Exercice I

1- Première épreuve

1) $B_1 = V_1 \vee V_2$.
2) $B_2 = \overline{V_1}$.
3) On note F la proposition résultant de la règle du jeu (et qui est donc vraie).

$$\begin{aligned} F &= (B_1 \wedge B_2) \vee (\overline{B_1} \wedge \overline{B_2}) \\ &\equiv ((V_1 \vee P_2) \wedge \overline{V_1}) \vee (\overline{V_1} \wedge \overline{V_2} \wedge V_1) \\ &\equiv (V_1 \wedge \overline{V_1}) \vee (V_2 \wedge \overline{V_1}) \\ &\equiv V_2 \wedge \overline{V_1} \end{aligned}$$

4) Il faut donc choisir la boîte 2.

2 - Deuxième épreuve

1) En notant B_i l'inscription de la boîte i,

$$B_1 = \overline{V_3} \wedge \overline{R_3},\ B_2 = R_1,\ B_3 = \overline{V_3} \wedge \overline{R_3}$$

(on traduit la vacuité par l'absence de clé verte et de clé rouge).

2) Les informations données par l'animateur peuvent se synthétiser par la formule

$$\bigwedge_{i=1}^{3} \left(\left(R_i \wedge \overline{B_i} \right) \wedge \left(\overline{R_i} \wedge B_i \right) \right)$$

En effet, pour chaque boite, l'inscription est fausse si et seulement si elle contient la clé rouge.

3) Supposons que la clé verte soit dans la boîte 2. Ainsi, $V_2 = 1$ donc $B_2 = 1$ ce qui implique que $R_1 = 1$ et donc que c'est la boite 3 qui est vide. Cependant, $R_1 = 1$ signifie que $B_1 = 0$. C'est absurde.

4) De même si on suppose que la clé verte est dans la boite 3 on obtient directement une contradiction (la boite ne peut pas être vide).

On obtient donc que la clé verte est dans la boite 1 ; cela implique $B_1 = 1$ et donc que la boite 3 est vide. La clé rouge est dans la boite 2.

Exercice II

1 - Transformation de Burrows-Wheeler (BWT)

1)

t	u	r	l	u	t	u	t	u	$\vert$
$\vert$	t	u	r	l	u	t	u	t	u
u	$\vert$	t	u	r	l	u	t	u	t
t	u	$\vert$	t	u	r	l	u	t	u
u	t	u	$\vert$	t	u	r	l	u	t
t	u	t	u	$\vert$	t	u	r	l	u
u	t	u	t	u	$\vert$	t	u	r	l
l	u	t	u	t	u	$\vert$	t	u	r
r	l	u	t	u	t	u	$\vert$	t	u
u	r	l	u	t	u	t	u	$\vert$	t

2) Le plus simple semble d'utiliser une fonction auxiliaire `dernier : mot -> char * mot` qui renvoie un couple `c,deb` où `c` est le dernier élément de la liste et `deb` la liste sans son dernier élément. La fonction `circulaire` s'en déduit.

```
let circulaire m =
  let rec dernier mot = match mot with
      | [] -> failwith "pas␣interessant"
      | [x] -> x, []
      | t :: q -> let c, deb = dernier q in c, (t :: deb)
  in match m with
    | [] -> []
    | _ -> let c, deb = dernier m in c :: deb;;
```

3) On utilise une fonction récursive auxilaire `construit : int` $\to$ `mot` $\to$ `mot list`. Dans l'appel `construit k mot`, on renvoie la liste de taille k contenant `mot` et les `k-1` décalages suivants de ce ce mot.

```
let matrice_mot mu =
   let rec construit k mot =
      if k=0 then []
      else mot::(construit (k-1) (circulaire mot))
   in construit (List.length mu) mu;;
```

Si on ne veut pas utiliser `List.length` on peut écrire une fonction `longueur : mot -> int`

```
let rec longueur l = match l with
    | [] -> 0
    | t :: q -> 1 + longueur q
```

4) La matrice M' est la suivante

|	t	u	r	l	u	t	u	t	u
l	u	t	u	t	u	|	t	u	r
r	l	u	t	u	t	u	|	t	u
t	u	|	t	u	r	l	u	t	u
t	u	r	l	u	t	u	t	u	|
t	u	t	u	|	t	u	r	l	u
u	|	t	u	r	l	u	t	u	t
u	r	l	u	t	u	t	u	|	t
u	t	u	|	t	u	r	l	u	t
u	t	u	t	u	|	t	u	r	l

et on a

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

5) On applique la définition de l'ordre lexicographique.

```
let rec compare mot1 mot2 = match mot1, mot2 with
    | [], _ -> true
    | _, [] -> false
    | x1 :: q1, x2 :: q2 -> (x1 < x2) || ((x1 = x2) && (compare q1 q2
       ));;
```

6) Une fonction `insere : 'a → 'a list` prend en argument un élément, une liste triée par ordre croissant et renvoie la liste triée obtenue en ajoutant l'élément à la liste. Elle sera utilisé pour le type `'a = mot`.
La fonction principale s'en déduit.

```
let rec insere x l = match l with
   |[] -> [x]
   |y::q -> if x<=y then x::l
            else y::(insere x q);;

let rec tri l = match l with
   |[] -> []
   |x::q -> insere x (tri q);;
```

7) Il suffit de trier la matrice M des mots permutés.

```
let matrice_mot_triee mu =
    tri (matrice_mot mu) ;;
```

8) On peut penser qu'il s'agit ici d'estimer la complexité de la comparaison de deux mots de taille k, c'est à dire le nombre d'opérations dans l'appel `compare m1 m2` quand `m1` et `m2` sont deux listes de même taille k.

En notant C_k ce nombre d'opérations, on a $C_k = O(1)+C_{k-1}$ et ainsi $C_k = O(k)$. La complexité est linéaire en fonction de la taille.

9) On sait que le nombre de comparaisons lors d'un tri par insertion d'une liste de k éléments est majorée par k^2. On en déduit une complexité totale en $O(k^3)$.

10) On veut une fonction `last : mot` $\to$ `char` renvoyant la dernière lettre d'un mot supposé non vide. On procède comme à la question 2.

```
let rec last l = match l with
    |[] -> failwith "erreur"
    |[x] -> x
    |x::q -> last q ;;
```

Il suffit alors de créer la matrice M' et de récupérer les dernières lettres. On écrit pour cela une fonction `parcours : mot list` $\to$ `mot` renvoyant le mot correspondant à la dernière colonne de l'argument. On suppose ici que l'argument donné à `codageBWT` est le mot $\hat{\mu}$ (c'est à dire que le symbole | a déjà été ajouté).

```
let codageBWT mu =
   let rec parcours mat = match mat with
       |[] -> []
       |mot::q ->(last mot)::(parcours q)
   in parcours (matrice_mot_triee mu);;
```

D'après la question 4, dans le cadre de l'exemple, le codage est

$$uruu|utttl$$

11) $\ell = BWT(\mu)$ contient exactement les mêmes lettres que $\hat{\mu}$ (par construction de M' où chaque lettre de $\hat{\mu}$ se retrouve une fois en dernière position d'une ligne).
Notons m la version triée de ℓ. Comme M' est triée par ordre lexicographique, m correspond alors exactement à la première colonne de M' (puisque dans cet ordre, on prend d'abord en compte la première lettre).
Dans le cadre de l'exemple, la première colonne sera

$$|adeegnnv$$

12) Comme indiqué par l'énoncé, les sous-mots de taille 2 de $\hat{\mu}$ sont alors

$$e|, da, nd, ge, ve, ng, en, an, |v$$

Pour obtenir la colonne suivante, on ordonne cette liste de mots de taille 2

$$|v, an, da, e|, en, ge, nd, ng, ve$$

A nouveau, par définition de M', on obtient là les deux premières colonnes de M'. En particulier, la seconde colonne est

$$vna|nedge$$

13) De façon générale, on suppose connues les $i-1$ première colonnes.
En ajoutant DEVANT la dernière colonne, on obtient tous les facteurs de taille i.
On ordonne ces facteurs de taille i et on obtient alors les i premières colonnes de M' (et donc la i-ième).

14) L'algorithme est donc le suivant. On prend en argument le mot $\ell = BWT(\mu)$ que l'on suppose de taille k. On initialise une liste m de taille k composée de mots tous vides.

1. Ajouter devant chaque mot de m la lettre correspondante de ℓ (ceci modifie m en ajoutant une lettre à chaque mot).
2. Trier m (i.e. remplacer m par sa version triée)
3. retourner au point 1 si les éléments de m sont de taille $\leq k$
4. Renvoyer la ligne de m se terminant par |.

15) En appliquant l'algorithme, on obtient la liste de listes suivante

```
[['|'; 'v'; 'e'; 'n'; 'd'; 'a'; 'n'; 'g'; 'e'];
 ['a'; 'n'; 'g'; 'e'; '|'; 'v'; 'e'; 'n'; 'd'];
 ['d'; 'a'; 'n'; 'g'; 'e'; '|'; 'v'; 'e'; 'n'];
 ['e'; '|'; 'v'; 'e'; 'n'; 'd'; 'a'; 'n'; 'g'];
 ['e'; 'n'; 'd'; 'a'; 'n'; 'g'; 'e'; '|'; 'v'];
 ['g'; 'e'; '|'; 'v'; 'e'; 'n'; 'd'; 'a'; 'n'];
 ['n'; 'd'; 'a'; 'n'; 'g'; 'e'; '|'; 'v'; 'e'];
 ['n'; 'g'; 'e'; '|'; 'v'; 'e'; 'n'; 'd'; 'a'];
 ['v'; 'e'; 'n'; 'd'; 'a'; 'n'; 'g'; 'e'; '|']]
```

Le mot décodé est donc

vendange

2 - Codage par plages RLE

16) On utilise une fonction auxiliaire `aux : char -> int -> mot -> (char * int) list` qui permet de parcourir la liste passée en paramètre (notée `l`) en mémorisant le dernier caractère vu dans le paramètre `c` ainsi que son nombre d'occurrences consécutives avant (noté `nb`). La fonction teste alors la tête de la liste `l` :
— Si cette tête est égal à `c` on incrémente `nb` de 1.
— Sinon, on ajoute le couple `(c,nb)` à la liste renvoyée et on rappelle la fonction mais avec `nb` qui vaut 1.

```
let rle m =
  let rec aux c nb l = match l with
      | [] -> [(c, nb)]
      | t :: q when t = c -> aux t (nb + 1) q
      | t :: q -> (c, nb) :: (aux t 1 q)
  in match m with
  |[] -> []
  | t :: q -> aux t 1 q;;
```

17) Il suffit de parcourir la liste et d'ajouter les caractères au fur et à mesure.

```
let rec decodeRLE m =
  match m with
    | [] -> []
    | (c, 0) :: q -> decodeRLE q
    | (c, nb) :: q -> c :: (decodeRLE ((c, nb - 1) :: q));;
```