

## Examen

### -I- Complexité moyenne (1+(1.5+1.5+2))

Considérons l'algorithme suivant:

```
algorithme
const N = ...
type tab = tableau[1..N] de caractère
var t: tab
fonction lecture(var t:tab) : entier
  debut
    pour i<--1 à N faire
      writeln('entrez une lettre de l'alphabet')
      readln(t[i])
      si t[i]='b' alors
        pour j<--1 à N faire
          res<--res+1
        finpour
      sinon
        si t[i]<> 's' alors
          res<--res+res+1
        fsi
      finsi
    finpour
  retourner(res)
fin
debut
writeln(lecture(t))
fin
```

- 1) Quel est le nombre d'additions du à l'exécution de *lecture* dans le pire des cas?
- 2) Il y a deux utilisateurs possibles Birgit et Rodolphe.
  - Birgit a tendance à entrer des 'b' ou des 's' : une lettre sur 2 est un 'b', une lettre sur 4 est un 's', les autres lettres sont équiprobables (il y a autant de chances d'entrer n'importe laquelle d'entre elles).
  - Rodolphe lui penche pour les 'r' ou les 's' : une lettre sur 3 est un 'r', une lettre sur 2 est un 's', les autres lettres sont équiprobables.

Quels sont les nombres d'additions (et les équivalents à l'infini) dans les cas moyens suivant :

- a) l'utilisateur est Birgit
- b) l'utilisateur est Rodolphe
- c) la probabilité que l'utilisateur soit Birgit est 0.75

NB: On aura intérêt à calculer le nombre moyen d'additions pour chaque itération de la boucle, puis en déduire le nombre moyen pour l'ensemble de l'exécution.

\$\$Correction :

1)  $n^2$   
2)

Si la lettre est 'b' il ya n operations,

Si la lettre est 's' il ya 0 operation,

Si la lettre est une autre lettre, il y a 2 opérations

A chaque iteration il ya donc en moyenne  $p('b')*n + p(\text{autre})*2$  operations. On a donc

$$nmoy(n) = n * (p('b')*n + p(\text{autre})*2)$$

a) Birgit :  $p('b') = 1/2$  et  $p('s') = 1/4$  donc  $p(\text{autre}) = 1/4$

On a donc

$$nmoy(n) = n * (1/2 * n + (1/4)*2) = n^2/2 + n/2 \text{ équivalent à } n^2/2$$

b) Rodolphe :  $p('b') = 1/3$  et  $p('s') = 1/2$  donc  $p(\text{autre}) = 1 - 1/2 - 1/3 = 1/6$

On a donc

$$nmoy(n) = n * (1/3 * n + (1/6)*2) = n^2/3 + n/3 \text{ équivalent à } n^2/3$$

d) La moyenne est la moyenne des moyennes :

$$nmoy(n) = 3/4 (n^2/2 + n/2) + 1/4 (n^2/3 + n/3) = 11 n^2/24 + 11 n/24$$

équivalent à  $11 n^2/24$

\$\$

## -II- vecteurs creux(( 3+5).

Nous considérons dans cet exercice que les vecteurs de réels que nous voulons manipuler sont creux, c'est à dire contiennent beaucoup de 0. Dans ce cas la représentation par tableau est bien moins économique qu'une représentation chaînée.

Ainsi le vecteur  $V$  suivant de taille  $N=25$  peut être représenté par une liste  $VL$  de cellules contenant chacune un indice dans le vecteur et la valeur correspondante.

Exemple :

$$V = (0, 0, 0, 1.5, 7.2, 0, 0, 0, 0, 0, 0, 0, 4.2, 0, 0, 0, 3.7, 0, 0, 0, 0, 0, 0, 0)$$

s'écrit aussi comme une liste de 4 éléments:

$$VL = ((4, 1.5), (5, 7.2), (14, 4.2), (18, 3.7))$$

Chaque cellule contient donc les champs Indice et Valeur et un champ Suivant. La liste chaînée est représentée par l'adresse de son premier élément. Nous supposons dans ce

qui suit que les vecteurs sont de taille constante N. On utilisera les déclarations suivantes:

```
const N = ...
type pcellule = ^cellule
      cellule =enregistrement
              val: reel
              ind:entier
              suiv:pcellule
      finEnregistrement
type vList = pcellule
```

1) Ecrire une fonction *ProduitScalaire(vL1, vL2)* qui calcule le produit scalaire des deux vecteurs sous forme de listes *vL1* et *vL2*

Rappel: le produit scalaire de deux vecteurs est la somme des produits des termes de même rang des vecteurs, par exemple :

$(1, 2, 0)$  ,  $(2, 0, 3) \rightarrow 1*2 + 2*0 + 0*3 = 2$

Avec la représentation sous forme de liste on a:

$VL1 = ((1,1), (2,2))$  ,  $VL2 = ((1,2), (3,3))$  et *PrduitScalaire(vL1, vL2)* renvoie 2

NB: Attention, la complexité de *ProduitScalaire* doit être (asymptotiquement) linéaire avec le nombre total *n* d'éléments dans *vL1* et *vL2*. (ci-dessus  $n = 4$ ).

Pour cela il faut avancer dans les deux listes *vL1* et *vL2* en même temps en comparant les indices. Si les deux indices sont identiques alors il faut ajouter au résultat le produit des deux valeurs et on avance dans les deux listes, sinon on avance dans la liste correspondant au plus petit indice.

Dans l'exemple précédent, on commence par calculer le produit des deux éléments d'indice 1, on avance dans les deux listes, puis dans *VL1* et le calcul est terminé car *VL1* est vide.

\$\$Correction.

```
fonction ProduitScalaire(vL1:vList, vL2:vList): entier
var p1, p2: pcellule
debut
  p1<--vL1
  p2<--vL2
  res<--0
  tantque p1<>nil et p2<>nil faire
    si p1^.ind = p2^.ind alors
      res<--res+ (p1^.val* p2^.val)
    sinon
      si p1^.ind< p2^.ind alors
        p1<--p1^.suiv
      sinon
        p2<--p2^.suiv
  fsi
```

```

    fintantque
    retourner(res)
fin

```

complexité : on avance au moins d'un élément à chaque itération jusqu'à épuiser les deux listes.

\$\$

2) Ecrire une procédure *PrintList* qui, à partir de la représentation chaînée *VL*, affiche le vecteur sous la forme "(0, 0, 0, 1.5, 7.2, 0, 0, 0, 0, 0, 0, 0, 4.2, 0, 0, 0, 3.7, 0, 0, 0, 0, 0, 0, 0). Cette procédure n'utilise pas de boucle mais fait appel à une procédure récursive *PrintListRec(i, vp)* (à écrire également) où *i* est l'indice de la dernière valeur déjà affichée, et *vp* représente l'adresse de l'élément courant dans la liste).

Exemple pour  $VL = ((4, 1.5), (5, 7.2), (14, 4.2), (18, 3.7))$  (en cours d'exécution):

.....

- affichage partiel:

(0,0,0,1.5, 7.2

- l'appel de *PrintListRec* se fait avec  $i=5$  et  $vp =$  adresse du 3<sup>ème</sup> élément de VL

ce qui a pour effet de compléter partiellement l'affichage avant l'appel récursif suivant :

(0, 0, 0, 1.5, 7.2, 0, 0, 0, 0, 0, 0, 0, 4.2,

.....

NB: attention à l'affichage du début et de la fin du vecteur.

\$\$Correction :

```

Procédure printListRec(i: entier; vp:pcellule)
var
    i : entier
debut
    si vp<> Nil alors
        (* on affiche les valeurs nulles *)
        i<--i+1
        tantQue vp^.ind <> i faire
            write('0,')
            i<--i+1
        fintq
        (* on affiche la valeur non nulle correspondant à
        l'élément d'adresse vp*)
        write (vp^.val)
        vp<--vp^.suiv
        si vp<>nil alors
            (* s'il reste des éléments dans la liste on
            ajoute une virgule et on affiche le reste
            des valeurs du vecteur par un appel
            récursif*)
            write(',')
            printListRec(i, vp)
        sinon
            (* on ajoute les dernières valeurs nulles à
            afficher *)

```

```

        tantQue i <= n faire
            write(',0')
            i<--i+1
        fintq

        fsi
    fsi
fin

Procédure printList(vL:vList)
debut
    write('(')
    printListRec(0,vL)
    write(')')
fin
$$

```

### -III- Arbres (1+3+2)

On suppose que l'on dispose d'arbres colorés : chaque noeud à un label indiquant sa couleur. Le nombre et le nom des couleurs n'est pas connu au départ. On utilise la représentation fils de gauche / frère de droite: Un arbre est représenté comme l'adresse de sa racine et chaque noeud est une cellule avec un lien *filDeG* vers son premier fils et un lien *frèreDeD* sur son premier frère.

Une liste de couleurs est une liste dont chaque élément est constitué d'une couleur et d'un entier supérieur à 0. Une couleur n'apparaît qu'une fois dans la liste. Les couleurs apparaissent par ordre alphabétique dans la liste.

La fusion des deux listes de couleurs  $C1$  et  $C2$  est une liste contenant les couleurs apparaissant dans  $C1$  ou  $C2$ , et leur associant la somme des valeurs entières correspondantes.

Exemple :

$$\begin{aligned}
 C1 &= \{("blanc", 2), ("rouge", 3), ("vert", 1)\} \\
 C2 &= \{("bleu", 3), ("rouge", 2), ("vert", 5)\} \\
 fusion(C1, C2) &= \{("blanc", 2), ("bleu", 3), ("rouge", 5), ("vert", 6)\}
 \end{aligned}$$

On suppose que la fonction *fusion* ( $C1, C2$ ) existe (donc inutile de l'écrire) et renvoie la liste résultant de la fusion. Attention cette fonction détruit les listes  $C1$  et  $C2$ .

0) Déclarer les types utiles ici.

1) Ecrire une fonction *RépertorieCouleurs* ( $p$ ) qui renvoie la liste de couleurs  $C$  dont chaque élément est constitué d'une couleur apparaissant dans l'arbre de racine  $p$  et du nombre de noeuds de l'arbre ayant cette couleur.

2) Ecrire une fonction *Dominante*( $p$ ) qui renvoie la couleur dominante de l'arbre de racine  $p$ , c'est à dire celle que l'on trouve sur le plus grand nombre de noeuds.

\$\$Correction

```

fonction repartitCouleur (p:noeud): pcellule
var  pf :pnoeud
     lc: pcellule
debut
  si p=nil alors
    retourner (nil)
  fsi
  new(lc)
  lc^couleur<--p^.couleur
  lc^.nombre =1
  lc^.suiv<--nil
  pf<- p^.filsDeG
  tantque pf <> nil faire
    lcfiles<-- repartitCouleur(pf)
    lc<--fusion(lc, lcfiles)
    pf<--pf^.frereDeD
  fintantque
  retourner(lc)
fin

fonction dominante(p:pnoeud):chaîne
(* suppose que p ne soit pas vide *)
var  cmax: chaîne
     pc :pcellule
     max :entier
     lc: pcellule
debut
  lc<--repartitCouleur(p)
  pc<-- lc
  max<--pc^.nombre
  cmax<--pc^.couleur
  pc<--pc^.suiv
  tant que pc<> nil faire
    si pc^.nombre>max alors
      max<-- pc^.nombre
      cmax<-- pc^.couleur
    fsi
    pc<--pc^.suiv
  fintq
retourner(cmax)
fin

$$

```