Deug Mias-2
Informatique
Institut Galilée
partiel de novembre 2000

-

-I- Ensembles et Listes Triées

Nous utiliserons pour représenter une liste un enregistrement contenant un champ "dernier" indiquant la position du dernier élément de la liste dans un tableau, et le champ "contenu" qui représente ce tableau. Le premier élément est toujours en position 1 . Un ensemble vide est représenté par une liste vide, c'est à dire que le champ "dernier" vaut 0. On ne peut ranger plus de NMAX élément dans le tableau et donc les listes sont de taille au plus égale à NMAX (dans la suite on ne testera pas le dépassement)

1) Ecrire une procédure itérative *append(Ed,k,Er)*qui ajoute les éléments entre le rang *k* et le rang *Ed.dernier* de la liste *Ed* à la liste *Er*.
Exemple : *Ed={7, 8, 9} , Er = {2, 4, 6},* après *append(Ed,2,Er) on a Er= {2, 4, 6,8,9}*

2) Ecrire une version récursive de *append* et dire en une phrase le principe du raisonnement récursif.

3) Nous considérons ici des ensembles d'entiers naturels et nous cherchons à écrire des fonctions correspondant à des opérations simples sur ces ensembles. Un ensemble d'entiers sera représenté par une liste triée par ordre croissant de ses éléments.
Nous voulons ici construire la réunion de deux ensembles E1 et E2 . Le résultat EU sera représenté par une troisième liste (le calcul ne modifie donc pas E1 et E2).
Nous supposons que E1 et E2 n'ont pas d'éléments en commun.
Exemple: *E1={1, 3, 5}, E2 = {2, 4, 6}* --> *EU = {1, 2, 3, 4, 5, 6}*

a)Ecrire une procédure itérative *Fusion(E1, E2, EU)* qui calcule la réunion *EU* de *E1* et *E2*. Le principe de base est d'avancer dans les deux listes *E1* et *E2* : comparer les deux éléments courants, ajouter le plus petit des deux éléments dans la liste *EU* et passer à l'élément suivant dans la liste où se trouvait cet élément (celui-ci devient le nouvel élément courant de cette liste). Attention quand une des deux listes est épuisée, il convient d'ajouter (avec *append*) les éléments restants de l'autre liste à la liste *EU*.

b) Ecrire une version récursive *FusionRec* de *Fusion*. Donner l'arbre des appels (les appels à *FusionREc* et à *append*) pour *E1={1,4}, E2={2,3}*

Correction :

```
1) Procédure append(var Ed :ens ; k :entier ; var Er :ens)
        debut
        (*ajoute les éléments entre  le rang k et le rang Ed.dernier de la liste Ed à
    la liste Er*)
        tantque k ≤ Ed.dernier faire
```

```
                Er.dernier<-- Er.dernier +1
                Er.contenu[Er.dernier]<-- Er.contenu[k]
                k<--k+1
        ftq
        fin
```

2) Procédure appendR(var Ed :ens ; k :entier ; var Er :ens)

```
        debut
        (*ajoute les éléments entre  le rang k et le rang Ed.dernier de la liste Ed à
la liste Er*)
        si k ≤ Ed.dernier alors
                Er.dernier<-- Er.dernier +1
                Er.contenu[Er.dernier]<-- Ed.contenu[k]
                append(Ed,k+1, Er)
        fsi
        fin
```

Ajouter les k derniers éléments de E1 à E2 revient à ajouter le $k^{ième}$ élément de E1 à E2 puis à ajouter les k+1ème derniers de E1 à E2.

3) Certains étudiants ont pu croire que (1,5, 7, 5, 1) (cas impair) etait un palindrome. Nous acceptons cette interprétation, mais donnons ici la solution lorsqu'on considère qu' avec un nombre impair d'éléments il ne s'agit pas d'un palindrome. EN revanche une liste vide est un palindrome.

Fonction Palindrome(var Ed :ens): Booléen

```
        var k : entier
        debut
        (renvoie vraie si ED est un palindrome (avec un nombre pair d'
éléments*)
        si Ed.dernier mod 2 = 1 alors
                retourner (Faux)
        finsi
        si Ed.dernier = 0 alors
                retourner (Vrai)
        finsi
        k<-- 1
        tantque         (k ≤ Ed.dernier div 2)  ET
                        ( Ed.contenu[k]= Ed.contenu[n-k+1]) faire
                k<--k+1
        ftq
        si k ≤  Ed.dernier div 2  alors
                 (* la dernière comparaison a donné Faux*)
                retourner(faux)
        finsi
        (*on est sorti parceque k  >  Ed.dernier div 2*)
        retourner(vrai)
        fin
```

2) Fonction Palindrome(var Ed :ens): Booléen

```
        debut
        (*renvoie vraie si ED est un palindrome (avec un nombre pair d'
éléments*)
```

```
        si Ed.dernier mod 2 = 1 alors
            (* Ed de longueur impaire donc n'est pas un palindrome*)
            retourner (Faux)
        finsi
        retourner(palindromeRec(Ed,1, Ed.dernier)
        fin

Fonction PalindromeRec(var Ed :ens; idebut :entier; ifin :entier): Booléen
        debut
        (*renvoie vrai si Ed est un palindrome (avec un nombre pair d'
    éléments)  entre l'élément d'indice idebut et l'élément d'indice ifin)
        si ifin < idebut alors
            (* la liste centrale à examiner est vide donc est un palindrome *)
            retourner (vrai)
        finsi
        si Ed.contenu[idebut] <> Ed.contenu[ifin] alors
            (*  Ed entre ideb et ifin ne peut plus être un palindrome *)
            retourner(faux)
        finsi
        (* Ed est un palindrome entre ideb et ifin si c'est un palindrome entre
ideb+1 et ifin-1*)
        retourner (PalindromeRec(Ed, ideb+1, ifin -1))
        fin
```

Ed est un *palindrome entre l'élément d'indice idebut et l'élément d'indice ifin*
soit -si il n'y a pas d'éléments entre idebut et ifin (ifin > idebut)
soit -si la valeur en idebut est la même que la valeur en ifin ET Ed est un *palindrome entre l'élément d'indice idebut+1 et l'élément d'indice ifin-1*

Ex: (5, 3, 1, 1, 3, 5) est un palindrome parceque le premier et le dernier élément sont égaux et que la liste tronquée de ces deux éléments: (3, 1, 1, 3) est un palindrome.

-II- Nombre de parties de k éléments d'un ensemble

Nous allons écrire une fonction *Parties(k, n)* qui renvoie le nombre de parties de *k* éléments d'un ensemble de *n* éléments. Pour cela nous allons utiliser les formules de récurrences suivante:
Parties(k,n) = Parties(k, n-1) + Parties (k-1, n-1)
Parties (n,n) = Parties(0,n) =1
Parties(1,n) = n

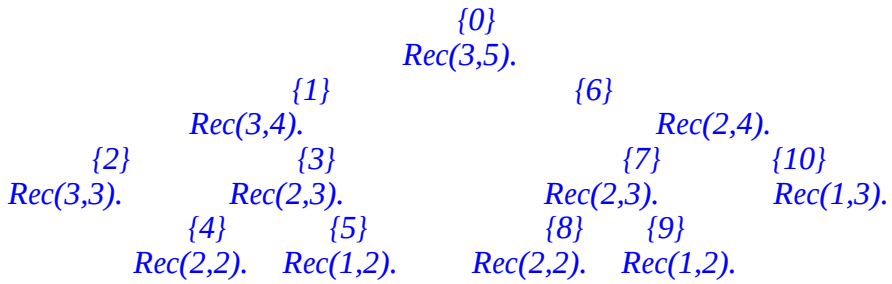
NB: Elles proviennent de la remarque suivante : si on fixe un élément *e* parmi les *n*, les parties de *k* éléments se partagent en deux catégories : celles qui contiennent *e* (il reste alors *k-1* éléments à fixer parmi *n-1*) et celles qui ne contiennent pas *e* (il reste *k* éléments à fixer parmi *n-1*)

1) Ecrire la fonction récursive *PartiesRec(k,n)* et décrire l'arbre des appels de *PartiesRec(3,5)* en numérotant les appels selon l'ordre chronologique.
2) Ecrire une version (très) économique *Parties (k,n)* qui appelle une fonction récursive *PartiesRecM(k,n)* (à écrire également). *Parties* a un tableau local *M* bidimensionnel où sont rangés les résultats des appels de *PartiesRecM*, ce qui

évite de refaire plusieurs fois le même appel. Décrire également ici l'arbre des appels déclenchés par *Parties (3,5)*

Correction :

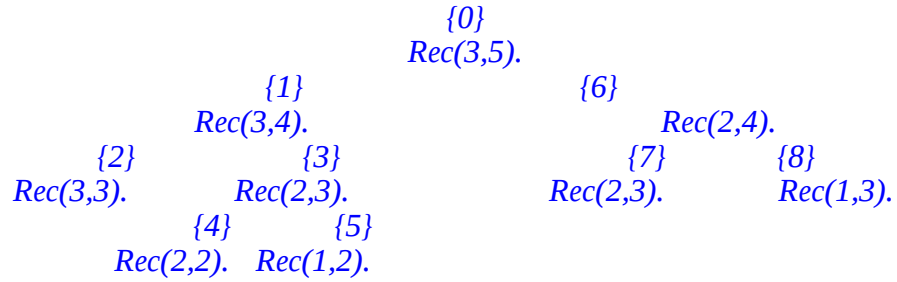
```
fonction PartiesRec(k:entier; n:entier): entier
        début
        (*contrainte : k ≤ n *)
        si k =0 OU k=n alors
                retourner(1)
        fsi
        si k=1 alors
                retourner(n)
        fsi
        retourner (partiesRec(k, n-1) + partiesRec(k-1, n-1))
        fin
```




```
fonction PartiesRec(k:entier; n:entier; varM: Memoire): entier
        début
        (*contrainte : k ≤ n *)
        si k =0 OU k=n alors
                retourner(1)
        fsi
        si k=1 alors
                retourner(n)
        fsi
        si M[k,n]=0 alors
                M[k,n]<-- partiesRec(k, n-1) + partiesRec(k-1, n-1))
        fsi
        retourner(M[k,n]
        fin

fonction Parties (k:entier; n:entier): entier
        (* type Memoire = tableau[1..Nmax, 1..Nmax] de entier*)
        (* k et n doivent ≤ Nmax, et k≤ n *}
        var kk, nn: entier
        début
        pour kk<--1 à k faire
                pour nn<--kk+1 à n faire
                        (*M sera toujours utilisé pour des valeurs telles que  kk
                        est < nn donc on ne met à 0 que les valeurs
                        correpondantes de M*)
```

```
                M[kk,nn]<--0
            fpour
        fpour
        fin
```



-III-Mystère

Soit un tableau de *N* réels *T*. Le système ci-dessous défini le tableau *TL* à partir du tableau *T*:

--

TL[1] = (T[1]+T[2]+...+T[k])/k

TL[p] = TL[p-1] + (-T[p-1] + T[p+k-1])/k

pour *p>1* et *p ≤n-k+1*

*TL[p] = (TL[p-1] - (T[p-1] / k-m+1)) * (k-m+1/ k-m)*

pour *p = n-k+1+m* et *m* variant de *1 à k-1*

--

1) Que représente le tableau *TL* ? pourquoi y a t'il deux sortes d'équations de récurrence ?

2) Ecrire une procédure récursive *ConstruitL(T,TL,k)* et simulez l'appel de *ConstruitL(Tor,Tres,2)* sur le tableau de3 éléments suivant :

Tor = { 1, 0 ,1 }

En réalité on construira un procédure directement récursive *ConstruitLRec(T,TL, k , p)* qui construit *TL* entre les indices *1* et *p*, puis la procédure *ConstruitL(T,TL,k)* qui fait un appel à *ConstruitLRec* pour construire le tableau TL en entier.

Correction :

1) Tl contient en chaque position p la moyenne T(p)+T(p+1) ... +T(p+k-1) / k sauf pour les dernières positions où la moyenne est faite sur les positions restantes jusqu'à n.

2)

```
Procédure ConstruitLRec(var T: tab; var TL :tab; k:entier; p:entier)
        var i : entier
        debut
        si p=1 alors
                TL[1]<--0
                pour i<--1 à k faire
```

```
                    TL[1]<-- TL[1]+T[i]
                fpour
                TL[1]<-- TL[1] / k
        fsi
        si  p>1  ET p  ≤n-k+1 alors
                ConstruitLRec(T,TL, k , p-1)
                TL[p] = TL[p-1] + (-T[p-1]  + T[p+k-1])/k
        fsi
        si  p>1  ET p  > n-k+1 alors
                 ConstruitLRec(T,TL, k , p-1)
                m<--p-n+k-1
                TL[p] = (TL[p-1] - (T[p-1] / k-m+1) ) * (k-m+1/ k-m)
        fsi
        fin
Procédure ConstruitL (var T: tab; var TL :tab; k:entier)
        debut
        ConstruitLRec(T,TL, k , N)
        fin

{ ConstruitL (Tor,Tres, 3 )}
        {T est Tor ={1,0,1} , Tres={?}, k=2}
        {ConstruitLRec(T,TL, k , N)
                {T est Tor ={1,0,1} , TL est Tres={?}, k=2, p=3}}
                {3=1 faux}
                {3>1 ET 3≤3-2+1 faux}
                {3>1 ET 3 >3-2+1 vrai}
                {ConstruitLRec(T,TL, k , p-1)}
                        {{T est Tor ={1,0,1} , TL est Tres={?}, k=2, p=2}}
                        {2=1 faux}
                        {2>1 ET 2≤3-2+1 vrai}
                        {ConstruitLRec(T,TL, k , p-1)}
                                {{T est Tor ={1,0,1} , TL est Tres={?}, k=2, p=1}}
                                {2=1 vrai}
                                {TL[1] = T[1]+T[2] /2 = 0.5}
                        {modif: TL={0.5,?}}
                        { TL[2] = TL[1] + (-T[1]  + T[2+2-1])/k = 0.5 + (-1 + 1)/2 = 0.5}
                {modif : TL={0.5,0.5,?}}
                m =3-3+2-1 = 1
                {TL[3] = (TL[2] - (T[2] / 2-1+1) ) * (2-1+1/ 2-1) = (0.5-(0/2))*2=1}
        {modif: TL={0.5,0.5,1}}
{modif : Tres={0.5,0.5,1}}
```