1) Question 1

Un problème de décision est une question dont la solution est oui ou non. Intuitivement, un problème peut être défini par ses paramètres, on peut alors décrire le problème par l'ensemble des instances du problème tel que la réponse au problème est oui (instances positives). Ainsi, un alphabet peut être utilisé pour formaliser les instances positives du problème en créant un langage (donc un ensemble de mots), c'est-à-dire que tous les mots du langage amène 'oui' comme solution au problème de décision.

Pour formaliser ce problème, il faut d'abord définir un alphabet, par exemple :

$$\Sigma = \{[0-9], !, \#\}$$

Et représenter un graphe par un mot de type

$$\mathrm{w} = (\mathrm{N_x}!\,\mathrm{N_y}\#)^*$$

où les N sont des entiers représentant les sommets du graphes et '!' représentant l'existence d'une arrête de $\mathrm{N_x}$ à $\mathrm{N_y}$. Le caractère # est utilisé pour séparer les différents arcs.

Il faudrait alors créer un langage répondant au problème en utilisant ce formalisme. On peut définir les instances positives du problème comme le langage suivant :

$\mathrm{L} = \{\mathrm{w} \mid \mathrm{w} \in \{(\mathrm{N_x}!\,\mathrm{N_y}\#)^* \mathrm{s}!\,\mathrm{t}\}$ et tel qu'il y ait un chemin entre s et t$\}$

2) Question 2

La configuration d'une machine de Turing est définie par un triplet $(\mathrm{w}_1\ \mathrm{q}\ \mathrm{w}_2)$ où :

- w_1 est le mot se trouvant à gauche de la tête de lecture
- q est l'état de contrôle et
- w_2 est le mot à droite de la tête de lecture.

La computation est une séquence (finie ou infinie) de configurations. A chaque étape, on regarde ce que la tête lit σ et l'état actuel de la machine q. On regarde la fonction de transition $\delta(\mathrm{q}, \sigma)$ correspondant à ce tuple sous la forme de $(\mathrm{q}', \gamma, \mathrm{D})$ où :

- q' définit le prochain état de la machine
- γ définit ce qu'on écrit à l'emplacement de la tête de lecture
- D définit la direction dans laquelle bouger la tête de lecture après l'écriture (gauche ou droite).

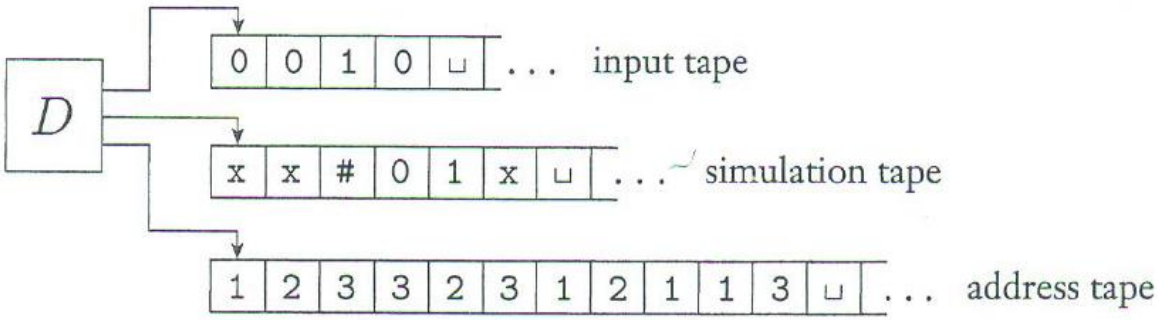
Par exemple, si la configuration courante est $(w_1 c, q, a w_2)$ où a et c sont des lettres et que (q', b, D) appartient à $\delta(q, a)$, alors :

- Si D = droite, la nouvelle configuration est $(w_1 cb, q', w_2)$
- Si D = gauche, la nouvelle configuration est (w_1, q', cbw_2)

Une computation est acceptante si elle a un nombre fini d'étapes et qu'à la fin de celle-ci, l'état de contrôle fait partie des états acceptants.

3) Question 3

Toute machine de Turing non-déterministe a un équivalent déterministe. En effet, l'exécution d'une machine de Turing non-déterministe peut se représenter comme un arbre parcouru en profondeur et si une des branches accepte, la machine accepte. Il suffit de faire une machine de Turing déterministe à 3 rubans : un ruban d'entrée, un ruban de simulation et un ruban d'adresse :



La TM déterministe simule le parcours de l'arbre ND en le parcourant en largeur pour mettre à jour pour chaque entrée les états auquel le NTM pourrait arriver.

De fait, la classe de problème résolvable par des machines de Turing non-déterministes est équivalant à la classe de problème résolvable par des machines de Turing détermistes à 3 bandes. De plus, n'importe quelle machine de Turing multi-ruban a un équivalent à un seul ruban. Du coup, la classe de problème résolvable par des machines de Turing non-déterministes est équivalant à la classe de problème résolvable par des machines de Turing déterministes à une bande.

4) Question 4

Une collection de mots w qui sont acceptés par une machine de Turing M forme le langage reconnu par M, à savoir L(M). Un mot w est accepté par M si une séquence d'états r_i existe dans Q tel que :

1. $r_0 = q_0$
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i = 0, \ldots, n-1$
3. $r_n \in F$

Un langage L est décidable s'il existe une machine de Turing M qui reconnait L et qui s'arrête quel que soit l'entrée.

La différence fondamentale est que les *recogniser* s'arrêtent sur toutes les instances positives du problème mais peuvent boucler à l'infini sur les instances négatives.

On utilise des *deciders* pour formaliser la notion d'algorithme car un algorithme doit avoir un nombre fini d'étape quelque soit son entrée, ce qui correspond à la définition des *deciders*.

5) Question 5

Church thesis:

La notion intuitive d'algorithme est égale aux algorithmes de machine de Turing.

On considère généralement qu'une méthode effective de calcul doit satisfaire aux obligations suivantes :

1. l'algorithme consiste en un ensemble fini d'instructions simples et précises qui sont décrites avec un nombre limité de symboles (l'alphabet d'une machine de Turing est limité).
2. l'algorithme doit toujours produire le résultat en un nombre fini d'étapes (définition des *deciders*)
3. l'algorithme peut en principe être suivi par un humain avec seulement du papier et un crayon ; (machine de Turing avec son/ses ruban(s) et têtes de lecture/écriture)
4. l'exécution de l'algorithme ne requiert pas d'intelligence de l'humain sauf celle qui est nécessaire pour comprendre et exécuter les instructions (la machine de Turing est configurée pour comprendre et exécuter les instructions).

Ce n'est pas un théorème car pour cela, il faudrait le démontrer, or on parle de notion d'algorithme qu'on formalise justement avec cette définition utilisant les TM.

6) Question 6

Les *finite automata* déterminent des langages réguliers (donc des langages déterminables par des expressions régulières) et les *pushdown automata* déterminent des *context-free language* (donc des langages définis par des *context-free grammar*).

- Les langages réguliers sont plus restrictifs que les langages décidables, par exemple $\{0^n1^n \mid n \in \mathbb{N}_0\}$n'est pas un langage régulier mais est un langage decidable. En effet, on ne peut pas construire de DFA pour ce langage mais on peut, par exemple, utilisé une TM à 2 rubans pour le déterminer.
- Les *context-free language* sont plus restrictifs que les langages décidables, par exemple $\{0^n1^n2^n \mid n \in \mathbb{N}_0\}$n'est pas un *context-free language* (il faudrait 2 stacks idéalement) mais est un langage decidable (TM à 3 bandes, lecture et 2 stacks). Par contre $\{0^n1^n \mid n \in \text{naturels}\}$ est un *context-free language* (comme dit plus haut, on peut utiliser une TM à 2 bandes, qui peut être équivalent à un PDA) et donc les *context-free language* ont plus d'expressivité.

7) Question 7

Les machines de Turing sont des ensembles d'états et de transitions finis, qu'on peut décrire, par exemple, sous forme binaire finie, et donc on peut dénombrer et ordonner les différentes machines de Turing. En effet, les mots binaires finis sont dénombrables car il existe une fonction permettant de ramener chaque nombre binaire à un nombre entier et donc par définition, les nombres binaires finis sont dénombrables, et donc les TM aussi.

8) Question 8

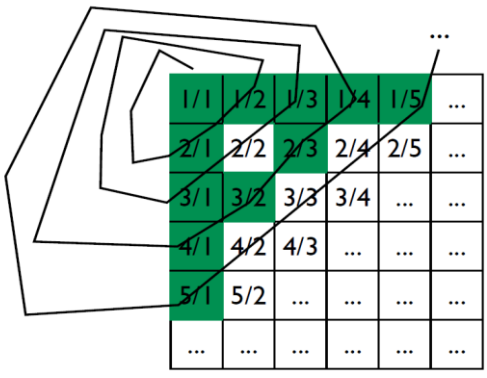
Par contradiction, imaginons que l'ensemble des réels est dénombrable, on peut donc faire un tableau regroupant tous les réels, cependant, on peut toujours rajouter un élément en plus dans cet ensemble en utilisant la diagonale de Cantor. Ce qui prouve que cet ensemble ne contient pas tous les réels. On construit le nombre x comme suit en considérant avoir regrouper tous les réels entre 0 et 1 :

$r_1 = 0,\mathbf{0}1051 10 \ldots$
$r_2 = 0,4\mathbf{1}32043 \ldots$
$r_3 = 0,82\mathbf{4}5026 \ldots$
$r_4 = 0,233\mathbf{0}126 \ldots$
$r_5 = 0,4107\mathbf{2}46 \ldots$
$r_6 = 0,99378\mathbf{1}8 \ldots$
$r_7 = 0,0105130 \ldots$
$\ldots$

Si la nième décimale de r_n est différente de 1, alors la nième décimale de x vaut 1, sinon elle vaut 2. Dans notre exemple on obtient $x = 0{,}1211121$ …. De fait, nous avons construit un nombre qui n'est pas dans la liste ($x \neq r_1$ car 1ère décimale différents,…). De fait, les nombres réels ne sont pas dénombrables.

9) Question 9

Pour qu'un ensemble soit dénombrable, il suffit d'avoir une technique pour dénombrer les éléments de cet ensemble, pour le cas des rationnels, on peut appliquer la méthode suivante :



10) Question 10

	M_0	M_1	M_2	...	M_n	...
w_0	1 ~~Ø~~	1	0	...	0	...
w_1	1	1 ~~Ø~~	1	...	0	...
w_2	1	1	1 ~~Ø~~	...	1	...
...	...	...	...	...	...	...
w_n	1	1	0	...	0 ~~1~~	...
...	...	...	...	...	...	...

Soit une machine M_i et le mot w_i. Si $L(M_i) = L_0$, alors si M_i est sensé accepté w_i, il est aussi, par définition de L_0, sensé ne pas l'accepté, ce qui est contradictoire. L_0 n'est donc pas reconnaissable par une machine de Turing.

11) Question 11

Par l'absurde : Soit A_{TM} est décidable, il existe donc un $H((< M >, w))$ tel que :

- il accepte si M accepte w
- il rejette si M n'accepte pas w

On construit alors la machine $D(< M >)$ sur base de H tel que :

- On execute $H((< M >, < M >))$
- On flippe la sortie de H, s'il accepte on rejette, et s'il rejette on accepte.

On fait alors $D(< D >)$:

- accepte si $D(< D >)$ rejette
- rejette si $D(< D >)$ accepte

De fait, on a une contradiction, et donc A_{TM} n'est pas décidable.

12) Question 12

A_{TM} est reconnaissable car il suffit d'appliquer M sur w et d'accepter si M accepte w, l'inverse sinon. A_{TM} n'est pas décidable (cf. question 11). Donc $RE \neq R$.

13) Question 13

Prouvons d'abord que si A est décidable, A est reconnaissable et co-reconnaissable. Si M est un décideur de A, créons une machine M' qui renvoie l'opposé de M. Cette machine est un codécideur de A. Comme $R \subseteq RE$, nous savons que A est reconnaissable et co-reconnaissable.

Ensuite, si nous possédons deux machines M (qui reconnait A) et M' (qui co-reconnait A), il nous suffit de créer une nouvelle machine H, qui fait tourner M et M' en parallèle, et qui accepte quand M accepte et rejette quand M' accepte. H est alors un décideur de A.

14)

Soit la TM M décidant $A_{DFA} = \{(B, w) | B \text{ is a DFA and accepts input string } w\}$:

Sur une entrée (B, w) où B est un DFA et w un string :

1. Simuler B sur w
2. Si la simulation s'arrête sur un état acceptant, accepte, sinon rejette.

M est un décideur pour A_{DFA} et donc les langages réguliers sont décidables car définis par des DFA.

Soit la TM S décidant $A_{CFG} = \{(G, w) | G \text{ is a CFG that generates string } w\}$:

Sur une entrée (G, w) où G est une CFG et w un string :

1. Convertir G en une grammaire équivalente en *Chomsky normal form*.
2. Liste toutes les dérivations avec $2n - 1$ étapes, sauf si $n = 0$, alors lister toute les dérivations à une étape.
3. Si une de ces dérivations génère w, accepte, sinon rejette

S est un décideur pour A_{CFG} et comme les CFL sont générées par des CFG, les CFL sont décidables.

15)

Une machine de Turing multi-rubans est une machine de turing ordinaire à plusieurs rubans. Toute TM multi-rubans peut être transformée en une machine mono-ruban en utilisant un caractère spécial pour séparer le ruban en plusieurs sous-rubans. On émule ainsi la présence de plusieurs rubans.

16)

Preuve par contradiction : imaginons qu'on ait R qui soit un décideur pour HALT : Créons la TM S((<M>,w)) tel que :

- On exécute R((<M>,w))
- Si R rejette, S rejette
- Si R accepte, on sait que M va s'arrêter sur w, et on peut le faire tourner sur w jusqu'à ce qu'il s'arrête.
    - Si M accepte, on accepte, si M refuse, on rejette.

Notons que S est donc un décideur pour $\mathrm{A_{TM}}$, ce qui est impossible.

17)

Preuve par contradiction : imaginons qu'on ait R qui soit un décideur pour REGULAR : Créons la TM S((<M>,w)) tel que :

- On construit la TM $\mathrm{M_2(x)}$ tel que
    - Si x est sous la forme 0^n1^n (donc pas régulier), on accepte
    - Si x n'est pas de la forme, on exécute M sur w, et on accepte si M accepte w.
- On exécute $\mathrm{R(M_2)}$

- Si R accepte, S accepte, sinon S rejette.

On peut voir que

- Si $(M, w) \notin A_{TM}$, alors $L(M_2) = \{0^n 1^n | n \in N\}$ et donc R rejette
- Si $(M, w) \in A_{TM}$, alors $L(M_2) = \Sigma^*$ et donc R accepte

On a donc que S est un décideur de A_{TM}.

18)

Preuve par contradiction : imaginons qu'on ait R qui soit un décideur pour EQ : Créons la TM S(<M>) tel que :

- On effectue $R((< M >, < M_1 >))$. M1 étant une machine qui rejette toutes les entrées.
- Si R accepte, S accepte, sinon S rejette.

On a ici un décideur pour E_{TM}.

19)

Une fonction de réduction d'un problème A vers un problème B est une fonction de $\Sigma^* \to \Sigma^*$, telle que si A accepte w, B accepte f(w), et si A refuse w, B refuse f(w). Si celle-ci existe, on dit que A est réductible à B. Ceci prouve que si B est décidable, A est aussi décidable, car il suffit d'utiliser B(f(w)) pour obtenir le décideur de A.

20)

$TIME(t(n))$ est la classe de complexité des langages décidables en un temps $O(t(n))$ par une machine de Turing M. $t(n)$ est le nombre maximal d'étapes par lesquelles M doit passer avant de s'arrêter sur une entrée de taille n, ce qui fait que $t(n)$ est le pire cas possible.

Les notations O sont utilisées pour approximer l'expression de la complexité temporelle, car elle est souvent complexe.

21)

$$f = O(g) \Leftrightarrow \exists c, n_0 \in N | \forall n \geq n_0 : f(n) \leq cg(n)$$

Ceci signifie que g est une borne asymptotique supérieure de f. Ceci nous permet de simplifier les fonctions de complexité en négligeant les complexités minimes des phases spécifiques de l'algorithme (par exemple l'initialisation).

$$f = o(g) \Leftrightarrow \lim_{n \to \infty} f(n)/g(n) = 0 \Leftrightarrow \exists c, n_0 \in N | \forall n \geq n_0 : f(n) < cg(n)$$

Ceci signifie que g est une borne asymptotique supérieure stricte de f.

Aucune fonction n'est un petit o d'elle-même, par contre toutes les fonctions sont des grand o d'elles-mêmes (car l'inégalité n'est pas stricte).

22)

Soit M une machine de Turing qui s'arrête quelle que soit l'entrée. La Complexité Temporelle est une fonction $f: N \to N$, telle que f(n) est le nombre maximal d'étapes par lesquelles M doit passer avant de s'arrêter sur une entrée de taille n.

Si M est une machine de turing non déterministe, alors nous pouvons créer une machine de Turing D qui effectue une recherche en largeur dans l'arbre de calcul de M. Comme la machine M s'arrête en t(n) opérations, la longueur de chaque branche est au maximum t(n). Le nombre de feuilles de l'arbre est donc au maximum de $b^{t(n)}$, où b est le nombre de choix possibles laissés par la fonction de transition de M).

Le temps nécessaire à D pour analyser tout l'arbre est donc le nombre de feuilles, multiplié par le temps qu'il faut pour parcourir une branche :

$$O(t(n) \times b^{t(n)}) = O(t(n)) \times b^{O(t(n))} = b^{O(t(n))} = 2^{O(t(n))}$$

Dans notre modèle, nous utilisons une machine à trois bandes, mais la complexité de cette machine à trois bande sera la même que celle d'une machine mono-bande :

$$2^{O(t(n))} = 2^{O(2t(n))}$$

23)

La classe P comprend tous les problèmes dont la complexité temporelle est sous forme polynomiale ($O(n^k)$). Quel que soit l'exposant k et le nombre d'étapes n, les différences seront relativement petites comparé à une complexité non polynomiale (du type $O(k^n)$).

$$P = \bigcup_{k \in N} TIME(n^k)$$

24)

$RELPRIME \in P$ car l'algorithme d'Euclide permet de résoudre le problème en un temps polynomial.

Nous avons une machine de Turing E((x,y)) telle que :

1. Tant que $y \neq 0$
    2. Faire $x \leftarrow x \bmod y$
    3. Echanger x et y
4. La réponse est x

Pour RELPRIME, nous pouvons dès lors utiliser une machine de Turing R((x,y)) telle que décrite:

- On effectue E((x,y))
- Si le résultat est 1, on accepte, sinon on rejette

Le nombre de fois que l'on peut rentrer dans la boucle de E est majoré par $O(2\log x)$ et $O(2\log y)$ (car une fois sur 2, x ou y est au minimum divisé par 2), ce qui fait que la limite est proportionnelle à l'encodage de x et y (binaire), et donc les étapes 2 et 3 s'exécute au maximum $O(n)$ fois. De plus, 2 et 3 s'exécutent en temps polynomial donc E est globalement polynomial.

25)

Soit s = min(p,q)

La complexité de cet algorithme est en $\mathrm{O(s)}$.

Soit n la taille de l'entrée, on a une complexité en $\mathrm{O}(2^{\mathrm{n}})$

Cet algorithme ne nous permet donc pas de prouver que $\mathrm{RELPRIME} \in \mathrm{P}$.

26)

Soit la grammaire G de type CFG générant le langage L de type CFL. Et soit S la variable de départ de la grammaire. Soit l'algorithme $D =$ Sur entrée $w = w_1 \dots w_n$:

1) If $w = \varepsilon$ and $S \to \varepsilon$ is a rule, accept
2) For $i = 1$ to n (examine tous les substrings de taille 1)
3) For each variable A :
4) If $A \to w_i$ is a rule
5) Place A in $table(i, i)$
6) For $l = 2$ to n (l longueur du substring)
7) For $i = 1$ to $n - l + 1$ (i point de depart du substring)
8) $j = i + l - 1$ (j position de fin du substring)
9) For $k = i$ to $j - 1$ (k position de séparation)
10) For each rule $A \to BC$
11) If $table(i, k)$ contains B and $table(k + 1, j)$ contains C
12) Place A in $table(i, j)$
13) If S is in $table(1, n)$, accept. Otherwise reject.

Chaque étape s'exécute en un temps polynomial. Les étapes 4 et 5 s'exécutent au plus nv fois où v est le nombre de variables dans G indépendant de n. Ces étapes s'exécutent donc en $O(n)$. L'étape 6 s'exécute au plus n fois. À chaque fois que l'étape 6 s'exécute, l'étape 7 s'exécute au plus n fois. À chaque fois que l'étape 7 s'exécute, les étapes 8 et 9 s'exécutent au plus n fois. À chaque fois que 9 s'exécute, 10 s'exécute au plus r fois, où r est le nombre de reègle dans G, indépendant de n. Au final, l'étape 11 s'exécute en $O(rn^3) = O(n^3)$Au total, D s'exécute en $O(n^3) + O(n) = O(n^3)$ étapes.

27)

1. La classe NP est la classe des langages qui ont un vérifieur s'exécutant en un temps polynomial. NP vient de *non-deterministic polynomial time*.

Un vérifieur pour un langage A est un algorithme V tel que :

$A = \{w | \exists string\ c : V \text{ accepte } (w, c)\}$

La complexité de V ne dépend que de w, et donc A est polynomialement vérifiable si V s'exécute en un temps polynomial. Le vérifieur utilise c pour vérifier qu'un mot w appartient au langage A. c est appelé le certificat pour w dans A.

2. Un langage est dans NP ssi il est décidé par une machine de Turing non-déterministe s'exécutant dans un temps polynomial.

Preuve d'équivalence :

Soit $A \in NP$ et V étant un vérifieur s'exécutant en un temps polynomial pour A. Supposons que V est une TM qui s'exécute en un temps n^k et construisons N comme suit :

- Sur une entrée w de longueur n :
  - Sélectionne de manière non-déterministe un string c de longueur plus petite ou égale à n^k
  - Exécute V sur l'entrée $\langle w, c \rangle$
  - Si V accepte, **accepte**, sinon **rejette**

On a donc une NTM qui décide A et qui s'exécute en un temps polynomial.

Dans l'autre direction, supposons que A est décidé par une machine de Turing non-déterministe (NTM) N et construisons le vérifieur à temps polynomial V suivant :

- Sur une entrée $\langle w, c \rangle$ où w et c sont des strings
  - Simule N sur w, en traitant chaque symbole de c comme une description du choix non-déterministe à faire à chaque étape
  - Si cette branche de l'exécution de N accepte, **accepte**, sinon **rejette**

Ce qui prouve l'équivalence dans les 2 sens.

28)

Un problème appartenant à NP est le problème de satisfaisabilité.

$$SAT = \{\Phi | \Phi \; is \; a \; boolean \; formula \; and \; is \; satisfiable\}$$

Ce problème est facilement vérifiable, le seul certificat dont nous avons besoin est un ensemble d'assignations de variables qui évaluent Φ à TRUE.

Ce problème est aussi NP Complet (par le théorème de Cook), et donc, s'il existe un algorithme en temps polynomial déterminant SAT, tel que $SAT \in P$, nous avons que $P = NP$. Comme il est conjecturé que $P \neq NP$, SAT n'est probablement pas dans P.

29)

Toute machine de Turing déterministe s'exécutant en un temps polynomial est équivalent à une machine non-déterministe s'exécutant en temps polynomial à une seule branche d'exécution, donc $P \subseteq NP$.

Donc, si $NP \subseteq P$, alors on a $P = NP$, mais il est de conjecture que $P \neq NP$ et il est donc peu probable que $NP \subseteq P$.

30)

Une machine de Turing N non déterministe ayant une complexité polynomiale t(n) aura un équivalent déterministe $2^{O(t(n))}$. Ceci rentre dans la catégorie des algorithmes à temps exponentiel.

$$NP \subseteq ExpTime$$

31)

$B \in$ NP-Complete si :

- $B \in NP$
- $\forall A \in NP : A \leq_P B$

Certains problèmes de NP sont tels que leur complexité est liée à toute la classe NP. Un algorithme à temps polynomial pour l'un des problèmes NP complet permettrait de trouver un algorithme à temps polynomial pour tous les autres problèmes de NP.

La notion de problème NP Complet est importante car :

- Pour prouver que $P = NP$, il suffirait de trouver un algorithme à temps polynomial pour un seul problème NP Complet, en effet, celui-ci permettrait alors de trouver des algorithmes à temps polynomiaux pour tous les autres problèmes.
- Cela permet aussi d'éviter de perdre du temps à chercher un algorithme polynomial pour un problème s'il s'avère qu'il est NP Complet. En effet, il est conjecturé que $P \neq NP$.

32)

Une fonction $f: \Sigma^* \to \Sigma^*$ est une fonction calculable en un temps polynomial si $\exists TM\ M$ s'arrêtant en temps polynomial et qui s'arrête avec juste $f(w)$ sur son ruban et en ayant pour input w.

Un langage A est réductible en temps polynomial à un autre langage B si et seulement si il existe une fonction $f: \Sigma^* \to \Sigma^*$ calculable en un temps polynomial tel que si $w \in A$, alors $f(w) \in B$.

Soit M un machine de Turing à temps polynomial, décidant B, tel que $B \in P$, et f la fonction de réduction en temps polynomial de A vers B.

Construisons N(w) ainsi :

- Calculer f(w)
- Effectuer M(f(w)), et répondre ce que répond M.

N s'exécute en temps polynomial et est déterministe, ce qui prouve que $A \in P$.

33)

(i) Par définition de NP-Complete :

- $B \in NP$
- Tout A de NP est réductible polynomialement à B.

Si $B \in P$, alors comme tout A de NP peut se réduire à B, on a que $A \in P$ (voir Q.32), et donc $P = NP$.

(ii) Comme tout A de NP peut se réduire à B (car il est NP Complet), et que B peut se réduire à C, on a que tout A de NP peut se réduire à C, car la réduction est transitive. Comme $C \in NP$, on a donc par définition que C est NP-Complet.

34)

Le théorème de Cook dit que le problème de Satisfaction est NP-Complet.

En effet, $SAT \in NP$, car il suffit d'avoir comme certificat une assignation des variables pour lesquelles $\Phi = TRUE$ pour prouver que $\Phi \in SAT$.

Pour prouver que $SAT \in NP - Complete$, on construit une réduction en temps polynomial $\forall A \in NP$ vers SAT. La réduction pour A prends un string w et produit une formule booléenne Φ qui simule la machine NP pour A sur l'entrée w. Si la machine accepte, Φ a une assignation satisfaisante qui correspond à la computation acceptante. Si la machine n'accepte pas, aucun assignement ne satisfait Φ. Dès lors, $w \in A \Leftrightarrow \Phi$ est satisfiable.

35)

$$VERTEX - COVER = \{(G, k) | G \text{ is an undirected graph that has a k-node vertex cover}\}$$

VERTEX-COVER ∈ NP, il suffit d'avoir un VERTEX-COVER de taille k comme certificat.

Réduisons maintenant 3SAT à VERTEX-COVER :

Pour chaque variable x de l'expression Φ, créons deux sommets dans un graphe G : $x \; et \; \neg x$. Choisir x dans le vertex cover simulera un assignment à TRUE, et sélectionner $\neg x$ un assignment à false. Pour s'assurer que au moins x ou $\neg x$ soit inclus dans le cover, traçons une arête entre les deux. Appelons V_1 cet ensemble de sommets.

Ensuite, pour chaque clause $c_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$, on rajoute un triple de points dans G, ainsi que des arêtes les reliants. Ceci assurera que l'on doive prendre deux sommets de ce triple dans le cover. On relie ensuite chacun de ces sommets au sommet de V_1 correspondant.

Fixons $k = m + 2L$.

Prouvons que (G,k) est une instance positive de VERTEX-COVER ssi $\Phi \in SAT$.

Dans un sens :

Soit $v : P \rightarrow \{0,1\}$ une assignation des littéraux P tel que $\Phi = TRUE$. Considérons un sous ensemble de sommets C, comprenant les sommets $x_i \in V_1 \; si \; v(x_I) = 1, et \; \neg x_i \in V_1 \; si \; v(x_i) = 0$, ainsi que deux des trois sommets de chaque triple de sommets, tel que le sommet x qui n'est pas pris soit tel que v(x) = 1.

Pour peu que chaque triple contienne au moins un sommet pour lequel v(x) vaut 1, le VERTEX-COVER contiendra 1 sommet de chaque couple $(x_i, \neg x_i)$, ainsi que deux sommets de chaque triples. Chaque arête de G touchera donc bien le VERTEX-COVER, si $\Phi \in 3SAT$.

Dans l'autre sens, si C est un k-cover de G, alors un sommet de chaque couple $(x_i, \neg x_i)$ de V_1 a été sélectionné. (Pas moins, sinon on ne couvre pas une arête, et pas plus sinon, on n'a plus assez de sommets pour couvrir les triples). Donc, dans chaque triple, au moins deux sommets sont dans C, et le sommet qui n'est pas dans C est relié à un sommet de V_1 qui est dans C.

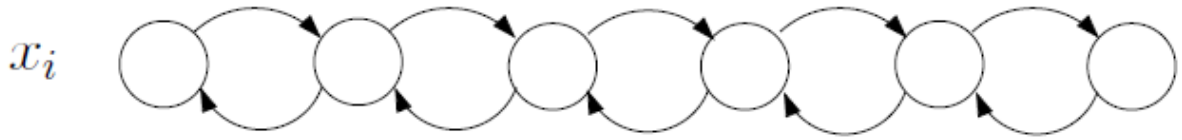
Définissons $v(x) = 1 \; si \; x \in C \; et \; v(x) = 0 \; si \; x \notin C$.

Comme v(x) doit être égal à 1 pour au moins un des littéraux dans chaque clause de Φ, on a $\Phi \in 3SAT$.

36) http://sma.epfl.ch/~moustafa/Other/Complexityslides/lec7.pdf

HAMPATH ∈ NP, car il suffit d'avoir la soite de sommets comme certificat pour vérifier qu'ils sont bien reliés.

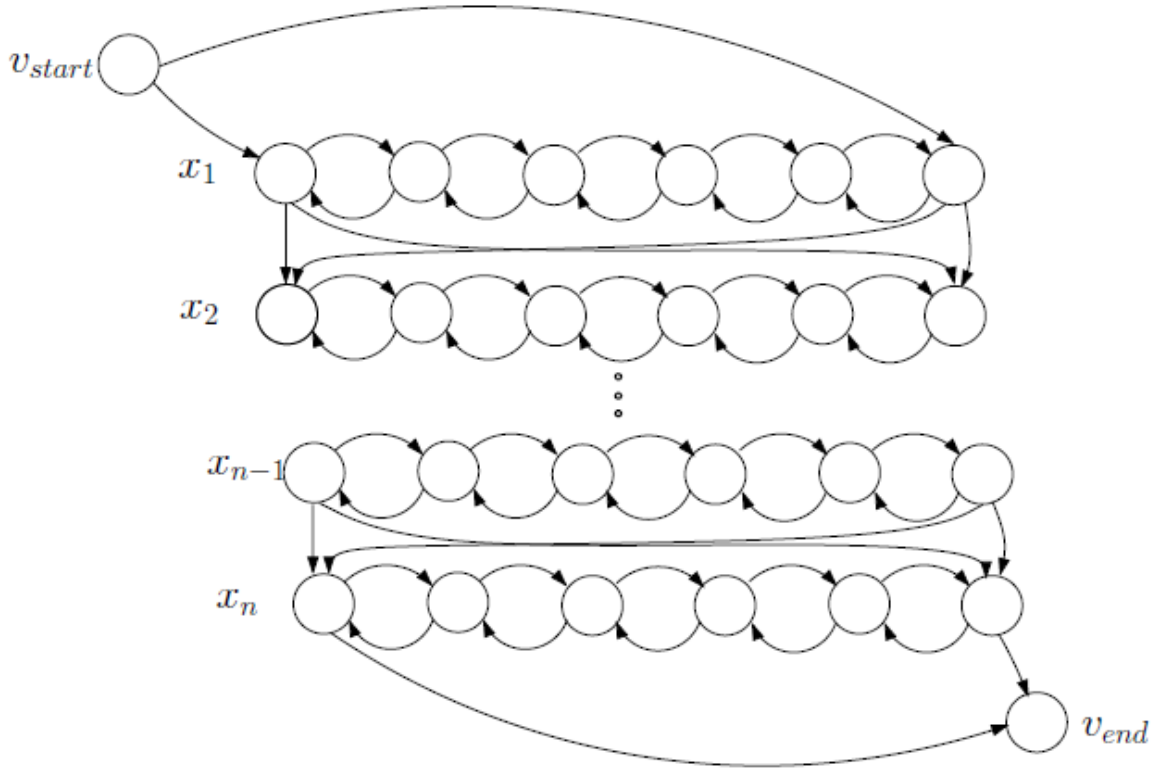
Réduisons 3SAT à HAMPATH : Soit un graph G, tel que chaque x_i est représenté par un chemin de $6 * m$ sommets :



Rajoutons deux sommets v_{start}, qui possède deux arêtes partant vers le premier et le dernier sommet de x_1, et v_{end} qui possède deux arêtes provenant du premier et du dernier sommet de x_n (où x_n est la dernière variable).

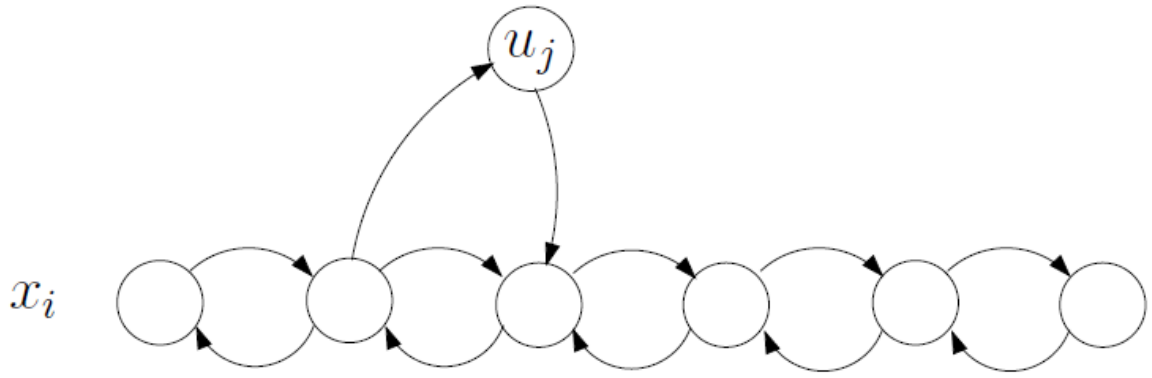
Du permier et dernier sommet de chaque x_i, on crée deux arêtes allant au premier et au dernier sommet de x_{i+1}.

Tout chemin hamiltonien doit donc commencer en v_{start}, finir en v_{end} et traverser chaque chaine de droite à gauche (signifiant $x_i = 0$) ou de gauche à droite (signifiant $x_i = 1$).



A chaque clause c_j, créons un nouveau sommet u_j.

Pour chaque littéral non négationné de la clause, rajoutons une arête d'un des maillons de la chaine du littéral vers la clause, puis de la clause au maillon suivant.



Si le littéral est négationné, inverser le sens des arêtes.

Remarquons donc, que si $x_i \in C_j$ et $x_i = 1$, on peut visiter u_j en chemin. De même, si $\neg x_i \in C_j$ et $x_i = 0$, on peut visiter u_j en chemin. On a donc que $\Phi \in 3SAT \Leftrightarrow G \in HAMPATH$.

37)

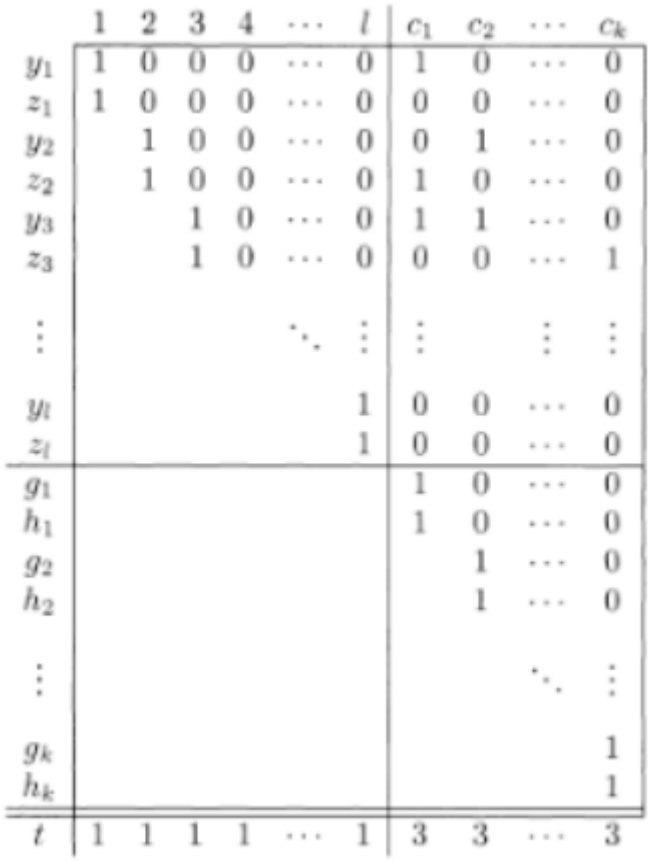
Créons un vérificateur pour SUBSET-SUM V((S,t),c) ainsi :

- Tester si la somme des nombres dans c vaut t.
- Tester si S contient tous les nombres de c.
- Si les deux acceptent, accept, sinon rejette.

Donc, $SUBSET - SUM \in NP$.

Prouvons maintenant que 3SAT réduit à SUBSET-SUM.

Partant d'un problème Φ exprimé en 3SAT, créons le problème SUBSET-SUM(S,t) où S et t sont les concaténations des lignes de la table suivante :



	1	2	3	4	⋯	l	c_1	c_2	⋯	c_k
y_1	1	0	0	0	⋯	0	1	0	⋯	0
z_1	1	0	0	0	⋯	0	0	0	⋯	0
y_2		1	0	0	⋯	0	0	1	⋯	0
z_2		1	0	0	⋯	0	1	0	⋯	0
y_3			1	0	⋯	0	1	1	⋯	0
z_3			1	0	⋯	0	0	0	⋯	1
⋮					⋱	⋮	⋮		⋮	⋮
y_l						1	0	0	⋯	0
z_l						1	0	0	⋯	0
g_1							1	0	⋯	0
h_1							1	0	⋯	0
g_2								1	⋯	0
h_2								1	⋯	0
⋮									⋱	⋮
g_k										1
h_k										1
t	1	1	1	1	⋯	1	3	3	⋯	3

- $y_i \; et \; z_i$ sont associés à x_i
    - En choisissant la ligne y_i dans S, on simule x_i à TRUE
    - En choisissant la ligne z_i dans S, on simule x_i à FALSE
- Dans la ligne y_i et la colonne c_i, on met 1 si $x_i \in c_i$, 0 sinon
- Dans la ligne z_i et la colonne c_i, on met 1 si $\neg x_i \in c_i$, 0 sinon
- Les $g_i et \; h_i$ ne contiennent 1 que sur la colonne c_i, 0 autre part.
- $t = 1^L 3^k$ (concaténation de L « 1 » et de k « 3 »).

Si Φ est satisfaisable : soit la fonction $v : P \rightarrow \{0,1\}$, tel que $\Phi = TRUE$.

Choisissons les éléments de S ainsi : $\{y_i | v(x_i) = 1\} \cup \{z_i | v(x_i) = 0\}$. En additionnant ces nombres, on obtient la somme suivant : $1^L\{1,2,3\}^k$. Il suffit alors de rajouter le bon nombre de lignes de $g_i \; et \; h_i$, on obtient la somme $1^L 3^k = t$. Donc, pour que $(S,t) \in SUBSET - SUM$ il faut que $\Phi \in 3SAT$.

Dans l'autre sens, il suffit de choisir, pour chaque colonne de 0 à L du tableau, une des lignes $x_i \; ou \; y_i$, afin d'avoir une somme de 1. Soit T l'ensemble des lignes sélectionnées. Définissons $v : P \rightarrow \{0,1\}$ tel que $v(p_i) = 1 \; si \; y_i \in T$ et $v(p_i) = 0 \; si \; z_i \in T$.

Pour les colonnes de L+1 à k, il nous faut une somme de 3. Comme on ne peut obtenir qu'une somme de 2 avec les $g_i et \; h_i$, il faut qu'un littéral soit vrai dans chaque colonne c_1 à c_k. Donc, pour que $\Phi \in 3SAT$, il faut que $(S,t) \in SUBSET - SUM$.

38)

CLIQUE appartient à NP, il nous suffit d'avoir comme certificat un ensemble de sommets de taille k, reliés par un à un par des arêtes. Ensuite, on peut créer un vérificateur V((G,k),c) ainsi :

- Tester si c est un ensemble de sommets de G
- Tester si G contient des arêtes reliant ensemble tous les sommets de c
- Si les deux accepte, on accepte, sinon on rejette.

Pour prouver que CLIQUE est NP-Complet, prouvons que l'on peut réduire 3SAT à CLIQUE :

k vaut le nombre de conjonctions dans l'expression 3SAT.

Soit $\Phi = \wedge_{i=1..k} (a_i \vee b_i \vee c_i)$ une instance de 3SAT. Construisons le graphe G ainsi :

Pour chaque clause de Φ, nous associons un triple de sommets de G. Chacun de ces triples est noté $t_i = (a_i, b_i, c_i)$.

Créons ensuite les arêtes de G telles qu'elles connectent tous les sommets de G, excepté :

- Les sommets d'un même triple
- Les sommes avec labels contradictoires : $x \; et \; \neg x$

Il faut maintenant montrer que $(G, k) = f(\Phi)$ est une instance positive de CLIQUE ssi Φ est une instance positive de 3SAT.

Pour cela, soit la fonction $v : P \rightarrow \{0,1\}$, P étant l'ensemble des littéraux. v étant une assignation de vérités satisfaisant Φ.

Comme $v(p_i)$ doit au moins être vrai pour un littéral de chaque triple, et qu'on ne peut avoir $p_i \; et \; \neg p_i$ vrais en même temps, si $\Phi \in 3SAT$, alors $(G, k) \in CLIQUE$.

Inversement, s'il existe un k-CLIQUE C dans G, et soit la fonction $v : P \rightarrow \{0,1\}$, telle que pour tous $c \in C$,

- $v(p) = 1$ si $c \in C \; et \; p \in c$
- $v(p) = 0 \; si \; c \in C \; et \; \neg p \in c$

Toutes les assignations de vérité étendant v seront telles que $\Phi = TRUE$. Donc, s'il existe un k-CLIQUE dans G, $\Phi \in 3SAT$.

39)

Tout d'abord, il est clair que $3SAT \in NP$, de la même façon que $SAT \in NP$.

Pour le prouver, nous allons modifier la preuve du théorème de Cook, afin de montrer que toutes les formules utilisées dedans peuvent être écrites en 3CNF.

Les Φ_{Cell}, $\Phi_{Accept} \; et \; \Phi_{Start}$ sont déjà en CNF. Il faut encore passer Φ_{Move} en CNF.

Φ_{Move} est une conjonction de disjonctions de conjonctions. On peut transformer la disjonction de conjonctions en conjonctions de disjonctions avec les formules de distributivité booléennes.

On peut alors transformer Φ_{Move} en CNF.

Ensuite, comme on peut transformer une conjonction de plus de trois litérals, du type :

$(a_1 \vee a_2 \vee a_3 \vee \ldots \vee a_m)$

En conjonctions de disjonctions de trois litéraux du type :

$(a_1 \vee a_2 \vee z_1) \wedge (\neg z_1 \vee a_2 \vee z_2) \wedge \ldots \wedge (\neg z_{m-3} \vee a_{m-1} \vee a_m)$

On peut transformer tous les Φ en 3SAT, ce qui prouve que tout A de NP se réduit à une expression 3CNF, et donc 3SAT est NP-Complet.

40)

Soit M une *TM decider*. La complexité spatiale de M est une fonction $f: \mathbb{N} \to \mathbb{N}$, où $f(n)$ est le nombre maximum de cellule de ruban que M scanne sur n'importe quel entrée de taille n. On dit que M s'exécute dans l'espace $f(n)$. Si M est non-déterministe, on définit sa complexité $f(n)$ comme étant le nombre maximum de cellule de ruban que M scanne dans toutes les branches de son exécution pour une entrée de taille n.

Exemple de comparaison temps/espace :

Soit l'algorithme suivant résolvant SAT :

- Sur une entrée $\langle \phi \rangle$ où ϕ est une expression booléenne
- Pour chaque assignement de vérité pour les variables $x_1, \ldots, x_m$ de ϕ
    - Évaluer ϕ pour cette assignation de vérité
- Si à un moment ϕ a été évalué à 1, **accepte**, sinon on **rejette**.

Soit n la taille de la formule de ϕ. Alors, le nombre de variables m est inférieur à n. Donc, comme l'algorithme n'a besoin que des assignements actuels des variables, il utilise seulement $O(m) = O(n)$ espace mais utilise $O(2^m) = O(2^n)$ en temps.

41)

La configuration d'une machine de Turing déterministe peut s'écrire de cette façon : $(w_1 q w_2)$. Le nombre de configurations différentes qu'une machine de Turing peut avoir est du coup :

$$k \times Q \times L^k$$

Où Q est le nombre d'état que possède la machine de Turing, L est le nombre de symboles dans l'alphabet, et k est la quantité de mémoire disponible (fois k car position de la tête de lecture).

Nous savons que cet algorithme s'exécute en $SPACE\left(O\left(t(n)\right)\right)$, et qu'une machine de Turing qui passe deux fois par le même état est en train de boucler (ce qui est impossible pour un décideur), son temps maximal d'exécution vaut

$$t(n) \times Q \times L^{O(t(n))}$$

Cette machine s'exécute donc en $O\left(2^{O(t(n))}\right)$.

42)

Énoncé :

$$\forall f(n) | f(n) \geq n; NSpace\left(O\left(f(n)\right)\right) \subseteq Space\left(O\left(f^2(n)\right)\right)$$

Une $NTM\ N$ qui s'exécute en espace $O\left(f(n)\right)$ peut avoir des branches qui sont de longueur (en temps) de $2^{O(f(n))}$. Durant ces exécutions, il peut il y avoir jusqu'à $2^{O(f(n))}$ choix non-déterministes. Donc, afin de simuler toutes les branches, si on veut le faire branche par branche, cela requiert de stocker tous ces choix ce qui requiert un espace mémoire exponentiel. Ainsi, une simple simulation ne fera pas l'affaire.

Soit 2 configurations c_1 et c_2, et $t \in \mathbb{N}$, on veut établir s'il existe une exécution de N entre c_1 et c_2 en t étapes. Ceci est appelé le problème de *yieldability.* En résolvant ce problème, nous serons capable de décider si N accepte une entrée w.

Établissons de manière récursive l'existence de la configuration intermédiaire c_m tel que :

- c_1 peut aller à c_m avec jusqu'à $\frac{t}{2}$ étapes
- c_m peut aller à c_2 avec jusqu'à $\frac{t}{2}$ étapes

Réutilisé la mémoire pour chacune de ces étapes récursives permet une économie drastique de ressource. Cet algorithme a besoin d'un espace pour stocker le *stack* des appels récursifs. Chaque niveau de récursion utilise $O\left(f(n)\right)$ en espace pour stocker la configuration c_m associée. La profondeur de la récursion est $\log t$ où t est le nombre maximal d'étapes pour lesquelles la machine peut s'exécuter. Comme t est $2^{O(f(n))}$, alors $\log t$ est $O\left(f(n)\right)$.

De fait, la simulation déterministe utilise $O\left(f^2(n)\right)$ en espace (stockage d'une configuration en $O\left(f(n)\right)$et profondeur de la récursion en $O\left(f(n)\right)$).

43)

La classe $PSpace$ est la classe des langages qui sont décidables dans un espace polynomial par une TM déterministe.

La classe $NPSpace$ est la classe des langages qui sont décidables dans un espace polynomial par une TM non-déterministe.

La classe $coPSpace$ est la classe des langages qui sont co-décidables dans un espace polynomial par une TM déterministe.

La classe $coNPSpace$ est la classe des langages qui sont co-décidables dans un espace polynomial par une TM non-déterministe.

Par définition de $PSpace$: $PSpace \subseteq NPSpace$

Par le théorème de Savitch : $NPSpace \subseteq PSpace$

Et donc $PSpace = NPSpace$

Comme $PSpace = coPSpace$ et $NPSpace = coNPSpace$ alors $PSpace = NPSpace = coPSpace = coNPSpace$

44)

Un langage B est $PSpace - Complete$ si :

- $B \in PSpace$
- $\forall A \in PSpace : A \leq_P B$.

Un problème satisfaisant seulement la deuxième condition est appelé $PSpace - hard$. Cette notion est importante car si on peut ramener toute langage A à u langage complexe B dont on connait la solution, il suffit de trouver les fonvtion de réduction en temps polynomial pour chaque A auquel on s'intéresse pour en trouver la solution.

TQBF est une généralisation de SAT mais avec des quantifieurs en plus (∀, ∃, …). Une formule QBF est totalement quantifiée quand chaque variable booléenne est quantifiée.

$$TQBF = \{(\Phi) | \Phi \text{ est une formule booléenne totalement quantifiée vraie}\}$$

45)

Un algorithme qui assigne des valeurs à des variables et qui vérifie de manière récursive la valeur de la nouvelle formule résout le problème. À partir de l'information à propos de la valeur de vérité de la formule, l'algorithme peut calculer la vraie valeur de la formule originelle.

Soit une formule du type $\exists p | \Phi$, l'algorithme fait 2 appels récursifs, un pour Φ pour p vrai et un pour Φ pour p faux. Si un de ces appels retourne vrai, l'algorithme retourne vrai, sinon il retourne faux. Sur une formule de type $\forall p | \Phi$ l'algorithme fait 2 appels récursifs, un pour Φ pour p vrai et un pour Φ pour p faux. Si un de ces appels retourne faux, l'algorithme retourne faux, sinon il retourne vrai.

Comme la profondeur de la récursion est le nombre de variables de la formule, et qu'à chaque niveau, nous n'avons besoin que de la valeur de la sous-formule, l'algorithme n'a besoin que d'un espace linéaire.

46)

Dans le cas d'une TM qui s'exécute dans un espace polynomial, l'exécution de la machine sur un mot w peut être exponentiellement long. Donc, la table servant à représenter l'exécution serait de l'ordre de $2^{O(nk)} \times O(n^k)$. Chaque ligne est toujours de taille polynomiale comme la machine opère dans un espace polynomial. Donc, écrire une formule booléenne qui exprime les contraintes de la table de la même manière que nous l'avons écrit pour Cook-Levin requerrait des formules exponentiellement longues, on ne peut pas se permettre ça.

47)

$$P \subseteq NP \subseteq PSpace = NPSpace \subseteq ExpTime$$

Clairement, $P \subseteq PSpace$ car en temps polynomial, seul les espaces polynomiaux peuvent être utilisés si on considère à chaque étape un scan de cellule. De manière similaire $NP \subseteq NPSpace$.

De plus, $PSpace \subseteq ExpTime$ car n'importe quelle TM qui utilise un espace polynomial $O(n^k)$ peut avoir jusqu'à $n^k 2^{O(n^k)}$.