Option MP

EPREUVE D'INFORMATIQUE

Durée : 4 heures

L'usage des calculatrices est autorisé. Les algorithmes demandés peuvent être codés en CAML ou en PASCAL.
Quand aucune directive n'est donnée le candidat peut établir une version récursive ou non. Quel que soit le choix du langage, les algorithmes doivent être écrits de la manière la plus courte possible, parfaitement lisible, avec une indentation convenable, sans aucune rature et en ***respectant scrupuleusement les notations introduites****. Ils doivent être documentés par des explications concises et précises sur les points qui le nécessitent.*
La lecture de l'annexe *correspondant au langage choisi est vivement conseillée avant d'aborder les questions de programmation.*
Les réponses qui ne respecteraient pas les consignes précédentes ne seront pas prises en considération.
Les questions sont assez souvent indépendantes les unes des autres, il est cependant conseillé de respecter la chronologie proposée.
Les parties I, II et III sont totalement indépendantes.

I- Matrices et chemins dans un graphe.

Notations, définitions:

Soit $n \geq 1$ un entier, et $S=\{1, 2, ..., n\}$. On note M l'ensemble des matrices carrées, d'ordre n, à coefficients dans $\{0,1\}$.
Une matrice $A=(a_{ij}) \in M$ sert à représenter un graphe sur S : $a_{ij}=1$ signifie qu'il existe un arc reliant le sommet i au sommet j.
Soit $A=(a_{ij}) \in M$ et soient $i,j \in S$, un chemin de i vers j de longueur $m \geq 0$ (dans A) est une suite $(i_0, i_1, \cdots, i_m)$ d'éléments de S telle que $i_0=i$, $i_m=j$ et $a_{i_{k-1}, i_k}=1$ pour $k=1, 2, ..., m$.
Un élément j est accessible à partir de i si et seulement il existe un chemin de i vers j.
Sur l'ensemble M, on définit une addition notée $\oplus$ et une multiplication notée $\otimes$, comme suit:
soient A, B, C, D$\in M$. Alors $C=(c_{ij})=A \oplus B$ est défini par:

$$c_{ij} = \begin{cases} 0 \text{ si } a_{ij} = b_{ij} = 0 \\ 1 \text{ sinon} \end{cases}$$

et $D=(d_{ij})=A \otimes B$ est défini par :

$$d_{ij} = \begin{cases} 0 \text{ si } a_{ik} * b_{kj} = 0 \text{ pour tous } k \in S \\ 1 \text{ sinon} \end{cases}$$

On définit $A^{\otimes k}$ par $A^{\otimes 0}$=Id (Id est la matrice identité) et $A^{\otimes k+1} = A \otimes A^{\otimes k}$ pour $k \geq 0$. On note $a_{ij}^{(k)}$ l'élément d'indice (i,j) de $A^{\otimes k}$.

1) Montrer que $(Id \oplus A)^{\otimes n} = Id \oplus A \oplus A^{\otimes 2} \oplus \ldots \oplus A^{\otimes n}$.

2) Soient $i, j \in S$, on peut démontrer qu'il existe un chemin de i vers j de longueur k si et seulement si $a_{ij}^{(k)} = 1$.
En déduire un algorithme pour construire la matrice C des sommets accessibles, plus précisément c_{ij} est égal à 1 si et seulement si le sommet j est accessible à partir du sommet i (i.e il existe un chemin conduisant de i à j).On s'efforcera de minimiser le nombre d'opérations sur les matrices.

3) Ecrire les deux fonctions (procédures en PASCAL) `somme` et `produit` qui calculent respectivement $A \oplus B$ et $A \otimes B$.

4) Ecrire la fonction `accessible` (procédure en PASCAL) qui calcule la matrice des sommets accessibles. On demande ici de programmer l'algorithme établit à la question 2.

5) Ecrire la procédure `affiche_tous_les_chemins` qui pour un couple $i,j \in S$, affiche tous les chemins, du graphe décrit par la matrice $A=(a_{ij}) \in M$, allant du sommet i vers le sommet j. Les chemins affichés ne doivent comporter aucun cycle, i.e on ne peut passer qu'une et une seule fois par un sommet.

II- Recherche du k-ième plus petit élément.

Une façon simple trouver le k-ième plus petit élément parmi n éléments est de trier par ordre croissant ces éléments et de prendre l'élément qui se trouve en position k. Le but de cet exercice est d'établir un algorithme plus efficace.

On note T[g..d] la partie du tableau T dont les composantes ont un indice k compris entre g et d ($g \le k \le d$). Soit X la composante du tableau T ayant pour indice g (X se note T[g] en PASCAL et T.(g) en CAML).
On dit qu'on réalise une *coupe* de T[g..d] par rapport à X lorsqu'on réarrange T[g..d] de manière à ce que l'élément X **soit à sa place** dans T[g..d].
Plus précisément, T[g..d] est resté globalement invariant et si **p** ($g \le p \le d$) désigne le **nouvel emplacement de X**, tous les éléments ayant un indice compris entre g et p sont **plus petits ou égaux** à X, tous les éléments ayant un indice compris entre p+1 et d sont **strictement plus grand** que X.
On n'exige pas que les deux parties de la *coupe* soient triées.

1) Proposer un algorithme qui, sans utiliser de tableau auxiliaire, parcours une seule fois T[g..d] pour réaliser une *coupe* de T[g..d]. L'unique opération permise est l'échange de deux composantes.

2) En utilisant l'algorithme précédent établir un algorithme qui permette de trouver le k-ième plus petit élément d'une suite de n éléments.

3) Ecrire la fonction `k_ieme` qui renvoie comme résultat le k-ième plus petit éléments du tableau T ayant `n` éléments.

4) En supposant que chaque *coupe* divise par deux la taille du sous-tableau dans lequel se trouve l'élément recherché, combien faut-il faire de comparaisons dans le pire des cas pour déterminer cet élément ? Qu'en pensez-vous ?

III- Décomposition d'un nombre entier en somme de nombres de Fibonacci.

Les nombres de Fibonacci sont définis par :

$$F_0 = 0, \quad F_1 = 1, \qquad F_{n+2} = F_{n+1} + F_n, \quad n \ge 0$$

Notation : k et m étant des nombres entiers, k>>m signifie que $k \ge m+2$.

On démontre que tout entier n a **une unique** décomposition : $n = F_{k_1} + F_{k_2} + \cdots + F_{k_r}$ où $k_1 >> k_2 >> \cdots >> k_r >> 0$.

1) Démontrer que $F_{k+1} = F_k + F_{k-2} + F_{k-4} + \cdots + F_{k-2i} + \cdots + 1$ pour tous les indices tels que $k-2i \geq 2$.

En utilisant cette propriété on peut déduire assez facilement la décomposition de n+1 à partir de celle de n.
En effet, soit $n = F_{k_1} + F_{k_2} + \cdots + F_{k_r}$ et soit $j \in [1,r]$ le plus petit entier tel que $\forall s,\ s \in [j,r[\Rightarrow k_{s+1} = k_s - 2$. Autrement dit, dans la décomposition de n les indices diminuent de deux en deux, vers la droite, seulement à partir de k_j :

$$n = F_{k_1} + \cdots + F_{k_{j-1}} + (F_{k_j} + F_{k_j-2} + F_{k_j-4} + \cdots + F_{k_r})$$

On distingue trois cas :

1- Si $k_r > 3$ alors $n+1 = F_{k_1} + F_{k_2} + \cdots + F_{k_r} + F_2$.

2- Si $k_r \leq 3$ et j=1 alors $n+1 = F_{k_1+1}$.

3- Si $k_r \leq 3$ et j>1 alors $n+1 = F_{k_1} + F_{k_2} + \cdots + F_{k_{j-1}} + F_{k_j+1}$.

Ainsi, sachant que $1 = F_2$ on déduit :
$1 = F_2$ (cas 2) -> $2 = F_3$ (cas 2) -> $3 = F_4$ (cas 1) -> $4 = F_4 + F_2$ (cas 2) -> $5 = F_5$ (cas 1) -> $6 = F_5 + F_2$ (cas 3) -> $7 = F_5 + F_3$ (cas 2) -> $8 = F_6$ (cas 1) -> ...

Dans ce qui suit on représentera la décomposition d'un entier n par le couple (`j`, `lst`) où `lst` est la liste ordonnée (par ordre décroissant) des indices des nombres de Fibonacci qui interviennent dans la décomposision de n et `j` est l'entier qui indique le rang à partir duquel les éléments de `lst` diminuent de deux en deux.

Exemples :

$n=33 = F_8 + F_6 + F_4 + F_2$ sera représenté par le couple (1, [8; 6; 4; 2]).
$n=40 = F_9 + F_5 + F_2$ sera représenté par le couple (3, [9; 5; 2]).

2) Ecriture de quelques fonctions qui peuvent s'avérer utiles pour la suite.

- Ecrire la fonction `ieme` qui renvoie le i-ème élément d'une liste `lst`.

- Ecrire la fonction `tete` qui renvoie une nouvelle liste construite en recopiant les `i` premiers éléments de la liste `lst`. Si cette sous-liste n'existe pas la fonction renvoie la liste vide.

- Ecrire la fonction `queue` qui renvoie une nouvelle liste construite en recopiant les éléments de la liste `lst` à partir du rang `i`. Si cette sous-liste n'existe pas la fonction renvoie la liste vide.

Exemples :

Si `lst`=[10; 8; 6; 2] et si `i`=3, `ieme` donne pour résultat 6, `tete` donne [10 ; 8 ; 6], `queue` donne la liste [6; 2],

3) Ecrire la fonction `enumere` qui à partir de la décomposition $1 = F_2$, engendre les décompositions en somme de Fibonacci de tous les entiers jusqu'à n. Le résultat de cette fonction doit être une liste de couples :

[(1, [2]) ; (1 , [3]) ; (1, [4]) ; (1, [4; 2]) ; (1, [5]) ; (2, [5; 2]) ; (1, [5; 3]) ; (1, [6]) ;.....]

4) Soit $n = F_{i_1} + F_{i_2} + \cdots + F_{i_s}$ où $i_1 \geq i_2 \geq \cdots \geq i_s \geq 2$, une somme quelconque de nombres de Fibonacci.
Ecrire la fonction `decomposition` qui renvoie la liste des indices des nombres de Fibonacci de l'unique décomposition de $n = F_{k_1} + F_{k_2} + \cdots + F_{k_r}$ où $k_1 >> k_2 >> \cdots >> k_r >> 0$. Si la fonction `decomposition` reçoit pour paramètre effectif la liste $[i_1; i_2; \cdots; i_s]$, elle retourne pour résultat la liste $[k_1; k_2; \cdots; k_r]$.

5) Ecrire la fonction `somme` qui prend en arguments deux listes correspondant à l'unique décomposition de deux entiers et retourne pour résultat la liste correspondant à l'unique décomposition de la somme de ces deux entiers.

ANNEXE PASCAL

I) Matrices et chemins dans un graphe.

Déclarations :

```
type   matrice = array[1..10,1..10] of byte;
```

En-tête des procédures demandées :

```
procedure somme(A,B : matrice ; var C : matrice ; n :integer ) ;
procedure produit(A,B : matrice ; var C : matrice ; n :integer  ) ;
procedure accessible(A : matrice ; var C : matrice ; n :integer ) ;
procedure affiche_tous_les_chemins(.....................) ;
```

II) Recherche du k-ième plus petit élément.

Déclarations :

```
type      table = array[1..100] of integer ;
```

En-tête de la fonction demandée :

```
function k_ieme ( T :table ; k ,n: integer ) :integer ;
```

III) Décomposition d'un nombre entier en somme de nombre de Fibonacci.

Déclarations :

```
type
        liste = ^doublet ;
      doublet = record
                    ind : integer ;
                   suiv : liste;
                end;
       couple = record
                     j : integer ;
                   lst : liste ;
                end ;
  listecouples = ^noeud ;
        noeud = record
                     c : couple ;
                  suiv : listecouples;
                end ;
```

En-tête des fonctions demandées :

```
function i_eme (i:integer ;lst :liste) :liste ;
function tete  (i:integer ;lst :liste) :liste ;
function queue (i:integer ;lst :liste) :liste ;
function enumere(n :integer) :listecouples ;
function decomposition(lst :liste) :liste ;
function somme(l1,l2 :liste) :liste ;
```

Fonctions disponibles :

Les fonctions ci-dessous **sont données et peuvent être utilisées** :

```
function longueur(lst :liste) :integer ;
function copie(lst :liste) :liste ;
function concatene(l1,l2 :liste) :liste ;
function mp(elmt :integer ;lst :liste) :liste ;
function insere(elmt :integer ;lst :liste) :liste ;
```

La fonction fonction `longueur` rend le nombre d'éléments de la liste `lst`.
La fonction fonction `copie` rend une liste identique à la liste `lst` (création d'une nouvelle liste).
La fonction fonction `concatene` rend la liste concaténée des listes `l1` et `l2` (création d'une nouvelle liste).
La fonction fonction `mp` rend une liste en ajoutant `elmt` en tête d'une liste `lst` (création d'une nouvelle liste).
La fonction fonction `insere` insère `elmt` à sa place dans une liste triée `lst` (création d'une nouvelle liste).

ANNEXE CAML

I) Matrices et chemins dans un graphe.

Liste des fonctions demandées :

```
somme :int vect vect->int vect vect->int vect vect=<fun>
produit :int vect vect->int vect vect->int vect vect=<fun>
accessible :int vect vect->int vect vect=<fun>
affiche_tous_les_chemins :.....->unit=<fun>
```

II) Recherche du k-ième plus petit élément.

Fonction demandée :

```
kieme :int vect->int->int=<fun>
```

III) Décomposition d'un nombre entier en somme de nombre de Fibonacci.

Liste des fonctions demandées :

```
ieme :int->'a list->'a=<fun>
tete :int->'a list->'a list=<fun>
queue :int->'a list->'a list=<fun>
enumere :int->(int*int list)list=<fun>
decomposition :int list->int list=<fun>
somme :int list->int list->int list=<fun>
```

Quelques fonctions de la bibliothèque CAML :

`list_length` : `'a list ->int` renvoie le nombre d'éléments de la liste donnée.
`hd` : `'a list -> 'a` renvoie le premier élément de la liste donnée.
`tl` : `'a list -> 'a list` renvoie la liste donnée privée de son premier élément.

On dispose également de la fonction :

`insere: 'a->'a list->'a list=<fun>` qui insère un élément `elmt` à sa place dans une liste triée `lst`.