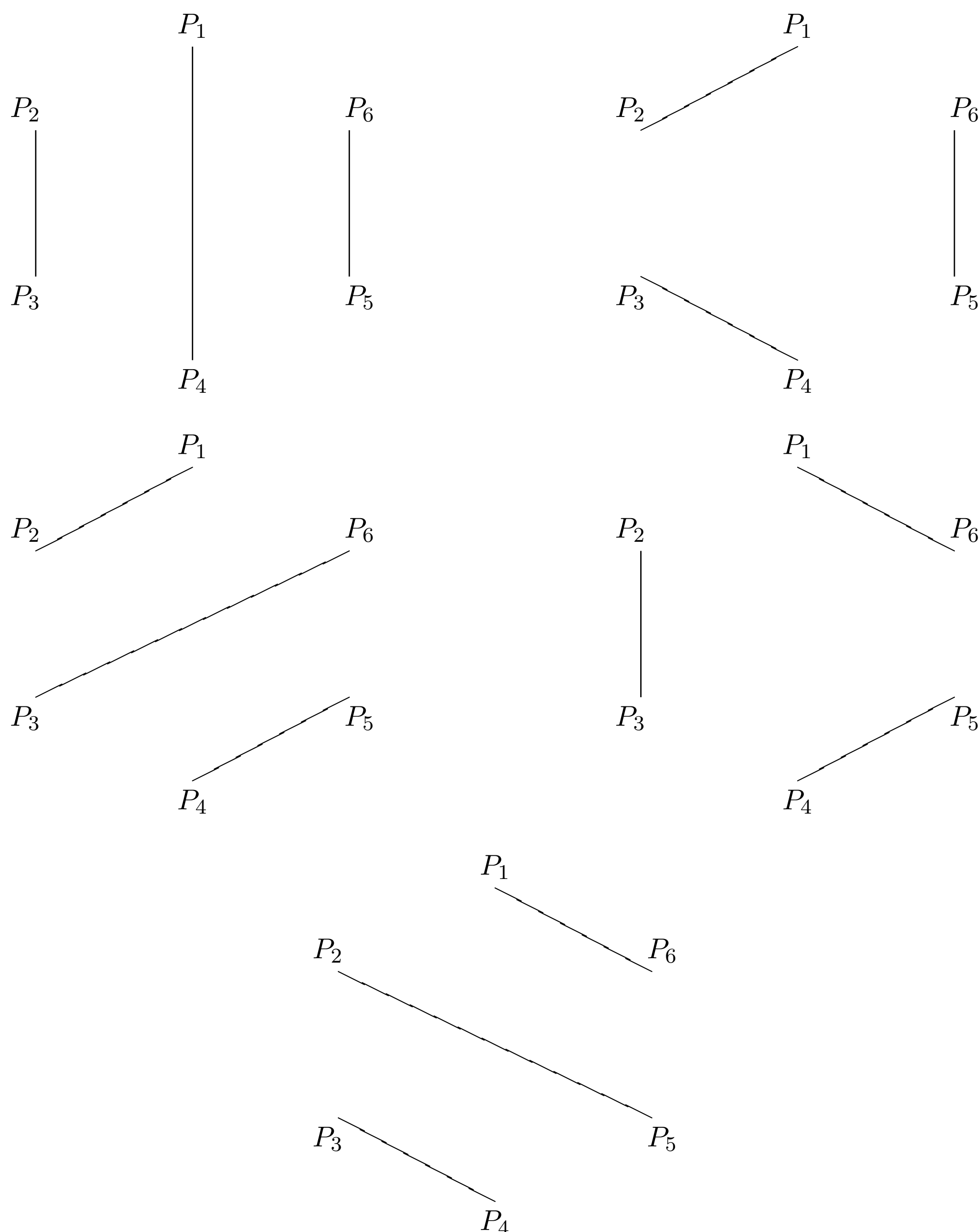
Centrale 2009 - Informatique
Un corrigé.

1 La chasse aux fantômes.

A.1. Il y 5 choix pour $c[1]$. Un choix étant fait ($c[1] = k \neq 1$), on a $c[k] = 1$. Soit $j \notin \{1, k\}$. Il reste trois choix pour $c[j] = i \neq j$ (chaque point est l'extremité d'une unique corde et $c[j] \notin \{1, k\}$) ce qui impose $c[i] = j$. Les deux éléments qui restent doivent alors être reliés.

Il y a $5 * 3 = 15$ stratégies possibles

A.2. Dans une stratégie adimssible, $c[1]$ doit être pair (sinon un sommet se trouve isolé). On obtient cinq stratégies représentées ci-dessous



A.3. On a $2n - 1$ choix pour $c[1]$ ce qui fixe une corde. Il reste alors $2n - 2$ points. En notant S_n le nombre de stratégies d'ordre n possibles, on a donc

$$S_n = (2n - 1)S_{n-1}$$

On en déduit par récurrence que

$$S_n = \prod_{k=1}^{n}(2k-1) = \frac{(2n)!}{2^n n!}$$

B.1. Avec les hypothèses faites, les segments $[P_iP_j]$ et $[P_kP_l]$ se croisent uniquement si

$$i < k < j < l \quad \text{ou} \quad k < i < l < j$$

B.2. Il suffit d'utiliser la question précédente.

```
let croise i j k l =
  (i<k & k<j & j<l) or (k<i & i<l & l<j) ;;
```

B.3. L'idée est simple puisqu'il suffit de comparer deux à deux les segments de la stratégie pour voir s'il se croisent. Comme on peut s'arrêter dès qu'un croisement est détecté, l'utilisation d'une boucle conditionnelle ou d'une exception s'impose. Par ailleurs, comme il n'est pas nécessaire de tester deux fois un segment, on peut se contenter de comparer les segments $(i, c[i])$ avec $c[i] > i$). Il suffit de comparer un tel segment avec les segment d'origine $k \in [i+1, c[i]-1]$ (si aucun de ceux là ne posent problème, il en sera de même des autres). On obtient les fonctions suivantes (version boucle conditionnelle où `test` indique si un problème a été détecté et version avec exception).

```
let estAdmissible c =
  let p=vect_length c in  (*p=2n+1*)
  let i=ref 1 and test=ref true in
  while !i<p & !test do
     if c.(!i) > !i then begin
        let k=ref (!i+1) in
        while !k<c.(!i) & !test do
              test:=not(croise !i c.(!i) (min !k c.(!k)) (max !k c.(!k))) ;
              incr k done ;
        end;
        incr i
     done ;
  !test ;;

exception secant ;;

let estAdmissible c =
  let p=vect_length c in  (*p=2n+1*)
  try
  for i=1 to p-1 do
     if c.(i) > i then begin
        for k=i+1 to c.(i)-1 do
              if croise i c.(i) (min k c.(k)) (max k c.(k)))
              then raise secant
              done ;
        end ;
      done;
      true
   with secant -> false ;;
```

Remarque : on a supposé ici que le vecteur donné en argument est bien une stratégie et on ne l'a pas vérifié. Cette vérification ne semble pas demandée par l'énoncé et se ferait en temps linéaire en fonction de n, puisque l'on a pour chaque case un nombre constant de vérification à faire.

B.4. Le calcul de complexité est simple sous la seconde forme puisque l'on a deux boucle inconditionnelles emboîtées et que chaque itération se fait en temps constant. Comme le nombre d'itérations est majoré par n^2, la complexité est quadratique ($O(n^2)$) en fonction de n.

C.1.

a. Si $j < i$ alors la première proposition est vrai (puisqu'il n'existe pas de k tel que $i \leq k \leq j$). Il en est de même de la seconde car il n'y a pas de point dans l'arc fermé $[P_iP_j]$ (l'énoncé n'est pas clair sur la définition de cet objet et je choisis cette convention). Dans le cas $j < i$, `evalue i j` vaut `true`.

b. `evalue i j` vaut `false` si $j \geq i$ et $c[i] < i$ car alors la première proposition est mise en défaut pour $k = i$.

c. `evalue i j` vaut `false` si $j \geq i$ et $c[i] > j$ car alors la première proposition est mise en défaut pour $k = i$.

C.2. On se place dans le cas $i < c[i] < j$.

- Supposons que `evalue i j` soit égale à `true` : on a donc deux proposition P_1 et P_2 qui sont vraies. Pour $k \in [i+1..c[i]-1]$, $c[k]$ doit être dans le même ensemble car sinon les cordes $(k, c[k])$ et $(i, c[i])$ se coupent ce qui est exclus (proposition P_2). De plus les segment ayant leurs extrémités dans l'arc $[P_{i+1}, P_{c[i]-1}]$ ne se coupent pas (car c'est vrai pour un arc plus grand d'après P_2). Ainsi `evalue (i+1) c[i]-1` vaut `true`. De façon similaire, `evalue c[i]+1 j` vaut `true`.
- Réciproquement, supposons que `evalue (i+1) c[i]-1` et `evalue c[i]+1 j` soient égales à `true` (on a quatre propositions P_1', P_2', P_1'', P_2'' vraies). Si $k \in [i..j]$ alors avec P_1', P_1'' et $c[c[i]]$, on obtient que $c[k] \in [i..j]$ (propriété P_1). De plus, avec P_1' (resp. P_1''), tout segment ayant une extrémité dans l'arc $[P_{i+1}P_{c[i]-1}]$ (resp. $[P_{c[i]+1}P_j]$) a l'autre extremité dans le même arc. Avec P_2' et P_2'', on obtient la seconde propriété P_2 voulue (la corde $(i, c[i])$ "sépare" deux ensembles de cordes).

On a finalement

$$\texttt{evalue i j} = \texttt{evalue (i+1) c[i]-1} \text{ et } \texttt{evalue c[i]+1 j}$$

C.3. Une stratégie d'ordre n sera admissible ssi `evalue 1 2n` est vraie. Il nous suffit d'crire une fonction locale d'évaluation qui repose sur les cas de base et formule récurrente des questions précédentes.

```
let testStrategie c =
  let rec evalue i j =
   if j<i then true
   else if c.(i)<i or c.(i)>j then false
   else (evalue (i+1) (c.(i)-1)) & (evalue (c.(i)+1) j)
in evalue 1 (vect_length c -1) ;;
```

C.4. Notons C_p le nombre maximal d'opérations effectuée par `evalue` quand ses arguments i et j vérifient $j - i + 1 = p$. Quand on effectue cet appel avec $j \geq i$ ($p \geq 1$) soit $c[i] \notin [i, j]$ et on s'arrête soit $c[i] \in [i+1, j]$ (on suppose $c[i] \neq i$) et on fait deux appels récursifs avec $i + 1$ et $c[i] - 1$ (et $k = c[i] - 1 - (i + 1) + 1 = c[i] - (i + 1) \in [0..p - 2]$) puis avec $c[i] + 1$ et j (et $j - (c[i] + 1) - 1 = p - 2 - k$). On a alors

$$\forall p \geq 2,\ \exists\, k \in [0..p-2] /\ C_p = O(1) + C_k + C_{p-2-k}$$

La majoration est vraie que les appels récursifs se fassent ou non.

Notons α une constante qui majore le côut d'un appel pour $p \leq 1$ et qui majore le coût constant $O(1)$ ci-dessus. On montre alors immédiatement par récurrence que

$$\forall p \geq 0,\ C_p \leq \alpha(p+1)$$

- C'est vrai si $p \leq 0$ ou $p = 1$.
- Si c'est vrai jusqu'à un rang $p - 1 \geq 1$ alors on trouve $k \in [0..p-2]$ tel que $C_p \leq \alpha + \alpha(k + 1) + \alpha(p - 2 - k + 1) = \alpha(p + 1)$ et le résultat est vrai au rang p.

`testStrategie` est bien de complexité linaéire en fonction de la taille de la stratégie (et comme avant, on n'a pas vérifié que c était une stratégie...).

D.1. Quitte à renuméroter, il nous suffit de prouver le résultat quand $i = 1$. On suppose donc que P_1 est un chasseur. Soit E l'ensemble des indices $k \in [1..2n]$ tels que dans $\{P_1, \ldots, P_k\}$ il y ait strictement plus de chasseurs que de fantômes. Comme $1 \in E$, E est non vide et comme $E \subset \mathbb{N}$, E possède un minimum m et $m < 2n$ (au total, il y a autant de chasseurs que de fantômes). Par définition du minimum,
- dans $\{P_1, \ldots, P_m\}$ il y a strictement plus de chasseurs que de fantômes
- dans $\{P_1, \ldots, P_{m+1}\}$ il y a moins de chasseurs que de fantômes (au sens large).

Comme "l'ajout" de P_{m+1} ne change le nombre de chasseurs ou (exclusif) de fantôme que d'une unité, P_{m+1} est un fantôme et dans $\{P_1, \ldots, P_{m+1}\}$ il y a autant de chasseurs que de fantômes. Il en est alors de même dans $\{P_{m+1}, \ldots, P_{2n}, P_0\}$ (puisque les nombres de fantômes et de chasseurs sont gloabelemnt égaux).

D.2. On décrit une stratégie prenant en argument i et j avec $i \leq j$ tels que dans $\{P_i, \ldots, P_j\}$ il y ait autant de fantômes que de chasseurs et renvoyant une stratégie admissible pour le sous-ensemble $\{P_i, \ldots, P_j\}$.
- Si $j = i + 1$ alors on associe P_i et P_j (il y a un fantôme et un chasseur).
- Si $j \geq i + 2$ (et donc $j \geq i + 3$ puisqu'il y a alors au moins quatre éléments) alors il existe un $k \in [i..j]$ tel que dans $\{P_i, \ldots, P_k\}$ et $\{P_k, \ldots, P_j, P_i\}$ il y ait autant de chaque "espèce" et tels que P_i et P_k soient d'espèces opposées (voir la question précédente). On associe P_i et P_k et on fait deux appels récursifs, l'un avec $i + 1$ et $k - 1$ et l'autre avec $k + 1$ et $i - 1$ (l'appel avec u et v ne se faisant pas si $u > v$) ; ceci a un sens car on fait des appels avec des données ayant les bonnes propriétés (autant de chasseurs que de fantômes).

Il suffit de faire l'appel initial avec 1 et $2n$.

D.3.

a. Notons C_p le nombre d'opération de notre algorithme appelé avec i et j tels que $j - i + 1 = 2p$ (le nombre d'éléments est forcément pair). La recherche de k séparant l'ensemble en deux se fait en $O(p)$ opérations. On a donc

$$\forall p \geq 1,\ \exists\, q \in [0..p-1] / C_p = O(p) + C_q + C_{p-1-q}$$

On montre alors à l'aide d'une récurrence que $C_p = O(p^2)$ (c'est similaire à ce qui a été fait en $C.4$; l'étape de récurrence peut se faire en étudiant la fonction $x \mapsto p + x^2 + (p - 1 - x)^2$ sur $[0, p - 1]$).

La complexité dans le cas le pire est $O(n^2)$ (atteinte quand on a tous les chasseurs puis tous les fantômes).

b. Comme dans le cas du tri rapide, la complexité en moyenne est $O(n \log(n))$ (c'est la complexité quand le découpage se fait toujours en deux comme dans une récurrence diviser pour régner).

D.4. On va mettre en oeuvre la strétégie précédente. Une première fonction auxiliaire locale
`milieu : int -> int -> int`
prend en argument $i \leq j$ et renvoie k tel qu'il y ait autant de fantômes et chasseurs dans $\{P_i, \ldots, P_k\}$ et $\{P_k, \ldots, P_j, P_i\}$ (on suppose que la situation est telle que k existe).

Une seconde fonction auxiliaire
`algo_aux : int -> int -> int list`
prend en argument des entiers i et j ainsi qu'une liste de couples (les cordes déjà construites) et renvoie la nouvelle liste de cordes. On utilise cette stratégie "accumulative" pour éviter d'avoir à effectuer des concaténations de listes (on pourrait aussi utiliser une référence de liste). La ligne commentée avec $*$ est la plus embêtante à comprendre : un permier appel récursif est fait et son résultat est utilisé dans le second pour continuer à accumuler.

```
let cibles p =
   let m=vect_length p  in (*m=2n+1*)
   let milieu i j =
```

```
    let compte=ref p.(i) and k=ref i in
    while !compte<>0 do incr k ; compte:= !compte + p.(!k) done ;
    !k
in let rec algo_aux i j l=
    if j<=i then l
    else if j=i+1 then (i,j)::l
    else begin
        let k=milieu i j in
        (i,j)::(algo_aux (i+1) (k-1) (algo_aux (k+1) (j-1) l))   (* * *)
        end
in algo_aux 1 (m-1) [];;
```

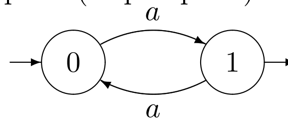
2 Automates finis.

A. Mots répétés.

A.1. Si $A = \{a\}$ alors $L = \{a^{2k+1}/\ k \in \mathbb{N}\}$ est constitué des mots de longueur impaire. On a

$$L = (a^2)^* a$$

On dessine aisément un automate à deux états reconnaissant L : on atteint l'état 0 (resp. 1) après lecture d'un mot de longueur paire (resp.impaire).



A.2. On suppose que $a, b \in A$ et $a \neq b$. Supposons, par l'absurde, que L soit reconnaissable. Son complémentaire, ensemble des mots du type w^2 avec $w \in A^*$, serait alors lui aussi reconnaissable. Notons alors n le nombre d'états d'un automate $\mathcal{A}'$ reconnaissant $L' = A^* \setminus L$. $a^n b^n a^n b^n \in L'$ et est donc un calcul réussi de $\mathcal{A}'$. La lecture de a^n nous fait visiter $n+1$ états et donc au moins deux fois le même. Il existe donc un motif a^p avec $p \geq 1$ qui est itérable et $a^{n+p} b^n a^n b^n$ sera l'étiquette d'un autre calcul réussi. Or, ce mot n'est pas dans L' et on obtient une contradiction.

A.3.

- Si $A = \{a\}$ alors L_p est l'ensemble des mots de longueur différente de 0 modulo p. On a ainsi

$$L_p = \bigcup_{k=1}^{p-1} (a^p)^* a^k$$

qui est rationnel. Soit $\mathcal{A}$ l'automate à p états numérotés $0, \dots, p-1$ dont 0 est l'état initial et dont les états terminaux sont les $1, \dots, p-1$. Les transitions de $\mathcal{A}$ sont les $(i, a, i+1)$ pour $i \in [0..p-2]$ et $(p-1, a, 0)$. La lecture de a^k nous mène en i ssi $k = i[p]$. L'automate reconnaît donc exactement L_p.

- On suppose que $a, b \in A$ et $a \neq b$ et on note $L'_p = A^* \setminus L_p$. Supposons, par l'absurde, que L_p soit reconnaissable. L'_p est alors lui aussi reconnaissable. Notons alors n le nombre d'états d'un automate $\mathcal{A}'$ reconnaissant L'_p. $(a^n b^n)^p \in L'$ et est donc un calcul réussi de $\mathcal{A}'$. La lecture de a^n nous fait visiter $n+1$ états et donc au moins deux fois le même. Il existe donc un motif a^p avec $p \geq 1$ qui est itérable et $a^{n+p} b^n (a^n b^n)^{p-1}$ sera l'étiquette d'un autre calcul réussi. Or, ce mot n'est pas dans L'_p et on obtient une contradiction.

B. Logique de Presburger.

B.1. (4) est représenté par $u = (0), (0), (1)$ (car 4 s'écrit 100 en base 2).

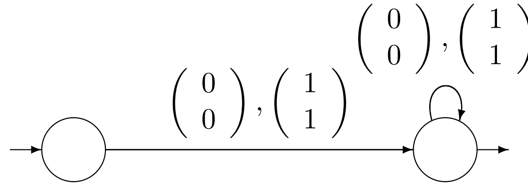
De façon similaire, $(2, 3, 0)$ est représenté par $\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ et $(2, 3, 5)$ est représenté par $\begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$.

Remarque : il n'y a pas unicité d'une représentation puisque l'on peut ajouter à droite des n-uplets nuls (en base 2, on peut ajouter des bits nuls à gauche du bit de poids fort).

B.2. On a (le mot vide n'est pas reconnu)

$$E_g = \left(\begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right)^+$$

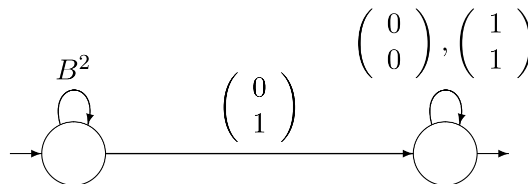
La relation E_g est rationnelle et reconnue par l'automate suivant



Supposons que x et y s'écrivent $x_p \dots x_1$ et $y_p \dots y_1$ en base 2 (avec le même nombre de bits quitte à ajouter des 0). On a alors $x < y$ s'il existe un k tel que $x_k < y_k$ et $x_i = y_i$ pour $i \geq k+1$. Ainsi

$$Inf = (B^2)^* \begin{pmatrix} 0 \\ 1 \end{pmatrix} \left(\begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right)^*$$

La relation Inf est rationnelle et reconnue par l'automate suivant



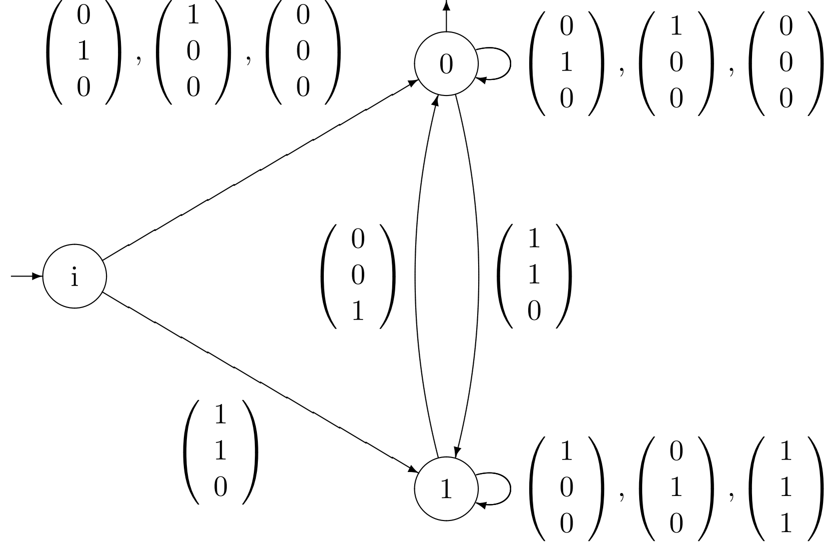
B.3. Pour la relation Add, se pose le problème de la retenue dans le calcul de la somme $x+y$. Si x, y et z s'écrivent $x_k \dots x_1$, $y_k \dots y_1$ et $z_k \dots z_1$ en base 2 alors $x_i \dots x_1 + y_i \dots y_1$ vaut $z_i \dots z_1$ ou $1z_i \dots z_1 = 2^i + z_i \dots z_1$. Il s'agit, au moment de lire les bits suivants, de savoir dans laquelle des situations on est (doit-on avoir $x_{i+1} + y_{i+1} = z_{i+1}[2]$ ou $x_{i+1} + y_{i+1} + 1 = z_{i+1}[2]$ et transmet-on une retenue ?).

Je crée un automate à trois états :

- Un état initial i utile pour éviter que le mot vide soit reconnu. On ne se trouve en cet état qu'initialement.
- Un état 0 dans lequel on se trouve quand tout s'est bien passé et que l'on ne transmet pas de retenue.
- Un état 1 dans lequel on se trouve quand tout s'est bien passé et qu'on transmet une retenue.

Les transitions sont choisies pour que les propriétés ci-dessus soient vérifiée. Par exemple si on est dans l'état 0 et qu'on lit (x_i, y_i, z_i), il faut que $x_i + y_i = z_i[2]$ pour que le mot ait une chance de convenir et on va dans l'état 0 si $x_i + y_i = z_i$ et dans l'état 1 si $x_i + y_i = z_i + 2$. De même, si

on est dans l'état 1 et qu'on lit (x_i, y_i, z_i), il faut que $1 + x_i + y_i = z_i[2]$ pour que le mot ait une chance de convenir et on va dans l'état 0 si $1+x_i+y_i = z_i$ et dans l'état 1 si $x_i+y_i+1 = z_i+2$. On reconnaîtra *Add* en choisissant 0 comme état final (on ne transmet pas de retenue en fin de calcul).



B.4. Soit $\mathcal{A} = (Q, B^n, i, \delta, F)$ un automate déterministe reconnaissant R.
Notons $\mathcal{A}_Q = (Q, B^{n-1}, i, \delta', F)$ l'automate déduit de $\mathcal{A}$ en remplaçant toutes les transitions $(q, (b_1, \dots, b_n), q')$ par les nouvelles transitions $(q, (b_1, \dots, b_{n-1}), q')$ (deux transitions de $\mathcal{A}$ peuvent bien sûr donner naissance à la même transition de $\mathcal{A}_Q$). Il est alors immédiat que $\mathcal{A}_Q$ reconnaît Q et Q est ainsi rationnelle.
Notons $\mathcal{A}_S = (Q, B^{n-1}, i, \delta'', F)$ l'automate tel que

$$(q, (b_1, \dots, b_{n-1}), q') \in \delta'' \iff \begin{cases} (q, (b_1, \dots, b_{n-1}, 0), q') \in \delta \\ (q, (b_1, \dots, b_{n-1}, 1), q') \in \delta \end{cases}$$

L'automate $\mathcal{A}_S$ reconnaît S et S est ainsi rationnelle.

C. Automate minimal.

C.1. Soit $\mathcal{A} = (Q, A, i, F, \delta)$ avec
- $Q = \{0, \dots, N+1\} \cup \{R\}$
- $i = 0$
- $F = \{N+1\}$
- $\forall q \in \{0, \dots, N-1\},\ \delta(q, a) = \delta(q, b) = q+1,\ \delta(N, a) = N+1,\ \delta(N, b) = R,\ \delta(N+1, a) = \delta(N+1, b) = N+1,\ \delta(R, a) = \delta(R, b) = R$.

Cet automate est déterministe complet et reconnaît G_N (l'état R est rebut et est atteint quand on lit un b en position $N+1$).

C.2. Soit $\mathcal{A} = (Q, A, i, F, \delta)$ avec
- $Q = \{0, \dots, N+1\}$
- $i = 0$
- $F = \{N+1\}$
- $\forall q \in \{1, \dots, N\},\ \delta(q, a) = \delta(q, b) = q+1,\ \delta(0, a) = \{0, 1\}$ et $\delta(0, b) = 0$.

Cet automate reconnaît D_N (quand on quitte l'état 0, c'est par un a et il reste N lettres à lire).

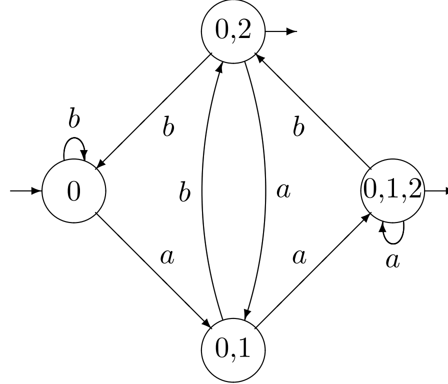
C.3. Soit $\mathcal{A} = (Q, A, i, F, \delta)$ un automate déterministe complet qui reconnaît G_N. Quitte à renommer les états, on peut supposer que $i = 0$. Posons $q_k = \delta^*(0, a^k)$ (état atteint après lecture de

a^k). Soient $0 \leq i < j \leq N+1$; $\delta^*(q_j, a^{N+1-j}) = \delta^*(0, a^{N+1})$ et $\delta^*(q_i, a^{N+1-j}) = \delta^*(0, a^{N+1+i-j})$ sont différents puisque l'un est final (a^{N+1} est reconnu) et pas l'autre ($a^{N+1+i-j}$ ne l'est pas). On a donc $q_i \neq q_j$. Q possède donc au moins $N+2$ élement $(q_0, \ldots, q_{N+1})$.
Posons $\delta(q_N, b) = q_{N+2}$; pour $0 \leq i \leq N+1$, on a $\delta^*(q_i, a^{N+1-i}) = \delta^*(0, a^{N+1})$ et $\delta^*(q_{N+2}, a^{N+1-i}) = \delta^*(0, a^N b a^{N+1-i})$ et comme plus haut, q_{N+2} est différent des q_i ce qui donne encore un état de plus.
Finalement l'automate contient au moins $N+3$ états.

C.4. En déterminisant par la méthode des sous-parties, on obtient la table suivante :

	0	$0,1$	$0,1,2$	$0,2$
a	$0,1$	$0,1,2$	$0,1,2$	$0,1$
b	0	$0,2$	$0,2$	0

L'automate déterminisé a donc l'allure suivante



C.5. Soit $\mathcal{A} = (Q, A, i, F, \delta)$ un automate déterministe complet qui reconnaît D_N. Quitte à renommer les états, on peut supposer que $i = 0$. Pour tout mot u, on note $q_u = \delta^*(0, u)$. Soient $u = u_1 \ldots u_{N+1}$ et $v = v_1 \ldots v_{N+1}$ deux mots distincts de longueur $N+1$. Soit k l'indice maximal (il existe) tel que $u_k \neq v_k$. Par exemple, on a $u_k = a$ et $v_k = b$. On a alors ua^k qui est reconnu (la $N+1$-ème avant la fin est un a) mais pas va^k (la $N+1$-ème avant la fin est un b) et donc $\delta^*(q_u, a^k) \neq \delta^*(q_v, a^k)$ et donc $q_u \neq q_v$. Il y a donc au moins autant d'états que de mots de longueur $N+1$ c'est à dire 2^{N+1}.

C.6. Le plus petit mot de D_N ou G_N est de longueur $N+1$. S'il a moins de $N+1$ états, un automate reconnaissant au moins un mot en reconnaît un de longueur N (quitte à supprimer les cycles d'un calcul réussi, on peut en trouver un qui ne passe pas deux fois par le même état). Ainsi, tout automate reconnaissant D_N ou G_N a au moins $N+2$ états.