e3a 2015 - Informatique
un corrigé

Exercice 1

1.a Notons P_k le nombre de multiplications effectuées dans l'appel `puissance n k`. On a $P_1 = 0$ et $\forall k \geq 2,\ P_k = 1 + P_{k-1}$. On en déduit (suite arithmétique) qu e$P_k = k - 1$. Si on veut un compte moins spécifique, on obtiendra $P_k = O(k)$.

1.b

i. Les choses se déroulent comme suit.
On fait un appel avec $n = 2$ et $k = 7$.
On fait un appel avec $n = 2$ et $k = 3$.
On fait un appel avec $n = 2$ et $k = 1$.
On renvoie la valeur 2
On renvoie la valeur $2 * 2 * 2 = 2^3$ (car $k = 3$ est impair)
On renvoie la valeur $2^3 * 2^3 * 2 = 2^7$ (car $k = 7$ est impair)

ii. Les choses se déroulent comme suit.
On fait un appel avec $n = 2$ et $k = 8$.
On fait un appel avec $n = 2$ et $k = 4$.
On fait un appel avec $n = 2$ et $k = 2$.
On fait un appel avec $n = 2$ et $k = 1$.
On renvoie la valeur 2
On renvoie la valeur $2 * 2 = 2^2$ (car $k = 2$ est pair)
On renvoie la valeur $2^2 * 2^2 = 2^4$ (car $k = 4$ est pair)
On renvoie la valeur $2 * 4 * 2^4 = 2^8$ (car $k = 8$ est pair)

iii. Notons r_p le nombre d'appels récursifs à l'intérieur de la fonction `puissance2` quand on l'appelle avec un entier n et un autre k tel que $2^p \leq k < 2^{p+1}$. On a $r_0 = 0$ et pour $p \geq 1,\ r_p = 1 + r_{p-1}$ (si $2^p \leq k < 2^{p+1}$ alors $2^{p-1} \leq \lfloor k/2 \rfloor < 2^p$). Ainsi (suite arithmétique) $r_p = p$. Or, $2^p \leq k < 2^{p+1}$ entraîne $p \leq \log_2(k)$. Ainsin quand on appelle `puissance2` avec l'entier k comme second argument, il y a au plus $\log_2(k)$ appels récursifs internes.
Remarque : c'est cohérent pour $k = 1$ ou $k = 2$. C'est un peu mieux que ce que propose l'énoncé.

iv. On prouve par récurrence que l'entier $k \geq 1$ que pour tout entier naturel n, l'appel `puissance2 n k` renvoie l'entier n^k.
- Initialisation : c'est immédiatement vrai si $k = 1$.
- Hérédité : supposons le résultat vrai jusqu'à un certain rang $k - 1 \geq 1$. Soit $n \in \mathbb{N}$; on commence par introduire un entier x qui d'après l'hypothèse de récurrence (avec $\lfloor k/2 \rfloor = p$ qui est bien dans $[1, k-1]$) est égal à n^p.
Si k est pair alors $p = k/2$ et on renvoie $x^2 = n^{2p} = n^k$.
Sinon, $p = (k-1)/2$ et on renvoie $nx^2 = n^{2p+1} = n^k$.
Dans les deux cas, on renvoie le bon résultat.

v. Lors d'un appel récursif, on ne fait qu'un nombre constant d'opérations. La complexité est donc $O(\log_2(k))$.

2.a On calcule 2^k, 3^k en cotinuant tant que l'on reste strictement en deça de n. En sortie de boucle, il reste à voir si on est tombé exactement sur n.

```
let test_puissance n k =
   let m=ref 2 and mk=ref (puissance2 2 k) in
   while !mk<n do incr m; mk:=puissance2 !m k done ;
   !mk=n;;
```

2.b

i. Si $n = m^k$ alors $k = \frac{\ln(n)}{\ln(m)} \leq \frac{\ln(n)}{\ln(2)} = \log_2(n)$ (par croissance du logarithme).

ii. On teste si n est une puissance 2-ème, 3-ème etc. Evidemment, si n n'est PAS une puissance entière, on ne s'arrêtera jamais. Mais la question précédente indique qu'on peut s'arrêter de tester si n est une puissance k-ème quand $2^k > n$. On gère donc non seulement une référence `k` mais aussi une autre `pk` dont la valeur est celle de 2^k. En sortie de boucle, si `pk` est inférieur ou égal à n, c'est qu'on est sorti en trouvant un bon k.

```
let test_puissance_entiere n =
   let k=ref 2 and pk=ref 4 in
   while !pk<=n && not (test_puissance n !k) do
       incr k;
       pk:= !pk*2
       done ;
   !pk<=n ;;
```

iii. Dans `test_puissance`, on a au plus de l'ordre de n itérations (puisque `mk` est incrémenté à chaque étape) et une itération coûte de l'ordre de $O(\log_2(k))$ opérations (question 1.*b.v*). L'appel `test_puissance n k` a donc un coût $O(n \log_2(k))$.
Dans l'appel `test_puissance_entiere n`, on a immédiatement `pk`$= 2^{i+1}$ au début de la i-ème itération. On atteint donc la valeur n en un nombre d'itérations de l'ordre de $\log_2(n)$. L'itération numéro i a un côut $O(n \log(i+1))$ et i prend au pire une valeur de l'ordre de $\log_2(n)$. Finalement, le coût est

$$O(n \log_2(n) \log_2(\log_2(n)))$$

Remarque : à mon sens, l'énoncé ne veut rien dire puisque k n'est pas une donnée de l'algorithme.

iv. Pour la beauté du geste, on évite d'utiliser des références de listes. On écrit donc une fonction auxiliaire `construit : int → int → int list` telle que dans l'appel `construit a b`, on renvoie la liste des puissances entières entre `a` et `b`. Il suffit de l'appeler avec 2 et n. La fonction elle même teste a et l'ajoute éventuellement au résultat obtenu par appel récursif entre $a+1$ et b.

```
let liste1_puissances_entieres n =
   let rec construit a b =
      if a>b then []
      else begin
         if test_puissance_entiere a then a::(construit (a+1) b)
         else construit (a+1) b
         end
   in construit 2 n;;
```

2.c Dans le crible d'Erathosthène, on dispose d'un tableau d'entiers $2, 3, \ldots$. On entoure 2 et on raye tous ses multiples stricts. On considère alors la première case non rayée, 3, on entoure le contenu et on raye ses multiples stricts. On recommence avec la première case non rayée 5 etc. Les éléments entourés sont alors les nombres premiers (inférieurs à la borne que l'on se sera fixée en construisant le tableau initial).
Ici, on procède de même sauf qu'il ne s'agit pas de rayer les multiples mais de selectionner les puissances. On gère donc un tableau `tab` d'entiers que l'on fait évoluer. Une case vaut la valeur `true` si son numero est un entier détecté comme une puissance d'entiers. On selectionne d'abord les puissances de 2 (les cases associées de `tab` prennent la valeur `true`). On recommence avec la case suivant 2 et non encore selectionnée etc. Il reste à construire la liste des numéros de cases valant `true`.

`cocher : int → unit` est tel que l'appel `cocher m` selectionne les puissances de `m`.
`construit : int → int list` crée la liste des numéros $\leq k$ des cases cochées (on écrit une telle fonction pour éviter le recours à une référence de liste).

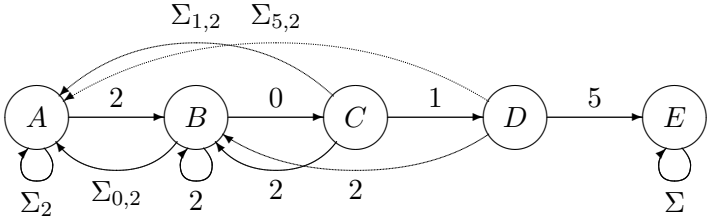
```
let liste2_puissances_entieres n =
   let tab=make_vect (n+1) false in
   let cocher m =
      let mk=ref (m*m) in
      while !mk<=n do
          tab.(!mk) <- true ;
          mk:= !mk*m
          done
   in
   let rec construit k =
       if k=n+1 then []
       else begin
           if tab.(k) then k::(construit (k+1))
           else construit (k+1)
           end

   in for i=2 to n do if not tab.(i) then cocher i done ;
   construit 2;;
```

Exercice 2

1. On opte pour un automate à quatre états :
   - on est dans l'état E quand on a déjà lu le motif 2015 ;
   - on est dans l'état D quand on n'a jamais lu 2015 mais que l'on vient de lire 201 ;
   - on est dans l'état C quand on n'a jamais lu 2015 mais que l'on vient de lire 20 ;
   - on est dans l'état B quand on n'a jamais lu 2015 mais que l'on vient de lire 2 ;
   - on est dans l'état A dans les autres cas.

   En notant $\Sigma_i = \Sigma \setminus \{i\}$ et $\Sigma_{i,j} = \Sigma \setminus \{i, j\}$, on obtient l'automate suivant.



2.a Il y a immédiatement 10000 valeurs possibles (G est injective et Σ^4 de cardinal 10^4).

2.b On a

$$G(bcde) = e + 10G(abcd) \texttt{ mod } 10000$$

2.c On a donc immédiatement

```
let decalG g e =
   (e + (10*g)) mod 10000;;
```

2.d On traite à part le cas où il y a moins de 3 éléments. Dans le cas contraire, on calcule l'image par G du premier motif de 4 chiffres puis l'image par G du motif suivant. On s'arrête quand on a lu tous les motifs ou quand on en a trouvé un convenable. La référence `g` sert pour stocker la valeur des images succesives par G des motifs.

```
let testG_motif t=
   let n=vect_length t in
   if n<=3 then false
   else begin
      let i=ref 3 in
      let g=ref (t.(3)+10*t.(2)+100*t.(1)+1000*t.(0)) in
      while !g<>2015  && !i < n-1 do
          incr i ;
          g:=decalG !g t.(!i)
          done;
      !g=2015
      end;;
```

2.e La boucle est effectuée au plus $n-3$ fois (si on doit lire tous les motifs de quatre chiffres successifs). Une itération se fait en temps constant. Hors de la boucle, on a aussi un nombre constant d'opérations. Finalement la complexité est $O(n)$ où n est la taille du tableau.

2.f C'est presque la même chose. On gère cette fois une référence `nb` indiquant le nombre de fois où 2015 a été rencontré. On va jusq'u bout du tableau et on peut utiliser une boucle `for`.

```
let nb_motif t=
   let n=vect_length t in
   if n<=3 then 0
   else begin
      let nb=ref 0 in
      let g=ref (t.(3)+10*t.(2)+100*t.(1)+1000*t.(0)) in
      if !g=2015 then incr nb;
      for i=4 to n-1 do
          g:=decalG !g t.(i);
          if !g=2015 then incr nb ;
          done;
      !nb
      end;;
```

3.a $H(abcd)$ est à valeurs dans $[|0,10|]$ (comme reste modulo 11) et toutes ces valeurs sont atteintes. H peut donc prendre 11 valeurs. Comme $10=-1[11]$, on peut remarquer que

$$H(abcd) = -a+b-c+d \texttt{ mod } 11$$

3.b On suppose ici que le tableau argument a exactement 4 cases. Avec la remarque précédente, on a

```
let calculH t = (t.(3)-t.(2)+t.(1)-t.(0)) mod 11;;
```

3.c Modulo 11, on a$H(bcde) = e-d+c-b = e-(d-c+b-a)-a$ et ainsi

$$H(bcde) = e - H(abcd) - a \texttt{ mod } 11$$

3.d Il est aisé, avec la formule précédente de calculer les valeurs successives des images par H motifs de taille 4 du mot. Comme $H(2015)=2$, il FAUT que cette image soit égale à 2 pour qu'il soit envisageable d'avoir rencontré le motif. Mais comme ce n'est pas suffisant, il convient alors de vérifier. Pour cette vérification, il suffit de voir si le motif commence par 201 (le chiffre suivant est forcément 5 si l'image par H est 2).

```
let testH_motif t =
    let n=vect_length t in
```

```
if n<=3 then false
else begin
   let i=ref 3 in
   let h=ref (t.(3)-t.(2)+t.(1)-t.(0) mod 11) in
   let test=ref false in
   while !i<n-1 && (not !test) do
      if !h=2 then
             begin
             test:=(t.(!i-1)=1 && t.(!i-2)=0 && t.(!i-3)=2)
             end;
      incr i;
      h:=t.(!i)- !h-t.(!i-4)
      done ;
   !test or (!h=2 && t.(n-2)=1 && t.(n-3)=0 && t.(n-4)=2)
   end;;
```

3.e Dans la boucle, la mise à jour de `test` coûte au plus trois accès à une case du tableau et, en tenant compte de la mise à jour de la référence `h`, chaque itération côute au plus 5 accès. Cependant, si la mise à jour de `test` induit trois accès, `test` devient égal à `true` et on s'arrête. Du fait de l'évaluation paresseuse, c'est donc en réalité au plus $4(n-1)$ accès que coûte la boucle. Il y a donc au pire $4n+3$ accès à une case de tableau. Peut-être faut-il être plus précis et regarder exactement ce que coûtent les mises à jour de `test`. Je n'en vois pas l'intérêt ! Peut-être y-a-t-il une meilleure façon d'utiliser la fonction `H`...

Exercice 3

1.a Simple parcours de tableau avec gestion d'une référence pour compter le nombre de `a` rencontrés.

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

1.b Il y a N itérations et chacune se fait en temps constant. Hors de la boucle, il y a un nombre constant d'opérations et la complexité est finalement $O(N)$.

1.c L'idée est de chercher successivement si $0, 1, 2, \ldots$ est élu en s'arrêtant quand on trouve un élu ou quand on arrive à la personne k. Pour cela, il faut connaître k. Je suppose ici que la fonction prend en argument le tableau ET k. Noter qu'en Caml $N/2$ est une division entière. Cependant, pour x entier, les conditions $x \leq N/2$ et $x \leq \lfloor N/2 \rfloor$ sont équivalentes.

```
let elu1 tab k =
    let N=vect_length tab in
    let a=ref 0 in
    while !a<k && ((nb tab !a)<=N/2) do incr a done ;
    if !a=k then (-1) else !a ;;
```

Remarque : on pourrait aussi créer la liste des éléments présents dans le tableau et tester chaque élément de la liste. Voici la fonction associée. `inserer` est une fonction d'insertion dans une liste triée. `creer_liste` renvoie la liste ordonnée des candidats. `parcours` teste les candidats de la liste.

```
let rec inserer k l =
```

```
   match l with
   [] -> [k]
  |x::q -> if x=k then l
           else if x<k then x::(inserer k q)
           else k::l;;

let elu1 tab =
    let n=vect_length tab in
    let rec creer_liste i =
       if i=n then []
       else inserer tab.(i) (creer_liste (i+1))
    in
    let rec parcours l =
        match l with
        [] -> -1
       |a::q -> if nb tab a <= n/2 then parcours q
                else a
    in
    parcours (creer_liste 0);;
```

1.d Il y a au plus k itération et chacune a un coût $O(N)$. Hors de la boucle, il y a un nombre constant d'opérations. Finalement, la complexité de la fonction est $O(kn)$.

2.a Scindons `tab` (de taille n) en deux sous-tableaux de tailles `ta` et `tb` (de tailles n_a et n_b avec $n_a + n_b = n$). Si x n'est élu ni pour `ta` ni pour `tb` alors son nombre d'apparitions dans `tab` est inférieur ou égal à $\frac{(n_a+n_b)}{2} = \frac{n}{2}$. x n'est donc pas élu pour `tab`. On obtient le résultat demandé en contraposant et en prenant le cas particuliers des demi-tableaux à droite et gauche.

2.b Un appel récursif sur chaque demi-tableau nous donne les seuls éligibles. Il suffit donc de vétrifier si ces éligibles atteignent effectivement la majorité. Je choisis comme cas de base celui où il n'y a qu'une seule case (qui est évident : le seul candidat est élu et il a une voix).

```
let rec elu2 tab =
   let n=vect_length tab in
   if n=1 then (tab.(0),1)
   else begin
       let tabg=(miGauche tab) and tabd=(miDroite tab) in
       let (xg,ng)=elu2 tabg and (xd,nd)=elu2 tabd in
       let mg=nb tabd xg and md=nb tabg xd in
       if mg+ng>n/2 then (xg,mg+ng)
       else if md+nd>n/2 then (xd,md+nd)
       else (-1,0)
       end;;
```

Remarque : on pourrait éviter de faire l'appel à droite si l'appel à gauche donne un élu.

2.c Notons C_p le nombre maximal d'opérations effectué par la fonction quand on l'appelle avec un tableau de longueur $\leq 2^p$. On a alors

$$C_0 = O(1) \quad \text{et} \quad \forall p \geq 1,\ C_p = 2C_{p-1} + O(2^p)$$

le $O(2^p)$ couvrant les appels aux fonctions `miGauche`, `miDroite` et `nb`. On en déduit que

$$\frac{C_p}{2^p} = \frac{C_{p-1}}{2^{p-1}} + O(1)$$

puis que $\frac{C_p}{2^p} = O(p)$ ou encore $C_p = O(p2^n)$. En revenant à la variable N, on en déduit que la complexité de la fonction est

$$O(N \log_2(N))$$

c'est à dire quasi-linéaire en fonction de la taille du tableau.

3.a Supposons que le tableau T (de taille N) donne l'élu a et notons n le nombre des apparitions de a dans T. Par définition, on a $n > N/2$, a apparaît au plus (et même exactement) n fois dans T et tout autre élément apparaît au plus $N - n$ fois (puisque $N - n$ est le nombre de positions où a n'est pas). a est donc postulant de T (pour cette valeur n).

3.b Supposons que a soit un postulant du tableau T (de taille N) associé à la valeur n. Si $b \neq a$ alors b apparaît au plus $N - n$ fois et comme $n > N/2$, b apparait au plus $N/2$ fois et n'est donc pas élu. a est donc le seul élément qui a une chance d'être élu.

3.c Dans le tableau $[|1; 1; 2; 3|]$, on a $N = 4$ et donc $N/2 = 2$. 1 apparaît au plus 3 fois et tout autre élément apparaît au plus 1 fois. 1 est donc postulant pour la valeur 3 mais n'est cependant pas élu.

Dans le tableau $[|1; 1; 2; 2|]$, on a encore $N/4$. Un postulant a doit être associé à $n = 3$ ou $n = 4$. Mais dans ce cas, l'autre élément doit apparaître au plus 0 ou 1 fois ce qui n'est pas le cas. Il n'y a donc pas de postulant.

3.d.i Posons $n = l + \lfloor (N+2)/4 \rfloor$ et montrons que a est postulant pour T associé à la valeur n. Pour cela, on note qu'un élément apparaît dans TD au plus $\lfloor N/4 \rfloor$ fois (sinon il serait élu) et que $l > N/4$ (définition d'un élu).

* a apparaît donc au plus $l + \lfloor N/4 \rfloor$ fois T (on ajoute les apparitions dans TG et TD) et donc a fortiori au plus n fois
* un élément $b \neq a$ apparaît au plus $(N/2 - l) + \lfloor N/4 \rfloor$ fois dans T et il suffit de montrer que ceci est inférieur à $N - n$ c'est à dire que $N/2 \geq \lfloor N/4 \rfloor + \lfloor (N+2)/4 \rfloor$ ce que l'on justifie en traitant les cas $N = 4k$ ou $N = 4k + 2$
* $n = l + \lfloor (N+2)/4 \rfloor$. Si $N = 4k$ alors $l > k$ et $n > 2k = N/2$. Si $N = 4k + 2$ alors $l \geq k + 1$ et $n \geq 2k + 2 > N/2$.

3.d.ii On distingue les cas.

A. Si $a = b$, on pose $n = l + m$. Comme $l, m > N/4$, $n > N/2$. De plus, a apparaît au plus $n = l + m$ fois. Enfin, si $x \neq a$ alors le nombre d'apparition de x dans T est majoré par $(N/2 - l) + (N/2 - m) = N - n$. a est donc postulant de T pour la valeur $l + m$.

B. On suppose $a \neq b$ et $m > l$. Posons $n = \frac{N}{2} + m - l$ et montrons que b est postulant de T pour cette valeur.
   * On a bien $n > m/2$ puisque $m > l$.
   * b apparaît au plus $(N/2 - l) + m$ fois dans T (au plus $N/2 - l$ fois dans TG et m fois dans TD).
   * Si $x \notin \{a, b\}$, x apparaît au plus $(N/2 - l) + (N/2 - m) = N - (l + m)$ fois dans T et cette quantité est plus petite que $N - n$ car $N/4 \leq l$ (ce qui donne facilement $N - (l + m) \leq N - n$).
   * a apparaît au plus $l + (N/2 - m) = N - n$ fois dans T.

C. On suppose $a \neq b$ et $m = l$. S'il existait un élu pour T, il serait élu pour TG ou TD (question 2) et donc serait égal à a ou b (question 3.b). Par symétrie des rôles, on peut supposer qu'il s'agirait de a. Mais a apparaît au plus $l + (N/2 - m) = N/2$ fois dans T. On a donc une contradiction.

3.e Il s'agit d'utiliser les questions précédentes. On peut en effet déduire l'existence et la valeur d'un postulant pour le tableau à partir d'une connaissance similaire sur les tableaux gauche et droit. En effet :

- si il n'y a aucun postulant à droite et gauche alors il n'y a pas d'élu ni à droite ni à gauche et donc pas d'élu pour le tableau et donc pas de postulant
- s'il y a un postulant d'un seul côté, $d.i$ donne un postulant du tableau entier

- s'il y a un postulant des deux côtés, $d.ii$ donne un postulanty global ou montre qu'il n'en existe pas (cas C où il n'y a pas d'élu)

```
let rec postulant tab =
   let n=vect_length tab in
   if n=1 then (tab.(0),1)
   else begin
      let tabg=(miGauche tab) and tabd=(miDroite tab) in
      let (xg,ng)=postulant tabg and (xd,nd)=postulant tabd in
      if xg=(-1) && xd=(-1) then (-1,0)
      else if xd=(-1) then (xg,ng+(n+2)/4)
      else if xg=(-1) then (xd,nd+(n+2)/4)
      else if xg=xd then (xg,ng+nd)
      else if ng>nd then (xg,n/2+ng-nd)
      else if nd>ng then (xd,n/2+nd-ng)
      else (-1,0)
      end;;
```

3.f Il suffit alors de chercher un postulant. S'il n'y en a pas, il n'y a pas d'élu. Sinon, il suffit de tester ce postulant.

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

Le calcul de complexité de `postulant` se fait comme celui de `elu2` sauf qu'il n'y a pas d'appel à `nb`. **MAIS** il reste les appels à `miGauche` et `miGauche` qui font qu'on ne gagne rien (sauf à supposer qu'elles agissent en temps constant, ce qui n'est pas raisonnable en Caml).
Je pense que le sujet oublie de compter la complexité de ces appels et veut que l'on conclue en disant que `postulant` est de complexité linéaire (même calcul qu'en 2.c avec $O(2^p)$ remplacé par $O(1)$) et qu'il en est de même pour `elu3` (complexité de `postulant` plus celle de `nb` qu'on n'appelle plus qu'une fois).
Si on veut **vraiment** une complexité $O(N)$, il faut écrire une version de `postulant` qui n'utilise pas de recopie de sous-tableau. L'idée est de faire porter la récursivité sur les indices du tableau via une fonction auxiliaire locale `etude` dont les arguments sont les indices a et b délimitant le bout de tableau étudié. Voici la version mettant cette tactique en oeuvre.

```
let postulant tab =
  let rec etude a b =
   if b=a then (tab.(a),1)
   else begin
      let (xg,ng)=etude a ((a+b-1)/2) and (xd,nd)=etude ((a+b+1)/2) b in
      if xg=(-1) && xd=(-1) then (-1,0)
      else if xd=(-1) then (xg,ng+(b-a+3)/4)
      else if xg=(-1) then (xd,nd+(b-a+3)/4)
      else if xg=xd then (xg,ng+nd)
      else if ng>nd then (xg,(b-a+1)/2+ng-nd)
      else if nd>ng then (xd,(b-a+1)/2+nd-ng)
      else (-1,0)
      end
  in etude 0 (vect_length tab-1) ;;

let elu3 tab =
```

```
let (x,n)=postulant tab in
if nb tab x > (vect_length tab)/2 then x
else -1;;
```