Correction du devoir surveillé n°2 : concours blanc

MPSI Lycée Clemenceau : option informatique

Lundi 28 avril 2025

Exercice I : manipulations de listes

1) Ecrire une fonction `pas_nulle` qui prend en entrée une liste d'entier et retourne la même liste privée des zéros éventuels : `pas_nulle : int list -> int list`

Exemple :

```
# pas_nulle [4;2;0;5;-3;0;1;8];;
    - : int list = [4; 2; 5; -3; 1; 8]
```

Correction : on utilise le constructeur d'une liste pour construire la liste par appels récursifs.

```
# let rec pas_nulle lst = match lst with
        | [] -> []
        | a::q -> if a=0 then (pas_nulle q) else a::(pas_nulle q);;
```

2) Ecrire une fonction `intervalle` qui prend en donnée deux entier `a` et `b` (avec `a<b`) et une liste `lst` et renvoie un triplet de listes `(l1,l2,l3)` tel que `l1` contient les éléments de `lst` strictement inférieur à `a`, `l3` contient les éléments de `lst` strictement supérieur à `b` et `l2` le reste de la liste `l`.

Exemple :

```
# intervalle 5 10 [2;4;5;16;54;12;15;8;94;1;0;5;6;7;10];;
    - : int list * int list * int list = [2; 4; 1; 0], [5; 8; 5; 6; 7; 10], [16; 54; 12; 15; 94]
```

Correction : on utilise un appel récursif en construisant le triplet avant l'appel sur la queue de la liste afin de conserver l'ordre des éléments de la liste.

```
# let rec intervalle a b lst = match lst with
        | [] - > [],[],[]
        | x::q -> let lst1,lst2,lst3 = intervalle a b q in
            if x < a then x::lst1,lst2,lst3 else
              if x > b then lst1,lst2,x::lst3 else lst1,x::lst2,lst3;;
```

3) Ecrire une fonction `indiceTripleMax` qui renvoie le couple `(indice, somme)` d'une liste où `indice` est l'indice du premier élément du triplet de somme maximale.

`indiceTripleMax : int list -> int * int`

Exemple :

```
# indiceTripleMax [1;5;7;8;6;2;7;9;4];;
    - : int * int = 3, 21
```

Remarque : on parle d'indice d'un élément de la liste en commençant à 1, ce n'est pas un tableau.

Correction : on commence par tester si la liste est assez longue, puis on utilise une fonction auxiliaire qui « remonte »récursivement au bout de la liste pour calculer la maximum de la somme d'un triplet.

```
# let indiceTripleMax lst =
        if lst=[] || List.tl lst=[] || List.tl (List.tl lst)=[] then failwith "liste trop courte"
            else let rec aux = function
                    | a::b::c::[] -> (1,a+b+c)
                    | a::b::c::q -> let (i,m) = aux (b::c::q) in
                            if a+b+c>= m then (1,a+b+c) else (i+1,m)
                    in aux lst;;
```

Exercice II : e3a 2015

On admettra que la multiplication de deux entiers naturels se fait en temps constant.

1) Ecrire une fonction `puissance` de type `int -> int -> int` qui calcule, pour deux entiers n et k non nuls, n^k.

Ce programme devra être récursif ou faisant appel à des fonctions auxiliaires récursives, et devra utiliser le principe de diviser pour régner.

Donner la complexité de ce programme.

Correction : il suffit de savoir son cours.

On peut écrire par exemple :

```
let rec puissance n k = match k with
    | 0 -> 1 (*inutile avec la demande du texte*)
    | 1 -> n (*non obligatoire avec le premier cas, évite un appel récursif*)
    | _ -> let s = puissance n (k/2) in
           if (k mod 2) = 0 then s*s else n*s*s;;
```

La complexité est en $\Theta(\log_2(k))$.

2) On s'intéresse ici au problème de déterminer si un entier naturel est une puissance non triviale d'un nombre entier ou non : on dit qu'un nombre entier naturel n est une *puissance entière* s'il existe deux entiers naturels k et m tous deux > 1 tels que $n = m^k$.

a) Écrire la fonction :

`test_puissance : int -> int -> bool`

qui prend en entrée les entiers naturels $n > 1$ et $k > 1$ et renvoie le booléen `true` s'il existe un entier naturel m tel que $n = m^k$ et `false` sinon.

On proposera une version itérative et une version récursive.

Correction : on utilise la fonction `puissance` de la première question et une boucle conditionnelle car il est inutile de tester tous les entiers si on en trouve un.

```
let test_puissance n k = let m= ref 2 in
                         let p= ref puissance !m k in
                         while !p <n do  incr m ;
                                         p:= puissance !m k
                                         done ;
                         !p = n ;;
```

Version récursive :

```
let test_puissance n k =
let rec aux m = match (puissance m k) with
                |  p when p>n -> false
                |  p -> if p=n then true else aux (m+1)
                in aux 2;;
```

b) **i)** Soit n un entier naturel. On suppose que n est une puissance entière : Soient k et m deux entiers naturels > 1 tels que $n = m^k$. Justifier que $k \leqslant \log_2(n)$.

Correction : On a $\log_2(n) = k\log_2(m)$ comme $m \geq 2$ on a $\log_2(m) \geq 1$ et comme $k \geq 1$

On obtient $\boxed{k \leq \log_2(n)}$

ii) Écrire la fonction :

`test_puissance_entiere : int -> bool`

qui prend en entrée l'entier naturel $n > 1$ et renvoie le booléen `true` s'il existe deux entiers naturels k et m tous deux > 1 tels que $n = m^k$ et `false` sinon.

On proposera une version itérative et une version récursive.

Correction : on utilise la fonction précédente en faisant varier les puissances possibles k. On ne peut pas calculer $\log_2(n)$ il faut donc tester au fur et à mesure la puissance de 2 par rapport à n.

```
let test_puissance_entiere n=let res= ref false in
                             let p = ref 4 in
                             let k = ref 2 in
                             while !p <= n && not(!res)
```

```
do
   res:= test_puissance n !k;
   incr k ;
   p:= 2* !p
done ;
!res ;;
```

En récursif :

```
let test_puissance_entiere n =
     let rec aux k p = match p with
        | p when p > n -> false
        | _ -> test_puissance n k || aux (k+1) (2*p)
     in aux 2 4;;
```

Ici si le test `test_puissance n k` est vrai il n'a pas besoin de calculer `aux (k+1) (2*p)`, il n'y a donc pas d'appel récursif.

iii) En déduire la fonction :

```
liste1_puissances_entieres : int -> int list
```

qui prend en entrée l'entier naturel $n > 1$ et renvoie la liste des entiers naturels compris entre 2 et n qui sont des puissances entières.

Correction :

```
let rec liste1_puissances_entieres n=match n with
                                  |1->[]
                                  |_-> if test_puissance_entieres n
                                     then
                                     n::(liste1_puissances_entieres (n-1))
                                     else
                                     liste1_puissances_entieres (n-1) ;;
```

Exercice III : minimum local

Dans tout le problème on considère des tableaux de la forme $t = [|t_0; t_1; t_2; \ldots; t_{n-2}; t_{n-1}|]$ de longueur $n > 1$. On n'aura pas besoin de vérifier que les tableaux sont non vides. On cherche un minimum local de ce tableau, c'est-à-dire la valeur t_i avec i compris entre 0 et $n-1$ telle que t_i est inférieur ou égal à ses voisins ; on remarquera que t_i peut avoir 2 voisins mais n'en a qu'un seul si $i = 0$ ou $i = n-1$ ou même aucun si $n = 1$.

Exemples :

- t_1 et t_5 sont les minimums locaux de $[|14; 5; 8; 7; 4; 2|]$,
- t_1, t_2 et t_3 sont les minimums locaux de $[|14; 5; 5; 5; 8|]$,

On peut remarquer que le minimum d'un tableau est un minimum local.

1) Ecrire une fonction `minimum : 'a array -> 'a*int` qui reçoit un tableau pour paramètre et qui renvoie la valeur et l'indice d'un minimum (global).

Correction :

```
let minimum tab =
    let n = Array . length tab in
    let valmin = ref tab .(0) and indmin = ref 0 in
    for i = 1 to (n -1) do
       if tab .(i) < ! valmin
          then (valmin := tab .(i);indmin:=i)
    done ;
   (!valmin,indmin) ;;
```

2) Montrer que le premier minimum local d'un tableau `t` de longueur n est

- soit t_{n-1}
- soit t_i où i est le premier indice tel que $i < n-1$ et $t_i \leqslant t_{i+1}$.

Dans quel cas le premier minimum local vaut-il t_{n-1} ?

Correction : Si on a $t_0 \leqslant t_1$ alors t_0 est un minimum local, c'est le premier. Si on a $t_0 > t_1 > \cdots > t_i$ et $t_i \leqslant t_{i+1}$ alors t_i est un minimum local et t_k n'en est pas un pour $k < i$, t_i est le premier minimum local.

Si on a $t_0 > t_1 > \cdots > t_{n-1}$, le tableau est strictement décroissant et t_{n_1} est l'unique minimum local.

3) En déduire une fonction `premier_ml : 'a array -> 'a`, qui renvoie le premier minimum local d'un tableau.
Correction :

```
let premier_ml tab =
    let n = Array.length tab in
    let i = ref 0 in
    while !i < n -1 && tab.(!i) > tab.(!i +1) do i := !i + 1 done ;
    tab.(!i);;
```

4) Adapter la fonction précédente en une fonction `premier_ml_rec : ' list -> 'a` pour calculer un minimum local d'une liste.
Correction :

```
let rec premier_ml_rec lst = match lst with
         |[] -> failwith "La liste est vide , pas de minimum local "
         |[a] -> a
         |a::b::q when a <= b -> a
         |t::q -> premier_ml_rec q
```

5) k est un indice tel que $0 < k < n-1$ et t_k n'est pas un minimum local.
Prouver que :
- si $t_k > t_{k-1}$ alors un minimum local de $[|t_0; \ldots; t_{k-1}|]$ est un minimum local du tableau,
- si $t_k > t_{k+1}$ alors un minimum local de $[|t_{k+1}; \ldots; tn-1|]$ est un minimum local du tableau.

Correction : Un minimum local de $[|t_0; \ldots; t_{k-1}|]$ est un minimum local du tableau sauf s'il est atteint en t_{k-1}. Mais comme on a $t_{k-1} < t_k$, t_{k-1} reste un minimum local. De même pour l'autre cas.

6) En déduire une fonction `minimum_local : 'a array -> a` utilisant la méthode diviser pour régner qui calcule un minimum local d'un tableau avec une complexité logarithme (on ne demande pas de justifier la complexité).
Correction : On écrit une fonction auxiliaire qui reçoit les bornes dans lesquelles chercher un minimum local. Les cas terminaux sont ceux où il reste un seul élément ou deux car on va chercher au milieu et celui-ci ne doit pas être une des bornes.

```
let minimum_local tab =
         let rec aux_ml i j =
             if j > i + 1
             then let k = (i+j)/2 in
                  if tab .(k) > tab .(k -1)
                  then aux_ml i (k -1)
                  else if tab .(k) > tab .(k+1)
                       then aux_ml (k+1) j
                       else tab .(k)
             else if j = i + 1
                  then min tab .(j) tab .(i)
                  else tab .(i)
          in aux_ml 0 ( Array . length tab -1) ;;
```

La complexité est logarithmique car le tableau est coupé en 2 et on étudie qu'une seule partie du tableau.

Exercice IV : sélection du $(k+1)^e$ plus petit élément, CCINP 2023

La sélection du $(k+1)^e$ plus petit élément d'une liste d'entiers `L`, non nécessairement triée, consiste à trouver le $(k+1)^e$ élément de la liste obtenue en triant `L` dans l'ordre croissant.
Par exemple, si `L = [9; 1; 2; 4; 7; 8]` le 3^e plus petit élément de `L` est 4. On pourra remarquer que si la liste `L` est triée dans l'ordre croissant, le $(k+1)^e$ plus petit élément est l'élément de rang k dans `L`.

On présente un algorithme permettant de résoudre ce problème de sélection avec une complexité temporelle linéaire dans le pire cas. Celui-ci est basé sur le principe de « diviser pour régner »et sur le choix d'un bon pivot pour partager la liste en deux sous-listes.

Étant donné un réel a, on note $\lfloor a \rfloor$ le plus grand entier inférieur ou égal à a.

Fonctions utiles

Dans cette section, on écrit des fonctions auxiliaires qui sont utiles pour la fonction principale.

1) Ecrire une fonction récursive de signature : `longueur : 'a list -> int`, telle que `longueur lst` est la longueur de la liste `lst`.

Correction :

```
let rec longueur lst = match lst with
     | [] -> 0
     | t::q -> 1+ longueur q;;
```

2) Ecrire une fonction récursive de signature : `insertion : 'a list -> 'a -> 'a list`, telle que `insertion lst a` est la liste triée dans l'ordre croissant obtenue en ajoutant l'élément `a` dans la liste croissante `lst`.

Correction :

```
let rec insertion lst a = match lst with
         | [] -> [a]
         | t::q when t >= a -> a::lst
         | t::q -> t::(insertion q a);;
```

3) En déduire une fonction récursive de signature : `tri_insertion : 'a list -> 'a list`, telle que `tri_insertion lst` est la liste obtenue en triant `lst` dans l'ordre croissant.

Correction :

```
let tri_insertion lst = let rec aux lst res = match lst with
     | [] -> res
     | t::q ->aux q (insertion res t)
     in aux lst [];;
```

4) Écrire une fonction récursive de signature : `selection_n : 'a list -> int -> 'a`, telle que `selection_n lst n` est l'élément de rang n de la liste `lst`.

Par exemple, `selection_n [4;2;6;4;1;15] 3` est égal à 4.

Correction :

```
let rec selection_n lst i = match lst,i with
    | [],_ -> failwith "liste pas assez grande"
    | t::q,0 -> t
    | t::q,i -> selection_n q (i-1);;
```

5) Écrire une fonction récursive de signature : `paquets_de_cinq : 'a list -> 'a list list`, telle que `paquets_de_cinq lst` est une liste de listes obtenue en regroupant les éléments de la liste `lst` par paquets de cinq sauf éventuellement le dernier paquet qui est non vide et qui contient au plus cinq éléments.

Par exemple :
- `paquets_de_cinq []` est égal à `[]`,
- `paquets_de_cinq [2;1;2;1;3]` est égal à `[[2;1;2;1;3]]`,
- `paquets_de_cinq [3;4;2;1;5;6;3]` est égal à `[[3;4;2;1;5];[6;3]]`.

Correction : vouloir utiliser la longueur de la liste et la concaténation n'est pas une bonne idée en terme de complexité. Voici une version qui ne les utilise pas et qui ne parcourt qu'une fois la liste.

```
let paquet_de_cinq lst = let rec aux lst i = match(lst,i)with
                           | [],_ -> [],[]
                           | t::q, i when i<5 -> let l2,qq = aux q (i+1) in
                               t::l2,qq
                           | lst,5 -> let l2,qq = aux lst 0 in
                               [],l2::qq
                         in let rl,qr = aux lst 0 in rl::qr;;
```

Autre version : plus simple à mettre en place mais plus lourde dans l'écriture :

```
let rec paquet_de_cinq lst = match lst with
     | [] -> []
     | [a] -> [lst]
     | [a;b] -> [lst]
     | [a;b;c] -> [lst]
     | [a;b;c;d] -> [lst]
     | a::b::c::d::e::q ->[a;b;c;d;e]::( paquet_de_cinq q);;
```

ou encore

```
let rec paquet_de_cinq3 lst = match lst with
    | [] -> []
    | [_] | [_;_] | [_;_;_] | [_;_;_;_] -> [lst]
    | a::b::c::d::e::q ->[a;b;c;d;e]::( paquet_de_cinq3 q);;
```

6) Écrire une fonction récursive de signature : `medians : 'a list list -> 'a list`, telle que `medians lst` est la liste `m` obtenue en prenant dans chaque liste lst_k apparaissant dans la liste de listes `lst`, l'élément médian de lst_k.
On convient que pour une liste `A` dont les éléments sont exactement $a_0 \leqslant a_1 \leqslant \cdots \leqslant a_{n-1}$, l'élément médian désigne $a_{\lfloor \frac{n}{2} \rfloor}$.
Dans le cas où la liste `L` n'est pas triée, l'élément médian désigne l'élément médian de la liste obtenue en triant `L` par ordre croissant.
Par exemple : `medians [[3;1;5;3;2];[4;3;1];[1;3];[5;1;2;4]]` est égal à `[3;3;3;4]`

Correction : on va devoir trier chaque liste et regarder sa longueur afin de trouver le médian.

```
let rec medians lstlst = match lstlst with
   | [] -> []
   | lst::qlst -> let lst2 = tri_insertion lst in
                  let n = longueur lst in
                  let a = selection lst2 (n/2) in
                  a::(medians qlst);;
```

7) Écrire une fonction de signature : `partage : 'a -> 'a list -> 'a list * 'a list * int * int`, telle que `partage p lst` est un quadruplet (l_1, l_2, n_1, n_2) où l_1 est la liste des éléments de `lst` plus petit que `p`, l_2 est la liste des éléments de `lst` strictement plus grand que `p`, n_1 et n_2 sont respectivement les longueurs de l_1 et l_2.

Correction : il suffit de parcourir la liste avec des accumulateurs pou donner le quadruplet demandé. L'ordre n'étant pas obligatoirement respecté.

```
let partage p lst = let rec aux  lst (l1,l2,n1,n2) = match lst with
            | [] -> (l1,l2,n1,n2)
            | t::q when t<=p -> aux q (t::l1,l2,n1+1,n2)
            | t::q -> aux q (l1,t::l2,n1,n2+1)
            in aux lst ([],[],0,0);;
```

Si on veut respecter l'ordre on fait l'appel de la queue avant :

```
let partage p lst = let rec aux  lst  = match lst with
            | [] -> ([],[],0,0)
            | t::q when t<=p -> let (l1,l2,n1,n2) = aux q in (t::l1,l2,n1+1,n2)
            | t::q -> let (l1,l2,n1,n2) = aux q in (l1,t::l2,n1,n2+1)
            in aux lst ;;
```

La fonction de sélection et sa complexité

On détaille la fonction de sélection :

Algorithme 1 - Sélection du $(k+1)^e$ plus petit élément

```
1  SELECTION L k :
   /* L est une liste, k est un entier positif                          */
2  début
3      n ←LONGUEUR L
4      si n ≤ 5 alors
5          M ← (TRI_INSERTION L)
6          retourner l'élément de rang k de M
7      fin
8      sinon
9          L_Cinq ← PAQUETS_DE_CINQ L
10         M ← MEDIANS L_Cinq
11         pivot ← SELECTION M ((n+4)//5)//2
           /* L'opérateur // désigne le quotient d'entiers. Le rang ((n+4)//5)//2
              correspond au rang du médian de la liste M                 */
12         L1, L2, n1, n2 ← PARTAGE pivot L
13         si k < n1 alors
14             retourner SELECTION L1 k
15         fin
16         sinon
17             retourner SELECTION L2 (k − n1)
18         fin
19     fin
20 fin
```

8) Écrire une fonction récursive de signature : `selection : 'a list -> int -> 'a`, telle que `selection lst k` est le $(k+1)^e$ plus petit élément de la liste `lst`. L'écriture de la fonction sera une traduction en OCaml de l'Algorithme 1.

Correction : tout est dit dans la question

```
let rec selection lst k = let n = longueur lst in
       if n <=5 then (let lst2 = tri_insertion lst in selection_n lst2 k)
       else begin
         let lstlst = paquet_de_cinq lst in
         let lstm = medians lstlst in
         let pivot = selection lstm (((n+4)/5)/2) in
         let l1,l2,n1,n2 = partage pivot lst in
         if k<n1 then selection l1 k
                 else selection l2 (k-n1)
         end;;
```

On cherche à déterminer la complexité en nombre de comparaisons de la fonction `selection`. Pour tout $n \in \mathbb{N}$, on note $T(n)$ le nombre maximum de comparaisons entre éléments lors d'une sélection d'un élément quelconque dans des listes `L`, sans répétition, de taille n.

En analysant l'Algorithme 1, il est possible de démontrer que :

$$\forall n \geqslant 55, T(n) \leqslant T\left(\left\lfloor \frac{n+4}{5} \right\rfloor\right) + T\left(\left\lfloor \frac{8n}{11} \right\rfloor\right) + 4n \qquad (I)$$

9) En admettant la proposition (I), montrer que pour tout entier n supérieur à 1, on a :

$$T(n) \leqslant (200 + T(55))n$$

Pour l'initialisation, on pourra remarquer que T est une fonction croissante.

Correction : on écrit une récurrence sur n, en majorant les parties entières des fractions par les fractions, ce qui permet d'avoir $n+1$.

Exercice V : le tri fusion

L'objectif de cet exercice est d'étudier une implantation particulière d'un algorithme de tri d'une séquence d'entiers.

Définition 1 Une séquence s de taille n de valeurs v_i avec $1 \leqslant i \leqslant n$ est notée $\langle v_1, ..., v_n \rangle$. Une même valeur v peut figurer plusieurs fois dans s, nous noterons $card(v, s)$ le cardinal de v dans s, c'est-à-dire le nombre de fois que v figure dans s avec : $card(v, \langle v_1, ..., v_n \rangle) = card\{i \in \mathbb{N} \mid 1 \leqslant i \leqslant n,\ v = v_i\}$

Une séquence d'entiers est représentée par le type séquence et une paire de séquences d'entiers est représentée par le type paire dont les définitions sont :

```
type sequence = int list;;
type paire = sequence * sequence;;
```

Soit le programme en langage CaML :

```
let rec eclater s =
 match s with
  | [] -> ([],[])
  | [e] -> ([e],[])
  | e1::e2::q ->
     let (p1,p2) = eclater q in
       (e1::p1,e2::p2);;

let rec fusionner s1 s2 =
 match s1,s2 with
  | (e1::ql,e2::q2) ->
      if (e1 <= e2)
       then e1::(fusionner q1 s2)
       else e2::(fusionner s1 q2)
  | (_,_) -> sl @ S2;;

let rec trier s =
  match s with
    | [] -> []
    | [e] -> [e]
    | _ ->
       let (s1,s2) = eclater s in
         fusionner (trier s1) (trier s2);;
```

Soit la constante `exemple` définie et initialisée par :

```
let exemple = [4; 3; 2; 1] ;;
```

Question 1 Détailler les étapes du calcul de (`trier exemple`) en précisant pour chaque appel aux fonctions `eclater`, `fusionner` et `trier`, la valeur du paramètre et du résultat.

Correction : En utilisant la syntaxe : `->` pour appel de la fonction et `<-` pour le résultat obtenu :

```
trier [4;3;2;1]
  -> eclater [4;3;2;1]
        -> eclater [2;1]
               -> eclater []
               [],[] <-
        <- [2],[1]
  <- [4;2],[3,1]
  trier [4;2]
        -> eclater [4;2]
                -> eclater []
                <- [],[]
        <- [4],[2]
  fusionner [4] [2]
        -> 2::(fusionner [4] [])
              <- 2::[4]
        <- [2;4]
  trier [3;1]
```

```
        -> eclater [3;1]
                -> eclater []
                <- [],[]
        <- [3],[1]
  fusionner [3] [1]
        -> 1::(fusionner [3] [])
              <- 1::[3]
        <- [1;3]
  fusionner [2;4] [1;3]
        -> 1::(fusionner [2;4] [3])
               -> 1::2::(fusionner [4] [3]
                    -> 1::2::3::(fusionner [4] [])
                         -> 1::2::3::[4]
        <- [1;2;3;4]
<- [1;2;3;4]
```

Question 2 Soit la séquence `p` $= \langle p_1, ..., p_n \rangle$ de taille n, soient les séquences `a` $= \langle a_1, ..., a_r \rangle$ de taille r et `b` $= \langle b_1, ..., b_s \rangle$ de taille s telles que `(a,b) = (eclater p)`, montrer que :

$$\begin{array}{l} \forall i,\ 1 \leqslant i \leqslant n,\ card(p_i, p) = card(p_i, a) + card(p_i, b) \\ s = E(n/2) \\ r = n - E(n/2) \end{array}$$

Correction : Montrons le résultat sur n la longueur de la séquence p. On considère l'hypothèse de récurrence :

H_n : Pour toute séquence $\langle p_1, \ldots, p_n \rangle$, si `(a,b) = eclater de p` avec $a = \langle a_1, \ldots, a_r \rangle$ et $b = \langle b_1, \ldots, b_s \rangle$ on a :

$$\begin{array}{l} \forall i,\ 1 \leqslant i \leqslant n,\ card(p_i, p) = card(p_i, a) + card(p_i, b) \\ s = E(n/2) \\ r = n - E(n/2) \end{array}$$

Pour $n = 0$, on a nécessairement $a = b = \langle\rangle$, $s = 0 = r$. Donc H_0 est vraie.

Pour $n = 1$, par définition de la fonction `eclater`, avec $p = \langle x \rangle$, on a $a = \langle x \rangle$ et $b = \langle\rangle$.

D'où, $s = 0 = E(1/2)$ et $r = 1 = 1 - s$. De plus $card(x, r) = 1 = 1 + 0 = card(x, a) + car(x, b)$. Donc H_1 est vraie.

$H_n \Rightarrow H_{n+1}$: Soit $p = \langle p_1, \ldots, p_n, p_{n+1} \rangle$. Par définition de la fonction `eclater`, on a $(a, b) = (p_1 :: a', p_2 :: b') =$ `eclater`(p) où $(a', b') =$ `eclater`$(\langle p_3, \ldots, p_{n+1} \rangle)$.

D'après H_n, $s' = |a'| = E((n-1)/2)$, $r' = |b'| = (n-1) - E((n-1)/2)$ d'où, $s = 1 + s' = 1 + E((n-1)/2) = E((n+1)/2)$ et $r = 1 + r' = n - E((n-1)/2) = n + 1 - \underbrace{\left[E(\frac{n-1)}{2}) + 1\right]}_{E(\frac{n+1}{2})}$.

$\forall i \in [\![1, n+1]\!]$,

$$card(p_i, p) = \begin{cases} card(p_i, a') + card(p_i, b') = card(p_i, a) + card(p_i, b) & \text{si } p_i \neq a_1 \text{ et } p_i \neq b_1 \\ 2 + card(p_i, a') + card(p_i, b') = card(p_i, a) + card(p_i, b) & \text{si } p_i = a_1 = b_1 \\ 1 + card(p_i, a') + card(p_i, b') = card(p_i, a) + card(p_i, b) & \text{si } p_i = a_1 \text{ et } p_i \neq b_1 \\ card(p_i, a') + card(p_i, b') + 1 = card(p_i, a) + card(p_i, b) & \text{si } p_i = b_1 \text{ et } p_i \neq a_1 \end{cases}$$

On a bien prouvé H_{n+1}.

conclusion : H_0 étant vraie et $\forall n \in \mathbb{N},\ H_n \Rightarrow H_{n+1}$, la propriété est vraie à tout ordre.

Question 3 Soient les séquences `a` $= \langle a_1, ..., a_m \rangle$ de taille m et `b` $= \langle b_1, ..., b_n \rangle$ de taille n, soit la séquence `r` $= \langle r_1, ..., r_l \rangle$ de taille l telle que `r = (fusionner a b)`, montrer que :

$$\begin{array}{l} l = m + n \\ \forall k,\ 1 \leqslant k \leqslant l,\ card(r_k, r) = card(r_k, a) + card(r_k, b) \\ \left\{ \begin{array}{l} \forall i,\ 1 \leqslant i < m,\ a_i \leqslant a_{i+1} \\ \forall j,\ 1 \leqslant j < n,\ b_j \leqslant b_{j+1} \end{array} \right. \Rightarrow \forall k,\ 1 \leqslant k < l,\ r_k \leqslant r_{k+1} \end{array}$$

Correction : On fait une récurrence sur $p = m + n$.

Si $m + n = 0$ alors $a = \langle\rangle = b = r$ d'où $l = 0 = 0+0 = m+n$ et $\forall k \in \varnothing, card(r_k, r) = card(r_k, a) + card(r_k, b)$ et $\forall k \in \varnothing, r_k \leqslant r_{k+1}$. On a donc H_0

$H_p \Rightarrow H_{p+1}$? Supposons H_p considérons deux séquences `a` $= \langle a_1, ..., a_m \rangle$ de taille m et `b` $= \langle b_1, ..., b_n \rangle$ de taille n telles que $m + n = p + 1$.

– si $a = \langle\rangle$ alors $r = b$, $m = 0$, $n = p + 1$ donc $l = k + 1 = m + n$,
$\forall k \in [\![1, l]\!]$, $card(r_k, r) = \underbrace{card(r_k, a)}_{=0} + card(r_k, b)$, et $\forall k \in [\![1, p]\!]\ r_k = b_k \leqslant b_{k+1} = r_{k+1}$.

– Si $b = \langle\rangle$ alors le cas est identique au cas précédent.

– Si $a \neq \langle\rangle$ et $b \neq \langle\rangle$ alors $\mathtt{a} = \langle a_1, ..., a_m\rangle$ de taille m et $\mathtt{b} = \langle b_1, ..., b_n\rangle$

  – si $a_1 \leqslant b_1$ alors avec $\mathtt{a'} = \langle a_2, ..., a_m\rangle$, $r_1 = a_1$ et $r' = \langle r_2, ..., r_l\rangle =$ `fusionner` a' b.
  D'après l'hypothèse de récurrence H_p, on a $l - 1 = m - 1 + n \Rightarrow l = m + n$.
  $card(r_k, r') = card(r_k, a') + card(r_k, b) = \begin{cases} card(r_k, a) + card(r_k, b) & \text{si } r_k \neq a_1 \\ card(r_k, a) - 1 + card(r_k, b) & \text{si } r_k = a_1 \end{cases}$
  $\Rightarrow card(r_k, a) = card(r_k, a) + card(r_k, b)$. $\forall k \in [\![2, l - 1]\!]$, $r_k \leqslant r_{k+1}$ et comme $r_1 = a_1$ est plus petit que tout élément de a et de b, on a $\forall k \in [\![1, l - 1]\!]$, $r_k \leqslant r_{k+1}$.

  – si $a_1 > b_1$ alors avec $\mathtt{b'} = \langle b_2, ..., b_n\rangle$, $r_1 = b_1$ et $r' = \langle r_2, ..., r_l\rangle =$ `fusionner` a b'.
  On procède comme dans le cas précédent

conclusion : H_0 étant vraie et $\forall p \in \mathbb{N}$, $H_p \Rightarrow H_{p+1}$, la propriété est vraie à tout ordre.

Question 4 Soit la séquence $\mathtt{p} = \langle p_1, ..., p_m\rangle$ de taille m, soit la séquence $\mathtt{r} = \langle r_1, ..., r_n\rangle$ telle que `r = (trier p)`, montrer que :

$$\begin{array}{l} m = n \\ \forall i,\ 1 \leqslant i \leqslant m,\ card(p_i, r) = card(p_i, p) \\ \forall i,\ 1 \leqslant i < n,\ r_i \leqslant r_{i+1} \end{array}$$

Correction : Montrons la propriété sur m.

Si $m = 0$, alors, par définition de la fonction `trier`, $p = r = \langle\rangle$, d'où $m = n = 0$ et $\forall i \in \varnothing,\ card(p_k, r) = 0 = card(p_i, p),\ r_i \leqslant r_{i+1}$, d'où H_0.

Si $m = 1$, alors, par définition de la fonction `trier`, $p = r = \langle p_1\rangle$, d'où $m = n = 1$ et $\forall i \in \{1\},\ card(p_i, r) = 0, = card(p_i, p),\ \forall i \in \varnothing,\ r_i \leqslant r_{i+1}$, d'où H_1.

Si $m \geqslant 2$, on suppose H_k pour $k \in [\![0, m - 1]\!]$. Soit la séquence $\mathtt{p} = \langle p_1, ..., p_m\rangle$

On commence par calculer $(s1, s2) =$ `eclater` p. D'après la question III.2, si $s_1 = \langle a_1, ..., a_r\rangle$ et $s_2 = \langle b_1, ..., b_s\rangle$, on a

$$\begin{array}{l} \forall k,\ 1 \leqslant k \leqslant l,\ card(p_k, p) = card(p_k, s_1) + card(p_k, s_2) \\ s = E(m/2) \\ r = m - E(m/2) \end{array}$$

Avec l'hypothèse de récurrence, les séquences $s'_1 =$ `trier` $s_1 = \langle a'_1, ..., a'_{r'}\rangle$ et $s'_2 =$ `trier` $s_2 = \langle b'_1, ..., b'_{s'}\rangle$ vérifient

$$\begin{array}{l} r' = r,\ s' = s, \\ \forall i,\ 1 \leqslant i \leqslant r,\ card(a'_i, s'_1) = card(a'_i, s_1) \\ \forall i,\ 1 \leqslant i < r,\ a'_i \leqslant a'_{i+1} \\ \forall j,\ 1 \leqslant j \leqslant s,\ card(b'_j, s'_2) = card(b'_j, s_2) \\ \forall j,\ 1 \leqslant j < r,\ b'_j \leqslant b'_{j+1} \end{array}$$

La séquence $\mathtt{r} = \langle r_1, ..., r_n\rangle =$ `trier` $p =$ `fusionner` s'_1 s'_2 vérifie d'après la question III.3

$$\begin{array}{l} l = r' + s' = r + s = m \\ \forall k,\ 1 \leqslant k \leqslant n,\ card(r_k, r) = card(r_k, s'_1) + card(r_k, s'_2) \\ \qquad\qquad\qquad\qquad = card(r_k, s_1) + card(r_k, s_2) = card(r_k, p) \\ \forall k,\ 1 \leqslant k < n,\ r_k \leqslant r_{k+1} \end{array}$$

conclusion : H_0 étant vraie et $\forall m \in \mathbb{N}$, $(\forall k \in [\![1, m]\!],\ H_k) \Rightarrow H_{m+1}$, la propriété est vraie à tout ordre.

Question 5 Montrer que le calcul des fonctions `eclater`, `fusionner` et `trier` se termine quelles que soient les valeurs de leurs paramètres respectant le type des fonctions.

Correction : `eclater s` s'arrête si la longueur de la séquence est inférieure ou égale à 1 ou bien s'appelle elle-même avec comme argument un séquence strictement plus petite que la séquence de départ. La suite des longueurs des séquences en argument de la fonction `eclater` ne pouvant être strictement décroissante, finie par atteindre la valeur 0 ou 1, donc

conclusion : le calcul de la fonction `eclater` se termine .

La fonction `fusionner s1 s2` s'arrête si la longueur de l'une des séquences en argument est nulle ou bien s'appelle elle-même avec comme arguments deux séquences strictement plus petites que les séquences de départ. La suite des longueurs des séquences en argument de la fonction `fusionner` ne pouvant être strictement décroissante, finie par atteindre la valeur 0, donc

conclusion : le calcul de la fonction `fusionner` se termine .

La fonction `trier s` s'arrête si la longueur de la séquence est inférieure ou égale à 1 ou bien elle fait appel aux fonctions `eclater` et `fusionner` dont le calcul se termine toujours et s'appelle elle-même avec comme argument un séquence strictement plus petite que la séquence de départ. La suite des longueurs des séquences en argument de la fonction `trier` ne pouvant être strictement décroissante, finie par atteindre la valeur 0 ou 1, donc

conclusion : le calcul de la fonction `trier` se termine .

Question 6 Donner des exemples de valeurs du paramètre `s` de la fonction `trier` qui correspondent aux meilleur et pire cas en nombre d'appels récursifs effectués.

Calculer une estimation de la complexité dans les meilleur et pire cas des fonctions `eclater`, `fusionner` et `trier` en fonction du nombre de valeurs dans les séquences données en paramètre. Cette estimation ne prendra en compte que le nombre d'appels récursifs effectués.

Correction : Quelque soit la longueur de la liste, le nombre d'appels de la fonction `eclater` et de l'ordre $\frac{n}{2}$ où n est la longueur de la liste. On a donc une complexité en $O(n)$.

Pour la fonction `fusionner`, le cas favorable est celui où l'une des deux listes est vide (exemple : `fusionner` $[x_1; ...; x_n][]$). Il y a un seul appel à la fonction `fusionner`. Dans le cas défavorable, il faut parcourir les deux listes passées en argument pour arriver au résultat (exemple : `fusionner [0;n+1] [1;2;...;n]`). Le nombre d'appels à la fonction `fusionner` est en $O(n_1+n_2)$ où n_1 et n_2 sont les longueurs des listes passées en paramètres de la fonction.

Pour la fonction `trier`, la complexité est en $O(n \ln n)$ dans tous les cas.