TD N°2

Exercice 1 :

1) calculer la complexité de l'algorithme d'addition en binaire suivant :

sum : $\{0,1\} \times \{0,1\} \times \{0,1\} \to \{0,1,2,3\}$, $\text{sum}(a,b,c) = a+b+c$.

Algorithm 2.3 *Binary integer addition.*

Input: integers $a \geq b \geq 0$ encoded in binary as $a_n \cdots a_1$ and $b_n \cdots b_1$.
Output: $a+b$ in binary.
Algorithm:
$c \leftarrow 0$
for $i = 1$ to n
  if sum(a_i, b_i, c) equals 1 or 3
    then $d_i \leftarrow 1$
    else $d_i \leftarrow 0$
  if sum$(a_i, b_i, c) \geq 2$
    then $c \leftarrow 1$
    else $c \leftarrow 0$
next i
if $c = 1$
  then output $1 d_n d_{n-1} \cdots d_1$
  else output $d_n d_{n-1} \cdots d_1$.

2) l'algorithme d'addition binaire est-il plus efficace que celui d'addition unaire vue en cours ?

Solution :

1) Pour chaque exécution de la boucle for on a :

Le cout de sum(a,b,c)=2 car il y a 2 opération addition, Le cout de de l'affection = 1

Donc Le cout de premier if =2+1=3 et Le 2eme if=3 donc Le cout d'une exécution=3+3=6

Le cout d'exécution de boucle de taille $n = \lfloor \log(a) \rfloor + 1$ est $6(\lfloor \log(a) \rfloor + 1)$

En ajoutant le cout de la 1ere opération d'affection =1 plus le cout de dernier if = 2 (comparaison plus affichage de résultat)

Donc la complexité = $6(\lfloor \log(a) \rfloor + 1) + 3 = 6\log(a) + 9$

2)

Temps pris par l'algo à code unaire est 3b+1 et l'algo à code binaire est $6\log a + 9$

Pour les gros entrées, l'algo à code binaire est clairement bien plus efficace que l'algo à code unaire comme il démontre le tableau ci-dessous

Input a, b	Unary machine steps	Binary machine steps
10	22	< 200
1000	2000	< 900
10^6	2×10^6	< 3000
2^{100}	2.5×10^{30}	$< 65\,000$

Exercice 2 : Soit l'algorithme suivant de teste de primalité :

Algorithm *Naive Primality Testing.*

Input: an integer $N \geq 2$.
Output: true if N is prime and false otherwise.
Algorithm:
$D \leftarrow 2$
$P \leftarrow$ true
while P is true and $D \leq \sqrt{N}$
    if D divides N exactly
        then $P \leftarrow$ false
        else $D \leftarrow D + 1$
end-while
output P

1)Calculer ça complexité.

2)Est-il efficace ? justifier en donnant un exemple.

Solution : *1)* $O\left(\sqrt{N}\right)$ *2) Non ; car la complexité en binaire est exponentielle* $O\left(2^{\frac{n}{2}}\right), n = \lfloor \log(N) \rfloor + 1$

Pour un nombre sur 1024 bit ,il faut un temp de test 2^{512} *.C'est non seulement au-delà des limites des ordinateurs modernes, mais sans doute au-delà de la portée de tout ce que nous pourrions envisager.*
***Utilisation** : pour utiliser certains crypto-systèmes modernes, nous devons être en mesure de tester la primalité de tels nombres.*

Exercice 3 :

1) Donner la machine DTM de l'algorithme « addition en binaire » de l'exercice 1
2) Tester pour 5+2 en binaire
3) Calculer le temps de DTM

Solution :
(Notez que dans l'algorithme ci-dessous, nous utilisons des abréviations pour réduire le nombre de valeurs de la fonction de transition que nous devons décrire. Par exemple (γ3 / γ4, 0/1, s, s,

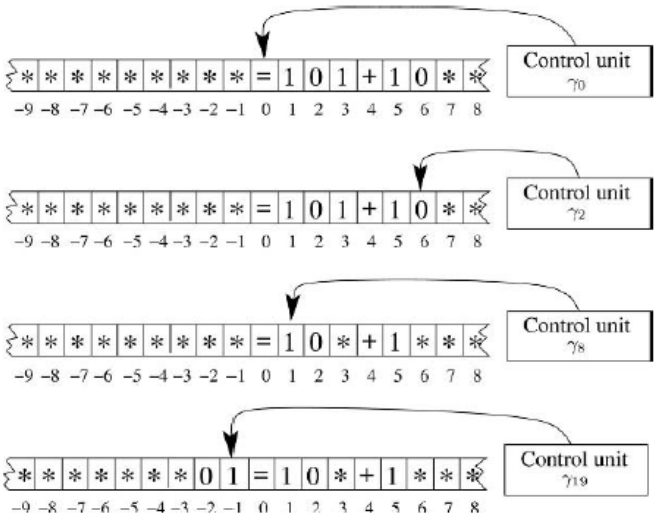
←) est une abréviation pour (γ3, 0 , γ3, 0, ←), (γ3, 1, γ3, 1, ←), (γ4, 0, γ4, 0, ←) et (γ4, 1, γ4, 1, ←). La lettre s désigne le fait que *l'état / symbole* reste le même.)

1)

Algorithm *Binary Addition DTM*
The set of states is $\Gamma = \{\gamma_0, \gamma_1, \ldots, \gamma_{24}\}$ the starting state is γ_0, the only halting state is γ_{24}. The alphabet is $\Sigma = \{*, 0, 1, +, =\}$.
Input: integers $a \geq b \geq 0$ in binary with $+, =$. (For example to compute $31 + 18$ we would write '$= 11111 + 10010$' on the machine's tape, with the symbol '$=$' in the starting square.)
Output: $a + b$ in binary.

$(\gamma_0, =, \gamma_1, =, \rightarrow)$	# move the head to the right end of the input
$(\gamma_1, 0/1/+, \gamma_1, s, \rightarrow)$	# 〃
$(\gamma_1, *, \gamma_2, *, \leftarrow)$	# found end of input
$(\gamma_2, 0, \gamma_3, *, \leftarrow)$	# the least significant bit of b is 0
$(\gamma_2, 1, \gamma_4, *, \leftarrow)$	# the least significant bit of b is 1
$(\gamma_2, +, \gamma_5, +, \leftarrow)$	# no more bits of b
$(\gamma_3/\gamma_4, 0/1, s, s, \leftarrow)$	# keep moving left until we have finished read b
$(\gamma_3, +, \gamma_5, +, \leftarrow)$	# finished reading b
$(\gamma_4, +, \gamma_6, +, \leftarrow)$	# 〃
$(\gamma_5, =, \gamma_{23}, * \rightarrow)$	# no more bits of a erase $=$
$(\gamma_5/\gamma_6, *, s, *, \leftarrow)$	# moving left looking for a
$(\gamma_5, 0, \gamma_7, *, \leftarrow)$	# sum of least significant bits of a and b is 0
$(\gamma_5, 1, \gamma_8, *, \leftarrow)$	# sum of least significant bits of a and b is 1
$(\gamma_6, 0, \gamma_8, *, \leftarrow)$	# sum of least significant bits of a and b is 1
$(\gamma_6, 1, \gamma_9, *, \leftarrow)$	# sum of least significant bits of a and b is 2
$(\gamma_7/\gamma_8/\gamma_9, 0/1, s, s, \leftarrow)$	# moving left looking for $=$
$(\gamma_7, =, \gamma_{10}, =, \leftarrow)$	# finished reading a, found $=$
$(\gamma_8, =, \gamma_{11}, =, \leftarrow)$	# 〃
$(\gamma_9, =, \gamma_{12}, =, \leftarrow)$	# 〃
$(\gamma_{10}/\gamma_{11}/\gamma_{12}, 0/1, s, s, \leftarrow)$	# moving left looking for the end of the answer
$(\gamma_{10}, *, \gamma_{13}, *, \rightarrow)$	# finished reading answer, now find the carry bit
$(\gamma_{11}, *, \gamma_{14}, *, \rightarrow)$	# 〃
$(\gamma_{12}, *, \gamma_{15}, *, \rightarrow)$	# 〃
$(\gamma_{13}, 0, \gamma_{16}, 0, \leftarrow)$	# carry bit and least sig bits of a and b sum to 0
$(\gamma_{13}, 1, \gamma_{16}, 1, \leftarrow)$	# carry bit and least sig bits of a and b sum to 1
$(\gamma_{14}, 0, \gamma_{16}, 1, \leftarrow)$	# carry bit and least sig bits of a and b sum to 1
$(\gamma_{14}, 1, \gamma_{17}, 0, \leftarrow)$	# carry bit and least sig bits of a and b sum to 2
$(\gamma_{15}, 0, \gamma_{17}, 0, \leftarrow)$	# carry bit and least sig bits of a and b sum to 2
$(\gamma_{15}, 1, \gamma_{17}, 1, \leftarrow)$	# carry bit and least sig bits of a and b sum to 3
$(\gamma_{13}, =, \gamma_{18}, =, \leftarrow)$	# first part of answer is 0
$(\gamma_{14}, =, \gamma_{19}, =, \leftarrow)$	# first part of answer is 1
$(\gamma_{15}, =, \gamma_{20}, =, \leftarrow)$	# first part of answer is 0 and carry bit is 1
$(\gamma_{16}, *, \gamma_{21}, 0, \rightarrow)$	# set carry bit to 0 and now return to start
$(\gamma_{17}, *, \gamma_{21}, 1, \rightarrow)$	# set carry bit to 1 and now return to start
$(\gamma_{18}, *, \gamma_{16}, 0, \leftarrow)$	# first part of answer is 0
$(\gamma_{19}, *, \gamma_{16}, 1, \leftarrow)$	# first part of answer is 1
$(\gamma_{20}, *, \gamma_{17}, 0, \leftarrow)$	# first part of answer is 0 and carry bit is 1
$(\gamma_{21}, 0/1/=/*, \gamma_{21}, s, \rightarrow)$	# return to start
$(\gamma_{21}, +, \gamma_{22}, +, \rightarrow)$	# finished rereading a, found $+$
$(\gamma_{22}, 0/1, \gamma_{22}, s, \rightarrow)$	# now rereading b
$(\gamma_{22}, *, \gamma_2, *, \leftarrow)$	# reached start of the input
$(\gamma_{23}, *, \gamma_{23}, *, \rightarrow)$	# keep moving right
$(\gamma_{23}, +, \gamma_{24}, *, \rightarrow)$	# erase $+$ and halt

2) Tester l'exécution pour 5+2



3)

a ≥ b ≥ 0, où a est un entier de k bits, alors il est raisonnablement facile de voir que la machine fait au plus 2k + 3 pas avant que la tête de lecture-écriture ne soit positionnée sur le symbole non vide le plus à droite et l'unité de commande est à l'état γ2. La machine effectue alors au plus 6 (k + 2) pas avant de se retrouver à nouveau dans l'état γ2 et la tête de lecture-écriture balaie à nouveau le symbole non vide le plus à droite. La machine fait cela k fois, une fois pour chaque bit dans a. Enfin, il efface le signe égal et plus. Au total, il faut moins de $6 (k + 2)^2$ étapes.

Exercice 4 :

Soit un problème de décision le graphe CLIQUE.

CLIQUE
Entrée: un graphe G d'ordre n et un entier $2 \leq k \leq n$.
Question: G contient-il une clique d'ordre k?

1) Est-ce que CLIQUE ∈ P ?

Solution :

La taille d'entrée est la somme des tailles d'entrée des différentes parties de l'entrée. Donc, dans ce cas, l'entrée est un graphe, qui a une taille d'entrée $O(n^2)$, en utilisant la matrice adjacente, et un entier $2 \leq k \leq n$, avec une taille d'entrée $O(\log k)$ en utilisant un codage binaire. D'où la taille d'entrée totale pour CLIQUE est $O(n^2) + O(\log k) = O(n^2)$.
Par conséquent, tout algorithme de temps polynomial pour CLIQUE doit avoir un temps d'exécution limité par un polynôme en n. Cependant, si nous utilisions l'algorithme ci-dessus pour essayer de décider d'une instance de CLIQUE avec $k = \sqrt{n}$ alors, dans le pire des cas, il faudrait vérifier $\binom{n}{\sqrt{n}} \binom{\sqrt{n}}{2}$ arêtes possibles et aurait donc du temps d'exécution $\Omega\left(n^{\frac{\sqrt{n}}{2}}\right)$ qui n'est pas polynomial en n. Donc on ne sait pas si CLIQUE appartient à P.

Exercice 5 :

1) Monter que 2-SAT ∈ P ?

Preuve : Supposons que $f(x_1,...,x_n) = \wedge_{k=1}^{m} C_k$ dans CNF est notre entrée dans 2-SAT. Ensuite, chaque clause, C_k, est la disjonction d'au plus deux littéraux. Si une clause contient un seul littéral, xi, nous pouvons supposer que la clause est remplacée par xi ∨ xi et ainsi chaque clause de f contient exactement deux littéraux.

On définit un digraphe associé G_f = (V, E) dont les sommets sont constitués des littéraux

$$V = \{x_1, \ldots, x_n, \overline{x}_1 \ldots, \overline{x}_n\}$$

et dont les arêtes sont définies par

$$E = \{(a, b) \mid \overline{a} \vee b \text{ is a clause in } f\}$$

Notez que G_f a la propriété que

$$(a, b) \in E \iff (\overline{b}, \overline{a}) \in E$$

Considérez le problème de décision suivant pour les digraphes

ACCESSIBILITÉ

Entrée: un digraphe G = (V, E) et deux sommets v, w ∈ V.

Question: y a-t-il un chemin dirigé de v à w dans G?

Nous affirmons que:

(a) f est insatisfaisable si et seulement s'il existe une variable x_i pour laquelle il existe un chemin dirigé de x_i vers $\overline{x}_i$ et de $\overline{x}_i$ vers x_i dans le digraphe G_f.

(b) ACCESSIBILITÉ ∈ P.

Preuve de (a) : Nous pouvons maintenant décrire un algorithme de temps polynomial pour 2-SAT. Définissez d'abord une fonction r: V × V → {1, 0} par

$$r(v, w) = \begin{cases} 1, & \text{if there is a directed path from } v \text{ to } w \text{ in } G_f \\ 0, & \text{otherwise.} \end{cases}$$

Entrée: une formule 2-SAT f.

Sortie: vrai si f est satisfiable et faux dans le cas contraire.

***Algorithme**:*

Construire le graphe G_f

pour i = 1 à n

si $r(x_i, \overline{x}_i) = 1$ et $r(\overline{x}_i, x_i) = 1$ alors sortie faux

next i

sortie vrai

Est satisfiable : le fait que cet algorithme décide correctement si l'entrée est satisfiable découle directement de (a) ci-dessus

Est polynomiale : Premièrement, la construction de G_f peut être réalisée en temps polynomial puisque nous avons simplement besoin de lire l'entrée et, pour chaque clause (a ∨ b), nous insérons deux arêtes dans notre graphe : $(\bar{a}, b)$ et $(\bar{b}, a)$. Deuxièmement, la fonction r (·, ·) peut être évaluée en temps polynomial (par (b) ci-dessus). Enfin, la boucle for de l'algorithme est répétée au plus *n* fois et donc r (·, ·) est appelée au plus 2n fois. Il s'agit donc d'un algorithme de temps polynomial.

Nous donnons maintenant une preuve de (a). Supposons d'abord que pour un certain $1 \leq i \leq n$, il existe des chemins dirigés dans G_f de x_i vers $\bar{x}_i$ et de $\bar{x}_i$ vers x_i. Nous montrerons que dans ce cas *f* n'est pas satisfaisable puisqu'aucune valeur de vérité ne peut être choisie pour x_i. Les deux chemins dirigés impliquent que les clauses suivantes appartiennent à *f*:

$$(\bar{x}_i \vee y_1), (\bar{y}_1 \vee y_2), \ldots, (\bar{y}_{j-1} \vee y_j), (\bar{y}_j \vee \bar{x}_i),$$
$$(x_i \vee z_1), (\bar{z}_1 \vee z_2), \ldots, (\bar{z}_{k-1} \vee z_k), (\bar{z}_k \vee x_i).$$

Les clauses de la première ligne impliquent simplement que x_i ne peut pas être vraie alors que celles de la deuxième ligne impliquent que x_i ne peut pas être fausse. Donc *f* n'est pas satisfaisable.

A l'inverse, supposons que pour chaque $1 \leq i \leq n$ il n'y ait pas de chemin dirigé dans G_f de x_i à $\bar{x}_i$ ou qu'il n'y ait pas de chemin dirigé de $\bar{x}_i$ à xi. Pour un littéral a, nous définissons R(a) comme étant les littéraux qui peuvent être atteints par des chemins dirigés à partir de a (avec a lui-même). Nous définissons également $\bar{R}(a)$ comme étant les négations des littéraux dans R (a). Nous construisons une affectation de vérité satisfaisante en utilisant la procédure suivante.

i ← 1

Tanque i ≤ n

si r (xi, $\bar{x}_i$) = 0

alors a ← xi

sinon un ← $\bar{x}_i$

définir tous les littéraux de R (a) sur true

définir tous les littéraux de R (a) sur faux

si une variable n'a pas encore reçu de valeur de vérité

alors i ← min {j | x_j n'est pas attribué}

sinon i ← n + 1

fin Tanque

Pour voir que cela fonctionne, nous devons vérifier que nous n'avons jamais à la fois *v* et $\overline{v}$ dans R(a). Si nous le faisions, alors il existerait des chemins dirigés de *a* vers *v* et de *a* vers $\overline{v}$. Mais G_f a la propriété qu'il y a un chemin dirigé de *c* vers *d* si et seulement s'il existe un chemin dirigé de $\overline{d}$ vers $\overline{c}$ .Par conséquent, dans ce cas, il y aurait un chemin dirigé de $\overline{v}$ à $\overline{a}$. Ainsi, il y aurait un chemin dirigé de *a* vers $\overline{a}$ (via $\overline{v}$), contredisant notre hypothèse selon laquelle un tel chemin n'existe pas (puisque $r(a,\overline{a})$ = 0).

Enfin, nous notons que nous ne pouvons pas rencontrer de problèmes à un stade ultérieur car si nous choisissons un littéral b non attribué tel que $r(b,\overline{b})$ = 0, alors il n'y a pas de chemin dirigé de *b* vers un littéral auquel la valeur *false* a déjà été attribuée (si tel était le cas, vous pouvez vérifier que *b* aurait également déjà reçu la valeur *false*).

Preuve (b) : montrer que le problème d'ACCESSIBILITÉ ∈ P ?

Exercice 6 :

1) Soit la fonction multiplication $multi:\mathbb{Z}^+ \to \mathbb{Z}^+$ où $multi(a,b)=ab$
   Montrer que multi ∈ FP ?

2) Soit la fonction diviseur $div:\mathbb{Z}^+ \times \mathbb{N} \to \mathbb{Z}^+$ où $div(a,b)=\lfloor a/b \rfloor$
   Montrer que div ∈ FP ?

3) Soit la fonction exponentiation $\exp(a,b,c):\mathbb{Z}^+ \times \mathbb{Z}^+ \times \mathbb{Z}^+ \to \mathbb{Z}_c$ où
   $\exp(a,b,c)=a^b \bmod c$
   Montrer que exp ∈ FP ?

4) Soit la fonction le plus grand diviseur commun $pgdc:\mathbb{N}\times\mathbb{N}\to\mathbb{N}$ où
   $pgcd(a,b)=\max\{d \geq 1 \mid d \text{ divise a et d divise b}\}$
   Montrer que pgdc ∈ FP ?

Solution :

1)
Notez tout d'abord que la multiplication par deux est facile à mettre en œuvre. Pour un entier binaire a, nous déplaçons simplement tous ses chiffres vers la gauche d'un seul endroit et ajoutons un zéro à droite de a. Nous notons cette opération par 2 × a. Considérez l'algorithme suivant.

Algorithm *Integer Multiplication.*

Input: n-bit binary integers $a = a_n \cdots a_1$ and $b = b_n \cdots b_1$.
Output: mult(a, b) in binary.
Algorithm:
$m \leftarrow 0$
for $i = 1$ to n
    if $b_i = 1$ then $m \leftarrow m + a$
    $a \leftarrow 2 \times a$
next i
output m.

Il est facile de voir que cet algorithme fonctionne. Le fait qu'il s'exécute en temps polynomial découle simplement de l'observation que la boucle for est répétée au plus n fois et chaque ligne de l'algorithme implique des opérations temporelles polynomiales de base sur des entiers d'au plus 2n bits. D'où mult ∈ FP.

2)
Si a <b sortie 0. Sinon, régler c ← 0 alors trouver $k = \lfloor \log a/b \rfloor$ (vérifier valeurs successives de k pour trouver celle qui satisfait $2^k b \leq a < 2^{k+1} b$). Ensuite, définissez a ← a - 2^k b et c ← c + 2^k. Répétez maintenant avec les nouvelles valeurs de a et c. Arrêtez une fois a <b puis sortez c. Dans une itération, nous réduisons a par un facteur d'au moins deux et nous devons vérifier au plus log a des valeurs possibles pour trouver k. Par conséquent, cet algorithme prend le temps $O(n^2)$ lorsque a est un entier de n bits. Ainsi div ∈ FP.

3)
Preuve: Nous utiliserons l'algorithme suivant.

Algorithm *Exponentiation.*

Input: binary integers $a = a_k \cdots a_1$, $b = b_m \cdots b_1$, $c = c_n \cdots c_1$.
Output: $a^b \bmod c$.
Algorithm:
$e \leftarrow 1$
for $i = 1$ to m
    if $b_i = 1$ then $e \leftarrow$ mult$(e, a) \bmod c$
    $a \leftarrow$ mult$(a, a) \bmod c$
next i
output e.

Puisque mult ∈ FP et tous les nombres entiers multipliés dans l'algorithme exponentiation sont bornés ci-dessus par c, alors chaque ligne de l'algorithme exponentiation peut être exécutée en temps polynomial. La boucle for est répétée m fois donc tout l'algorithme est en temps polynomial. D'où exp ∈ FP.

4)
Preuve: L'exemple évident d'algorithme de temps polynomial pour calculer le plus grand diviseur commun de deux entiers est l'algorithme d'Euclide.

Algorithm *Euclid's algorithm*

Input: binary integers $a \geq b \geq 1$.
Output: gcd(a, b).
Algorithm:
$r_0 \leftarrow a$
$r_1 \leftarrow b$
$i \leftarrow 1$
while $r_i \neq 0$
    $i \leftarrow i + 1$
    $r_i \leftarrow r_{i-2} \bmod r_{i-1}$
end-while
output r_{i-1}.

Si les entiers d'entrée sont a ≥ b ≥ 1 alors l'algorithme procède par division répétée avec reste. (Dans chaque cas, $q_i = \lfloor r_{i-2}/r_{i-1} \rfloor$.)

$$\begin{aligned}
a &= q_2 b + r_2, & 0 \leq r_2 < b,\\
b &= q_3 r_2 + r_3, & 0 \leq r_3 < r_2,\\
r_2 &= q_4 r_3 + r_4, & 0 \leq r_4 < r_3,\\
&\vdots & \vdots\\
r_{k-3} &= q_{k-1} r_{k-2} + r_{k-1}, & 0 \leq r_{k-1} < r_{k-2},\\
r_{k-2} &= q_k r_{k-1} + r_k, & r_k = 0.
\end{aligned}$$

L'algorithme s'arrête lorsque $r_k = 0$ et renvoie ensuite gcd (a, b) = r_{k-1}. Il est facile de vérifier que cet algorithme est correct. (Pour les entiers c et d, nous notons le fait que c divise d exactement par c | d.)
Notons d'abord que $r_k = 0$ implique que $r_{k-1} \mid r_{k-2}$ et donc $r_{k-1} \mid r_{k-3}$. En poursuivant le tableau d'équations, nous voyons que $r_{k-1} \mid r_i$ pour tout $2 \leq i \leq k - 2$ et donc $r_{k-1} \mid a$ et $r_{k-1} \mid b$. Ainsi $r_{k-1} \mid$ pgcd (a, b). Inversement si d | a et d | b alors en partant de la première équation descendante, on voit que $d \mid r_i$ pour $2 \leq i \leq k$ donc pgcd (a, b) $\mid r_{k-1}$, d'où r_{k-1} = pgcd (a, b) comme demandé.

Pour compléter la preuve, nous devons montrer qu'il s'agit d'un algorithme de temps polynomial. Chaque ligne de l'algorithme Euclidien peut être exécutée en temps polynomial (puisque les opérations arithmétiques de base impliquées peuvent être exécutées en temps polynomial).
Nous devons simplement prouver que le nombre de répétitions de la boucle while est limité par un polynôme dans la taille d'entrée : log a + log b.

Considérons les tailles relatives de r_i et r_{i+2} pour $2 \leq i \leq k - 2$. Puisque $q_{i+2} \geq 1$, $r_i = q_{i+2} r_{i+1} + r_{i+2}$ et $0 \leq r_{i+2} < r_{i+1}$, on a $r_{i+2} < (r_i/2)$. Par conséquent, la boucle while est répétée au plus $2\lceil \log a \rceil$ fois et donc pgdc ∈ FP .

Exercice 7 :
Prouver que P ⊆ PSPACE ⊆ EXP ?

Preuve: Nous avons déjà vu que P ⊆ PSPACE, il faut prouver PSPACE ⊆ EXP.

Supposons qu'un langage L appartient à PSPACE. Alors il existe un polynôme p (n) et un DTM M tels que M décide L et s'arrête après avoir utilisé au plus p (| x |) carrés de bande sur l'entrée x ∈ Σ_0^*. L'idée de base que nous utilisons est que puisque M s'arrête, il ne peut jamais entrer deux fois dans la même configuration (où une configuration se compose de l'état de la machine, de la position de la tête de lecture-écriture et du contenu de la bande) car si c'était le cas, elle serait dans une boucle infinie et donc ne jamais s'arrêter.

Pour être précis, considérons une entrée x ∈ Σ_0^n. Si |Σ | = m et |Γ | = k alors à tout moment du calcul la configuration actuelle de la machine peut être décrite en spécifiant :

(i) l'état actuel,
(ii) la position de la tête de lecture-écriture,
(iii) le contenu de la bande.

Il y a k possibilités pour (i) et, puisque le calcul utilise au plus p (n) carrés de bande, il y a au plus p (n) possibilités pour (ii). Maintenant, puisque chaque carré de bande contient un symbole de Σ et que le contenu de tout carré qui n'est pas visité par la tête de lecture-écriture ne peut pas changer pendant le calcul, il existe $m^{p(n)}$ possibilités pour (iii). Il y a donc au total $k\, p(n)\, m^{p(n)}$ configurations possibles pour M lors de son calcul sur l'entrée x.

L'une de ces configurations peut-elle être répétée ? Clairement non, car si c'était le cas, la machine serait entrée dans une boucle et ne s'arrêterait jamais. D'où

$$t_M(x) \leq kp(n)m^{p(n)}$$

Donc si q(n) est un polynôme satisfaisant

$$\log k + \log p(n) + p(n) \log m \leq q(n),$$

Alors L peut être décidé dans le temps $2^{q(n)}$ et donc L ∈ EXP.