OI 2010 **Demi-finale** 24 Mars 2010	**Remplissez ce cadre en MAJUSCULES et LISIBLEMENT, svp** PRÉNOM : ..... NOM : ..... ÉCOLE :	**Réservé**

Olympiades belges d'Informatique (durée : 3h maximum)

Ce document est le questionnaire de la demi-finale des Olympiades belges d'Informatique pour la catégorie supérieur. Il comporte six questions. Les deux premières sont des QCM et les quatre dernières sont des questions ouvertes. 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.

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**. Pas de réponses laissées au crayon. Si vous désirez des feuilles de brouillon, demandez-en auprès d'un surveillant.

4. Pour les questions de type QCM, vous ne devez choisir qu'**une seule réponse**. Une réponse correcte rapporte 1 point, une abstention vaut 0 point et une mauvaise réponse sera sanctionnée par $-0{,}5$ point.

5. Vous pouvez répondre aux questions ouvertes en pseudo-code, à l'aide d'organigramme ou avec un des langages de programmation autorisés : Java, C, C++, C#, Pascal, Python, Ruby, PHP et Visual Basic. Dans ce cas, vous ne pouvez utiliser **aucune librairie**, vous ne pouvez utiliser que les constructions standards du langage. Les erreurs de syntaxe ne sont pas prises en compte pour l'évaluation. En cas de doute, faites appel à un surveillant.

6. Vous pouvez définir des sous-fonctions et utiliser de la récursion, sauf mention explicite contraire.

7. Faites bien attention à l'indiçage des tableaux, les conventions peuvent changer d'une question à l'autre.

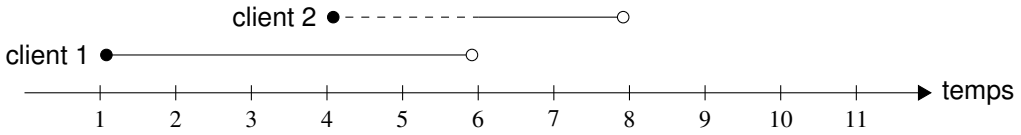
8. Vous avez **exactement trois heures** pour répondre à toutes les questions.

Bonne chance !

Question 1 – Le supermarché (10 min)

Vous êtes en charge d'étudier le comportement des clients d'un supermarché, au moment où ceux-ci passent à la caisse. En particulier, une des tâches qui vous incombe consiste à calculer les instants de départ des clients (après paiement des marchandises achetées), connaissant leurs instants d'arrivée et le temps qu'a duré leur service (c'est-à-dire le temps durant lequel ils étaient pris en charge par la caissière).

Soient n clients, numérotés de 1 à n, à une même caisse. La liste in contient les instants d'arrivée des clients, la liste $service$ contient le temps passé à la caisse et la liste out contient le résultat à calculer. Prenons un exemple avec deux clients. Le premier arrive au temps 1 et son service dure 5 et le second arrive au temps 4 et son service dure 2. Le premier client va partir en 6 $(1 + 5)$. Le second client va devoir attendre 2 que le premier client soit servi, puis encore 2 le temps d'être servi et il partira donc en 8.



Voici l'algorithme qu'on vous propose (un tableau de taille n est indicé de 0 à $n-1$) :

```
Input  : in, liste de longueur n+1 des instants d'arrivée des clients, in[0] = 0
         service, liste de longueur n+1 des durées de service des clients, service[0] = 0
         out, liste de longueur n+1 et out[j]=0 pour 0 ≤ j ≤ n+1
         De plus, n>0, in[j]>0 et service[j]>0 pour 1 ≤ j ≤ n et in est trié de manière
         croissante
Output : out contient les instants de départ des clients

for (i ← 1 to n step 1)
{
    if ([...])
    {
        out[i] ← out[i-1] + service[i]
    }
    else
    {
        out[i] ← in[i] + service[i]
    }
}
return out
```

Quelle est la condition manquante à l'algorithme pour que ce dernier calcule correctement les instants de départ ?

☐	$in[i] + service[i-1] > out[i-1]$
☐	$in[i-1] + service[i-1] > in[i+1]$
☐	$out[i-1] > in[i]$
☐	$out[i] > service[i-1] + in[i-1]$

Question 2 – Fonction mystère (10 min)

L'algorithme suivant permet d'effectuer un test sur une chaine de caractères s. Deux fonctions sont utilisées dans l'algorithme suivant, voici leurs descriptions :

`length (s)`	renvoie la longueur de la chaine s (le nombre de caractères)
`substr (s, i, n)`	renvoie la sous-chaine de s, commençant à l'indice i et de longueur n

On suppose que les chaines de caractères sont indicées à partir de 1 (le premier caractère est donc celui d'indice 1). Voici l'algorithme mystère :

```
Input  : s, une chaine de caractères non-vide
Output : ?

function mystery (s)
{
    if (length (s) < 2)
    {
        result ← true
    }
    else
    {
        if (s[1] = s[length (s)])
        {
            result ← mystery (substr (s, 2, length (s) − 2))
        }
        else
        {
            result ← false
        }
    }
}
```

Que renvoie la fonction `mystery` pour les entrées suivantes :

1. `"engage le jeu que je le gagne"`
2. `"un port trop nu"`
3. `"elu par cette crapule"`

☐	**true**, **false**, **true**
☐	**true**, **true**, **true**
☐	**false**, **true**, **false**
☐	**false**, **true**, **true**

Question 3 – Sudoku (20 min)

Pour un nouveau site web, qui va proposer aux utilisateurs de résoudre des sudokus en ligne, vous devez écrire un algorithme qui permet de vérifier si un sudoku est bien rempli. Pour un sudoku de taille n, il faut notamment vérifier que chaque ligne et colonne contienne les entiers de 1 à n, chacun apparaissant exactement une fois. Vous trouvez deux algorithmes qui prennent en entrée un tableau d'entiers $line$, indicé de 1 à n, avec $n > 0$ et qui renvoient un booléen : **`true`** si $line$ contient une permutation des entiers de 1 à n et **`false`** sinon. Par exemple, `[2,4,1,3,5]` est une permutation des entiers de 1 à 5 et les algorithmes renverront **`true`**. Par contre, pour `[1,4,2]`, ils renverront **`false`**.

Algorithme 1

```
temp ← tableau d'entiers de taille n, indicé de 1 à n, tel que temp[j] = 0, ∀j : 1 ≤ j ≤ n
b ← true
i ← 1

while (i ≤ n and b)
{
    b ← (1 ≤ tab[i] and tab[i] ≤ n)
    if (b)
    {
        temp[tab[i]] ← temp[tab[i]] + 1
        b ← (temp[tab[i]] = 1)
    }
    i ← i + 1
}
return b
```

Algorithme 2

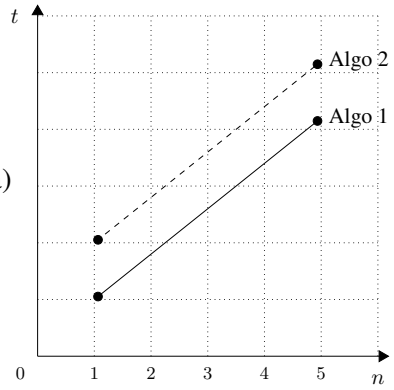
```
b ← true
i ← 1

while (i ≤ n and b)
{
    b ← (1 ≤ tab[i] and tab[i] ≤ n)
    j ← i + 1
    while (j ≤ n and b)
    {
        b ← not (tab[i] = tab[j])
        j ← j + 1
    }
    i ← i + 1
}
return b
```

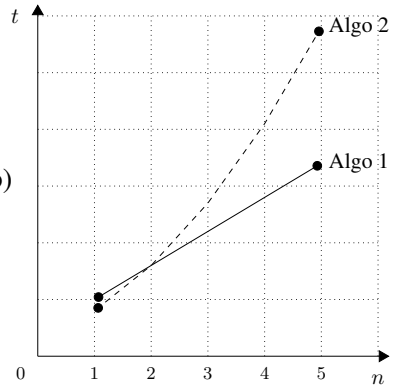
On vous demande d'analyser la complexité temporelle des deux algorithmes, afin de les comparer et d'identifier celui qui sera le plus rapide à s'exécuter. En effet, votre site web doit être le plus réactif possible et prendre le moins de temps possible à s'exécuter, surtout pour des sudokus de tailles plus grandes que 9×9.

Pour compter le temps d'exécution, on admet les hypothèses simplificatrices suivantes : exécuter une affectation compte pour 1 et évaluer la condition d'un **`if`** ou d'un **`while`** compte également pour 1. Les deux algorithmes ont été exécutés avec un tableau de taille 1, 2, 3, 4 et 5 et le temps d'exécution a été à chaque fois calculé. Parmi les quatre graphes suivants, lequel est celui qu'on a obtenu avec nos tests ?

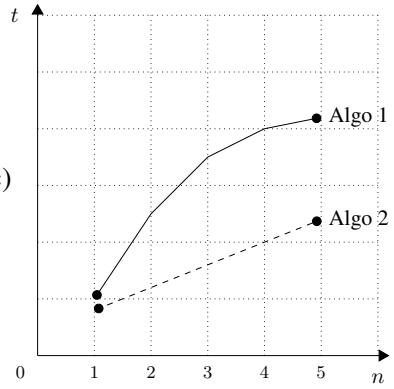
☐ (a)



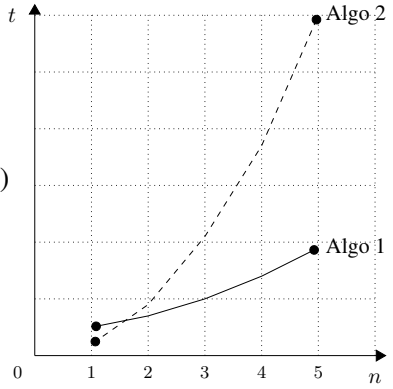
☐ (b)



☐ (c)



☐ (d)



En observant les deux courbes, quel est l'algorithme qui a la meilleure efficacité temporelle, lorsque la valeur de n augmente. Quel sacrifice a dû être fait pour que cet algorithme soit plus rapide que l'autre ?

Q3

Question 4 – C'est trop compliqué ! (15 min)

Votre frère a suivi un stage d'informatique durant l'été de l'année passée. Il était tout fier d'avoir appris plein de chouettes choses et d'avoir réussi à écrire son premier algorithme tout seul. Ce dernier prend en entrée deux nombres naturels `a` et `b` et produit comme résultat un nombre naturel. Voici son algorithme, où les opérateurs `/` et `%` calculent respectivement le quotient et le reste de la division entière.

```
Input  : a et b, deux nombres naturels
Output : ?

c ← 0
p ← 1
while (a ≠ 0 and b ≠ 0)
{
    c ← c + p * ((a % 10) + (b % 10))
    p ← p * 10
    a ← a / 10
    b ← b / 10
}
return c
```

Commencez par comprendre ce que fait cet algorithme. Proposez ensuite une version simplifiée de celui-ci.

Q4

Question 5 – Combien de mots y a-t-il ? (50 min)

On considère une chaine de caractères composée de mots séparés (précédés ou suivis) par un ou plusieurs séparateurs. Les mots sont constitués des caractères `a-z` et `A-Z` et les séparateurs peuvent être un espace ( ), une virgule (`,`) ou un point (`.`).

Considérons que cette chaine de caractères se trouve dans un tableau de caractères `s`. On vous demande deux choses :

1. Soit x un nombre entier positif. Écrivez une fonction `countWords (s, x)` qui compte le nombre de mots de longueur x se trouvant dans la chaine de caractères `s` ;
2. Écrivez une fonction `countWordsWithChar (s, c)` qui compte, dans la chaine de caractères `s`, le nombre de mots qui contiennent le caractère `c` donné.

Voyons quelques exemples :

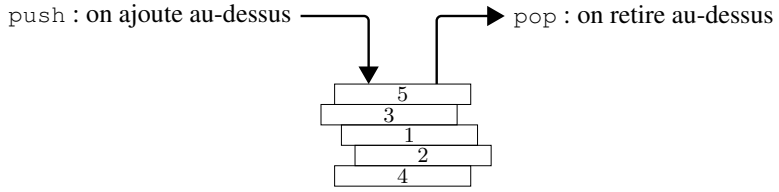
– La phrase `"Toto a vu lulu ... Mais Lulu, lui ne voit pas Toto."` contient 6 mots de taille 4. Le caractère `'o'` apparait dans 3 mots ; le caractère `'u'` apparait dans 4 mots.
– La phrase `"Vive les olympiades en informatique"` contient 1 mot de taille 4. Le caractère `'i'` apparait dans 3 mots.

Q5

Q5 (suite)

Question 6 – Faut ranger tes BDs ! (50 min)

Votre frère est un grand fan de la série de BD *« Computer Man »* et il possède tous les numéros de cette dernière. Le seul souci est que lorsqu'il les lit, il les range n'importe comment et n'importe où. Un jour, il a décidé de faire un peu de rangement et il a empilé tous ses numéros les uns sur les autres.



Une structure de données appelée *pile* permet de représenter une telle situation. Lorsqu'on dispose d'une pile, on peut connaitre sa taille (le nombre d'éléments qu'elle contient), on peut lui ajouter un élément (qui ira tout au-dessus) et on peut retirer un élément (qui sera toujours celui situé tout au-dessus). Les quatre opérations possibles sont donc :

`new`	crée une nouvelle pile vide
`push (p, i)`	ajoute l'élément i au-dessus de la pile p
`pop (p)`	retire et renvoie l'élément se trouvant au-dessus de la pile p
`size (p)`	renvoie la taille de la pile

Voici par exemple comment on peut créer la pile illustrée plus haut, et ensuite la vider complètement :

```
BDs ← new
push (BDs, 4)
push (BDs, 2)
push (BDs, 1)
push (BDs, 3)
push (BDs, 5)
while (size (BDs) ≠ 0)
{
    bd ← pop (BDs)
}
```

Donnez un algorithme utilisant uniquement des piles et des entiers à l'exclusion de toute autre structure (liste, tableau, dictionnaire, etc.) qui prend en entrée une pile d'entiers tous différents. Après exécution de l'algorithme, la pile reçue en entrée est modifiée de manière à contenir les mêmes éléments qu'initialement, mais dans l'ordre. Ainsi, si la pile de départ était non vide, faire `pop` dessus après exécution de l'algorithme renvoie l'élément minimum de la pile. De plus, pouvez-vous y arriver en utilisant l'opération `pop` au plus $n * (n + 1)$ fois pour une pile en entrée de taille n ? (bonus)

Q6

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Q6 (suite)