be-OI 2011 **Finale** 30 mars 2011	**Remplissez ce cadre en MAJUSCULES et LISIBLEMENT, svp** PRÉNOM : ..... NOM : ..... ÉCOLE :	**Réservé**

Olympiades belges d'Informatique (durée : 1h40 maximum)

Ce document est le questionnaire de la finale des Olympiades belges d'Informatique 2011 pour **la catégorie supérieur**. Il comporte six questions qui doivent être résolues en **1h40 au maximum**. Chaque question est accompagnée d'un temps de résolution, donné à titre purement indicatif.

Notes générales (à lire attentivement avant de répondre aux questions)

1. N'indiquez votre nom, prénom et école **que sur la première page**. Sur toutes les autres pages, vous ne pouvez écrire que dans les **cadres prévus** pour votre réponse. Si, suite à une rature, vous êtes amené à écrire hors d'un cadre, répondez obligatoirement sur la même feuille, sans quoi votre réponse ne pourra être corrigée.

2. Vous ne pouvez avoir que de quoi écrire avec vous; les calculatrices, GSM... sont **interdits**.

3. Vos réponses doivent être écrites **au stylo ou au bic** bleu ou noir. Pas de réponses laissées au crayon. Si vous désirez des feuilles de brouillon, demandez-en auprès d'un surveillant.

4. Vous **devez** répondre aux questions ouvertes en **pseudo-code** ou dans l'un des **langages de programmation autorisés** (Java, C, C++, Pascal, Python et PHP). Les erreurs de syntaxe ne sont pas prises en compte pour l'évaluation. Vous ne pouvez utiliser aucune fonction prédéfinie à l'exception de celles présentées à la page suivante.

5. Vous **ne pouvez** à aucun moment **communiquer** avec qui que ce soit, excepté avec les surveillants ou les organisateurs. Toute question portant sur la compréhension de la question ou liée à des problèmes techniques ne peut être posée qu'aux organisateurs. Toute question logistique peut être posée aux surveillants.

6. Les participants **ne peuvent en aucun cas quitter leur place** pendant l'épreuve, par exemple pour aller aux toilettes ou pour fumer une cigarette. Selon le lieu où vous présentez l'épreuve, il peut vous être interdit de manger ou boire durant cette dernière.

7. Vous avez **exactement une heure et quarante minutes** pour répondre à ce questionnaire. Un **aide-mémoire** sur le pseudo-code et les conventions prises se trouve à la page suivante.

Bonne chance !

001

Questionnaire finale supérieur

Aide-mémoire du pseudo-code

Dans ce questionnaire, il y a des questions pour lesquelles vous devez compléter une partie de code. On vous demande de les compléter avec soit des *expressions*, soit des *instructions*. Les expressions ont une valeur, les instructions effectuent une action.

Les expressions possèdent une valeur et sont utilisées dans les calculs, pour modifier les valeurs des variable et dans les conditions. Certaines expressions ont comme valeur un nombre (entier ou réel) et d'autres un booléen (**`true`** ou **`false`**). Les instructions représentent des actions à exécuter. Un algorithme est une suite d'instructions.

Les expressions les plus basiques sont les entiers (comme 3, -12...) et les variables (comme x, sum...). On peut également construire des expressions avec des opérateurs comme le montre le tableau suivant.

opération	**notation**	**exemples**	
égalité	$=$	$3 = 2$ vaut **`false`**	$1 = 1$ vaut **`true`**
différence	$\neq$	$1 \neq 4$ vaut **`true`**	$3 \neq 3$ vaut **`false`**
comparaison	$>, \geq, <, \leq$	$1 > 3$ vaut **`false`**	$5 \leq 5$ vaut **`true`**
addition et soustraction	$+, -$	$3 + 2$ vaut 5	$1 - 9$ vaut -8
multiplication	$\times$	2×7 vaut 14	-2×3 vaut -6
division entière	$/$	$10/3$ vaut 3	$9/3$ vaut 3
reste de la division entière	$\%$	$10\%3$ vaut 1	$9\%3$ vaut 0
« non » logique	**`not`**	**`not`** $3 = 2$ vaut **`true`**	**`not`** $1 = 1$ vaut **`false`**
« et » logique	**`and`**	$3 = 3$ **`and`** $5 \leq 9$ vaut **`true`**	$1 \geq 3$ **`and`** $2 = 2$ vaut **`false`**
« ou » logique	**`or`**	$3 = 3$ **`or`** $12 \leq 9$ vaut **`true`**	$1 \geq 3$ **`or`** $2 \neq 2$ vaut **`false`**
accès à un élément de tableau	`[]`	soit tab valant `[1,2,3]`, alors tab`[1]` vaut 2	
appel à une fonction retournant une valeur		`size (`*list*`)`	`min (12, 5)`

Les fonctions que vous pouvez utiliser sont, en plus de celles définies dans l'énoncé et celles définies sur les tableaux ci-dessous, les fonctions `max (`a`, `b`)`, `min (`a`, `b`)` et `pow (`a`, `b`)` qui permettent respectivement de calculer le maximum entre deux nombres, le minimum entre deux nombres et la puissance (a^b).

Le tableau suivant reprend les instructions de base :

opération	**notation**	**exemple**
affectation	$\leftarrow$	$x \leftarrow 20 + 11 \times 2$
renvoi	**`return`**	**`return`** 42
appel à une fonction (qui ne renvoie rien)	nom de la fonction	`sort (`*list*`)`

En plus, il y a également des *instructions de contrôle* qui sont au nombre de trois : **`if`**–**`else`**, **`while`** et **`for`**. Ces instructions vont faire en sorte d'exécuter un groupe d'instructions ou non, en fonction de conditions. La notation **`for`** `(`$i \leftarrow a$ **`to`** b **`step`** k`) { [...] }` indique une boucle qui se répète tant que $i \leq b$ pour i initialisé à a et incrémenté de k à la fin de chaque itération.

Dans un algorithme, outre les variables permettant de stocker une valeur, on peut également utiliser des *tableaux* pour stocker plusieurs valeurs. Un tableau tab de taille n est indicé de 0 à $n - 1$. La notation tab`[`i`]` permet d'accéder au $(i + 1)^{\text{e}}$ élément du tableau. Ainsi, le premier élément du tableau est tab`[0]`. On peut créer un nouveau tableau tab de taille n, dont les éléments sont initialisés à 0, avec la notation $tab \leftarrow$ `newArray (`n`)`. On peut faire une copie d'un tableau tab de taille n vers un nouveau tableau $newTab$ avec la notation $newTab \leftarrow tab$.

À propos du cout des opérations

Les questions 2 et 3 vous demandent d'écrire des algorithmes efficaces afin d'obtenir le score maximum.

Lorsqu'on s'intéresse à la complexité temporelle d'un algorithme (une estimation du temps qu'il va prendre pour s'exécuter), une manière de procéder consiste à compter le nombre d'opérations élémentaires qu'il va effectuer. Toutes les instructions de base coutent une unité de temps. La création d'un tableau de taille n, et également la copie d'un tel tableau, coutent n unités de temps. Pour les instructions **`if`**–**`else`**, **`while`** et **`for`**, il faut compter une unité de temps pour le calcul de la condition. Puis, bien sûr compter les instructions qui seront exécutées.

Question 1 – La plus longue séquence strictement croissante (10 min)

L'algorithme donné ci-dessous a été conçu pour afficher, étant donné un tableau tab contenant n entiers, la longueur de la plus longue portion contigüe dont le contenu des cases forme une suite strictement croissante d'entiers. Par exemple, pour le tableau suivant :

1	4	2	5	7	2	2

,

on constate l'existence de 10 séquences strictement croissantes, à savoir, classées par longueur,

- `[]`
- `[1]`, `[4]`, `[2]`, `[5]`, `[7]`
- `[1,4]`, `[2,5]`, `[5,7]`
- `[2,5,7]`.

Pour cet exemple, l'algorithme va donc renvoyer la valeur 3 (la longueur de `[2,5,7]`). On vous demande de compléter l'algorithme ci-dessous afin qu'il calcule le bon résultat quelle que soit la taille du tableau tab.

Note : La fonction `max (a, b)` calcule le maximum entre a et b, c'est-à-dire la valeur la plus grande entre les deux.

```
Input  : • tab, un tableau d'entiers.
         • n, un entier positif (n ≥ 0), la taille du tableau tab.
Output : • la longueur de la plus longue séquence strictement croissante est renvoyée.
         • le tableau tab n'a pas été modifié.

length ← [...]                                                    // Q1(a)
m ← [...]                                                         // Q1(b)

for (i ← [...] to [...] step 1)                                   // Q1(c,d)
{
    if ([...])                                                    // Q1(e)
    {
        length ← 1
    }
    else
    {
        length ← length + 1
    }
    m ← max (m, length)
}
return m
```

Q1(a) **(une expression)**

. . . . . .

Q1(b) **(une expression)**

. . . . . .

Q1(c) **(une expression)**

. . . . . .

Q1(d) **(une expression)**

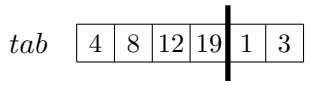
. . . . . .

Q1(e) **(une expression)**

. . . . . .

Question 2 – Quel est le plus grand élément ? (15 min)

Soit un tableau d'entiers positifs distincts (tous différents) dont les éléments sont triés par ordre croissant et qui a subi une rotation d'un certain nombre de cases. Une rotation de k cases signifie que tous les éléments sont décalés de k positions vers la droite du tableau, en faisant en sorte que les derniers éléments du tableau soient toujours remis au début de celui-ci. Voici un exemple d'un tel tableau, avec une ligne épaisse se situant à gauche du premier élément du tableau original qui était $[1, 3, 4, 8, 12, 19]$ et ayant subi une rotation de 4 cases :



On vous demande d'écrire un algorithme qui permet de retrouver le plus grand élément se trouvant dans le tableau. De plus, votre algorithme doit être **le plus efficace possible**.

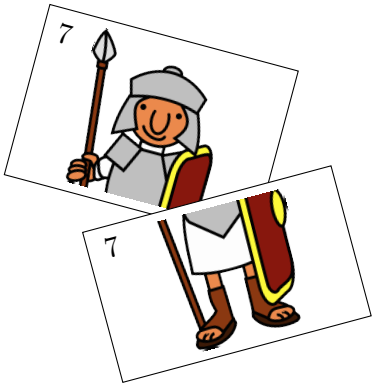
Input : • tab, un tableau d'entiers positifs distincts (tous différents) dont les éléments sont triés en ordre croissant et qui a subi une rotation d'un certain nombre de cases (voir énoncé).
• n, un entier strictement positif ($n > 0$), la taille du tableau tab.
Output : • le plus grand élément du tableau est renvoyé.
• le tableau tab n'a pas été modifié.

Seront pris en compte lors de la correction de cette question, outre le fait que l'algorithme calcule bien ce qui est demandé :

- le nombre d'opérations d'accès au tableau tab qui seront effectuées **dans le pire des cas** ;
- la simplicité de votre algorithme, qui devrait idéalement ne contenir qu'une seule boucle **`while`** et une seule instruction **`if`**–**`else`**, en plus d'instructions d'affectation de variables et d'une instruction **`return`**.

Q2

Question 3 – Collection d'autocollants (15 min)



Une toute nouvelle collection de 84 autocollants a été mise en place par PANINI© autour du thème d'*« Harry Potter et les Reliques de la Mort »*. Ces autocollants ont la particularité que l'image qu'ils contiennent ne représente pas un personnage complet, mais seulement une moitié de personnage. Quarante-deux autocollants représentent le haut des personnages et quarante-deux autres autocollants représentent le bas. Deux autocollants (haut et bas) qui correspondent au même personnage possèdent le même numéro.

Vous aimeriez savoir combien de personnages complets vous pouvez former avec votre collection d'autocollants. Pour ce faire, vous faites appel à votre frère Adrien pour vous aider. Vous prenez tous les autocollants correspondant à des parties hautes et votre frère prends ceux correspondant à des parties basses. Vous aimeriez maintenant faire le compte des personnages complets, le plus rapidement possible.

Écrivez un algorithme qui va compter le nombre de personnages complets que vous pouvez former. Cet algorithme reçoit deux tableaux d'entiers t_1 et t_2 représentant respectivement les numéros des cartes *« hautes »* et ceux des cartes *« basses »*. Par exemple, si t_1 vaut `[1,11,9,7,7,3,9,2]` et t_2 vaut `[7,7,2,8]`, alors l'algorithme doit renvoyer 3 puisqu'on peut reconstituer deux fois le personnage numéro 7 et une fois le numéro 2.

Input :
- t_1, un tableau d'entiers compris entre 0 et 41 inclus, représentant les numéros des cartes "hautes" de votre collection.
- t_2, un tableau d'entiers compris entre 0 et 41 inclus, représentant les numéros des cartes "basses" de votre collection.
- m, un entier positif ($m \geq 0$), la taille du tableau t_1.
- n, un entier positif ($n \geq 0$), la taille du tableau t_2.

Output :
- le nombre de personnages complets que vous pourrez former avec votre collection.
- les tableaux t_1 et t_2 n'ont pas été modifiés.

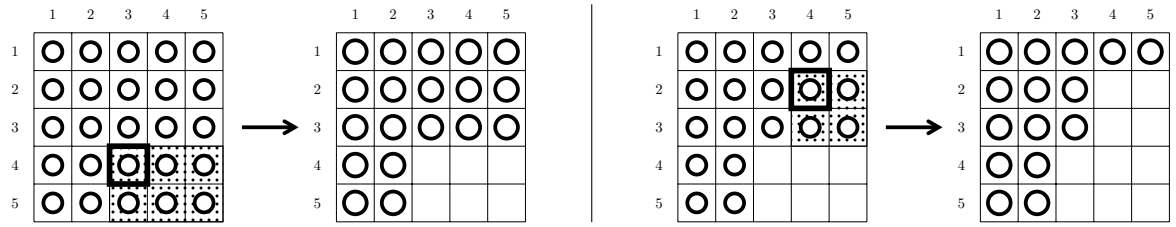
Votre algorithme doit être **le plus efficace possible**.

Q3

Question 4 – Le jeu de Chomp (20 min)

Le jeu de *« Chomp »* se joue sur un plateau formant une grille de taille $n \times n$ cases avec $n > 2$. Au tout début de la partie, un jeton est placé sur chacune des cases du plateau. Chacun à son tour, un joueur choisit un jeton et retire du jeu tous les jetons se trouvant sur le quart inférieur droit de celui-ci. Le joueur qui prend le jeton du coin supérieur gauche a perdu la partie.

Prenons par exemple un plateau de taille 5×5, dans lequel on calcule les indices à partir du coin supérieur gauche. Supposons que le premier joueur choisisse le jeton à l'indice $(3, 4)$. Il devra alors retirer tous les jetons dont les indices sont supérieurs ou égaux à 4 verticalement et 3 horizontalement (figure de gauche). Le second joueur peut alors, par exemple, jouer $(4, 2)$, ce qui amène le plateau dans la situation représentée ci-dessous (figure de droite).



On vous demande de trouver une stratégie de jeu qui vous permette de toujours gagner pour autant que ce soit vous qui commencez la partie. Cette stratégie ne doit pas dépendre de la manière avec laquelle l'autre joueur joue, ni de la taille du plateau. Complétez l'algorithme suivant que vous utiliserez à chaque fois que c'est votre tour de jouer.

Input : • n, un entier positif ($n > 2$), la taille du plateau de jeu.
• La grille actuelle est une configuration pour laquelle vous avez appliqué cette même stratégie depuis le début de la partie (grille remplie), chaque joueur respectant les règles du jeu.
• $last_i$ et $last_j$, deux entiers compris entre 0 et n, respectivement l'indice horizontal et l'indice vertical du dernier jeton retiré l'adversaire. Ces deux entiers valent 0 lorsqu'il s'agit du premier tour de la partie.

Output : • Un couple de valeur (i, j) (avec i et j entre 1 et n (inclus)) correspondant au jeton que vous choisissez et qui vous assure de gagner si vous jouez de manière adéquate est renvoyé.

```
if (last_i = 0 and last_j = 0)
{
    return ([...], [...])                                   // Q4(a)
}
else
{
    return ([...], [...])                                   // Q4(b)
}
```

Q4(a) **(2 expressions)**

. . . . . .

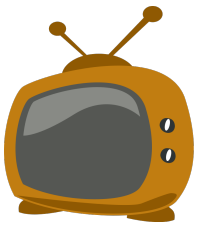
. . . . . .

Q4(b) **(2 expressions)**

. . . . . .

. . . . . .

Question 5 – Que vais-je regarder ce soir ? (20 min)



Je télécharge régulièrement et légalement sur mon disque dur les derniers épisodes de Dexter et d'autres séries. Dès qu'un épisode est téléchargeable, je le télécharge; dès que j'ai regardé un épisode, je l'enlève de mon disque dur. Bien sûr, il m'arrive d'accumuler plusieurs épisodes parce que je n'ai pas le temps de les regarder. J'ai donc une file d'attente d'épisodes à regarder et les épisodes y arrivent chronologiquement. Ainsi, lorsque j'ai le temps, je prends le premier épisode dans cette file :

Une structure de données appelée *file* permet de représenter une telle situation. Lorsque l'on dispose d'une file, on peut connaître sa taille (le nombre d'éléments qu'elle contient), on peut lui ajouter un élément (qui ira toujours à la fin de la file) et on peut retirer un élément (qui sera toujours le premier dans la file). On parle de notion *FIFO* (First-In First-Out) parce que le premier entré dans la file en sera le premier sorti.

Voici les trois opérations possibles :

- `add (f,e)` ajoute l'élément e à la fin de la file f
- `remove (f)` retire et renvoie l'élément se trouvant à la tête de la file f
- `size (f)` renvoie la taille de la file f

Voici un exemple qui construit une file avec trois éléments et ensuite la vide complètement :

```
fileDexter ← une nouvelle file vide
add (fileDexter, episode1)
add (fileDexter, episode2)
add (fileDexter, episode3)
while (size (fileDexter) ≠ 0)
{
    episode ← remove (fileDexter)
}
```

Note : Fixons-nous une convention de représentation de la file pour la suite de l'énoncé. Pour représenter le contenu de la file $fileDexter$, nous écrirons $fileDexter = (episode1, episode2, episode3)$ où $episode1$ est à la tête de la file.

Le problème qui me préoccupe pour le moment est que je voudrais venir à bout des séries que j'ai téléchargées et qui sont clôturées (plus d'épisode à venir). Chaque soir, j'hésite longtemps avant de me fixer sur le choix de la série que je vais regarder. Voici l'exemple de deux séries clôturées qui me préoccupent:

$F1 = (a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12})$ et $F2 = (b_{12}, b_{13}, b_{14})$

Il me reste 8 épisodes de $F1$ à regarder et 3 épisodes de $F2$. Vais-je regarder l'épisode a_3 ou l'épisode b_{12}? Afin de ne plus hésiter, je propose un algorithme qui choisira lui-même : **il s'agit d'intercaler les éléments de $F2$ dans $F1$ de telle sorte que l'avancement des épisodes soit équilibré et que les deux séries $F1$ et $F2$ se terminent plus ou moins en même temps.** Le nombre d'épisodes de la liste $F1$ qui seront regardés entre chaque épisodes de la liste $F2$ sera toujours le même, sauf à la fin où il pourrait y en avoir moins.

Après avoir intercalé la file $F2$ dans la file $F1$, $F1$ devient : $(a_3, a_4, a_5, \underline{b_{12}}, a_6, a_7, a_8, \underline{b_{13}}, a_9, a_{10}, a_{11}, \underline{b_{14}}, a_{12})$ et la file $F2$ est vide. Je regarderai à chaque fois 3 épisodes de $F1$ avant de regarder un épisode de $F2$, sauf à la fin où je n'en regarde qu'un seul. L'algorithme laissant bien les épisodes de chaque série apparaître dans l'ordre tel qu'ils sont rentrés dans leur file respective.

Avant de voir l'algorithme général, on va définir un algorithme qui permet de déplacer les n premiers éléments d'une file du début de la file vers la fin de celle-ci.

```
Input  : • F une file non-vide avec au moins n éléments.
         • n, entier positif.
Output : • Les n premier éléments de la file F ont été déplacés à la fin de la file
           tout en conservant l'ordre dans lequel ils étaient.

function move (F, n)
{
    [...]                                                          // Q5(a)
}
```

Q5(a) **(2 à 5 instructions)**

On vous demande maintenant de compléter l'algorithme général qui va intercaler les éléments de $F2$ dans $F1$. Pour cela, vous pouvez utiliser la fonction `move` qui vient d'être définie.

```
Input  : • F1 et F2, deux files non-vides telles que F2 est moins remplie que F1.
Output : • Les éléments de la file F2 ont été intercalés dans la file F1
           de sorte que celle-ci soit équilibrée. L'algorithme ne renvoie rien.

[...]                                                          // Q5(b)
while ([...])                                                  // Q5(c)
{
    [...]                                                      // Q5(d)
}
[...]                                                          // Q5(e)
```

Q5(b) **(une instruction)**

. . . . . .

Q5(c) **(une expression)**

. . . . . .

Q5(d) **(deux instructions)**

. . . . . .

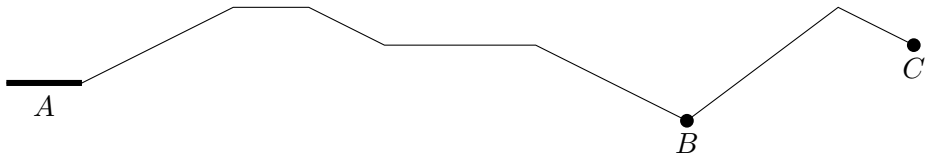
. . . . . .

Q5(e) **(une instruction)**

. . . . . .

Question 6 – Combien de plateaux minimums ? (20 min)

Vous allez partir en expédition avec votre cousin Simon, dans une région vallonée et vous vous posez une étrange question. Vous voulez savoir combien de *plateaux minimums* se trouveront sur votre parcours. La figure suivante montre l'évolution de l'altitude tout le long du trajet que vous comptez faire.



Sur ce trajet, on compte trois **plateaux minimums**. Un *plateau minimum* est un lieu (ponctuel ou un ensemble de points à altitude égale) pour lequel il est nécessaire de monter pour accéder à tout autre point du trajet (se situant avant ou après le plateau). Par exemple, pour le schéma ci-dessus, tous les points et lignes en gras sont des *plateaux minimums* (indiqués par les lettres A, B et C). En outre, un trajet totalement plat contient donc un unique plateau minimum.

Votre ami José vous propose un algorithme qui permet de compter le nombre de plateaux minimums. Pour ce faire, il vous propose de représenter les altitudes avec un tableau non-vide d'entiers. Par exemple, le tableau `[4]` contient un plateau minimum. Les tableaux `[3,8]`, `[2,2]` et `[12,5]` en contiennent chacun un. Le tableau `[3,1,8]` en contient également un. Enfin, le tableau `[1,1,1,5,5,9,9,9,8,6,12,17,17,4,4,1,-12]` en contient 3. On vous demande de compléter l'algorithme suivant pour qu'il calcule ce qui est demandé.

```
Input  : • tab, un tableau non-vide d'entiers.
         • n, un entier strictement positif (n > 0), la taille du tableau tab.
Output : • Le nombre de "plateaux minimums" est renvoyé.
         • Le tableau tab n'a pas été modifié.

nb ← 1
onPlateau ← true

for (i ← 1 to n − 1 step 1)
{
    if ([...])                                                    // Q6(a)
    {
        nb ← nb + 1
    }
    onPlateau ← [...]                                             // Q6(b)
}
return nb
```

Q6(a) **(une expression)**

. . . . . .

Q6(b) **(une expression)**

. . . . . .