Sujet Mines-Centrale

Exercice Mines-Ponts

1. Sa signature est `int list -> int`. En effet, le filtrage impose à l'argument d'être une liste et, "max int" étant un entier, la valeur renvoyée par la fonction est un entier. De ce fait (cf. l'instruction "[a] -> a"), la liste doit être constituée d'entiers.

2. Par définition, la fonction se termine sur une liste d'entiers de longueur 0 ou 1. Sur une liste de taille $n \geq 2$, il y a une comparaison entre deux entiers et il y a un appel récursif sur une liste constituée d'un élément de tête et d'une queue de taille $n-2$; cette liste est donc de longueur $n-1$. Par récurrence, la fonction se termine sur toute liste d'entiers. Par ailleurs, le nombre C_n d'appels à elle-même sur une liste de taille n vérifie : $C_0 = C_1 = 1$, pour tout $n \geq 2$, $C_n = 1 + C_{n-1}$. Ainsi, pour tout $n \geq 1$, $C_n = n$ et $C_0 = 1$.

3. En convenant que min([ ]) =max_int, la fonction mystère calcule le minimum de la liste (qui est bien son minimum pour une liste non vide et max_int sinon). Cette propriété est vraie sur une liste de taille au plus un. Supposons le résultat pour toute liste de taille $n-1$ si $n \geq 2$. Soit une liste $l = [a_1, a_2, \cdots, a_n]$. L'instruction "(if a<=b then a else b) : : y" crée une liste qui contient en queue $[a_3, \cdots, a_n]$ et en tête $\min(a_1, a_2)$. L'appel récursif renvoie, par hypothèse de récurrence $\min([\min(a_1, a_2), a_3, \cdots, a_n])$ qui est $\min(a_1, \cdots, a_n) = \min(l)$. Ceci prouve le résultat par récurrence.

4. Chaque appel récursif se fait une une liste dont la queue existe déjà et dont la tête est l'un des deux éléments de tête précédents. Il n'y a donc pas d'utilisation d'espace mémoire supplémentaire que celui de la liste en argument.

Problème Centrale

Q11.

```
let rec succ_list l x =
    let tete liste = match liste with
        | [] -> -1
        | t::q -> t
    in match l with
        | [] -> -1
        | t::q -> if t=x then tete q else succ_list q x;;
```

Q12.

- Pour déterminer le maximum, il suffit d'appeler t à sa position n. La complexité est en $O(1)$ dans le pire des cas (dans tous les cas en fait).
- Pour tester l'appartenance d'un élément, on peut effectuer une recherche dichotomique (on compare récursivement l'élément cherché à l'élément médian de la partie du tableau à laquelle l'élément peut appartenir), ce qui exige une complexité dans le pire des cas en $O(\log(n))$.
- On ajoute un élément en trouvant sa place en faisant une recherche dichotomique ce qui repose sur une complexité au pire des cas en $O(\log(n))$. Ensuite, il faut décaler tous les éléments plus grands l'élément ajouté. Cette étape a une complexité au pire $O(n)$ (si l'on doit décaler tous les éléments du tableau).

Q13.

```
let succ_vect t x =
    let rec trouve t x i j =
        if j<i then -1
        else let k=(i+j)/2 in if x=t.(k) then k
                              else if x<t.(k) then trouve t x i (k-1) else trouve t x (k+1) j
    in let p=trouve t x 1 t.(0) in if p=t.(0) || p=-1 then -1 else t.(p+1);;
```

Q14. Dans le pire des cas, la recherche dichotomique demande une complexité $O(\log(n))$. Le reste des opérations est en temps constant. La complexité dans le pire des cas est en $O(\log(n))$.

Q15.

```
let union_vect t_1 t_2 = let n_maj=Array.length t_1 in let u=Array.make n_maj 0 in
    let rec aux i j k =
        if i>n_maj && j<=n_maj then begin u.(k) <- t_2.(j); u.(0) <- u.(0)+1; aux i (j+1) (k+1) end
        else if i<=n_maj && j>n_maj then begin u.(k) <- t_1(i); u.(0) <- u.(0)+1; aux (i+1) j (k+1) end
        else if i<=n_maj && j<=n_maj then
        if t_1.(i)<t_2.(j) then begin u.(k) <- t_1(i); u.(0) <- u.(0)+1; aux (i+1) j (k+1) end
```

```
        else if t_1.(i)>t_2.(j) then begin u.(k) <- t_2.(j); u.(0) <- u.(0)+1; aux i (j+1) (k+1) end
        else begin u.(k) <- t_1.(i); u.(0) <- u.(0)+1; aux (i+1) (j+1) (k+1) end
    in aux 0 0 0;
    u;;
```

Q16.

```
let min_abr a = match a with
    | Nil -> -1
    | Noeud(x,g,_) when g=Nil -> x
    | Noeud(_,g,_) -> min_abr g;;
```

Q17.

```
let rec partitionne_abr a = match a with
    | Nil -> (false, Nil, Nil)
    | Noeud(y,g,d) when y=x -> (true,g,d)
    | Noeud(y,g,d) when x>y -> let (b,ag,ad)=partitionne_abr d in (b,Noeud(y,g,ag),ad)
    | Noeud(y,g,d) -> let (b,ag,ad)=partitionne_abr g in (b,ag,Noeud(y,ad,d));;
```

Q18.

```
let insertion_abr a x = let (b,ag,ad)=partitionne_abr a in Noeud(x,ag,ad);;
```

La fonction de partition de la question précédente effectue, sur un binaire de hauteur h un appel récursif sur un arbre de hauteur $h-1$. De ce fait, cette fonction a une complexité au pire cas $O(h)$. La fonction d'insertion a donc également une complexité au pire cas $O(h) = O(n)$.

Q19. On peut, par exemple, insérer tous les éléments de E_2 dans E_1.

```
let union_abr a1 a2 = match a2 with
    | Nil -> a1
    | Noeud(x,g,d) -> union_abr d (union_abr g (insertion x a1));;
```

Q20. Les noeuds de profondeur $k \in [\![0, p]\!]$ sont numérotés de 2^k à $2^{k+1} - 1$. Le sous-arbre dont la racine a le numéro i a un nombre de feuilles 2^{p-k} si sa profondeur est k. D'après la première partie de la question, $2^k \leq i < 2^{k+1}$ donc $k = \lfloor \log_2(i) \rfloor$.

Q21. Ces numéros sont $2i$ pour le fils gauche, $2i+1$ pour le fils droit. En effet, pour le noeud le plus à gauche de profondeur p, son numéro est $i = 2^k$. Les deux fils sont les deux premiers de profondeur $k+1$: ils sont donc $2^{k+1} = 2i$ et $2^{k+1}+1 = 2i+1$. Si les fils du numéro $i \leq 2^k - 2$ sont $2i$ et $2i+1$, les fils du noeud suivant $i+1$ de profondeur k sont les deux noeuds suivant $2i$ et $2i+1$ qui sont $2i+2$ et $2i+3$. Les fils de $i+1$ sont donc $2(i+1)$ et $2(i+1)+1$. Par récurrence, les fils d'un noeud de profondeur k de numéro i sont $2i$ et $2i+1$, ceci pour tout $0 \leq k \leq p$.

Q22.

```
let rec appartient t x = let n=Array.length t in t.(i+n/2);;
```

La complexité de cette fonction est $O(1)$.

Q23.

```
let fabrique l x = let t=Array.make x false in
    let rec aux l = match l with
        | [] -> ()
        | y::q -> let k=ref y+x in while not t.(!k) do t.(!k) <- true; k:=!k/2 done; aux q
    in aux l;
    t;;
```

Q24.

```
let insere t x = let k=ref x+(Array.length t)/2 in
    while not t.(!k) do t.(!k) <- true; k:=!k/2 done;;
```

Q25.

```
let supprime t x = let k = ref x+(Array.length t)/2 in let p=ref !k/2 in
    t.(!k) <- false;
    while !k>0 && not t.(!k) do
        k:=!k/2;
        if not(t.(2*!k)) && not(t.(2*!k+1)) then t.(!k) <- false
    done;
```

k vaut initialement une valeur entre 2^p et $2^{p+1} - 1$, est minoré par 0 et est divisé par deux à chaque passage de boucle. Dans le pire cas, la complexité est donc $O(p)$.

Q26.

```
let minlocal t i = let n=(Array.length t)/2 and m=ref 1 and p= ref i in
    while !p<n do
        p:=2*!p;
        m:=2*!m
    done;
    let b=ref false and mini=ref -1 and imax=!p+m! in
    while !p<imax && not(!b) do
        if t.(!p) then begin b:=true; mini:=t.(!p) end done;
    !mini;;
```

Q27. On montre par récurrence que, tant que $t.(i+1)$ vaut "false", le successeur de l'élément initial n'est pas issu du sous-arbre du père de i.
Lorsque i est l'élément initial, si $t.(i+1)$ est "false", le successeur de i n'est pas $i+1$. Si les fils du père de i sont i et $i+1$, le successeur de i n'est donc pas dans l'arbre issu du père de i. Si $i+1$ et i n'ont pas le même père, l'arbre du père de i n'a pas de feuille à droite de i donc la propriété est encore vraie.
Supposons la propriété vraie pour une valeur i telle que $t.(i+1)$ est "false". Dans ce cas, la nouvelle valeur de i est son père. Par hypothèse les feuilles de son sous-arbre ne contiennent pas le successeur de la valeur initiale de i. Si $t.(i+1)$ est "false", l'arbre issu de $i+1$ n'a pas de feuilles dans l'ensemble E donc n'a pas le successeur de i. Si le père de i a pour fils i et $i+1$, ce père n'a donc pas le successeur de i ; si le père de i a pour fils i et $i-1$, $i-1$ ne peut avoir le successeur de i donc la propriété est vraie.
Par récurrence, cette propriété est donc vraie pour toutes les valeurs de i.
Lorsque $t.(i+1)$ devient "true", l'arbre issu de i n'a pas le successeur de i ; en revanche, $t.(i+1)$ étant "true", l'arbre issu de $i+1$ a un élément dans E. L'élément de E le plus à gauche de ce sous-arbre issu de $i+1$ est donc le premier élément de E situé à droite de la valeur initiale de i. C'est bien son successeur. De plus, il s'agit du minimum du sous-arbre issu de $i+1$. L'algorithme est donc valide dans ce cas.
Si $t.(i+1)$ reste toujours "false", on ne trouve jamais d'élément à droite de l'élément initial qui n'a donc pas de successeur. L'algorithme est donc encore valide.
Q28.

```
let successeur t x = let ref i = ref x and let imax= ref Array.length t in
    while !i<!imax-1 && not t.(!i+1) do
        i:=!i/2;
        imax:=!imax/2
    done;
    if !i=!imax-1 then -1 else minlocal t (!i+1);;
```

Q29. Plus généralement, on appelle successeur d'un noeud x le premier élément à sa droite (de même profondeur) ayant valeur "true". Montrons alors que la complexité pour la recherche du successeur de x est majorée par $K(\log_2(succeseur(x)-x)+2)$. Si $successeur(x) = x+1$, $\log_2(successeur(x)-x)+2 = 2$ qui est bien la complexité dans ce cas. Sinon, en notant $y = pere(x)$, $y = x/2$, $successeur(y) = successeur(x)/2$ donc $\log_2(successeur(x)-x) = 1+\log_2(successeur(y)-y)$; si la complexité de l'algorithme appliqué à y est majoré par $O(\log_2(successeur(y)-y)+2)$, alors, l'algorithme appliqué à x demande un appel récursif à l'algorithme appliqué à y donc la complexité de l'algorithme appliqué à x est $O(1+\log_2(successeur(y)-y)+2) = O(\log_2(successeur(x)-x)+2)$. Ceci montre, par récurrence décroissante sur la profondeur de x, que la complexité de cet algorithme est bien $O(\log_2(successeur(x)-x)+2)$.
Q30.

```
let cardinal t =
    let card=ref 0 and elem=ref minlocal t 0 in
    while !elem>-1 do
        card:=!card+1;
        elem:=successeur t !elem
    done;
    !card;;
```

Q31. D'après la question 29, la complexité de la fonction précédente est majorée par $\sum_{x\in E, x\neq\max(E)} K(\log_2(s(x)-x)+2) + K(\log_2(2^p-\max(E))+2)$. Par concavité de la fonction $\log_2$, $\sum_{x\in E, x\neq\max(E)} \log_2(s(x)-x) + \log_2(2^p-\max(E)) \leq \mathrm{card}(E)\log_2(\frac{1}{\mathrm{card}(E)}(\sum_{x\in E, x\neq\max(E)}(s(x)-x)+2^p-\max(E))) = \mathrm{card}(E)\log_2(\frac{2^p-\min(E)}{\mathrm{card}(E)}) \leq \mathrm{card}(E)\log_2(\frac{2^p}{\mathrm{card}(E)}) = n(p-\log_2(n))$. La complexité est donc majorée par $K(2n+n(p-\log_2(n))) = O(n(p-\log_2(n)))$.
Q32. Un inconvénient est que la taille maximale N est fixée à l'avance. Un avantage est que l'insertion d'un élément est plus rapide que dans une liste en moyenne, ainsi que le test d'appartenance à un ensemble. Un autre avantage est que la recherche d'un successeur est plus rapide que dans un tableau. La complexité dépend de la valeur de $p-\log_2(n)$. Cette représentation a tendance a être efficace pour de grands ensembles et pour des valeurs de n proches de 2^p (sachant que $n \leq 2^p$).

Q33. Ces ensembles sont $\{2,3\}$, $\{5-4,7-4\}=\{1,3\}$, $\emptyset$, $\{13-3\times4,14-3\times4\}=\{1,2\}$ et $\{0,1,3\}$. La valeur 3 de cette table est {`mini=1; maxi=2; table=[| {mini=-1; maxi=-1; table = [| |]}; {mini=0; maxi=0; table = [| |]}; {mini=1; maxi=1; table = [| |]} |]`}.

Q34.

```
let rec creer_veb p =
    let rec aux p = if p=1 then 2 else let x=aux p in x*x in
    if p=0 then {mini=-1; maxi=-1; table=[| |]}
    else {mini=-1; maxi=-1; table=Array.make (aux p +1) creer_veb (p-1)};;
```

Q35. $C(2^{2^s})=C(2^{2^{s-1}})+O(1)$. Notons $u_s=C(2^{2^s})$. Alors $u_s=u_{s-1}+O(1)$ donc $u_s=O(s)$ donc $C(q)=\log_2\log_2(q)$.

Q36.

```
let rec appartient_veb v x =
    if v.mini=-1 then false
    else
    if v.mini=x then true
    else let n=Array.length v.table-1 in
        if n=2 then if x=0 then v.mini=0 else v.maxi=1
        else let k=x/n and r=x mod n in appartient_veb v.table.(k) r;;
```

La complexité sur un arbre d'ordre $2^{2^s}=q$ vérifie la relation de récurrence $C(q)=C(\sqrt{q})+O(1)$ donc, dans le pire des cas, elle est de $O(\log_2(\log_2(N))$ sur un arbre d'ordre N.

Q37.

```
let rec successeur_veb v x =
    if v.mini=-1 then -1
    else let n=Array.length v.table-1 in
        if n=2 then
            if x=-1 then v.mini
            else if x=0 then if v.maxi=1 then 1 else -1
            else -1
        else
            if v.mini=x then let k=successeur_veb v.(n) -1 in successeur_veb v.(k) -1
            else let k=x/n in let y=successeur_veb v.table.(k) (x-k*n) in
                if y=-1 then
                    let p=successeur_veb v.table.(n) k in successeur_veb v.table.(p) (-1)
                else y+k*n;;
```

Dans le pire cas, la complexité sur un arbre de taille N vérifie la relation de récurrence $C(N)=3C(\sqrt{N})+O(1)$. Si $N=2^{2^p}$, en notant $C(N)=u_p$, $u_p=3u_{p-1}+O(1)$. Une telle suite vérifie $u_p=O(3^p)$ donc $C(N)=O(3^{\log_2(\log_2(N))})=O(\log_2(N))$.

Q38.

```
let rec insertion_veb v x =
    let n=Array.length v.table -1 in
    if n=2 then
        if x=0 then v.mini <- 0
        else v.maxi <- 1
    else
        if x<v.mini then
            begin
            let y=v.mini in v.mini <- x;
            insertion_ veb v y
            end
        else
            begin
            let k=x/n in
            insertion_veb v.(k) (x-k*n);
            insertion_veb v.(n) k
            end;;
```

Q39.

On a $M(N)=(\sqrt{N}+1)M(\sqrt{N})+O(1)=$ donc $\frac{M(N)}{N}=2\frac{M(\sqrt{N})}{\sqrt{N}}+O(\frac{1}{N})$. Si $N=2^{2^p}$ et $u_p=\frac{M(N)}{N}$, $u_p=2u_{p-1}+O(\frac{1}{2^{2^p}})$ donc $\frac{u_p}{2^p}=\frac{u_{p-1}}{2^{p-1}}+O(\frac{1}{2^p}\frac{1}{2^{2^p}})$; en notant $v_p=\frac{u_p}{2^p}$, $v_p=v_{p-1}+O(\frac{1}{2^p})$. La série $\sum\frac{1}{2^p}$ converge donc $\sum_{p=1}^{P}(v_p-v_{p-1})=O(1)$ donc $v_p=O(1)$ donc $u_p=O(2^p)$ donc $\frac{M(N)}{N}=O(\log_2(N))$ donc $M(N)=O(N\log_2(N))$.