Centrale 2023 — Représentations d'ensembles d'entiers

I. Langages et automates

Question 1 *On ne sait pas à quel point justifier les réponses, sachant qu'à la question suivante, on demande explicitement de justifier une partie de la réponse.*

Pour le premier exemple, on intercale simplement $\mathtt{a}^\star$ au milieu d'un mot de la forme $\mathtt{b}^\star$:

$$\mathtt{a}^\star \triangleright \mathtt{b}^\star = \mathtt{b}^\star\mathtt{a}^\star\mathtt{b}^\star$$

Pour l'exemple suivant, deux cas sont possible, suivant que l'on coupe le mot de K entre $\mathtt{a}$ et $\mathtt{b}$ ou entre $\mathtt{b}$ et $\mathtt{a}$.

$$(\mathtt{ba})^\star \triangleright (\mathtt{ab})^\star = (\mathtt{ab})^\star(\mathtt{ba})^\star(\mathtt{ab})^\star \mid \underbrace{(\mathtt{ab})^\star\mathtt{a}(\mathtt{ba})^\star\mathtt{b}(\mathtt{ab})^\star}_{(\mathtt{ab})^+}$$

Pour le troisième exemple, le mot de L s'ajoute soit avant le caractère $\mathtt{b}$ du mot de K, soit après, ce qui supprime la contrainte sur p et q :

$$\mathtt{a}^\star \triangleright \left\{\mathtt{a}^p\mathtt{b}\mathtt{a}^q \;\middle|\; (p,q) \in \mathbf{N}^2, p \leq q\right\} = \mathtt{a}^\star\mathtt{b}\mathtt{a}^\star$$

Question 2 On a tout d'abord :

$$L \triangleright (K_1|K_2) = (L \triangleright K_1)|(L \triangleright K_2) \qquad L \triangleright (K_1 \cdot K_2) = \big((L \triangleright K_1) \cdot K_2\big) \mid \big(K_1 \cdot (L \triangleright K_2)\big)$$

Justifions la troisième transformation. On peut écrire $K^\star$ sous la forme :

$$K^\star = \{\varepsilon\} \cup \bigcup_{n=1}^{\infty} K^n$$

Ainsi,

$$L \triangleright K^\star = L \triangleright \{\varepsilon\} \cup \bigcup_{n=1}^{\infty} L \triangleright K^n$$

Or, $L \triangleright \{\varepsilon\} = L$ et d'après la précédente,

$$L \triangleright K^n = \bigcup_{k=1}^{n-1} K^k \cdot (L \triangleright K) \cdot K^{n-1-k}$$

d'où

$$L \triangleright K^\star = L \cup K^\star \cdot (L \triangleright K) \cdot K^\star$$

Question 3 On déduit des relations précédentes que si L et K sont réguliers, alors $L \triangleright K$ l'est aussi. En effet, par induction structurelle sur K, il suffit de considérer les cas suivants :

- ⋆ Si $K = \emptyset$, alors $L \triangleright K = \emptyset$ et ilest régulier ;
- ⋆ Si $K = \{\varepsilon\}$, alors $L \triangleright K = L$ et il est régulier ;
- ⋆ Si $K = \{c\}$ avec $c \in \{\mathtt{a}, \mathtt{b}\}$, alors $L \triangleright K = c \cdot L \cup L \cdot c$ et il est régulier ;
- ⋆ Si $K = K_1 | K_2$ alors $L \triangleright K = (L \triangleright K_1) | (L \triangleright K_2)$ et si les $L \triangleright K_i$ sont réguliers, il en est de même pour $L \triangleright K$;
- ⋆ Les opérateurs de concaténation et d'étoile se traitent de façon similaire en se basant sur la question précédente.

Question 4 En s'inspirant des exemples précédents, on a

$$\mathtt{a}^\star \triangleright \left\{ \mathtt{a}^n \mathtt{b} \mathtt{a}^n \;\middle|\; n \in \mathbf{N} \right\} = \mathtt{a}^\star \mathtt{b} \mathtt{a}^\star$$

Ce dernier langage est bien régulier alors que $K' = \left\{ \mathtt{a}^n \mathtt{b} \mathtt{a}^n \;\middle|\; n \in \mathbf{N} \right\}$. En effet, supposons par l'absurde qu'il l'est. Par application du lemme de l'étoile, il existe un entier n tel que tout mot $u \in K'$ de longueur au moins n peut s'écrire sous la forme :

$$u = xyz$$

avec $|xy| \leq n$, $y \neq \varepsilon$ et $xy^\star z \subseteq K'$. Or, pour tout u, une telle décomposition est impossible. En effet, le facteur y ne peut contenir la lettre $\mathtt{b}$, celle-ci est donc présente soit dans le facteur x, soit dans le facteur z. Le facteur y contient donc nécessairement un nombre non nul de lettres $\mathtt{a}$. En particulier, on ne peut avoir $xz \in K'$ car le nombre de $\mathtt{a}$ de part et d'autre de la lettre $\mathtt{b}$ n'est plus le même.

Question 5 En notant $\mathcal{A}_L = (Q_L, I_L, F_L, \Delta_L)$ (resp. $\mathcal{A}_K = (Q_K, I_K, F_K, \Delta_K)$) un automate reconnaissant L (resp. K), on peut construire un automate $\mathcal{A}_\triangleright = (Q_\triangleright, I_\triangleright, F_\triangleright, K_\triangleright)$ reconnaissant $L \triangleright K$ de la façon suivante : on considère deux copies $\mathcal{A}_{K,1}$ et $\mathcal{A}_{K,2}$ de $\mathcal{A}$ pour représenter, pour un mot de $L \triangleright K$, la partie du mot dans K avant (resp. après) la partie dans L, et autant de copies de $\mathcal{A}_L$ qu'il y a d'états dans $\mathcal{A}_K$. En les notant $\left(\mathcal{A}_{L,q}\right)_{q \in Q_K}$, on passe d'une copie d'automate à une autre à l'aide d'ε-transitions de la façon suivante : il existe une ε-transition entre tout état q de $\mathcal{A}_{K,1}$ et tout état initial de $\mathcal{A}_{L,q}$ et entre tout état final de $\mathcal{A}_{L,q}$ et l'état q de $\mathcal{A}_{K,2}$. Le premier ensemble d'ε transitions indiquent le début des transitions sur un mot de L, l'autre

ensemble indiquant la fin des transitions sur un mot de L et le retour à l'état précédent dans $\mathcal{A}_K$.

Formellement, en notant $\uplus$ l'union disjointe :

$$\begin{aligned}
Q_\triangleright &= Q_{K,1} \uplus Q_{K,2} \uplus \biguplus_{q\in Q_K} Q_{L,q} \\
I_\triangleright &= I_{K,1} \\
F_\triangleright &= F_{K,2} \\
\Delta_\triangleright &= \Delta_{K,1} \uplus \Delta_{K,2} \uplus \biguplus_{q\in Q_K} \Delta_{L,q} \\
&\qquad \uplus \biguplus_{q\in Q_K, i\in I_L} (q_{K,1}, \varepsilon, i_{L,q}) \uplus \biguplus_{q\in Q_K, f\in F_L} (f_{L,q}, \varepsilon, q_{K,2})
\end{aligned}$$

Question 6 La fonction σ consiste à faire passer en tête la dernière lettre d'un mot, et la construction $\widehat{\cdot}$ consiste à prendre toutes les permutations circulaires des mots de L. Ainsi,

$$\sigma(L_1) = \varepsilon \mid \mathtt{b}(\mathtt{ab})^\star\mathtt{a} \qquad \widehat{L_1} = (\mathtt{ab})^\star \mid (\mathtt{ba})^\star$$
$$\sigma(L_2) = \mathtt{a}^\star\mathtt{b} \mid \mathtt{ba}^\star \qquad \widehat{L_2} = \mathtt{a}^\star\mathtt{ba}^\star$$

Question 7 Un mot non vide de L peut s'écrire sous la forme ux avec les notations précédentes. C'est l'étiquette d'un chemin acceptant dans $\mathcal{A}$. En notant $i \in I$ l'état de départ, $f \in F$ l'état final et q l'état précédant f, on a un chemin allant de i à q d'étiquette u, autrement dit $u \in L_{i,q}$ et $(q, x, f) \in \Delta$.

Ainsi, un mot non vide xu appartient à $\sigma(L)$ si et seulement si il existe des états $i \in I$, $q \in Q$ et $f \in F$ tels que $u \in L_{i,q}$ et $(q, x, f) \in \Delta$.

De plus,

$$\varepsilon \in \sigma(L) \iff \varepsilon \in L \iff I \cap F \neq \emptyset$$

Ainsi, on peut écrire :

$$\sigma(L) = \{\varepsilon \mid q \in I \cap F\} \cup \bigcup_{x\in X} \bigcup_{i\in I} \bigcup_{\exists f\in F:\,(q,x,f)\in\Delta} \bigcup_{q\in Q} x \cdot L_{i,q}$$

Question 8 D'après les questions précédentes, il est aisé, à partir d'un automate $\mathcal{A} = (Q, I, F, \Delta)$ reconnaissant L, de définir un automate $\mathcal{A}'$ reconnaissant $\sigma(L)$. Ce nouveau va être défini en juxtaposant, pour chaque lettre x de

l'alphabet, l'automate obtenu de la façon suivante : on part d'une nouvelle copie de $\mathcal{A}$, notons-à $\mathcal{A}_x$, auquel on ajoute un nouvel état i_x qui sera l'unique état initial de $\mathcal{A}_x$. Les états finaux de $\mathcal{A}_x$ sont exactement les états q pour lesquels il existe un état final $f \in F$ tel que $(q, x, f) \in \Delta$, ce qui correspond à la condition de la fonction précédente.

L'automate $\mathcal{A}'$ s'obtient alors en faisant l'union disjointe des différents $\mathcal{A}_x$ pour x décrivant X, auquel on ajoute au besoin un automate à un état reconnaissant le langage $\{\varepsilon\}$.

Question 9 En considérant la contraposée, cela revient à prouver que si $\sigma(L)$ est régulier, alors L est régulier. C'est directe, car on peut de même considérer la fonction σ^{-1} qui associe à xu le mot ux et qui s'étudie de façon similaire à σ.

Question 10 En s'inspirant des constructions précédentes, on peut définir un automate reconnaissant $\widehat{L}$ de la façon suivante : pour chaque état $q \in Q$, on considère deux copies $\mathcal{A}_{q,1}$ et $\mathcal{A}_{q,2}$ de $\mathcal{A}$, où $\mathcal{A}_{q,1}$ a q comme unique état initial et pas d'état final, et où $\mathcal{A}_{q,2}$ n'a pas d'état initial et a q comme unique état final, et où l'on a ajouté des ε-transitions entre tous les états de F dans $\mathcal{A}_{q,1}$ vers tous les états de I dans $\mathcal{A}_{q,2}$. Ainsi, un mot reconnu par l'automate obtenu correspond à un chemin qui doit partir de q dans $\mathcal{A}_{q,1}$, aller jusqu'à un état final « de $\mathcal{A}$ », emprunter une ε-transition vers un état initial « de $\mathcal{A}$ » puis finir en q dans $\mathcal{A}_{q,2}$. Il s'agit donc d'une permutation circulaire d'un mot reconnu par $\mathcal{A}$.

L'union disjointe de tous ces automates pour les différents états de $\mathcal{A}$ donne bien un automate reconnaissant $\widehat{L}$.

Concernant la réciproque, il est tout-à-fait possible d'avoir $\widehat{L}$ régulier sans que L ne le soit. On peut par exemple considérer le langage

$$L = \left\{ \mathtt{a}^p \mathtt{b} \mathtt{a}^q \;\middle|\; p \leq q \right\}$$

qui n'est pas régulier (la preuve est similaire à celle vue en question 2) et pour lequel $\widehat{L} = \mathtt{a}^\star \mathtt{b} \mathtt{a}^\star$, lequel est régulier.

II. Représentation classiques d'ensembles

1. Avec une liste triée

Question 11 L'écriture de cette fonction ne présente aucune difficulté.

```
let rec succ_list l x =
  match l with
  | [] -> -1 (* x plus grand que tous les éléments de la liste *)
  | v :: l' -> if v <= x then succ_list l' x else v
```

2. Avec un vecteur trié

Question 12

1. Le maximum du tableau est le dernier élément de celui-ci, il est situé à l'indice égal au nombre d'éléments (soit la valeur à l'indice 0). Il convient de faire attention au cas où le tableau ne contient aucune valeur. Cette opération s'effectue en temps constant (puis que l'on a un test et entre un et deux accés dans un tableau).
2. L'appartenance d'un élément se fait à l'aide d'une recherche dichotomique, de complexité logarithmique en le nombre d'éléments de E.
3. L'ajout d'un élément se fait en deux étapes. On commence par déterminer, à l'aide d'une recherche dichotomique, la position où insérer l'élément (et sa présence éventuelle), puis on effectue l'insertion proprement dite en décalant

Question 13 Il s'agit d'une question bien plus difficile et piégeante que ce que l'on peut croire, l'écriture d'une fonction de recherche par dichotomie correcte étant connue pour être particulièrement délicate.

La version classique se base sur l'utilisation de deux bornes `deb` et `fin` avec comme invariants :

$$\forall\, k < \texttt{deb}, \texttt{tab.}(k) < \texttt{v} \qquad \text{et} \qquad \forall\, \texttt{fin} < k, \texttt{v} < \texttt{tab.}(k)$$

L'échec de la recherche arrive alors lorsque `fin` $<$ `deb`. Dans ce cas, la plus petite valeur du tableau supérieure à celle recherche est située en `fin` $+ 1$. On a donc le programme suivant :

```
let succ_vect tab v =
  let rec aux deb fin =
    (* Invariant :
       k > fin => tab.(k) > v
       k < deb => tab.(k) < v *)
    if fin < deb then
      if fin + 1 <= tab.(0) then tab.(fin + 1) else -1
    else
```

```
    let mil = (deb + fin) / 2 in
    if tab.(mil) = v then
      if mil < tab.(0) then tab.(mil + 1) else -1
    else if tab.(mil) > v then aux deb (mil - 1)
    else aux (mil + 1) fin
  in
  aux 1 tab.(0)
```

Néanmoins, il existe une manière assez simple de procéder. On a toujours une fonction auxiliaire entre deux bornes `deb` et `fin`, mais cette fois-ci, on a comme invariants :

$$\forall\, k < \texttt{deb},\ \texttt{tab.}(k) < \texttt{v} \qquad \text{et} \qquad \forall\, \texttt{fin} \leq k,\ \texttt{v} \leq \texttt{tab.}(k)$$

en considérant que l'on a des valeurs $-\infty$ pour des indices négatifs et $+\infty$ pour des indices *trop grands*.

La condition de sortie est alors `deb` = `fin` et dans ce cas `tab.(fin)` est la plus petite valeur supérieure ou égale à `v` (valeur qui peut être égale à $+\infty$ avec la convention précédente, que l'on transforme alors en -1).

```
let succ_vect tab v =
  let rec aux deb fin =
    if deb = fin then if fin <= tab.(0) then tab.(fin) else -1
    else
      let mil = (deb + fin) / 2 in
      if tab.(mil) <= v then aux (mil + 1) fin else aux deb mil
  in
  aux 1 (tab.(0) + 1)
```

Question 14 La complexité dans le pire des cas de la fonction précédente est en $O(\log_2 n)$. Nous allons donner la justification pour la première version. Puisque à chaque itération, la largeur de l'intervalle $[\texttt{deb}, \texttt{fin}]$ est divisé par deux (quelque soit la version utilisée). On a donc, à la k-ème itération,

$$\texttt{fin} - \texttt{deb} \leq n2^k$$

et l'on sort de la boucle lorsque la largeur de cet intervalle est strictement inférieur à 1. On note que chaque itération s'effectue en temps constant.

Question 15 On a une fonction du type *fusion* du tri fusion, où l'on parcourt simultanément les deux vecteurs.

```
let union_vect v1 v2 =
  let v = Array.make (Array.length v1) 0 in
```

```
  let p1 = ref 1 and p2 = ref 1 and p = ref 1 in
  while !p1 <= v1.(0) && !p2 <= v2.(0) do
    if v1.(!p1) = v2.(!p2) then (
      v.(!p) <- v1.(!p1);
      incr p1;
      incr p2)
    else if v1.(!p1) < v2.(!p2) then (
      v.(!p) <- v1.(!p1);
      incr p1)
    else (
      v.(!p) <- v2.(!p2);
      incr p2);
    incr p
  done;
  while !p1 <= v1.(0) do
    v.(!p) <- v1.(!p1);
    incr p1;
    incr p
  done;
  while !p2 <= v2.(0) do
    v.(!p) <- v2.(!p2);
    incr p2;
    incr p
  done;
  v.(0) <- !p - 1;
  v
```

3. Avec un arbre binaire de recherche

Question 16 Pour déterminer le minimum dans un a.b.r., il suffit d'aller « le plus à gauche possible ».

```
let rec min_abr = function
  | Nil -> -1
  | Noeud (v, Nil, _) -> v
  | Noeud (_, g, _) -> min_abr g
```

Question 17 On retrouve une fonction classique sur les arbres binaires de recherche.

```
let rec partitionne_abr abr v =
  match abr with
  | Nil -> (false, Nil, Nil)
  | Noeud (e, g, d) ->
```

```
    if e = v then (true, g, d)
    else if e < v then
      let b, gd, dd = partitionne_abr d v in
      (b, Noeud (e, g, gd), dd)
    else
      let b, gg, gd = partitionne_abr g v in
      (b, gg, Noeud (e, gd, d))
```

Question 18 Il s'agit d'une insertion à la racine. On utilise donc la fonction précédente :

```
let insertion_abr abr v =
  let _, g, d = partitionne_abr abr v in
  Noeud (v, g, d)
```

La fonction `partitionne_abr` étant de complexité linéaire en la hauteur de l'arbre, il en est de même pour `insertion_abr` puisque les traitements complémentaires s'effectuent en temps constant.

Question 19 Une manière de procéder, illustrée par le programme suivant, consiste à faire une disjonction de cas selon le premier arbre. En particulier, si celui-ci est non vide, on partitionne le second arbre par rapport à la racine du premier, et on fusionne récursivement les arbres correspondant aux valeurs inférieures et supérieures à la racine.

```
let rec union_abr a1 a2 =
  match a1 with
  | Nil -> a2
  | Noeud (v1, g1, d1) ->
      let _, g2, d2 = partitionne_abr a2 v1 in
      Noeud (v1, union_abr g1 g2, union_abr g1 g2)
```

III. Représentation par arbres binaires complets

Question 20 Il y a 2^k nœuds à la profondeur k. Ainsi, un nœud à la profondeur k a un numéro compris entre

$$1 + \sum_{i=0}^{k-1} 2^i = 1 + 2^k - 1 = 2^k$$

et

$$\sum_{i=0}^{k} = 2^{k+1} - 1$$

Le nœud de numéro i est donc à une profondeur $\lfloor \log_2 i \rfloor$, et le sous-arbre de racine numéro i a dont $2^{p-\lfloor \log_2 i \rfloor}$ feuilles.

Question 21 Le nœud de numéro i est à une profondeur $\lfloor \log_2 i \rfloor$ et est à cette profondeur, il y a $i - 2^{\lfloor \log_2 i \rfloor}$ nœuds à sa gauche. Son fils gauche est alors à une profondeur $\lfloor \log_2 i \rfloor + 1$ et il y a $2 \times (i - 2^{\lfloor \log_2 i \rfloor})$ nœuds à sa gauche. Son numéro est donc $i_g = 2i$, et le fils droit du nœud i est $2i + 1$. De même, son père a pour numéro $\lfloor \frac{i}{2} \rfloor$.

Question 22 Comme indiqué dans l'énoncé, les feuilles ont les numéros partant 2^p qui n'est autre que la moitié de la taille du tableau.

```
let appartient ens i =
  let n = Array.length ens in
  ens.((n / 2) + i)
```

Question 23 Pour cette fonction, on commence par allouer un tableau à la bonne taille et remplir les cases correspondant aux feuilles, puis on remplit le reste du tableau du bas vers le haut, chaque nœud autre qu'une feuille prenant comme valeur la disjonction de ses fils.

```
let fabrique l t =
  let ens = Array.make (2 * t) false in
  List.iter (fun v -> ens.(t + v) <- true) l;
  for i = t - 1 downto 1 do
    ens.(i) <- ens.(2 * i) || ens.((2 * i) + 1)
  done;
  ens
```

Question 24 On part de l'indice correspondant à la valeur à ajouter et on mets à `true` la case d'indice correspondant ainsi que ses parents. Pour avoir une complexité en $O(1)$ dans le meilleur des cas, il faut arrêter la récursion dès que possible.

```
let insere ens v =
  let rec aux v =
    if v > 0 && not ens.(v) then begin
      ens.(v) <- true;
      aux (v / 2)
    end
  in
  aux ((Array.length ens / 2) + v)
```

Question 25 La suppression est un peu plus délicate puisque la valeur des parents doit prendre pour valeur la disjonction des valeurs des fils.

```
let supprime ens v =
  let rec aux p =
    if v > 0 then begin
      ens.(p) <- ens.(2 * p) || ens.((2 * p) + 1);
      aux (p / 2)
    end
  in
  let pos = (Array.length ens / 2) + v in
  ens.(pos) <- false;
  aux (pos / 2)
```

La complexité dans le pire des cas sera en $O(p)$.

Question 26 Plusieurs cas sont à considérer. Tout d'abord, si l'on est à la profondeur maximale, on renvoie soit l'indice courant si la valeur est présente, soit -1. Sinon, on explore les fils, en commençant par celui de gauche.

```
let rec minlocal ens i =
  if i >= Array.length ens / 2 then
    if ens.(i) then i - (Array.length ens / 2) else -1
  else
    let v = minlocal ens (2 * i) in
    if v > -1 then v else minlocal ens ((2 * i) + 1)
```

À chaque profondeur visitée, on traite au plus un nœud en temps constant. Ainsi, dans le pire des cas, comme on part de la profondeur $\lfloor \log_2 i \rfloor$ jusqu'à p inclus, on a donc une complexité en $O(p - \lfloor \log_2 i \rfloor + 1)$.

Question 27 Un nombre y plus grand que x se situe, dans l'arbre, à droite de x. Il existe une profondeur p à laquelle x et y se trouvent dans deux nœuds successifs. Le successeur sera, sous réserve d'existence l'unique élément présent dans l'ensemble et pour lequel la profondeur est maximal (minimisant ainsi l'écart entre x et y).

Question 28 On déduit directement la fonction de l'analyse précédente.

```
let successeur ens v =
  let rec aux d i =
    if i + 1 = d then -1 (* à droite de la ligne *)
    else if ens.(i + 1) then minlocal ens (i + 1)
```

```
    else aux (d / 2) (i / 2)
  in
  aux (Array.length ens) ((Array.length ens / 2) + v)
```

Question 29 Pour trouver le successeur y de x, il faut remonter à la profondeur $p_m = p - \lfloor \log_2(y-x) \rfloor$ où l'on va calculer le minimum local. La recherche de successeur va implique $p - p_m + 1$ appels successifs à la fonction auxilliaire de la fonction successeur suivi d'un appel à `minlocal` à partir de la profondeur p_m, pour une complexité en

$$\mathrm{O}(\underbrace{(p-p_m+1)}_{\texttt{aux}} + \underbrace{(p-p_m+1)}_{\texttt{minlocal}}) = \mathrm{O}\Big(2\log_2\big(\texttt{successeur}(x)-x\big)+2\Big)$$

Question 30 On utilise `minlocal` pour déterminer le plus petit élément de l'ensemble.

```
let cardinal ens =
  let rec aux v =
    if v = -1 then 0 else 1 + aux (successeur ens v)
  in
  aux (minlocal ens 0)
```

Question 31 En notant $(x_1, \dots, x_n)$ les éléments de E, la complexité de `cardinal` se décompose en $\mathrm{O}(p+1)$ pour déterminer x_1 à l'aide de `minlocal`, suivi de $n-1$ appels à `successeur`, pour une complexité totale de

$$\mathrm{O}\left(p+1+\sum_{i=1}^{n-1}(2\log_2(x_{i+1}-x_i)+2)\right)$$

Par concavité du logarithme, on a

$$\sum_{i=1}^{n-1}\log_2(x_{i+1}-x_i) \leq (n-1)\log_2\left(\frac{x_n-x_1}{n-1}\right) \leq (n-1)\log_2(x_n-x_1)$$

Comme $x_n - x_1 \leq 2^p$, on a finalement un complexité pour `cardinal` en

$$\mathrm{O}(np+n+p+1)$$

Question 32 Donnons quels éléments de réponse.

La taille de cette structure de donnée est linéaire en la largeur de l'intervalle dont on veut représenter une partie, et permet d'avoir les fonctions de base

(test d'appartenance, ajout, suppression et même successeur) en complexité logarithmique en fonction de cette largeur.

Il en est de même pour, par exemple, un simple tableau de booléens indiquant ou non l'entier correspondant à son indice. Pour un tel tableau, l'appartenance, l'ajout et la suppression se fait en temps constant, mais la recherche du sucesseur se fait en temps linéaire (de la taille de l'intervalle) dans le pire des cas, ce qui est moins bien. Par contre, le calcul du cardinal se fait lui aussi en temps linéaire, ce qui est plus efficace.

Comparé à une structure type arbre binaire comme vue précédemment, la taille de la structure n'est pas linéaire en le nombre d'éléments contenus, mais les complexités logarithmiques pour les opérations de base nécessitent un mécanisme d'équilibrage.

IV. Arbres de van Emde Boas

Question 33 Les ensembles sont respectivement $\{0,2,3\}$, $\{1,3\}$ (soit $\{5,7\}$ dont on a soustrait 4 à chaque élément), $\emptyset$ et enfin $\{1,2\}$. On a, pour la dernière case du tableau, $R = \{0,1,3\}$.

Pour représenter E_3, on utilise l'arbre suivant :

```
let t0 = { mini = -1; maxi = -1; table = [||] }
  and t1 = { mini = 0; maxi = 0; table = [||] }
  and t2 = { mini = 1; maxi = 1; table = [||] } in
  { mini = 1; maxi = 2; table = [| t0; t1; t2 |] }
```

Question 34 Commençons par définir une fonction auxilliaire pour calculer les puissances de 2 :

```
let rec puiss2 n = if n = 0 then 1 else 2 * puiss2 (n - 1)
```

La fonction demandée s'écrit alors sans difficulté de façon récursive :

```
let rec creer_veb p =
  if p = 0 then { mini = -1; maxi = -1; table = [||] }
  else
    {
      mini = -1;
      maxi = -1;
      table =
        Array.init
          (puiss2 (puiss2 (p - 1)) + 1)
          (fun _ -> creer_veb (p - 1))
```

```
    }
```

Question 35 Pour $q = 2^{2^s}$, la relation devient $C(2^{2^s}) = C(2^{2^{s-1}}) + O(1)$ de solution $C(2^{2^s}) = O(s)$.

Question 36 L'écriture de la fonction d'appartenance doit traîter plusieurs cas, suivant que l'arbre code l'ensemble vide (le *mini* vaut −1), la valeur recherchée est le minimum, ou bien l'arbre est d'ordre au moins 1, auquel cas on fait un appel récursif.

```
let rec appartient_veb v n =
  if v.mini = -1 then false
  else if v.mini = n then true
  else if Array.length v.table = 0 then false
  else
    let racine_N = Array.length v.table - 1 in
    appartient_veb v.table.(n / racine_N) (n mod racine_N)
```

Chaque appel récursif est en temps constant, et on en effectue $C(N) = O(p)$ dans le pire des cas.

Question 37 À chaque étape, il faut vérifier si l'arbre est vide, et, dans le cas contraire, déterminer si le successeur est le plus petit élément ou bien chercher (en renumérotant) dans le bon sous-arbre.

En voici une première version, assez peu efficace à cause de la boucle **while**.

```
let rec successeur_veb v n =
  if v.mini > n then v.mini
  else if v.maxi <= n then -1
  else
    let r = Array.length v.table - 1 in
    let p = ref n / r in
    while v.table.(!p).maxi <= n - (!p * r) do
      incr p
    done;
    (!p * r) + successeur_veb v.table.(!p) (n - (!p * r))
```

Sa complexité, dans le pire des cas, pour traiter un arbre de rang p, se décompose en la boucle, en $O(\sqrt{N}) = O(2^{2^{p-1}})$ suivi du traitement d'un arbre de rang $p-1$.

Pour avoir une bien meilleure complexité, nous allons exploiter tous les sous-arbres, y compris celui encodant l'ensemble R indiquant les sous-arbres non vides (pour reprendre les notations de l'énoncé).

Ainsi, pour trouver le successeur d'un entier n dans un arbre v, on commence par le comparer aux minimum et maximum de l'arbre. Dans le cas non trivial, on va chercher le successeur de n dans l'un des sous-arbres de v. Pour déterminer ce successeur, commençons par regarder dans le sous-arbre v_k qui devrait, en cas de présence, contenir n.

- ⋆ Si n est strictement inférieur au maximum de v_k, alors on va donc renvoyer le successeur de n dans v_k (ou, plus précisément, $k\sqrt{N}$ plus le successeur dans v_k de $n - k\sqrt{N}$).
- ⋆ Sinon, on cherche le prochain sous-arbre non vide de v et on en renvoie le minimum. Pour cela, on recherche le successeur de k dans R en utilisant pour cela le dernier sous-arbre de la table.

Dans les deux cas, on effectue un appel récursif plus des opérations en temps constant. Ainsi, la complexité est de l'ordre de $p = \log_2(\log_2 N)$.

```
let rec successeur_veb v n =
  if v.mini > n then
    v.mini
  else if v.maxi <= n then
    -1
  else
    let r = Array.length v.table - 1 in
    (* Si l'ordre est nul, on renvoie le maximum. *)
    if r = -1 then
      v.maxi
    else
      (* On détermine le bon sous-arbre *)
      let p = n / r in
      if v.table.(p).maxi > n - (p * r) then
        (* S'il contient un élément strictement plus grand,
           on y cherche le successeur *)
        (p * r) + successeur_veb v.table.(p) (n mod r)
      else
      (* Sinon, on veut le minimum du "sous-arbre" successeur *)
        let p' = successeur_veb v.table.(r) p in
        (p' * r) + v.table.(p').mini
```

Question 38 À nouveau, pour avoir la bonne complexité, il faut impérativement avoir éviter d'avoir deux appels récursifs. Cela est rendu possible par la gestion de la valeur minimale qui n'est pas présente dans un sous-arbre. En effet, l'insertion dans un sous-arbre vide se fait en temps constant, et c'est seulement dans ce cas-là que l'on modifie le sous-arbre représentant R.

```
let rec insertion_veb v n =
  (* L'arbre est vide *)
  if v.mini = -1 && v.maxi = -1 then (
    v.mini <- n;
    v.maxi <- n
  ) else if
      (* La valeur à insérer est le nouveau minimum. *)
      n < v.mini
    then (
    let mini = v.mini in
    v.mini <- n;
    insertion_veb v mini
  ) else (
    if n > v.maxi then v.maxi <- n;
    let r = Array.length v.table - 1 in
    (* L'insertion proprement dite n'a lieu que
       si l'ordre est non nul. *)
    if r > 0 then (
      let sub_veb = v.table.(n / r) in
      (* On va insérer la valeur dans la bonne sous-table. *)
      insertion_veb sub_veb (n mod r);
      (* Si l'arbre était vide... *)
      if sub_veb.mini = sub_veb.maxi then
        (* ...on modifie l'ensemble R. *)
        insertion_veb v.table.(r) (n / r)
    )
  )
```

Question 39 Commençons par définir :

$$\forall p, M'(p) = M\left(2^{2^p}\right)$$

On a alors

$$M'(1) = \mathrm{O}(1) \qquad \forall p, M'(p+1) = \mathrm{O}\left(2^{2^p}\right) + 2^{2^p} M'(p)$$

où, dans la relation de récurrence, le $\mathrm{O}(2^{2^p})$ correspond à l'allocation d'une structure `veb` et d'un tableau de taille $\sqrt{2^{2^{p+1}}} = 2^{2^p}$. On a alors

$$\forall p, \frac{M'(p+1)}{2^{2^{p+1}}} = \frac{\mathrm{O}\left(2^{2^p}\right)}{2^{2^{p+1}}} + \frac{2^{2^p} M'(p)}{2^{2^{p+1}}} = \frac{\mathrm{O}(1)}{2^{2^p}} + \frac{M'(p)}{2^{2^p}}$$

Ainsi,

$$\forall p, \frac{M'(p)}{2^{2^p}} = \sum_{k=0}^{p-1} \frac{\mathrm{O}(1)}{2^{2^k}}$$

et, finalement, la somme de droite étant clairement convergente,

$$M'(p) = \mathrm{O}\left(2^{2^p}\right) \quad \text{soit} \quad M(N) = \mathrm{O}(N)$$