Correction de la feuille de TD n°3 : diviser pour régner

MPSI option informatique

Mars 2025

Exercice 1 : Ecrire en Caml une fonction qui calcule x^n pour x et n entiers, utilisant l'exponentiation rapide.

On proposera une version itérative et une version récursive.

Correction : version itérative : utilisation de l'écriture binaire de la puissance

```
let puiss a n = let res = ref 1 and m = ref n and x = ref a in
                    while !m<>0 do
                       if !m mod 2 = 1 then res:= !res * !x;
                       m:=!m/2;
                       x:=!x* !x;
                       done;
                    !res;;
```

première version : on appelle récursivement : dans le cas pair $x^n = (x \cdot x)^{n/2}$

```
let rec puiss_rec x n = match n with
                        | 0 -> 1
                        | n when n mod 2 = 0 -> puiss_rec (x*x) (n/2)
                        | _ -> x*(puiss_rec (x*x) (n/2));;
```

seconde version : dans le cas pair $x^n = x^{n/2} \cdot x^{n/2}$, en ne faisant qu'une seule fois l'appel récursif.

```
let rec puiss_rec2 x n = match n with
                         | 0 -> 1
                         | _ -> let y = puiss_rec2 x (n/2) in
                                if n mod 2 = 0 then y*y
                                     else x*y*y;;
```

Pour obtenir une récursivité terminale on peut aussi utiliser le principe de la version itérative :

```
let puiss_rec3 x n = let rec aux y n res = match n with
                         | 0 -> res
                         | _ -> if n mod 2 = 1 then aux (y*y) (n/2) (res*y)
                                       else aux (y*y) (n/2) res
                         in aux x n 1;;
```

Exercice 2 : Programmer une fonction de fusion de deux segments adjacents triés d'un même tableau, en un segment de tableau trié *(on pourra utiliser un tableau auxiliaire)*.

En déduire un programme mettant en œuvre l'algorithme de "tri fusion" d'un tableau.

Correction : voici la version finale de la fonction de tri

```
let tri_fusion_tableau t =
    let fusion  g m d =
        let i= ref g and j = ref m and aux = Array.make (d-g) 0 in
        while !i< m && !j<d do
            if t.(!j)<t.(!i)
               then begin
                aux.(!i+ !j-g-m)<- t.(!j);
                incr(j);
               end
               else begin
                aux.(!i+ !j-g-m)<- t.(!i);
                incr(i);
               end;
        done;
        for k=(!i) to m-1 do  (* ceci ne fera rien si !i>m-1 *)
          aux.(k+ d-g-m)<- t.(k) done;
(* on remet les éléments à leur place sauf ceux déjà correctement placés*)
        for k=g to !j-1 do
          t.(k)<- aux.(k-g) done;
    in
    let rec tri g d = match d-g with
          | 0 -> () (*normalement inutile*)
          | 1 -> ()
          | _ -> let m = (g+d)/2 in
                 tri g m;
                 tri m d;
                 fusion  g m d;
    in tri 0 (Array.length t);
    t;;(*t a été modifié par effet de bord, il n'est pas obligatoire de le redonner*)
```

Exercice 3 : *Tri par fusion d'une liste* :

1) Écrire une procédure de partage d'une liste en deux listes de tailles égales (éventuellement à 1 près !), puis une fonction réalisant la fusion de deux listes triées.

2) En déduire un programme de tri par fusion d'une liste. En évaluer la complexité.

Correction :

```
let tri_fusion_liste lst =
    let rec partage lst = match lst with
            | [] -> [],[]
            | [a] -> [a],[]
            | a::b::q -> let lst1,lst2 = partage q in
            (* ce partage permet de garder l'ordre*)
                    (a::lst1),(b::lst2)
     and
        fusion lst1 lst2 = match lst1,lst2 with
            | [],_ -> lst2
            | _ ,[]-> lst1
            | a::q1,b::q2 -> if a <b then
                                a::(fusion q1 lst2)
                            else b::(fusion lst1 q2)
     and
        tri lst = match lst with
            | [] -> []
            | [a] -> [a]
            | _ -> let lst1,lst2 = partage lst in
               fusion (tri lst1) (tri lst2)
     in
     tri lst;;
```

La séparation et la fusion sont linéaires donc la complexité est du même ordre de grandeur ($n \log_2(n)$) que pour les tableaux.

Exercice 4 : Écrire un programme déterminant le maximum d'un tableau par une méthode « diviser pour régner ». Évaluer sa complexité.

Correction :

```
let maximum t =
    let rec max_troncon i j =
        if i=j then t.(i)
            else let k = (i+j)/2 in
                let maxg = max_troncon i k and maxd = max_troncon (k+1) j in
                max maxg maxd
    in max_troncon 0 (Array.length t - 1);;
```

La fonction `max_troncon` termine bien puisque dans le cas $i < j$, on aura $i - k < j$, donc $0 \leqslant k - i < j - i$ et $0 \leqslant j - (k + 1) < j - i$.

Si $C(n)$ est la complexité en nombre de comparaisons pour un tableau de taille n, on a : $C(n) = C(b\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(1)$
Avec les notations du cours, $a + b = 2$, $\omega = 1 > 0 = p$, donc $C(n) = \Theta(n)$, on n'a rien gagné par rapport à l'algorithme naïf.

Exercice 5 : Calcul des termes de la suite de Fibonacci.

La suite de Fibonacci est définie par les relations : $F_0 = F_1 = 1$ et pour tout $n \geq 0$, $F_{n+2} = F_{n+1} + F_n$.

1) Ecrire un programme récursif calculant F_n pour $n \geq 0$.

Déterminer le nombre d'appels à la fonction selon n. On posera T(n) le nombre d'appels à la fonction pour le calcul de F_n, puis on donnera une relation entre $T'(n) = T(n) + 1$, $T'(n-1)$ et $T'(n-2)$ pour $n \geq 2$.

Correction : pas de nouveautés !!

```
rec fibo n = match n with
             | 0 -> 1
             | 1 -> 1
             | _ -> fibo (n-1) + fibo (n-2);;
```

On a immédiatement que $T(n) = 1 + T(n-1) + T(n-2)$ avec $T(0) = T(1) = 1$.

On en déduit que T' vérifie la récurrence de la suite de Fibonacci : $T'(n) = T'(n-1) + T'(n-2)$, avec $T'(0) = T'(1) = 2$. On en déduit que $T(n) = \left(\frac{\sqrt{5}}{5} + 1\right)\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{-\sqrt{5}}{5} + 1\right)\left(\frac{1-\sqrt{5}}{2}\right)^n + 1$.

2) Ecrire un programme récursif calculant le couple (F_{n-1}, F_n).

Déterminer le nombre d'appels à la procédure selon n.

Correction :

```
let  fibo2 n =
         let rec aux (a , b) n = match n with
            | 1 -> (a,b)
            | _ -> aux ( b , b + a  ) (n-1);
         in snd (aux (1,1) n);;
```

On a immédiatement n appels à la fonction.

3) Démontrer la relation : $\forall (n,p) \in \mathbb{N}^{*2}$, $F_{n+p} = F_n F_p + F_{n-1} F_{p-1}$.

En déduire un programme de calcul de F_n selon la méthode "diviser pour régner".

Déterminer le nombre d'appels à la procédure selon n.

Correction : cela se fait par récurrence double sur p (ou sur n).

Pour la fonction il faudra distinguer le cas n pair et n impair.

On a en effet : $F_{2n} = F_n^2 + F_{n-1}^2$, $F_{2n-1} = F_{n-1}(2F_n - F_{n-1})$ et $F_{2n+1} = F_n(F_n + 2F_{n-1})$.

On va encore calculer les couples $T_n = (F_{n-1}, F_n)$.

On a $T_{2n} = \big(F_{n-1}(2F_n - F_{n-1}), F_n^2 + F_{n-1}^2\big)$ et $T_{2n+1} = \big(F_n^2 + F_{n-1}^2, F_n(F_n + 2F_{n-1})\big)$.

```
let fibo3 n =
    let rec aux  n = match n with
        | 0 -> (0,1)
        | 1 -> (1,1)
        | _ -> let (a,b) = aux (n/2) in
                 if n mod 2 = 0
                    then (a*(2*b-a),a*a+b*b)
                    else (a*a +b*b, b*(b+2*a))
    in snd (aux n);;
```

Le nombre d'appel à la fonction vérifie $C_n = 1 + C_{\lfloor n/2 \rfloor}$ et $C_1 = 1$, d'où $C_n = 1 + \lfloor \log_2(n) \rfloor$

Exercice 6 : Une médiane d'un ensemble E de cardinal n est un élément $m \in E$ tel que

$$Card\{x \in E / x < m\} = \left\lfloor \frac{n}{2} \right\rfloor$$

1) Montrer l'unicité de la médiane d'un ensemble.

Correction : Pour $m < m'$ on a $m \in \{x \in E / x < m'\}$, d'où

$$Card\{x \in E / x < m\} + 1 \leqslant Card\{x \in E / x < m'\}$$

On en déduit que $Card\{x \in E / x < m\} + 1 \neq Card\{x \in E / x < m'\}$ et donc m et m' ne peuvent être simultanément médiane, d'où l'unicité.

2) Donner un algorithme de complexité $\Theta(n^2)$ (mesurée en comparaisons) pour trouver la médiane d'un tableau.

Correction : On parcourt le tableau et, pour chaque élément, on compte le nombre d'éléments qui lui sont strictement inférieurs. On renvoie l'élément pour le quel ce nombre est $\left\lfloor \frac{n}{2} \right\rfloor$.

```
let mediane t =
    let n = Array.length t
    and i = ref 0 and res = ref 0
    and compte = ref 0 in
    while (!i < n) do
       for k = 0 to (n-1) do
         if t.(k) < t.(!i) then compte:=!compte+1
       done;
       if !compte = n/2
          then res :=!i
          else compte := 0;
       incr(i)
    done;
    t.(!res);;
```

3) Donner un algorithme de complexité $\Theta(n)$ (mesurée en comparaisons) pour trouver la médiane de deux tableaux ordonnés de même longueur n.

Correction : On parcourt le tableau et, pour chaque élément, on compte le nombre d'éléments qui lui sont strictement inférieurs. On renvoie l'élément pour le quel ce nombre est $\left\lfloor \frac{n}{2} \right\rfloor$.

```
let mediane t =
    let n = Array.length t
    and i = ref 0 and res = ref 0
    and compte = ref 0 in
    while (!i < n) do
       for k = 0 to (n-1) do
         if t.(k) < t.(!i) then compte:=!compte+1
       done;
       if !compte = n/2
          then res :=!i
          else compte := 0;
       incr(i)
    done;
    t.(!res);;
```

Exercice 7 : Programmer le tri rapide sur un tableau. La fonction programmée doit modifier le tableau en entrée, sans jamais créer de nouveau tableau. La partition doit effectuer un nombre d'opération linéaire en la taille du tableau.

Correction :

```
let quicksort t =
    let interversion i j =
        (*Intervertit t[i] et t[j]*)
        let temp = t.(i) in t.(i) <- t.(j) ; t.(j) <- temp
    in
    let partition_pivot i j =
        (*Partitionne le tronçon t[i]...t[j], en utilisant son premier
        élément comme pivot. Renvoie la position du pivot après partition*)
        let pivot = t.(i) and pos_debut = ref (i+1) and pos_fin = ref j in
        while !pos_fin - !pos_debut > -1 do
            (*le tronçon est de la forme (<pivot)(pivot)(atrier)(>pivot)
            pos_debut et pos_fin délimitent la partie à trier*)
            if t.(!pos_debut) < pivot
                then begin interversion (!pos_debut - 1) !pos_debut;
                    pos_debut := !pos_debut + 1; end
                else begin interversion !pos_debut !pos_fin;
                    pos_fin := !pos_fin - 1; end
        done;
        !pos_debut - 1
    in
    let rec tri_troncon i j =
        if j-i > 0
            then begin let pos_pivot = partition_pivot i j in
                tri_troncon i (pos_pivot-1); tri_troncon (pos_pivot+1) j end
    in
    tri_troncon 0 (Array.length t - 1);;
```

Autre méthode : sans envoyer d'élément à la fin, toujours avec le premier élément comme pivot. On ne bouge pas le pivot avant la fin.
Si on note `pk` la position que devrait avoir le pivot après avoir traité le cas du $k-1$ élément. Si $t.(k) < pivot$ on échange $t.(pk+1)$ avec $t.(k)$ car celui-ci est plus grand que le pivot (penser au fait que $k > pk$ et on incrémente `pk`.
A la fin dans $t.(pk)$ il y a un élément plus petit que le pivot. On l'échange donc avec $t.(i)$.

```
let quicksort t =
    let interversion i j =
        (*Intervertit t[i] et t[j]*)
        let temp = t.(i) in t.(i) <- t.(j) ; t.(j) <- temp
    in
    let partition_pivot i j =
        (*Partitionne le tronçon t[i]...t[j], en utilisant son premier
        élément comme pivot. Renvoie la position du pivot après partition*)
        let pivot = t.(i) and  k = ref (i+1) and pk = ref i in
        while !k<=j do
            if t.(!k) < pivot
```

```
            then begin interversion (!pk+1) !k;
                pk := !pk + 1; end;
        incr(k);
    done;
    interversion i !pk;
    !pk
in
let rec tri_troncon i j =
    if j-i > 0
        then begin let pos_pivot = partition_pivot i j in
            tri_troncon i (pos_pivot-1); tri_troncon (pos_pivot+1) j end
in
tri_troncon 0 (Array.length t - 1);;
```

Exercice 8 : Étant donnée une suite finie d'entiers $x = (x_1, \ldots, x_n)$, on appelle inversion de x tout couple (i, j) tel que $i < j$ et $x_i > x_j$. Par exemple, $(2, 3, 1, 5, 4)$ possède 3 inversions : les couples $(1, 3)$, $(2, 3)$, $(4, 5)$. On s'intéresse au calcul du nombre d'inversions de x.

1) Rédiger en Caml l'algorithme naïf, et montrer que son coût est un $\Theta(n^2)$. On représentera les suites finies d'entiers par le type `int array`.

Correction : on teste tous les couples

```
let inversion t =
    let n = Array.length t in
    let s = ref 0 in
    for i = 0 to n-2 do
      for j = i+1 to n-1 do
        if t.(i) > t.(j) then s := !s + 1
      done
    done ;
    !s ;;
```

Le nombre de comparaisons effectuées entre éléments du tableau vaut :

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n - i - 1) = \frac{n(n-1)}{2} = \Theta(n^2)$$

2) Adopter une méthode « diviser pour régner »pour faire mieux (indication : adapter l'algorithme de tri fusion).

Correction : Adopter une stratégie « diviser pour régner »consiste à séparer le tableau en deux parties sensiblement égales de longueur k et $n - k$, à calculer récursivement le nombre d'inversions présentes dans chacune des deux parties, puis à déterminer les inversions non comptabilisées : celles qui sont à cheval sur les deux parties, c'est à dire pour lesquelles $i \leqslant k < j$. D'après le théorème maître, ceci doit pouvoir être réalisé en temps linéaire pour faire mieux que l'algorithme naïf.

Nous allons montrer que ceci est possible lorsque les parties gauche et droite sont triées. Ainsi, nous allons supposer que les appels récursifs retournent non seulement le nombre d'inversions mais aussi la liste triée (par ordre croissant) des éléments.

Nous aurons besoin d'une fonction prenant en argument deux listes triées (y_i) et (z_j) et dénombrant le nombre de couples (y_i, z_j) vérifiant $y_i > z_j$:

```
let acheval l1 l2 =
    let rec aux acc n y z = match (y,z) with
         | ([],_) -> acc
         | (_,[]) -> acc
         | (a::qa,b::qb) when a > b -> aux (n + acc) n y qb
         | (a::qa,z) -> aux acc (n-1) qa z
    in aux 0 ( List.length l1) l1 l2 ;;
```

Il est clair que cette fonction a un coût linéaire vis-à-vis de $|y| + |z|$ puisque chaque appel récursif diminue la longueur d'une des deux listes d'une unité. L'utilisation de l'instruction `List.length` est aussi linéaire.

La validité de cette fonction résulte de la constatation suivante :
si $y = (y_1 \leqslant y_2 \leqslant \cdots \leqslant y_n)$ et $z = (z_1 \leqslant z_2 \leqslant \cdots \leqslant z_p)$, alors :
— si $y_1 > z_1$ les n couples (y_i, z_1) vérifient $y_i > z_1$
— si $y_1 \leqslant z_1$, aucun couple (y_1, z_j) ne vérifie $y_1 > z_j$
La fonction générale se définit alors ainsi (la fonction fusion est celle de l'exercice 4, elle a pour objet de fusionner deux listes triées en coût linéaire) :

```
let inversionbis t =
    let rec aux = function
         | (i, j) when i = j -> 0, [t.(i)]
         | (i, j) -> let k = (j + i) / 2 in
                     let (c1,y) = aux (i,k) and (c2,z) = aux (k+1,j) in
                     (c1 + c2 + acheval y z), (fusion y z)
    in match (Array.length t) with
       | 0 | 1 -> 0
       | n -> fst (aux (0, n-1)) ;;
```

Le coût de cette fonction vérifie la relation :

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(n) = \Theta(n \log_2(n))$$

On peut regrouper toutes les fonctions en une seule qui permettra de ne pas utiliser `List.length` puisque la longueur est connue car dépendante de `(i,j)`.

```
let inversionter t =
let rec fusion lst1 lst2 = match lst1,lst2 with
            | [],_ -> lst2
            | _ ,[]-> lst1
            | a::q1,b::q2 -> if a <b then
                                a::(fusion q1 lst2)
                             else b::(fusion lst1 q2)
         in
         let acheval l1 l2 n1 =
    let rec aux acc n y z = match (y,z) with
           | ([],_) -> acc
          | (_,[]) -> acc
          | (a::qa,b::qb) when a > b -> aux (n + acc) n y  qb
          | (a::qa,z) -> aux acc (n-1) qa z
     in aux 0 n1 l1 l2
    in
    let rec aux (i,j) = match (i,j) with
```

```
      | (i, j) when i = j -> 0, [t.(i)]
      | (i, j) -> let k = (j + i) / 2 in
                  let (c1,y) = aux (i,k) and (c2,z) = aux (k+1,j) in
                  (c1 + c2 + acheval y z (k-i+1)), (fusion y z)
in match (Array.length t) with
   | 0 | 1 -> 0
   | n -> fst (aux (0, n-1)) ;;
```