Introduction à la Programmation (IP1)
Examen – **Durée : 3 heures**

Université Paris-Diderot – Lundi 12 décembre 2016

- Aucun document ni aucune machine ne sont autorisés. Les téléphones doivent être rangés.
- Les exercices sont tous indépendants.
- **Attention** : Indiquez au début de votre copie quel langage de programmation vous utiliserez dans le restant de votre copie. Il n'est pas possible de changer de langage une fois celui-ci choisi. Pour rappel, les filières MATHS et MIASHS doivent composer en PYTHON tandis que les filières MATHS-INFOS et INFOS doivent composer en JAVA.
- Par convention, dans les énoncés, le code JAVA se trouve à gauche et le code PYTHON à droite.
- Une réponse peut utiliser les réponses attendues à une question précédente (même si elle est non traitée).
- Les fragments de code doivent être correctement indentés.
- Dans les énoncés, on propose parfois une liste de fonctions et procédures utilisables. Vous pouvez cependant utiliser tout équivalent JAVA ou PYTHON. (Par exemple, `System.out.print` à la place de `printString`.) Attention cependant à ne pas utiliser des fonctions ou procédures de la bibliothèque standard qui n'auraient pas été abordées en cours.

Exercice 1 (Demi-vie)

Un nouvel atome découvert très prochainement s'appelle le Programmium et sa décomposition est régie par la règle suivante : si à un instant t, il y a N atomes de Programmium dans une solution alors à l'instant $t+1$ heure, il reste $N - N/9$ atomes dans la solution. Ici, $N/9$ représente la division entière de l'entier N par l'entier 9.

Dans cet exercice, on simule la dégradation de l'atome de Programmium et on calcule au bout de combien de temps le nombre d'atomes est divisé par au moins deux en fonction du nombre d'atomes initial.

Q1.1: *Écrire une fonction* `nextHour` *qui prend en paramètre un entier positif* `x` *représentant le nombre d'atomes de Programmium dans la solution et qui renvoie le nombre d'atomes de Programmium dans la solution une heure plus tard.*

Q1.2: *Écrire une fonction* `halfLife` *qui prend en paramètre un entier positif* `x` *supérieur ou égal à* `16` *représentant le nombre d'atomes de Programmium dans la solution et qui renvoie le nombre d'heures au bout desquelles le nombre d'atomes de Programmium est divisé par au moins deux. (Indice : utiliser une boucle* ***while****.)*

Q1.3: *Si* `halfLife` *est appelée avec l'entier* `15`*, est-ce que le programme termine ? Justifiez votre réponse.*

Q1.4: *Écrire une fonction* `enoughProgrammium` *qui prend en paramètre un entier positif* `x` *représentant le nombre d'atomes de Programmium dans la solution et un entier positif* `n` *qui représente un nombre d'heures. Cette fonction renvoie le booléen vrai si et seulement si au bout de* `n` *heures, le nombre d'atomes restant dans la solution est supérieur strictement à la moitié de* `x`*.*

□

Exercice 2 (Erreurs dans les statistiques)
L'été dernier, l'institut de sondage IPFRES a recruté un programmeur stagiaire pour lui faire écrire des fonctions d'analyse de données. Malheureusement, ce programmeur a commis de nombreuses erreurs dans ses fonctions. Dans cet exercice, on vous demande de mettre en évidence les erreurs commises par le programmeur et de proposer une version corrigée de chacune des fonctions qu'il a écrites. On rappelle que pour montrer qu'une fonction est incorrecte, il suffit de trouver une entrée pour laquelle sa sortie ne respecte pas le contrat demandé.

Q2.1: *La fonction* `lessThan` *prend en paramètre* `t`*, qui est un tableau d'entiers en* JAVA *et une liste d'entiers en* PYTHON*, ainsi qu'un entier* `n`*. Elle doit renvoyer le booléen vrai si tous les entiers de* `t` *sont inférieurs ou égaux à* `n`*. Pourquoi la fonction suivante est-elle incorrecte ? Proposez une correction.*

```java
public static
boolean lessThan (int[] t, int n) {
  for (int i = 0; i < t.length; i++) {
    if (t[i] < n) {
      return true;
    } else {
      return false;
    }
  }
  return true;
}
```

```python
def lessThan (t, n):
  for i in range (0, len (t), 1):
    if t[i] < n:
      return True
    else
      return False
  return True
```

Q2.2: *La fonction* `decrease` *prend en paramètre* `t`*, qui est un tableau d'entiers en* JAVA *et une liste d'entiers en* PYTHON*. Elle doit renvoyer le booléen faux si et seulement si* `t` *contient un entier x immédiatement suivi d'un entier y strictement plus grand que x. Pourquoi la fonction suivante est-elle incorrecte ? Proposez une correction.*

```java
public static
boolean decrease (int[] t) {
  for (int i = 0; i < t.length; i++) {
    if (t[i] >= t[i + 1]) {
      return true;
    } else {
      return false;
    }
  }
  return false;
}
```

```python
def decrease (t, n):
  for i in range (0, len (t), 1):
    if t[i] >= t[i + 1]:
      return True
    else
      return False
  return False
```

Q2.3: *La fonction* `isMedian` *prend en paramètre* `t`*, un tableau d'entiers de longueur* `n` *en* JAVA *et une liste d'entiers de longueur* `n` *en* PYTHON*, ainsi qu'un entier* `m`*. Elle doit renvoyer le booléen vrai si* `m` *est un entier apparaissant dans* `t` *et si le nombre d'éléments de* `t` *strictement inférieurs à* `m` *est* `n / 2`*.*
Pourquoi la fonction suivante est-elle incorrecte ? Proposez une correction.

```java
public static
boolean isMedian (int[] t, int m) {
  int nb = 0;
  for (int i = 0; i < t.length; i++) {
    if (t[i] < m) {
      nb = nb + 1;
    }
  }
  return (nb == t.length / 2);
}
```

```python
def isMedian (t, m):
  nb = 0
  for i in range (0, len (t), 1):
    if t[i] < m:
      nb = nb + 1
  return (nb == len (t) // 2)
```

□

Exercice 3 (La suite de k-bonacci)

On rappelle que la suite de Fibonacci est définie par :

$$\begin{cases} u_0 & = & 1 \\ u_1 & = & 1 \\ u_n & = & u_{n-1} + u_{n-2} \quad \textit{pour } n \geq 2 \end{cases}$$

Par extension, les k premiers termes de la suite de k-bonacci valent 1. Les termes suivants sont calculés en faisant la somme des k termes qui les précèdent :

$$\begin{cases} u_0 & = & 1 \\ \dots & & \\ u_{k-1} & = & 1 \\ u_n & = & u_{n-1} + u_{n-2} + \dots + u_{n-k} \quad \textit{pour } n \geq k \end{cases}$$

Q3.1: *Écrire une procédure* `sumPrevious` *qui prend en paramètre un entier* `k`*, un entier positif* `i` *strictement supérieur à* `k`*, et* `f`*, un tableau de longueur* `n > i` *en* JAVA *ou une liste de longueur* `n > i` *en* PYTHON*. Cette procédure modifie la case* `i` *de* `f` *en lui affectant la somme des* `k` *cases qui la précèdent.*

Q3.2: *Écrire une fonction* `initialize` *qui prend en paramètre un entier positif* `n` *et un entier positif* `k`*. Cette fonction renvoie* `f`*, un tableau de longueur* `n` *en* JAVA *ou une liste de longueur* `n` *en* PYTHON*. Les* `k` *premières cases de* `f` *valent* `1` *tandis que les autres cases valent* `0`*.*

Q3.3: *Écrire une fonction* `kbonacci` *qui prend en paramètre un entier positif* `n` *et un entier positif* `k` *et qui renvoie le terme u_n de la suite de k-bonacci.*

□

Exercice 4 (Roméo à la recherche de Juliette)

On suppose que Juliette est enfermée dans un château dont les pièces sont numérotées de 0 à $n-1$. La chambre de Juliette a pour numéro $n-1$ et Roméo arrive dans le château par la pièce numéro 0.

Chaque pièce est connectée à d'autres pièces par des portes. On suppose donné `doors` *qui est un tableau en* JAVA *et une liste en* PYTHON*. Pour toute pièce* `i`*,* `doors[i]` *est un tableau en* JAVA *et une liste en* PYTHON*. Son contenu représente les pièces accessibles depuis la pièce* `i`*.*

Par exemple, si le château a 4 pièces et si dans la pièce 0, on peut se rendre dans la pièce 1 et la pièce 2 ; dans la pièce 1, on peut se rendre dans la pièce 0 ; dans la pièce 2, on peut se rendre dans la pièce 0 et la pièce 3 et dans la pièce 3, on peut se rendre dans la pièce 2 ; alors `doors` *vaut* `{{ 1, 2 }, { 0 }, { 0, 3 }, { 2 }}` *en* JAVA *et* `[[ 1, 2 ], [ 0 ], [ 0, 3 ], [ 2 ]]` *en* PYTHON*. Roméo rejoint Juliette en passant de la pièce 0 à la pièce 2 et de la pièce 2 à la pièce 3.*

Q4.1: *Soit un château a 5 pièces et tel que dans la pièce 0, on peut se rendre dans la pièce 1 ; dans la pièce 1, on peut se rendre dans la pièce 0, la pièce 2 et la pièce 3 ; dans la pièce 2, on peut se rendre dans la pièce 1, la pièce 3 et la pièce 4 ; dans la pièce 3, on peut se rendre dans la pièce 1 et la pièce 2 ; dans la pièce 4, on peut se rendre dans la pièce 2. Donnez la valeur de* `doors` *qui représentent les portes des pièces de ce château. Peut-on se rendre de la pièce 0 à la pièce 4 en empruntant les pièces $1, 2$ puis 3 ? Même question en empruntant la pièce 1 puis la pièce 2.*

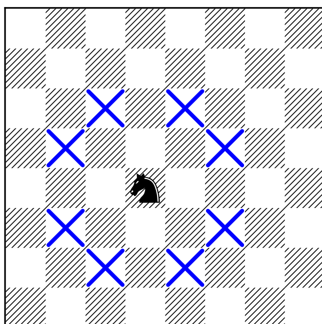
Q4.2: *Écrivez une fonction* `canMove` *qui prend en paramètre un entier positif* `n` *qui correspond au nombre de pièces du château,* `doors` *qui représente les portes des pièces du château comme décrit plus tôt, un entier* `p1` *qui représente un numéro de pièce et enfin un entier* `p2` *qui représente aussi un numéro de pièce. Cette fonction renvoie le booléen vrai si et seulement si une porte permet de se rendre de* `p1` *à* `p2`*.*

Q4.3: *Écrivez une procédure* `checkPath` *qui prend en paramètre un entier positif* `n` *qui correspond au nombre de pièces du château,* `doors` *qui représente les portes des pièces du château comme décrit plus tôt et enfin* `p`*, un tableau en* JAVA *et une liste en* PYTHON*, qui représente la suite des pièces à visiter par Roméo pour rejoindre Juliette. (*`p` *ne contient ni la pièce de départ, qui est toujours* `0`*, ni la pièce d'arrivée, qui est toujours* `n - 1`*.) Cette procédure affiche* `"Roméo a trouvé Juliette!"` *si les portes du château permettent effectivement à Roméo de parcourir la suite des pièces décrites par* `p` *depuis la pièce 0 jusqu'à la pièce $n-1$. Dans le cas contraire, cette procédure affiche* `"Pauvre Roméo!"`*.*

□

Exercice 5 (Échecs)
Dans cet exercice, vous allez écrire un programme qui calcule le nombre de coups nécessaires à un cavalier pour se rendre d'une case à une autre sur un échiquier vide. Un échiquier est une grille de 8 lignes et 8 colonnes. Pour simplifier votre travail, nous n'allons pas utiliser la numérotation standard : on numérotera les lignes de 0 *à* 7 *(*0 *étant la ligne la plus haute et* 7 *la plus basse) et les colonnes de* 0 *à* 7 *(*0 *étant la colonne la plus à gauche et* 7 *la plus à droite).*

Un cavalier se déplace en faisant deux pas dans une direction puis un pas en perpendiculaire à gauche ou à droite dans cette même direction. La figure suivante illustre toutes les positions accessibles par le cavalier depuis la case de coordonnées $(4, 3)$*, c'est-à-dire l'ensemble des positions* $\{(3, 1), (5, 1), (2, 2), (6, 2), (2, 4), (3, 5), (5, 5), (6, 4)\}$.



Voici une description générale en 6 étapes de l'algorithme que les questions Q5.1 à Q5.5 vont vous aider à réaliser. Cet algorithme calcule le nombre de coups nécessaires pour aller d'une case de coordonnées (x_0, y_0) *à une autre de coordonnées* (x_1, y_1)*. Voici les étapes d'initialisation de l'algorithme :*

1. *Créer une grille d'entiers à deux dimensions de 8 lignes et 8 colonnes.*
2. *Mettre la valeur* -1 *dans toutes les cases de cette grille.*
3. *Écrire* 0 *à la position initiale* (x_0, y_0) *du cavalier.*

Tant que l'on n'a pas atteint la position recherchée (x_1, y_1)*, on procède comme suit :*

Étape 1 *Calculer les positions accessibles en un coup depuis la position initiale et mettre la valeur* 1 *dans la grille à chacune de ces positions. Si la position finale* (x_1, y_1) *fait partie de l'ensemble de ces positions alors arrêter l'algorithme en renvoyant* 1.

Étape 2 *Calculer les positions accessibles en un coup depuis les positions où on trouve un* 1 *dans la grille et mettre la valeur* 2 *dans la grille à chacune de ces positions* ***sauf*** *si on y trouve déjà une valeur différente de* -1*. Si la position finale* (x_1, y_1) *fait partie de l'ensemble de ces positions alors arrêter l'algorithme en renvoyant* 2.

Étape i *Calculer les positions accessibles en un coup depuis les positions où on trouve un* $i - 1$ *dans la grille et mettre la valeur* i *dans la grille à chacune de ces positions* ***sauf*** *si on y trouve déjà une valeur différente de* -1*. Si la position finale* (x_1, y_1) *fait partie de l'ensemble de ces positions alors arrêter l'algorithme en renvoyant* i.

Remarquez qu'à chaque étape, une case de la grille doit avoir la valeur -1 *si on ne sait pas encore en combien de coups elle est accessible et la valeur* $i \geq 0$ *si la case est accessible en* i *coups à partir de la case de départ.*

On trouve dans la figure 1 (ci-dessous) les étapes de l'algorithme qui montre que le nombre de coups nécessaires pour aller de $(2,3)$ à $(6,5)$ est 2.

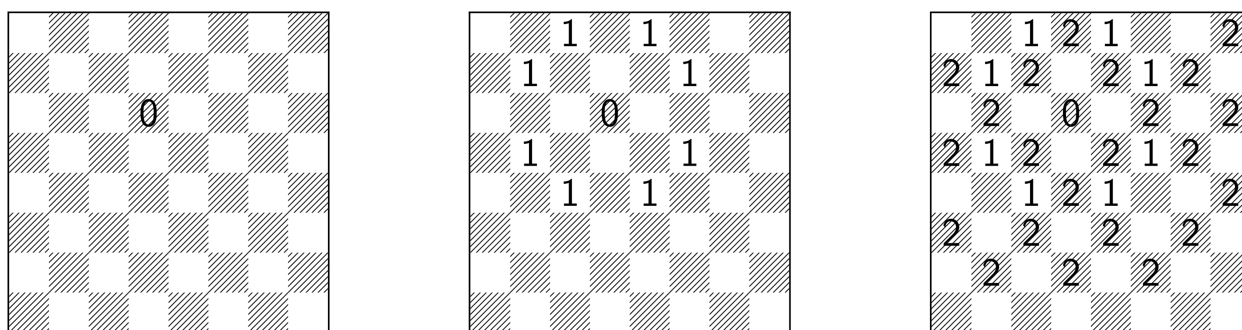


Figure 1 – Coups accessibles depuis la position $(2,3)$.

La première question permet de vérifier que vous avez compris l'algorithme et les suivantes vous guident dans sa programmation.

Q5.1: *Donnez les étapes de l'algorithme pour (x_0, y_0) valant $(4,1)$ et (x_1, y_1) valant $(2,2)$.*

Q5.2: *Écrire une fonction* `makeGrid` *qui ne prend aucun paramètre. En* Java, *elle renvoie un tableau d'entiers à deux dimensions formé de 8 lignes et 8 colonnes et dont les cases contiennent la valeur* `-1`. *En* Python, *elle renvoie une liste de 8 listes contenant chacune 8 entiers valant* `-1`.

Q5.3: *Écrire une fonction* `validPos` *qui prend en paramètre un entier* `r` *qui représente un numéro de ligne et un entier* `c` *qui représente un numéro de colonne. Cette fonction renvoie le booléen vrai si la position (r, c) est bien une position de la grille.*

Q5.4: *Écrire une fonction* `nextMoves` *qui prend en paramètre :*

- `grid` *qui représente la grille comme décrit dans la question 2 ;*
- *un entier* `k` *représentant le numéro du coup ;*
- *un entier* `r` *qui représente une ligne ;*
- *un entier* `c` *qui représente une colonne ;.*
- *un entier* `r1` *qui représente la ligne de la position finale P_1 ;*
- *un entier* `c1` *qui représente la colonne de la position finale P_1.*

Cette fonction doit mettre à jour `grid` *en mettant l'entier* `k` *dans toutes les cases qui sont accessibles par le cavalier depuis la position* `(r, c)` *et qui contiennent la valeur* `-1`.

Si l'une des cases accessibles considérées est `(r1, c1)` *alors la fonction renvoie le booléen vrai, sinon elle renvoie le booléen faux.*

Q5.5: *Écrire une fonction* `nbMove` *qui prend en paramètre un entier* `r0` *représentant la ligne de la position initiale P_0, un entier* `c0` *représentant la colonne de la position initiale P_0, un entier* `r1` *représentant la ligne de la position initiale P_1 et un entier* `c1` *représentant la colonne de la position initiale P_1. Elle renvoie le nombre de coups nécessaires à un cavalier pour se rendre de la case P_0 à la case P_1.*

Q5.6: *(Bonus) Écrire une fonction* `nbMoveReal` *qui a les mêmes paramètres que la fonction* `nbMove` *et un argument supplémentaire* `taken` *qui est une grille de booléens à deux dimensions contenant 8 lignes et 8 colonnes telle que* `taken[r][c]` *est vrai si la case de ligne* `r` *et de colonne* `c` *est déjà occupée par une pièce. Cette fonction doit renvoyer le nombre de coups nécessaires à un cavalier pour se rendre de la case P_0 à la case P_1 sans passer par une case déjà occupée, ou bien* `-1` *si une telle séquence de coups n'existe pas. Pourquoi votre programme termine-t-il ?*

□