CONCOURS E3A - INFORMATIQUE - 2002

Corrigé rédigé par Alain SCHAUBER - alain.schauber@prepas.org

Le langage utilisé dans ce corrigé est CAML.

1. Exponentiation rapide modulo m

1.1. Pour que cette fonction soit efficace du point de vue complexité, il convient de ne pas faire dans le programme plusieurs appels récursifs pour calculer un même résultat.

```
let rec exp_mod_m x n m =
  match n with
    0 -> 1 (* il s'agit de calculs modulo m, donc le resultat est un entier *)
   |_ -> let p = n / 2 and r = n mod 2
         in let y = exp_mod_m x p m
            in if r = 0 then (y * y) mod m
               else (x * y * y) mod m
;;
```

1.2. Si on appelle u_n le nombre de multiplications et de calculs de modulo en fonction de n, on constate que $u_0 = 0$ et que pour $n \geq 1$, $u_n = u_{E(n/2)} + 2$ si n est pair et $u_n = u_{E(n/2)} + 3$ si n est impair.

Faisons alors intervenir pour $n \neq 0$ l'écriture de n en base 2 : $n = \sum_{i=0}^{p} n_i 2^i$, avec $n_p \neq 0$.

Alors $E(n/2) = \sum_{i=1}^{p} n_i 2^{i-1}$ et $n_0 = 0$ ou 1 selon la parité de n. De plus, pour $n = 0$ on a $p = 0$ et $n_0 = 0$.

On en déduit alors par récurrence que $\boxed{u_n = \sum_{i=0}^{p} n_i + 2p}$. Cette « formule » exacte est néanmoins de peu d'intérêt en soi. Cependant, les n_i étant des chiffres binaires, on a $\sum_{i=0}^{p} n_i \leq p + 1$, de sorte que $u_n \leq 3p + 1$.

Comme $p = E[log_2 n]$ pour $n \geq 1$, on en déduit une évaluation intéressante de la complexité de cette fonction :

La complexité de cette fonction par rapport au nombre de multiplications et de calculs de modulo est un $O(\ln n)$

2. Résolution de l'équation $f(x) = 0$ par la méthode des sécantes

2.1. La droite passant par les points $[x_1, y_1]$ et $[x_2, y_2]$ a pour équation $(y - y_1)(x_2 - x_1) = (y_2 - y_1)(x - x_1)$. Par conséquent, le point d'intersection a pour abscisse $\dfrac{x_1 y_2 - x_2 y_1}{y_2 - y_1}$.

```
let intersection x1 y1 x2 y2 = (x1 *. y2 -. y1 *. x2) /. (y2 -. y1) ;;
```

2.2 Une programmation récursive permet ici de faire l'économie de la gestion de variables locales de type référence (c'est le compilateur qui s'en charge).

```
let rec zero_secante f x0 x1 =
  if abs_float (x1 -. x0) <= 0.00001 then x1
  else zero_secante f x1 (intersection x0 (f x0) x1 (f x1))
;;
```

3. Carré magique

3.1. La somme des N^2 termes est $\sum_{k=0}^{N^2} k = \dfrac{N^2\left(N^2+1\right)}{2}$, par suite la somme sur une ligne (par exemple) s'obtient en divisant ce résultat par N, donc la constante d'un carré magique d'ordre N vaut $\dfrac{N\left(N^2+1\right)}{2}$

3.2. La vérification du caractère magique d'une matrice consiste à contrôler que les termes de la matrice prennent bijectivement toutes les valeurs entre 1 et N^2, puis à calculer toutes les sommes des lignes, colonnes et diagonales

afin de les comparer à la constante d'une matrice magique d'ordre N. A chaque étape on utilise une boucle qui met à jour une variable booléenne « résultat » contenant in fine le résultat final.
Pour le contrôle des valeurs entre 1 et N, on crée un vecteur de booléens de longueur N^2, dont chaque cellule est mise à VRAI dès lors que l'indice de la cellule est présent dans la matrice. Puis on parcourt ce vecteur en faisant un « et logique » avec la variable « résultat ». A noter que cette étape est incontournablement en $O(N^2)$, ce qui attribue à l'ensemble de la fonction la même complexité. Hormis d'un point de vue esthétique, il est donc d'un point de vue théorique inutile de chercher à raccourcir le programme en introduisant un arrêt de l'une des boucles dès qu'une condition à vérifier s'avère fausse.

```
let est_magique matrice =
  let N = vect_length matrice and resultat = ref true
  in (* d'abord on teste si tous les termes sont entre 1 et N^2 et distincts *)
     let v = make_vect (N*N) false
     in for i = 1 to N do
          for j = 1 to N do
            if matrice.(i-1).(j-1) >= 1 & matrice.(i-1).(j-1) <= N*N
            then v.(matrice.(i-1).(j-1) - 1) <- true
          done
        done;
     for k = 0 to N*N - 1 do resultat := !resultat & v.(k) done;
     (* ensuite on examine la somme de chaque ligne *)
     for i = 1 to N do
       let s = ref 0
       in for j = 1 to N do s := !s + matrice.(i-1).(j-1) done;
          resultat := !resultat & (!s = N*(N*N+1)/2)
     done;
     (* puis on examine la somme de chaque colonne *)
     for j = 1 to N do
       let s = ref 0
       in for i = 1 to N do s := !s + matrice.(i-1).(j-1) done;
          resultat := !resultat & (!s = N*(N*N+1)/2)
     done;
     (* enfin on examine les sommes des deux diagonales *)
     let s = ref 0
     in for i = 1 to N do
          s := !s + matrice.(i-1).(i-1)
        done;
     resultat := !resultat & (!s = N*(N*N+1)/2);
     s := 0;
     for i = 1 to N do
       s := !s + matrice.(i-1).(N-i)
     done;
     resultat := !resultat & (!s = N*(N*N+1)/2);
     !resultat
;;
```

3.3. En suivant l'algorithme proposé, on trouve magiquement :

11	24	7	20	3
4	12	25	8	16
17	5	13	21	9
10	18	1	14	22
23	6	19	2	15

3.4. Le programme consiste à initialiser une matrice de taille n avec des zéros partout, puis à initialiser les indices i et j de ligne et de colonne respectivement à $E(n/2)+1$ et $E(n/2)$, ainsi qu'un compteur k à 1.
Ensuite, on lance une boucle conditionnelle contrôlée par $k \leq n^2$, qui met à jour la cellule courante de la matrice avec la valeur courante de k, incrémente k, puis déplace la cellule courante à celle en dessous à droite en vision torique, c'est-à-dire modulo n. Si celle-ci est déjà remplie (valeur non nulle dans la cellule), on se lance dans une boucle de recherche d'une cellule libre en se déplaçant cette fois systématiquement en dessous à gauche (toujours modulo n). C'est l'énoncé qui nous assure que l'algorithme se termine bien, faisons-lui confiance...

```
let construit_carre_magique n = (* on suppose que n est impair superieur ou egal a 3 *)
  (* dans les indices i et j ci-dessous, on decale de 1 a cause de l'indexation CAML qui debute a 0 *)
  let m = make_matrix n n 0 and i = ref (n/2 + 1) and j = ref (n/2) and k = ref 1
  in while !k <= n*n do
       m.(!i).(!j) <- !k;
       incr k;
       if !k <> 26 then (* si k = 26 la matrice est pleine et la boucle ci-dessous ne termine jamais *)
       begin
         i := (!i + 1) mod n;
         j := (!j + 1) mod n;
         while m.(!i).(!j) <> 0 do
           i := (!i + 1) mod n;
           j := (!j + n - 1) mod n (* en CAML, "mod" n'est pas fiable avec un nombre negatif *)
         done
       end
     done;
     m
;;
```

4. Grands nombres

Les constantes et fonctions prédéfinies dans l'énoncé pourraient se définir ainsi en CAML :

```
type booleen == bool;;
type chiffre == int;;
(* la liste vide est deja define par [] en CAML *)
let estVide(lst) = lst = [];;
let cons(c,lst) = c::lst;;
let tete(lst) = hd lst;;
let queue(lst) = tl lst;;
let resultatPlus(c1,c2) = (c1 + c2) mod 10;;
let retenuePlus(c1,c2) = (c1 + c2) / 10;;
let resultatFois(c1,c2) = (c1 * c2) mod 10;;
let retenueFois(c1,c2) = (c1 * c2) / 10;;
```

4.1. Parcours classique d'une liste pour vérification qu'elle ne contient que des 0 :

```
let rec estNul(lst) = estVide(lst) or (tete(lst) = 0 & estNul(queue(lst)));;
```

4.2. Parcours simultané classique de deux listes :

```
let rec estEgal(lst1,lst2) =
  (estNul(lst1) & estNul(lst2))
  or (not estNul(lst1))
     & (not estNul(lst2))
     & (tete(lst1) = tete(lst2))
     & estEgal(queue(lst1),queue(lst2)))
;;
```

4.3. D'après l'exemple, il semble que le résultat de cette fonction doive être de la longueur de la plus longue des deux listes, sans gestion des zéros inutiles éventuellement présents :

```
let rec resultatAjoute(lst1,lst2) =
  if estVide(lst1) then lst2 (* d'apres la remarque, plutot estVide que estNul *)
  else if estVide(lst2) then lst1
       else cons(resultatPlus(tete(lst1),tete(lst2)),resultatAjoute(queue(lst1),queue(lst2)))
;;
```

4.4. Apparemment, la liste des retenues doit avoir la même longueur que celle de la fonction précedente, ce qui oblige à un petit effort pour insérer les zéros éventuels à droite :

```
let rec retenueAjoute(lst1,lst2) =
  if estVide(lst1) & estVide(lst2) then []
  else if estVide(lst1) then cons(0,retenueAjoute([],queue(lst2)))
       else if estVide(lst2) then retenueAjoute([],lst1) (* commutativite de l'addition *)
            else cons(retenuePlus(tete(lst1),tete(lst2)),retenueAjoute(queue(lst1),queue(lst2)))
;;
```

4.5. Les listes des additions et des retenues ont la même longueur, on peut donc suivre l'algorithme classique d'addition en base 10 enseigné à l'école primaire. On effectue l'addition de la retenue de rang i avec le chiffre d'addition de rang i+1 en mettant à jour la retenue de rang i+1, ce qui se programme à l'aide d'une fonction récursive locale "somme" :

```
let nombreAjoute(lst1,lst2) =
  let a = resultatAjoute(lst1,lst2) and r = retenueAjoute(lst1,lst2)
  in let rec somme(lst3,lst4) =
       if estVide(lst3) then []
       else if estVide(queue(lst3)) then if tete(lst4) <> 0 then cons(tete(lst3),lst4) else lst3
            else cons(tete(lst3),
                      somme(cons(resultatPlus(tete(queue(lst3)),tete(lst4)),queue(queue(lst3))),
                            cons(resultatPlus(retenuePlus(tete(queue(lst3)),tete(lst4)),
                                                          tete(queue(lst4))),
                                 queue(queue(lst4)))))
     in somme(a,r)
;;
```

4.6. Parcours classique d'une liste :

```
let rec resultatMultiplie(lst,c) =
  if estVide(lst) then []
  else cons(resultatFois(tete(lst),c),resultatMultiplie(queue(lst),c))
;;
```

4.7. Idem :

```
let rec retenueMultiplie(lst,c) =
  if estVide(lst) then []
  else cons(retenueFois(tete(lst),c),retenueMultiplie(queue(lst),c))
;;
```

4.8. Il suffit d'ajouter à la liste des multiplications la liste des retenues, décalée d'un rang vers la droite, ce qui peut se faire en reprenant la fonction auxiliaire "somme" de la question 4.5. :

```
let multiplie(lst,c) =
  let a = resultatMultiplie(lst,c) and r = retenueMultiplie(lst,c)
  in let rec somme(lst3,lst4) =
       if estVide(lst3) then []
       else if estVide(queue(lst3)) then if tete(lst4) <>0 then cons(tete(lst3),lst4) else lst3
            else cons(tete(lst3),somme(cons(resultatPlus(tete(queue(lst3)),tete(lst4)),
                                            queue(queue(lst3))),
                      cons(resultatPlus(retenuePlus(tete(queue(lst3)),tete(lst4)),tete(queue(lst4))),
                           queue(queue(lst4)))))
     in somme(a,r)
;;
```

4.9. On fait l'addition des multiplications de lst1 par chaque chiffre de lst2, en respectant un décalage à chaque étape, pour tenir compte du chiffre des unités, des dizaines, etc. :

```
let rec nombreMultiplie(lst1,lst2) =
  if estVide(lst2) then []
  else nombreAjoute(multiplie(lst1,tete(lst2)),nombreMultiplie(cons(0,lst1),queue(lst2)))
;;
```

4.10. Fonction itérative classique du calcul d'une factorielle :

```
let factorielle(lst) =
  let n = ref [] and facto = ref (cons(1,[])) (* n initialise a 0 et facto a 1 *)
  in while not estEgal(!n,lst) do
       n := nombreAjoute(!n,cons(1,[]));      (* n <- n+1 *)
       facto := nombreMultiplie(!facto,!n);   (* facto <- facto x n *)
     done;
  !facto
;;
```

4.11. Cette fonction aurait pu être définie plus tôt !

```
let rec enleverZero(lst) =
  if estNul(lst) then []
  else cons(tete(lst),enleverZero(queue(lst)))
;;
```

4.12. Il s'agit là du calcul classique de la longueur d'une liste, après avoir débarassé celle-ci de ses zéros non significatifs :

```
let nombreChiffres(lst) =
  let rec compteChiffres(lst1) = (* s'applique a une liste epuree de ses zeros non significatifs *)
    if estVide(lst1) then cons(0,[]) (* puisque 0 se represente par une liste vide *)
    else nombreAjoute(compteChiffres(queue(lst1)),cons(1,[]))
  in compteChiffres(enleverZero(lst))
;;
```