20 / 20

Numéro de place [redacted]

Numéro d'inscription

Nom

Prénom

CONCOURS CENTRALE•SUPÉLEC

Épreuve Informatique

Ne rien porter sur cette feuille avant d'avoir complètement rempli l'entête

Feuille 01 / 06

1) On a :

$$a^* \triangleright b^* = \{ b^k a^l b^m \ / \ (k,l,m) \in \mathbb{N}^3 \} = b^* a^* b^*$$

$$(ba)^* \triangleright (ab)^* = \{ (ab)^k (ba)^l (ab)^m \ / \ (k,l,m) \in \mathbb{N}^3 \} = (ab)^* (ba)^* (ab)^*$$

$$a^* \triangleright \{ a^p b a^q \ / \ p \leq q \} = \{ a^{p+k} b a^q \ / \ p \leq q \} \ | \ \{ a^p b a^{q+k} \ / \ p \leq q \} = a^* b a^*$$

2) On a :

$$L \triangleright (K_1 | K_2) = (L \triangleright K_1) | (L \triangleright K_2)$$

$$L \triangleright (K_1 \cdot K_2) = (K_1 \cdot (L \triangleright K_2)) | ((L \triangleright K_1) \cdot K_2)$$

Si $m = uvw \in L \triangleright (K_1^*)$ où $uw \in K_1^*$ et $v \in L$. Il existe $q_0, \dots, q_{n-1} \in K_1$ tels que $uw = q_0 \cdot q_1 \cdots q_{n-1}$

En notant i le plus ~~grand~~ petit indice tel que q_i a au moins une lettre dans w, $n-1$ sinon,

m s'écrit $\underbrace{q_0 \cdots q_{i-1}}_{\in K_1^*} \; q_{i,1} \underbrace{v}_{\in L} q_{i,2} \; \underbrace{q_{i+1} \cdots q_{n-1}}_{\in K_1^*}$

où $q_{i,1}$ est la partie, potentiellement ε, de q_i dans v et $q_{i,2}$ celle dans w.

Comme $q_{i,1} q_{i,2} = q_i \in K_1$, $q_{i,1} v q_{i,2} \in L \Delta K_1$

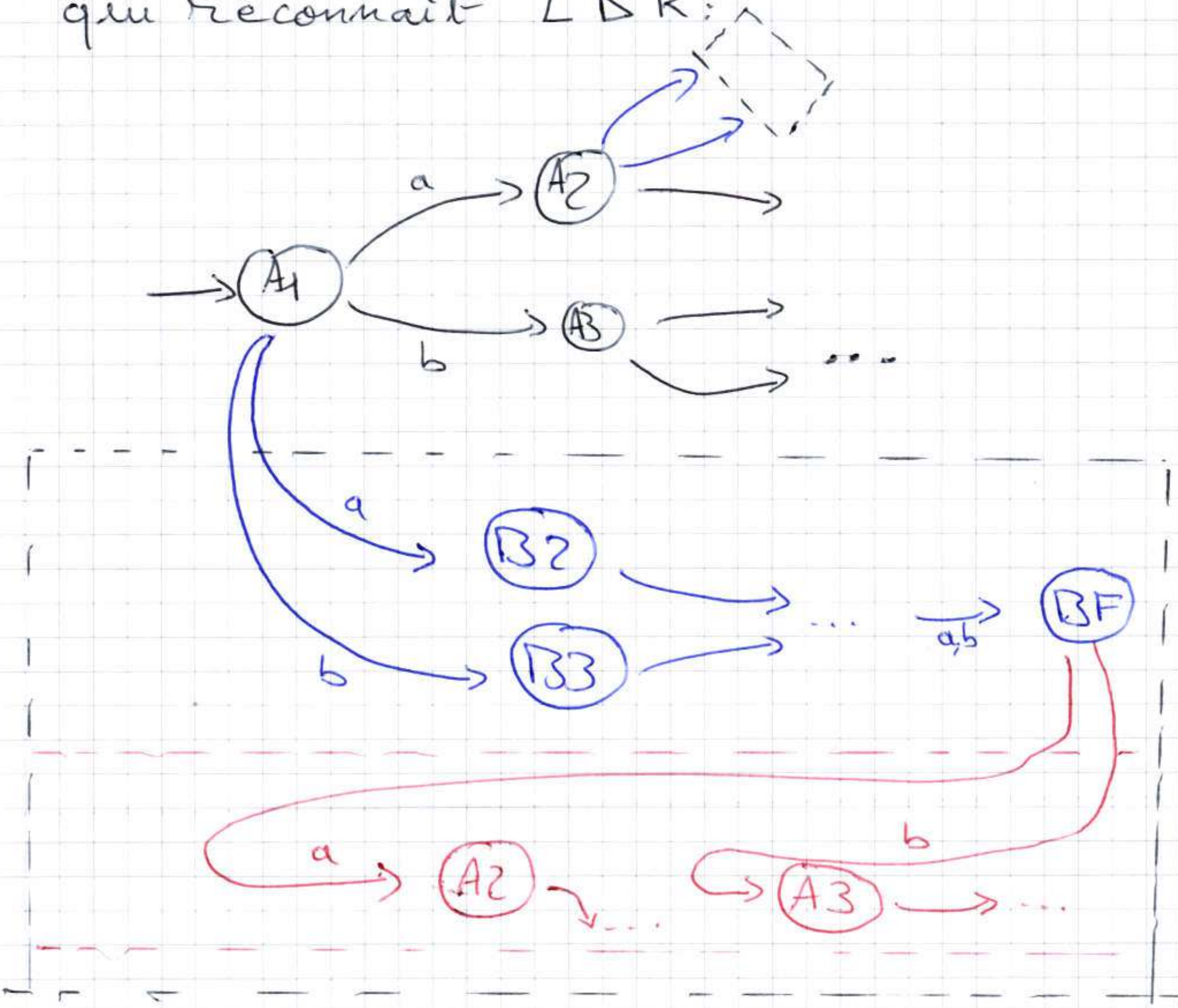
Donc $(L \Delta (K_1^*)) \subset K_1^* (L \Delta K_1) K_1^* \mid L$

Il est clair que tout mot de $K_1^*(L \Delta K_1) K_1^*$ est dans $L \Delta (K_1^*)$ tout comme tout mot de L : si $v \in L$, comme $\varepsilon = \varepsilon \cdot \varepsilon \in K_1^*$, $\varepsilon v \varepsilon \in L \Delta K_1^*$

Donc $L \Delta K_1^* = L \mid K_1^* (L \Delta K_1) K_1^*$

3) Si L et K sont réguliers, on se donne un automate A reconnaissant K et un B reconnaissant L. En ajoutant une transition de chaque état de ~~k~~ A ~~étiquetée par~~ vers chaque voisin d'un état initial de B étiquetée par la même lettre que dans B, et une transition de chaque état final de B vers chaque état d'une copie de A et en ne conservant les états

initiaux (respectivement finaux) que du premier A (resp. second), on obtient un automate qui reconnaît $L \Delta K$:



L'encadré noir existe pour chaque sommet q et contient une copie (en bleu) de B dont l'état initial a été remplacé par q et une copie de A ne contenant que les états accessibles depuis q, q étant remplacé par chaque état final de B.
On note $B_1, \ldots, B_n$ les états de B, $B_{f_1}, \ldots, B_{f_m}$ les états finaux de B, et de même pour A.

On construit $A' = (Q', I', F', \Delta')$ où :

- $Q' = Q_A \cup \bigcup_{q \in Q} \{ B_{i,q} / i \leq m \} \cup \{ B_{f,i,q} / i \leq m \}$
$\cup \, A(q) \setminus \{q\}$

où $A(q)$ est l'ensemble des états de A accessibles depuis q, renommés $A_{i,q}$

- $I' = I_A$
- $F' = \bigcup_{q \in Q} \{ A_{i,q} / A_i \in F_A \}$
- Δ' l'ensemble des triplets (q_1, σ, q_2)

où :
- Si $q_1, q_2 \in A(q)$, $(q_1, \sigma, q_2) \in \Delta_A$
- Si $q_1 \in Q_A$ il existe $q' \in I_B$ tel que $(q', \sigma, q_2) \in \Delta_B$
- ~~Si $q_2 \in A(q)$, il existe $q_f \in F_B$~~
Si $q_2 \in A(q)$, $q_1 \in \{ B_{f,i,q} / i \leq m \}$ et $(q, \sigma, q_2) \in \Delta_A$
- Si $q_1, q_2 \in Q_A$, $(q_1, \sigma, q_2) \in \Delta_A$
- Si $q_1, q_2 \in \{ B_{i,q} / i \leq m \} \cup \{ B_{f,i,q} / i \leq m \}$, $(q_1, \sigma, q_2) \in \Delta_B$

Alors, A' reconnait $L \triangleright K$, puisqu'il reconnait le début d'un mot de K, puis un mot de L, puis la fin d'un mot de K qui débute par la première séquence reconnue.

20 / 20

Numéro de place [redacted]

Numéro d'inscription

Nom

Prénom

CONCOURS CENTRALE-SUPÉLEC

Épreuve Informatique

Ne rien porter sur cette feuille avant d'avoir complètement rempli l'entête

Feuille 02 / 06

4) On pose $L = \cancel{\{a\}}\ \varepsilon$,

5) La construction d'un automate sans ε-transitions à été détaillée et formalisée question 3)

6) On remarque $\sigma^k(\sigma_0 \cdots \sigma_{n-1}) = \sigma_{n-1-k} \cdots \sigma_{n-1} \sigma_0 \cdots \sigma_{n-1-k}$

On a : si $u = (ab)^{\cancel{*}k} \in L_1$, $\sigma(u) = \begin{cases} \varepsilon \text{ si } u = \varepsilon \\ b(ab)^{k-1} a \text{ sinon} \end{cases}$

Donc $\sigma(L_1) = \{\varepsilon\} \cup \{b(ab)^k a \ / \ k \in \mathbb{N}\}$

De même, si $u = a^k ba \in L_2$, $\sigma(u) = a^{k+1} b$

~~si~~ $u = a^k b$, $\sigma(u) = ba^k$

Donc $\sigma(L_2) = \{ba^k, a^{k+1}b \ / \ k \in \mathbb{N}\}$

De plus, $\hat{L}_1 = \bigcup_{k=0}^{+\infty} \sigma^k(L)$

$= \bigcup_{u \in L} \{\sigma^k(u) \ / \ k \in \mathbb{N}\}$

Or pour $u = (ab)^k$, $\sigma^l(u) = \begin{cases} \varepsilon \text{ si } u = \varepsilon \\ u \text{ si } \cancel{2k}\ l = 2k \\ u \text{ si } l = 0[2] \\ b(ab)^{k-1} a \text{ sinon} \end{cases}$

Donc $\hat{L}_1 = L_1 \cup \sigma(L_1)$

De même : pour $u = a^k ba \in L_2$:

$\sigma^{\cancel{l}}(u) = a^{k+1} b$

et pour $u = a^k b$, $\sigma(u) = ba^k$

Donc si $u = a^k ba$, $\sigma^l(u) = \sigma^{l-1}(a^{k+1} b)$

Or, $\sigma^l(a^k b) = a^{l-1} b\, a^{k-l+1}$

Donc $\sigma^{k}(a^k b) = a^{k-1} b\, a$

Ainsi, $\hat{L}_2 = \{ a^k b\, a^l \,/\, (k,l) \in \mathbb{N} \}$

$= a^* b\, a^*$

7) On note $q_1, -, q_n$ les éléments de I,

On note $r_1, -, r_m$ les états tels que

~~existe~~ $\{ \sigma \in X \cup \{\varepsilon\} \,/\, \exists f \in F, (r_i, \sigma, f) \} = X_{r_i} \neq \emptyset$
si $\sigma \neq \varepsilon$ ou $r_i \in F$ sinon $\}$

On a alors :

$$\sigma(L) = \bigcup_{i=1}^{m} \bigcup_{j=1}^{n} X_{r_i} \cdot L_{q_j r_i}$$

puisque X_{r_i} représente l'ensemble des étiqu-

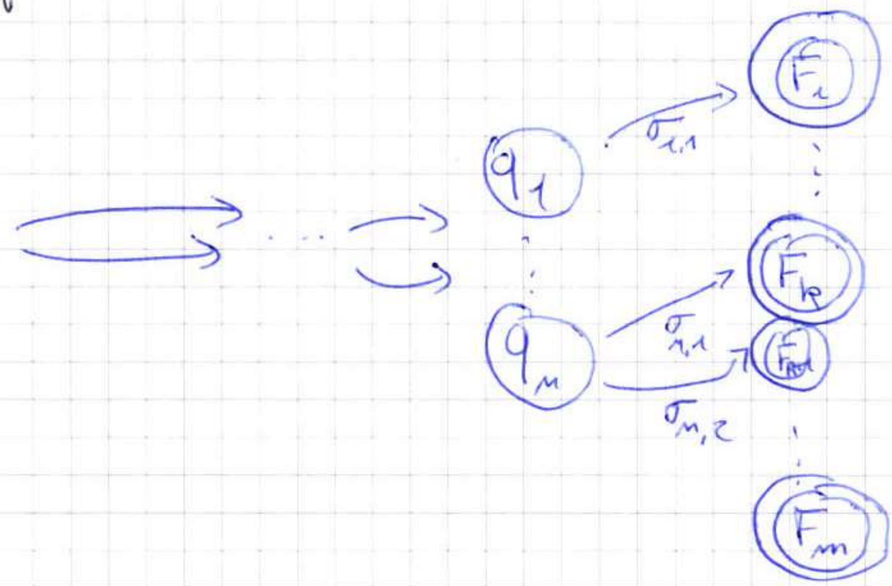
il existe $f_i \in F$ tel que $\{ |u| \,/\, u \in L_{r_i, f_i} \}$

8) Si L est régulier, on ajoute des ε-transitions de chaque état final de A reconnaissant L vers chaque état initial, puis chaque état accessible en une seule transition de A devient initial

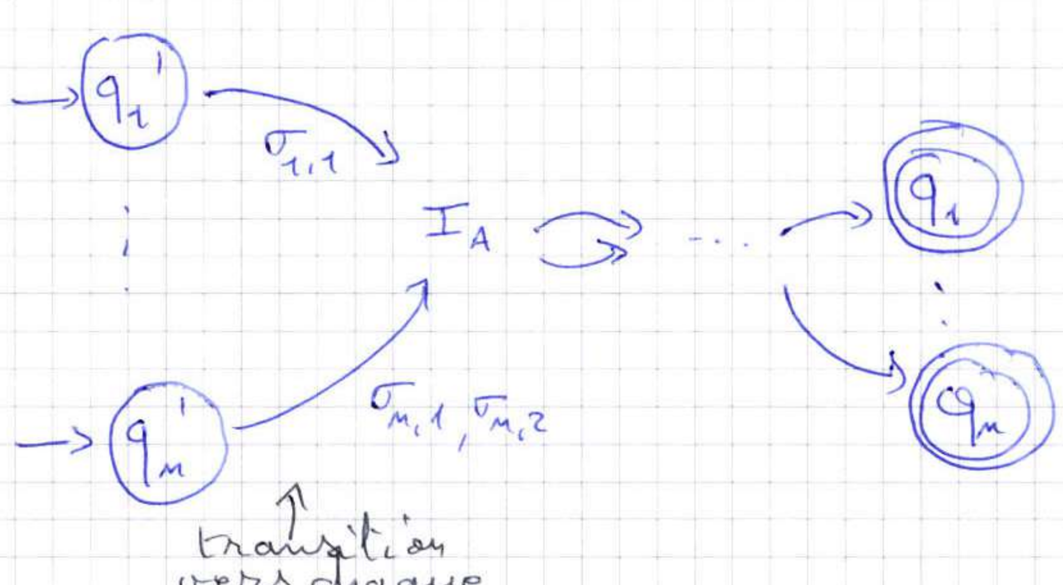
8) Si A reconnait L :

On crée A' en ajoutant à A des transitions étiquetées de même que celles de chaque état co-accessible en une transition de A à un état final de A allant de nouveaux états qui deviennent initiaux vers les états initiaux de A. Ensuite, on supprime les états finaux non coaccessibles en une transition et on rend les états coaccessibles en une transition finaux :

Si A est :



A' sera :



A' reconnait $\sigma(L)$

9) On peut procéder à l'inverse avec les états accessibles en exactement une transition pour reconstruire A partant de A'. Donc, si $\sigma(L)$ est régulier, L l'est.

Donc si L n'est pas régulier $\sigma(L)$ n'est pas régulier.

10) Si L est régulier,

11)

```
let rec succ_list l x = match l with
    | [] -> -1
    | h::t when h > x -> h
    | h::t -> succ_list t x
;;
```

~~12~~ On appelle la fonction au plus une fois par élément, ce qui donne une complexité en $\mathcal{O}(n)$, atteinte pour succ_liste [1, 1, ..., 1, 2] 1 par exemple.

12) Pour déterminer le maximum, il suffit d'accéder à la valeur de la case d'indice n où n est la valeur de la case 0.
Pour tester l'appartenance, on peut, par dichotomie, partir de l'indice $\frac{n-1}{2}+1=\frac{n+1}{2}$ et selon la valeur de la case étudier la partie gauche ou la partie droite.
Pour insérer, on modifie la case 0, puis on parcourt jusqu'à trouver où insérer et on modifie les valeurs de proche en proche.

~~13)~~
~~let succ_vect t x =~~
~~let g, d = 1, t.(0) in~~
~~let zm=~~
~~while d > g do begin~~
~~let m = (g+d) / 2 in~~
~~if (g+d) mod 2 = 0 then~~
~~let m = (g+d)/2 in~~
~~if~~

13)

```
let succ_vect t x =
    let rec find g d = ~~if d=g then t.(g)~~
      if t.(d) <= x then -1
        else if t.(g) > x then t.(g)
        else if (g+d) mod 2 = 0 then
            if t.((g+d)/2) > x then
                find g ((g+d)/2)
            else find ((g+d)/2) d
        else if t.((g+d)/2 +1) > x then
                find g ((g+d)/2 +1)
            else find ((g+d)/2 + 1) d

    in find 0 t.(0)

;;
```

14) A chaque itération, on divise $|g-d|$ par 2. On a au départ $d-g = n$, on a donc une complexité en $\mathcal{O}(\log_2(n)) = \mathcal{O}(\ln n)$

15)

```
let insere t1 x =
    let ~~ref~~ ov = ref t1.(1) in
    let ~~ref~~ place = ref true in
    ~~let~~ t1.(0) <- t1.(0) + 1 ;
    for i = 1 to t1.(0) do
        if !place && t1.(i) >= x then
            place := false;
            ov := t1.(i);
            t1.(i) <- x ;
        else ~~begin~~ if t1.(i) = x then ()
        else let temp = t1.(i) in
            t1.(i) <- !ov ;
            ov := temp;
    done;
;;
```

#O(|E₁|)

```
let union_vect t1 t2 =
    for i = 1 to t2.(0) do
        insere t1 t2.(i)
    done;
    t1
;;
```

#O (|E₂|)

16)

```
let rec min_abr ~~a~~ = function
    |Nil -> -1
    |Noeud(e, Nil, _) -> e
    |Noeud(_, g, _) -> min_abr g
;;
```

20 / 20

Numéro de place
Numéro d'inscription
Nom
Prénom
[redacted]

CONCOURS CENTRALE-SUPÉLEC

Épreuve Informatique

Ne rien porter sur cette feuille avant d'avoir complètement rempli l'entête

Feuille 04 / 06

17)

```
let rec partition_abr a x =
    match a with
    | Nil -> (false, Nil, Nil)
    | Noeud(e, g, d) when e < x ->
            let b, ag, ad = partition_abr d x in
            let l = Noeud(e, g, ag) in
            let r = ad in (b, l, r)
    | Noeud(e, g, d) when e > x ->
            let b, ag, ad = partition_abr g x in
            let r = Noeud(e, ad, d) in (b, ag, r)
    | Noeud(e, g, d) -> true, g, d
;;
```

18)

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

partition_abr a, dans le cas où l'arbre est enraciné ~~dans~~ en ~~un~~ min(E) une complexité

dans le pire des cas en $\mathcal{O}(n)$.
En effet, on va dans le pire des cas, avoir une complexité en $\mathcal{O}(h)$ où h désigne la hauteur de l'arbre, puisqu'on descend toujours en profondeur. Pour un arbre équilibré, $h = \mathcal{O}(\ln(n))$

Donc, en moyenne, on a une complexité dans le pire des cas en $\mathcal{O}(\ln(n))$

1g)

```
let rec union_abr a1 a2 =
    match a1, a2 with
    | Nil, _ -> a2
    | _, Nil -> a1
    | Noeud (e1, g1, d1), Noeud (e2, g2, d2) when e1=e2
          -> Noeud (e1, union_abr g1 g2,
                        union_abr d1 d2)
    | Noeud (e1, g1, d1), Noeud (e2, g2, d2) ~~when e1<e2~~
          -> let b, ag, ad = partition_abr e1 a2
             in ~~if b then~~ Noeud (e1, union_abr g1 ag,
                                       union_abr d1 ad)
```

ii
On insère les éléments un par un en appelant

récursivement union_abr pour fusionner les fils droits et gauches récursivement.

20) Un sommet à la profondeur k a son étiquette comprise entre 2^k et $2^{k+1}-1$, chaque profondeur comportant 2 fois plus de noeuds que la précédente.

Le sous-arbre dont la racine a le numéro i est enraciné à la profondeur $2^{\lfloor \log_2 i \rfloor}$ et a donc $2^{p-\lfloor \log_2(i) \rfloor}$ feuilles.

21) Il y a, entre ~~un~~ le sommet i et son fils gauche $2^{\lfloor \log_2(i) \rfloor+1} - 1 - i + 2\times(2^{\lfloor \log_2(i) \rfloor} - i)$ sommets, i.e. son fils gauche sera numéroté : $1 + i + (i-1) = 2i$, et son fils droit $2i+1$.

22)

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

Cette fonction s'exécute en temps constant

23)

```
let fabrique l n =
  let e = Array.make (2*n) false in        } O(2^p)
  let rec aux = function
    | [] -> ()                               } O(|l|)
    | h::t -> e.(h+n) <- true                }  = O(2^p)
  in aux l ;
  for i = (n-1) downto 1 do                  # O(2^p)
    e.(i) <- e.(2*i) || e.(2*i+1)
  done;
  e
;;
```

24)

```
let insere e k =
  let n = (Array.length e) / 2 in
  if e.(k+n) then ()      } O(1) dans le meilleur des cas : k ∈ E
  else begin
    e.(k+n) <- true; let i = ref (k+n)/2 in
    while !i > 0 do
      e.(!i) <- true ;
      i := !i / 2 ;
    done;
  end;
;;
```

20 / 20

Numéro de place [redacted]

Numéro d'inscription

Nom

Prénom

CONCOURS CENTRALE-SUPÉLEC

Épreuve Informatique

Ne rien porter sur cette feuille avant d'avoir complètement rempli l'entête

Feuille 05 / 06

25)

```
let supprime e k =
    let n = (Array.length e) / 2 in
      e.(k+n) <- false;
    let i = ref (k+n) / 2 in
    let w = ref true in
     while w && !i > 0 do
          e.(!i) <- e.(2*!i) && e.(2*!i+1);
          if e.(!i) then w := false
          else  i := !i / 2
     done;

;;
```

Dans le pire des cas on doit remonter l'arbre entièrement et on a une complexité en $O(p)$

Ne rien écrire dans la partie barrée

26) ~~let minlocal e i =~~
~~let rec aux = function~~
let rec minlocal e i =
if not e.(i) then -1
else if (2 * i) > Array.length e then e.(i)
else if e.(2*i) then minlocal e (2*i)
else minlocal e (2*i+1)
;;

On a une complexité linéaire en la hauteur du sous arbre enraciné en i, ~~donc~~ en $O(2^{p-\lfloor \log_2 i \rfloor})$

27) Il suffit de montrer que si l'on s'arrête en étant tout à droite, ~~on renvoie~~ x n'a pas de successeur.
En effet sinon, puisqu'on remonte depuis x, si l'un des fils droits d'un ancêtre de x vaut true, ~~et~~ x a un successeur qui est le minimum du ~~sous~~-arbre enraciné en le premier fils droit d'un ancêtre valant true i.e. la case i+1 où i est le fils découvert en remontant.
Dans le cas où on atteint tout à droite d'une profondeur, puisqu'on ne s'est pas arrêté précédemment, le fils droit de ce

sommet vaut false et donc il n'y a pas d'élément de E plus grand que x.

Donc l'algorithme renvoie bien le successeur de x dans E.

28)

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

;;
```

29) La fonction ci-dessus effectue un nombre constant d'opérations à chaque tour de boucle et si x a un successeur, on effectue un tour de boucle par profondeur du premier ancêtre commun à x et son successeur.
De plus minlocal a aussi une complexité linéaire en cette profondeur.
Or, cette profondeur vaut $2 + \log_2(s(x) - x)$ puisque leur ancêtre commun est obtenu $1 + \log_2(s(x) - x)$ étapes au dessus de x puisque

il faut $(s(x) - x)$ ~~sommets~~ noeuds internes dans l'arbre enracine en le fils droit de cet ancêtre commun

Donc on a une complexité ~~en O~~ majorée par $K(\log_2(s(x)-x)+2)$

30)

```
let cardinal e =
    let i = ref 0 in
    let x = ref 0 in
    if e.((Array.length e)/2) then i:=1;
    while !x <> -1 do
        x := successeur x;
        incr i
    done;
 !i
;;
```

31) On appelle $|E|$ fois successeur(x).

On a donc une complexité en

$$\sum_{\substack{x\in E \\ x\neq \max E}} K(\log_2(\text{successeur}(x)-x)+2) + K(\log_2(2^P-x)+2)$$

~~Par concavité : $\log_2(\text{successeur}(x)-x) \leq \log_2(2^P)$~~

ce qui vaut :

$$C \leq 2Km + K\sum_{\substack{x\in E \\ x\neq \max E}} \log_2(s(x)-x) + K\log_2(2^P-x)$$

20 / 20

Numéro de place [redacted]

Numéro d'inscription

Nom

Prénom

CONCOURS CENTRALE-SUPÉLEC

Épreuve Informatique

Ne rien porter sur cette feuille avant d'avoir complètement rempli l'entête

Feuille 06 / 06

31)

On a donc

$$C \leq 2K_m + K_m \log(2^p)$$
$$\leq 2Km + Km \log(m)$$

par concavité et croissance de $\log_2$

Donc on a une complexité en $\mathcal{O}(m \ln m)$

32) Une telle structure permet d'énumérer plus rapidement les éléments dans l'ordre, mais est plus volumineuse et est plus lente pour le calcul du cardinal via l'arbre et les successeurs. Cependant, elle est plus efficace pour l'insertion la suppression et l'appartenance. Elle est donc utile pour stocker de larges ensembles de l'ordre de 2^p.

33) On a ici : $N = 16$ donc $\sqrt{N} = 4$

et $\hat{E} = \{2, 3, 5, 7, 13, 14\}$

Donc $E_0 = \{2, 3\}$ codé ~~m~~ exe_veb.table.(0)

$E_1 = \{1, 3\}$ exe_veb.table.(1)

$E_2 = \emptyset$ exe_veb.table.(2)

$E_3 = \{1, 2\}$ exe_veb.table.(3)

On a ainsi :

exe_veb.table.(3) = {mini = 1 ; maxi = 2 ; table = [||]}

et exe_veb.table.(4) code $\{0, 1, 3\}$

34)

```
let rec creer_veb p =
      if p = 0 then {mini = -1 ; maxi = -1 ; table = [||]}
      else {mini = -1 ; maxi = -1 ; ~~table = Array.make~~
            table = Array.make (expo (p-1) + 1) (creer_veb (p-1))
           }
;;
```

où on aura défini :

```
let rec expo = function
      | 0 -> 2
      | p -> ~~expo (p-1)~~ let n = expo (p-1) in n*n.
;;
```

~~35) On pose $U = \log_2(q)$~~

~~On a : $C(U) = C(\frac{U}{2}) + O(1)$~~

~~Puis ainsi, avec $V = \log_2(U)$~~

~~$C(V) = C(V-1) + O(1)$~~

~~Donc $C(\log_2(\log_2(\frac{m}{q}))) = O(\log_2(\log_2(\frac{m}{q})))$~~

~~Donc en particulier :~~

~~$C(q) = O(s)$~~

~~36) let rec appartient_veb v x =~~

35) On a $C(q) = C(\sqrt{q}) + O(1)$

Donc, avec $s = \log_2(\log_2(q))$: $C(2^{2^s}) = C(2^{2^{s-1}}) + O(1)$

~~$C(2^U) = C(\frac{2^{U-1}}{2}) + O(1)$~~

~~Donc : $C(2^{2^s}) = C(2^{2^s/2}) + O(1)$~~

~~Donc~~ : $C(2^{2^s}) = O(s)$

Donc $C(q) = O(\log_2(\log_2(q)))$

```
36) let rec appartient_veb v x =   #C(q)
#O(1)  let s = (Array.length v.table) - 1 in
       ~~let k = x / s in~~ if x = v.mini then true
       ~~if v.table.(s~~
#O(1)  elif s < 0 then ~~false~~ x = v.mini || x = v.maxi
       else let k = x/s in
#O(1)    appartient_veb v.table.(k) (x - k*s)
         #C(√q)
;;
```

On a une complexité dans le pire des cas en $\mathcal{O}(\log_2(\log_2 N))$ d'après 35)

37)

```
let rec successeur_veb v x =                          C(N)
O(1) { let s = (Array.length v.table) - 1 in
       if s < 0 then if v.mini > x then v.mini
O(1) {    else if v.maxi > x then v.maxi
          else -1
O(1) { else let k = x / s in
C(√N) let s1 = successeur_veb v.table.(k) (x - k * s) in
O(1) if s1 <> -1 then s1 + k * s
O(1) else let i = ~~ref k+1 in~~ v.table.(s).mini
     ~~while~~
# O(1) { in ~~successe~~ v.table.(i).mini + i * s
;;
```

On a une complexité dans le pire des cas en $\mathcal{O}(\log_2(\log_2(N))$ par 35)

38)

39) On a :

$$M(N) = 2 + \sqrt{N+1}\, M(\sqrt{N})$$

Donc $M(N) = \sqrt{N}\, M(\sqrt{N}) + \mathcal{O}(1)$

Donc $M(N) = \mathcal{O}\left(\prod_{i=1}^{N} \sqrt{i} + N\right) = \mathcal{O}(\sqrt{N!} + N)$

$= \mathcal{O}(\sqrt{N!})$