ECOLE POLYTECHNIQUE

FILIERE PT

Épreuve orale d'Informatique

Sujet 0

Ce document a pour but de préciser les modalités de l'épreuve orale d'informatique de la filière PT pour le concours d'entrée à l'École Polytechnique.

Outre les objectifs et le format de l'épreuve, il contient également un exercice qui est dans l'esprit de ce qu'attendent les examinateurs de l'épreuve.

Objectifs et format de l'épreuve

Cette épreuve a pour but de tester les connaissances des candidats sur le programme officiel d'informatique (première et seconde années) pour la filière PT.

L'épreuve se déroule de la façon suivante. Le candidat dispose d'une heure de préparation sur un sujet donné par les examinateurs. Une fois cette heure écoulée le candidat présente pendant quarante-cinq minutes ses résultats devant les examinateurs qui pourront bien entendu lui demander des précisions et lui poser des questions ou des exercices complémentaires.

À aucun moment de l'épreuve nous n'utiliserons un ordinateur, il ne s'agit donc absolument pas d'une épreuve sur machine. Cependant, les examinateurs seront particulièrement attentifs à la capacité du candidat à écrire un programme syntaxiquement correct dans le langage `Python`. Ainsi, le candidat devra être capable pour chaque question demandant l'écriture d'un programme de proposer un code `Python` valide.

Exemple d'exercice

Dans ce qui suit nous donnons un exemple typique d'exercice pour cette épreuve.

Jeu de Nim

Complexité. La complexité, ou le temps d'exécution, d'un programme P (fonction ou procédure) est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, etc...) nécessaires à l'exécution de P. Lorsque cette complexité dépend de plusieurs paramètres n et m, on dira que P a une complexité en $O(\phi(n,m))$, lorsqu'il existe trois constantes A, n_0 et m_0 telles que la complexité de P soit inférieure ou égale à $A \times \phi(n,m)$, pour tout $n \geqslant n_0$ et $m \geqslant m_0$.

Lorsqu'il est demandé de donner une certaine complexité, le candidat devra justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

Implémentation. Dans ce sujet, nous adopterons la syntaxe du langage `Python`.

On rappelle qu'en `Python`, on dispose des opérations suivantes, qui ont toutes une complexité constante (car en `Python`, les listes sont en fait des tableaux de taille dynamique) :

- `[]` crée une liste vide (c.-à-d. ne contenant aucun élément)
- `[x]*n` qui crée une liste (ou un tableau) à `n` éléments contenant tous la valeur contenue dans `x`. Par exemple, `[1]*3` renvoie le tableau (ou la liste) `[1,1,1]` à 3 cases contenant toutes la même valeur `1`.
- `len(liste)` renvoie la longueur de la liste `liste`
- `liste[i]` désigne le `(i+1)`-ème élément de la liste `liste` s'il existe et produit une erreur sinon (noter que le premier élément de la liste est `liste[0]`).
- `liste.append(x)` ajoute le contenu de `x` à la fin de la liste `liste` qui s'allonge ainsi d'un élément. Par exemple, après l'exécution de la suite d'instructions "`liste = []; liste.append(2); liste.append([1,3]);`", la variable `liste` a pour valeur la liste `[2, [1, 3]]`. Si ensuite on fait l'instruction `liste[1].append([7,5]);`, la variable `liste` a pour valeur la liste `[2, [1, 3, [7,5]]]`.
- `liste.pop()` renvoie la valeur du dernier élément de la liste `liste` et l'élimine de la liste. Ainsi, après l'exécution de la suite d'instructions "`listeA = [1,[2,3]]; listeB = listeA.pop(); c = listeB.pop();`", les trois variables `listeA`, `listeB` et `c` ont pour valeurs respectives `[1]`, `[2]` et `3`.
- `random.randint(a,b)` renvoie un entier tiré (pseudo)aléatoirement et uniformément dans l'ensemble $\{\mathtt{a}, \mathtt{a}+1, \ldots, \mathtt{b}-1, \mathtt{b}\}$.

Important : L'usage de toute autre fonction sur les listes telle que `liste.insert(i,x)`, `liste.remove(x)`, `liste.index(x)`, ou encore `liste.sort(x)` est rigoureusement interdit (ces fonctions devront être programmées explicitement si nécessaire).

Le jeu de Nim se joue à deux joueurs. On considère m vases, le i-ème vase contenant x_i fruits. Les joueurs, Alice et Bob, jouent alternativement et Alice commence. Un tour de jeu consiste à retirer autant de fruits que l'on désire (mais au moins un) dans un **même** vase. Le joueur qui gagne est celui qui enlève le dernier fruit.

Question 1. On appelle *configuration* du jeu un m-uplet $(y_1, \ldots, y_m)$ décrivant le nombre de fruits dans chaque vase à un instant de la partie. Exprimer en fonction des nombre de fruits initialement dans chaque vase le nombre de configurations possibles du jeu. Commentez cet ordre de grandeur.

On considère la fonction suivante G appelée *fonction de Grundy* qui prend en argument les nombres de fruits dans chaque vase et vaut un entier $G(x_1, x_2, \ldots, x_m) = a$ défini de la façon suivante : si l'on note $a[i]$ le i-ème bit de la décomposition en base 2 de a, c'est-à-dire que $a = \sum_{i=0}^{n} 2^i a[i]$, alors $a[i]$ est défini par $a[i] = \left(\sum_{l=1}^{m} x_l[i] \right) \mod 2$, où $x_l[i]$ désigne le i-ème bit de la décomposition en base 2 de x_l.

Ainsi, pour calculer $G(6, 9, 1, 10)$, on écrit tout d'abord $6, 9, 1, 10$ en base 2 et on fait la somme par colonne modulo 2, ce qui donne la représentation en base 2 de a. On n'a alors plus qu'a convertir a en base 10 :

$$\begin{array}{rcccc} 6 & = & 0 & 1 & 1 & 0 \\ 9 & = & 1 & 0 & 0 & 1 \\ 1 & = & 0 & 0 & 0 & 1 \\ 10 & = & 1 & 0 & 1 & 0 \\ \hline a & = & 0 & 1 & 0 & 0 \end{array}$$

Ainsi on a $G(6, 9, 1, 10) = a = 4$.

Question 2. Calculer $G(10, 5, 2, 4)$.

Question 3. Ecrire une fonction `taille` qui prend en argument un entier x et qui renvoie le plus petit n tel que x puisse s'écrire en base 2 sur n bits. Votre programme ne devra pas utiliser la fonction `log` de Python. Vous préciserez la complexité.

Question 4. Ecrire une fonction `binaire` qui prend en argument un entier x et un entier n (tel que x puisse s'écrire en base 2 sur n bits) et qui renvoie une liste a de n bits tels que $a[i]$ le i-ème bit de la décomposition en base 2 de x sur n bits. Vous préciserez la complexité.

Question 5. Ecrire une fonction `decimal` qui prend en argument un suite a de bits et renvoie l'entier x dont a est la décomposition binaire. Vous préciserez la complexité.

Question 6. Ecrire une fonction `Grundy` qui prend en argument une liste x de m entiers $x_1, \ldots, x_m$ et renvoie la valeur de $G(x_1, \ldots, x_m)$. Vous préciserez la complexité.

On définit une *stratégie* pour Alice comme une fonction qui étant donnée un configuration $(x_1, \ldots, x_m)$ du jeu donne un indice $1 \leqslant i \leqslant m$ et un entier $1 \leqslant z \leqslant x_i$ de fruits à retirer d'un vase i. On définit les stratégies de Bob de la même façon. Une *stratégie* est respectée par un joueur au cours d'une partie si à chaque fois que ce joueur doit jouer il joue le coup indiqué par sa stratégie. On dit qu'un stratégie est gagnante pour Alice si elle assure la victoire à cette dernière dans toute partie où elle la respecte. Enfin, une configuration est gagnante pour Alice si cette dernière possède une stratégie gagnante pour toute partie débutant dans cette configuration.

Question 7. Ecrire une fonction `jouer_coup` qui prend en argument une liste x de m entiers $x_1, \ldots, x_m$, un indice $1 \leqslant i \leqslant m$ et un entier $1 \leqslant z \leqslant x_i$ et renvoie une liste y décrivant la configuration obtenue en retirant y éléments au vase d'indice i.

On va maintenant prouver le résultat suivant :

> *Soit la configuration de jeu où les vases contiennent respectivement x_1, x_2, …, x_m fruits. Alors le joueur dont c'est le tour a une stratégie gagnante si et seulement si $G(x_1, \ldots, x_m) \neq 0$.*

Comme ce résultat est une équivalence, on aura aussi que lorsque $G(x_1, \ldots, x_m) = 0$, le joueur qui doit jouer va perdre si son adversaire joue correctement. On partitionne en Y_0 et Y_1 l'ensemble des configurations de jeu, celles de Y_0 correspondant aux cas où la fonction de Grundy est nulle.

Question 8. Prouver que, pour toute configuration dans Y_0, et quel que soit le coup joué, la configuration suivante est dans Y_1.

Question 9. Réciproquement, prouver que, pour toute configuration dans Y_1, il existe un coup tel que la configuration suivante soit dans Y_0.

Question 10. Conclure et déduire de la preuve une stratégie gagnante associée.

Question 11. Ecrire une fonction `strategie` qui prend en argument une liste x de m entiers $x_1, \ldots, x_m$ et renvoie `None` si la configuration n'est pas gagnante pour le joueur dont c'est le tour, et sinon renvoie (i, z) où (i, z) est un coup qui permet d'arriver dans une configuration de Y_0. Vous préciserez la complexité.