X 2006 - MP
un corrigé de l'épreuve d'informatique

1 Codage de Reed-Solomon, première version.

Q.1. On utilise l'algorithme de Hörner qui repose sur la formule suivante :

$$U(x) = \sum_{i=0}^{d} u_i x^i = \left(\sum_{i=1}^{d} u_i x^{i-1} \right) x + a_0 = V(x)x + a_0$$

Le polynôme V est simplement la queue de liste constituant le polynôme U. Ceci décrit l'étape récursive et la cas de base est celui où U est la liste vide et correspond au polynôme nul ; on renvoie alors 0.

```
let rec valeur U x =
  match U with
   []  -> 0
| a::Q -> ((((valeur Q x) * (x mod p)) mod p) + a) mod p ;;
```

Q.2. Fonction récurrente classique où l'on parcourt la liste argument α.

```
let rec codage alpha A =
  match alpha with
     []   -> []
| x::beta -> (valeur A x)::(codage beta A) ;;
```

Q.3. Notons V_k le nombre d'opération dans l'appel `(valeur U x)` quand `U` est de longueur k. On a $V_0 = O(1)$ et $\forall k \geq 1,\ V_k = O(1) + V_{k-1}$. On en déduit immédiatement que

$$V_k = O(k)$$

Notons $C_{n,k}$ le nombre d'opérations dans l'appel `(codage alpha A)` avec `alpha` de longueur n et `A` de longueur k. On a $C_{0,k} = O(1)$ et pour $n \geq 1$, $C_{n,k} = V_k + C_{n-1,k} + O(1) = C_{n-1,k} + O(k)$. On en déduit que

$$C_{n,k} = O(nk)$$

2 Polynômes.

Q.4. On fait la somme terme à terme des coefficients en s'arrêtant quand l'une des liste est épuisée.

```
let rec addition U V =
  match (U,V) with
     [],_    -> V
|    _,[]    -> U
| a::U',b::V'-> ((a+b) mod p)::(addition U' V') ;;
```

La quantité $\min(\ell_U, \ell_V)$ diminue de 1 dans les appels récursifs. C'est la quantité qui caractérise l'algorithme. Le cas de base est celui où cette quantité est nulle (un des deux polynômes est nul). Outre l'appel récursif, on a un nombre constant d'opérations. La complexité est donc

$$O(\min(\ell_U, \ell_V))$$

Q.5. On pourrait écrire la même fonction en remplaçant l'addition par une soustraction. Le seul souci peut, éventuellement, être l'apparition de coefficients négatif (la fonction `mod` renvoyant un entier entre $-p+1$ et $p-1$, elle renvoie un résultat négatif quand on l'appelle avec un entier négatif). C'est pourquoi je cherche le reste modulo p de $a - b + p$ (ou de $p - x$)et non de $a - b$. On ne traite bien sûr pas de la même façon les cas où U et V sont nuls.

```
let rec soustraction U V =
   match (U,V) with
|    _,[]    -> U
|  [],x::V' -> ((p-x) mod p)::(soustraction [] V')
| a::U',b::V'-> ((a-b+p) mod p)::(soustraction U' V') ;;
```

La complexité est cette fois $O(\ell_V)$ puisque cette fois on épuise la liste V (rôles non symétriques).

Q.6. C'est le même type de fonction (en plus simple). Nous n'avons pas mis de côté le cas $s = 0$.

```
let rec produitParScalaire U s =
   match U with
    []  -> []
| x::U' -> ((x*s) mod p)::(produitParScalaire U' s) ;;
```

C'est un simple parcourt de la liste U est la complexité est $O(\ell_U)$.

Q.7. Si U ou V est une liste vide (correspondant au polynôme nul) alors le produit UV est nul et on doit renvoyer la liste vide (polynôme nul). Sinon, U et V s'écrivent $U = a + XU_1$ et $V = b + XV_1$ (U_1 et V_1 étant représentés par les queues des listes U et V) et

$$UV = ab + X\left(aV1 + bU_2 + XU_1V_1\right)$$

C'est cette formule que l'on exploite (multiplier par X revient à ajouter un zéro à gauche). Pour éviter l'apparition d'un zéro inutile à droite du résultat, on traite aussi comme cas de base le cas où l'un des polynômes est constant.

```
let rec produit U V =
  match (U,V) with
     [],_    -> []
|    _,[]    -> []
|   [a],_    -> produitParScalaire V a
|   _,[b]    -> produitParScalaire U b
|a::U1,b::V1 -> (a*b mod p)::(addition
                                (addition
                                    (produitParScalaire V1 a)
                                    (produitParScalaire U1 b) )
                                (0::(produit U1 V1)) ) ;;
```

Je choisit de prendre $\ell_u + \ell_v$ comme quantité caractéristique de l'algorithme (elle diminue bien à chaque étape ce qui assure, compte tenu des cas de base, la terminaison de l'algorithme). Soit P_n le nombre d'opérations dans l'appel `produit U V` quand la quantité caractéristique est plus petite que n. On a alors, P_n qui est soit constant soit égal à $P_{n-2} + O(n)$ (les appels à `produitParScalaire` coûtent $O(\ell_{U_1}) + O(\ell_{V_1})$ et donc $O(n)$; l'addition $aU_1 + bV_1$ coûte $O(\min(\ell_{U_1}, \ell_{V_1}))$ et donc aussi $O(n)$; il en est de même pour l'autre addition car les polynômes restent de degré plus petit que n). On a alors facilement $P_n = O(n^2)$ et la complexité est finalement

$$O((\ell_u + \ell_v)^2)$$

Remarque : on peut bien sur aussi écrire que $(a + XU_1)V = aV + XU_1V$ et écrire à partir de cette formule une fonction de même complexité quadratique. J'ai préféré écrire la première pour donner des rôles symétriques à U et V.

Q.8. - **Première méthode.**

Si U est de degré strictement inférieur à V, le quotient cherché est nul. Sinon, on note a le coefficient de X^{ℓ_U-1} dans U et b le coefficient dominant de V ; on note c l'inverse de b. On a alors U et $(acX^{\ell_U-1-d_V})V$ qui ont même degré et même coefficient devant X^{ℓ_U-1}. Le quotient cherché est alors $acX^{\ell_U-1-d_V}$ plus le quotient dans la division euclidienne de $(U - (acX^{\ell_U-1-d_V})V)$ par V. C'est ce processus récurrent que l'on va utiliser.

Pour gérer la récurrence, on a besoin du "dernier" coefficient de V et mieux vaut travailler

avec l'image miroir de U (pour récupérer les coefficients "dans l'ordre"). On note UU l'image miroir de U et VV celle de V. Pour effectuer la soustraction $(U - (acX^{\ell_U - 1 - d_V})V)$, on vérifie aisément que l'on peut utiliser les fonctions déjà écrites avec cette fois UU et VV (il suffit de les réinterpréter dans le contexte de listes ordonnées par degré décroissant). Le résultat obtenu commencera alors par un zéro et le nouveau polynôme à diviser s'obtiendra en supprimant ce zéro c'est à dire en prenant la queue de la liste résultat.
On utilise une fonction auxiliaire
`divise : poly -> int -> poly`
dont les argument sont
- l'image miroir du polynôme U à diviser ;
- un entier correspondant à la différence entre la longeur de U et celle du polynôme par lequel on divise ;
- un polynôme `accu` dans lequel on stocke les coefficients obtenus au fur et à mesure et qui évolue donc comme le quotient quand on pose la division (il est rempli dans le bon ordre car on commence par lui ajouter le terme de plus haut degré).

A tout moment, le polynôme représentant `UU` plus le produit des polynômes représentés par `V` et `accu` vaut le polynôme initial `U`.
Nous nous sommes permis l'utilisation de `rev` qui donne l'image miroir d'une liste et de `list_length` donnant la taille d'une liste.

```
let division U V =
let VV=rev V in
let c=inv (hd VV) in
let rec divise UU long accu=
 if long <0 then accu
 else begin
      let  a=hd UU in
      divise (tl (soustraction UU
                     (produitParScalaire VV ((a*c) mod p)) ) )
             (long - 1)
             (a::accu)
      end
in divise (rev U) ((list_length U)-(list_length V)) [] ;;
```

Les fonctions `rev` et `list_length` sont de complexité linéaire en fonction de la taille des arguments. Dans la fonction `divise`, c'est le second argument qui régit la récurrence. En notant D_l le nombre d'opération quand ce second argument vaut l, on a $D_l = D_{l-1} + O(\ell_V)$ (les produit par scalaire et soustractions se font sur ce nombre d'éléments) et ainsi $D_l = O(l\ell_v)$. Initialement, $l \leq \ell_U$ et l'appel à `divise` coûte de l'ordre de $\ell_U \ell_V$ opérations. Les appels aux autres fonctions sont alors de coût négligeable. On obtient une complexité

$$O(\ell_U \ell_V)$$

- **Seconde méthode.**
La méthode précédente a l'avantage de suivre l'algorithme de division euclidienne tel qu'on l'apprend à l'école. L'inconvénient est bien sûr l'ordre dans lequel on a besoin des coefficients ce qui nous a amené à faire des contorsions avec les images miroir. Pour éviter ces contorsions, on peut faire les remarques suivantes.
  - Si U est nul alors quotient et reste cherchés sont nuls.
  - Sinon, $U = a + xU_1$; soit $U_1 = Q_1 V + R_1$ la division euclidienne de U_1 par V. On a alors

$$U = (a + XR_1) + (XQ_1)V$$

    et il convient, pour conclure, de faire la division euclienne de $a + XR_1$ par V. On est donc amenés à écrire une fonction spéciale de division euclidienne dans le cas où le diviseur est

de degré inférieur au dividende. Cette fonction écrite, on saura écrire $a + XR_1 = cV + R_2$ (c étant un scalaire) et on aura donc

$$U = (c + XQ_1)V + R_2$$

$c + XQ_1$ s'obtenant par simple concaténation de c sur la liste Q_1.

On est donc amenés à écrire une fonction auxiliaire `div_elem` prenant en argument des polynômes U et V tels que $\deg(U) \leq \deg(V)$ et renvoyant le couple $(U/V, U \bmod V)$. On travaille encore récursivement.

- Si U est nul alors le quotient et le reste sont nuls.
- Si $V = b$ est constant ainsi que $U = a$, le quotient est a/b et le reste nul.
- Sinon, $U = a + xU_1$ et $V = b + xV_1$; on a $\deg(U_1) \leq \deg(V_1)$ et on peut faire un appel récursif pour obtenir c et R_1 tels que $U_1 = cV_1 + R_1$. On en déduit que

$$U = (a - cb) + XR_1 + cV$$

$(a - cb) + XR_1$ (obtenu avec un consage) est le reste et c est le quotient

Les fonctions décrites sont les suivantes.

```
let rec div_elem U V =
 match (U,V) with
|[],_ -> 0,[]
|[a],[b] -> (a*(inv b) mod p),[]
|a::U1,b::V1 -> let (c,R1)=div_elem U1 V1 in
                c,((a+(p-b)*c) mod p)::R' ;;

let rec divise2 U V =
 match U with
  [] -> [],[]
|a::U1 -> let (Q1,R1) = divise U1 V in
          let (c,R2) = div_elem (a::R1) V in
          c::Q1,R2 ;;
```

Si on ne veut que le quotient, on selectionne uniquement le premier élément du couple résultat. La complexité est la même que précédemment.

Q.9. La seconde méthode donne déjà le reste. Passons à la première.

On pourait écrire que $R = U - QV$ mais cela me semble maladroit. Dans la fonction auxiliaire `divise` de la question précédente, les "restes successifs" correspondent aux valeurs successives de `UU` et c'est donc la valeur finale de `UU` qui donne le reste escompté (ou plutôt son image miroir pour remettre les coefficients dans l'ordre). On n'a plus besoin de l'accumulateur.

```
let modulo U V =
let VV=rev V in
let c=inv (hd VV) in
let rec divise UU long =
 if long <0 then rev UU
 else begin
      let  a=hd UU in
      divise (tl (soustraction UU
                    (produitParScalaire VV ((a*c) mod p)) ) )
             (long - 1)
      end
in divise (rev U) ((list_length U)-(list_length V)) ;;
```

La complexité est alors bien sûr la même que pour `division`.

3 Multiplication et division rapide.

Dans toute cette partie, la taille des listes est importante (il y a des contraintes sur les tailles des listes arguments) et on a parfois besoin d'un accès au dernier élément d'une liste. Pour ces raisons, il serait parfois plus simple de travailler avec des tableaux. Nous avons cependant respecté ce qui semble être l'esprit de l'énoncé et travaillé uniquement avec des listes.

Q.10. En remarquant que $(U_h + U_b)(V_h + V_b)$, on a

$$U(X)V(X) = W_h X^{2e} + W_m X^e + W_b \text{ où } \begin{cases} W_h = U_h V_h \\ W_b = U_b V_b \\ W_m = (U_h + U_b)(V_h + V_b) - W_h - W_b \end{cases}$$

Q.11. On va utiliser la stratégie récursive décrite. Pour cela, je commence par écrire deux fonctions auxiliaires.

- `decoupe : poly -> int -> poly*poly` prend en argument une liste `U` et un entier `n` tels que `U` est de longueur plus petite que `n`. Cette fonction renvoie le couple $[u_0; \ldots; u_{n-1}], [u_n; \ldots; u_{\ell-1}]$ (ℓ est la longueur de U).
- `decale : poly -> int -> poly` telle que l'appel `(decale P e)` renvoie la liste correspondant à $X^e P$ (en ajoutant e zéros à la liste P).

```
let rec decoupe U n=
   if n=0 then ([],U)
   else let (Ub,Uh)=decoupe (tl U) (n-1) in (((hd U)::Ub),Uh) ;;

let rec decale P e =
 if e=0 then P
 else 0::(decale P (e-1)) ;;
```

Passons à la fonction principale. Pour éviter de multiples appels à `list_length`, j'écris une fonction auxiliaire `prod : poly -> poly -> int -> poly*poly` dont les arguments sont des polynômes U et V et un entier ℓ tel que U et V sont de longueur ℓ (c'est à dire de degré $\ell - 1$). L'étape récurrente à déjà été décrite (il faut juste penser à donner la bonne valeur au troisième argument en calculant la taille des listes en jeu). Le cas de base est celui d'un polynôme ayant un seul élément (j'ai ajouté le cas d'un polynôme nul, $\ell = 0$, mais il est inutile) où on obtient un polynôme à un seul élément comme résultat.

```
let produitRapide U V =
  let rec prod U V l =
   if l=0 then []
   else if l=1 then [((hd U)*(hd V)) mod p]
   else begin
      let (Ub,Uh)=decoupe U (l/2) and (Vb,Vh)=decoupe V (l/2) in
      let Wb=prod Ub Vb (l/2)
      and Wh=prod Uh Vh (l-(l/2)) in
      let Wm=soustraction (prod (addition Ub Uh) (addition Vb Vh) (l-(l/2)))
                          (addition Wh Wb)
      in addition (addition Wb (decale Wm (l/2))) (decale Wh (2*(l/2)))
      end
in prod U V (list_length U) ;;
```

Les fonctions de découpage et de décalage sont de complexité linéaire en fonction de la taille de l'entier argument.

Soit PR_n le nombre d'opérations effectuées dans l'appel `prod U V l` quand $l = 2^n$. On a alors (trois appels récursifs, des additions et soustraction de complexités $O(2^n)$, de découpages et

décalages de complexités $O(2^n)$, d'autres opérations en temps constant)

$$PR_n = 3PR_{n-1} + O(2^n)$$

En notant $v_n = \frac{PR_n}{3^n}$, il existe une constante c telle que pour tout n,

$$v_n \leq v_{n-1} + c\left(\frac{2}{3}\right)^n$$

On a alors, par récurrence

$$v_n \leq v_0 + c\sum_{k=1}^{n}\left(\frac{2}{3}\right)^k = O(1)$$

On a donc $PR_n = O(3^n)$. Comme $n = \log_2(\ell)$, on a donc un nombre d'opération de l'ordre de

$$3^{\log_2(\ell)} = e^{\frac{\ln(3)\ln(\ell)}{\ln(2)}} = \ell^{\log_2(3)}$$

Q.12. On a ici de grosses contraintes de tailles :

- quand on utilise (appel récursifs) la division de U par V on doit être sûr que la taille de U est deux fois celle de V moins un ;
- quand on utilise le produit rapide, il faut le faire avec deux listes de même taille.

Remarque : on pourrait remarquer que le produit rapide fonctionne tout aussi bien avec deux opérandes de longueurs différentes. Il suffit de découper les deux polynômes au même endroit, la moitié de l'un d'entre eux. Avec cette modification, on n'a plus à se soucier des tailles. Je n'ai pas utilisé cette "facilité" ici.

Avec les notations de l'énoncé, on aura les tailles suivantes :

U	V	U_b	V_b	U_h	V_h
$2\ell-1$	ℓ	$2e$	e	$2(\ell-e)-1$	$\ell-e$

Il faut que les listes intermédiaires aient les tailles suivantes (attention, la taille n'est pas le degré)

Q	R	Q_b	R_b	Q_h	R_h
ℓ	$\ell-1$	$e-1$	$e-1$	$\ell-e$	$\ell-e-1$

Le cas de base est celui où $\ell = 1$. V et U contiennent un élément. On doit renvoyer la liste contenant l'unique élément de U fois l'inverse de celui de V.

L'énoncé décrit l'étape récursive ; occupons nous des problèmes de taille des listes.

- Si la fonction est correcte, on pourra récupérer Q_h ayant la bonne taille.
- Le produit Q_hV_h peut être calculé par multiplication rapide (mêmes tailles) et on peut alors calculer $R_h = U_h - Q_hV_h$ mais on obtiendra une liste de $2(\ell-e)-1$ éléments. Cette liste est trop longue. Elle contient certainement des zéros à droite qu'il faut enlever pour arriver à une taille $e-1$. Le nombre de zéros à supprimer vaut $\ell - e$. C'est la fonction `ecrete : poly -> int -> poly` qui fera le travail (elle prend en argument une liste P et un entier n et renvoie la liste privée de ses n éléments les plus à droite).
- Pour pouvoir effectuer le produit rapide Q_hV_b, il convient que Q_h et V_b aient même taille. Il faut donc ajouter $\ell - 2e$ (qui vaut 0 ou 1) zéro à droite de la liste V_b. Le produit a alors une taille $2(\ell-e)-1$. L'ajout se fait grâce à la fonction `ajoute : poly -> int -> poly` (elle prend en argument une liste P et un entier n et renvoie la liste à laquelle on ajoute à droite n zéros).
- Le calcul de $X^eR_h + u_{2e-1}X^{e-1} = X^{e-1}(u_{2e-1} + XR_h)$ se fait en consant l'élément u_{2e-1} (dernier élément de U_b et donc tête de l'image miroir) à la liste R_h puis en ajoutant $e-1$ zéros (fonction dé décalage). On obtient alors une liste de $\ell - 1$ éléments.

- Comme $\ell \geq 2e$, $2(\ell - e) - 1 \geq \ell - 1$ et le calcul de $S = X^e R_h + u_{2e-1} X^{e-1} - Q_h V_b$ donnera une liste de $2(\ell - e) - 1$ éléments. Comme V_h en a $\ell - e$, on peut faire la division euclidienne et obtenir Q_b de taille $\ell - e$.
- $X^e Q_h$ s'obtient à partir de Q_h avec `decale` et donne une liste de ℓ éléments. Si on ajoute Q_b, la taille reste ℓ, ce que l'on voulait.

Voici d'abord les deux fonctions auxiliaires.

```
let ajoute Vb n =
  if n<>0 then rev (decale (rev Vb) n) else Vb;;

let rec ecrete P n =
if n=0 then []
else (hd P)::(ecrete (tl P) (n-1)) ;;
```

Pour la fonction principale, comme dans le cas de la multiplication rapide, j'ai choisi d'écrire une fonction auxiliaire prenant aussi en argument la taille ℓ. Par ailleurs, les polynômes diviseur a toujours le même coefficient dominant dont on stocke une fois pour toutes l'inverse dans la variable c.

```
let divisionRapide U V =
  let c=inv (hd (rev V)) in
   let rec divi U V l =
    if l=1 then [((hd U)*c) mod p]
    else begin
       let (Vb,Vh)=decoupe V (l/2) in
       let (Ub,Uh)=decoupe U (2*(l/2)) in
       let Qh=divi Uh Vh (l-(l/2)) in
       let Rh=ecrete (soustraction Uh (produitRapide Qh Vh)) (l-(l/2)) in
       let Qb=divi (soustraction
                          (decale ((hd (rev Ub))::Rh) ((l/2)-1))
                          (produitRapide Qh (ajoute Vb (l-(2*(l/2))))))
                   Vh
                   (l-(l/2))
       in addition (decale Qh (l/2)) Qb
       end
  in divi U V (list_length V) ;;
```

Soit D_n le nombre d'opération effectuées dans l'appel `(divi U V l)` quand l'entier vaut 2^n. On a

$$D_n = 2D_{n-1} + O(3^n)$$

et donc (étude de $D_n/2^n$ comme en question 11) $D_n = O(3^n)$. L'appel à `list_length` et celui à `rev` (calcul de `c`) coûtent de l'ordre de 2^n opérations et ce coût est négligeable. La complexité est donc encore

$$O(\ell^{log_2(3)})$$

Q.13. On pourrait modifier la fonction précédente pour qu'elle renvoie Q et R. On peut aussi l'utiliser pour obtenir Q et en déduire R. Ci-dessous, on utilise encore `ecrete` pour supprimer des zéros inutiles.

```
let moduloRapide U V =
 let l=list_length V in
 let Q=divisionRapide U V in
 ecrete (soustraction U (produitRapide Q V)) l ;;
```

On a immédiatement une complexité $O(\ell^{log_2(3)})$ pour cette fonction.

4 Codage par évaluation multi-points.

Q.14. On procède de manière récurrente.

- Si $n = 1$ alors l'arbre a retourner comporte un unique noeud.
- Sinon, on pose $e = \lfloor n/2 \rfloor$, $\alpha_b = [\alpha_0; \ldots; \alpha_{e-1}]$ et $\alpha_h = [\alpha_e, \ldots, \alpha_{n-1}]$. On construit les sous arbres f_h et f_b associés à α_h et α_b et on en déduit celui associé à α : c'est un noeud étiqueté par le produit des racines de f_h et f_b et dont f_h et f_b sont respectivement les fils droit et gauche.

On dispose déjà (question 11) de la fonction de découpage. Pour avoir des coefficients dans $[0, p-1]$, on note que $-\alpha_i = p - \alpha_i[p]$. Enfin, comme pour la multiplication rapide, on écrit une fonction auxiliaire `construit` qui prend aussi n en argument.

```
let arbreSousProduits alpha =
  let rec construit alpha n =
  if n=1 then Noeud ([p-(hd alpha) mod p ;1],Vide,Vide)
  else begin
     let (alphab,alphah)=decoupe alpha (n/2) in
     let (Noeud (Ph,gh,dh))=construit alphah (n-(n/2)) in
     let (Noeud (Pb,gb,db))=construit alphab (n/2) in
     Noeud(produit Pb Ph,Noeud (Pb,gb,db),Noeud (Ph,gh,dh))
     end
in construit alpha (list_length alpha) ;;
```

Soit A_h le nombre d'opération effectuées dans l'appel `(construit alpha n)` quand l'entier vaut 2^h. On a (le produit coûte de l'ordre de $(2^h)^2$ opérations)

$$A_h = 2A_{h-1} + O(4^h)$$

On en déduit que $A_h = O(4^h)$ (en passant par $A_h/2^h$) et on a donc une comlexité $O(n^2)$.

Si l'on veut utiliser la multiplication rapide, il faut toujours faire le produit avec deux polynômes de même degré. Il y a donc problème quand n est impair : α_h possède alors un élément de plus que α_b et il faut donc ajouter un zéro à P_b (avec la fonction `ajoute` de la question 12).

```
let arbreSousProduits alpha =
  let rec construit alpha n =
  if n=1 then Noeud ([p-(hd alpha);1],Vide,Vide)
  else begin
     let (alphah,alphab)=decoupe alpha (n/2) in
     let (Noeud (Ph,gh,dh))=construit alphah (n-(n/2)) in
     let (Noeud (Pb,gb,db))=construit alphab (n/2) in
     Noeud(produitRapide (ajoute Pb (n-2*(n/2))) Ph,
                                 Noeud (Pb,gb,db),
                                 Noeud (Ph,gh,dh))
in construit alpha (list_length alpha) ;;
```

Pour le calcul de complexité, la récurrence devient $A_h = 2A_{h-1} + O(3^h)$ et on obtient une complexité $O(n^{log_2(3)})$.

Q.15. Si T a une étiquette R et des fils gauche et droit T_1 et T_2 alors d'après l'observation de l'énoncé, l'arbre des restes associé au couple (A, T) a pour fils gauche celui associé au couple $(A \ mod \ R, T_1)$ et pour fils droit celui associé au couple $(A \ mod \ R, T_2)$. Le cas de base étant celui évident d'un arbre vide, on obtient la fonction suivante.

```
let rec arbreRestes T A  =
  match T with
  Vide -> Vide
| Noeud(R,T1,T2) -> let B=modulo A R in
                    Noeud(B,arbreRestes T1 B,arbreRestes T2 B) ;;
```

Si on veut utiliser la fonction `moduloRapide` il faut que les tailles soient correctes (initialement, que si ℓ est la taille de l'étiquette de la racine de T alors A soit de taille $2\ell-1$; ceci ne correspond d'ailleur pas aux hypothèses initiales car A est de degré $k < n$ où n est le nombre de points).
La complexité de la fonction précédente va faire intervenir la taille k de la liste A et la hauteur h de l'arbre T (ou du nombre n de point sachant que n est de l'ordre de 2^h). Distinguons deux cas.

- Si on utilise la fonction rapide, il faut transformer A de façon à lui donner la bonne taille (de l'ordre de 2^{h+1} puisque la racine de T a un taille de l'ordre de 2^h). Le calcul du reste coûte de l'ordre de 3^h opérations). On obtient pour le nombre d'opérations une récurrence du type $C_h \leq 2C_{h-1} + O(3^h)$ et la complexité globale est $O(3^h)$ ou encore $O(n^{log_2(3)})$.
- Si on utilise la fonction "simple", les polynômes dividendes ont touours un degré plus petit que k et donc plus petit que 2^h. Le calcul du reste côute de l'ordre de 4^h opérations. On obtient pour le nombre d'opérations une récurrence du type $C_h \leq 2C_{h-1} + O(4^h)$ et la complexité globale est $O(4^h)$ ou encore $O(n^2)$.

Q.16. On crée l'arbre des restes et il suffit de donner la liste des étiquettes des feuilles de cet arbre (dans le bon ordre). Je commence par écrire une fonction `parcourt : arbre -> int list` qui renvoie la liste des étiquettes d'un arbre des restes (les feuilles d'un tel arbre sont étiquetées par des polynômes constants, le coefficient intéressant étant donc la tête de la liste). On utilise une stratégie accumulative classique pour obtenir cette liste.

```
let rec parcourt T accu =
 match T with
 Vide -> []
|Noeud(P,Vide,Vide) -> (hd P)::accu
|Noeud(P,T1,T2) -> parcourt T1 (parcourt T2 accu) ;;
```

Cette fonction est de complexité linéaire vis à vis du nombre de noeuds de l'arbre (on fait un unique parcourt et on ne fait que des consages). La fonction princiaple s'en déduit.

```
let codageArbre alpha A =
parcourt  (arbreRestes (arbreSousProduits alpha) A) [] ;;
```

La création de la liste est de complexité $O(n)$ négligeable devant la création de l'arbre qui, elle, coûte $O(n^{\log_2(3)})$ ou $O(n^2)$ selon que l'on utilise ou pas les fonctions rapides.