1 Exercice du cours

Liste chaînée

```
let l4 = 4 in
let l3 = 3, ref l4 in
let l2 = 2, ref l3 in
let l1 = 1, ref l2 in
         0, ref l1;;
```

l'objet construit est de type `int * (int * (int * (int * int ref) ref) ref) ref`.
L'accès au 2ème élément (en comptant à partir de 0) d'une telle liste `l` se fait par

```
fst !(snd !(snd l))
```

2 Le crible d'Ératosthne

Question 1

On crée un tableau `crible : bool array` de taille $n+1$ commençant à zéro pour garantir que `crible.(i)` corresponde bien à la propriété `i` est premier (et pas `i+1`).

```
let eratosthene n =
  let crible = Array.make (n+1) true in
  crible.(0) <- false;
  crible.(1) <- false;
  let i = ref 2 in
  let k = ref 0 in
  while !i * !i <= n do
    if crible.(!i) then
      (k := 2* !i;
      while !k <= n do
        crible.(!k) <- false;
        k:=!k+ !i
      done);
    incr i;
  done;
  crible;;
```

Question 2

Si s est impair et s'écrit comme somme de deux nombres premiers alors nécessairement l'un des deux est 2. On effectue donc aucun parcours dans ce cas.
Sinon on parcourt le tableau en vérifiant à chaque k premier si $s-k$ est premier.

```
let decompose_somme t s =
  if  s>2 && (s=4 || s mod 2 = 1) then
    t.(s-2)
  else
    (let b = ref false in
     let k = ref 3 in
     while not !b && 2* !k <= s do
       b :=  t.(!k) && t.(s- !k);
       k := !k+2
     done;
     !b);;
```

```
(* en admettant la conjecture de Goldbach *)
let decompose_somme_Gb t s = (s>2) && (s mod 2=0 ||t.(s-2) );;
```

3 Polynômes

1)

On ne change qu'une seule chose le `r.(i) + q.(i)` devient une soustraction

```
(* soustraction *)
let (--) p q =
  let lp = Array.length p
  and lq = Array.length q
  in
  let r = Array.make (max lp lq) 0 in
  for i=0 to lp-1 do
    r.(i) <- p.(i)
  done;
  for i=0 to lq-1 do
    r.(i) <- r.(i) - q.(i)
  done;
  r;;
```

et pour la multiplication on parcourt les degrés k entre 0 et la somme des degrés de `p` et `q`. Et pour chaque k on a $k+1$ manière d'obtenir x^k : tous les $x^i x^j$ pour $i+j=k$. On parcourt donc ces possibilités en ajoutant à chaque fois le terme obtenu par le produit du monôme de degré i de p et du monôme de degré j de q.

```
(* multiplication *)
let ( ** ) p q =
  let lp = Array.length p
  and lq = Array.length q
  in
  let r = Array.make (lp + lq) 0 in
  for k = 0 to (lp+lq-1) do
    for i = 0 to min k (lp-1) do
      let j = k - i in
      if j < lq then
        r.(k) <- r.(k) + p.(i)*q.(j) ;
    done;
  done;
  r;;
```

2) Tchebychev

Mauvaise solution récursive

On reconnaît le calcul naïf de Fibonacci donc la complexité *en nombre d'opérations sur les polynômes* est exponentielle de base ϕ.

```
let rec tchebychev k =
  if k = 0 then
    [| 1 |]
  else if k = 1 then
    [| 0 ; 1 |]
  else
    [| 0 ; 2 |] ** tchebychev (k-1) -- tchebychev (k-2);;
```

Solution tableau

On crée une matrice de taille $(k+1)\times(k+1)$ telle que la ligne i contienne les coefficients de T_i. En effet on montre facilement par récurrence que T_i est de degré i.

```
let tchebychev k =
  let t = Array.make_matrix (k+1) (k+1) 0 in
  t.(0).(0) <- 1 ;
  t.(1).(1) <- 1 ;
  for i = 2 to k do
    let ti = t.(i-1) ** [|0;2|] -- t.(i-2) in
    let lti = Array.length ti in
    for j = 0 to min k (lti-1) do
      t.(i).(j) <- ti.(j);
    done;
  done;
  t.(k);;
```

Solution récursive terminale

On garde à chaque coup en mémoire les T_i et T_{i-1}.

```
let tchebychev k =
  let rec tchebychev_recursive d t_prec t_ante =
    if d = k then
      t_ante
    else
      tchebychev_recursive (d+1) ([|0;2|] ** t_prec -- t_ante) t_prec
  in
  tchebychev_recursive 0 [|0;1|] [|1|];;
```

3) Composition

```
let compose p q =
  let n = Array.length p in
  let rec compose_Horner p q i =
    if i >= n then
      [||]
    else
      [|p.(i)|] ++ (q ** (compose_Horner p q (i+1)))
  in
  compose_Horner p q 0;;
```

4) Division Euclidienne

1) Il s'agit de l'algorithme de division euclidienne.

2) À chaque étape de la boucle, si la condition est vérifiée, l'opération q_iX^iB coûte m multiplications et la soustraction à R de q_iX_iB coûte $\deg B = m$ additions car on ne modifie pas les autres coefficients de R. Ainsi on a $2m$ opérations auxquelles il faut ajouter le calcul $cd(R)u$. Une itération de la boucle coûte donc au plus $2m+1$, elles sont répétés $n-m+1$ fois et l'initialisation de u coûte une inversion. Ainsi on a une complexité en $O(2m(n-m))$ or si $n<m$ l'algorithme s'arrête immédiatement donc $O(mn)$.

3) On ne peut pas l'appliquer dans $\mathbb{Z}[X]$ car il n'est pas euclidien. Par exemple X^2-1 ne peut pas être divisé par $3X$ dans $\mathbb{Z}[X]$ puisque $1/3$ n'existe pas.

4)
```
let degre p =
    let n = ref ((Array.length p) - 1) in
    while p.(!n) = 0. do
      decr n;
    done;
    !n
  ;;

  let division a b =
    let n = degre a in
    let m = degre b in
    if  n < m then
      [|0.|], a
    else
      let r = Array.copy a in
      let q = Array.make (n-m+1) 0. in
      let u = (1./.b.(m)) in
      for i = (n-m) downto 0 do
        let d = degre r in
        if d = m+i then
          begin
            q.(i) <- r.(d) *. u;
            for j = 0 to m do
              r.(j+i) <- r.(j+i) -. q.(i) *. b.(j)
            done;
          end
        else
          q.(i) <- 0.;
      done;
      q, r;;
```

4 Jeu de Hex

On distingue tous les cas particuliers suivant i, j.

```
let voisins i j n =

  if i = 0 then                (* premi\‘ere ligne *)

    if j = 0 then              (* - coin en haut \‘a gauche *)
      [(0,1);(1,0)]
    else if j = n-1 then       (* - coin en haut \‘a droite *)
      [(0,n-1);(1,n-1);(1,n-2)]
    else                       (* - premi\‘ere ligne, cas g\'en\'eral *)
      [(i+1,j) ; (i,j-1) ; (i,j+1) ; (i+1,j-1)]

  else if i  = n - 1 then      (* derni\‘ere ligne *)

    if j = 0 then              (* - coin en bas \‘a gauche *)
      [(i-1,0);(i-1,1);(i,1)]
    else if j = n-1 then       (* - coin en bas \‘a droite *)
      [(i,n-2);(i-1,n-1)]
    else                       (* - derni\‘ere ligne, cas g\'en\'eral *)
      [(i-1,j) ; (i,j-1) ; (i,j+1) ; (i-1,j-1)]
```

```
  else                          (* ligne g\'en\'erale *)

    if j = 0 then               (* - colonne gauche *)
      [(i-1,j) ; (i-1,j+1) ; (i,j+1) ; (i+1,j)]
    else if j = n-1 then        (* - colonne droite *)
      [(i-1,j) ; (i+1,j-1) ; (i,j-1) ; (i+1,j)]
    else                        (* - cas g\'en\'eral *)
    [(i-1,j) ; (i+1,j) ; (i,j-1) ; (i,j+1) ; (i-1,j+1) ; (i+1,j-1)];;
```

On décompose de façon suivante

- Une fonction `init : unit -> bool * int * int * ( int -> int -> bool)` qui pour un joueur fixé cherche une de ses pièces sur la première ligne ou la première colonne (si le joueur est vainqueur une telle pièce existe forcément). Elle renvoie un quadruple (b, i_0, j_0, c) où `b` indique l'existence de la pièce recherchée, (i_0, j_0) ses coordonnées (si elle existe), et si en partant de (i_0, j_0) on visite à un certain moment la case (i, j), alors cij indique si la case atteint le bord *opposé* à (i_0, j_0).
- La fonction `parcours : (int * int) list -> (int -> int -> bool) -> bool` qui effectue le parcours des pièces du jeu. À chaque pièce non-finale appartenant au joueur courant elle ajoute ses voisins au parcours. En cas d'échec du parcours elle tente de relancer `init` (repartir d'une nouvelle pièce) et en cas de nouvel échec elle renvoie `false`.

```
let hex jeu =
  let n = Array.length jeu in
  let plateau = Array.make_matrix n n ' ' in
  for i = 0 to n-1 do
    plateau.(i) <- Array.copy jeu.(i)
  done;  (* On commence par copier le plateau pour pouvoir le modifier *)
  let minusc = function 'O' -> 'o' | 'X' -> 'x' | c -> c in
  (* On indique par des minuscules les cases d\'ej\`a visit\'es *)
  let cherchevictoire joueur =
    let init () = (* cherche une pi\`ece du joueur sur le bord haut ou gauche *)
      let k = ref 0 in
      let i0 = ref (-1) in
      let j0 = ref (-1) in
      while !k < n do
        begin
          if plateau.(!k).(0) = joueur then
            begin
              i0 := !k;
              j0 := 0;
              plateau.(!k).(0) <- minusc joueur; (* on marque la pi\`ece comme \'etant vue *)
              k := n;
            end
          else if plateau.(0).(!k) = joueur then
            begin
              i0 := 0;
              j0 := !k;
              plateau.(0).(!k) <- minusc joueur;
              k := n;
            end;
        end;
        incr k;
      done;
      let cond_victoire i j = if !i0 = 0 then i = n-1 else j = n-1 in
```

```
      if !k = n+1 then
        (true, !i0, !j0, cond_victoire)
      else
        (false, -1, -1, fun i j -> false)
    in
    let rec parcours l condition_victoire=
      match l with
      | [] -> let b, i0,j0, c_v = init () in
              if b then
                parcours (voisins i0 j0 n) c_v
              else false
      | (i,j) :: q when (plateau.(i).(j) = joueur) && (condition_victoire i j) ->  true
      | (i,j) :: q when plateau.(i).(j) = joueur -> begin
          plateau.(i).(j) <- minusc joueur;
          parcours (l @ (voisins i j n)) condition_victoire
        end
      | _ :: q -> parcours q condition_victoire
    in
  parcours [] (fun i j -> false)
  in
  if cherchevictoire 'O' then
    "vainqueur : O"
else if cherchevictoire 'X' then
    "vainqueur : X"
else
  "pas (encore) de vainqueur";;
```