞	7	9	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
$\{a_1, a_2, a_3\}$	0	2	1	3	7	9	8	10	∞	∞	∞	∞	∞	∞	∞	∞
$\{a_1, a_2, a_3, a_4\}$	0	2	1	3	3	5	8	10	10	12	∞	∞	∞	∞	∞	∞
						OPT										

P(i + 1, v) = P(i, v) si val(a_{i+1}) > v
min {P(i, v), poids(a_{i+1}) + P(i, v) - val(a_{i+1})} sinon
OPT = max {v= P(i, v) ≤ B}
Complexité : Θ (n * (nV_{max} + 1)) = Θ ($n^2 V_{max}$)

Un schéma d'approximation totalement polynomial

1. Étant donné $\epsilon > 0$, poser K = $\frac{\epsilon V_{max}}{n} \Rightarrow \frac{V_{max}}{k} = \frac{n}{\epsilon}$
2. Pour chaque objet a, poser valeur $(a_i) = \lfloor \frac{valeur(a_i)}{K} \rfloor$
3. Lancer l'algorithme de programmation dynamique avec ces nouvelles valeurs arrondies et trouver un ensemble S' de valeur maximum.
4. Retourner S'.

Théorème : Cet algorithme est un schéma d'approximation.
Complexité : O (n * nV_{max}') = O (n * n$\lfloor \frac{V_{max}}{K} \rfloor$) = O (n * n$\lfloor \frac{n}{\epsilon} \rfloor$) = O ($\frac{n^3}{\epsilon}$)

On montre maintenant que valeur(S') ≥ (1 - ϵ) OPT.
Notons val(a_i) la vraie valeur et val'(a_i) la valeur arrondie.
— $V_{max} \leq$ OPT
— Soit X un sous-ensemble d'objets
— Soit $a_i \in$ X : val(a_i) = K val'(a_i) + x_i avec $0 \leq x_i \leq$ K

Définition :
val(X) = $\sum_{a_i \in X}$ val(a_i)
val'(X) = $\sum_{a_i \in X}$ val'(a_i)

On a :

$$\text{val(X)} = \text{Kval'(X)} + \sum x_i \text{ avec } 0 \leq \sum x_i \leq \text{nK}$$

Ceci est vrai pour X = S'.
Donc :

$$val(S') \geq Kval'(S') \tag{1}$$

Ceci est vrai pour X = O, où O est sous-ensemble d'objets d'une solution optimale.

$$\begin{aligned} \text{val(O)} &= \text{Kval'(O)} + \sum x_i \\ Kval'(O) &\geq \underbrace{val(O)}_{=OPT} - nK \end{aligned} \tag{2}$$

De plus, on a :

$$val'(S') \geq val'(O) \tag{3}$$

car S' est la solution optimale du problème avec les valeurs arrondies.
On a donc :

$$\begin{aligned} val(S') &\geq_1 Kval'(S') \\ &\geq_3 Kval'(O) \\ &\geq_2 val(O) - nk = OPT - \epsilon V_{max} \\ &\geq (1 - \epsilon) OPT \end{aligned}$$

Deuxième partie

Algorithmes probabilistes, classes de complexité probabilistes

5 Algorithme probabiliste

5.1 Introduction

Aiguille de Buffon (1773) - Proposition d'une expérience aléatoire :
Si on prend ces "aiguilles" de longueur d, si les lancers d'aiguilles sont "indépendants" et si le nombre de lancers est suffisamment grand, le quotient du nombre d'aiguilles touchant une ligne par le nombre d'aiguilles total tend vers $\frac{2}{\pi}$.
http://www.ventrella.com/buffon/
Méthode de Monté Carlo (1947)
Métropolis et Ulam : une autre approche pour calculer π.
On peut tirer au hasard des points (x,y) dans le carré $0 \leq x \leq 1$, $0 \leq y \leq 1$ et compter ceux qui sont à distances inférieure à 1 de l'origine (i.e. $x^2 + y^2 \leq 1$).
Le quotient du nombre de points à l'intérieur du quart de cercle par le nombre de points total tend vers $\frac{\pi}{4}$.
Utilisation de l'aléatoire en algorithmique :

- Éviter les pires des cas d'exécution d'un algorithme
  → Tri rapide
- Échantillonner
  → Choix du pivot dans le tri rapide
- Hachage
  → Rabin-Karp
  → Filtre de Bloom
- Structure de données aléatoire (tarbres (treaps), liste de décomposition (split lists))
- Casser la symétrie
- etc.

5.2 Un premier exemple : recherche dans un tableau binaire

Problème (Recherche binaire)
Entrée : T un tableau binaire de longueur paire n qui contient $\frac{n}{2}$ éléments 0 et $\frac{n}{2}$ éléments 1.
Sortie : i $\in \{1, .., n\}$ tel que T[i] contient un 1.

5.2.1 Algorithme déterministe

Proposition : Il existe un algorithme déterministe qui résout le problème de recherche binaire en regardant $\frac{n}{2}$ cases dans le tableau.
Démonstration :

```
1 pour i := 1 → n/2 faire
2 |  si T[i] == 1 alors
3 |  |  retourner i
4 |  fin
5 fin
6 retourner n/2 + 1
```

Proposition : Il n'existe pas d'algorithme déterministe qui résout le problème de recherche binaire en regardant moins de $\frac{n}{2}$ - 1 cases dans le tableau.
Démonstration :
Pour démontrer cette borne inférieure (de complexité), on utilise la méthode de l'adversaire.
Pour tout algorithme A qui tente de résoudre le problème de recherche binaire en $\frac{n}{2}$ - 1 accès au tableau, on construit un algorithme B qui construit une entrée pour A sur la quelle A va échouer.
A accède à certaines cases du tableau i_1, i_2, .., $i_{\frac{n}{2}-1} \in \{1, .., n\}$ (où la requête i_j peut prendre du contenu des cases T[i_1], T[i_2], ..., T[i_{j-1}]). L'algorithme B construit le tableau de sorte que T[i_j] = 0 pour j $\in \{1, .., \frac{n}{2} - 1\}$.
L'algorithme A retourne un indice i^* comme réponse et B peut fixer T[i^*] = 0 et l'algorithme A échoue sur l'entrée ainsi construite (qui vérifie les contraintes du problème).

5.2.2 Algorithmes probabilistes

Nous considérons les 2 algorithmes suivants :

Algorithme 1 : A_1

```
1 i ←R {1, .., n}
  /* on tire une valeur uniformément aléatoire et dans {1, .., n} et on assigne
     le résultat à i                                                          */
2 tant que T[i] == 0 faire
3 |  i ←R {1, .., n}
4 fin
5 retourner i
```

Algorithme 2 : A_2

```
1 c ← 0
2 i ←R {1, .., n}
3 tant que T[i] == 0 et c < k faire
4 |  i ←R {1, .., n}
5 |  c ← c + 1
6 fin
7 retourner i
```

Remarques :
L'algorithme A_1 retourne toujours un indice i tel que T[i] = 1 mais nous n'avons aucune garantie qu'il termine.
Notons X_i la variable aléatoire définie par :
- $X_i = 1$ si la boucle est exécutée au moins i fois
- $X_i = 0$ sinon

Le temps d'exécution de l'algorithme en moyenne (pour toute entrée) est :

$$\sum_{i=0}^{\infty} \Pr(X_i = 1) = \sum_{i=0}^{\infty} \frac{1}{2^i} = 2$$

Un tel algorithme (qui retourne toujours une solution et dont le temps d'exécution en moyenne est borné) est dit de type Las Vegas.
L'algorithme A_2 termine toujours en exécutant la boucle au plus k fois (indépendamment de ces choix aléatoires). Par contre, il retourne un indice i tel que T[i] = 0 (i.e. il se trompe) avec une probabilité $\frac{1}{2^{k+1}}$. Un tel algorithme est de type Monté Carlo.
Il existe d'autres types d'algorithmes probabilistes (notamment Atlantic City ou Bellagio).

5.3 Algorithme de sélection

Nous considérons les 2 problèmes suivants :
Problème : Sélection
Entrée :
— T un tableau d'entiers de longueur n ne contenant que des entiers 2 à 2 distincts.
— k ∈ 1, .., n
Sortie : L'élément T[i] de T qui est de rang k (i.e. tel que T contienne exactement k - 1 éléments inférieur à T[i]).
Problème : Sélection ϵ - approché
Entrée : identique
Sortie : Un élément T[i] de T qui est de rang compris dans l'intervalle [K(1 - ϵ), K(1 + ϵ)]].

5.3.1 Algorithmes déterministes

Une première approche pour résoudre ces 2 problèmes consiste à trier T et à retourne son k-ième élément. Avec un algorithme déterministe de tri optimal (par exemple le tri fusion) on obtient une complexité de l'ordre de O(nlogn) comparaisons.
On peut montrer par la méthode de l'adversaire que les 2 problèmes demandent au moins Ω(n) comparaisons.
Il existe un algorithme déterministe de complexité O(n) du à Blum, Floyd, Pratt, Rivest et Tarjan qui repose sur l'algorithme de sélection rapide due à Hoare (et que nous verrons dans la suite).

5.3.2 Algorithme probabiliste

Supposons qu'on veuille résoudre le problème de sélection approchée pour retourner un élément de rang dans l'intervalle $[\frac{\pi}{10}, \frac{9\pi}{10}]$.
Considérons l'algorithme suivant :

```
1 i ←R {1, .., n}
2 retourner T[i]
```

Cet algorithme s'exécute en temps constant et retourne un élément dont le rang est effectuant dans $[\frac{\pi}{10}, \frac{9\pi}{10}]$ avec une probabilité $\frac{8}{10} = \frac{4}{5}$. Il est donc de type Monté Carlo.
Pour améliorer cette probabilité de succès, il suffit de répéter l'algorithme plusieurs fois et de retourner l'élément médian des élément obtenus.
Soit k un paramètre (qui peut éventuellement déprendre de n). Considérons l'algorithme suivant :

```
1 L ← φ
2 I ← φ
3 pour j := 1 → k faire
4 |   i ←R {1, .., n} \ I
5 |   L ← L ∪ T[i]
6 fin
7 Trier L
8 retourner L[⌊k/2⌋]
```

Cet algorithme est de complexité O(klogk) (où O(k) avec l'algorithme de Blum et al.).
Il reste à calculer sa probabilité de succès.
Considérons les cas d'échec de l'algorithme en regardant les positions des éléments T[i] pour i ∈ I dans le tableau T' qui contient les éléments de T.
L'algorithme échoue si parmi les éléments tirés au hasard, il y a plus de $\frac{k}{2}$ éléments dans les éléments de T'[1 ... $\frac{n}{10}$] ou plus de $\frac{k}{2}$ dans les éléments de T'[$\frac{9n}{10}$ ... n].

Estimons la probabilité du premier événement :

$$\begin{aligned} P &= \sum_{j=\frac{k}{2}}^{k} \binom{k}{j} \left(\frac{1}{10}\right)^j \left(\frac{9}{10}\right)^{k-j} \\ &\leq \sum_{j=\frac{k}{2}}^{k} \binom{k}{\frac{k}{2}} \left(\frac{1}{10}\right)^j \left(\frac{9}{10}\right)^{k-j} \end{aligned}$$

Remarque : $\binom{k}{\frac{k}{2}} \leq 4^{\frac{k}{2}} \Rightarrow 4^{\frac{k}{2}} = (1+1)^k = \sum_{j=0}^{k} \binom{k}{j}$

Donc :

$$\begin{aligned} P &\leq \sum_{j=\frac{k}{2}}^{k} 4^{\frac{k}{2}} \left(\frac{1}{10}\right)^j \left(\frac{9}{10}\right)^{k-j} \\ &\leq 4^{\frac{k}{2}} \sum_{j=\frac{k}{2}}^{k} \left(\frac{1}{10}\right)^j \left(\frac{10}{9}\right)^j \left(\frac{9}{10}\right)^j \left(\frac{9}{10}\right)^{k-j} \\ &= 2^k \left(\frac{9}{10}\right)^k \sum_{j=\frac{k}{2}}^{k} \left(\frac{1}{9}\right)^j \\ &= 2^k \left(\frac{9}{10}\right)^k \left(\frac{1}{9}\right)^{\frac{k}{2}} c \qquad \text{(c est indépendante de k)} \\ &= \left(\frac{2.9}{10.3}\right)^k \\ &= \left(\frac{3}{5}\right)^k \end{aligned}$$

Donc pour k suffisamment grand, la probabilité d'erreur peut être rendue aussi petite que possible. Par exemple, pour k = Θ(logn), on obtient une probabilité d'erreur de l'ordre n^α pour α constante (et un temps d'exécution de O(lognloglogn)).
Nous considérons maintenant un algorithme de type Las Vegas pour le problème de sélection (exacte). Il s'agit de l'algorithme de sélection rapide (de Hoare) (Quickselect).

Algorithme 3 : Algorithme Quickselect (T, n, k)

```
1  i ←R {1, .., n}
2  α ← T[i]      // α est le pivot
3  l ← 0
4  L ← φ
5  R ← φ
6  pour j := 1 → n (j ≠ i) faire
7  |  si T[j] < α alors
8  |  |  l ← l + 1
9  |  |  L ← T[j]
10 |  sinon
11 |  |  R ← T[j]
12 |  fin
13 fin
14 si l = k - 1 alors
15 |  retourner α
16 fin
17 si l < k - 1 alors
18 |  retourner Quickselect(R, n - l - 1, k - l - 1)
19 fin
20 si l > k - 1 alors
21 |  retourner Quickselect(L, l , k)
22 fin
```

Si l'algorithme fait des "mauvais" choix de pivot (par exemple L = ϕ à chaque itération), le coût de l'algorithme peut-être en $\Theta(n^2)$.
Si l'algorithme a de la chance, on va couper en 2 à chaque fois et la complexité est en O(n).
Notons C(n) le nombre de comparaisons effectué en moyenne par Quickselect (pour tous les tableaux de longueur n).
Proposition : C(n) $\leq$ 4n
Démonstration : Par récurrence

- C(1) = 0, donc l'hypothèse est vérifiée pour n = 1
- Nous avons C(n) = n - 1 + $\sum_{t=1}^{n}$ Pr(α est de rang t).max(C(t - 1), C(n - t)

$$
\begin{aligned}
C(n) &= n - 1 + \sum_{t=1}^{n} \frac{1}{n}.C(max(t-1, n-t)) \\
&= n - 1 + \frac{1}{n} \sum_{t=\frac{n}{2}}^{n-1} 2C(t) \\
&= n - 1 + \frac{2}{n} \sum_{t=\frac{n}{2}}^{n-1} 4t \\
&= n - 1 + \frac{8}{n} \left(\frac{1}{8} n(3n-2) \right) \\
&= 4n - 3 \\
&\leq 4n
\end{aligned}
$$

Quickselect retourne toujours le k-ième élément et a un temps d'exécution espéré en O(n) (4n comparaisons). Il est de type Las Vegas.

6 Primalité des entiers

$\mathbb{P}$ désigne l'ensemble des nombres premiers.
$\mathbb{N} \setminus (\mathbb{P} \cup \{0,\ 1\})$ est l'ensemble des nombres dits composés.

6.1 Test de Fermat

Rappel :
Lemme : Soit n $\in \mathbb{N}$, n ≥ 2. L'entier n est premier si et seulement s'il existe un entier a $\in \mathbb{N}$, a ≥ 2 tel que :

- $a^{n-1} \equiv 1[n]$
- Pour tout diviseur premier p de n - 1 : $a^{\frac{n-1}{p}} \not\equiv 1[n]$

$\Rightarrow$ Ce lemme permet de démontrer que le problème de la primalité est dans NP.
Lemme (Petit théorème de Fermat) :
Soit p $\in \mathbb{P}$, pour tout entier a premier avec p, nous avons $a^{p-1} \equiv 1[p]$
Remarque : Dans un groupe $\mathbb{G}$ muni d'une loi noté multiplicativement, étant donné g $\in \mathbb{G}$ et x $\in \mathbb{N}$, on peut calculer g^x = g * ... * g (x fois) en O(log x) opérations dans le groupe.
Un algorithme récursif simple avec cette complexité consiste à utiliser l'égalité suivante :

$$g^x = \begin{cases} 1_{\mathbb{G}} & \text{si x} = 0 \\ (g^2)^{\frac{x}{2}} & \text{si x} \neq 0 \text{ est pair} \\ g.(g^2)^{\frac{x-1}{2}} & \text{sinon} \end{cases}$$

Pour vérifier l'égalité de Fermat, il faut calculer a^{p-1} dans $(\mathbb{Z}/p\mathbb{Z})^*$, ce qui se fait en O(logp) multiplications dans $(\mathbb{Z}/p\mathbb{Z})^*$. Chaque opération dans $(\mathbb{Z}/p\mathbb{Z})^*$ a un coût en O(log^2 p) opérations binaires (par les algorithmes de multiplication et de division).
Le coût total est en O(log^3 p) opérations élémentaires.
Corollaire : Soit n $\in \mathbb{N}$, n ≥ 2. S'il existe un entier a $\in \{2, .., n - 1\}$ tel que $a^{n-1} \not\equiv 1[n]$, alors n est composé.

Algorithme - Test de Fermat en base a

Entrée : n $\in \mathbb{N}$, n ≥ 2
Sortie : COMPOSÉ ou PROBABLEMENT PREMIER

```
1 b ← a^{n-1}[n]
2 si b = 1 alors
3 |  retourner COMPOSÉ
4 sinon
5 |  retourner PROBABLEMENT PREMIER
6 fin
```

Définition : Un entier composé n $\in \mathbb{N}$, n ≥ 2 est dit pseudo-premier de Fermat en base a (pour a $\in \mathbb{N}$, a ≥ 2) si $a^{n-1} \equiv 1[n]$.
Remarque : Le test de Fermat retourne "PROBABLEMENT PREMIER" sur un tel entier.
Exemples : Les premiers nombres pseudo-premiers de Fermat en base 2 sont 341, 561, 645, 1105, 1387, 1729, 1905, etc.
Proposition : Soit a ≥ 2 un entier. Il existe une infinité de nombre pseudo-premiers de Fermat en base a.
Démonstration : Soit p un nombre premier impair ne divisant pas a^2 - 1. Nous allons montrer que : n = $\dfrac{a^{2p}-1}{a^2-1}$ est pseudo-premier de Fermat en base a.
Nous avons : n = $\dfrac{a^{2p}-1}{a^2-1} = \underbrace{\dfrac{a^p-1}{a-1}}_{\in\mathbb{N}}.\underbrace{\dfrac{a^p+1}{a+1}}_{\in\mathbb{N}}$

a - 1 divise $a^p - 1$ car 1 est la racine de $a^p - 1$ (respt. a + 1 divise $a^p + 1$).
Donc n $\in \mathbb{N}$ et n est composé.
Nous voulons démontrer que $a^{n-1} \equiv 1[n]$.
n - 1 = $\dfrac{a^{2p-1}}{a^2-1}$ - 1 = $\dfrac{a^{2p}-a^2}{a^2-1}$
Nous allons montrer que 2p divise n - 1.
Nous avons $a^p \equiv a[p]$ par le petit théorème de Fermat, donc $a^{2p} \equiv a^2[p]$ et p divise $a^{2p} - a^2$ mais p ne divise pas a^2 - 1 donc p divise n - 1.
n - 1 = $\dfrac{a^{2p}-a^2}{a^2-1} = a^{2p-2} + a^{2p-4} + \ldots + a^2 = \sum_{i=1}^{p-1} a^{2i}$
n - 1 est la somme de (p - 1) termes de même parité (celle de a) donc n - 1 est pair.
Donc 2p divise n - 1, n - 1 = 2p - k pour k $\in \mathbb{N}$.
Pour conclure, nous avons a^{2p} - 1 = n(a^2 - 1) et $a^{2p} \equiv 1[n]$ donc $a^{n-1} = a^{2pk} = (a^{2p})^k = 1^k = 1$ [n] et il existe bien une infinité de nombres pseudo-premiers de Fermet en base a.
Définition : Un entier n $\in \mathbb{N}$, n ≥ 2 composé est un nombre de Carmichael s'il est pseudo-premier de Fermat en toute base a première avec n.
Théorème (Alford-Crandall-Pomerance 1994) : Il existe une infinité de nombres de Carmichael.
Exemple : 561 est le plus petit nombre de Carmichael.
Remarque : Il existe 44706 nombres de Carmichael inférieurs à 10^{14} (alors que le théorème des nombres premiers affirme qu'il y a $\dfrac{10^{14}}{ln 10^{14}} \backsimeq 3.10^{12}$ nombres premiers inférieurs à 10^{14}).

6.2 Test de Miller-Rabin

Lemme : Soient p $\in \mathbb{P}$ impair, a un entier premier avec p. Nous avons $a^{\frac{p-1}{2}} \equiv \pm 1[p]$.
Démonstration : $(a^{\frac{p-1}{2}})^p \equiv a^{p-1} \equiv 1[p]$
Comme p est premier $(\mathbb{Z}/p\mathbb{Z})^*$ est un corps donc les seules racines carrées de 1[p] sont 1 et -1.
Plus généralement, nous avons le lemme :
Lemme : Soient p $\in \mathbb{P}$, x un entier si $x^2 \equiv 1[p]$ alors $x \equiv \pm 1[p]$.
Proposition : Soit p $\in \mathbb{P}$ impair et notons p = 1 + 2^hm avec m impair. Soit a $\in \mathbb{N}$, a ≥ 2 premier avec p et considérons la suite définie par :
- $b_0 \equiv a^m[p]$
- $b_i \equiv b_{i-1}^2[p]$ pour i $\in$ {1, .., h}

Nous avons $b_h \equiv 1[p]$ et si $b_0 \not\equiv 1[p]$ il existe i $\in$ {0, .., h - 1} tel que $b_i \not\equiv 1[p]$
Remarque :
$b_i \equiv a^{m2^i}[p]$
$b_h \equiv a^{m2^h} \equiv a^{p-1} \equiv 1[p]$
Démonstration : Nous avons $b_h \equiv 1[p]$ et pour tout i $\in$ {0, .., h - 1} b_i est la racine carrée de b_{i+1}. Si $b_0 \equiv 1$, par le lemme précédent une de ces racines carrées vaut -1[p].
Corollaire : Soit n $\in \mathbb{N}$ impair et posons n = 1 + 2^hm avec m impair. S'il existe a $\in$ {2, .., n -1} tel que pour la suite définie par $b_0 \equiv a^m[n]$ et $b_i \equiv b_{i-1}^2[n]$ pour i $\in$ {1, .., h} nous avons $b_h \neq 1[n]$ ou $b_0 \neq 1[n]$ et $\forall$i $\in$ {0, .., h - 1} $b_i \not\equiv -1[n]$, alors n est composé.
Démonstration : Il s'agit de la contraposée de la proposition précédente.

6.3 Algorithme - Test de primalité Fort en base a

Entrée : n $\in \mathbb{N}$ impair, n = 1 + 2^hm
Sortie : COMPOSÉ ou PROBABLEMENT PREMIER

```
1  b ← a^M[n]
2  si b = 1 ou b = -1 alors
3  |  retourner PROBABLEMENT PREMIER
4  sinon
5  |  pour i := 1 → h - 1 faire
6  |  |  si b ≢ −1[n] et b^2 ≡ 1[n] alors
7  |  |  |  retourner COMPOSÉ
8  |  |  fin
9  |  |  b ← b^2[n]
10 |  |  si b ≢ 1[n] alors
11 |  |  |  retourner COMPOSÉ
12 |  |  sinon
13 |  |  |  retourner PROBABLEMENT PREMIER
14 |  |  fin
15 |  fin
16 fin
```

Définition : Un entier n ∈ ℕ impair composé sur lequel l'algorithme précédent retourne "PROBABLEMENT PREMIER" est dit pseudo-premier fort en base a.
Remarque : Si un nombre est pseudo-premier fort en base a, il est pseudo-premier de Fermat en base a.
Théorème (admis) : Pour tout a ≥ 2, il existe une infinité de nombres pseudo-premiers forts en base a.
Théorème (admis) : Soit n ∈ ℕ impair, le nombre d'entiers a ∈ $(\mathbb{Z}/n\mathbb{Z})^*$ pour lesquels n est pseudo-premier fort en base a inférieur à $\frac{n-1}{4}$.

6.4 Algorithme - Test de primalité de Miller-Rabin

Entrée : n ∈ ℕ impair
Sortie : COMPOSÉ ou PREMIER

```
1  pour i := 1 → T faire
2  |  Tirer a ← (ℤ/nℤ)* uniformément aléatoirement
3  |  si Test de primalité Fort (n, a) = COMPOSÉ alors
4  |  |  retourner COMPOSÉ
5  |  fin
6  fin
7  retourner PREMIER
```

Cet algorithme probabiliste de type Monté Carlo a un temps d'exécution en O(Tlog^3p) opérations élémentaires. Lorsqu'il retourne "COMPOSÉ", l'entier n est certainement composé et il retourne "PREMIER" lorsque n est composé avec une probabilité inférieur à $(\frac{1}{4})^T$.
Remarque : Le problème de primalité a été prouvé appartenir à la classe P en 2001 (publié en 2004) par Agrawal-Kayal-Saxena. Leur algorithme repose sur l'identité : $n \in \mathbb{P} \iff (X+a)^n \equiv X^n + a[n]$ pour a ∈ $(\mathbb{Z}/n\mathbb{Z})^*$.

7 Test d'identité polynomial

Dans ce chapitre, nous allons étudier le problème de décision qui étant donnés 2 polynômes P et Q en n variables définis sur un corps $\mathbb{K}$, retourner 1 si les 2 polynômes sont égaux et 0 sinon.
Nous supposons que les algorithmes ont un accès en boîte noire à ces 2 polynômes et qu'ils obtiennent en temps constant la valeur de ces polynôme en tout point de $\mathbb{K}^n$ de leur choix.

Remarque : On veut tester si P et Q sont égaux en tant que polynômes. Attention, dans $\mathbb{Z}/2\mathbb{Z}$, $X^2 + X$ vaut 0 en tout x $\in \mathbb{Z}/2\mathbb{Z}$ mais $X^2 + X \neq 0$.
Exemple :

$$\det \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{pmatrix} = (x_2 - x_1)(x_3 - x_2)(x_3 - x_1) \qquad \text{sur n'importe quel corps } \mathbb{K}$$

En particulier sur $\mathbb{Z}/5\mathbb{Z}$, prenons $x_1 = 1$, $x_2 = 2$, $x_3 = 4$.
Nous avons $(x_2$ - $x_1)(x_3$ - $x_2)(x_3$ - $x_1) = 1.3.2 = 6 \equiv 1[5]$ et

$$\begin{vmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 4 & 1 \end{vmatrix} = \begin{vmatrix} 1 & 1 & 1 \\ 0 & 1 & 3 \\ 0 & 3 & 0 \end{vmatrix} = \begin{vmatrix} 1 & 3 \\ 3 & 0 \end{vmatrix} = -3.3 = -9 \equiv 1[5]$$

Polynôme en 101 variables de degré 5050 avec 101 ! termes :

$$\det \begin{pmatrix} 1 & x_1 & \dots & x_1^{100} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{101} & \dots & x_{101}^{100} \end{pmatrix} = \prod_{1 \leq j < i \leq 101} (x_i - x_j)$$

Pour résoudre le problème de l'identité de polynôme, nous allons utiliser un théorème du à Schwartz et Zippel (The Curious History of the Schwartz-Zippel(-DeMillo-Lipton) Lemma, *blog de Richard Lipton*) en 1978 - 1979.

Théorème (Lemme de Schwartz-Zippel) :

Soient $\mathbb{K}$ un corps, n $\in \mathbb{N}$, n ≥ 1 et S un sous-ensemble fini de $\mathbb{K}$.
Soit P un polynôme à coefficents dans $\mathbb{K}$ à n variables, P $\in \mathbb{K}[x_1, .., x_n]$ non nul, alors :

$$\underset{(x_1,..,x_n) \in S}{Pr}(P(x_1, .., x_n) = 0) \leq \frac{degP}{\#S}$$

où degP désigne le degré totem du polynôme P.
Remarque : Le résultat s'écrit également :

$$\#\{(x_1, .., x_n) \in S^n, P(x_1, .., x_n) = 0\} \leq \frac{degP}{(\#S)^{1-n}}$$

Démonstration : Nous allons montrer le résultat par récurrence sur n :

- Considérons le cas n = 1
  Le polynôme P $\in \mathbb{K}[X_1]$ est un polynôme en une variable définit sur un corps $\mathbb{K}$ et P a donc au plus degP racine, donc :
  $$\#\{x_1 \in S, P(x_1) = 0\} \leq \#\{x_1 \in \mathbb{K}, P(x_1) = 0\} \leq degP = \frac{degP}{(\#S)^{1-n}}$$
- Supposons le résultat montré pout tout entier k < n avec n ≥ 2. On veut montrer pour l'entier n.
  Soit P $\in \mathbb{K}[x_1, .., x_n]$ un polynôme non nul. Si P n'est pas un polynôme constant (auquel cas le résultat est immédiat car P = c $\neq 0$ n'a aucune racine), il existe une variable apparaissant dans P avec un coefficient non nul. Nous pouvons supposer sans perdre de généralité qu'il s'agit de x_1.
  Nous pouvons écrire :
  $$P(x_1, .., x_n) = \sum_{i=0}^{k} f_i(x_2, .., x_n) x_1^i$$

Remarque : Pour $P(x_1, x_2, x_3) = x_1^3 x_2 + 2x_1^2(x_2 + x_3) - x_1 + x_3^3$ de degré total 4, on a :

$P(x_1, x_2, x_3) = f_0(x_2, x_3) + f_1(x_2, x_3)x_1 + f_2(x_2, x_3)x_1^2 + f_3(x_2, x_3)x_1^3$ avec $f_0(x_2, x_3) = x_3^3$
$f_1(x_2, x_3) = -1$
$f_2(x_2, x_3) = 2(x_2 + x_3)$
$f_3(x_2, x_3) = x_2$

Dans cette écriture, nous avons k ≤ n et degf_k ≤ degP - k = n - k
Nous considérons les éléments $(x_1, .., x_n) \in S^n$ tels que P$(x_1, .., x_n) = 0$.
Soit $(x_2^*, .., x_n^*) \in S^{n-1}$.
→ Si $f_2(x_2^*, .., x_n^*) \neq 0$, les $x_1 \in S$ tels que $P(x_1, x_2^*, .., x_n^*) = 0$ sont polynôme $P^*(x_1) = \sum_{i=0}^{k} f_i(x_2^*, .., x_n^*)x_1^i$ qui est degré k. Le nombre de x_1 par $(x_2^*, .., x_n^*)$ fixé est inférieur à k.
→ Le nombre de $(x_2^*, .., x_n^*) \in S^{n-1}$ tel que $f_k(x_2, .., x_n) = 0$ est inférieur à $\frac{degf_k}{\#S^{1-(n-1)}} \leq \frac{n-k}{\#S^{1-(n-1)}}$ par hypothèse de récurrence sur f_k polynôme en (n - 1) variables de degré au plus n - k.
Donc le nombre de $(x_1, .., x_n) \in S^n$ tels que $P(x_1, .., x_n) = 0$ est inférieur à

$$k.\#\{(x_2, .., x_n) \in S^{n-1}, f_2(x_2, .., x_n) \neq 0\} + \#\{x_1 \in S\}.\#\{(x_2, .., x_n) \in S^{n-1}, f_2(x_2, .., x_n) = 0\}$$
$$\leq k.(\#S)^{n-1} + (\#S).\frac{(n-k)(\#S)^{n-1}}{(\#S)^1}$$
$$\leq n.(\#S)^{n-1} = \frac{degP}{(\#S)^{n-1}}$$
$$\#\{(x_1, .., x_n) \in S^n, P(x_1, .., x_n) = 0\} \leq degP(\#S)^{n-1}$$
$$\Pr_{(x_1,..,x_n)\in S^n}(P(x_1, .., x_n) = 0) \leq \frac{degP}{\#S}$$

Ce résultat est donc un algorithme déterministe pour décider si un polynôme est normal sur un corps $\mathbb{K}$. On choisit un ensemble S de $\mathbb{K}$ avec #S > degP et in choisit degP$(\#S)^{n-1}$ + 1 vecteurs dans S^n (de façon arbitraire). Si le polynôme s'annule en tous ces points, il s'agit du polynôme nul. Cet algorithme est exponentiel en la taille de l'entrée.
Il est possible d'utiliser un algorithme de type Monté Carlo qui consiste à prendre en ensemble S (avec #S > degP) et à tirer uniformément aléatoirement des vecteurs dans S^n un nombre T de fois. La probabilité que S s'annule en tous ces points si P est non nul est inférieure à $(\frac{degP}{\#S})^T$ et peut donc être rendu aussi petite que souhaitée.

8 Problème MAX 3-SAT

Dans le problème 3-SAT, une entrée est donnée par une liste de clauses qui sont les disjonctions de 3 termes (appelés littéraux) définit comme étant une variable x_i ou sa négation $\overline{x_i}$. La formule logique est la conjonction de ces clauses et elle est dite satisfiable s'il existe un choix booléen pour toutes les variables rendant la formule vraie.
Exemple :

$c_1 = x_1 \vee \overline{x_2} \vee x_3$
$c_2 = x_1 \vee x_3 \vee \overline{x_4}$
$c_3 = \overline{x_1} \vee x_2 \vee \overline{x_4}$
$c_4 = x_1 \vee \overline{x_3} \vee x_4$

Avec $x_1 = x_3 = 0, x_2 = x_4 = 1$, c_1 n'est pas vérifiée mais c_2, c_3, c_4 sont vérifiées. Mais $x_1 = x_2 = 1$ rendent vraies les 4 clauses et donc la formule est satisfiable.
Ce problème MAX 3-SAT consiste à rechercher un choix des variables qui satisfait le maximum de clauses dans la formule.

Le problème MAX 3-SAT est plus difficile que 3-SAT. Nous allons étudier en algorithme probabiliste polynomial qui retournera une approximation de MAX 3-SAT.
Lemme : Soit $\phi = c_1 \wedge .. \wedge c_p$ une formule 3-SAT où c_i pour i $\in \{1, .., p\}$ est une clause. Notons $x1, .., x_n$ les variables de ϕ.
Une affectation aléatoire uniforme des n variables permet de satisfaire en moyenne $\frac{7p}{8}$ clauses.
Démonstration : Il suffit d'utiliser la linéarité de l'espérance. Notons X_i la variable aléatoire définie par :

$$X_i = \begin{cases} 1 & \text{si la clause } c_i \text{ est satisfaite} \\ 0 & \text{sinon} \end{cases}$$

La clause c_i est une disjonction de 3 littéraux qui est satisfaite dès que l'un des 3 littéraux est vrai. Elle n'est pas donc pas satisfaite que si les 3 littéraux sont faux ce qui a une probabilité $\frac{1}{8}$.
La probabilité que la clause c_i soit satisfaite est 1 - $\frac{1}{8} = \frac{7}{8}$.
Notons X la variable aléatoire correspondant au nombre de clauses satisfaites. Nous avons :

$$X = X_1 + .. + X_p$$

et

$$\mathbb{E}(X) = \mathbb{E}(X_1 + .. + X_p) = \mathbb{E}(X_1) + .. + \mathbb{E}(X_p) = \sum_{i=1}^{p} \mathbb{E}(X_i)$$

et

$$\mathbb{E}(X_i) = 1.Pr(X_i = 1) + 0.Pr(X_i = 0) = \frac{7}{8} \text{ pour tout i} \in \{1, .., p\}$$

donc $\mathbb{E}(X) = \frac{7p}{8}$.
Corollaire : Une formule 3-SAT avec p clauses admet toujours un choix de variables qui satisfait au moins $\frac{7p}{8}$ de ces clauses.
Démonstration : En effet, si le nombre de clauses satisfaites par toutes las affectations était strictement inférieur à $\frac{7p}{8}$, alors l'espérance du lemme précédent serait strictement inférieur à $\frac{7p}{8}$.
Corollaire : Toute formule 3-SAT ayant moins de 7 clauses est satisfiable.
Nous allons construire un algorithme pour trouver un tel choix d'affectation (satisfaisant au moins $\frac{7p}{8}$ clauses).

Algorithme de Johnson

Entrée : Une formule 3-SAT à p clauses et n variables.
Sortie : Une affectation des variables satisfait $\geq \frac{7p}{8}$ clauses.

---

1 Choisir une affectation des variables uniformément aléatoirement.
2 **tant que** *Le nombre de clauses satisfaites par ce choix est* $\leq \frac{7p}{8}$ **faire**
3 | Choisir une affectation des variables uniformément aléatoirement.
4 **fin**
5 **retourner** Ce choix

L'algorithme est de type Las Vegas car la réponse retournée sera toujours correcte. Il reste à estimer son temps d'exécution.
Notons α la probabilité qu'une affectation aléatoire satisfasse $\geq \frac{7p}{8}$ clauses.
Notons α_j la probabilité qu'une affectation aléatoire satisfasse exactement j clauses (pour j $\in \{0, .., p\}$).
Avec les notations de la démonstration du lemme, nous avons :

$$\begin{aligned} \frac{7p}{8} = \mathbb{E}(X) = \sum_{j=0}^{p} j.\alpha_j &= \sum_{1 \leq j \leq \frac{7p}{8}} j.\alpha_j + \sum_{\frac{7p}{8} \leq j \leq p} j.\alpha_j \\ &\leq (\frac{7p}{8} - \frac{1}{8}) \underbrace{\sum_{1 \leq j \leq \frac{7p}{8}} \alpha_j}_{\leq 1 \text{ (car c'est la somme des probabilité)}} + p \sum_{\frac{7p}{8} \leq j \leq p} \alpha_j \end{aligned}$$

donc $\frac{7p}{8} \leq (\frac{7p}{8} - \frac{1}{8}).1 + p.\alpha$
donc $\alpha \geq \frac{1}{8p}$.
Il reste à estimer le temps d'exécution de l'algorithme de Johnson. Notons Y la variable aléatoire correspondant au nombre de fois où la condition de la boucle "tant que" est testée.

$$\begin{aligned}\mathbb{E}(Y) &= \sum_{i=1}^{\infty} i.Pr(\text{les i - 1 premiers choix ne conviennent pas} \\ &\qquad\qquad \text{et le i-ème convient dans la condition du tant que}) \\ &= \sum_{i=1}^{\infty} i.(1-\alpha)^{i-1}.\alpha \\ &= \alpha \sum_{i=0}^{\infty} i.(1-\alpha)^{i-1} \\ &= \alpha \frac{1}{\alpha^2} = \frac{1}{\alpha}\end{aligned}$$

et $\mathbb{E}(Y) = 8p$.
Donc le nombre d'itérations est linéaire en le nombre de clauses.

9 Complexité randomisée (BPP, RP, co-RP, ZPP)

9.1 Machine de Turing probabiliste

Définition : Une machine de Turing probabiliste (MTP) est un 8-uplet $(Q, \Sigma, \Gamma, \delta_0, \delta_1, q_0, q_{acc}, q_{rej})$ où Q, Σ, Γ sont des ensembles finis (appelés respectivement ensemble d'état, alphabet du langage et alphabet de bande) avec $\Gamma \supseteq \Sigma \cup \{\square\}$.

— δ_0 et δ_1 sont des fonctions de transition :

$$\delta_0 : \text{Q x } \Gamma \longmapsto \text{Q x } \Gamma \text{ x } \{\triangleleft, \triangleright\}$$

— $q_0, q_{acc}, q_{rej} \in$ Q sont respectivement l'état initial, l'état d'acceptation, l'état de rejet.

Un calcul d'une telle MT est défini de manière analogue à celui d'une MT classique (où déterministe), sauf qu'à chaque étape du calcul, la fonction de transition à appliquer est tirée uniformément aléatoirement parmi $\{\delta_0, \delta_1\}$.
Pour une MTP M et une entrée x $\in \Sigma^*$, on note M(x) la variable aléatoire qui vaut 1 si M s'arrête dans l'état q_{acc} (et 0 si M s'arrête en q_{rej}) sur l'entrée x (pour les choix aléatoires de la fonction de transition).
Pour une MTP M, on dit M a pour temps d'exécution une fonction T : $\mathbb{N} \longmapsto \mathbb{N}$ telle que pour toute entrée x $\in \mathbb{Z}^*$, un calcul de M sur x se termine en au plus T(|x|) étapes. Une MTP est dite polynomiale si elle a pour temps d'exécution un polynôme.

9.2 Bounded Probabilistic Time (BPTIME) et Bounded Probabilistic Polynomial Time (BPP)

Définition : Soient T : $\mathbb{N} \longmapsto \mathbb{N}$ et L $\subseteq \Sigma^*$.
Une MTP M accepte le langage si et seulement si pour toute entrée x $\in \Sigma^*$ Pr(M(x) = L(x)) $\geq \frac{2}{3}$ où

$$L(x) = \begin{cases} 1 & \text{x} \in \text{L} \\ 0 & \text{sinon} \end{cases}$$

Dans ce cas, on dit que L appartient à la classe BPTIME(T) si T est un temps d'exécution de la machine M.
Nous notons BPP = $\bigcup_{k \geq 0}$ BPTIME (n $\longmapsto n^k$).

Remarque : Comme par la caractérisation de NP par des témoins/certificats, on peut également définir les MTP comme une machine à 2 rubans dont l'un est à lecture seule et ne contient que des bits dans {0, 1} et la transition s'effectue en lisant un bit de ce ruban d'aléatoire, le caractère du ruban classique, et le déplacement sur le ruban d'aléatoire se fait toujours vers la droite.
Si l'on note le contenu du ruban aléatoire r, on peur définir M(x, r) comme le calcul de M sur l'entrée x en utilisant l'aléatoire r et on peut définir de façon équivalente. BPTIME(T) comme l'ensemble des langages L tels qu'ils existe une MTP (à 2 rubans) avec un temps d'exécution T telle que
$\forall$ x $\in \Sigma^*$, $\underset{r \leftarrow \{0,1\}^{T(|x|)}}{Pr}(M(x,r) = L(x)) \geq \frac{2}{3}$
Exemples :

- Médian
- Primalité (Miller-Rabin)
- Identité polynomiale

9.3 Randomised Time (RTIME) et Randomised Polynomial Time (RP)

Définition : Avec les même notations, on définit la classe RTIME(T), la classe des langages L $\subseteq \Sigma^*$ pour lesquels il existe une MTP M de temps d'exécution T telle que, pour tout x $\in \Sigma^*$:

- Si x $\in$ L, Pr(M(x) = 1) $\geq \frac{2}{3}$
- Si x $\notin$ L, Pr(M(x) = 0) = 1

On note RP = $\bigcup_{k \geq 0}$ RTIME (n $\longmapsto n^k$)
Définition : L $\in$ co-RTIME(T) si et seulement si $\overline{L} = \Sigma^* \setminus$ L $\in$ RTIME(T).
Lemme :
RP $\subseteq$ BPP
co-RP $\subseteq$ BPP
P $\subseteq$ BPP
BPP $\subseteq$ EXP = $\bigcup_{k \geq 0}$ TIME (n $\longmapsto 2^{n^k}$)

9.4 "Zero-sided error" Time (ZTIME)

Définiton : Avex les mêmes notations ; on définit la calsse ZTIME(T), la classe des langages L $\subseteq \Sigma^*$ pour lesquels il existe une MTP M dont le temps d'exécution en moyenne est borné par T(|x|) sur toute entrée x et telle que pour tout x $\in \Sigma^*$: Pr(M(x) = L(x)) = 1
(autrement dit :

- Si x $\in$ L, Pr(M(x) = 1) = 1
- Si x $\notin$ L, Pr(M(x) = 0) = 1).

On note ZPP = $\bigcup_{k \geq 0}$ ZTIME (n $\longmapsto n^k$).
Théorème : ZPP = RP $\cap$ co-RP.
Démonstration : Soit L un langage de ZPP
Soit M une MT qui accepte L (sans erreur) en temps espéré polynomial T(n).
Nous considérons la MTP M'/M" qui réalise les mêmes calculs que M mais s'arrête dès qu'elle a exécuté 3T(n) étapes et dans ce cas elle passe dans l'état $q_{acc}/qrej$.
M' retourne une réponse correcte dès que M s'arrête en moins de 3T(|x|) étapes sur l'entrée x.
Rappelons l'inégalité de Markov : pour une variable réelle X, nous avons :

$$Pr(X \geq a) \leq \frac{\mathbb{E}(X)}{a}$$

En particulier considérons la variable réelle x correspondant au nombre d'étapes de calcul de M sur x.

$$Pr(x \geq 3T(|x|)) \leq \frac{\mathbb{E}(x)}{3T(|x|)} \leq \frac{T(|x|)}{3T(|x|)} = \frac{1}{3}$$

donc M' retourne toujours une réponse correcte avec une probabilité $\geq \frac{2}{3}$.
Si le temps d'exécution de M dépasse 3T(|x|), M' passe dans l'état q_{acc} et accepte donc tous les mots.
Nous avons donc pour x ∈ L :

$$\begin{aligned}
&Pr(M'(x) = 1) \\
=&Pr(M'(x) = 1|X < 3T(|x|)).Pr(X < 3T(|x|)) + Pr(M'(x) = 1|X \geq 3T(|x|)).Pr(X \geq 3T(|x|)) \\
=& \underbrace{1}_{\text{M' ne se trompe pas}}.Pr(X < 3T(|x|)) + \underbrace{1}_{\text{M' accepte}}.Pr(X \geq 3T(|x|)) \\
=&1
\end{aligned}$$

pour x ∉ L :

$$\begin{aligned}
&Pr(M'(x) = 0) \\
=&Pr(M'(x) = 0|X < 3T(|x|)).Pr(X < 3T(|x|)) + Pr(M'(x) = 0|X \geq 3T(|x|)).Pr(X \geq 3T(|x|)) \\
=& \underbrace{1}_{\text{M' ne se trompe pas}}.Pr(X < 3T(|x|)) + \underbrace{0}_{\text{M' accepte}}.Pr(X \geq 3T(|x|)) \\
=&Pr(X < 3T(|x|)) \geq \frac{2}{3}
\end{aligned}$$

Nous avons donc :
- x ∈ L, Pr(M'(x) = 1) = 1
- x ∉ L, Pr(M'(x) = 0) $\geq \frac{2}{3}$

donc L ∈ co-RP.
De même argument avec la machine M" (qui rejette tous les mots après un temps 3T) donne L ∈ RP.
$\Longrightarrow$ L ∈ RP ∩ co-RP.
Réciproquement, soit L ∈ RP ∩ co-RP, il existe 2 MTP polynomiales M et M' telles que :
- x ∈ L, Pr(M(x) = 1) $\geq \frac{2}{3}$ et Pr(M'(x) = 1) = 1
- x ∉ L, Pr(M(x) = 0) = 1 et Pr(M'(x) = 0) $\geq \frac{2}{3}$

Nous allons construire une MT M" qui exécute (plusieurs fois) M et M' simultanément sur la même entrée x de sorte que :
- Si M et M' retournent 1, M" retourne 1
- Si M et M' retournent 0, M" retourne 0
- Si M et M' retournent les valeurs 0 et 1, M" relance M et M' sur x (avec de nouveaux choix aléatoires).

Lorsque M' s'arrête, nous avons :

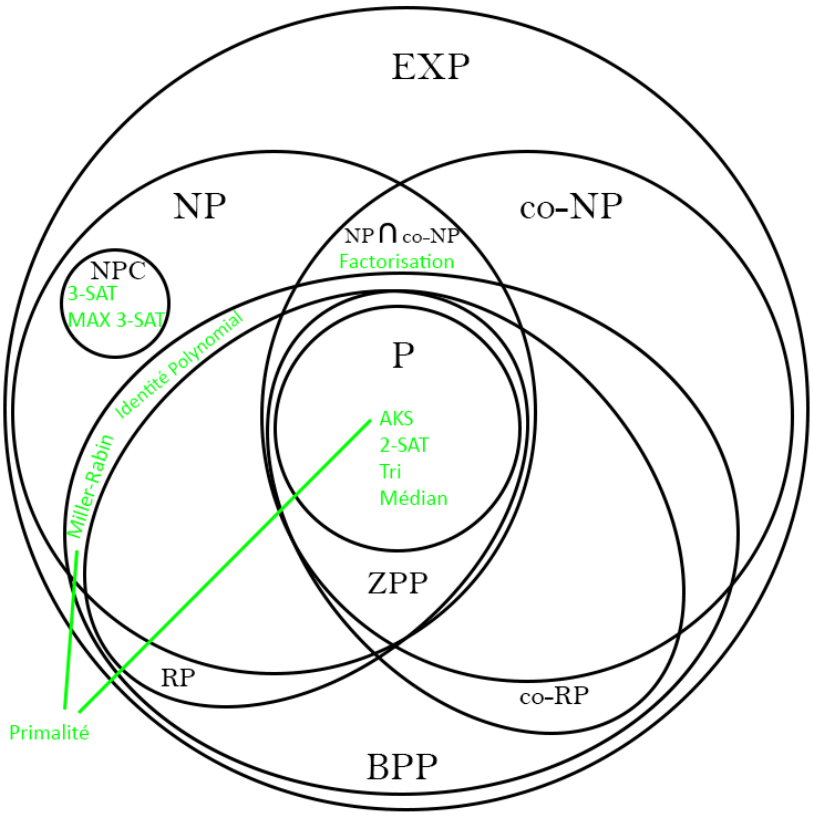
$$\text{Pr(M"(x) = L(x)) = 1}$$

Il reste à estimer le temps d'exécution de M".
Le 3ème cas de l'analyse précédente (M retourne 0 et M' retourne 1) se produit avec une probabilité au plus $\frac{1}{3}$ (indépendamment de x). Le nombre d'itérations moyen de M" est donc inférieur à $\frac{3}{2}^*$. Donc si M et M' ont un temps d'exécution polynomial, M" a un temps d'exécutions moyen polynomial.
Remarque (*) : Le nombre de répétitions moyen pour qu'un événement de probabilité p se produise

est $\frac{1}{p}$. En effet, si on note X la variable aléatoire comptant le nombre de répétitions :

$$\begin{aligned}
\mathbb{E}(X) &= \sum_{i=0}^{\infty} Pr(X = i).i \\
&= \sum_{i=0}^{\infty} (1-p)^{i-1}.p.i \\
&= p.\sum_{i=0}^{\infty} i.(1-p)^{i-1} \\
&= \frac{p}{1-p}.\frac{1-p}{p^2} \\
&= \frac{1}{p}
\end{aligned}$$

Remarque : Les définitions que nous venons de voir sont robustes. Les classes BPP, RP, co-RP ne sont pas modifiées si on remplace la constante $\frac{2}{3}$ par une constante c $> \frac{1}{2}$. Elle ne sont pas modifiées non plus si on change la source d'aléaoire (utilisé un tirage dans {1, .., n}) au lieu de {0, 1} ou une source d'aléatoire biaisée).
En effet, si on dispose d'une source d'aléatoire avec Pr(b = 0) = $\frac{2}{3}^p$ et Pr(b = 1) = $\frac{1}{3}^{1-p}$ pour produire des bits b, on peut construire une autre source d'aléatoire non biaisée en temps espéré constant. Pour cela on utilise l'extracteur de Von Neumann qui tire des bits selon la source biaisée 2 par 2 et s'il reçoit 01 il retourne 0, 10 il retourne 1, 00 ou 11 il recommence.
$\longrightarrow$ On peut définir les réduction randomisées (ou probabilistes) et ainsi définir de nouvelles classes de complexité comme : BPNB ={L, L $\leq_r$ 3-SAT}.



Relations entre les classes de complexité