POLYTECH LILLE

Université Lille1 Sciences et Technologies

Département IMA / 3A (S5) **Programmation Structurée 2011/2012**

http://laure.gonnord.org/pro/teaching/

Examen - 2h

éléments de correction

A – Exercice : Programme Mystère

Question 1 Ce programme affiche sur le terminal :

```
1 2 3
1 3 2
2 1 3
2 3 1
3 2 1
3 1 2
```

Question 2 Chaque appel à `mysteriousRec(t,i)` avec $i \in [0, N-1]$ termine car chaque appel récursif est effectué avec un deuxième argument strictement plus grand qui finit par valoir $N-1$, cas terminal. Il en est de même pour `mysteriousRec(montab,0)`.

Question 3 Ce programme commence par initialiser un tableau avec les valeurs de 1 à N, et ensuite imprime sur le terminal toutes les permutations de ce tableau. Pour justifier, on peut par exemple prouver par récurrence descendante sur k la propriété suivante : $\mathcal{P}_k$ = "L'appel `mysteriousRec(montab,k)` affiche le contenu du tableau pour toutes les permutations de `montab[k]..montab[N-1]` (et laisse le tableau inchangé)".

- Pour $k = N-1$, la seule permutation du tableau est lui-même, et on affiche bien `montab`.
- Supposons $\mathcal{P}_{N-1}, \mathcal{P}_{n-2}, \ldots, \mathcal{P}_{k+1}$. Alors l'appel `mysteriousRec(montab,k)` :
  - (boucle for, avec j=k) l'appel à `mysteriousRec(montab,k+1)` affiche tous les tableaux constitués de : `montab[0]..montab[k]` suivi d'une permutation de `montab[k+1]..montab[N-1]` (hypothèse de récurrence)
  - (boucle for, j=k+1) échange des cases k et k+1 du tableau, donc l'appel `mysteriousRec(montab,k+1)` affiche tous les tableaux constitués de `montab[0]..montab[k-1]montab[k+1]` suivi d'une permutation de `montab[k]montab[k+2]..montab[N-1]` (hypothèse de récurrence). Ensuite l'échange inverse des cases est effectué.
  - (boucle for, j=k+2) échange des cases k et k+2 du tableau, donc l'appel `mysteriousRec(montab,k+1)` affiche tous les tableaux constitués de `montab[0]..montab[k-1]montab[k+2]` suivi d'une permutation de `montab[k]montab[k+1]montab[k+1]..montab[N-1]` (hypothèse de récurrence). Ensuite l'échange inverse est effectué
  - et ainsi de suite ... Finalement, on a bien $\mathcal{P}_k$.

B – Problème (BZIP)

1 - Transformation de Burrows-Wheeler

Question 0 w a pour taille 5.

Question 1 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`

Question 2 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;
}
```

Question 3 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';
}
```

Question 4 Cette fonction est une modif 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];
}
```

Question 5 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");
  }

}
```

Question 6 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);
  }
}
```

Question 7 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;
}
```

Question 8 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.

Question 9 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
| | | Fsi
| | Fsi
| Fpour
| Retourner 0
FFonction
```

(Faire un test pour voir si ça fonctionne)

Question 10 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)
FAction
```

Question 11 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 ;
            Fsi
            ;
        Fpour
        echange_lignes(mat,indice_max,k,l) ;
    Fpour
FAction
```

Question 12 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...

Question 13 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}
    Fpour
FAction
```

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 - 20 min

Question 14 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 Par 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]
|   Fpour
| Fpour
| Pour i de 0 à l-1 Faire
|                                                  {copie de mot dans la colonne 0}
|   mat[i][0] ← mot[i]
| Fpour
FAction
```

Question 15 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 ;
|   |   Fsi
|   |   ;
|   Fpour
|   echange_lignes(mat,indice_max,k,l) ;
| Fpour
FAction
```

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] ;
    Fpour
    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)
    Fpour
    Pour i de 1 à l-1 Faire
                                  {Le mot initial est sur la première ligne sauf #}
        motresu[i-1] ← mat[0][i] ;
    Fpour
    motresu[l] ←' \0'
FAction
```

Question 16 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.