Interrogation d'ITC n°3

Semaine du 11/13/2024

Durée : 35 minutes

Nom :		Prénom :	

Consignes

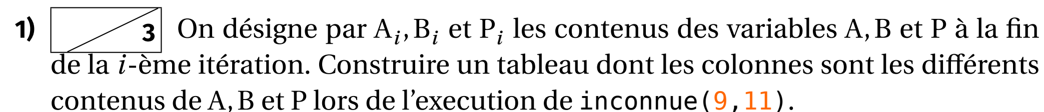
- Les codes doivent être présentés en mentionnant explicitement l'indentation, au moyen par exemple de barres verticales sur la gauche.
- Pour qu'un code soit compréhensible, il convient de choisir des noms de variables les plus explicites possibles.
- La performance du code proposé à chaque question entrera dans l'évaluation, de-même que l'utilisation de boucles appropriées.
- Vous pouvez bien sûr utiliser une feuille de brouillon en parallèle.

/20

Présentation (écriture, propreté, marqueurs d'indentation du code, ...) : /1

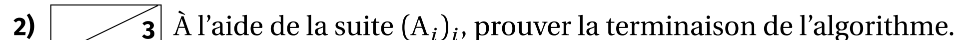
Exercice 1 **Méthode soviétique pour...** On considère la fonction suivante où a et b sont supposés être des éléments de $\mathbb{N}$.

```python
def inconnue(a:int,b:int)->int:
    A, B, P = a, b, 0
    while A != 0:
        if A%2 != 0:
            P = P+B
        A = A//2
        B = 2*B
    return P
```



1) /3 On désigne par A_i, B_i et P_i les contenus des variables A, B et P à la fin de la i-ème itération. Construire un tableau dont les colonnes sont les différents contenus de A, B et P lors de l'execution de `inconnue(9,11)`.

/1 Pouvez-vous conjecturer le résultat renvoyé par `inconnue(a, b)` ?



2) /3 À l'aide de la suite $(A_i)_i$, prouver la terminaison de l'algorithme.

Exercice 2 **Permutations d'une liste** Pour $n \in \mathbb{N}$, on définit $A_n = $ [1, 2,.., n] si $n \geqslant 1$, et A_0 désignant la liste vide par convention.

On dit qu'une liste L est une permutation de A_n lorsque cette liste contient tous les éléments de A_n une et une seule fois. Par exemple [1, 3, 2, 4] et [1, 2, 3, 4] sont des permutations de $A_4 = $ [1, 2, 3, 4], mais [1, 1, 2, 4] n'en est pas une. Les fonctions ne doivent s'appliquer que sur des listes ayant n éléments de A_n. On veillera à mettre en place ce test via des commandes `assert`.

1) On considère la fonction suivante :

```
1  def estPer1(L, n):
2      assert len(L) == ___________
3      for k in L:
4          assert(______________)
5      perm = True
6      for i in range(n-1):
7          for j in range(i+1, n):
8              if L[i] == L[j]:
9                  perm = False
10     return perm
```

1.1) [1] Recopier et compléter les lignes 1 à 4, en ajoutant notamment la signature de cette fonction.

1.2) [2] On désigne par $C(n)$ le nombre d'affectations et de comparaisons lors de l'exécution de la fonction précédente. Justifier que $C(n) = O(n^a)$ avec a à préciser.

2) On considère une seconde fonction :

```
def estPer2(L, n):
    assert len(L) == ___________
    for k in L:
        assert(______________)
    present = [False]*n
    for k in L:
        present[_______] = True
    l = 0
    while ________ and present[l]:
        l += 1
    return l == n
```

2.1) [3] Compléter ci-dessus cette fonction afin qu'elle retourne **True** si L est une permutation, **False** sinon.

2.2) [4] On pose $\mathscr{P}_\ell$ « L contient tous les éléments de A_ℓ ». Montrer que $\mathscr{P}_\ell$ est un invariant de la boucle `while`, en déduire la correction du programme `estPer2`.

2.3) [/2] Justifier, à l'aide d'un calcul de complexité, que `estPer2` est plus performante que `estPer1`.

Correction
de l'Interrogation d'ITC N°3

Solution 1

1) Utilisons Python pour cela.

```
>>> inconnue(9,11)
Ai Bi Ci
__________
9 11 0
4 22 11
__________
2 44 11
__________
1 88 11
__________
0 176 99
__________
99
```

On conjecture que `inconnue(a, b)` renvoie : $\boxed{a \times b}$. *La preuve a été faite en cours, à l'aide d'un invariant.*

2) Montrons la terminaison. Notons A_i la valeur de la variable `A` à la fin de l'itération i. S'il y a une itération $i+1$, alors $A_i \neq 0$. Peu importe la parité de A_i, on a à l'itération $i+1$: $A_{i+1} = A_i//2$.

- **[1 er cas]** Si A_i est pair, alors $A_i = 2k$ avec $k \in \mathbb{N}^\star$ car $A_i \neq 0$. Donc $A_{i+1} = k < 2k = A_i$.
- **[2ème cas]** Si A_i est impair, alors $A_i = 2k+1$ avec $k \in \mathbb{N}$. Donc $A_{i+1} = k < 2k+1 = A_i$.

Ainsi, $(A_i)_i$ est une suite strictement décroissante d'entiers.

Si par l'absurde la boucle **`while`** ne s'arrête pas, alors on a $A_i > 0$ pour tout $i \in \mathbb{N}$. Or, par stricte décroissance, il existe en entier i_0 de sorte que $A_{i_0} \leqslant 0$ — **Contradiction**.

Solution 2

1) **1.1)**

```python
def estPer1(L, n):
    assert len(L) == n
    for k in L:
        assert(k >= 1 and k <= n)
    perm = True
    for i in range(n-1):
        for j in range(i+1, n):
            if L[i] == L[j]:
                perm = False
    return perm
```

```
>>> estPer1([1, 3, 2, 5, 4], 5)
True
>>> estPer1([1, 3, 2, 5, 6], 5)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 4, in estPer1
AssertionError
>>> estPer1([1, 2, 2, 5, 4], 5)
False
```

1.2) Ici n désigne la longueur de la liste. On a dans le pire des cas (tous les éléments sont identiques) :

- 1 test, puis $2n$ tests dans les `assert`,
- ensuite 1 affectation,
- puis dans la boucle **`for`** une complexité de :

$$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \underbrace{2}_{1 \text{ test} + 1 \text{ aff.}} = \sum_{i=0}^{n-2} 2(n-1-i) = n(n-1).$$

Au total : $C_{\text{pire}}(n) = 1 + 2n + 1 + n(n-1) = \boxed{O(n^2)}$. Dans le meilleur des cas (cas d'une permutation), on enlève une affectation dans la somme, ce qui donne :

$$C_{\text{meilleur}}(n) = 1 + 2n + 1 + \frac{n(n-1)}{2} = \boxed{O(n^2)}.$$

Donc : $C(n) = \boxed{O(n^2)}$.

2) 2.1)

```python
def estPer2(L:list, n:int)->bool:
    assert len(L) == n
    for k in L:
        assert(k >= 1 and k <= n)
    present = [False]*n
    for k in L:
        present[k-1] = True
    l = 0
    while l < n and present[l]:
        # l'entier l+1 est présent dans L
        l += 1
    return l == n
```

```
>>> estPer2([1, 3, 2, 5, 4], 5)
True
>>> estPer2([1, 3, 2, 5, 6], 5)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 4, in estPer2
AssertionError
>>> estPer2([1, 2, 2, 5, 4], 5)
False
```

2.2) On pose $\mathscr{P}_\ell$ « `L` contient tous les éléments de A_ℓ ». Montrons que $\mathscr{P}_\ell$ est un invariant de boucle `while`.

Initialisation. À l'itération $\ell = 0$: on a `L` qui contient tous les éléments de A_0 car elle est vide par convention.

Hérédité. Soit ℓ tel qu'il y ait une itération $\ell + 1$, c'est-à-dire $\ell < n$ et `present[l]` qui vaut `True`, et tel que `L` contient tous les éléments de A_ℓ. Il s'agit de montrer que :

`L` contient tous les éléments de $A_{\ell+1} = A_\ell \cup$ `[l+1]` à la fin de l'itération $\ell + 1$

- D'après l'invariant, tous les éléments de A_ℓ sont contenus dans `L`.
- De plus, la condition `present[l] = True` indique que `l+1` est présent dans `L`. C'est ce qu'on voulait.

L'invariant est donc encore vrai à la fin de l'itération $\ell + 1$. Donc :

$\boxed{\mathscr{P}_\ell \text{ est un invariant de boucle.}}$

Correction. En sortie de boucle :

- **[1er cas]** Si $\ell = n$, cela signifie d'après l'invariant que `L` contient tous les éléments de A_n, donc comme elle est de longueur n, c'est bien une permutation.
- **[2ème cas]** Si `present[l]` vaut `False` pour un certain ℓ avec `l < n`, alors l'élément `l+1` $\in [\![1\,,\,n]\!]$ n'est pas présent dans `L`. Donc `L` n'est pas une permutation de A_n.

D'où la correction.

2.3) Calculons à présent la complexité de la seconde : en notant encore n la longueur de `L`.

- Lignes 1 à 4 : déjà calculée, complexité $1 + 2n$,
- Ligne 5 : 1 affectation,
- Ligne 6-7-8 : 1 affectation répétée n fois, et 1 affectation,
- Lignes 9->12 : pour montrer que cette fonction est meilleure que la 1ère on peut calculer la complexité dans le pire des cas uniquement, obtenu lorsque `L` contient tous les entiers de A_n. Dans ce cas on a : 2 tests, 1 addition et 1 affectation (répétés n fois), puis enfin 2 derniers tests finaux en sortie de `while`, et un tout dernier test dans le `return` (ouf!).

Bilan des courses : $C_{\text{pire}}(n) = 1 + 2n + 1 + n + 1 + (2+1+1)n + 2 + 1 = \boxed{O(n)}$. Le second programme est donc meilleur que le 1er.