Corrigé DS1, sujet CCP 2012-Mines 1999

Sujet CCP 2012
Question III. 1

```
let rec ajoutSequence v s = match s with
    | [] -> [v]
    | x::t -> if v<=x then v::s else x::(ajoutSequence v t);;
```

Question III. 2
Dans le meilleur des cas, l'élément `v` est plus petit que tous ceux de la liste ; il n'y a dans ce cas aucun appel récursif, autrement dit un seul appel à la fonction `ajoutSequence`. Dans le pire des cas, on ajoute la liste à la fin de la liste ; dans ce cas il y a autant d'appels récursifs que d'éléments dans la liste initiale. En moyenne, on peut estimer qu'il y aura $n/2$ appels récursifs, où n est la longueur de la liste. En tout cas, le nombre d'appels récursifs est $O(n)$ où n est la longueur de la liste.
Question III. 3

```
let scissionSequence s =
    let rec aux p s i=
        if i=0 then let v::s1=s in p,v,s1
        else let v::s1=s in aux (p@[v] s1 (i+1))
    in aux [] s (List.length s /2);;
```

Question III. 4
Dans l'appel "eliminer 2 exemple", la fonction va effectuer l'instruction "rg=eliminer 2 Noeud(Noeud(Vide,1,Vide),2,Noeud(Vide,3,Vide))". Ce deuxième appel va effectuer ensuite "rg=eliminer 2 Noeud(Vide,1,Vide)". Ce troisième appel va renvoyer "Noeud(Vide, 1, eliminer 2 Vide)" ; l'appel "eliminer 2 Vide" renvoie "Vide" donc le troisième appel renvoie "Noeud(Vide,1,Vide)". Le deuxième appel effectue donc "rg=Noeud(Vide,1,Vide)" puis "(vm,gp)=aux rg" ; cet appel à "aux" renvoie "(1,Vide)". On a donc "(vm,gp)=(1,Vide)". Le 2ème appel renvoie donc "Noeud(Vide,1,Vide)". Le premier appel effectue donc "(vm,gp)=aux Noeud(Vide,1,Vide)" ; "aux Noeud(Vide,1,Vide)" renvoyant "(1,Vide)", le premier appel renvoie "Noeud(Vide,1,Noeud(Vide,3,Vide))".
Question III. 5
La fonction "aux" renvoie le maximum de l'arbre binaire de recherche (non vide) en argument et le sous-arbre amputé de l'étiquette maximale.
Montrons le par récurrence sur la profondeur du sous-arbre droit de l'arbre a en argument de la fonction.
Si le sous-arbre droit est vide, par définition d'un arbre binaire de recherche, le maximum est à la racine. Le renvoi "(v,g)" est bien le résultat demandé.
Supposons que le résultat soit valable pour tout sous-arbre droit de profondeur au plus n. Supposons que a=Noeud(g,v,d) soit tel que la profondeur de d est n+1. Alors le sous-arbre droit de d est au plus n donc "aux d" renvoie le maximum "vd" de d et le sous-arbre "ar" de d amputé de ce maximum. Par définition d'un arbre binaire de recherche, le maximum de a est celui de d et l'arbre Noeud(g,v, ar) est l'arbre a amputé du maximum de a.
Par récurrence, aux a renvoie le maximum de a et a amputé de ce maximum (pour a un arbre binaire de recherche non vide).
Passons au résultat demandé par l'énoncé. Procédons par induction sur l'ensemble des arbre.
Supposons que a soit vide. Alors la fonction "eliminer" renvoie "r=a" qui est un arbre binaire de recherche donc ABR(r). De plus, l'ensemble des étiquettes de a et de r sont vides donc $\mathcal{C}(r) = \mathcal{C}(a)\backslash\{v\}$.
Supposons que a=Noeud(g,vp,d) et que "rg=eliminer g v", ou "rd=eliminer d v" vérifie les conditions "$ABR(rg) \wedge \mathcal{C}(rg) = \mathcal{C}(g)\backslash\{v\}$" et "$ABR(rd) \wedge \mathcal{C}(rd) = \mathcal{C}(d)\backslash\{v\}$".
Si v<vp, alors r=Noeud(rg,vp,d) avec ABR(rg) par hypothèse d'induction, ABR(d), pour tout r dans $\mathcal{C}(rg)$, r est dans $\mathcal{C}\backslash\{v\}$ donc r≤vp ; de plus, pour tout r dans $\mathcal{C}(d)$, vp<r. Enfin, $\mathcal{C}(r) = \mathcal{C}(rg) \cup \{vp\} \cup \mathcal{C}(d) = \mathcal{C}(g)\backslash\{v\} \cup \{vp\}\mathcal{C}(d) = \mathcal{C}(a)\backslash\{v\}$.
Si v>vp, de même, r=Noeud(g,vp,rd) avec ABR(g), ABR(rd) par hypothèse d'induction et, pour tout r dans $\mathcal{C}(g)$, r≤vp ; pour tout r dans $\mathcal{C}(rd)$, r est dans $\mathcal{C}(d)$ donc r>vp. Enfin, $\mathcal{C}(r) = \mathcal{C}(g) \cup \{vp\}\mathcal{C}(rd) = \mathcal{C}(g) \cup \{vp\} \cup \mathcal{C}(d)\backslash\{v\} = \mathcal{C}(a)\backslash\{v\}$.
Enfin, supposons que vp=v. Par hypothèse d'induction, rg=eliminer v g renvoie un arbre rg tel que ABR(rg) et $\mathcal{C}(rg) = \mathcal{C}(g)\backslash\{v\}$.
Si rg=Vide, alors l'arbre d vérifie ABR(d) et $\mathcal{C}(d) = \mathcal{C}(a)\backslash(\{v\} \cup \mathcal{C}(g))$; comme rg=Vide, $\mathcal{C}(g) = \{v\}$ donc $\mathcal{C}(d) = \mathcal{C}(a)\backslash\{v\}$. Le résultat est donc vérifier lorsque rg=Vide.
Sinon, l'instruction (vm,gp)=aux rg fait en sorte que vm est le maximum de rg et gp est l'arbre privé de ce maximum. Ainsi, r=Noeud(gp,vm,d) a pour étiquettes $\mathcal{C}(r) = \mathcal{C}(gd) \cup \{vm\} \cup \mathcal{C}(d) = \mathcal{C}(rg) \cup \mathcal{C}(d) = (\mathcal{C}(g)\backslash\{v\}) \cup \mathcal{C}(d) = \mathcal{C}(a)\backslash\{v\}$. Enfin, on a bien ABR(gp), ABR(d) et, comme vm est le maximum de rg, pour toute étiquette r de gp, r≤vm et pour toute étiquette r de d, r>v≥vm. Ainsi, ABR(r).
Ceci démontre que $ABR(a) \Rightarrow ABR(r) \wedge \mathcal{C}(r) = \mathcal{C}(a)\backslash\{v\}$ par induction.
Question III. 6
La fonction "aux" se termine pour un appel sur un sous-arbre droit vide, c'est-à-dire de profondeur strictement négative ; sinon, elle effectue un appel récursif sur un sous-arbre de hauteur strictement inférieure donc sur un sous-arbre dont le sous-arbre droit est de profondeur strictement inférieure à la profondeur du sous-arbre droit de l'appel initial. Par conséquent, les appels récursifs successifs font décroître strictement la profondeur du sous-arbre droit jusqu'à un sous-arbre droit de profondeur strictement négative, ce qui termine les appels successifs.
La fonction "eliminer" se termine pour un arbre vide. Sur un arbre non vide, elle effectue un appel à "eliminer" sur un arbre dqui peut-être le sous-arbre gauche ou le sous-arbre droit, qui est donc de profondeur strictement inférieure. La fonction "eliminer" se termine

donc également.

Question III. 7

Pour un arbre a qui possède une valeur à éliminer à un noeud de profondeur maximale, le nombre d'appels récursifs est maximal, à savoir la profondeur de l'arbre. Dans le meilleur des cas, la valeur à éliminer est celle de la racine de l'arbre et le sous-arbre gauche n'a pas l'étiquette v alors que le sous-arbre droit de ce sous-arbre gauche est vide. Dans ce cas, le nombre d'appels récursifs de aux et de eliminer est bornée.

Question III. 8

Dans le meilleur des cas, il y a deux appels récursifs (si la racine du l'arbre est l'étiquette à éliminer). Dans le pire des cas, on parcourt l'arbre sur toute sa profondeur, qui peut être le nombre de valeurs si l'arbre est un arbre "peigne". Dans ce cas, on peut avoir $n+1$ appels à "eliminer" et n appels à "aux", si n est la profondeur.

Question III. 9

```
let estComplet a = match a with
    | Feuille(s) -> 2*ordre=List.length s +1
    | Noeud(p,n) -> 2*ordre=List.length s +1;;
```

Question III. 10

```
let taille a = match a with
     | Feuille(s) -> List.length s
     | Noeud(a1,f) -> taille a1 + tailleFreres f
 and tailleFreres f = match f with
        | [] -> 0
        | (i,a2)::l0 -> 1+taille a2 + tailleFreres l0::
```

Quesiton III. 11

```
let rec rechercheBArbre i a = match a with
    | Feuille(s) -> rechercheListe i s
    | Noeud(a1,l) -> rechercheBArbre i a1 || rechercheFreres i l
and rechercheListe i l = match l with
    | [] -> false
    | x::l0 when x=i -> true
    | x::l0 when x<i -> false
    | x::l0 -> rechercheListe i l0
and rechercheFreres i l = match l with
    | [] -> false
    | (j,a2)::l0 when i=j -> true
    | (j,a2)::l0 when i<j -> false
    | (j,a2)::l0 -> rechercheBArbre i a2 || rechercheFreres i l0;;
```

Question III. 12

```
let rec scissionBArbre a = match a with
    | Feuille(s) -> let (s1,v,s2)=scissionSequence s in (Feuille s1 , v, Feuille s2)
    | Noeud(a1, freres) -> let (f1,(v,a2),f2)=scissionSequence freres in (Noeud(a1,f1), v, Noeud(a2,f2));;
```

Question III. 13

Le premier noeud est complet, on fait donc une scission pour obtenir 12 comme racine, le sous-arbre gauche ayant 9 comme racine, le droit 15. Le sous-arbre suivant à considérer est celui de racine 7 qui n'est pas complet. Le sous-arbre suivant à considérer est la feuille `Feuille([2;4;6])` qui est complète. On la scinde donc en `Noeud(Feuille([2]),4,Feuille([6]))`. On ajoute ensuite l'étiquette 3 à la suite de 2. Cela donne l'arbre

```
Noeud(
    Noeud(
        Noeud(
            Noeud(
                Feuille([2;3]),
                [ ( 4, Feuille( [6] ) ) ]
            ),
            [ ( 7,Feuille( [8] ) )]
        ),
        [ (9, Feuille( [10] ) )]
    ),
    [ (12, Noeud(
        Feuille( [13] )
        [ (15, Feuille( [17] ))]
    ) ) ]
    );;
```

Question III. 14

Si le B-arbre est une feuille, alors on ajoute un élément à une séquence croissante ou on effectue la scission d'un arbre, ce qui est un algorithme qui se termine.

Sinon, le B-arbre est constitué d'une séquence et d'une suite d'arbre. Si la racine n'est pas complète, on effectue un appel récursif sur un B-arbre de hauteur strictement inférieure. Sinon(si la racine est complète), il y a scission du B-arbre (qui se termine) puis application

de l'algorithme au nouveau B-arbre ce qui crée un appel récursif sur un B-arbre de hauteur égale au B-arbre initiale, mais désormais non complet ; on est ramené au premier cas, celui d'un B-arbre de racine non complète, ce qui crée un appel récursif sur un B-arbre de hauteur strictement inférieure.
Ainsi, par récurrence sur la hauteur du B-arbre, l'algorithme se termine bien.

Question III. 15

Si les noeuds et feuilles traversés du B-arbre sont tous non complets, l'algorithme maintient pour chaque noeud la structure de B-arbre. Pour la feuille à laquelle est ajoutée l'élément, on garde une séquence croissante ; sa taille est désormais $p+1$ avec $p+1 < 2n$ (comme la feuille initiale est non complète) donc $(p+1)+1 \leq 2n$ donc cette feuille est un B-arbre. Le B-arbre obtenu est bien un B-arbre.
Sinon, pour chaque noeud incomplet ou feuille incomplète traversé, on effectue une scission, opération qui maintient une structure de B-arbre, ce qui nous ramène à un B-arbre du type précédent.

Question III. 16

```
let rec ajouter v a = match a with
    | Feuille(s) -> if not(estComplet a) then Feuille( ajouterSequence v s )
                    else ajouter v (scissionBArbre a)
    | Noeud( a1, freres) -> if not(estComplet a)
                           then let (i,a1)::freres0=freres in
                                if v<i then Noeud( ajouter v a1 , freres)
                                else Noeud( a1, ajouter v freres)
                           else ajouter v (scissionBArbre a)
and ajouterFreres v freres = match freres with
    | (i,a2):[] -> Noeud( a1, [(i, ajouter v a2)])
    | (i,a2):frere2 -> let (j,a3)::frere3=frere2 in
                       if v<j then (i, ajouter v a2)::frere2
                       else (i,a2)::(ajouterFreres v freres2);;
```

Sujet Mines-Ponts 1999

1. Supposons que a est un arbre binaire de hauteur 0. Alors a est réduit à une feuille donc son nombre maximal de feuilles est $1 = 2^0$. Soit $h \in \mathbb{N}$. Supposons que, pour tout arbre binaire de hauteur $k \leq h$, le nombre maximal de feuilles de cet arbre est 2^k. Soit un arbre binaire de hauteur $h+1$. Alors le nombre de ses feuilles est la somme des nombres de feuilles de ses sous-arbres droit et gauche. Ses sous-arbres droit et gauche ont hauteur au plus h donc leur nombre maximal de feuilles est 2^h. Le nombre maximal de feuilles de l'arbre binaire de hauteur $h+1$ est donc $2^h + 2^h = 2^{h+1}$.
Par récurrence, le nombre maximal de feuilles d'un arbre binaire de hauteur h est 2^h.

2. a. Soit h la hauteur d'un arbre de décision. Il y a au moins une feuille pour chaque élément de C donc au moins n feuilles. D'après la question précédente, $n \leq 2^h$ donc $\log_2(n) \leq h$ donc $\lceil \log_2(n) \rceil \leq h$.

b. Montrons que la hauteur maximale d'un arbre de décision de n est $n-1$. Si $n = 1$, un arbre de décision de n a une seule feuille donc a hauteur $0 = 1 - 1$. Supposons que tout arbre de décision de n a hauteur maximale $n-1$. Considérons un arbre de décision de n de hauteur $n-1$. Considérons l'arbre de décision de $n+1$ ayant pour sous-arbre gauche l'arbre de décision précédent de n ayant hauteur $n-1$ et pour sous-arbre droit une feuille, celle correspondant à l'élément n'appartenant pas à l'ensemble des éléments de l'ensemble du sous-arbre gauche précédent. Alors l'arbre de décision considéré de $n+1$ a hauteur $1 + n - 1 = n = n + 1 - 1$.
Par récurrence, la hauteur maximale d'une arbre de décision de n est $n-1$.

3. Le nombre de bijections de $[\![1, n]\!]$ est $n!$ donc le nombre de permutations de t est $n!$.

4. Cette hauteur minimale est $\lceil \log_2(n!) \rceil \sim \log_2(n!)$. Or $\log_2(n!) = \sum_{k=1}^{n} \log_2(k)$. Par comparaison série/intégrale, $\sum_{k=1}^{n} \log_2(k) \sim \int_1^n \log_2(t) = n\log_2(n) - \frac{n-1}{\ln(2)} \sim n\log_2(n)$. La hauteur minimale d'un arbre de décision correspondant au choix d'une permutation d'un tableau de taille n d'éléments deux à deux distincts parmi les $n!$ possibles est donc équivalente à $n\log_2(n)$.

5. Tout algorithme de tri d'un tableau de n éléments distincts aura hauteur minimale équivalente à $n\log_2(n)$. Dans le pire des cas, pour parvenir à une feuille de profondeur h, il faudra donc effectuer $h \sim n\log_2(n)$ choix successifs. Le nombre de choix au pire sera donc au moins en $O(n\log_2(n)) = O(n\ln(n))$.

6. On utilise un tableau d'entiers donc le type `int array`. On accède à la valeur d'indice $i \in [\![1, n]\!]$ par l'exécution de `t.(i-1)` ; si $i \in [\![0, n-1]\!]$, on accède à la valeur d'indice i par l'appel `t.(i)`. La fonction "echanger" pourra être

```
let echanger t i j = let x=t.(i) in t.(i)<-t.(j); t.(j)<-x;;
```

Cet algorithme effectue deux affectations et une création de variable. Il est donc à temps constant borné.

7. Cet arbre a racine 23, fils gauche 12, fils droit 3. 12 a fils gauche 4 et fils droit 67. 3 a fils gauche 10 et fils droit 9. 4 a fils gauche 6 et fils droit 7. 67 a fils gauche 12. Les autres noeuds sont des feuilles.

8.a.

```
let ordonner t i j =
    if 2*(i+1)=j+1 && t.(i)<t.(j) then echanger t i j;
    if 2*(i+1)<j+1 then begin
        if t.(2*i+1)<t.(2*i+2) && t.(i)<t.(2*i+2) then
            begin
            echanger t i (2*i+2);
            ordonner t (2*i+2) j end;
        if t.(2*i+1)>=t.(2*i+2) && t.(i)<t.(2i+1) then
           begin
           echanger t i (2*i+1);
           ordonner t (2*i+1) j end;
      end;;
```

b. Cet algorithme sur un couple (i,j) contient un nombre borné α d'opération pour $i > j/2$. Pour $i \leq j/2$, cet algorithme effectue un nombre borné β d'opérations et un appel récursif sur le couple $(2i,j)$ ou $(2i+1,j)$. Cette complexité $C_{i,j}$ vérifie donc : si $2i \leq j$, $C_{i,j} \leq \max(C_{2i,j}, C_{2i+1,j}) + \beta$, si $2i > j$, $C_{i,j} = \alpha$.
Montrons par récurrence que $C_{i,j} \leq \alpha + \beta\lceil \log_2(\frac{j}{j})\rceil$. Si $2i > j$, $C_{i,j} = \alpha$ donc la propriété est vraie. Supposons que, si $\lceil \log_2(\frac{j}{i})\rceil \leq k$, la propriété est vraie. Supposons que $\lceil \log_2(\frac{j}{i})\rceil = k+1$. Alors $j \leq 2^{k+1}i$ et $2i \leq j$ donc $C_{i,j} \leq \max(C_{2i,j}, C_{2i+1,j}) + \beta$; comme $\frac{j}{2i+1} < \frac{j}{2i} \leq 2^k$, par hypothèse de récurrence, $C_{2i,j} \leq \alpha + \beta k$ et $C_{2i,j+1} \leq \alpha + \beta k$ donc $C_{i,j} \leq \max(\alpha+\beta k, \alpha+\beta k) + \beta = \alpha + \beta k + \beta = \alpha + \beta(k+1)$.
On a donc, pour tout i,j, $C_{i,j} \leq \alpha + \beta\lceil \log_2(\frac{j}{i})\rceil = O(\log_2(\frac{j}{i}))$.
9. Montrons cette propriété par récurrence sur j.
Si $j = 1$, les éléments de 1 à 1 sont réduits au premier élément du tableau ; l'arbre $\alpha(t,1,1)$ est une feuille réduite à la racine dont l'étique est le premier élément tableau. Ainsi, tout élément du tableau entre 1 et 1 est bien une feuille de $\alpha(t,1,1)$.
Supposons que les éléments de 1 à j sont des noeuds ou des feuilles de $\alpha(t,1,j)$. Les noeuds et feuilles de $\alpha(t,1,j)$ sont inclus dans les noeuds de $\alpha(t,1,j+1)$.
Soit i le quotient de la division euclidienne de $j+1$ par 2 : $j+1 = 2i$ ou $j+1 = 2i+1$. Si $j+1 = 2i+1$, $j+1$ est la racine du fils droit de $\alpha(t,i,j)$. Comme i est une feuille ou un noeud de $\alpha(t,1,j)$, c'en est un de $\alpha(t,1,j+1)$ donc $j+1 = 2i$ est aussi une feuille ou un noeud de $\alpha(t,1,j+1)$. De même, si $j+1 = 2i$, $j+1$ est la racine du fils gauche de $\alpha(t,i,j)$; i étant un noeud ou une feuille de $\alpha(t,1,j)$ donc de $\alpha(t,1,j+1)$, $j+1$ est une feuille ou un noeud de $\alpha(t,1,j+1)$.
Par récurrence, toute élément d'indice 1 à j de t est une feuille ou un noeud de $\alpha(t,1,j)$.
10.a.

```
let ordonner_totalement t =
    let n=Array.length t in
    for j=(n-1)/2 downto 0 do
        ordonner t j n-1 done;;
```

b. Montrons par récurrence que, pour tout j, à l'entrée de la boucle j tous les arbres $\alpha(t,k,n)$ sont ordonnés pour $2k > j+1$.
Initialement, $2j = n-1$ ou $2j+1 = n-1$ donc $2(j+1) = n$ ou $n+1$. Si $2k > j+1$, $2k > (j+1) \geq n$ donc $\alpha(t,j,n)$ est réduit à une feuille ; il est donc ordonné.
Supposons qu'à l'entrée de la boucle j, les arbres $\alpha(t,k,n)$ sont ordonnés pour $2k > j+1$. Supposons que $2k = j+1$. Alors l'appel `ordonner t j n-1` se fait sur des arbres $\alpha(t,2(j+1),n)$ et $\alpha(t,2(j+1)+1,n)$ ordonnés donc, en supposant que la fonction `ordonner` est correcte, l'arbre $\alpha(t,j+1,n)$ qui en ressort est ordonné. Par conséquent, les sous-arbres $\alpha(t,k,n)$ pour $k > j-1+1$ sont ordonnés à la sortie de la boucle d'indice j donc à l'entrée de la boucle d'indice $j-1$.
Par récurrence, le résultat est vrai pour tout $j \geq -1$ donc en particulier pour $j = -1$. Les sous-arbres $\alpha(t,k,n)$ pour $2k > j+1 = 0$ sont ordonnés ; en particulier l'arbre $\alpha(t,1,n)$ est ordonné.
c. Chaque boucle effectue un appel à `ordonner` sur des paramètre j et n donc de complexité, d'après la question 8, $O(\log_2(\frac{n}{j})) = O(\log_2(n))$. La boucle est de taille $O(n)$ donc la complexité de la fonction `ordonner_totalement` est au pire $nO(\log_2(n)) = O(n\log_2(n))$.
11.a.

```
let bulle t j = echanger t 0 j; ordonner t 0 (j-1);;
```

b.

```
let tri t = ordonner_totalement t; for j=n-1 downto 1 do bulle t j;;
```

c.
d. `bulle t j` effectue un appel à `ordonner t 0 j` qui a complexité $O(\log_2(j))$ et un échange qui a complexité $O(1)$. La complexité de bulle t j est donc $O(\log_2(j))$. La fonction `tri` effectue un appel à `ordonner_totalement` qui a complexité $O(n\log_2(n))$ puis effectue une boucle de taille n, chaque passage de boucle ayant complexité $O(\log_2(j))$. La complexité de la boucle est donc $\sum_{j=1}^{n-1} O(\log_2(j)) = O(n\log_2(n))$. La complexité de `tri` est donc $O(n\log_2(n))$.