INFORMATIQUE, COMPOSITION N^O 2

Lundi 20 janvier 2014

▷ *Le candidat attachera la plus grande importance à la clarté, à la précision et à la concision de la rédaction.*

▷ *Si le candidat découvre en cours d'épreuve ce qu'il croît être une erreur d'énoncé, il le précisera dans sa copie.*

▷ *Le sujet comporte un problème sur les algorithmes de couverture minimale.*

1. RECOUVREMENT OPTIMAL

Ce sujet est une libre adaptation de l'épreuve d'informatique posée à l'École Polytechnique dans la filière PSI en 2012. Il a pour motivation principale le problème d'allocation de ressources, abordé explicitement dans la Partie 3. L'approche choisie consiste ici à considérer ce problème comme une instance de celui du recouvrement optimal d'ensemble.

Les deux premières parties sont indépendantes.

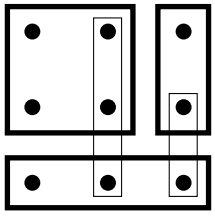
Les fonctions programmées dans les deux premières parties du problème seront utilisées dans la troisième et dernière partie.

La terminaison, la correction et la complexité d'une fonction ne seront démontrées que si l'énoncé le mentionne explicitement.

L'utilisation des fonctions puissance (c'est-à-dire $**$) et logarithme de Python est interdit. On rappelle que les opérateurs // et % donnent respectivement le quotient et le reste dans une division euclidienne d'entiers naturels.

PRÉAMBULE

Le problème s'énonce comme suit : étant donnés un ensemble fini U et une famille F constituée d'ensembles inclus dans U telle que F soit un *recouvrement* de U (c'est-à-dire que chaque élément de U appartient à au moins un ensemble de F), de combien d'ensembles de F a-t-on besoin, au minimum, pour recouvrir U ?



Dans l'exemple ci-contre, l'ensemble U est constitué de 9 points. La famille F contient 5 ensembles et recouvre U. Aucune sous-famille de F constituée d'au plus 2 ensembles n'est un recouvrement de U. En revanche, la sous-famille constituée des 3 ensembles en traits épais sur le dessin est un recouvrement de U. Une famille recouvrante de cardinal minimal est dite *optimale*.

PARTIE 1 – REPRÉSENTATION DES ENSEMBLES PAR DES TABLEAUX

Dans cette partie, on représente les ensembles finis d'entiers positifs ou nuls par des tableaux à valeurs dans $\{0,1\}$. On adopte la convention suivante : l'ensemble E correspondant au tableau t contient exactement les entiers i tels que $t[i]$ est égal à `1` ; E ne contient pas les entiers i tels que $t[i]$ est égal à `0`, ou tel que $t[i]$ n'est pas défini.

Ainsi, le tableau `[0,1,1,0,1]` représente l'ensemble $\{1,2,4\}$.

On remarque qu'un même ensemble S peut être représenté de multiples façons, en jouant sur la longueur du tableau. Par exemple

`[0,1,1,0,1,0]` et `[0,1,1,0,1,0,0,0,0]`

sont d'autres représentations de l'ensemble $\{1,2,4\}$.

a) Écrire une fonction `cardinal(t)`, qui prend en argument un tableau t représentant l'ensemble E, et qui renvoie le cardinal de l'ensemble E, c'est-à-dire le nombre d'élément(s) de E.

b) Écrire une fonction `inListe(i,t)`, qui prend en argument un entier naturel i quelconque et un tableau t représentant un ensemble E, et qui renvoie `1` si i appartient à E et `0` sinon.

c) Écrire une fonction `diff(t1,t2)`, qui prend en argument deux tableaux `t1` et `t2` représentant les ensembles E_1 et E_2, et qui renvoie un tableau représentant la différence ensembliste $E_1 \setminus E_2$. Ici $E_1 \setminus E_2$ désigne l'ensemble constitué des éléments de E_1 qui ne sont pas dans E_2.

PARTIE 2 – REPRÉSENTATION DES ENSEMBLES PAR DES ENTIERS

Afin de pouvoir manipuler simplement des familles d'ensembles, on souhaite maintenant représenter les ensembles par des entiers. Pour ce faire, on associe à un ensemble E d'entiers positifs ou nuls l'entier

$$s(\mathrm{E}) = \sum_{i \in \mathrm{E}} 2^i$$

Par exemple, l'ensemble $\{1,2,4\}$ correspond à l'entier $2^1+2^2+2^4$, soit 22.

On prendra garde à ne pas confondre les entiers représentant des ensembles avec les entiers contenus dans ces ensembles.

a) On considère un ensemble E représenté par un tableau `t`.
Que représente `t` pour l'entier naturel $s(\mathrm{E})$?

b) Si E_1 et E_2 sont deux sous-ensembles finis de $\mathbb{N}$ tels que $s(E_1) = s(E_2)$, peut-on en déduire que $E_1 = E_2$?

c) *Algorithme d'Hörner.*
On considère l'algorithme suivant :

Algorithme 1 : Algorithme de Hörner

```
Données : Un tableau t à valeurs réelles et un réel a
Résultat : t[0]a^0 + t[1]a + t[2]a^2 + ··· + t[n−1]a^(n−1)
Initialisation : n=len(t), s=0;
pour i∈ ⟦0,n-1⟧ faire
    s = t[n-1-i]+a*s
Renvoyer s
```

i) Prouver que la propriété suivante :

$$s = \sum_{k=0}^{i} t[n-1-k]\,a^{i-k}$$

est un invariant de boucle et en déduire la correction de l'algorithme.

ii) Quelle est la complexité de cet algorithme ?

iii) Écrire une fonction `setToInt(t)` qui prend en argument un tableau `t` représentant un ensemble E, qui renvoie l'entier s(E) représentant E par l'algorithme de Hörner.

d) Écrire une fonction `intToSet(e)` qui prend en argument un entier `e` représentant un ensemble E, et qui renvoie un tableau représentant E.

PARTIE 3 – RECOUVREMENTS

On aborde maintenant le problème de couverture optimale, exposé dans le préambule du problème.

a) *HX6 Ôssskkkiii !*
Une école de ski souhaite proposer des activités à ses élèves. Il y a plusieurs activités possibles, et chaque élève est invité à préciser lesquelles lui conviennent, en en choisissant au moins une. L'école souhaite minimiser le nombre d'activités à organiser, tout en garantissant que chaque élève puisse suivre une activité qui lui convient. Montrer que ce problème peut être vu comme un problème de recouvrement optimal.

On peut représenter une famille finie ordonnée $F = (F_0, F_1, ...)$ constituée d'ensembles finis d'entiers naturels par un tableau `f`, dont chaque case `f[i]` est l'entier qui représente l'ensemble F_i au sens de la partie 2.

Par exemple, le tableau `[17,3,8,22]` représente la famille $(\{0,4\},\{0,1\},\{3\},\{1,2,4\})$, puisque

$$2^0+2^4=17,\ \ 2^0+2^1=3,\ \ 2^3=8 \text{ et } 2^1+2^2+2^4=22$$

On fixe désormais $U = [\![0, n-1]\!]$ et une famille F d'ensembles inclus dans U telle que F soit un recouvrement de l'ensemble U. Dans tout le reste du sujet, on suppose que F est donnée par un tableau `f` d'entiers.

Une sous-famille G de F est constituée de certains des ensembles de la famille F, et est donc de la forme $(F_{\ell_0}, F_{\ell_1}, ...)$, où $\ell_0, \ell_1, ...$ est une sous-suite finie strictement croissante des indices de `f`. Par conséquent G est définie par l'ensemble des indices $\{\ell_0, \ell_1, ...\}$. Le cardinal de G est donc le cardinal de l'ensemble $\{\ell_0, \ell_1, ...\}$. Au sens de la partie 2, cet ensemble d'indices peut une nouvelle fois être représenté par un entier.

Par exemple, si F est la famille $(\{0,4\},\{0,1\},\{3\},\{1,2,4\})$, la sous-famille $G = (\{0,4\},\{3\})$, qui est constituée des ensembles F_0 et F_2, est de cardinal 2 et est représentée par l'entier 5, puisque l'on a $2^0+2^2=5$.

Noter que désormais trois objets distincts sont représentés par des entiers :

- Les éléments de U = $[\![0, n-1]\!]$: dans ce cas la lettre i sera de préférence utilisée, et il n'y a pas de codage particulier, i représente l'entier i ;
- Les sous-ensembles de U : dans ce cas la lettre e sera de préférence utilisée, et e représente l'ensemble E =`intToSet(e)` ;
- Les sous-familles de F : dans ce cas la lettre g sera de préférence utilisée, et g représente la sous-famille définie par les ensembles F_i pour $i \in$ `intToSet(g)`. De plus, chaque ensemble F_i est lui-même représenté par l'entier `f[i]`.

b) *Un algorithme glouton.*
Une première façon de trouver une sous-famille recouvrante de plus petit cardinal consiste à utiliser un algorithme dit *glouton*. L'idée est de construire une sous-famille étape par étape, en ajoutant à chaque fois l'ensemble qui permet de recouvrir le plus de nouveaux éléments (il peut y avoir plusieurs choix possibles).

i) Proposer un exemple de famille où l'algorithme glouton ne renvoie pas une sous-famille recouvrante optimale.

ii) Écrire une fonction `reste(e1,e2)`, qui prend en argument deux entiers `e1` et `e2` représentant les sous-ensembles E_1 et E_2 de U = $[\![0, n-1]\!]$, et qui renvoie le nombre d'éléments de E_1 qui ne sont pas dans E_2.

iii) Écrire une fonction `glouton(n,f)`, qui renvoie un entier g représentant une sous-famille recouvrante G de F grâce à l'algorithme glouton décrit ci-dessus.

c) *Un algorithme « force brute ».*
Résoudre un problème d'optimum par « *force brute* », c'est parcourir l'ensemble des configurations possibles afin de déterminer une solution optimale.

i) Écrire une fonction `estRecouvrante(n,g)`, qui prend en argument un entier g représentant une sous-famille G de F et qui renvoie `1` si G est recouvrante de U et `0` sinon.

ii) Écrire une fonction `forceBrute(n,f)` qui parcourt toutes les sous-familles possibles afin de renvoyer une sous-famille de F recouvrante optimale de U.

CORRIGÉ

PARTIE 1 – REPRÉSENTATION DES ENSEMBLES PAR DES TABLEAUX

a) On parcourt la liste `t` en incrémentant un compteur `c` (initialement nul) à chaque apparition de `1`.

```
def cardinal(t):
    c=0
    for a in t:
        if a:
            c=c+1
    return(c)
```

REMARQUE – *On peut aussi calculer la somme des $t[i]$ qui est égale à* #E.

b) On traite le cas où `i≥len(t)` au moyen d'un test :

```
def inListe(t,i):
    n=len(t)
    if i>=n:
        return(0)
    else:
        return(t[i])
```

c) On parcourt une copie de la liste représentant E_1 en la modifiant à chaque apparition d'un élément de E_2.

```
def diff(t1,t2):
    t,n=t1[:],len(t1)
    for i in range(n):
        if inListe(t1,i) and inListe(t2,i):
            t[i]=0
    return(t)
```

PARTIE 2 – REPRÉSENTATION DES ENSEMBLES PAR DES ENTIERS

a) Soient E une partie finie de $\mathbb{N}$ représentée par un tableau `t` de longueur n. Comme $t[k] \in \mathrm{E}$ *si et seulement si* $t[k] = 1$ et $t[k] \notin \mathrm{E}$ *si et seulement si* $t[k] = 0$, on a

$$s(\mathrm{E}) = \sum_{i \in \mathrm{E}} 2^i = \sum_{k=0}^{n-1} t[k]2^k$$

Ainsi `t` est la liste inversée des chiffres en base 2 de l'entier s(E), complétée par d'éventuels zéros en fin de liste.

b) Notons `t1` et `t2` des tableaux représentant des parties finies E_1 et E_2 de $\mathbb{N}$ telles que $s(E_1) = s(E_2)$. Comme l'écriture en base 2 est unique, on déduit de la question 2) a) que `t1=[a0,...,am,0,...,0]` et `t2=[a0,...,am,0,...,0]` (le nombre de zéros n'est pas nécessairement le même et peut-être nul) où `[am,...,a0]` est la liste des chiffres du nombre $s(E_1)$ en base 2. On en déduit que $E_1 = E_2$.

c) *Algorithme d'Hörner.*

i) Prouvons que la propriété suivante :

$$s = \sum_{k=0}^{i} t[n-1-k]a^{i-k}$$

est un invariant de boucle.

▷ Pour $i = 0$, on a bien $s = t[n-1]$ en fin d'itération.

▷ Supposons la propriété vérifiée à la fin d'une itération où $i \in [\![0, n-2]\!]$. À l'itération suivante, i devient $i' = i+1$ et la nouvelle valeur de la variable s est $s' = t[n-1-i'] + as = t[n-1-i-1] + as$, on a donc

$$\begin{aligned} s' &= t[n-1-1-i] + a\sum_{k=0}^{i} t[n-1-k]a^{i-k} \\ &= t[n-1-i-1] + \sum_{k=0}^{i} t[n-1-k]a^{i+1-k} = \sum_{k=0}^{i+1} t[n-1-k]a^{i+1-k} \end{aligned}$$

donc la propriété est vérifiée en fin de boucle.

▷ On en déduit que la propriété est un invariant de boucle.

Puisqu'en fin de boucle $i = n-1$, on a

$$s = \sum_{k=0}^{n-1} t[n-1-k]a^{n-1-k} = \sum_{k=0}^{n-1} t[k]a^{k}$$

ce qui prouve la correction de l'algorithme.

ii) Comme la liste `t` n'est parcourue qu'une seule fois, la complexité de cet algorithme est linéaire.

iii) On applique l'algorithme d'Hörner avec $a = 2$:

```
def setToInt(t):
    n,s=len(t),0
    for i in range(n):
        s=t[n-1-i]+2*s
    return(s)
```

d) D'après la question 2) a), la liste inversée des chiffres de s(E) en base 2 représente l'entier s(E). Il suffit d'adapter l'algorithme de codage en base 2 :

```
def intToSet(n):
    t=[n%2]
    while n>1:
        n=n//2
        t.append(n%2)
    return(t)
```

PARTIE 3 – RECOUVREMENTS

a) Notons U l'ensemble des élèves et numérotons les activités de 0 à $m-1$. Pour tout $k \in [\![0,m]\!]$, notons F_k l'ensemble des élèves souhaitant participer à l'activité n^o k. Comme chaque élève effectue au moins un choix d'activité, la famille $F = (F_0, \dots, F_{m-1})$ recouvre U. Minimiser le nombre d'activités à organiser, tout en garantissant que chaque élève puisse suivre une activité qui lui convient, revient donc à trouver une sous-famille de F recouvrante et optimale.

b) *Un algorithme glouton.*

i) Considérons $U = [\![0,5]\!]$, $F_0 = \{0,1,2\}$, $F_1 = \{1,2,3,4\}$ et $F_2 = \{3,4,5\}$. La famille (F_0, F_1, F_2) recouvre U. L'algorithme glouton renvoie (F_0, F_1, F_2) qui n'est pas optimale : la seule solution optimale est (F_0, F_2).

ii) Il s'agit de calculer le cardinal de $E_1 \setminus E_2$.

```
def reste(e1,e2):
    d=diff(intToSet(e1),intToSet(e2))
    return(cardinal(d))
```

iii) On crée une variable r représentant les termes restant à recouvrir et représentant initialement U. On utilise une boucle inconditionnelle (pour rechercher un des F_i recouvrant un maximum de termes de r) dans une boucle conditionnelle (pour s'arrêter dès qu'il y a recouvrement de U) :

```
def glouton(n,f):
    lf,r=len(f),setToInt([1]*n)
    G=[0]*lf
    while cardinal(intToSet(r))>0:
        min=n
        for i in range(lf):
            if reste(r,f[i])<min:
                min,indmin=reste(r,f[i]),i
        G[indmin]=1
        a=diff(intToSet(r),intToSet(f[indmin]))
        r=setToInt(a)
    return(setToInt(G))
```

c) *Un algorithme « force brute ».*

i) On « *retranche* » au fur et à mesure les éléments de G à U : on retourne `1` si on aboutit à l'ensemble vide (cas d'une famille recouvrante) et `0` dans le cas contraire.

```
def estRecouvrante(n,f,g):
    R,i,G=[1]*n,0,intToSet(g)
    m=len(G)
    while cardinal(R)>0 and i<m:
        if inListe(G,i):
            R=diff(R,intToSet(f[i]))
        if cardinal(R)==0:
            return(1)
        i=i+1
```

```
    return(0)
```

ii) On décrit toutes les sous-familles de F afin de déterminer une sous-famille recouvrante et de cardinal minimal.

```
def forceBrute(n,f):
    ng=1+setToInt([1]*len(f))
    mini=len(f)
    for i in range(ng):
        m=cardinal(intToSet(i))
        if estRecouvrante(n,f,i) and m<=mini:
            g,mini=i,m
    return(g)
```