# Table des matières



## Inversion de permutations

Pour tout $n \in N \setminus \{0\}$, une permutation de taille $n$ est une bijection de l'ensemble $\{0, 1, \cdots, n-1\}$ dans lui-même. Dans la suite, l'ensemble des permutations de taille $n$ est noté $\mathfrak{S}_n$ . Étant donné une permutation de taille n, on la représente sous la forme $\sigma_0\sigma_1 \cdots \sigma_{n-1}$ où :

$$\forall i \in \{0, \cdots, n-1\}, \sigma_i = \sigma(i).$$

Par exemple, $\sigma = 1032$ représente la permutation envoyant 0 sur 1, 1 sur 0, 2 sur 3 et 3 sur 2.

Soit $\sigma = \sigma_0 \cdots \sigma_{n-1}$ une permutation de taille $n$. On dit que $(i, j) \in \{0, 1, \cdots, n-1\}^2$ est une inversion de $\sigma$ si $\sigma_i > \sigma_j$ et $i < j$. On note $inv(\sigma)$ le nombre d'inversions de $\sigma$. Tout d'abord, on relie ce nombre avec un algorithme de tri : le tri à bulles. Puis, on s'intéresse à la table d'inversions d'une permutation, un objet qui caractérise une permutation.

Dans la suite, une permutation de taille n est représentée en Python par une liste sans répétition conte- nant tous les entiers compris entre 0 et $n-1$. Par exemple, la liste `[1, 0, 3, 2]` représente la permutation $\sigma = 1032$.

Si A est un ensemble fini, Card(A) désigne le cardinal de l'ensemble A. Pour tout entier n 1, on pose $En = \Pi_{k=1}^{n}\{0, 1, \cdots, n-k\}$. Par exemple, on a $E3 = \{0, 1, 2\} \times \{0, 1\} \times \{0\}$.

### 0.1 Tri et inversions

1. Déterminer l'ensemble des inversions de la permutation $\sigma = 140253$.
   *solution :* On trouve 5 inversions :$(0, 2)$, $(1, 2)$, $(1, 3)$, $(1, 5)$ et $(4, 5)$.

   On rappelle l'algorithme de tri à bulles :

**Entrées :** Une liste d'entiers L
**pour** *i allant de (taille de L)-1 à 1 (bornes incluses)* **:**
  **pour** *j allant de (taille de L)-1 à (taille de L)-i (bornes incluses)* **:**
    **si** *L[j]<L[j-1]* **:**
      Echanger L[j] et L[j-1]
    **fin**
  **fin**
**fin**

2. Écrire une fonction Python `tri_bulle(L)` qui prend en argument une liste d'entiers L et qui trie cette liste à l'aide du tri à bulles. À l'issue de la fonction, la liste est triée.
   *solution :*

```
def nombre_inversions(L):
    n=len(L)
    for i in range(n):
        for j in range(n-1, n-i-1):
            if L[j]<L[j-1]:
                L[j], L[j-1] = L[j-1], L[j]
    return nb_i
### σ□σ
```

3. Montrer que si on effectue exactement un échange dans une liste lors du tri à bulles, la nouvelle liste a exactement une inversion en moins.
*solution :* Soit $j$ la valeur du compteur de la boucle interne au moment où on fait un échange. Soit $\sigma$, respectivement $\sigma'$, la permutation représentée par `L` juste avant, respectivement juste après cet échange.

   - Déja, $(j-1, j)$ est une inversion dans $\sigma$ et plus dans $\sigma'$.
   - Soit $(k, l) \in [\![0, n[\![^2$ tel que $k < l$ et $(k, l) \neq (j-1, j)$
     - ◇ Si $\{k, l\} \cap \{j-1, j\} = \emptyset$, $\sigma(k)$ et $\sigma(l)$ ne sont pas changés donc $(k, l)$ est une permutation dans $\sigma$ ssi c'en est une dans $\sigma'$.
     - ◇ Si $l = j$ alors $k < j-1$. Et $(k, j)$ est une inversion dans $\sigma$ ssi $(k, j-1)$ en est une dans $\sigma'$.

   On voit donc que hormis $(j-1, j)$, il y autant d'inversion dans $\sigma$ que dans $\sigma'$. Pour une justification précise : la fonction $\varphi : \begin{array}{|l} (k,j) \mapsto (k,j-1) \\ (k,j-1) \mapsto (k,j) \\ (j,l) \mapsto (j-1,l) \\ (j-1,l) \mapsto (j,l) \\ (k,l) \mapsto (k,l) \end{array}$ est une bijection entre l'ensemble des inversions de $\sigma$ hormis $(j-1, j)$ et celles de $\sigma'$.

4. En déduire une fonction Python `nombre_inversions(L)` qui prend en argument une liste `L` correspondant à une permutation et renvoyant le nombre d'inversions de celle-ci. Cette fonction sera une légère modification du tri à bulles.
*solution :*

```
def nombre_inversions(L):
    n=len(L)
    nb_i=0 # nb d'inversions détectées
    for i in range(n-1,0,-1):
        for j in range(n-1, n-i-1,-1):
            # Invariant de boucle : nb_i + nb d'inversion de L == nb d'inversions de la
                ↪ liste initiale
            if L[j]<L[j-1]:
                L[j], L[j-1] = L[j-1], L[j]
                nb_i +=1
    return nb_i
### σ□σ
```

## 0.2 Table d'inversions d'une permutation

**Définition 0.1.** ***(Table d'inversions).*** *Soit $\sigma$ une permutation de taille $n \geq 1$. La table d'inversions de $\sigma$ est le $n$-uplet $(\alpha_0, \cdots, \alpha_{n-1})$ tel que :*

$$\forall i \in 0, \cdots, n-1, \alpha_i = Card\left(\left\{ j \in \{i+1, \cdots, n-1\} \middle| \sigma_j < \sigma_i \right\}\right)$$

*Elle est notée $\boldsymbol{Tab}_\sigma$, De plus pour tout $i \in [\![0, n[\![$, $\boldsymbol{Tab}_\sigma[i]$ désigne $\alpha_i$.*
*Pour tout $n \in N^*$, on désigne par $\boldsymbol{Tab} : \mathfrak{S}_n \to E_n$ l'application qui à une permutation de taille $n$ associe sa table d'inversions.*

*solution :* Donc $\alpha_i$ est le nombre de $j > i$ en inversion avec $i$.

1. Déterminer la table d'inversions de la permutation $\sigma = 140253$.
*solution :* `[1, 3, 0, 0, 1, 0]`
2. Montrer que pour toute permutation $\sigma$ de taille $n \geq 1$, $\mathbf{Tab}_\sigma$ est bien un élément de $E_n$ .
*solution :* Soit $i \in [\![0, n[\![$. Par définition $\alpha_i$ est le nombre d'entiers $j \in [\![i+1, n[\![$ en inversant avec $i$. Ce nombre est donc au plus Card($[\![i+1, n[\![$), soit $n-i-1$.
Ainsi, $(\alpha_0, ..., \alpha_{n-1}) \in \prod_{i=0}^{n-1} [\![0, n-i-1]\!]$, qui est bien $E_n$ (faire un décalage d'indice par rapport à la définition).
3. Soit $\sigma$ une permutation de taille $n \geq 1$. Montrer que $\sigma_0 = \mathbf{Tab}_\sigma[0]$.
*solution :* Soit $j \in [\![1, n[\![$.

$$\begin{aligned} (0, j) \text{ est une inversion } &\Leftrightarrow \sigma_j < \sigma_0 \\ &\Leftrightarrow \sigma_j \in [\![0, \sigma_0[\![ \\ &\Leftrightarrow j \in \sigma^{-1}\left([\![0, \sigma_0[\![\right) \end{aligned}$$

Ainsi $\alpha_0 = \text{Card}\left(\sigma^{-1}\left([\![0,\sigma_0[\![\right) \cap [\![1,n[\![\right)$.

Or $\sigma^{-1}\left([\![0,\sigma_0[\![\right)$ ne contient pas 0 qui est $\sigma^{-1}(\sigma_0)$). Donc $\sigma^{-1}\left([\![0,\sigma_0[\![\right) \cap [\![1,n[\![ = \sigma^{-1}\left([\![0,\sigma_0[\![\right)$.

Et, $\text{Card}\left(\sigma^{-1}\left([\![0,\sigma_0[\![\right)\right) = \text{Card}([\![0,\sigma_0[\![) = \sigma_0$, car $\sigma$ est une bijection.

D'où $\alpha_0 = \sigma_0 = \textbf{Tab}[0]$.

4. Montrer que pour tout entier $n \geq 1$, l'application **Tab** est bijective.
*solution :*

Déjà, $\text{Card}(E_n) = n! = \text{Card}(\mathfrak{S}_n)$. Par conséquent, il suffit de prouver que $\sigma$ est injective (ou surjective) pour déduire qu'elle est bijective.

La question précédente montre que connaissant $\textbf{Tab}[0]$ on peut récupérer $\sigma_0$. Ceci nous permet de prouver l'injectivité par récurrence.

Soit $n \in \mathbb{N}^*$, soit $(\sigma, \sigma') \in \mathfrak{S}_n^2$ telles que $\textbf{Tab}_\sigma = \textbf{Tab}_{\sigma'}$.
Pour tout $k \in [\![0,n[\![$, soit $P(k)$ : «$\sigma_k = \sigma'_k$».

- $P(0)$ est vrai par la question précédente.
- Soit $i \in [\![1,n[\![$ tel que $P(0), ..., P(i-1)$. En reprenant la calcul de la question précédente :

$$\begin{aligned}
\alpha_i =& \text{Card}\left(\sigma^{-1}\left([\![0,\sigma_i[\![\right) \cap [\![i+1,n[\![\right) \\
=& \text{Card}\left([\![0,\sigma_i[\![ \cap \sigma\left([\![i+1,n[\![\right)\right) \\
=& \text{Card}\left([\![0,\sigma_i[\![ \cap \sigma\left([\![i,n[\![\right)\right) \qquad \text{car } \sigma_i \notin [\![0,\sigma_i[\![ \\
=& \text{Card}\left([\![0,\sigma_i[\![ \setminus \sigma\left([\![0,i[\![\right)\right)
\end{aligned}$$

Et nous avons la même formule pour $\sigma'$ : $\alpha_i = \text{Card}\left([\![0,\sigma'_i[\![ \setminus \sigma'\left([\![0,i[\![\right)\right)$. De plus, par hypothèse de récurrence, $\sigma'\left([\![0,i[\![\right) = \sigma\left([\![0,i[\![\right)$.

Maintenant, remarquons que grâce à cette formule, connaissant $\alpha_i$ on peut déduire $\sigma_i$ : prendre une variable `x` qui part de 0, augmenter de 1 en 1 en sautant les valeurs dans $\sigma([\![0,i[\![)$, jusqu'à ce que $\text{Card}\left([\![0,x[\![ \setminus \sigma\left([\![0,i[\![\right)\right) = \alpha_i$.

Pour une démonstration complète : je note $f : x \mapsto \text{Card}\left([\![0,x[\![ \setminus \sigma\left([\![0,i[\![\right)\right)$. Cette fonction est strictement croissante, et donc injective sur $[\![0,n[\![ \setminus \sigma\left([\![0,i[\![\right)$. En outre, $f(\sigma_i) = \sigma'_i$ (et $\sigma_i$ et $\sigma'_i$ sont bien dans $[\![0,n[\![ \setminus \sigma\left([\![0,i[\![\right)$). J'en déduis que $\sigma_i = \sigma'_i$.

Ainsi, par récurrence, pour tout $i \in [\![0,n[\![$, $\sigma_i = \sigma'_i$, et donc $\sigma = \sigma'$.

5. Écrire une fonction Python `permutation_vers_table(L)` qui prend en argument une permutation représentée par la liste `L` et qui renvoie la table d'inversions correspondante.
*solution :*

```
def permutation_vers_table(L):
    n=len(L)
    res = [0 for i in range(n)]
    for i in range(n):
        for j in range(i+1,n):
            if L[i]>L[j]:
                res[i]+=1
    return res
### σ□σ
```

6. Écrire une fonction Python `table_vers_permutation(L)` qui prend en argument une liste `L` qui correspond à une table d'inversions et qui renvoie la permutation qui lui est associée.
*solution :*

```
def table_vers_permutation(L):
    n=len(L)
    res = [-1 for i in range(n)]
    déjà_pris = [False for k in range(n)] #Indique les élément dans σ([0,i[)
    for i in range(n):
        alphai = L[i] # c'est juste pour avoir les mêmes notations que dans la preuve
        x=0
        card=0 # card( [0,x[ □σ ([0,i[) )
        while card < alphai or déjà_pris[x]:
            if not déjà_pris[x]:
                card+=1
```

```
            x+=1

        res[i]=x
        déjà_pris[x]=True

    return res
###
```

L'objectif de ce problème est de construire différents algorithmes pour vérifier si un mot comporte des facteurs carrés ou non.

Dans toute la suite,$\Sigma = \{a_1 < \cdots < a_p\}$ désigne un alphabet totalement ordonné comportant $p$ lettres, $\varepsilon$ représente le mot vide et $\Sigma^*$est l'ensemble des mots finis obtenus à partir de $\Sigma$. Pour tout réel $x$, on note $\lfloor x \rfloor$ la partie entière de $x$.

# 1 Définitions

**Définition 1.1.**
- *Soit $w = w_0 \cdots w_{n-1}$ un mot de $\Sigma^*$ . La longueur $n$ de $w$ est notée $|w|$, pour tout $0 \leq i \leq j < n$, $w[i, j]$ désigne le mot $w_i \cdots w_j$ . Par convention, si $j < i$, $w[i, j]$ désigne $\varepsilon$.*
- *Soit $w$ un mot de $\Sigma^*$ . On dit que $w$ est un* carré *s'il existe un mot $x$ tel que $w = x \cdot x$.*
- *Soient $v$ et $w$ deux mots de $\Sigma^*$ . On dit que $v$ est un* facteur *de $w$ s'il existe $r$ et $s$ deux mots (éventuellement vides) tels que $w = rvs$.*
- *On dit qu'un mot $w$ contient une répétition s'il contient un facteur carré différent de $\varepsilon$.*

*solution :*

Dans la suite, un mot sera représenté en Caml par la liste de ses lettres. Par exemple, le mot « baba » est représenté par la liste `[ ' b ' ; ' a ' ; ' b ' ; ' a ' ]` et le mot vide est représenté par la liste `[ ]`.

# 2 Fonctions utiles sur les listes

1. Écrire une fonction récursive Caml de signature longueur : `'a list -> int` qui renvoie la longueur de la liste.
*solution :*

```
let rec longueur = function
  |[] -> 0
  |_::q -> 1 + longueur q
;;
```

2. Écrire une fonction Caml de signature `sous_liste : 'a list -> int -> int -> 'a list` où `sous_liste l k long` renvoie une liste `S` qui est la sous-liste de `l` commençant à l'indice $k$ et de longueur `long`. On suppose que l'indexation des listes commence à 0.
*solution :*

```
let rec sous_liste l k long =
  (* Renvoie la sous-liste de l de longueur long débutant à l'indice k*)
  (* Si long est trop petit, renvoie la sous-liste débutant à l'indice k et finissant au
     ↪ bout du mot.*)
  if long=0 then []
  else
    match l with
    |[] -> []
    |t::q when k=0 -> t::sous_liste q 0 (long-1)
    |t::q -> sous_liste q (k-1) long
;;
```

On pourra dans la suite de l'énoncé utiliser les fonctions `longueur` et `sous_liste`.

# 3 Un algorithme naïf

3. Préciser si les mots suivants contiennent ou non une répétition :

(a) aabfa (b) abfdanq (c) ababa (d) avba

*solution :*

(a) aabfa contient le facteur carré aa.

(b) La seule lettre qui se répète est le a, mais les deux ne sont pas côte à côte et ne forment donc pas un facteur carré.

(c) abab est un facteur carré dans ababa.

(d) Même raisonement que pour abfdanq : pas de facteur carré.

4. Soit $w$ un mot contenant au plus deux lettres différentes. Montrer que si $|w| \geq 4$ alors $w$ contient au moins une répétition.
*solution :* Notons $a$ la première lettre de $w$. Si $w$ contient seulement des $a$ c'est évident. Sinon, soit $b$ l'autre lettre utilisée. Si $w$ contient $aa$ ou $bb$ c'est réglé. Sinon, les seuls facteurs le longueur 2 possibles sont $ab$ et $ba$. Alors $w$ commence par $abab$, qui est un facteur carré.

5. Écrire une fonction Caml de signature `estCarre : 'a list -> bool` prenant en argument une liste `w` et retournant `true` si `w` est un carré et `false` sinon.
*solution :*

```
let estCarre l =
  let n = longueur l in
  if n mod 2 <> 0 then false
  else
    let u = sous_liste l 0 (n/2) and v = sous_liste l (n/2) (n/2) in (* Ces deux appels
        ↪ ne sont pas optimaux *)
    u=v
;;
```

6. Déterminer la complexité en nombre de comparaisons de lettres de la fonction `estCarre`.
*solution :* Soit `l` une liste et $n$ sa longueur. Les appels à `longueur` et à `sous_liste` n'effectuent pas de comparaisons de lettres et donc ne sont pas comptabilisés ici. Reste le « `u=v` » final, qui nécessite $n/2$ comparaisons de lettres. D'où une complexité en $O(n)$.

7. Écrire une fonction Caml de signature `contientRepetitionAux : 'a list -> int -> bool` prenant en argument une liste $w$ et un entier $m$ et retournant `true` si $w$ contient une répétition de la forme $xx$ avec $x$ de longueur $m$ et `false` sinon.
*solution :*

```
let rec contientRepetitionAux l m =
  match l with
  |[] -> false
  |t::q -> if estCarre (sous_liste l 0 (2*m)) then true
           else contientRepetitionAux q m
;;
(* J'aurais pu optimiser légèrement en passant la longueur en argument et en arrêtant
    ↪ lorsque celle-ci est < 2m *)
```

8. Montrer que toute répétition d'un mot $w$ de longueur $n$ est de la forme $xx$ avec $|x| \leq \frac{n}{2}$.
*solution :* Soit $u$ une répétition dans $w$. Vu les définitions, cela signifie qu'il existe trois mots $p$, $x$ et $s$ tel que $w = pxxs$. Alors $n = |w| = |p| + 2|x| + |s|$ d'où $|x| \leqslant \frac{n}{2}$.

9. En déduire une fonction Caml de signature `contientRepetition : 'a list -> bool` prenant en argument une liste $w$ retournant `true` si $w$ contient une répétition et `false` sinon.
*solution :*

```
let rec contientRepetition l=
  let n = longueur l in
  let rec boucle m=
    if m=0 then false
    else contientRepetitionAux l m || boucle (m-1)
  in
  boucle (n/2)
;;
```

10. Quelle est la complexité en nombre de comparaisons de caractères de la fonction `contientRepetition` ?
*solution :* Soit $l$ une liste et $n$ sa longueur.
    - Soit $m$ un entier. L'exécution de |contientRepetitionAux l m| appelle `estCarre` sur chaque préfixe de $l$ de taille $2m$, ce qui coûte $O(nm)$ comparaisons de lettres.
    - Lorsqu'on exécute `contientRepetition l`, la boucle appelle la fonction auxiliaire pour tout $m$ entre 0 et $\lfloor n/2 \rfloor$, ce qui coûte $\sum_{m=0}^{\lfloor n/2 \rfloor} O(nm)$, qui vaut $O\left(n \sum_{m=0}^{\lfloor n/2 \rfloor} m\right)$, soit $O(n^3)$.

# 4 Algorithme de Main-Lorentz

L'algorithme de Main-Lorentz permet de détecter de manière plus efficace des répétitions d'un mot $w$. Il comporte essentiellement deux parties :

- la première consiste à voir si étant donné deux mots $u$ et $v$, le mot $uv$ contient un carré non nul issu de la concaténation ;
- la deuxième s'appuie sur le principe de « diviser pour régner ».

*solution :* Le premier point me semble inclus dans le second : c'est une partie du «régner».

Remarquons qu'un mot $uv$ contient une répétition si et seulement si $u$ ou $v$ contiennent une répétition ou $uv$ contient des répétitions provenant de la concaténation. Pour déterminer si un mot $uv$ contient de nouvelles répétitions, on commence par effectuer des prétraitements consistant à calculer des tables de valeurs de $u$ et de $v$ qui sont généralement appelées tables de préfixes (ou suffixes). Avant de présenter des algorithmes permettant de générer ces tables, on commence par justifier leur application dans la détection de répétitions.

**Définition 4.1.** *Soient $u$ et $v$ deux mots. On dit que $uv$ contient un carré centré sur $u$ (respectivement sur $v$) s'il existe un mot $w$ non vide et des mots $u'$ , $v'$ , $w'$ , $w''$ tels que $u = u'ww'$ , $v = w''v'$ , et $w = w'w''$ (respectivement $u = u'w'$ , $v = w''wv'$ , et $w = w'w''$ ).*

*solution :* Le carré est dit centré sur $u$ si $u$ contient en entier le mot répété, plus le début de la deuxième occurrence. Voici un schéma :

$$\underbrace{u'(w'w'')(w'}_{u}\ \underbrace{w'')v'}_{v}$$

**Définition 4.2.** *Soient $u$ et $v$ deux mots sur $\Sigma$. Le plus long préfixe (respectivement suffixe) commun de $u$ et $v$ est le plus long mot $w$ tel qu'il existe deux mots $r$ et $s$ tels que $u = wr$ et $v = ws$ (respectivement $u = rw$ et $v = sw$). On le note $lcp(u, v)$ (respectivement $lcs(u, v)$).*

## 4.1 À propos des carrés centrés

11. Dans cette question, $\Sigma = \{a, b\}$. Soient $u = abababaa$ et $v = ababaaa$. Déterminer le plus grand préfixe commun de $u$ et $v$.
*solution :* *ababa*
12. Soit $(u, v) \in (\Sigma^*)^2$. Montrer que $uv$ contient un carré centré sur $u$ ssi il existe $i \in [\![0, |u|[\![$ tel que

$$\big|lcs(u[0, i-1], u)\big| + \big|lcp(u[i, |u|-1], v)\big| \geqslant |u| - i$$

*solution :*

- Supposons que $uv$ contient un carré centré sur $u$. Reprenons les notations de la définition, de sorte que $\underbrace{u'(w'w'')(w'}_{u}\ \underbrace{w'')v'}_{u}$.
  Posons $i = |u'w'|$.
  Le mot $w'$ est un suffixe commun à $u[0, i-1]$ et à $u$, donc $\big|lcs(u[0, i-1], u)\big| \geqslant |w'|$.
  De même, $w''$ est un préfixe commun à $u[i, :]$ (notation à la Python...) et à $v$, donc $\big|lcp(u[i, |u|-1], v)\big| \geqslant |w''|$.
  Au total, $\big|lcs(u[0, i-1], u)\big| + \big|lcp(u[i, |u|-1], v)\big| \geqslant |w'| + |w''| = |u| - |u'| - |w'| = |u| - i$.
- Réciproquement, supposons qu'il existe $i \in [\![0, |u|[\![$ tel que $\big|lcs(u[0, i-1], u)\big| + \big|lcp(u[i, |u|-1], v)\big| \geqslant |w'| + |w''| = |u| - i$, fixons un tel $i$.
  Notons alors $w'$ le plus long suffixe commun de $u[0, i-1]$ et $u$, et $u'$ le «reste» de $u[0, i-1]$ (donc $u[0, i-1] = u'w'$).

Notre hypothèse est que $\left|lcp(u[i,|u|-1],v)\right| \geqslant |u|-|w'|-i$. Prenons alors pour $w''$ un préfixe commun à $u[i,:]$ et $v$ de longueur *égale* à $|u|-|w'|-i$. Et notons $v'$ le reste de $v$.
Maintenant, il faut remarquer que le mot $u[i,:]$ admet $w''$ comme préfixe, et $w'$ comme suffixe, et que $|u[i,:]| = |w'|+|w''|$. Par conséquent, $u[i,:] = w''w'$. Il vient ensuite $u = u'w'w''w'$, et comme $v = w''v'$, $uv$ a bien un facteur caré centré en $u$.

De la même manière, on peut montrer que $uv$ contient un facteur carré centré sur $v$ ssi il existe $j \in [\![1,|v|[\![$ tel que

$$\left|lcs(v[0,j-1],u)\right| + \left|lcp(v,v[j,|v|-1])\right| \geqslant |v|-j,$$

Ainsi pour déterminer s'il existe un carré centré sur $u$ ou sur $v$, on peut utiliser les quatre valeurs :

$$\left|lcs(v[0,j-1],u)\right|, \quad \left|lcp(v,v[j,|v|-1])\right|, \quad \left|lcs(u[0,i-1],u)\right|, \quad \left|lcp(u[i,|u|-1],v)\right|$$

Dans la suite étant donnés deux mots $u$ et $v$, on note $\text{pref}_u$, $\text{pref}_{u.v}$, $\text{suff}_u$, $\text{suff}_{u,v}$ les tableaux tels que pour tout $i \in [\![0,|u|]\!]$,

- $\text{pref}_u[i] = \left|lcp\left(u[i:],u\right)\right|$ ;
- $\text{pref}_{u,v}[i] = \left|lcp\left(u[i:],v\right)\right|$ ;
- $\text{suff}_u[i] = \left|lcs(u[0:i],u)\right|$ ;
- $\text{suff}_{u,v}[i]\left|lcs(u[0:i],v)\right|$.

**N.B.** Par rapport à l'énoncé initial, j'ai mis ici les notations à la Python, et j'ai changé un peu la définition des suff, ça me paraît plus pratique ainsi. En outre, l'énoncé définissait ceci pour $u \in [\![0,|u|[\![$ mais il me semble plus cohérent de fermer l'intervalle. Sachant que de toute façon $\text{pref}_u[n] = 0$ et $\text{suff}_u[n] = n$. Cela nécessite de prendre un tableau `pref` avec une case de plus dans l'algo 1. Cela permet une réponse plus élégante à la question 3

## 4.2 Calcul de table de préfixes

L'algorithme 1 permet le calcul de $\text{pref}_u$ en $O(|u|)$ de comparaisons de caractères. En adaptant cet algorithme, il est également possible de calculer la table $\text{pref}_{u,v}$.

**Entrées :** une chaîne de caractère $u$
**Sorties :** un tableau $\text{pref}_u$

```
1  i ← 0
2  pref ← tableau de taille |u| initialisé à 0
3  pref[i] ← |u|
4  g ← 0
5  pour i de 1 à |u| − 1 :
6  |  si i < g et pref[i − f] < g − i :
7  |  |  pref[i] ← pref[i − f]
8  |  sinon si i < g et pref[i − f] > g − i :
9  |  |  pref[i] ← g − i
10 |  sinon :
11 |  |  (f, g) ← (i, max(g, i))
12 |  |  tant que  g < |u| et u[g] == u[g − f] :
13 |  |  |  g ← g + 1
14 |  |  fin
15 |  |
16 |  |  pref[i] ← g − f
17 fin
```

**Algorithme 1 :** Calcul de la table $\text{pref}_u$

*solution :* La preuve de la complexité n'est pas demandée, cependant je constate qu'à chaque itération de la boucle «tant que», $g$ augmente de 1. De plus, vu la condition de cette boucle, il ne peut dépasser $|u|$. À ce stade, $|u| - g$ est un variant de boucle et donc le programme termine. Mais en plus, entre deux exécutions de la boucle «tant que», $g$ n'est pas diminué. De sorte que le nombre de tours de la boucle «tant que», au total sur toute l'éxecution de l'algorithme est au plus $|u|$.

*solution :* **Explication de l'algo :** Ça peut paraître magique au premier abord, et rien n'est fait pour faire comprendre dans l'énoncé à part traiter l'exemple....
Je reprends les notations à la Python avec la borne finale *exclue* parce que les $-1$ partout ça va bien deux minutes.

- *Invariant de boucle* : à la fin de chaque itération de la boucle pour, $u[f:g] = u[0:g-f]$ et $u[g] \neq u[g-f]$ (à condition que $u[g]$ existe).
- Premier cas : $f < i < g$ et $pref[i-f] < g-i$. Soit $k = pref[i-f]$.
  On a $u[0:k] = u[i-f:i-f+k]$ par définition de $pref[i-f]$.
  Et $u[i-f:i-f+k] = u[i:i+k]$ vu l'invariant de boucle et car $u[i:i+k] \subset u[f:g]$.
  D'où $u[0:k] = u[i:i+k]$.
  Enfin, encore par définition de $pref[i-f]$, $u[k] \neq u[i-f+k]$. Mais par notre invariant $u[i-f+k] = u[i+k]$, et car $f \leqslant i+k < g$. D'où $u[k] \neq u[i+k]$. Et donc $u[0:k]$ est le plus grand préfixe commun à $u$ et $u[i:]$.
- Deuxième cas $f < i < g$ et $pref[i-f] > g-i$.
  On a $u[i:g] = u[i-f:g-f]$ par l'invariant de boucle. Puis $u[i-f:g-f] = u[0:g-i]$ car $u[i-f:g-f]$ est un préfixe de $u[i-f:]$ de longueur $< pref[i-f]$, il est donc inclus dans le plus grand préfixe à $u[i-f:]$ et $u$.
  Ainsi, $u[i:g]$ est un préfixe commun à $u[i:]$ et à $u$.
  En outre, nous savons que $u[g] \neq u[g-f]$ par l'invariant de boucle. Comme $pref[i-f] > g-i$, nous savons que $u[g-i] = u[g-i+i-f]$ ($u[g-i]$ est dans le plus grand préfixe commun à $u$ et $u[i+f:]$). Autrement dit $u[g-f] = u[g-i]$. En combinant le tout, $u[g] \neq u[g-i]$.
  En conclusion, $u[i:g]$ est le plus grand préfixe commun à $u$ es $u[i:]$.
- Troisième cas : $i = g$ ou $pref[i-f] = g-i$.
  - ◇ Le cas ou $i = g$ est clair, car alors la boucle «tant que» qui suit fait exactement en sorte de satirfaire notre invariant de boucle. Et une fois cet invariant de boucle vérifié on a bien $pref[i] = g-f$.
  - ◇ Enfin suposons $pref[i-f] = g-i$. Comme dans les cas ci-dessus, on voit que $u[i:g] = u[0:g-i]$. (En revanche, on ne sait pas si $u[g] = u[g-i]$, c'est pourquoi on ne peut pas conclure sur $pref[i]$. )
    Après avoir effectué $f \leftarrow i$, on a donc $u[f:g] = u[0:g-f]$. et après la boucle «tant que», l'invariant de boucle est bien satisfait. Et comme dit ci-dessus, on a donc $pref[i] = g-f$.

*Remarque :* Dans les deux premiers cas, $f$ et $g$ ne bougent pas, il est donc inutile de vérifier que l'invariant de boucle reste vrai.

1. On pose $u = aabbba$ et $v = abbaab$. Déterminer les tableaux $\text{pref}_u$ et $\text{pref}_{u,v}$ sans justification.
   *solution :* $\text{pref}_u = [\![6;1;0;0;0;1]\!]$, $\text{pref}_{u,v} = [\![1;3;0;0;0;1]\!]$.
2. En déroulant l'algorithme 1 appliqué au mot u = aaabaaabaaab, compléter le tableau

| $i$ | $f$ | $g$ | pref[$i$] |
|---|---|---|---|
| 0 | _ | 0 | 12 |
| 1 | 1 | 3 | 2 |
| 2 | · | · | · |
| ⋮ | ⋮ | ⋮ | ⋮ |
| 11 | 4 | 12 | 0 |

   Par exemple, à l'initialisation, $i = 0$, $f$ n'est pas définie, $g$ vaut 0 et pref[0] = 12. Pour $i = 1$, à l'issue des instructions internes à la boucle, on a $f = 1$, $g = 3$, pref[1] = 2.

   *solution :*

| $i$ | $f$ | $g$ | pref[$i$] |
|---|---|---|---|
| 0 | _ | 0 | 12 |
| 1 | 1 | 3 | 2 |
| 2 | 2 | 3 | 1 |
| 3 | 3 | 3 | 0 |
| 4 | 4 | 12 | 8 |
| 5 | 4 | 12 | 2 |
| 6 | 4 | 12 | 1 |
| 7 | 4 | 12 | 0 |
| 8 | 4 | 12 | 4 |
| 9 | 4 | 12 | 2 |
| 10 | 4 | 12 | 1 |
| 11 | 4 | 12 | 0 |
| 12 | 12 | 12 | 0 |

   J'ai poussé l'algo un cran plus loin comme déjà expliqué. J'obtiens pref[12] = 0, ce qui est cohérent puisque $u[12:]$ est vide.

3. Déduire de l'algorithme 1 un algorithme calculant $\text{suff}_u$.
   *solution :* Pour tout mot $m$, je note $m^T$ l'image miroir de $m$. (Implémenté en Caml par `List.rev`.)
   Soit $n = |u|$. On a pour tout $i \in [\![0,n[\![$, $(u^T).[i] = u.[n-1-i]$.
   Soit $m \in \Sigma^*$. C'est un suffixe commun de $u$ et de $u.[:i]$ ssi $m^T$ est un préfixe commun de $u^T$ et de $(u^T).[n-i:]$. Par conséquent, $suff[i] = pref_{u^T}.[n-i]$.
   D'où l'algorithme :
   *Remarque :* Comme dit plus haut, je suppose ici que `pref` est un tableau de longueur $n+1$, autrement dit je définis $\text{pref}_u[i]$ pour $i \in [\![0,|u|]\!]$. Sinon, remplir à la main la case suff[0] par 0.

**Entrées :** Un mot $u$
**Sorties :** La table $\text{suff}_u$

```
n ← |u|
pref_u_miroir ← résultat de l'algo 1 appliqué à u^T
suff ← nouveau tableau de longueur n+1 initialisé à -1
pour i de 0 à n :
    suff[i] ← pref_u_miroir[n-i]
fin
Renvoyer suff
```

**Algorithme 2 :** tabSuff1, l'algo de la question 3

Dans la suite, on suppose qu'une fonction `tabpref(u,v)` qui prend en argument deux chaînes de caractères $u$ et $v$ et qui renvoie la table $\text{pref}_{u,v}$ nous est donnée. On admet que la complexité de cette fonction est de $O(|u|)$ en nombre de comparaisons de caractères.

4. Déduire des questions précédentes un algorithme qui, étant donnés deux mots $u$ et $v$, renvoie VRAI s'il existe un carré centré sur $u$ et FAUX sinon.
*solution :* J'appelle `tabsuff1` la fonction décrite par l'algorithme 1. On applique simplement la partie 4.1.

**Entrées :** Deux mots u et v
**Sorties :** Le booléen « $uv$ contient un facteur carré centré sur $u$»

```
pref_uv ← tabpref(u,v)
suff_u ← tabsuff1(u)
res ← Faux
n ← |u|
pour i de 0 à n − 1 :
    res ← res ou suff_u[i] + pref_uv[i] ⩾ n − i
fin
Renvoyer Vrai
```

**Algorithme 3 :** `contient_carré_centré_sur_u`, réponse à la question 4

5. Quelle est la complexité de cet algorithme en nombre de comparaisons de caractères ?
*solution :*
   - Le calcul des deux tableaux `pref_uv` et `suff_u` se fait en $O(|u|)$.
   - La boucle coûte aussi $O(|u|)$

   L'algo en entier est donc en $O(|u|)$.

## 4.3 Application des tables

6. Déduire des questions précédentes un algorithme récursif qui prend en argument une chaîne de caractères et qui renvoie VRAI si la chaîne contient une répétition et FAUX sinon.
*solution :*

   ⦚ Du bon vieux diviser pour régner.

   J'appelle `contient_carré` cette fonction. J'appelle `contient_carré_centré_sur_u` la fonction de la question précédente, et je suppose connue la fonction analogue `contient_carré_centré_sur_v`.

**Entrées :** Une chaîne de caractères $m$
**Sorties :** Le booléen «$m$ admet un facteur carré».

```
si |m| ⩽ 1 :
    Renvoyer vrai
sinon :
    Soient u et v tels que m = uv et ||u| − |v|| ⩽ 1
    Renvoyer contient_carré(u) ou contient_carré(v) ou contient_carré_centré_sur_u(u,v) ou
     contient_carré_centré_sur_v(u,v)
fin
```

**Algorithme 4 :** `contient_carré`, réponse à la question 6

7. Déterminer la complexité de cet algorithme en nombre de comparaisons de caractères.
   *solution :* Pour tout $n \in \mathbb{N}$, soit $C_n$ le nombre maximal de comparaisons de chaînes de caractères lors du calcul de `contient_carré` appliqué à une chaîne de longueur au plus $n$. Ainsi, $C_0 = 0$, $C_1 = 0$, et pour tout $n \in [\![2, \infty[\![$, $C_n = C_{\lfloor n/2 \rfloor} + C_{\lfloor \frac{n+1}{2} \rfloor} + O(n)$. (les deux premiers termes viennent des appels récursifs, et le $O(n)$ de `contient_carré_centré_sur_u(u,v)` et `contient_carré_centré_sur_v(u,v)`.)
   On reconnaît la relation de récurrence du tri par partition-fusion, on sait d'après le cours qu'on a alors $C_n = O(n \log)$.