Corrigé TD1

Habiba Drias

Exercice1.1

L'algorithme le plus simple que nous dénotons A1 découle directement de la définition d'un nombre premier. Rappelons qu'un nombre premier n est un nombre entier qui n'est divisible que par 1 et par lui-même. L'algorithme va donc consister en une boucle dans laquelle on va tester si le nombre n est divisible par 2, 3, …, n-1.

Algorithme A1 ;
début
    premier = **vrai** ;
    i = 2 ;
    tant que (i <= n-1) **et** premier **faire**
        si (n mod i = 0) **alors** premier = **faux sinon** i = i+1 ;
fin.

Le pire cas qui nécessite le plus long temps, correspond au cas où n est premier car c'est dans ce cas que la boucle s'exécute avec un nombre maximum d'itérations. Dans ce cas ce nombre est égal à n-2. La complexité est donc en O(n).

Nous savons que pour améliorer l'algorithme, il est judicieux d'arrêter la boucle à n/2 car si n est divisible par 2, il est aussi divisible par n/2 et s'il est divisible par 3, il est aussi divisible par n/3. De manière générale, si n est divisible par i pour i = 1 … [n/2] où [n/2] dénote la partie entière de n/2, il est aussi divisible par n/i. Il n'est donc pas nécessaire de vérifier qu'il est divisible par un nombre supérieur à n/2. Le deuxième algorithme est donc :

Algorithme A2 ;
début
    premier = **vrai** ;
    i = 2 ;
    tant que (i <= [n/2]) **et** premier **faire**
        si (n mod i = 0) **alors** premier = **faux sinon** i = i+1 ;
fin.

Le cas le plus défavorable qui nécessite le plus long temps correspond toujours au cas où n est premier et dans ce cas le nombre d'itérations est égal à n/2 -1. La complexité est donc en O(n)

Une autre amélioration possible consiste à tester si n est impair et dans ce cas dans la boucle, il ne faut tester la divisibilité de n que par les nombres impairs. L'algorithme A3 est donc comme suit :

```
Algorithme A3 ;
début
        premier = vrai ;
        si (n <> 2) et (n mod 2 = 0) alors premier = faux
        sinon si ( n <> 2) alors
                début
                        i=3 ;
                        tant que (i <= n-2) et premier  faire
                                si (n mod i = 0) alors premier = faux sinon i = i+2 ;
                fin
fin.
```

Le pire cas correspond au cas où n est premier et dans ce cas le nombre maximum d'itérations de la boucle est égal à [n/2] -2, la complexité est en O(n).

L'algorithme A4 peut être obtenu en hybridant A2 et A3 et on obtient :

```
Algorithme A4 ;
début
        premier = vrai ;
        si (n <> 2) et (n mod 2 = 0) alors premier = faux
        sinon si ( n <> 2) alors
                début
                        i=3 ;
                        tant que (i <= [n/2]) et premier  faire
                                si (n mod i = 0) alors premier = faux sinon i = i+2 ;
                fin
fin.
```

Le nombre d'itérations de la boucle pour un nombre premier est égal à la moitié du nombre d'itérations de A3, il est égal à [n/4] – 1. La complexité est donc en O(n).

Une bonne amélioration de l'algorithme serait d'arrêter la boucle non pas à [n/2] mais à $\sqrt{n}$ car en effet si n est divisible par x, il est aussi divisible par n/x et donc il serait judicieux de ne pas répéter le test de la divisibilité au-delà de x = n/x et dans ce cas $n = x^2$ et $x = \sqrt{n}$.

L'algorithme A5 s'écrit donc comme suit :

Algorithme A5 ;
début
premier = **vrai** ;
i = 2 ;
tant que (i <=[$\sqrt{n}$)] **et** premier **faire**
si (n mod i = 0) **alors** premier = **faux sinon** i = i+1 ;
fin.

Le nombre maximum d'itérations est égal à [$\sqrt{n}$)] -1, la complexité est en O($\sqrt{n}$). Enfin, on peut concevoir un algorithme en hybridant A5 et A3, on obtient l'algorithme A6 suivant:

Algorithme A6 ;
début
premier = **vrai** ;
si (n <> 2) **et** (n mod 2 = 0) **alors** premier = **faux**
sinon si (n <> 2) **alors**
début
i=3 ;
tant que (i <= [$\sqrt{n}$)]) **et** premier **faire**
si (n mod i = 0) **alors** premier = **faux sinon** i = i+2 ;
fin
fin.

Le nombre maximum d'itérations de la boucle est égal à ([$\sqrt{n}$)]/2 – 1). La complexité est donc en O($\sqrt{n}$).

<u>Récapitulatif</u>

Algorithme	**Nombre maximum d'itérations en fonction de n**	**Complexité théorique**	**Nombre réel d'itérations pour n = 990181**
A1	n-2	O(n)	990179
A2	[n/2] -1	O(n)	495089
A3	[n/2] -1	O(n)	495089
A4	[n/4] -2	O(n)	247563
A5	[$\sqrt{n}$)]-1	O($\sqrt{n}$)	994
A6	[$\sqrt{n}$/2] -2	O($\sqrt{n}$)	495

Nous remarquons que lorsque l'on change d'ordre de complexité, le temps réel change de grandeur : un nombre de 6 chiffres pour les algorithmes A1 à A4 et un nombre de 3 chiffres pour A5 et A6.
Pour conclure, nous faisons remarquer que de simples améliorations au niveau de l'algorithme initial qui est le plus basique, nous a conduit à écrire un code très rapide. En effectuant les différentes améliorations, nous avons fait chuter le nombre d'itérations de 990179 à 495.

Exercice1.2

1) 1 heure = 3600s = $3.6\ 10^3$s
1 jour = 86400s = $8.64\ 10^4$s
1 semaine = 604800s ≈ $6.05\ 10^5$s
1 mois= 2 592 000s ≈ $2.59\ 10^6$s
1 année =31 536 000 ≈ $3.15\ 10^7$s
1 siècle =3 153 600 000≈$3.15\ 10^9$s
1 millénaire = 31 536 000 000 ≈ $3.15\ 10^{10}$s

2) Le temps nécessaire au traitement des tailles du problème : n=10, n=100 et n=1000 pour une unité de temps égale à une milliseconde est montré dans le tableau suivant :

Algorithme	complexité	temps		
		n=10	n= 100	n=1000
A0	Ln n	0,002s	0,005s	0,009s
A1	$\sqrt{n}$	0,003s	0,01s	0,031s
A2	n	0,01s	0,1s	1s
A3	n^2	0,1s	10s	16mn40s
A4	n^3	1s	16 mn 40s	11j13h46mn40s
A5	n^4	10s	1j3h46mn40s	31 ans8 mois 15j 19h 3mn 28s
A6	2^n	1,02s	$3.2\ 10^{16}$ millénaires	$3.2\ 10^{286}$millénaires

2) Le temps nécessaire au traitement des tailles de problème n=10, n =100 et n=1000 pour une unité de temps égale à une microseconde est montré dans le tableau suivant :

Algorithme	complexité	temps		
		n=10	n= 100	n=1000
A0	Ln n	$2,3*10^{-6}$s	$4,6*10^{-6}$ s	$9,9*10^{-6}$s
A1	$\sqrt{n}$	$3,1*10^{-6}$s	10^{-5} s	$3,1*10^{-5}$s
A2	n	10^{-5}s	10^{-4} s	10^{-3}s
A3	n^2	10^{-4}s	0,01s	1s
A4	n^3	10^{-3}s	1s	16mn40s
A5	n^4	10^{-2}s	1mn40s	11j13h46mn40s
A6	2^n	10^{-3}s	$3.2\ 10^{13}$ millénaires	$3.2\ 10^{283}$ millénaires

1) Nous concluons que l'augmentation de la performance de la machine de calcul apporte les effets suivants :
   a. Améliore le temps de calcul pour des complexités polynomiales
   b. n'atténue en rien les valeurs prohibitives des complexités exponentielles des grandes tailles de problème et ne peut donc pas constituer une solution pour contourner le problème de l'explosion combinatoire.
   c. Pour les petites tailles, la fonction exponentielle est plus intéressante que certaines fonctions polynomiales (n = 10, A6 est plus rapide que A5)

Exercice 1.3

1) La fonction suivante calcule le pgcd selon la formule donnée :

```
fonction pgcd(a,b : entier) : entier ;
var
        x,y,q,r : entier;
début
        x := a;
        y :=b;
        q := x / y;
        r := x – q*y;
        tant que (r <> 0) faire
                début
                        x := y;
                        y := r;
                        q := x / y;
                        r := x – q*y ;
                fin;
        pgcd := y;
fin;
```

2) Complexité

Le pire cas correspond à la situation où à chaque itération, le quotient de la division de x par y est égal à 1 et donc au nombre maximum d'itérations. Dans ce cas le reste de la division est égal à la différence entre x et y puisque :

r := x – q*y = x – 1*y = x-y

Si x est supérieur à y, la séquence r, y et x fait partie de la suite de Fibonnacci, car les deux dernières itérations correspondent respectivement à r=1 et à r=0, qui coïncident avec l'initialisation de la suite de Fibonnacci. En résumé le cas le plus défavorable correspond au calcul du pgcd de deux nombres consécutifs de la suite de Fibonnacci F_{n-1} et F_{n-2} où

$F_n = F_{n-1} + F_{n-2}$, $F_1 = 1$ et $F_0 = 0$.

$F_n > 2*F_{n-2} > 2*2*F_{n-4} > \ldots > 2^k$

Où k est égal approximativement à la moitié de la longueur de la suite de Fibonnacci qui correspond au nombre d'itérations de la boucle. Le nombre d'itérations m est égal donc à 2k. Si a est supérieur à b, a=Fn et par conséquent :

$a > 2^k$

$\ln a > \ln 2^k = k \ln 2$

$k < (1/\ln 2) \ln a$

d'où : $m < (2/\ln 2) \ln a$, La complexité est donc en O(ln a).

3) Version récursive

```
fonction pgcd(a,b : entier) : entier ;
var
        q,r : entier ;
début
        si (b = 0) alors pgcd := a sinon début q := a div b; {division entière }
                                                r := a – q*b;
                                                pgcd := pgcd(b,r);
                                        fin;
fin;
```

La complexité de la fonction pgcd est la même que celle calculée pour la $2^{ème}$ question.

Exercice 1.4

1) Algorithme :

```
var     i,j,c : entier ;
        T[1..max] : tableau d'entiers;
début
        i:=1,
        j:= max ;
        tant que (T[i]=T[j]) et (i<j) faire
        début  i:= i+1;
                j := j-1;
        fin;
        si (i>j) alors écrire('mot correct') sinon écrire ('mot incorrect');
fin;
```

Le nombre d'instructions exécutées pour un mot du langage est de :

- 2 instructions d'initialisation
- 2 instructions de la 2ème boucle
- 1 instruction de fin de programme

Au total et dans le pire cas lorsque le mot est correct, on aura : 2+2*n/2+1 = 3+n

La complexité est donc en O(n)

2) Le programme assembleur est le suivant :

```
            MOV     T, R0
Boucle1:    INPUT   R1
            SUB     #2, R1
            JZ      L1
            ADD     #2, R1
            MOV     R1, (R0)
            ADD     #1, R0
            JMP     Boucle1
L1:         MOV     T, R1
Boucle2 :   SUB     R1, R0
            JGT     R0, Succès
            SUB     (R1), (R0)
            JZ      (R0), L2
            JMP     Echec
L2:         ADD     #1, R1
            SUB     #1, R0
            JMP     Boucle2
```

```
Echec:      OUTPUT    #0
            JMP       Fin
Succès:     OUTPUT    #1
Fin         STOP
T:
```

Le nombre d'instructions exécutées pour reconnaitre un mot est égal à:

- 1 instruction d'initialisation
- 6 instructions de la 1ère boucle
- 1 instruction d'initialisation de la 2ème boucle
- 8 instructions de la 2ème boucle
- 2 instructions de fin de programme

Au pire cas et si la longueur d'un mot est égale à n, ce nombre est égal au total à : 1 + 6*n + 1 +8*n/2 + 2 = 4 + 6*n + 4*n
= 6 + 10*n
La complexité temporelle est donc en O(n).
L'espace occupé est égal à 2 + n, pour l'utilisation des 2 registres R0, R1 et de la zone T de taille n. La complexité spatiale est donc en O(n).

3) **Machine de Turing** :

Première solution : Machine à un seul ruban :

On suppose que le mot se trouve initialement sur le premier ruban. Nous aurons besoin de définir 8 états avec les rôles suivants :

- q0 : état initial, met à blanc le 1er bit rencontré et transite vers q1 si le bit est égal à 1, à q2 s'il est égal à 0
- q1 : sert à comparer le 1er 1 de la chaine avec le dernier 1
- q2 : sert à comparer le 1er 0 de la chaine avec le dernier 0
- q3 : sert à mettre à blanc le dernier 1 de la chaine
- q4 : sert à mettre à blanc le dernier 0 de la chaine
- q5 : sert à se déplacer vers la gauche et positionner la tête de lecture vers le 1er bit de la chaine restante
- q6 signale un succès
- q7 signale un échec

La fonction de transition est alors la suivante :
(q0,1) → (q1, b, R)
(q0,0) → (q2, b, R)
(q1,1) → (q1, 1, R)

(q1,0) → (q1, 0, R)
(q1,b) → (q3, b, L)
(q3,1) → (q5, b, L)
(q3,0) → (q7, b, L) échec
(q3,b) → (q6, 1, L) succès
(q7,1) → (q7, b, L)
(q7,0) → (q7, b, L)
(q7,b) → (q7, 0, S)
(q5,1) → (q5, 1, L)
(q5,0) → (q5, 0, L)
(q5,b) → (q0, b, R)
(q2,1) → (q2, 1, R)
(q2,0) → (q2, 0, R)
(q2,b) → (q4, b, L)
(q4,0) → (q5, b, L)
(q4,1) → (q7, b, L) échec
(q4,b) → (q6, 1, L) succès

La taille du problème est celle du palindrome et est égale à n. Le pire cas correspond au cas où la donnée est un palindrome car c'est dans ce cas que la machine fait le maximum de transitions. Le nombre de transitions effectuées pour reconnaitre un palindrome est de :
$2(n+1)+2n+2(n-1)+\ldots+2 = \sum_1^{n+1} 2i = 2\sum_1^{n+1} i = 2(n+1)(n+2)/2 = (n^2+3n+2)$.
La complexité temporelle est donc en $O(n^2)$ et la complexité spatiale est en O(n).

<u>Deuxième solution : Machine à 2 rubans</u>
Une autre machine peut être conçue avec 2 rubans. Dans un des rubans, on stocke le palindrome puis on le copie sur l'autre ruban et enfin on les parcourt les deux rubans dans les sens opposés pour les comparer.
(q0,0,b) → (q1, (0,S), (b,R))
(q0,1,b) → (q1, (1,S), (b,R))
(q1,0,b) → (q1, (0,R), (0,R))
(q1,1,b) → (q1, (1,R), (1,R))
(q2,b,0) → (q2, (b,S), (0,L))
(q2,b,1) → (q2, (b,S), (1,L))
(q2,b,b) → (q3, (b,L), (b,R))
(q3,0,0) → (q4, (0,S), (b,R))
(q4,0,0) → (q5, (0,S), (b,L))
(q5,0,0) → (q6, (1,S), (b,S)) succès

(q5,0,1) → (q7, (0,S), (b,S)) échec
(q4,0,1) → (q3, (b,L), (1,S))
(q4,0,0) → (q3, (b,L), (0,S))
(q3,1,1) → (q4, (1,S), (b,R))
(q4,1,b) → (q(, (1,S), (b,L))
(q5,1,1) → (q6, (1,S), (b,S)) succès
(q5,1,0) → (q7, (0,S), (b,S)) échec
(q4,1,0) → (q3, (b,L), (0,S))
(q4,1,1) → (q3, (b,L), (1,S))

Le nombre de transitions effectuées est égal à :
1 pour insérer un blanc au début du deuxième ruban,
n pour copier le mot dans le deuxième ruban,
n+1 pour déplacer la tête de lecture du deuxième ruban vers l'extrémité gauche,
1 pour démarrer la comparaison,
2n pour la comparaison.
Au total, il est égal à 1+n+(n+1)+1+2n = 4n +3. La complexité est donc en O(n).

Exercice 1.5

1) **programme** L ;
var
    ecart, c : **entier** ;
début
    lire(c);
    écart := 0;
    tant que (c=0) ou (c=1) **faire**
        début
            si (c=0) **alors** écart := écart + 1
                sinon si (c=1) **alors** écart := écart – 1 ;
        fin;
    si (écart = 0) **alors écrire** ('mot correct') **sinon écrire** ('mot incorrect')
fin.
Le nombre d'instructions exécutées pour reconnaitre un mot du langage est égal à 2+ 3n+1.
La complexité de l'algorithme est donc en O(n).

2) Le programme assembleur est le suivant :

```
                MOV     #0, R0
Boucle :        INPUT   R1
                JZ      R1, L0
                SUB     #1, R1
                JZ      R1, L1
                JMP     FinBoucle
L0:             ADD     #1, R0
                JMP     Boucle
L1 :            SUB     #1, R0
                JMP     Boucle
FinBoucle :     JZ      R0, succès
                OUTPUT  #0
                JMP     Fin
succès :        OUTPUT  #1
Fin :           STOP
```

Le nombre d'instructions exécutées pour reconnaitre un mot du langage est égal à :

1 + 4n + 4(au maximum) = 5 + 4n

La complexité est donc en O(n)

3) La machine de Turing que nous proposons utilise 3 rubans. En lisant le mot stocké sur le 1er ruban de gauche à droite, la machine place les 0 rencontrés dans le 2ème ruban et les 1 dans le 3ème. En revenant en arrière, elle vérifie que le nombre de 0 du 2ème ruban est égal au nombre de 1 du 3ème ruban d'où la fonction de transition suivante :

(q0, 0, b,b) → (q1,(0,S),(b,R),(b,R))
(q0, 1, b,b) → (q1,(1,S),(b,R),(b,R))
(q0, b, b,b) → (q3,(1,S),(b,R),(b,R)) succès: chaine vide
(q1, 0, b,b) → (q1,(b,R),(0,R),(b,S))
(q1, 1, b,b) → (q1,(b,R),(b,S),(1,R))
(q1, b, b,b) → (q2,(b,L),(b,L),(b,L))
(q2, b, 0,1) → (q2,(b,L),(b,L),(b,L))
(q2, b, b,b) → (q3,(1,S),(b,S),(b,S)) succès
(q2, b, 0,b) → (q4,(0,S),(b,L),(b,L)) échec
(q2, b, b,1) → (q4,(0,S),(b,L),(b,L)) échec

La complexité temporelle est en O(n), de même est la complexité spatiale.

Exercice 1.7

1) Les différentes itérations sont les suivantes :

0	1	2	3	4	5	6	7	8	9	10
1	0	0	3	1	5	1	7	1	9	1
2	0	0	0	1	5	1	7	1	1	1
3	0	0	0	1	1	1	7	1	1	1
4	0	0	0	1	1	1	1	1	1	1

2) L'algorithme est le suivant :

```
var i, j, r: entier ;

écrire ('les nombres premiers sont);
T[1] :=0;

début
        pour i := 2 à n faire
        début
                si T[i] <> 1 alors début
                                        T[i] := 0;
                                        écrire (i);
                                fin;
                pour j := i+1 à n faire
                        si (T[j] mod i = 0) alors T[j] :=1;
        fin;
fin;
```

3) Il existe deux boucles dans l'algorithme. La boucle externe s'exécute de 2 à n donc (n-1) fois. La boucle interne par contre s'exécute un nombre de fois qui dépend de i qui varie de n-2 à 1. La complexité est donc de : $\sum_1^{n-2} i = \frac{(n-2)(n-1)}{2}$. Elle est en O($n^2$).

4) Le programme assembleur :

```
        MOV     #10, R4
        MOV     #2, R0
L5:     SUB     R4, R0
        JZ      R0, Fin
        ADD     R4, R0
        SUB     #1, T[R0]
        JZ      T[R0], L0
        MOV     #0, T[R0]
        OUTPUT  R0
L0:     MOV     R0, R1
L3:     ADD     #1, R1
        MOV     T[R1], R2
        MOV     T[R1], R3
        DIV     R0, R2
```

```
        IDIV    R0, R3
        SUB     R2, R3
        JZ      R3, L1
        JMP     L2
L1:     MOV     #0, T[R1]
L2:     SUB     R4, R1
        JZ      R1, L4
        ADD     R4, R1
        JMP     L3
L4:     ADD     #1, R0
        JMP     L5
Fin :   STOP
T :     0
        1
        2
        3
        4
        5
        6
        7
        8
        9
        10
```

5) La complexité du programme est identique à celle de l'algorithme de la question 2, car le programme est constitué de deux boucles imbriquées fonctionnant de la même manière que les boucles de l'algorithme. Elle est par conséquent en $O(n^2)$