e3a 2015 -Option Informatique : Un corrigé

Exercice 1

1. (a) Si on note C_n le nombre d'opération pour l'appel de `puissance n k`.
On a la relation : $C_{n+1} = C_n + A$ où A est le coût d'un test, d'une multiplication et d'un appel récursif.

la complexité de cet algorithme est linéaire en $O(n)$ pour l'appel de `puissance n k`

(b) i. Appel : `puissance2 2 7` ;n=2 ; k= 7 ; test `k>1` est vrai ; `7/2=3` ; `x=puissance2 2 3`
Appel : `puissance2 2 3` ; n=2 ; k= 3 ; test `k>1` est vrai ; `3/2=1` ; `x=puissance2 2 1`
Appel : `puissance2 2 1` ; n=2 ; k= 1 ; test `k>1` est faux ; `puissance2 2 1` renvoie 2
Execution : `puissance2 2 3` ; x=2 ; test `k mod 2 = 0` est faux ; retour 2*2*2
`puissance2 2 3` renvoie 8
Execution : `puissance2 2 7` ; x=8 ; test `k mod 2 = 0` est faux ; retour 8*8*2
L'exécution du programme `puissance2` sur l'entrée $(2, 7)$ renvoie 128.

ii. Appel : `puissance2 2 8` ;n=2 ; k= 8 ; test `k>1` est vrai ; `8/2=4` ; `x=puissance2 2 4`
Appel : `puissance2 2 4` ;n=2 ; k= 4 ; test `k>1` est vrai ; `4/2=2` ; `x=puissance2 2 2`
Appel : `puissance2 2 2` ; n=2 ; k= 2 ; test `k>1` est vrai ; `2/2=1` ; `x=puissance2 2 1`
Appel : `puissance2 2 1` ; n=2 ; k= 1 ; test `k>1` est faux ; `puissance2 2 1` renvoie 2
Execution : `puissance2 2 2` ; x=2 ; test `k mod 2 = 0` est vrai ; retour 2*2
`puissance2 2 2` renvoie 4
Execution : `puissance2 2 4` ; x=4 ; test `k mod 2 = 0` est vrai ; retour 4*4
`puissance2 2 4` renvoie 16
Execution : `puissance2 2 8` ; x=16 ; test `k mod 2 = 0` est vrai ; retour 16*16
L'exécution du programme `puissance2` sur l'entrée $(2, 7)$ renvoie 256.

iii. On montre par récurrence sur $p \in \mathbb{N}^*$: « pour tout $k \in \mathbb{N}^*$ tel que $k < 2^p$, le nombre d'appels récursifs est inférieur à $p - 1$ »
Initialisation : Pour $p = 1$, la seule valeur de k à envisagée est 1 et au premier test la fonction `puissance2 n 1` renvoie `n` sans appel récursif
On a bien $1 - 1 = 0$
Initialisation : Soit $p \in \mathbb{N}^*$ tel que la propriété soit vraie au rang p
On considère $k \in \mathbb{N}^*$ tel que $k < 2^{p+1}$
si k>1, l'exécution de `puissance2 n k` lance l'appel récursif `puissance2 n (q)`
en notant q le quotient de k par 2
Comme $q < 2^p$; il y a donc au plus p nouveaux appels récursifs
ce qui termine l'hérédité.
Conclusion : On a montré par récurrence la propriété souhaitée
Pour $k \in \mathbb{N}^*$, on peut trouver $p \in \mathbb{N}$ tel que $2^p \leq k < 2^{p+1}$
Le nombre d'appels récursifs est au plus de p d'après la récurrence.
on a $p \leq \log_2 k < p + 1$ car $\log_2$ est strictement croissante

le nombre d'appels récursifs dans `puissance2 n k` est au plus de $\log_2 k < \log_2 k + 1$

La seule instruction a posé des problèmes de terminaison étant l'appel récursif (pas de boucle conditionnelle ....) et ceux ci étant en nombre fini

On en déduit que le programme termine

iv. On montre par récurrence sur
$p \in \mathbb{N}^*$: « pour tout $k \in \mathbb{N}^*$ tel que $k < 2^p$, `puissance2 n k` renvoie n^k »
Initialisation : Pour $p = 1$, la seule valeur de k à envisagée est 1 et au premier test la fonction `puissance2 n 1` renvoie `n`
On a bien vérifié l'initialisation.
Initialisation : Soit $p \in \mathbb{N}^*$ tel que la propriété soit vraie au rang p
On considère $k \in \mathbb{N}^*$ tel que $k < 2^{p+1}$
si $k < 2^p$, il n'y a rien à faire par hypothèse.
Sinon on écrit $k = 2q + r$ avec $r \in \{0, 1\}$ d'après le théorème de la division euclidienne on a $1 \leq q < 2^p$
l'appel de `puissance2 n q` renvoie $x = n^q$
donc si $r = 1$, `puissance2 n k` renvoie $x \times x \times n = n^q \times n^q \times n = n^{2q+1} = n^k$
donc si $r = 0$, `puissance2 n k` renvoie $x \times x = n^q \times n^q \times n = n^{2q} = n^k$
ce qui prouve l'hérédité.
Conclusion : On a montré par récurrence la propriété souhaitée
Ce programme est correct

v. Quand $k \to +\infty$, on a $1 = o(\log_2 k)$
la complexité de ce programme est en $O(\log_2(k))$ d'après (iii)

2. (a)
```
let test_puissance n k=let m= ref 2 in
                       let p= ref 0 in
                       while !p <n do  incr m ;
                                       p:= puissance2 !m k
                                       done ;
                       !p = n ;;
```

(b) i. On a $\log_2(n) = k \log_2(m)$ comme $m \geq 2$ on a $\log_2(m) \geq 1$ et comme $k \geq 1$
On obtient $k \leq \log_2(n)$

ii.
```
let test_puissance_entiere n=let res= ref false in
                             let p = ref 4 in
                             let k = ref 2 in
                             while !p <= n
                                      do
                                      res:= !res || test_puissance n !k;
                                      incr k ; p:= 2* !p
                                      done ;
                            !res ;;
```

iii. Lors de l'exécution de `test_puissance_entiere n`, à part les opérations en temps constant il y a une boucle conditionnelle (`for`) que l'on répète au plus $\log_2(n) + 1$ fois. L'exécution de chacune de la boucle a une complexité majorée par $O(n \log_2(k))$ car il y a au plus n appels de `puissance2 m k` lors de l'exécution de `test_puissance n !k`.
la complexité de ce programme est au plus de $O(n \log_2(n) \log_2(k))$
ou mieux de $O(n \log_2(n) \log_2(\log_2(n)))$ en utilisant la majoration (i).

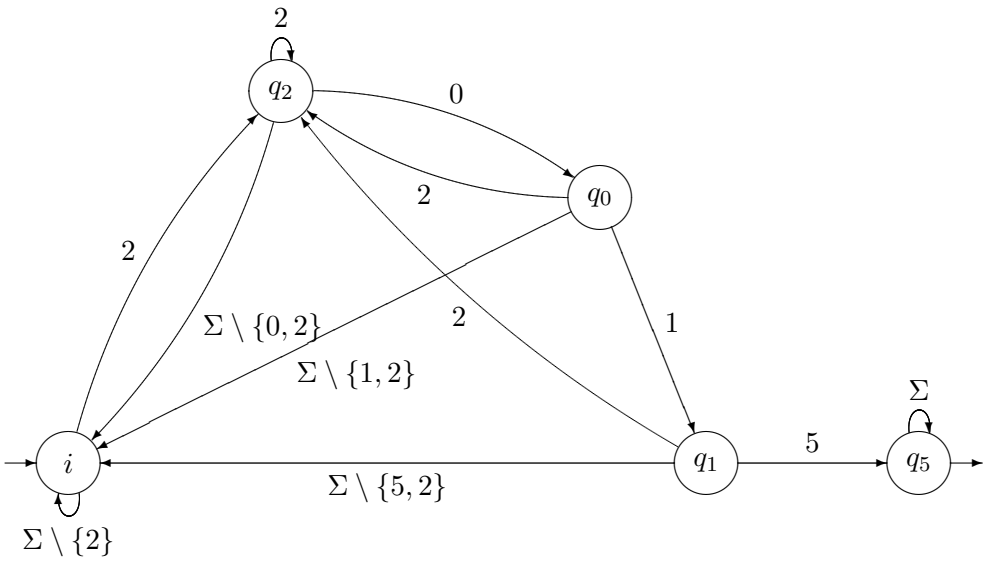
iv.
```
let rec liste1_puissances_entieres n=match n with
                                    |1->[]
                                    |_-> if test_puissance_entieres n
                                       then
                                       n::(liste1_puissances_entieres (n-1))
                                       else
                                       liste1_puissances_entieres (n-1) ;;
```

(c) Deux idées similaires utiles transposable pour le crible d'Erathosthène :
pour $n, k, m \in \mathbb{N}$ tels que $k > 1$, $m > 1$, $n \geq 1$ et $n \geq m^k$ on a
(i) $m \leq \sqrt{n}$
(ii) si il existe $r > 1$ et $l > 1$ tel que $m = r^l$ alors $r < m$ et $m^k = r^{kl}$

```
let liste2_puissances_entieres n=
     let rec tab_to_list t i l = match i with
            |1 -> l
            |_ -> if t.(i) then tab_to_list t (i-1) (i::l)
                           else tab_to_list t (i-1) l
                           in
     let tab = make_vect (n+1) false in
     let m=ref 2 in
     let p=ref 4 in
     while !p<=n do
                 if not tab.(!m) then
                              begin
                              while !p<= n do
                                          tab.(!p)<-true ;
                                          p:= !p * !m
                                          done
                              end;
                 incr m ;
                 p:= !m * !m
                 done ;
     tab_to_list tab n [];;
```

Exercice 2

1. Un automate déterministe complet qui reconnaît exactement les entiers naturels non nuls ayant la propriété P_{2015} peut être : $(\Sigma, Q, i, F, \delta)$ où $Q = \{i, q_2, q_0, q_1, q_5\}$; $F = \{q_5\}$ et les transitions sont représentées par le dessin ci-dessous :



2. (a) Ici, il semble que l'on qu'on considère que a peut s'annuler.

   La fonction G peut prendre $10000 = 10^4$ valeurs

   (b) On a $G(bcde) = 10G(abcd) - a \times 10^4 + e$

   (c)
```
let decalG G e=let a= G/1000 in
                10*(G-1000*a)+e;;
```

   (d)
```
let  testG_motif t =
     let n=vect_length t in
     let res=ref false in
     let GC=ref 0 in
     if n>=4 then begin
                 GC:= 1000*t.(0)+100*t.(1)+10*t.(2)+t.(3) ;
                 for i=0 to (n-5) do
  (* un for pour pouvoir réutiliser facilement le code par la suite *)
                                  res:= !res || !GC=2015 ;
                                  GC:=decalG !GC t.(i+4)
                                  done;
                 res:= !res || !GC=2015
                 end;
     !res;;
```

   (e) La fonction `decalG` s'effectue en temps constant. Lors de l'appel de la fonction `testG_motif`, ce qui s'exécute avec $n-4$ boucles itératives

La complexité de l'algorithme est en $O(n)$ où n est la taille du tableau

(f)
```
let  nbgG_motif t =
     let n=vect_length t in
     let res=ref 0 in
     let GC=ref 0 in
     if n>=4 then begin
                  GC:= 1000*t.(0)+100*t.(1)+10*t.(2)+t.(3) ;
                  for i=0 to (n-5) do
                                   if  !GC=2015 then incr res ;
                                   GC:=decalG !GC t.(i+4)
                                   done;
                  if  !GC=2015 then incr res
                  end;
     !res;;
```

3. (a) Soit $x \in \mathbb{N}$. Si $x < 10$, on peut prendre le mot $\overline{000x}$ et on a $H(\overline{000x}) = x$.
Si $x = 10$ on a $H(\overline{0010}) = 10 = x$
Donc toutes les valeurs entières entre 0 et 10 peuvent être atteintes.
la fonction H peut prendre 10 valeurs distinctes

(b)
```
let  modonz n=let r=(n mod 11) in if r>=11 then r-11
                                  else if r<0 then r+11
                                                   else r;;
 (*pour être certain d'obtenir un résultat entre 0 et 10*)
let calculH t=modonz ( t.(0)*1000+t.(1)*100+t.(2)*10+t.(3) )  ;;
```

(c) On a $b*1000+c*100+d*10+e \equiv 10(H(abcd)-a(10)^3)+e \equiv -H(abcd)-a+e$ [11] car $10 \equiv -1$ [11]

donc
$$H(bcde) = \begin{cases} -H(abcd)-a+e & \text{si} \quad 0 \leq H(abcd)+a+e \\ -H(abcd)-a+e+11 & \text{si} \quad -11 \leq H(abcd)+a+e < 0 \\ -H(abcd)-a+e+22 & \text{si} \quad -19 \leq H(abcd)+a+e < -11 \end{cases}$$

(d) Il s'agit de tester l'existence du motif $\overline{2015}$ mais en s'interdisant d'utiliser chaque valeur du tableau plus de trois fois.

Je teste le tableau `t` par tranche de quatre éléments et en se déplaçant a priori d'un cran si le test échoue.

L'exploration du tableau s'arrête dès que le motif est trouvé.

On remarque que pour tester un tableau `[|a;b;c;d|]` il suffit de connaître la valeur de $H(abcd)$ et les valeurs de b, c, d.

En effet, l'application $a \in [0,9] \mapsto H(\overline{abcd}) \in [0,10]$ est injective, ce qui donne :

$$\overline{abcd} = \overline{2015} \Leftrightarrow (H(abcd) = 2 \text{ et } b = 0 \text{ et } c = 1et \text{ et } d = 5)$$

On effectue les tests dans cet ordre mais si $b = 0$ et que les quatre éléments ne conviennent pas on pourra se déplacer de deux crans dans le tableau pour le prochain test sur quatre éléments consécutifs du tableau.

(de façon analogue de trois crans, si $b = 0$, $c = 1$ mais $d \neq 5$)

La variable `cran` sert à connaître le nombre de pas à parcourir dans le tableau pour le prochain test.

Dans le programme ci-dessous, les valeurs du tableau ne sont utilisées au plus que trois fois,

- une éventuelle première fois pour la faire rentrer dans la fonction H
dans la ligne `HC:= modonz ( t.(0)*1000+t.(1)*100+t.(2)*10+t.(3) );*` ou dans `HC:= decalH !HC t.(!j+4) t.(!j) ;`

- une éventuelle deuxième fois pour tester une égalité avec deux possibilités

  si le test est positif le prochain test n'aura pas lieu car grâce à l'utilisation de la variable `cran`, on se déplcera assez loin dans le tableau

  sinon en cas de test négatif, la valeur du tableau se retrouvera en première position dans la prochaine tranche de quatre éléments à tester et donc ne sera pas tester non plus

- Enfin une éventuelle troisième fois, pour faire "sortir" la valeur lors de l'utilisation de `decalH`

Une solution tirée par les cheveux :

```
let  modonz n=let r=(n mod 11) in if r>=11 then r-11
                                  else if r<0 then r+11
                                 else r;;
let decalH H e a=modonz(-H+e-a);;
let testH_motif t= let n=vect_length t in
                   let res=ref 0 in
                   let HC=ref 0 in
                   let i= ref 0 in
                   let cran= ref 1 in
                   if n>=4 then
                       HC:= modonz ( t.(0)*1000+t.(1)*100+t.(2)*10+t.(3) );
                           while  !i<= (n-4) && !res=0
                             do
                               if  !HC=2 then
                                 begin
                                 if t.(!i+1)=0 then
                                   begin incr cran  ;
                                   if t.(!i+2)=1 then
                                     begin incr cran ;
                                     if t.(!i+3)=5 then
                                               res:=1;
                                               cran:= -1
                                     end
                                   end
                                 end;
                               let j=ref !i in
                                while !res=0 && !j <(n-4) && !j <= !i+ !cran
                                       do  HC:= decalH !HC t.(!j+4) t.(!j) ;
                                           incr j
                                       done ;
                               i:=!i+ !cran ;
                               cran:=1
                             done ;
                      !res;;
```

Cette fonction ne "lit" que trois fois au plus chacun des caractères de l'écriture en base 10 de n : une fois pour l'évaluation de `GC` (G courant) en entrée, une autre fois pour le sortir de l'évaluation de `GC` et enfin cette valeur intervient dans un test au plus une fois grâce à la variable `cran`.

(e) Cet algorithme s'effectue en $O(n)$ où n est la taille du tableau.

Exercice 3

1. (a)
```
let nb tab a = let N=vect_length tab in
               let res=ref 0 in
               for i=0 to (N-1) do
                                if tab.(i)=a then incr res
                                done;
               !res;;
```

   (b) La complexité de `nb` est de $O(N)$ où `N` est la taille du tableau celle-ci ne dépend pas de `k`

   (c)
```
let elu1 tab=let N=vect_length tab in
             let res = ref (-1) in
             let i = ref 0 in
             while !i < k && !res= -1
                     do
                     if 2*(nb tab !i)> N then res:= !i;
                     incr i
                     done;
                     !res;;
```

   (d) La complexité de `nb` est de $O(Nk)$

   En effet, il y a k itération et chaque itération s'effectue en $O(N)$.

2. (a) Soit a donné comme élu de `tab`.

   Par l'absurde, on suppose que a ne soit donné élu ni de `miGauche tab` ni de `miDroite tab`.

   On note respectivement n_g et n_d le nombre d'occurrences de a dans les tableaux `miGauche tab` et `miDroite tab`.

   on aurait donc $2n_g \leq \lfloor N/2 \rfloor$ et $2n_d \leq N - \lfloor N/2 \rfloor$

   ainsi $2(n_g + n_d) \leq N$ ce qui est contradiction avec le fait que a est élu de `tab`.

   On a bien démontré que

   si `tab` donne a comme élu alors il est aussi élu par `miGauche tab` ou par `miDroite tab`

   (b) Je propose égaement les fonctions de demi tableaux afin de pouvoir analyser leurs complexités.

```
let miGauche tab=let n=vect_length tab in
                 let tg=  make_vect 0 (n/2) in
                 for i=0 to ((n/2)-1) do tg.(i)<- tab.(i)done ;
                 tg;;
let miDroite tab=let n=vect_length tab in
                 let td=  make_vect 0 (n-(n/2)) in
                 for i=(n/2) to (n-1) do td.(i-(n/2))<- tab.(i)done ;
                 td;;
let rec elu2 tab=let n=vect_length tab in match n with
                 |0 -> (-1,0)
                 |1 -> (tab.(0),1)
                 | _->    let tg =miGauche tab in
                          let td =miDroite tab in
                          let (ag,ng)=elu2 tg in
                          let (ad,nd)=elu2 td in
                          let mg= nb td ag +ng in (* si ng=0 alors mg=0*)
                          let md= nb tg ad +nd in (* si nd=0 alors md=0*)
                          if 2*mg > n then (ag,mg)
```

```
                    else if 2*md > n then (ad,md)
                                     else (-1,0);;
```

(c) Si on note C_p le coût en nombre d'opérations de `elu2` pour un tableau de taille 2^p
On a la relation $C_{p+1} \leq 2C_p + A2^{p+1} + B$
Cette relation provient des deux appels récursifs ainsi que des complexités linéaire des fonctions `miGauche`, `miDroite` et `nb`.
On a donc il existe $A' > 0$ tel que pour tout p, $\frac{C_{p+1}}{2^{p+1}} \leq \frac{C_p}{2^p} + A'$
donc $C_p = O(p2^p)$
En remarquant que le coût est croissant, on obtient
La complexité de cette fonction en fonction de N est $O(N \log_2 N)$

3. (a) On suppose que le tableau T donne a élu.
On note n le nombre d'occurrences de a. On a $n > N/2$.
Soit b distincts de a. b apparaît au plus $N - n$.
si le tableau T donne a élu alors a est un postulant de T

(b) Soit a postulant de T pour la valeur $n > N/2$
alors tout autre valeur apparaît au plus $N - n < N/2$ et ne peut être élu
si a est un postulant de T, alors aucun autre élément de T ne pourrait être élu

(c) Le tableau $[1, 2, 3]$ en Caml : `[|1;2;3|]` ; admet trois postulants et n'admet aucun élu
En effet l'entier 1 (par exemple) en est un postulant pour la valeur $2 > 3/2$ et les entiers apparaissent au plus une (3-2) fois.
Le tableau $[1, 2]$ en Caml `[|1;2|]` ; ne contient aucun postulant
Par l'absurde on suppose qu'il existe un postulant a pour une valeur n dans ce tableau de longueur $N = 2$
On a $n > N/2$ et $0 \leq N - n$ (en effet tout entier autre a doit apparaître au plus $N - n$ fois)
donc $n = 2$ et ainsi $N - n = 0$
Prenons $b \in \{1, 2\}$ tel que $b \neq a$, b apparaît au plus 0 fois.
Absurde

(d) On remarque que $N/2$ est un entier car N est pair.

i. Soit b un entier distinct de a.
b apparaît au plus $N/2 - l$ fois dans TG car a y est postulant pour la valeur l.
b apparaît au plus $\left\lfloor \frac{N}{4} \right\rfloor$ dans TD car TD n'a pas d'élu.
Ainsi b apparaît au plus $\left\lfloor \frac{N}{4} \right\rfloor + N/2 - l$ fois dans T
On note $n = N - \left(\left\lfloor \frac{N}{4} \right\rfloor + N/2 - l \right) = N/2 + l - \left\lfloor \frac{N}{4} \right\rfloor$
et b apparaît au plus $N - n$ fois
De plus $\left\lfloor \frac{N}{4} \right\rfloor \leq N/4$ et $l > N/4$ car a est postulant dans un tableau de longueur $N/2$ pour la valeur l
donc on a $n > N/2$
Ensuite a apparaît au plus $l + \left\lfloor \frac{N}{4} \right\rfloor$ dans T
or $l + \left\lfloor \frac{N}{4} \right\rfloor \leq n = N/2 + l - \left\lfloor \frac{N}{4} \right\rfloor \Leftrightarrow \left\lfloor \frac{N}{4} \right\rfloor \leq N/4$
ceci permet d'affirmer que a apparaît au plus n fois et que tout autre valeur apparaît au plus $N - n$ fois et que $n > N/2$

$\boxed{a \text{ est un postulant de } T \text{ pour la valeur } n = N/2 + l - \left\lfloor \frac{N}{4} \right\rfloor}$

ii.

A.
a apparaît au plus $n = m + l$ fois dans T.
Tout entier distinct de a apparaît au plus $N/2 - m + N/2 - l = N - n$
De plus $n = m + l \leq N/4 + N/4 = N/2$
$\boxed{a \text{ est un postulant de } T \text{ pour la valeur } n = m + l}$

B.
b apparaît au plus m fois dans TD et au plus $N/2 - l$ fois dans TG
donc b apparait au plus $N/2 + m - l$ fois dans T.
De plus $N/2 + m - l > N/2$ car $m > l$
Soit c entier distinct de b.
c apparaît au plus l fois dans TG car $N - l < l$ et au plus $N/2 - m$ fois dans TD
donc au plus $N/2 - m + l = N - (N/2 + m - l)$ dans T.
Ce qui prouve $\boxed{b \text{ est un postulant pour la valeur } N/2 + m - l \text{ de } T}$

C.
Par l'absurde on suppose que T admet un élu.
Celui-ci serait également un élu de TD ou TG (d'après 2(a))
Sans perte de généralités on suppose que l'élu est a (deux tableaux de taille $N/2$, $m = l$)
On note n le nombre d'occurrences dans T.
On a donc $n > N/2$ et $n \leq m + N/2 - m$ (a postulant à gauche et b postulant à droite)
Absurde
$\boxed{T \text{ ne donne pas d'élu}}$ dans ce cas

(e) Je propose deux fonctions `postulant` mais a priori comme on ne connaît pas la complexité des oracles il faut se méfier de la première version vues les contraintes de complxité par la suite :

```
let rec  postulant1 tab=let N=vect_length tab in match N with
                        |0-> (-1,0)
                        |1-> (tab.(0),1)
                        | _-> let tg =miGauche tab in (*complexité ?*)
                              let td =miDroite tab in (*complexité ?*)
                              let (a,l)=postulant1 tg in
                              let (b,m)=postulant1 td in
                              if a=b then (a, m+l)
              (*valable même si  TD et TG répondent "pas d'élu" 3diiA et2a*)
                              else if l*m=0 then
                                (max a b,  N/2 +  l+ m  - N/2)
                              (* TD ou bien TG répond "pas d'élu" voir 3di*)
                              else if  m>l then (b , N/2 + m - l) (*3diiB*)
                              else if  m<l then (a , N/2 - m + l) (*3diiB*)
                              else (-1,0);; (*3diiC*)

let   postulant tab=(* version dont je contrôle la complexité*)
             let rec aux t  d f=
            (*fonction qui cherche un postulant sur une portion de tableau*)
             let N= f- d in match N with
                        |0-> (-1,0)
                        |1-> (tab.(d),1)
                        | _-> let (dg,fg) =(d,d+N/2)  in
                              let (dd,fd) =(d,d+N/2) in
                              let (a,l)=aux t d (d+N/2) in
                              let (b,m)=aux t (d+N/2) f in
                              if a=b then (a, m+l)
          (*valable même si  TD et TG répondent "pas d'élu" 3diiA et2a*)
                              else if l*m=0 then
                              (max a b,  N/2 +  l+ m  - N/2)
                              (* TD ou bien TG répond "pas d'élu" voir 3di*)
                              else if  m>l then (b , N/2 + m - l) (*3diiB*)
                              else if  m<l then (a , N/2 - m + l) (*3diiB*)
                              else (-1,0) (*3diiC*);
              in aux tab 0  (vect_length tab);;
```

(f) Soit C_p le nombre d'opérations pour un tableau de longueur 2^p.
On a $C_0 = 2$
et pour tout p , $C_{p+1} = 2C_p + A$ où A est une constante
donc $C_p \sim 2^p$ (suite arithmético géométrique strictement croissante)
donc la complexité de `postulant` est bien linéaire

(g)
```
let elu3 tab=let (a,n) = postulant tab in
               if 2 * (nb tab a) > (vect_length tab) then a
               else -1;;
```