DS 1

CCINP MPI 2023, MP 2018 & 2002

MPI/MPI*, lycée Faidherbe

I CCINP MP 2018, parties I et II du problème

Nous proposons dans cette partie d'étudier une méthode de compression de données. L'algorithme proposé ici implémente plusieurs couches d'arrangement de données et de compressions successives, utilisées dans l'ordre suivant pour la compression et l'ordre inverse pour la décompression :

1. Transformation de Burrows-Wheeler,optimisation
2. Codage par plages (RLE),
3. Codage de Huffman

Soit Σ un alphabet de symboles, de cardinal $|\Sigma| = h$.
On munit Σ d'une relation d'ordre notée $\leqslant$.
Σ^k est l'ensemble des mots de longueur k construits à partir de Σ.
Σ^k est muni de la relation d'ordre lexicographique induite par la relation d'ordre $\leqslant$.
Pour $\mu \in \Sigma^k$, on note $|\mu| = k$ la taille de μ et $|\mu|_a$ le nombre d'occurrences de $a \in \Sigma$ dans μ.
optimisation Dans toute cette partie, lorsqu'il s'agira de coder une fonction `Ocaml`, un mot $\mu \in \Sigma^k$ sera représenté par une liste de caractères (`char list`).
Les paragraphes suivants étudient les algorithmes et propriétés de ces phases, chacun d'entre eux pouvant être abordé **de manière indépendante**.

I.1 Transformation de Burrows-Wheeler (BWT)

Soit $\mu \in \Sigma^k$ un mot. La transformation BWT réalise une permutation des symboles de μ de sorte que les symboles identiques sont regroupés dans de longues séquences. Cette transformation n'effectue pas de compression, mais prépare donc à une compression plus efficace.
Dans la suite, nous étudions le codage et le décodage d'un mot transformé par cette opération.

I.1.a Phase de codage

On rajoute à la fin de μ un marqueur de fin (par convention noté |, inférieur par $\leqslant$ à tous les autres symboles de Σ) [1]. Dans toute la suite, $\hat{\mu}$ désigne le mot auquel on a ajouté le symbole |.

1. Le choix de la barre verticale est curieux, il est situé **après** les lettres dans le code ASCII, il aurait été plus judicieux d'employer `!` ou `#`.

Question 1 Pour $\mu =$ `turlututu`, construire une matrice M dont les lignes sont les différentes permutations circulaires successives du mot $\hat{\mu}$. Les permutations seront ici envisagées par décalage à droite des caractères.

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

Question 2 Écrire une fonction récursive `circulaire : 'a list → 'a list` qui réalise une permutation à droite d'un mot μ donné en entrée.

La récursivité est dans une fonction auxiliaire qui découpe une liste en isolant le dernier terme

```
let circulaire l =
  let rec couper_dernier = function
    |[] -> failwith "Liste vide"
    |[x] -> [], x
    |t :: q -> let debut, fin = couper_dernier q in
               t :: debut, fin in
  match l with
  |[] -> []
  |_ -> let  debut, fin = couper_dernier l in
        fin :: debut
```

Question 3 Écrire une fonction `matrice_mot : 'a list → 'a list list` qui construit la matrice M à partir d'un mot passé en entrée. La valeur de retour est une liste de liste de symboles (une liste de mots). Cette fonction utilisera la fonction `circulaire`.

```
let matrice_mot mu =
  let n = List.length mu in
  let rec etape k mot =
    if k = n
    then []
    else mot :: (etape (k+1) (circulaire mot)) in
  etape 0 mu
```

Une permutation des lignes de M est alors effectuée, de sorte à classer les lignes par ordre lexicographique. On note $M' = P \cdot M$ la matrice obtenue, P étant la matrice de permutation.

Question 4 Donner les matrices P et M' dans le cas du mot $\hat{\mu}$, pour $\mu =$ `turlututu`.

|	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

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

Pour construire la matrice de permutation P, il faut trier la liste des mots définissant M. La méthode de tri choisie ici est le tri par insertion.

Question 5 Écrire une fonction récursive `tri : 'a list → 'a list` qui réalise le tri par insertion d'une liste d'éléments.

On admet que `OCaml` sait comparer deux listes selon l'ordre lexicographique.

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

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

Question 6 En déduire une fonction `matrice_mot_triée : 'a list → 'a list list` qui construit M' à partir de M.

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

Question 7 Pour $\mu \in \Sigma^k$, donner le nombre de comparaisons de symboles nécessaires au pire des cas, pour trier deux permutations circulaires du mot μ.

La complexité de la comparaison de deux listes de même taille est linéaire en fonction de la taille car on doit comparer terme-à-terme jusqu'à trouver deux termes distincts ou jusqu'à la fin si les deux mots sont égaux.

Question 8 En déduire la complexité dans le pire des cas pour le tri des k permutations circulaires d'un mot $\mu \in \Sigma^k$ (exprimée en nombre de comparaisons de symboles).

Le nombre de comparaisons de termes pour le tri d'une liste de k termes est en $\mathcal{O}(k^2)$ donc, comme chaque comparaison a une complexité linéaire en nombre de comparaisons d'éléments, la complexité totale est en $\mathcal{O}(k^3)$.

La transformation BWT consiste alors à coder le mot μ par la dernière colonne de la matrice M' obtenue à partir de $\hat{\mu}$. On note $BWT(\mu)$ ce codage.

Question 9 Écrire une fonction `codageBWT : char list → char list` qui encode un mot passé en entrée. On utilisera une fonction récursive pemettant de récupérer le dernier symbole de chacun des mots de M'. Donner le codage du mot $\mu = turlututu$.

```
let rec chapeau = function
  |[] -> ['!']
  |t :: q -> t :: (chapeau q)

let rec dernier = function
  |[] -> failwith "Liste vide"
  |[x] -> x
  |t :: q -> dernier q

let codageBWT mu =
  List.map dernier (matrice_mot_triee (chapeau mu))
```

On obtient `['u'; 'r'; 'u'; 'u'; '|'; 'u'; 't'; 't'; 't'; 'l'].`

I.1.b Phase de décodage

Pour décoder un mot codé par BWT, il est nécessaire de reconstruire itérativement M' à partir de la seule donnée du mot code $BWT(\mu)$. Par construction, $BWT(\mu)$ est la dernière colonne de M'. On pose ici comme exemple $BWT(\mu) =$ `edngvnea|`.

Question 10 Construire, à partir de la seule donnée de $BWT(\mu)$, la première colonne de M'. Justifier le principe de construction.

La première colonne de M', comme la dernière contient toutes les lettres de $\hat{\mu}$. De plus les premières lettres d'une liste triée sont triées. La première colonne est donc obtenue en triant la dernière (een fait, toutes les colonnes).
Dans l'exemple on obtient `['|'; 'a'; 'd'; 'e'; 'e'; 'g'; 'n'; 'n'; 'v'].`

La dernière et la première colonne de M' donnent alors tous les sous-mots de longueur 2 de μ.

Question 11 Proposer un algorithme permettant d'obtenir la deuxième colonne de M'. Donner cette deuxième colonne pour $BWT(\mu) =$ `edngvnea|`.

On crée la liste des facteurs de longueur 2 en prenant, dans l'odre, un terme de la dernière colonne puis un terme de la première colonne. Il suffit alors de trier pour obtenir la liste des 2 premiers termes de chaque ligne de M'.
Pour l'exemple, les facteurs d'ordre 2 sont
`['e|'; 'da'; 'nd'; 'ge'; 've'; 'ng'; 'en'; 'an'; '|v'].`
Les 2 premières colonnes de M' sont donc
`['|v'; 'an'; 'da'; 'e|'; 'en'; 'ge'; 'nd'; 'ng'; 've'].`

Question 12 On dispose à l'itération n des $(n-1)$ premières colonnes de M' et de sa dernière colonne. Proposer un principe algorithmique permettant de construire la n-ième colonne de M'.

De manière générale on crée la liste des facteurs de longueur n en prenant, dans l'ordre en ajoutant les terme de la dernière colonne devant les termes de la liste des facteurs de longueur $n-1$. Il suffit alors de trier pour obtenir la liste des n premiers termes de chaque ligne de M'.

Question 13 En déduire un algorithme itératif permettant de reconstruire M'.

On définit une fonction de collage d'une liste d'éléments et d'une liste de listes

```
let rec collage l1 l2 =
  match l1, l2 with
  |[], [] -> []
  |t1 :: q1, t2 :: q2 -> (t1 :: t2) :: (collage q1 q2)
  |_ -> failwith "Les deux listes n'ont pas la même longueur"
```

On itère n fois le collage de la dernière colonne puis un tri. La colonne de listes vides initiale est crée en transformant en `[]` *les éléments de la dernière colonne.*

```
let matricePrime mu =
  let n = List.length mu in
  let mPrime = ref (List.map (fun x -> []) mu) in
  for i = 0 to (n-1) do
    mPrime := tri (collage mu !mPrime) done;
  !mPrime
```

On obtient le mot initial en enlevant le premier élément de la première ligne.

```
let decodageBWT mu =
  List.tl (List.hd (matricePrime mu))
```

Question 14 Quel décodage obtient-on pour le mot $BWT(\mu)$ proposé ?

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

I.2 Codage par plages RLE

Le codage RLE (Run Length Coding), ou codage par plages, est une méthode de compression dont le principe est de remplacer dans une chaîne de symboles une sous-chaîne de symboles identiques par le couple constitué du nombre de symboles identiques et du symbole lui même.
Par exemple, la chaîne "`aaababb`" est compressée en `[(3,'a'); (1,'b'); (1,'a'); (2,'b')]`.

Question 15 Proposer un type Ocaml pour la compression RLE, qui permet de représenter le résultat comme indiqué précédemment.

```
type compRLE = (int * char) list
```

Question 16 Écrire une fonction `RLE` qui code un mot passé en entrée par codage RLE.

On commence par une fonction auxiliaire qui repère les termes égaux à la suite au début d'une liste

```
let rec egaux = function
  |[] -> failwith "Ceci ne devrait pas arriver"
  |[x] -> 1, x, []
  |x :: y :: q -> if x <> y
                  then 1, x, y :: q
                  else let k, a, reste = egaux (y :: q) in (k+1)
                      , a, reste
```

On extrait alors les termes

```
let rec codeRLE = function
  |[] -> []
  |l -> let k, x, reste  = egaux l in (k, x) :: (codeRLE reste)
```

Question 17 Écrire une fonction `decodeRLE` qui décode une liste `codeRLE` issue du codage RLE d'un mot.

```
let rec multi k x =
  assert (k >= 0);
  if k = 0
  then []
  else x :: (multi (k-1) x)

let rec decodeRLE = function
  |[] -> []
  | (k, x) :: reste -> (multi k x) @ (decodeRLE reste)
```

I.3 Codage de Huffman

Cette section est tirée du sujet CCINP MP, option info, de 2002 La suite de n caractères que nous allons traiter est notée $c_1 \dots c_n$. Un même caractère peut apparaître plusieurs fois. La suite est donc composée de p caractères distincts appartenant à l'ensemble $\{v_1, \dots, v_p\}$ avec $p \leqslant n$. Nous appellerons nombre d'occurrences d'un caractère v_i de la suite le nombre de fois que ce caractère ligure dans la suite. La fonction `occ` (v_i) renvoie le nombre d'occurrences de v_i dans la suite étudiée.

I.3.a Principe du codage

Définition 1

Une clé de codage binaire d'un ensemble de caractères est une application qui associe à chaque caractère un code composé d'une suite non vide de valeurs binaires 0 ou 1 (un code binaire).

Exemples : $1\{a \mapsto 00, b \mapsto 010, c \mapsto 011, d \mapsto 1\}$, $\{a \mapsto 00, b \mapsto 01, c \mapsto 10, d \mapsto 11\}$ et $\{a \mapsto 0, b \mapsto 01, c \mapsto 1, d \mapsto 10\}$ sont des clés de codage possibles pour l'ensemble $\{a, b, c, d\}$. La seconde clé est uniforme car elle utilise le même nombre de bits pour chaque caractère.
Une clé de codage permet de coder une suite de caractères par une suite de bits en remplaçant chaque caractère de la suite par son code.
Par contre, si nous considérons la suite `dadbcd`, son code est 10010100111 selon la première clé et 110011011011 selon la clé uniforme. Remarquons que le codage uniforme utilise 12 bits alors que le premier codage n'utilise que 11 bits.
Un code binaire est représenté par une liste de booléens. Une clé de codage est représentée par le type `cle`

```
type suite = char list
type code = boolean list
type cle = (char * code) list
```

Nous supposons pour l'instant que nous disposons déjà d'une clé de codage nous permettant de coder la suite de caractères.

Question 18 Écrire une fonction `coder : suite -> cle -> code` telle que l'appel `coder s c` renvoie le code de la suite de caractères `s` selon la clé de codage `c`.

On commence par la fonction qui associe une valeur à une clé

```
let rec valeur cle dict =
  match dict with
  |[] -> failwith "La clé n'existe pas"
  |(x,l) :: reste -> if x=cle
                     then l
                     else valeur cle reste
```

Il suffit alors d'associer les codes

```
let coder s c =
  let rec coder_aux s c =
    match s with
    |[] -> []
    |t :: q  -> (valeur t c) @ (coder_aux q c)
  in coder_aux s c
```

Question 19 Calculer une estimation de la complexité dans le pire cas de la fonction `coder` en fonction de la taille des listes `s` et `c`. Cette estimation ne prendra en compte que le nombre d'appels récursifs effectués.

La complexité de la fonction `valeur` *est au pire la longueur de liste des couples. Dans* `coder`*, on fait appel à la fonction* `valeur` *pour chaque élément puis on concatène la liste du code. La complexité totale est donc un* $\mathcal{O}\big(|s| \cdot |c| \cdot p\big)$ *où* p *est la longueur maximum des codes de* `c`*.*

La définition précédente est trop générale pour permettre une opération de décodage simple. Nous étudierons donc des codages possédant des propriétés supplémentaires.
Nous noterons $\mathcal{I}$ l'image de $\{v_1, \dots, v_p\}$ selon la clé de codage. $\mathcal{I}$ contient donc l'ensemble des valeurs de code possibles.

Définition 2

Une clé de codage est séparable si et seulement si :

$$\forall b \in \mathcal{I}, b = b_1 \dots b_r, \forall i \in [1, r-1], b_1 \dots b_i \notin \mathcal{I}$$

Cette propriété indique que le code d'un caractère n'est jamais un préfixe du code d'un autre caractère. Elle est nécessaire pour permettre le décodage.
Les deux premiers exemples sont des clés séparables. Le troisième exemple n'est pas séparable.

Définition 3

Une clé de codage est optimale si et seulement si :

$$\forall b \in \mathcal{I}, b = b_1 \dots b_r, \forall i \in [1, r], \exists b_1 \dots b_{i-1} b'_i b'_{i+1} \dots b'_s \in \mathcal{I}, b'_i = \neg b_i$$

Cette propriété indique que le i-ème bit permet de distinguer $b_1 \dots b_r$ de $b_1 \dots b'_s$.
Les deux premiers exemples sont optimaux. Le troisième exemple n'est pas optimal.
Cette propriété permet de minimiser la taille des codes indépendamment de la suite considérée.
La propriété suivante prend celle-cí en compte.
Nous noterons $\mathcal{C}(v)$ le nombre de bits utilisé pour coder le caractère v.
Le coût du codage de la suite est alors : $\mathcal{C}\left(c_1 \dots c_n\right) = \sum_{i=1}^{p} \texttt{occ}\left(v_i\right) \cdot \mathcal{C}\left(v_i\right)$

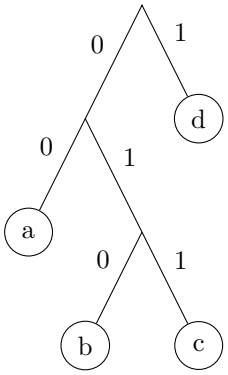
Définition 4

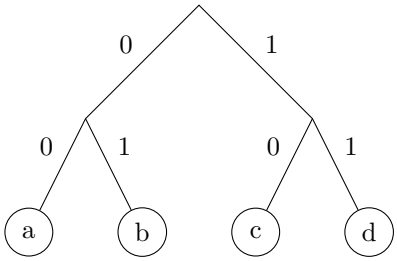
Un codage est minimal pour une suite de caractères donnée s'il permet de coder tous les caractères en utilisant le plus petit nombre de bits possible lors du codage de la suite, c'est-à-dire s'il permet de minimiser $\mathcal{C}\left(c_1 \dots c_n\right)$.

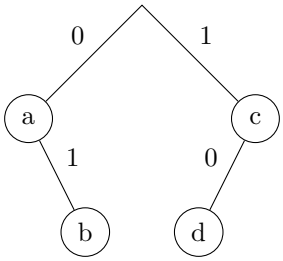
L'algorithme étudié dans cette partie construit une clé de codage optimale séparable assurant un codage minimal pour une suite de caractères donnée. Nous appellerons cette clé *minimale*.

I.3.b Arbre de Huffman

La clé de codage d'un ensemble de caractères peut être représentée par un arbre binaire dont certains nœuds sont étiquetés par un caractère et toutes les arêtes sont étiquetées par la valeur d'un bit de code.
Nous considérerons uniquement des arbres dont toutes les feuilles contiennent un caractère.
Les arbres suivants correspondent aux trois exemples précédents :







Définition 5

Un arbre binaire est complet si et seulement si chaque nœud interne possède deux fils.

Question 20 Montrer que la clé de codage associée à an arbre complet est optimale.

Pour $b_1 \dots b_r \in \mathcal{I}$, il existe dans l'arbre associé à la clé de codage un chemin étiquetée par $b_1 \dots b_r$ partant de la racine de l'arbre et aboutissant à une feuille f. Pour $1 \leqslant i \leqslant r$le nœud situé à la hauteur $i-1$ situé sur ce chemin admet un fils ; c'est un nœud interne n_i.
Si l'arbre est complet alors n_i admet deux fils dont l'un aboutit à f et l'autre doit aboutir à au moins une autre feuille f'.
Comme l'étiquette du chemin conduisant au nœud n est $b_1 \dots b_{i-1}$, alors celle du chemin aboutissant à f' est $b_1 \dots b_{i-1} b'_i \dots b'_s \in \mathcal{I}$ avec $b'_i = \neg b_i$.
La clé de codage associée à un arbre complet est bien optimale.

Question 21 Montrer que la clé de codage associée à un arbre complet est séparable si et seulement si seules les feuilles de cet arbre contiennent des caractères.

La condition d'arbre complet semble inutile.
Si un nœud d'un arbre associé à une clé de codage contient un caractère x alors en suivant un chemin depuis ce nœud vers une feuille contenant un code y, on prolonger le code de x en un code de y, celui de x est donc un préfixe de celui de y.
Inversement, s'il existe un code pour x préfixe du code pour y, le chemin associé à y doit passer par la lettre x dans un nœud.
On a prouver les contraposées des 2 implications de l'énoncé.

Les arbres utilisés pour construire une clé de codage optimale séparable seront donc tels que :

- les nœuds ont deux fils et ne contiennent pas de caractère ;
- les feuilles n'ont pas de fils et contiennent un caractère.

Nous appellerons ces arbres des arbres de codage. Nous supposerons par la suite qu'un même caractère ne figure pas dans plusieurs feuilles distinctes.

Définition 6

Soit une suite $c_1 \dots c_n$ de caractères appartenant à l'ensemble $\{v_1, \dots, v_n\}$ avec les nombres d'occurrences $\texttt{occ}(v_j)$, à chaque arbre de codage de cet ensemble est associé un arbre appelé arbre de Huffinan obtenu en ajoutant à chaque feuille de l'arbre de codage contenant le caractère v_j l'occurrence $\texttt{occ}(v_j)$ du caractère v_j. Chaque feuille de l'arbre de Huffman contient donc un caractère v_j et le nombre d'occurrence $\texttt{occ}(v_j)$ de v_j dans $c_1 \dots c_n$.

Un arbre de Huffman est représenté par le type

```
type arbre =   Vide
             | Feuille of char * int
             | Noeud of arbre * arbre
```

Une feuille contient un caractère ainsi que le nombre d'occurrences de ce caractère. Le cas `Vide` permet de construire des arbres non complets. Il ne sera utilisé que lors de la reconstruction de l'arbre dans la phase de décodage.
Les deux premiers exemples précedents seront codés pour la suite `daacadbed` par :

```
Noeud (Noeud (Feuille ('a', 3),
              Noeud (Feuille ('b', 1), Feuille ('c', 2))),
       Feuille('d', 3))

Noeud (Noeud (Feuille ('a', 3), Feuille('b', 1)),
       Noeud (Feuille ('c', 2), Feuille('d', 3)))
```

Le dermier exemple ne peur pas être codé car les nœuds ne peuvent pas contenir de caractères (clé de codage séparable).
Une liste d'arbres est représentée par `type liste = arbre list`.
Les feuilles des arbres sont étiquetées par un caractère et par un entier qui indique le nombre d'occurrences du caractère dans la suite étudiée. Le nombre global des occurrences des différents caractères qui figurent dans un arbre correspond alors à la somme des nombres d'occurrences de toutes les feuilles de cet arbre. La fonction `occ` est ainsi étendue à l'arbre a.

Question 22 Écrire une fonction `nombre : arbre -> int` telle que l'appel `nombre a` renvoie le nombre d'occurrences `occ`(a) des caractères figurant dans l'arbre `a`.

```
let rec nombre = function
  |Vide -> 0
  |Feuille(_,n) -> n
  |Noeud (g, d) -> (nombre g) + (nombre d)
```

Définition 7

Un arbre de Huffman `a` pour une chaîne c est minimal si le codage associé est minimal pour la chaîne c parmi codage associés aux arbre de Huffman.

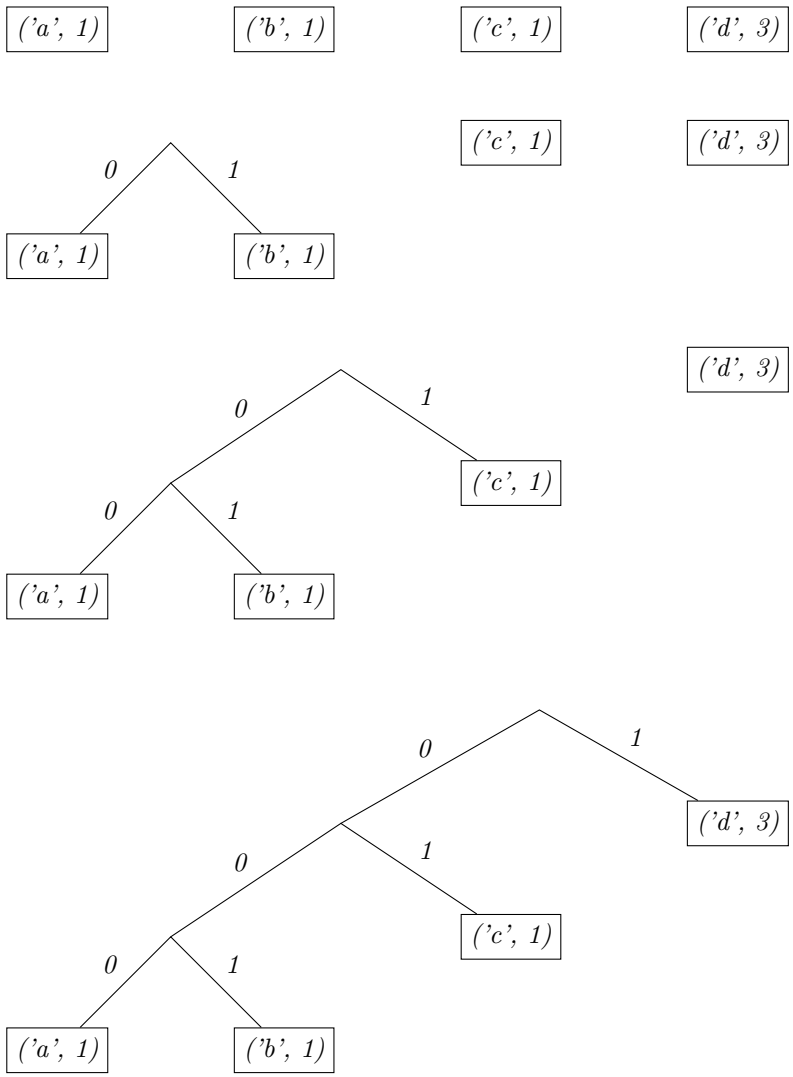
I.3.c Codage de Huffman

L'algorithme de Huffman construit un arbre de Huffman. Pour cela, il manipule un ensemble de sous-arbres de Huffman partiels.
Cet ensemble contient initialement l'ensemble des feuilles étiquetées par les caractères de $\{v_1, \ldots, v_p\}$.
Chaque étape de l'algorithme remplace les deux sous-arbres g et d contenant les plus petits nombres d'occurrences par un arbre de Huffman dont les fils sont g et d. L'algorithme s'arrête lorsque l'ensemble ne contient plus qu'un seul arbre.

Question 23 Appliquer l'algorithme précédent pour construire l'arbre de Huffman correspondant à l'exemple `dadbcd` en détaillant chaque ensemble intermédiaire.

Il y a plusieurs arbres de Huffman possibles car la première construction n'est pas déterminée.

$$\mathcal{C}(c_1 \ldots c_n) = \sum_{i=1}^{p} \texttt{occ}(v_i) \cdot \mathcal{C}(v_i)$$



Considérons H un arbre de Huffman de $\{v_1, \cdots, v_p\}$. L'arbre H' est obtenu en permutant les feuilles v_i et v_j dans H.

Question 24 Calculer la différence, Δ, entre le coût de H et le coût de H' en fonction des occurrences et des profondeurs de v_i et v_j.

Le coût d'un caractère pour un code associé à un arbre de Huffman est la profondeur de la feuille contenant ce caractère.
La contribution dans le coût de H de v_i et de v_j est $\mathcal{C}(v_i) \cdot \texttt{occ}(v_i) + \mathcal{C}(v_j) \cdot \texttt{occ}(v_j)$.
Dans H', les profondeurs donc le coût des feuilles sont permutés donc la contribution de v_i et de v_j est $\mathcal{C}(v_j) \cdot \texttt{occ}(v_i) + \mathcal{C}(v_i) \cdot \texttt{occ}(v_j)$
Ainsi $\Delta = \mathcal{C}(H) - \mathcal{C}(H') = \big(\mathcal{C}(v_i) - \mathcal{C}(v_j)\big) \cdot \big(\texttt{occ}(v_i) - \texttt{occ}(v_j)\big)$.

Question 25 Montrer que dans un arbre de Huffman minimal, les feuilles de profondeur maximale contiennent des caractères d'occurrence minimale.

La question est imprécise : on va prouver que les feuilles de profondeur maximale ne peuvent contenir un caractère dont l'occurence est strictement supérieure à une feuille de profondeur plus petite.
Par l'absurde, si ce n'était pas le cas, on aurait un caractère v à la profondeur maximale h et un caractère v' à la profondeur maximale h' avec $h' < h$ et $\texttt{occ}(v) > \texttt{occ}(v')$.

En échangeant ces deux caractères le coût varierait de $\Delta = (h - h') \cdot \big(\texttt{occ}(v) - \texttt{occ}(v')\big) < 0$ ce qui contredit la minimalité du coût de H : la proprité est vérifiée.

Considérons H un arbre de Huffman minimal de $\{v_1, \ldots, v_p\}$. v_i et v_j sont deux caractères d'occurrence minimale possédant un même nœud père n dans H. L'arbre H' est obtenu en enlevant les feuilles v_i et v_j dans H et en considérant que n est une feuille d'occurrence $\texttt{occ}(n) = \texttt{occ}(v_i) + \texttt{occ}(v_j)$.

Question 26 Montrer que H' est un arbre de Huffman minimal
Indication : supposer que H est minimal et que H' n'est pas minimal.

Supposons H minimal.
On note $V = \{v_1, \ldots, v_p\}$ et $V' = \big(V \setminus \{v_i, v_j\}\big) \cup \{n\}$.
En remplaçant v_i et v_j par n, comme la hauteur diminue de 1, la variation de coût est $\mathcal{C}(H) - \mathcal{C}(H') = \texttt{occ}(v_i) + \texttt{occ}(v_j)$
Si H' n'était pas minimal pour V', il existerait un arbre de Huffman H'_1 pour V' tel que $\mathcal{C}(H'_1) < \mathcal{C}(H')$.
On pourrait alors construire un arbre de Huffman pour V, H_1, obtenu à partir de H'_1 en remplaçant la feuille n par l'arbre binaire de hauteur 1 dont les feuilles sont v_i et v_j.
On aurait alors $\mathcal{C}(H_1) = \mathcal{C}(H'_1) + \texttt{occ}(v_i) + \texttt{occ}(v_j) < \mathcal{C}(H') + \texttt{occ}(v_i) + \texttt{occ}(v_j) = \mathcal{C}(H)$, ce qui contredit la minimalité de H. H' doit être minimal.

Question 27 Montrer que l'arbre construit par l'algorithme de Huffman est minimal.

On va montrer par récurrence la propriété

$\mathcal{P}(p)$ *: un arbre construit par l'algorithme de Huffman pour p caractères est minimal.*

$\mathcal{P}(1)$ est vraie car il n'y a qu'un seul arbre de Huffman possible pour un seul caractère (une feuille) : l'algorithme de Huffman donne donc un arbre minimal.
On suppose que $\mathcal{P}(p-1)$ est vraie.
On considère un ensemble V de p caractères, un arbre H construit par l'algorithme de Huffman pour V et un arbre de Huffman minimal pour V, H'.
On note v et v' les deux caractères choisis pour la première fusion dans l'algorithme de Huffman ; ce sont des caractères d'occurrence minimale. Ils ont le même père, n.

- *H' est un arbre complet donc il admet au moins deux feuilles à la profondeur maximale qui ont un même père..*
- *D'après l'exercice 25 H' contient, à la profondeur maximale, les caractères d'occurence minimale.*
- *En échangeant, si besoin, des caractères de même occurrence dans H', ce qui ne change pas le coût, on peut supposer que v et v' sont à la profondeur maximale dans H'.*
- *En échangeant, si besoin, des caractères de même profondeur dans H', ce qui ne change pas le coût, on peut supposer que v et v' ont un même père, n', dans H'.*
- *L'exercice 26 montre que si on remplace l'arbre n' par une feuille de caractère $\hat{v}$ n'appartenant pas à V et d'occurence $\texttt{occ}(v) + \texttt{occ}(v)$ on obtient un arbre de Huffman H'_1 minimal pour $V' = \big(V \setminus \{v, v\}\big) \cup \{\hat{v}\}$.*
- *En remplaçant le premier arbre construit lors de l'algorithme de Huffman par une feuille de caractère $\hat{v}$ et d'occurence $\texttt{occ}(v) + \texttt{occ}(v)$, l'algorithme de Huffman va se poursuivre sans changement et aboutir à l'arbre H_1 obtenu en remplaçant n par la feuille $\hat{v}$.*
- *D'après l'hypothèse de récurrence $\mathcal{C}(H_1) = \mathcal{C}(H'_1)$ car H'_1 est minimal et H_1 est construit par l'algorithme de Huffman sur un même ensemble de $p-1$ caractères.*
- *On a $\mathcal{C}(H) = \mathcal{C}(H_1) + \texttt{occ}(v) + \texttt{occ}(v)$ et $\mathcal{C}(H') = \mathcal{C}(H'_1) + \texttt{occ}(v) + \texttt{occ}(v)$ d'après la démonstration de l'exercice 26 donc $\mathcal{C}(H) = \mathcal{C}(H')$ d'où H est minimal.*

Ainsi $\mathcal{P}(p)$ est vraie : la récurrence prouve le résultat.

II CCINP MPI 2023, premier exercice : palindromes

En langue française, "ressasser" est le mot palindrome le plus long, tandis qu'il semble que "saippuakauppias" soit le plus long mot palindrome au monde, désignant un marchand de savon en Finlande. L'objet de cette partie est de compter le nombre de palindromes présents dans un mot donné.
Soit Σ un alphabet fini contenant au moins deux lettres. on note $u = u_0 \dots u_{n-1}$ un *mot* sur Σ, composé de n lettres $u_i \in \Sigma, i \in [\![0; n-1]\!]$. La longueur de u est notée $|u|$. Pour $0 \leqslant i < j \leqslant n$, on note $u[i, j]$ le mot $u_i \cdots u_{j-1}$. L'ensemble des mots construit sur Σ et contenant le mot vide ε est noté Σ^*.

Définition 8 : Miroir, palindrome

Soit $u \in \Sigma^*$.
Le *miroir* de $u = u_0 \dots u_{n-1}$, noté $\overline{u}$, est le mot $\overline{u} = u_{n-1} \dots u_0$. Par convention $\overline{\varepsilon} = \varepsilon$.
$u \in \Sigma^*$ est un *palindrome* si et seulement si $u = \overline{u}$ et $u \neq \varepsilon$.
On dira qu'un palindrome u est *pair* (respectivement *impair*) lorsque sa longueur $|u|$ est paire (resp. impaire).

On recherche donc dans cette partie le nombre de palindromes facteurs d'un mot $u \in \Sigma^*$ (comptés avec les multiplicités éventuelles), soit le cardinal de l'ensemble $\left\{(i, j), 0 \leqslant i < j \leqslant |u|, u[i, j] = \overline{u[i, j]}\right\}$.
Dit autrement, on recherche le nombre de palindromes contenus dans u.

Question 28 Si $\Sigma = \{a, b\}$, donner le nombre de palindromes contenus dans le mot $u =$ `babb`.

Les palindromes de `babb` *sont* `a`, `b` *(trois fois),* `bb` *et* `bab`*, soit 6 palindromes.*

Algorithme 1 : Décompte naïf du nombre de palindromes contenu dans un mot donné

```
fonction nombrePalindromes(u : mot) =
    nb ← 0
    n ← |u|
    pour i = 0 à n − 1 faire
        pour j = 0 à n − 1 faire
            estPalindrome ← Vrai
            pour k = i à j − 1 faire
                si u[i] ≠ u[j − k − 1] alors estPalindrome ← Faux
    si estPalindrome alors nb ← 1+nb
    retourner nb
```

En parcourant les lettres d'un mot u donné, on peut proposer l'algorithme1 en pseudo-code permettant de compter naïvement les palindromes de u.

Question 29 Évaluer la complexité au pire des cas de l'algorithme 1 en fonction de $|u|$.

Le code comporte beaucoup d'erreurs : tel qu'il est écrit, il va comptabiliser beaucoup de mots vides comme palindromes et il ne teste pas les facteurs comportant la dernière lettre, en plus d'une erreur d'indice pour la boucle de k.
Heureusement que la preuve n'est pas demandée.

Algorithme 2 : Décompte naïf corrigé

```
fonction nombrePalindromes(u : mot) =
    nb ← 0
    n ← |u|
    pour i = 0 à n − 1 faire
        pour j = i + 1 à n faire
            estPalindrome ← Vrai
            pour k = i à j − 1 faire
                si u[k] ≠ u[i + j − k − 1] alors estPalindrome ← Faux
    si estPalindrome alors nb ← 1+nb
    retourner nb
```

On remarque que la boucle pour k teste 2 fois les égalité de lettres (sauf celle du milieu).
Il y a 3 boucles imbriquées, chacune de longueur n au plus : la complexité est en $\mathcal{O}(n^3)$.

On souhaite bien sûr améliorer cette première idée. Pour ce faire, on utilise tout d'abord le paradigme de la programmation dynamique.
Pour $u \in \Sigma^*$, on définit un tableau de booléens P de taille $(|u|+1) \times (|u|+1)$, $P[i][j]$ étant vrai si $u[i,j]$ est un palindrome. on a donc pour tout $i \in [\![0, |u|-1]\!]$, $P[i][i+1] =$ `Vrai`.

Question 30 Soit $u[i,j]$ un mot.
À quelles conditions sur u_i, u_{j-1} et $u[i+1, j-1]$ le mot $u[i,j]$ est-il un palindrome ?

$u[i,j]$ est un palindrome ssi $u_i = u_{j-1}$ et $u[i+1, j-1]$ est un palindrome.

Question 31 En déduire une relation de récurrence vérifiée par les coefficients de P.

La question précédente se traduit par $P[i][j] = P[i+1][j-1] \wedge (u_i = u_{j-1})$.

Question 32 Écrire un algorithme de programmation dynamique en pseudo-code résolvant le problème. Évaluer sa complexité.

Les mots de deux lettres ne peuvent pas être définis depuis les mots de 0 lettre : on doit définir aussi $P[i][i+2]$ par $u_i = u_{i+1}$.
Ensuite on calcule les termes $P[i][i+d]$ en fonction des $P[i+1][i+d-1]$
pour d variant de 3 à $|u|$ et i variant de 0 à $n-d$.
Après avoir rempli la matrice, on compte les couples (i,j) pour lesquels $P[i][j]$ vaut `Vrai`.

Algorithme 3 : Décompte du nombre de palindromes par programmation dynamique

```
fonction nombrePalindromesProgDyn(u : mot) =
    n ← |u|
    P ← matrice (n + 1) × (n + 1) de termes Faux
    pour i = 0 à n − 1 faire P[i][i + 1] ← Vrai
    pour i = 0 à n − 2 faire P[i][i + 2] ← u[i] = u[i + 2]
    pour d = 3 à n faire
        pour i = 0 à n − d faire
            P[i][i + d] ← P[i + 1][i + d − 1] ∧ u[i] = u[i + d − 1]
    nb ← 0
    pour i = 0 à n − 1 faire
        pour j = i + 1 à n faire
            si P[i][j] alors 1 ← nb + 1
    retourner nb
```

La complexité est en $\mathcal{O}(n^2)$.

On insère maintenant entre chaque paire de lettres de u, ainsi qu'au début et à la fin du mot, un symbole spécial noté #. On appelle ce nouveau mot $u^{\#}$.
Ainsi le mot $u = \texttt{babb}$ se transforme en $u^{\#} = \#\texttt{a}\#\texttt{b}\#\texttt{b}\#\texttt{a}\#$.

Question 33 Montrer que les palindromes de $u^{\#}$, s'ils existent, sont tous impairs.

Si un facteur de longueur paire, $u[i, i+2p]$, est un palindrome alors $u[i+k] = u[i+2d-1-k]$. En particulier, pour $k = d-1$, $u[i+d-1] = u[i+d]$, ce qui est impossible car, parmi deux lettres consécutives, il y a # et une lettre de Σ donc distincte de #.
Il n'y a donc pas de palindrome pair dans $u^{\#}$.

Question 34 Soit v un palindrome de longueur $2k+1$ de u, $k \in \mathbb{N}$ On construit le mot $u^{\#}$. Donner les deux palindromes correspondant à v dans $u^{\#}$.

Si $\texttt{v} = \texttt{x}_1\texttt{x}_2\cdots\texttt{x}_{p-1}\texttt{x}_p\texttt{x}_{p-1}\cdots\texttt{x}_2\texttt{x}_1$ est un palindrome impair alors
$\texttt{v}^{\#} = \#\texttt{x}_1\#\texttt{x}_2\#\cdots\#\texttt{x}_{p-1}\#\texttt{x}_p\#\texttt{x}_{p-1}\#\cdots\#\texttt{x}_2\#\texttt{x}_1\#$ *est un palindrome de $u^{\#}$*
ainsi que $\texttt{x}_1\#\texttt{x}_2\#\cdots\#\texttt{x}_{p-1}\#\texttt{x}_p\#\texttt{x}_{p-1}\#\cdots\#\texttt{x}_2\#\texttt{x}_1$.

Question 35 Même question si v est un palindrome de longueur $2k$ de u.

La réponse est la même, si v est un palindrome de u alors $v^{\#}$ et $v^{\#}[1, m]$ sont des palindromes de $u^{\#}$ où $m = 2|v|+1$ est la longueur de $v^{\#}$.

Question 36 En déduire une stratégie de recherche de tous les palindromes de u.

Pour trouver les palindromes de u, on peut construire $u^{\#}$ et en chercher les palindromes. On considère alors les palindromes de $u^{\#}$ ne commençant pas par # (un sur 2) et on en retire les lettres #.
On notera que les palindromes # de $u^{\#}$ ne sont pas associés à des palindromes de u, on les a évités par le choix fait.
Si le but est simplement de calculer le nombre de palindromes de u, note $n^{\#}$ le nombre de palindromes de $u^{\#}$, le nombre de palindromes de u est alors $\dfrac{n^{\#} - |u| - 1}{2}$ car on doit retirer les palindromes #.

On voit que les palindromes impairs sont importants.
On va donc construire un algorithme de recherche de ce type de palindrome.

Définition 9 : Rayon d'un palindrome

Soient $u \in \Sigma^*$ et $i \in [\![0, |u|-1]\!]$. On dit qu'il existe un *palindrome centré en i de rayon* $\rho > 0$ si :

- $i - \rho \geqslant 0$,
- $i + \rho + 1 \leqslant |u|$,
- $u[i-\rho, i+\rho+1]$ est un palindrome.

Le *rayon maximal* $\hat{\rho}_i$ d'un palindrome centré en i est le plus grand rayon d'un palindrome centré en i.

Par exemple, si $u = \texttt{abbababab}$ et $i = 4$, alors $\texttt{b}$ est un palindrome centré en 4 de rayon 0, $\texttt{aba}$ est un palindrome centré en 4 de rayon 1, $\texttt{babab}$ est un palindrome centré en 4 de rayon 2 et c'est le plus grand, donc $\hat{\rho}_4 = 2$.

Question 37 En remarquant qu'un palindrome impair est centré sur une lettre (ou un symbole spécial dans le cas de $u^{\#}$), proposer un algorithme en pseudo-code permettant de compter le nombre de palindromes impairs d'un mot u. Évaluer sa complexité.

La définition pose problème car elle semble exclure le rayon nul alors que certains indices peuvent n'avoir que u_i comme palindrome centré en i. On note que, si $\hat{\rho}_i$ est le rayon maximal, alors il existe un palindrome centré en i de rayon ρ pour tout $\rho \leqslant \hat{\rho}_i$ (et $\rho \geqslant 0$).
On utilise le fait que u admet $\hat{\rho}_i + 1$ palindromes centrés en i.

Algorithme 4 : Décompte du nombre de palindromes par calcul du rayon

```
fonction rayon(u : mot, i : indice) =
    n ← |u|
    r ← 0
    tant que (i − r − 1 ⩾ 0) ∧ (i + r + 1 < n) ∧ (u[i − r − 1] = u[i + r + 1]) faire
        r ← r + 1
    retourner r
fonction nombrePalindromesRayon(u : mot) =
    n ← |u|
    nb ← 0
    pour i = 0 à n − 1 faire nb ← nb + rayon(i) + 1
    retourner nb
```

Question 38 Soient $u \in \Sigma^*, i \in [\![0, |u| - 1]\!]$ et $j \in [\![1, \hat{\rho}_i]\!]$. En exploitant les différentes symétries, montrer qu'il existe un palindrome centré en $i + j$ de rayon $\min(\hat{\rho}_i - j, \hat{\rho}_{i-j})$.
En déduire $\hat{\rho}_{i+j} \geqslant \min\big(\hat{\rho}_i - j, \hat{\rho}_{i-j}\big)$. Préciser à quelle condition il y a égalité.

- *Si on a $0 \leqslant k \leqslant \hat{\rho}_i - j$ alors $1 \leqslant j + k \leqslant \hat{\rho}_i$.*
  Par définition du rayon, on a $u[i + j + k] = u[i - j - k]$: (1).
  Pour $k = 1 + \hat{\rho}_i - j$ alors $k + j$ est la première valeur après $\hat{\rho}_i$ donc, d'après la définition du rayon, $i + j + k \geqslant n$, $i - j - k < 0$ ou $u[i + j + k] \neq u[i - j - k]$: (1′).
- *De même, si $k \leqslant \hat{\rho}_{i-j}$, $u[i - j - k] = u[i - j + k]$: (2).*
  Et, pour $k = \hat{\rho}_{i-j} + 1$, $i - j - k < 0$, $i - j + k \geqslant n$ ou $u[i - j - k] \neq u[i - j + k]$: (2′).
- *On a $j - k \leqslant j \leqslant \hat{\rho}_i$ et, pour $k \leqslant 1 + \hat{\rho}_i - j$, $k - j \leqslant 1 + \hat{\rho}_i - 2j \leqslant \hat{\rho}_i - 1$ pour $j \geqslant 1$; ainsi $|k - j| \leqslant \hat{\rho}_i$ d'où $u[i - j + k] = u[i + j - k]$: (3).*
- *Pour tout $k \leqslant \min\big(\hat{\rho}_i - j, \hat{\rho}_{i-j}\big)$, on obtient, en combinant (1), (2) et (3), $u[i + j + k] = u[i - j - k] = u[i - j + k] = u[i + j - k]$ d'où $\hat{\rho}_{i+j} \geqslant \min\big(\hat{\rho}_i - j, \hat{\rho}_{i-j}\big)$.*
- *Si on a $\hat{\rho}_i - j < \hat{\rho}_{i-j}$ alors, pour $k = \hat{\rho}_i - j + 1$, on a encore $k \leqslant \hat{\rho}_{i-j}$ et la combinaison de (1′), (2) et (3) donne $u[i + j + k] \neq u[i - j - k] = u[i - j + k] = u[i + j - k]$ d'où $\hat{\rho}_{i+j} < k$, c'est-à-dire $\hat{\rho}_{i+j} \leqslant k - 1 = \hat{\rho}_i - j = \min\big(\hat{\rho}_i - j, \hat{\rho}_{i-j}\big)$. On a donc égalité.*
- *Si on a $\hat{\rho}_i - j > \hat{\rho}_{i-j}$ alors, pour $k = \hat{\rho}_{i-j} + 1$, on a encore $k \leqslant \hat{\rho}_i - j$ et la combinaison de (1), (2) et (3′) donne $u[i + j + k] = u[i - j - k] = u[i - j + k] \neq u[i + j - k]$ d'où $\hat{\rho}_{i+j} < k$, c'est-à-dire $\hat{\rho}_{i+j} \leqslant k - 1 = \hat{\rho}_{i-j} = \min\big(\hat{\rho}_i - j, \hat{\rho}_{i-j}\big)$. On a donc égalité.*
- *Il y a donc égalité si $\hat{\rho}_i - j \neq \hat{\rho}_{i-j}$.*
  La réciproque n'est pas vraie, il peut y avoir l'égalité $\hat{\rho}_{i+j} = \min\big(\hat{\rho}_i - j, \hat{\rho}_{i-j}\big)$ aussi si on a $\hat{\rho}_i - j > \hat{\rho}_{i-j}$. Par exemple, pour $\mathtt{u} = \mathtt{abac}$, on a $\hat{\rho}_0 = 0$, $\hat{\rho}_1 = 1$, $\hat{\rho}_2 = 0$ et $\hat{\rho}_3 = 0$ donc, pour $i = 1$ et $j = 1$, $\hat{\rho}_i - j = 0 = \hat{\rho}_{i-j}$ et $\hat{\rho}_{i+j} = 0 = \min\big(\hat{\rho}_i - j, \hat{\rho}_{i-j}\big)$.

En utilisant cette remarque, on développe un algorithme, dit de Manacher, qui construit un tableau d'entiers T permettant de compter le nombre de palindromes d'un mot u. Plus précisément, pour chaque position $i \in [\![1, |u| - 1]\!]$, $T[i]$ indique le rayon maximal $\hat{\rho}_i$, donc tel que la sous-chaîne de $i - \hat{\rho}_i$ à $i + \hat{\rho}_i + 1$ [2] est un palindrome. L'algorithme 5 consiste à incrémenter $T[i]$ jusqu'à trouver le plus grand palindrome $u[i - T[i], i + T[i] + 1]$ centré en i.

2. Le sujet comportait une erreur (de plus) :il écrivait de $i - \hat{\rho}_i$ à $i - \hat{\rho}_i + 1$.

Algorithme 5 : Algorithme de Manacher

```
1  fonction manacher(u : mot) =
2      n ← |u|
3      pour i = 0 à n − 1 faire T[i] ← 0
4      k ← 0
5      pour i = 0 à n − 1 faire
6          j ← i − k
7          si T[k] ⩾ j alors T[i] ← min(T[k − j], T[k] − j)
8          r ← T[i] + 1                                  /* ajouté pour la lisibilité */
9          tant que (i − r ⩾ 0) ∧ (i + r < n) ∧ (u[i − r] = u[i + r]) faire
10             T[i] ← r
11             r ← r + 1
12             k ← i                                     /* La même instruction est répétée */
13     retourner T
```

Question 39 Comment trouver à partir de cet algorithme le nombre de palindromes de u ?

C'est la même chose qu'à la question 37 : il suffit de remplacer `rayon`(i) *par* $T[i]$ *dans la fonction* `nombrePalindromesRayon`*.*

Question 40 Quelle est la complexité de cet algorithme ? Justifier.

La question est difficile, surtout quand les explications du sujet n'expliquent pas du tout l'algorithme. Il semble que la complexité soit quadratique mais ...
Si on écrit la ligne 7 en ***si*** $T[k] \geqslant j$ ***alors*** $T[k+j] \leftarrow \min(T[k-j], T[k]-j)$*, on obtient la similitude avec l'algorithme 4 avec un point de départ optimisé par la question 38.*
On prouve facilement, avec la ligne 10 que k est majoré par i et ne peut que croître.
De plus k croît strictement ssi $T[i]$ est strictement supérieur à sa valeur initiale.
On remarque que $T[i]$ est augmentée de 1 à chaque passage dans la boucle **tant que***.*
La grosse astuce est alors de considérer la valeur de $T[k]+k$.
On note τ_i la valeur de $T[k]+k$ après le passage dans la boucle **tant que** *et p_i le nombre de passages dans la boucle* **tant que** *lors du traitement de i.*

- *Si $T[i]$ garde sa valeur initiale, c'est-à-dire si la boucle* **tant que** *n'est pas effectuée, $p_i = 0$, alors k est inchangé donc $\tau_i = \tau_{i-1} = \tau_{i-1} + p_i$*
- *Si $T[i]$ est modifiée alors k prend la valeur i.*
  - *Si on avait $T[k] < j = i - k$, alors $T[i]$ commence à 0 donc finit par p_i d'où $\tau_i = T[i] + i = p_i + j + k > p_i + T[k] + k = p_i + \tau_{i-1}$.*
  - *Si on avait $T[k] \geqslant j = i - k$, alors $T[i]$ commence à une valeur supérieure ou égale à $T[k] - j$ et est augmenté de p_i d'où $\tau_i = T[i] + i \geqslant T[k] - j + p_i + i = T[k] + k + p_i = p_i + \tau - i - 1$.*

On vérifie que $\tau_0 = 0$ et $p_0 = 0$ donc la relation $\tau_i \geqslant p_i + \tau_{i-1}$ donne $\tau_{n-1} \geqslant \sum_{i=0}^{n-1} p_i$.
Or $\tau_{n-1} = T[n-1] + n - 1 \leqslant 2n$ donc le nombre total de passages dans la boucle **tant que** *est un $\mathcal{O}(n)$. Comme les autres opérations sont en $\mathcal{O}(1)$ dans une boucle* **for** *de taille n on aboutit à une complexité linéaire.*