TD 5
Algorithmes de chaînes

About Ce problème est une annale d'examen (posé à Lille en 2011). Les exercices demandant du pseudo code seront effectués en pseudo code ou en C au choix.

Dans ce problème, nous allons coder une partie de l'algorithme de compression `bzip`, en fait la partie préparatoire du texte, qui consiste à faire apparaître des caractères identiques dans le texte à compresser. La partie compression des données n'est pas abordée.

1 - Transformation de Burrows-Wheeler

Cette transformation réalise une permutation des lettres du mot d'entrée (c'est-à-dire que toutes les lettres du mot d'entrée s'y retrouveront). Cette transformation a deux caractéristiques :

— dans le mot de sortie les répétitions de lettres identiques sont plus fréquentes.

— elle est bijective, c'est-à-dire qu'il existe une transformation inverse qui calcule à partir du mot transformé, le mot initial.

L'objet de cette partie est de calculer le codage d'un mot quelconque par la transformée.

Soit w un mot quelconque de taille ℓ, par exemple `banane`, qui est de taille 6. Dans un premier temps, on ajoute un marqueur de fin de mot, ♯ (on considère que ♯ n'est pas un caractère du mot). Ensuite, on construit une matrice qui contient toutes les permutations **circulaires** de ce texte, en décalant successivement le mot d'entrée vers la droite (matrice de gauche du dessin5.1).

$$\begin{pmatrix} b & a & n & a & n & e & \sharp \\ \sharp & b & a & n & a & n & e \\ e & \sharp & b & a & n & a & n \\ n & e & \sharp & b & a & n & a \\ a & n & e & \sharp & b & a & n \\ n & a & n & e & \sharp & b & a \\ a & n & a & n & e & \sharp & b \end{pmatrix} \qquad \begin{pmatrix} \sharp & b & a & n & a & n & \mathbf{e} \\ a & n & a & n & e & \sharp & \mathbf{b} \\ a & n & e & \sharp & b & a & \mathbf{n} \\ b & a & n & a & n & e & \mathbf{\sharp} \\ e & \sharp & b & a & n & a & \mathbf{n} \\ n & a & n & e & \sharp & b & \mathbf{a} \\ n & e & \sharp & b & a & n & \mathbf{a} \end{pmatrix}$$

FIGURE 5.1 – Matrice des permutations circulaires, puis la même, triée, et en gras le mot résultat.

Ensuite, les **lignes** de la matrice ainsi obtenue sont triées par ordre alphabétique (ordre du dictionnaire), en considérant les lignes comme des mots, ♯ étant inférieur à tous les autres caractères. Notre matrice devient alors la matrice de droite de la figure 5.1. (En effet, la ligne 1 : `anane#b` est inférieure à la ligne 2 : `ane#ban` car les deux premiers caractères sont identiques et le caractère 'a' est inférieur au caractère 'e'.)
Enfin, le mot résultat de la transformation est la dernière colonne de la matrice, ici en **gras**, c'est-à-dire `ebn#naa`.

Tous les tableaux de caractère (chaînes) auront une taille fixée à N (constante), N considéré toujours plus grand que la taille du mot à coder. Les tableaux de chaînes de caractères seront des matrices de caractères de taille $N \times N$. On rappelle qu'une chaîne de caractère possède un marqueur de fin, le caractère \0.

EXERCICE #1 ▶ **Questions simples, à préparer en avance**

1. Soit w le mot suivant (pour cette question on suppose N=9) :

h	e	l	l	o	\ 0			

Quelle est la taille du mot w ?

2. Calculer la transformée du mot `ima`.*On demande les calculs et le résultat, pas un algorithme*
3. Écrire en C une fonction `taille` qui prend en entrée une chaîne de caractères et qui renvoie sa taille.
4. Écrire en C une action `copie_tab` qui prend en paramètre deux chaînes de caractères, un entier ℓ, et qui copie la première chaîne dans la seconde sachant que la taille du mot est ℓ. *On n'oubliera pas de mettre le caractère de fin de chaîne à la bonne place.*
5. Écrire en C une action `decalage_droite` qui prend en entrée deux chaînes de caractères, un entier ℓ (la taille de la première chaîne), et qui modifie la deuxième chaîne de façon à ce qu'elle contienne la première chaîne décalée vers la droite. *Sur notre exemple, le décalé de* `banane#` *est* `#banane`, *le décalé de* `#banane` *est* `e#banan`.
6. Écrire en C une action `imprime_mat_carree` qui étant donnés une matrice mat de caractères de taille $N \times N$, et un entier ℓ imprime les caractères $mat[i][j]$ avec $0 \leq i \leq \ell - 1$ et $0 \leq j \leq \ell - 1$. *Cette fonction nous sera bien utile, car on va encoder des mots de taille quelconque dans des lignes de taille N, et on ne veut pas imprimer des caractères en trop.*
7. Écrire en C une action `mat_permut` qui étant donnés un mot d'entrée t, une matrice mat de caractères de taille $N \times N$, et un entier ℓ (taille du mot d'entrée), remplit la matrice mat avec les permutations du mot t. *On pourra s'apercevoir que $mat[i]$ est un tableau de taille N, et il est fortement recommandé d'utiliser les actions/fonctions précédentes.*
8. Écrire un programme **C complet** : main, fonctions (déclarations, et code avec des pointillés) , etc :
   - Inclusion des librairies `stdio.h` et `stdbool.h`.
   - Déclaration de la constante N valant 60
   - Déclaration et initialisation d'un tableau t contenant le mot `banane#`
   - Déclaration, construction et impression de la matrice de permutation de t.

Solution. 1 w a pour taille 5.

2 On commence par calculer les permutations circulaires de `ima#`, ce qui nous donne la matrice :

$$\begin{pmatrix} i & m & a & \sharp \\ \sharp & i & m & a \\ a & \sharp & i & m \\ m & a & \sharp & i \end{pmatrix}$$

Ensuite on trie par lignes, ce qui nous donne :

$$\begin{pmatrix} \sharp & i & m & a \\ a & \sharp & i & m \\ i & m & a & \sharp \\ m & a & \sharp & i \end{pmatrix}$$

La transformée est donc `am#i`

3 La fonction `taille` parcourt le mot jusqu'à rencontrer \0, en incrémentant un compteur :

```
int taille (char t[N]){
  int i=0;
  while(i<N && t[i] != '\0'){
    i = i+1;
  }
  return i;
}
```

4 Sans surprise, l'action `copie_tab` utilise une boucle "pour" pour parcourir en même temps les deux chaînes :

```
  void copie_tab (char t[N],char resu[N], int l){
```

```
  int i;
  for (i=0;i<l;i++){
    resu[i] = t[i];
  }
  if (l<N) resu[l] = '\0';
}
```

5 Cette fonction est une modification mineure de la précédente :

```
void decalage_droite (char t[N],char resu[N], int l){
  int i;
  for (i=1;i<l;i++){
    resu[i] = t[i-1];
  }
  resu[0] = t[l-1];
}
```

6 Pour imprimer, on parcourt la matrice ligne par ligne (avec deux boucles imbriquées), en faisant attention de s'arrêter aux indices $\ell - 1$:

```
void imprime_mat_carree(char mat[N][N], int l){
  int i,j;
  printf ("l= %d\n",l);
  for (i=0;i<l;i++){
    for (j=0;j<l;j++){
      printf("%c ",mat[i][j]);
    }
    printf("\n");
  }

}
```

7 La ligne 0 de la matrice mat va contenir une copie du tableau t, et ensuite chaque ligne $mat[i]$ contiendra le décalage à droite de la ligne précédente. On s'arrête lorsque l lignes ont été construites (boucle pour) :

```
void mat_permut(char t[N],char mat[N][N],int l){
  int i;
  copie_tab(t,mat[0],l);

  for (i=1;i<l;i++){
    decalage_droite(mat[i-1],mat[i],l);
  }
}
```

8 Il suffit de recoller les morceaux :

```
#include <stdio.h>
#include <stdlib.h>

#define N 60

int taille (char t[N]){
...
}
```

```
...

int main(){
  char t[N]=''banane#'';
  char mperm[N][N];

  int l = taille(t);
  mat_permut(t, mperm, l);
  imprime_mat_carree(mperm,l);

  return 0;
}
```

□

EXERCICE #2 ▸ **Complexité du codage**
Quelle est la complexité en nombre d'affectations de votre algorithme de construction de la matrice de permutations, en fonction de ℓ taille du mot initial ?

Solution. Chaque copie de tableau ou décalage coûte $O(\ell)$ affectations, et on réalise ℓ telles opérations. Le coût de la construction de la matrice est donc $O(\ell^2)$ affectations.

□

Maintenant, nous allons trier la matrice de permutation (en place, sans utilisation d'une matrice auxiliaire), par lignes, selon l'ordre alphabétique, toujours en considérant que *les lignes sont des mots*.

EXERCICE #3 ▸ **Tri de la matrice de permutation**

1. **En pseudo-code**, écrire une fonction `comp_tab_alpha` qui prend en argument deux chaînes de caractères de taille ℓ, ainsi que l'entier ℓ et qui retourne :
   — $r = 0$ si les deux chaînes sont identiques
   — $r = -1$ si la première chaîne est strictement inférieure (au sens de l'ordre alphabétique) à la deuxième chaîne.
   — $r = 1$ sinon.
2. **En pseudo-code**, écrire une action `echange_lignes` qui prend en paramètres une matrice mat de caractères de taille $N \times N$, deux entiers $i1$ et $i2$, un entier ℓ, et qui échange le contenu des lignes $i1$ et $i2$ de la matrice mat (en considérant que seules les cases d'indice 1 à $\ell - 1$ doivent être copiées dans une ligne). *On utilisera la fonction* `copie_tab` *de la question 4.*
3. En utilisant les deux fonctions précédentes, écrire un algorithme **en pseudo-code** qui étant donné une matrice mat de caractères de taille $N \times N$, ainsi que l'entier ℓ donnant la taille réelle de la sous-matrice à trier, trie la matrice en lignes par ordre alphabétique. *On modifiera un tri connu, par exemple le tri sélection, pour l'adapter au cas matriciel.*
4. Quelle est la complexité de votre algorithme précédent en nombre d'affectations ?
5. À l'aide de toutes les fonctions/actions écrites précédemment, écrire un algorithme **en pseudo-code** qui réalise le codage d'un mot initial quelconque (initialement sans marqueur de fin ♯). Quelle est la complexité de cet algorithme en nombre d'affectations ?

Solution. Les corrections données ici sont en pseudo-code. Néanmoins, les conventions utilisées (pas de copie de tableau, modification des tableaux passés en paramètre) sont les mêmes qu'en C.

— La fonction demandée va parcourir les chaînes $ch1$ et $ch2$ en parallèle :
  — on continue tant que les lettres sont identiques, et que l'on n'a pas obtenu la fin de $ch1$
  — si le parcours atteint la fin du mot $ch1$, cela signifie que les chaînes sont identiques, et la fonction retourne 0.
  — si il existe i tel que $ch1[i] < ch2[i]$ (resp $ch1[i] > ch2[i]$) alors on arrête le flot et on retourne -1 (resp 1).

En pseudo-code cela donne :

```
Fonction comp_tab_alpha(ch1,ch2,l) :Entier
    D: ch1,ch2 : Vecteurs[N] de caractères
    D: l : entier                                          {la taille de ch1,ch2}
    L: i : entier
    Pour i de 0 à l-1 Faire
        Si ch1[i]>ch2[i] alors
            Retourner 1
        Sinon
            Si ch1[i]<ch2[i] alors
                Retourner -1
    Retourner 0
```

(Faire un test pour voir si ça fonctionne)

— Pour échanger deux lignes dans une matrice, on prend un tableau auxiliaire qui permet d'en stocker une temporairement. Pour le reste on utilise `copie_tab` :

```
Action echange_lignes(mat,i1,i2,l)
    D: mat : Matrice[N][N] de caractères
    D: l : entier                                          {la "taille" des lignes}
    D: i1,i2 : entiers
    L: lignetmp : Vecteur[N] de caractères
    copie_tab(mat[i1],lignetmp,l)
    copie_tab(mat[i2],mat[i1],l)
    copie_tab(lignetmp,mat[i2],l)
```

— Comme l'énoncé nous le suggère, on va modifier le tri sélection sur les tableaux, en l'adaptant au cas d'une matrice à trier par ligne :

  — les comparaisons d'éléments sont remplacés par des comparaisons de lignes par ordre alphabétique ;

  — les échanges d'éléments sont remplacés par des échanges de lignes de la matrice.

Cela donne le pseudo-code-suivant :

```
Action trie_matrice_lexi(mat,l)
    D: mat : Matrice[N][N] de caractères
    D: l : entier                                  {la "taille" et le nb de lignes}
    L: i,k,indice_max : Entiers
    L: max_tmp : Vecteur[N] de caractères      {le max local est maintenant un tableau}
    Pour k de l-1 à 1 Faire
        copie_tab(mat[0],max_tmp,l)             {max_tmp ← mat[0] qui est une ligne !}
        indice_max ← 0
        Pour i de 1 à k Faire
            Si (comp_tab_alpha(mat[i],max_tmp,l)>0) alors
                                                                   {mat[i]>max_tmp}
                copie_tab(mat[i],max_tmp,l) ;                    {max_tmp ← mat[i]}
                indice_max = i ;
            ;
        echange_lignes(mat,indice_max,k,l) ;
```

— Chaque appel à la fonction d'échange coûte $O(\ell)$ affectations, et chaque copie aussi. Au pire, on effectue ℓ^2 copies, donc la complexité au pire est $O(\ell^3)$. Au mieux, la matrice est déjà triée, et la complexité est $O(\ell^2)$ affectations. Il est plus difficile d'obtenir la complexité moyenne…

— Il reste à recoller les morceaux, ce qui n'est pas très difficile. On écrit une action vu qu'il n'est pas

possible de retourner un tableau :

```
Action encodemot(motinit,motresu)
    D: motinit : Vecteur[N] de caractères
    R: motresu : Vecteur[N] de caractères
    L: mat : Matrice[N][N] de caractères
    motinit[tm] ←' #'                                   {ajout du marqueur}
    motinit[tm+1] ←' \0'
    tm ← tm+1
    mat_permut(motinit,thepermuts,tm)                   {construction de la matrice}
    trie_matrice_lexi(thepermuts,tm,tm)                 {tri}
    Pour i de 0 à tm -1 Faire
        motresu[i] ← thepermuts[i][tm-1]                {récup du mot codé}
```

Cet algorithme reste en $0(\ell^3)$ affectations vu que la dernière étape ne réalise que $O(\ell)$ affectations supplémentaires.

□

2 - Transformation de Burrows-Wheeler inverse

L'objet de cette partie est de calculer la transformée inverse. L'algorithme consiste à construire une matrice à l'aide du mot codé (voir la figure 5.2 pour les étapes) :

— On trie les lettres du mot codé (ici donc, `ebn\#naa`) par ordre alphabétique et on le stocke dans la colonne 0 de la matrice à construire.

— On ajoute à gauche de la matrice (en décalant la colonne déjà construite) le mot codé (étape (a) sur la figure). On trie par ordre alphabétique les *lignes de* la matrice obtenue. (étape (t))

— On recommence jusqu'à remplissage complet de la matrice. Le résultat (le mot décodé) est lisible sur la première ligne de la matrice.

$$\begin{pmatrix} \sharp & \cdot & \dots \\ a & \cdot & \dots \\ a & \cdot & \dots \\ b & \cdot & \dots \\ e & \cdot & \dots \\ n & \cdot & \dots \\ n & \cdot & \dots \end{pmatrix} \xrightarrow[(a)]{} \begin{pmatrix} e & \sharp & \dots \\ b & a & \dots \\ n & a & \dots \\ \sharp & b & \dots \\ n & e & \dots \\ a & n & \dots \\ a & n & \dots \end{pmatrix} \xrightarrow[(t)]{} \begin{pmatrix} \sharp & b & \dots \\ a & n & \dots \\ a & n & \dots \\ b & a & \dots \\ e & \sharp & \dots \\ n & a & \dots \\ n & e & \dots \end{pmatrix} \xrightarrow[(a)]{} \begin{pmatrix} e & \sharp & b & \dots \\ b & a & n & \dots \\ n & a & n & \dots \\ \sharp & b & a & \dots \\ n & e & \sharp & \dots \\ a & n & a & \dots \\ a & n & e & \dots \end{pmatrix} \xrightarrow[\dots]{} \begin{pmatrix} \sharp & \boldsymbol{b} & \boldsymbol{a} & \boldsymbol{n} & \boldsymbol{a} & \boldsymbol{n} & \boldsymbol{e} \\ a & n & a & n & e & \sharp & b \\ a & n & e & \sharp & b & a & n \\ b & a & n & a & n & e & \sharp \\ e & \sharp & b & a & n & a & n \\ n & a & n & e & \sharp & b & a \\ n & e & \sharp & b & a & n & a \end{pmatrix}$$

FIGURE 5.2 – Les différentes étapes du décodage

EXERCICE #4 ▸ **Décodage**

1. Écrire **en pseudo-code** un algorithme qui réalise l'ajout à gauche d'un mot t de taille ℓ dans une matrice m dont les j premières colonnes sont remplies.
2. Écrire **en pseudo-code** un algorithme qui calcule la transformée inverse d'un mot quelconque. *Bien détailler l'analyse* …
3. Quelle est la complexité de votre algorithme précédent ?

Solution. — Pour ajouter à gauche un vecteur dans une matrice, on commence par effectuer une copie de la colonne numero j vers la colonne $j+1$, de la colonne $j-1$ vers la colonne j, et ainsi de suite jusqu'à avoir décalé la colonne 0 vers 1. On est en mesure ensuite de copier le tableau t dans la première colonne :

```
Action ajoute_a_gauche(mot,mat,l,j)
    D: mot : Vecteur[N] de caractères (taille l)
    D: mat : Matrice[N][N] de caractères
    D: l,j :Entiers
    L: k,i :Entiers
    Pour k de j+1 à 1 (pas de -1) Faire
        {copie de la colonne k-1 dans la colonne k (l éléments)}
        Pour i de 0 à l-1 Faire
            mat[i][k] ← mat[i][k-1]
    Pour i de 0 à l-1 Faire
        {copie de mot dans la colonne 0}
        mat[i][0] ← mot[i]
```

— Pour faire la transformée inverse d'un mot, il faut pouvoir réaliser toutes les étapes écrites dans l'énoncé :

  — ajout à gauche (cf question précédente)

  — tri des lignes par ordre alphabétique : la fonction écrite à la question 11 doit être modifiée afin de pouvoir trier ℓ lignes de taille p avec $p < \ell$. En effet, à la première itération de l'algorithme, on trie la matrice selon les lignes de taille 1 ; à la deuxième itération, on trie des lignes de taille 2, etc.

Voici la nouvelle action de tri alphabétique :

```
Action trie_matrice_lexi2(mat,l,p)
    D: mat : Matrice[N][N] de caractères
    D: l : entier                                   {le nb de lignes}
    D: p : entier                                   {la taille des lignes}
    L: i,k,indice_max : Entiers
    L: max_tmp : Vecteur[N] de caractères
    Pour k de l-1 à 1 Faire
        copie_tab(mat[0],max_tmp,l)
        indice_max ← 0
        Pour i de 1 à k Faire
            Si (comp_tab_alpha(mat[i],max_tmp,p)>0)    {seule modif, ici p au lieu de l}
            alors
                copie_tab(mat[i],max_tmp,l);
                indice_max = i;
            ;
        echange_lignes(mat,indice_max,k,l);
```

Le reste est ensuite uniquement du recollage :

```
Action decodemot(motencode,motresu)
    D: motencode : Vecteur[N] de caractères
    R: motresu : Vecteur[N] de caractères
    L: mat : Matrice[N][N] de caractères
    L: l,j,i : Entiers
    l← taille(motencode)
    Pour i de 0 à l-1 Faire
        {copie dans la premiere colonne - étape 0}
        mat[i][0] ← motencode[i];
    trie_matrice_lexi2(mat,l,1)
    Pour j de 1 à l-1 Faire
        {ajout, décalage, ajout, ...}
        ajoute_a_gauche(motencode,mat,j,l)
        trie_matrice_lexi2(mat,l,j)
    Pour i de 1 à l-1 Faire
        {Le mot initial est sur la première ligne sauf #}
        motresu[i-1] ← mat[0][i];
    motresu[l] ←' \0'
```

— Chaque ajout à gauche coûte $O(\ell)$, chaque tri $O(\ell^2 j)$, donc la boucle du milieu coûte $O(\sum_{j=1}^{\ell} j\ell^2) = O(\ell^3)$. La complexité de l'algorithme en nombre d'affectations est donc $O(\ell^3)$. L'encodage et le décodage ont donc le même coût.

□