ESIM

SESSION DE 1999

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 indépendantes.

I- Enumérations d'ensembles.

E et F désignent des ensembles finis. On se propose dans cette partie d'écrire des fonctions qui permettent d'énumérer des ensembles tels que :
L'ensemble de toutes les parties de E, l'ensemble des parties à p éléments de E, l'ensemble des permutations sur E, l'ensemble des bijections de E dans F, l'ensemble des surjections de E sur F.
Bien que les notions d'ensemble et de liste soient différentes, **on choisit ici de représenter un ensemble par une liste,** par la liste des ses éléments. Plus précisément E et F sont considérés dans cette partie comme des listes de chaînes de caractères.

Si l'ensemble E est représenté par la liste [*a b c*] et l'ensemble F par la liste des deux caractères [*1 2*] alors :

- L'ensemble des parties de E peut être représenté par la liste de listes suivante :[[] [*a*] [*b*] [*c*] [*a b*] [*a c*] [*b c*] [*a b c*]]

- L'ensemble des permutations sur E peut être représenté par la liste de listes :
[[*a b c*] [*a c b*] [*b a c*] [*b c a*] [*c a b*] [*c b a*]]

- La liste de trois chaînes [(*a,1*) (*b,1*) (*c,2*)] représente une des surjections de E sur F : *a* et *b* ont pour image *1* et *c* a pour image *2*.

Attention dans la suite de cette question les couples qui servent à définir le graphe de l'application sont en fait des chaînes de 5 caractères :
La parenthèse ouvrante, l'élément de E, la virgule, l'élément de F, la parenthèse fermante.

- L'ensemble des surjections de E sur F sera représenté par une liste de telles listes :[[(*a,1*) (*b,1*) (*c,2*)] [(*a,1*) (*b,2*) (*c,1*)] .....[(*a,1*) (*b,2*) (*c,2*)]].

1) Ecriture de quelques fonctions utiles pour la suite de cette partie

a) Ecrire la fonction `prem` qui prend comme argument un ensemble `E` et renvoie pour résultat le premier élément de celui-ci.
b) Ecrire la fonction `sp` (Sauf Premier) qui prend comme argument un ensemble E et renvoie pour résultat un nouvel ensemble qui est identique à l'ensemble E privé de son premier élément.
c) Ecrire la fonction `siem` (Sauf Ième) qui a deux arguments, un nombre entier `i` et un ensemble E. Cette fonction renvoie pour résultat un nouvel ensemble qui est identique à l'ensemble E privé de son ième élément.
d) Ecrire la fonction `ieme` qui a deux arguments, un nombre entier `i` et un ensemble E. Cette fonction renvoie pour résultat le ième élément de l'ensemble E.
e) Ecrire la fonction `ajout` qui a deux arguments, un élément `e` et une liste d'ensembles `lst_ens`. Cette fonction renvoie une nouvelle liste où l'élément `e` a été ajouté en tête de chacune des sous-listes qui composent `list_ens`.

Attention pour les questions qui suivent on veillera à apporter le plus grand soin dans le traitement des cas où E et/ou F sont des ensembles vides.

2) Ecrire la fonction récursive `parties` qui prend comme argument un ensemble E et qui renvoie pour résultat une liste d'ensembles représentant l'ensemble de toutes les parties de E.(conseil : si on particularise un élément `e` de E (le premier par exemple) l'ensemble des parties qui contiennent `e` et l'ensemble des parties qui ne le contiennent pas forment une partition de l'ensemble des parties de E).

3) Ecrire la fonction récursive `combinaisons` qui a deux arguments, un nombre entier `p` et un ensemble E. Cette fonction renvoie pour résultat une liste d'ensembles représentant l'ensemble de toutes les parties de E ayant `p` éléments. (conseil : si on particularise un élément `e` de E (le premier par exemple) l'ensemble des parties à `p` éléments de E-{`e`} et l'ensemble des parties à `p-1` éléments de E-{`e`} auxquelles on rajoute `e` forment une partition de l'ensemble des parties à `p` éléments de E).

4) Ecrire la fonction récursive `permutations` qui prend comme argument un ensemble E et qui renvoie pour résultat une liste d'ensembles représentant l'ensemble de toutes les permutations de E.(conseil : si on particularise un élément `e` de E (on peut le choisir de `n` façons si E possède `n` éléments) il suffit d'ajouter `e` en tête de chacune des listes représentant les permutations de E-{`e`} pour obtenir toutes les permutations de E).

5) Ecrire la fonction récursive `bijections` qui prend comme argument deux ensembles E et F. Cette fonction renvoie pour résultat une liste d'ensembles représentant l'ensemble de toutes les bijections de E dans F.(conseil : si on particularise un élément `e` de E, on peut choisir de `p` façons son image f (si F possède `p` éléments) il suffit d'ajouter le couple (`e`, `f`) (couple représenté par une chaîne de 5 caractères) en tête de chacune des listes représentant les bijections de E-{`e`} dans F-{`f`} pour obtenir toutes les bijections de E dans F).

6) Ecrire la fonction récursive `surjections` qui prend comme argument deux ensembles E et F. Cette fonction renvoie pour résultat une liste d'ensembles représentant l'ensemble de toutes les surjections de E sur F.(conseil : si on particularise un élément `e` de E, on peut choisir de `p` façons son image f (si F possède `p` éléments). Ainsi l'ensemble des surjections où l'image `f` de `e` n'est l'image d'aucun autre élément de E et l'ensemble des surjections où l'image `f` de `e` est l'image d'au moins un autre élément de E forment une partition de l'ensemble des surjections de E sur F).

7) Déduire de la question précédente l'écriture de la fonction récursive `nb_surjections` qui prend pour arguments deux nombres entiers `n` et `p`. Cette fonction renvoie pour résultat **le nombre** de surjections d'un ensemble de `n` éléments sur un ensemble de `p` éléments.

II- Programmation dynamique : chemin de poids minimum.

Soit M un tableau de n lignes et p colonnes de nombres entiers. On se donne la règle de déplacement suivante :
Si on se trouve dans la case de coordonnées (i , j), on peut se rendre soit dans la case de droite (i, j+1), soit dans la case du dessous (i+1 , j), à condition de ne pas « sortir » du tableau.
Un chemin est une suite de couples d'indices, coordonnées de cases du tableau, telle que deux couples consécutifs de la suite vérifient la règle précédente.
Le poids d'un chemin est égal à la somme des nombres contenus dans les cases parcourues.

On se donne comme point de départ la case D située en haut à gauche et comme case d'arrivée A celle située en bas à droite.

1) Indiquer le nombre de chemins possibles menant de D à A ?

Dans les explications qui suivent on suppose que les lignes et les colonnes du tableau M sont indexées à partir de 1.
On cherche maintenant à déterminer le chemin de poids minimum parmi tous les chemins qui mènent de D à A.
Enumérer tous les chemins possibles peut être très coûteux suivant les valeurs de `n` et `p`.
La programmation dynamique suggère de procéder différemment en s'appuyant sur le principe suivant :
Toute politique optimale est composée de sous-politiques optimales.
Il est évident que lors de la dernière étape le poids du chemin optimal augmentera de la valeur de l'entier M[n,p] qui se trouve dans la case d'arrivée.
A l'avant dernière étape le chemin optimal passe obligatoirement par l'une des deux cases (n,p-1) ou (n-1,p) contiguës à la case d'arrivée, le poids de ce chemin optimal augmentera alors de la valeur de la case atteinte plus l'entier qui se trouve en A. Notons c1=M[n,p-1]+M[n,p] l'augmentation dans le cas où le chemin optimal passe par la case (n,p-1) et c2=M[n-1,p]+M[n,p] l'augmentation dans le cas où le chemin optimal passe par la case (n-1,p).
A l'antépénultième étape, le chemin optimal passe forcément par une des trois cases de l'antépénultième diagonale : (n,p-2), (n-1,p-1), (n-2,p). Pour les cases de cette diagonale qui sont sur le bord du tableau, on n'a pas de choix, le chemin optimal augmente dans un cas de M[n,p-2]+c1, dans l'autre cas de M[n-2,p]+c2.
En fait lorsqu'on atteint le bord du tableau (dernière colonne ou dernière ligne) il n'y a plus de choix à faire, on ne peut que descendre ou qu'aller à droite.
Le cas de la troisième case (n-1,p-1) est plus intéressant, car si le chemin optimal passe par elle, il y a un choix à faire. On peut en effet soit descendre, soit aller à droite. C'est la plus petite des deux valeurs entre c1 et c2 qui indique le bon choix. Le poids du chemin optimal, s'il atteint cette case, augmentera de M[n-1,p-1] + min(c1,c2). Il est également nécessaire de mémoriser quelque part le choix qui a été fait.
On continue de procéder ainsi en remontant dans les étapes (les étapes possibles de même rang sont sur une même diagonale) jusqu'à remonter à la case de départ D.
Les structures de données nécessaires à la mise en place de cet algorithme sont deux matrices de nombres entiers *cout* et *choix* de `n` lignes et `p` colonnes :

> *cout*[i , j] indique le poids du chemin optimal qui mène de la case (i , j) à la case d'arrivée A.
> *choix*[i , j] indique le bon choix à faire si on se trouve dans la case (i , j) pour continuer le chemin optimal.
> On adopte la convention que 0 $(\downarrow)$ indique de descendre et 1 $(\rightarrow)$ d'aller à droite.

Exemple :

Si M=

1	4	6	8	2
2	**4**	**0**	2	4
3	5	**0**	8	9
8	0	**7**	**6**	**0**
0	9	5	9	**1**
2	7	5	5	**2**

alors on obtient *cout*=

23	24	22	26	18
22	20	16	18	16
24	21	16	17	12
24	16	16	9	3
21	26	17	12	3
21	19	12	7	2

et *choix*=

0	0	0	1	0
1	1	0	1	0
1	1	0	0	0
1	1	1	1	0
0	1	1	1	0
1	1	1	1	

La case en haut à gauche de la matrice *cout* contient le poids du chemin optimal et grâce à la matrice *choix* il est possible de reconstituer ce chemin optimal qui est matérialisé par les nombres écrit en gras dans le tableau M.

Dans les questions qui suivent, les matrices M, *cout* et *choix*, ainsi que `n` et `p` sont considérés comme des variables globales.

2) Ecrire la procédure `construire_tableaux` qui remplit les matrices `cout` et `choix`. Ces deux matrices se remplissent en parallèle une diagonale après l'autre, comme le suggèrent les explications précédentes.
Conseil : on peut cependant les remplir de manière identique beaucoup plus simplement que diagonale après diagonale.

3) Ecrire la procédure `affiche_chemin` qui affiche les couples (i , j) coordonnées des cases empruntées par le chemin optimal.

Tournez la page S.V.P.

4) Combien de tests sur les contenus de case sont-ils nécessaires pour trouver et afficher le chemin optimal par cette méthode ?

III- Plus longue sous-suite de caractères commune à deux suites de caractères.

Etant donnée une suite de caractères X= $x_1, x_2 \cdots, x_n$, une autre suite de caractères Y= $y_1, y_2 \cdots, y_p$ est une sous-suite de X s'il existe une sous-suite d'indices $1 \le i_1 < i_2 < \cdots < i_p \le n$ extraite de la suite $1,2,\cdots,n$ tels que pour tout $k = 1,2,\ldots, p$, on a $x_{i_k} = y_k$.

Par exemple Y=a,g,i,e,s est une sous-suite de X=**a**,l,**g**,o,r,**i**,t,h,m,**e**,**s**.

1) Ecrire les deux versions itérative et récursive de la fonction booléenne `est_sous_suite` qui prend comme arguments deux suites de caractères X et Y et qui renvoie la valeur vrai si et seulement si Y est une sous-suite de X.

Etant données deux suites de caractères X et Y, une suite de caractères Z est une sous-suite commune de X et Y si Z est une sous-suite à la fois de X et Y.

On s'intéresse maintenant à la détermination d'une **p**lus **l**ongue **s**ous-**s**uite **c**ommune (`plssc`) à deux suites X et Y données.

2) a) Proposer en quelques mots un algorithme trivial, qui utilise la fonction booléenne `est_sous_suite`, permettant de déterminer une plus longue sous-suite commune à deux suites X et Y.

b) Quel est l'ordre de grandeur du temps d'exécution d'un tel algorithme.

3) Si Z est une plus longue sous-suite commune à deux suites X et Y, on démontre que Z possède les propriétés suivantes :

- Si $x_1 = y_1$ alors la suite Z privée de son premier caractère est une plus longue sous-suite commune à X et Y privée toutes les deux de leur premier caractère.
- Si $x_1 \neq y_1$ et $x_1 \neq z_1$ alors Z est une plus longue sous-suite commune de X privée de son premier caractère et de Y.
- Si $y_1 \neq x_1$ et $y_1 \neq z_1$ alors Z est une plus longue sous-suite commune de X et de Y privée de son premier caractère.

a) Déduire de ces propriétés l'écriture de la fonction récursive `plssc` qui prend comme arguments deux suites X et Y et qui renvoie pour résultat une plus longue sous-suite commune à X et Y.

b) Quel est l'ordre de grandeur du temps d'exécution de cette fonction.

4) On peut, ici encore, mettre en œuvre un algorithme inspiré par la programmation dynamique pour améliorer le coût en temps. Cette façon de procéder nécessite l'utilisation de deux tableaux de nombres entiers `cout` et `choix` de `n` lignes et `p` colonnes

Notation : On note `cout[i,j]` la composante d'indices (i,j) du tableau `cout`.

Si X= $x_1, x_2 \cdots, x_n$ et Y= $y_1, y_2 \cdots, y_p$ désignent les deux suites de caractères dont on cherche une plus longue sous-suite commune alors cout[i,j] **a pour valeur la longueur d'une plus longue sous-suite** commune aux suites $x_i, x_{i+1} \cdots, x_n$ et $y_j, x_{j+1} \cdots, x_p$.

Ces deux tableaux sont remplis en parallèle ligne après ligne en démarrant **par la dernière**, chaque ligne étant remplie **de droite à gauche**.

L'algorithme de remplissage des tableaux, excepté pour la dernière ligne et la dernière colonne, est le suivant :

Si $x_i = y_j$ alors `cout[i,j]` prend comme valeur `1+cout[i+1,j+1]` et `choix[i,j]` vaut 0 (↘)

sinon `cout[i,j]` prend comme valeur `max(cout[i+1,j],cout[i,j+1])` et `choix[i,j]` vaut 1(↓) si `cout[i+1,j]` ≥ `cout[i,j+1]` sinon −1(→).

Une fois ces deux tableaux remplis la composante en haut à gauche du tableau `cout` contient la longueur d'une suite optimale et le tableau `choix` permet de reconstituer cette suite optimale.

Il suffit pour cela de démarrer à partir de la composante en haut à gauche du tableau `choix` et de se déplacer en suivant le sens de parcours :

0 indique de se déplacer en diagonale (↘).
1 de descendre (↓).
-1 d'aller à droite (→).

Chaque fois que `choix[i,j]` est égal à 0 ($x_i = y_j$) on ajoute le caractère x_i à la solution optimale. Le parcours s'arrête lorsqu'on sort du tableau.

Exemple :

`cout` =

	o	*c*	*c*	*u*	*r*	*r*	*e*	*n*	*t*
c	4	4	4	3	3	3	3	2	0
e	3	3	3	3	3	3	3	2	0
i	3	3	3	3	2	2	2	2	0
n	3	3	3	3	2	2	2	2	0
t	3	3	3	3	2	2	1	1	1
u	3	3	3	3	2	2	1	0	0
r	2	2	2	2	2	2	1	0	0
e	1	1	1	1	1	1	1	0	0
r	0	0	0	0	1	1	0	0	0

et `choix` =

	o	**c**	*c*	**u**	**r**	**r**	*e*	*n*	*t*
c	-1	**0**	0	1	1	1	1	1	1
e	1	1	**1**	1	-1	-1	0	1	1
i	1	1	**1**	1	1	1	1	1	1
n	1	1	**1**	1	1	1	-1	0	1
t	1	1	**1**	1	1	1	1	-1	0
u	-1	-1	-1	**0**	1	1	1	1	1
r	-1	-1	-1	-1	**0**	0	1	1	1
e	-1	-1	-1	-1	1	**1**	0	1	1
r	-1	-1	-1	-1	0	**0**	-1	-1	-1

Le parcours est visualisé par les composantes en caractères gras, une plus longue sous-suite commune aux deux mots est le mot **curr**.

5) Ecrire la procédure `construire_tableaux` qui prend pour arguments deux suites X et Y et qui remplit les deux matrices `cout` et `choix`. Ces deux matrices sont des **variables globales**.

6) Ecrire la fonction `plssc_1` qui prend pour argument deux suites X et Y et qui renvoie pour résultat une plus longue sous-suite commune à X et Y. Cette fonction utilise le tableau `choix`.

7) Quel est l'ordre de grandeur du temps de calcul ce dernier algorithme ?

Tournez la page S.V.P.

Rappels de quelques fonctions standard de PASCAL :

- La fonction `length` a pour résultat la longueur de la chaîne passée en paramètre.
- L'opérateur `infixé` de concaténation de chaînes est noté +.
- La fonction `copy(ch, i, lg)` renvoie pour résultat la sous-chaîne extraite de la chaîne `ch` qui commence au caractère de rang `i` et qui a pour longueur `lg`.
- Si `ch` est une variable chaîne, ch[i] désigne le ième caractère de la chaîne `ch`.

I) Enumérations d'ensembles.

Déclarations :

```
type
            elmt = string[5] ;
        ensemble = ^doublet ;
         doublet = record
                      el : elmt ;
                      suiv : ensemble
                   end ;

   liste_ensembles = ^nœud ;
             nœud = record
                        tete : ensemble ;
                        suiv : liste_ensembles
                    end ;
```

En-tête des fonctions demandées :

```
Function prem (E : ensemble) : elmt ;
Function sp (E : ensemble) : ensemble ;
Function siem (i :integer; E : ensemble) : ensemble ;
Function ieme(i :integer ; E : ensemble) : elmt ;
Function ajout(e :elmt ; lst_ens : liste_ensembles) :liste_ensembles ;
Function permutations(E : ensemble) : liste_ensembles ;
Function parties(E : ensemble) : liste_ensembles ;
Function combinaisons(p :integer ; E : ensemble) : liste_ensembles ;
Function bijections(E,F : ensemble) : liste_ensembles ;
Function surjections(E,F : ensemble) : liste_ensembles ;
Function nb_surjections(n,p : integer) : integer ;
```

Fonctions disponibles :

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

```
Function nb_elmts(E : ensemble) :integer ;
Function copie(E : ensemble) : ensemble ;
Function union(e1, e2 : ensemble) : ensemble ;
Function copie_lst_ens(l :liste_ensembles) :liste_ensembles ;
Function conc_lst_ens(l1,l2 :liste_ensembles) : liste_ensembles ;
```

- La fonction `nb_elmts` renvoie le nombre d'éléments de l'ensemble E passé en paramètre.

Toutes les fonctions suivantes laissent inchangés leurs paramètres, il y a toujours **création effective d'une nouvelle liste pour le résultat**.

- La fonction `copie` renvoie un nouvel ensemble, identique à celui passé en paramètre.
- La fonction `union` renvoie un nouvel ensemble, union des ensembles `e1` et `e2` passés en paramètre.
- La fonction `copie_lst_ens` renvoie une nouvelle liste d'ensembles identique à celle passée en paramètre.
- La fonction `conc_lst_ens` renvoie une nouvelle liste d'ensembles, concaténation des listes d'ensembles `l1` et `l2` passées en paramètre.

II) Programmation dynamique : chemin de poids minimum.

Déclarations :

```
Const
        nmax = 20 ;
Type
     tableau = array [1..nmax,1..nmax] of integer ;
Var
     M, cout, choix : tableau ;
               n, p : integer;
```

En-tête des procédures demandés :

```
Procedure construire_tableaux ;

Procedure affiche_chemin ;
```

III) Plus longue sous-suite commune à deux suites de caractères.

Déclarations :

```
Const
          nmax = 20 ;
Type
        suite = string[nmax];
      tableau = array [1..nmax,1..nmax] of integer ;
Var
      cout, choix : tableau ;
```

En-tête des fonctions et des procédures demandées :

```
Function est_sous_suite (X, Y : suite) : boolean ;

Function plssc  (X, Y : suite) : suite ;

Procedure construire_tableaux( X, Y :suite) ;

Function  plssc_1 (X, Y : suite) : suite ;
```

Tournez la page S.V.P.

ANNEXE CAML

Rappels de quelques fonctions de la bibliothèque CAML :

`char_for_read : char -> string` renvoie une chaîne représentant le caractère donné.
`string_length : string -> int` renvoie la longueur de la chaîne donnée.
`nth_char : string -> int -> char`, `nth_char s n` renvoie le caractère numéro `n` de la chaîne `s`.
`sub_string : string -> int -> int -> string`, `sub_string s d l` renvoie une nouvelle chaîne de longueur `l` contenant les caractères de la chaîne `s` à partir du caractère numéro `d`.
`préfix ^ : string -> string -> string`, `s1^s2` renvoie une nouvelle chaîne concaténée des chaînes `s1` et `s2`.
`préfix @ : 'a list -> 'a list -> 'a list`, `l1 @ l2` renvoie une nouvelle liste concaténée des listes `l1` et `l2`.
`list_length : 'a list ->int` renvoie le nombre d'éléments de la liste donnée.

Si m est une matrice, `m.(i).(j)` désigne la composante qui se trouve sur la ligne `i` et la colonne `j`.

I) Enumérations d'ensembles.

Liste des fonctions demandées :

```
prem : 'a list -> 'a = <fun>
sp : 'a list -> 'a list = <fun>
siem : int -> 'a list -> 'a list = <fun>
ieme : int -> 'a list -> 'a = <fun>
ajout : 'a -> 'a list list -> 'a list list = <fun>
permutations : 'a list -> 'a list list = <fun>
parties : 'a list -> 'a list list = <fun>
combinaisons : int -> 'a list -> 'a list list = <fun>
bijections : string list -> string list -> string list list = <fun>
surjections : string list -> string list -> string list list = <fun>
nb_surjections : int -> int -> int = <fun>
```

II) Programmation dynamique : chemin de poids minimum.

Valeurs globales :

```
let M = make_matrix 10 10 0 ;;
let cout = make_matrix 10 10 0 ;;
let choix = make_matrix 10 10 0 ;;
let n = vect_length M -1 and p = vect_length M.(0)-1;;
```

procédures demandées :

```
construire_tableaux : unit -> unit = <fun>
affiche_chemin : unit -> unit = <fun>
```

III) Plus longue sous-suite commune à deux suites de caractères.

Valeurs globales :

```
let cout = make_matrix 10 10 0 and choix = make_matrix 10 10 0 ;;
```

Liste des fonctions et procédures demandées :

```
est_sous_suite : string -> string -> bool = <fun>
plssc : string -> string -> string = <fun>
construire_tableaux : string -> string -> unit = <fun>
plssc_1 : string -> string -> string = <fun>
```