POLYTECH LILLE

Université Lille1 Sciences et Technologies

Département IMA / 3A (S5) **Programmation Structurée 2010/2011**

Examen - Correction

A – Exercice : Schéma d'exécution

1. Écrire un algorithme qui prend en paramètre un caractère c et un entier x [...]
   CORRECTION :

   Action *exo(c,x)*
   | **D**: x :Entier
   | **D/R**: c :caractère
   | c ← (c + x) %256
   FAction

   ■

2. Écrire le main faisant un appel à votre algorithme [...]

   CORRECTION :

   Programme *Main()*
   | **L**: x :Entier,c :caractère
   | c ← 'a'
   | x ← -32
   | exo(c,x)
   | Imprimer(c)
   | **Retourner** *0* ;
   FProgramme

   ■

3. Mêmes questions en C.
   CORRECTION : La procédure en C utilise un pointeur pour passer le caractère par adresse :

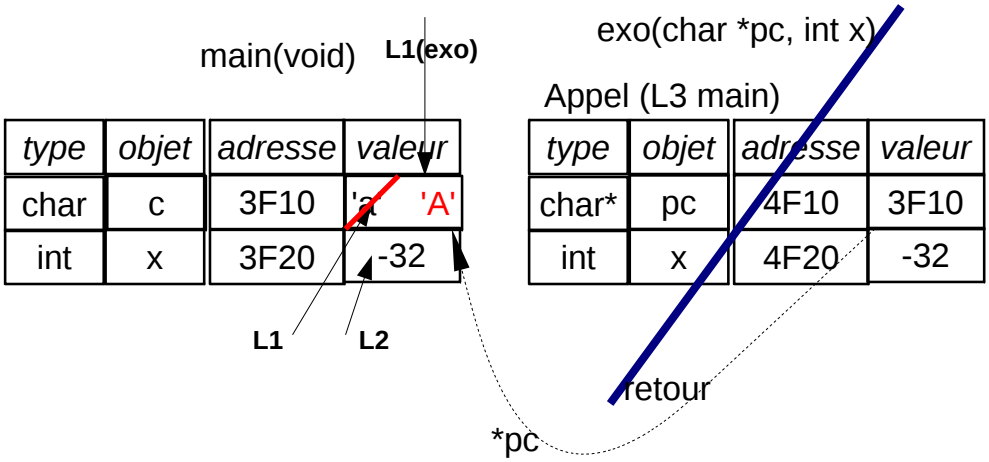
```
void exo(char* pc, int x)
{
  *pc = (*pc + x) %256;
}
```

   et dans le main, l'appel est fait à l'aide de l'adresse de caractère.

```
char c='a';
int x = -32;
exo(&c,x);
printf("exo 1 : c=%c\n",c);
```

   ■

4. Donner le schéma d'exécution de votre programme C.
   CORRECTION :



■

B – Problème - Recherche d'un mot dans un texte

Partie 1 - Méthode simple

Question 1 Quel est le suffixe numéro 3 du tableau `texte` contenant la chaîne "quelbonbon-bon" (celui qui est dessiné) ?

CORRECTION : `lbonbonbon` ■

Question 2 Quels sont les indices d'apparition du mot "bon" dans `texte` ?

CORRECTION : "bon" apparaît 3 fois : aux indices 4, 7, 10. ■

Question 3 Écrire une **action C** `afficheTexte` [...]

CORRECTION : Sans grande invention, on parcourt le tableau entièrement :

```
void afficheTexte(char texte[N])
{
  int i;
  for (i=0;i<N;i++)
    {
      printf("%c",texte[i]);
    }
}
```

■

Question 4 Écrire une **fonction C** nommée `enTetedeSuffixe` qui étant [...]

CORRECTION : La fonction retourne un booléen, et parcourt le mot tant que les lettres du mot sont égales à celles du texte (décalées de k) :

```
bool enTetedeSuffixe(char texte[N], char mot[N], int m, int k)
{
  bool resu = true;
```

```
  int i=0;
  while(resu && i<m)
    {
      if (mot[i]!=texte[k+i])
           resu = false;
      i++;
    }
  return resu;
}
```

■

Question 5 Écrire une **fonction C** nommée `apparaitV1`[...]

CORRECTION : Pour savoir si un mot apparaît, on regarde successivement si il apparaît en tête de chaque suffixe. Les indices des suffixes varient de 0 à $N - m$.

```
bool apparaitV1(char texte[N],char mot[N],int m)
{
  int dec = 0;
  bool trouve = false;

  while(!(trouve) && dec < N-m)
    {
      trouve = enTetedeSuffixe(texte,mot,m,dec);
      dec ++;
    }
  return trouve;
}
```

■

Question 6 Écrire une **fonction C** qui compte le nombre d'occurrences [...]

CORRECTION : Cette fonction est très similiaire à la précédente, mais cette fois on observe tous les décalages :

```
int nbOccurences(char texte[N],char mot[N],int m)
{
  int dec = 0;
  int  nbocc = 0;

   while(dec <= N-m)
    {
      if  (enTetedeSuffixe(texte,mot,m,dec))
        {
           printf("trouvé à  l'indice %d\n",dec);
           nbocc++;
        }
      dec ++;
    }
   return nbocc;
}
```

■

Question 7 Écrire un programme **C complet** (main, fonctions, etc) [...]

CORRECTION : Voici le programme C complet :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define N 13

//ici toutes les fonctions écrites avant.

int main(){
  char texte[N] = "quelbonbonbon";
  char mot[N]="bon";
  printf("nb occurrences = %d\n",nbOccurences(texte,mot1,3));

  return 0;
}
```

■

Question 8 Combien de comparaisons [...]

CORRECTION : Quel que soit le mot d'entrée (de taille m), l'algorithme fait $N - m$ fois appel à la fonction `enTetedeSuffixe`. Ensuite, le nombre de comparaisons de cette fonction dépend du mot :

- Si la première lettre du mot n'apparaît jamais dans le texte, à chaque fois la fonction `enTetedeSuffixe` fait une unique comparaison.
- Si le mot complet apparaît à chacun des décalages, on fait m comparaisons par décalages.

On a donc au mieux $O(N)$ comparaisons et au pire $O(N * m)$. ■

Partie 2 - Méthode avec tableau des suffixes

2.1 Construction du tableau des suffixes

Question 9 en pseudo-code Écrire une fonction `comparerSuffixes(texte,k1,k2)` [...]

CORRECTION : Pour comparer deux suffixes, on regarde chacune des lettres `texte[c1]` et `texte[c2]` pour c_1 commençant à k_1 et c_2 à k_2. On arrête lorsqu'une des lettres est différente (dans ce cas on retourne 1 ou -1), ou lorsqu'on a atteint la fin d'un des deux suffixes (dans ce cas il reste à vérifier de quel suffixe il s'agit).

```
Fonction comparerSuffixes(texte,k1,k2) :Entier
    D: texte : Vecteur[N] de caractères
    D: k1,k2 :entiers
    L: c1,c2 :entiers
    Si k1=k2 alors
        Retourner 0
    Fsi
    c1 ← k1
    c1 ← k2
    Tq c1 <N et c2<N faire
        Si texte[c1]<texte[c2] alors
            Retourner-1
        Sinon
            Si texte[c1]>texte[c2] alors
                Retourner 1
            Sinon
                c1 ← c1 + 1
                c2 ← c2 + 1
            Fsi
        Fsi
    Ftq
    Si c1<N alors
        Retourner 1
    Sinon
        Retourner-1
    Fsi
FFonction
```

■

Question 10 en pseudo-code. Construction du tableau des suffixes. L'algorithme va être une action, dont le but est d'initialiser le tableau des suffixes (qui est donc passé en D/R). Une première phase remplit le tableau `tabsuff` avec les entiers i de 0 à $N-1$. Ensuite, on trie ce tableau (ici on a utilisé le tri bulle) en utilisant comme critère de comparaison la fonction `comparerSuffixes` : on compare des suffixes d'indices voisins (j et $j-1$), et

on permute si ils ne sont pas dans le bon ordre.

```
Action tabSuffixes(texte,tabsuff)
| D: texte : Vecteur[N] de caractères
| D: tabsuff : Vecteur[N] d'entiers
| L: i,j,r :entiers
| Pour i de 0 à N-1 Faire
|  | tabsuff[i] ← i
| Fpour
| Pour i de 0 à N-2 Faire
|  | Pour j de N-1 à i+1 par pas de -1 Faire
|  |  | r ← comparerSuffixes(texte,tabsuff[j],tabsuff[j-1])
|  |  | Si r<0 alors
|  |  |  | permute(tabsuff,j,j-1) ;
|  |  | Fsi
|  | Fpour
| Fpour
FAction
```

Question 11 Complexité de la fonction précédente. À i fixé, on effectue toujours $N-i$ appels à la fonction de comparaison, qui coûte au mieux 1, au pire N. Donc on fait au mieux N^2 comparaisons , au pire N^3.

2.2 Utilisation pour la recherche

Question 12 **en pseudo-code** Écrire cet algorithme

CORRECTION : L'algorithme effectue des appels récursifs, et il s'arrête :
- lorsqu'on a trouvé un indice milieu tel que $ex[milieu]$ soit égal à l'élément que l'on recherche ;
- ou lorsqu'un tableau est réduit à 1 élément, on peut alors décider si cet élément est celui que l'on cherche puis retourner.

La structure est effectivement très similaire à celle du tri fusion, la recherche se faisant entre deux indices du tableau, que l'on passe aussi en paramètres.

```
Fonction recherche_DichoEntiersAux(tab,el,deb,fin) :Booleen
| D: tab : Vecteur[N] d'entiers
| D: el : entier
| L: milieu :entier
| Si deb<fin alors
| | milieu ← (fin+deb)/2
| | Si t[milieu] = el alors
| | | Retourner vrai
| | Sinon
| | | Si t[milieu] > el alors
| | | | Retourner recherche_DichoEntiersAux(tab,el,deb,milieu-1)
| | | Sinon
| | | | Retourner recherche_DichoEntiersAux(tab,el,milieu+1,fin)
| | | Fsi
| | Fsi
| Sinon
| | Retourner t[deb] = el
| Fsi
FFonction
```

On n'oublie pas de fournir la fonction de recherche "globale" :

```
Fonction recherche_DichoEntiers(tab,el) :Booleen
| D: tab : Vecteur[N] d'entiers
| D: el : entier
| Retourner recherche_DichoEntiers(tab,el,0,N-1)
FFonction
```

■

Question 13 en pseudo-code Écrire une fonction `comparerMotSuffixe(mot, texte, k)` [...]

CORRECTION :Pour cette fonction, on parcourt le mot jusqu'à la fin (en comparant $mot[i]$ et $texte[i + k]$) sauf si on peut décider si le suffixe est strictement plus petit ou strictement plus grand. Si l'on arrive à la fin du mot sans avoir retourné de résultat, c'est que le mot est égal au suffixe (et on retourne 0).

```
Fonction comparerMotSuffixe(texte,mot,k) :Entier
    D: texte,mot : Vecteur[N] de caractères
    D: k :entier
    L: i :entier
    i← 0
    Tq i<N et mot[i]≠ '\0' faire
        Si mot[i]<texte[k+i] alors
            Retourner-1 ;
        Sinon
            Si mot[i]>texte[k+i] alors
                Retourner 1
            Sinon
                i++ ;
            Fsi
        Fsi
        Retourner 0
    Ftq
FFonction
```

■

Question 14 en pseudo-code Recherche V2. On commence par écrire une recherche dichotomique. Ensuite la fonction recherche commence par construire le tableau des suffixes, puis appeler la recherche dichotomique. On modifie assez peu la recherche dichotomique afin que les types soient les bons. Pour avoir toutes les informations, il faut passer en paramètre le texte, le tableau des suffixes(tabsuff), le mot et sa taille(tmot), et les deux indices entre lesquels on fait la recherche.

```
Fonction rechercheDichoV2(texte,mot,tmot,tabsuff,deb,fin) :Booleen
    D: texte,mot : Vecteurs[N] de caractères
    D: tabsuff : Vecteurs[N] d'entiers
    D: deb,fin : entiers
    L: r,milieu :entiers
    Si deb<fin alors
        milieu ← (fin+deb)/2
        r = comparerMotSuffixe(mot,texte,tabsuff[milieu])
        Si r = 0 alors
            Retourner vrai
        Sinon
            Si r < 0 alors
                Retourner rechercheDichoV2(texte,mot,tmot,tabsuff,deb,milieu-1)
            Sinon
                Retourner rechercheDichoV2(texte,mot,tmot,tabsuff,milieu+1,fin)
            Fsi
        Fsi
    Sinon
        r = comparerMotSuffixe(mot,texte,tabsuff[milieu])
        Retourner r = 0
    Fsi
FFonction
```

Et finalement l'algorithme de recherche :

Fonction *rechercheV2(texte,mot,taillemot) :Booleen*
| **D**: texte,mot : Vecteur[N] de caractères
| **D**: taillemot : entier
| **L**: tabsuff :Vecteur[N] d'entiers
| tabSuffixes(texte,tabsuff)
| **Retourner** *rechercheDichoV2(texte,tabsuff,taillemot,tabsuff,0,N-1)*
FFonction

Question 15 Complexité de la solution précédente.

CORRECTION : La complexité de la recherche est la somme de la complexité de la construction du tableau des suffixes et de la recherche dichotomique. La complexité de la recherche dichotomique en nombre de comparaisons vérifie la loi :

$$C(N,m) = C_{compmot}(N,m) + C(\frac{N}{2},m)$$

avec $C_{compmot}$ le complexité de la fonction `comparerMotSuffixe`. Au pire, cette fonction effectue m comparaisons (taille du mot). La complexité à trouver devient : $C(N,m) = m + C(\frac{N}{2},m)$, qui se résout en $C(N,m) = O(mlogN)$ (à comparer au $O(Nm)$ de la recherche "naïve" de la partie 1. La construction du tableau des suffixes est cependant plus coûteux, cf question 9. On utilise donc cet algorithme lorsqu'on a plusieurs recherches à faire, le tableau des suffixes étant précalculé une seule fois au début.

■