Interrogation d'informatique

Ce nombre est-il une puissance de 2 ?

On souhaite savoir si un entier $n \geq 2$ peut s'écrire comme une puissance de 2, càd savoir s'il existe $p \in \mathbb{N}^*$ tel que $n = 2^p$. On propose le programme suivant :

```
1 def puissance(n):
2     '''n est un entier supérieur ou égal à 2. Cette fonction renvoie
         True si n est une puissance de 2, False sinon.'''
3     while n > 1:
4         if n % 2 == 0:
5             n = n / 2
6         else:
7             return False
8     return True
```

/2,5 **1)** Démontrer la terminaison de cet algorithme en précisant un variant de boucle.

On pose n_k la valeur de n après k itérations de la boucle `while`. Montrons que n_k est un variant de boucle.

Par hypothèse, n_0 est un entier tel que $n_0 \geq 2 > 0$. Ensuite, par récurrence, à la fin de l'itération k, si n_{k-1} est impair, la boucle se termine, et dans le cas contraire on a $n_k = \frac{1}{2}n_{k-1}$ qui est donc un entier vérifiant $n_k > 0$. Ainsi, on en déduit que pour tout k, on a $n_k \in \mathbb{N}$ et $n_k > 0$.

Enfin, comme $n_{k-1} > 0$, on a $n_k = \frac{1}{2}n_{k-1} < n_{k-1}$, donc la suite (n_k) est strictement décroissante, n_k est bien un variant de boucle. Finalement, l'algorithme termine.

/2 **2)** On note n_k la valeur de n après k itérations de la boucle `while`, la valeur n_0 correspondant à la valeur donnée en argument. Recopier et compléter le tableau suivant (rajouter des lignes le cas échéant).

k	n_k	n_k en base 2
0	24	...
⋮		
⋮		

Faire de même pour $n = 8$. Que constate-t-on ?

k	n_k	n_k en base 2
0	24	$\overline{11000}^{(2)}$
1	12	$\overline{1100}^{(2)}$
2	6	$\overline{110}^{(2)}$
3	3	$\overline{11}^{(2)}$

et

k	n_k	n_k en base 2
0	8	$\overline{1000}^{(2)}$
1	4	$\overline{100}^{(2)}$
2	2	$\overline{10}^{(2)}$
3	1	$\overline{1}^{(2)}$

On constate qu'à chaque itération, il y a un zéro de moins dans l'écriture en base 2. L'algorithme s'arrête lorsque le dernier chiffre de l'écriture en base 2 est 1.

/3 **3)** Étant donné un entier $n \geq 2$, on note q l'entier naturel tel que $2^{q-1} \leq n < 2^q$. Justifier que n s'écrit avec q chiffres en base 2.

On donne ici un corrigé avec un niveau de détail très élevé. Un tel niveau de justification n'était pas attendu au vu du peu de temps pour cette interrogation (mais si vous avez justifié très rigoureusement, vous aurez des points supplémentaires).

Soit k le nombre de chiffres de n en base 2. Montrons que $k = q$. On a :

$$n = \sum_{i=0}^{k-1} \alpha_i 2^i \qquad \text{avec } \alpha_0, \cdots, \alpha_{k-1} \in \{0,1\} \quad \text{et} \quad \alpha_{k-1} \neq 0$$

- D'une part, on a
$$n \quad \leq \sum_{i=0}^{k-1} 1 \times 2^i = \frac{2^k - 1}{2 - 1} = 2^k - 1 \quad < 2^k$$
et donc si $k < q$, on aurait $k \leq q - 1$ si bien que $n \leq 2^{k-1}$: contradiction. Donc $\boxed{k \geq q}$.
- D'autre part $\alpha_{k-1} \in \{0,1\}$ et $\alpha_{k-1} \neq 0$, donc
$$n \quad = 2^{k-1} + \sum_{i=0}^{k-2} \alpha_i 2^i \quad \geq 2^{k-1}$$
Si on avait $k > q$, on aurait $k - 1 \geq q$ et donc $n \geq 2^{k-1} \geq 2^q$, ce qui est absurde. Donc $\boxed{k \leq q}$.

/3 **4)** Exprimer la complexité de la fonction `puissance` en fonction de q.

Soit n tel que $2^{q-1} \leq n < 2^q$.

- Les opérations élémentaires sont : `>` l. 3, `%` et `==` l. 4 et `/` l.5.
- Le pire cas est lorsque $n = 2^{q-1}$: par ce qui précède n s'écrit en base 2 avec q chiffres dont $q - 1$ zéros à la fin. Ainsi, il y a donc $q - 1$ itérations de la boucle `while` : chacune va réaliser 4 opérations élémentaires (celle l. 5 est toujours exécutée sinon la fonction retournerait `False` ce qui est absurde). Il y a donc $4(q-1) + 1 = 4q - 3$ opérations élémentaires (le test l. 3 est exécuté une ultime fois pour sortir de la boucle).
- Finalement, la complexité est donc $\boxed{\text{d'ordre } q}$ (ce qui équivaut à un ordre $\log_2(n)$)

Recherche de point fixe

Étant donné une liste L et un entier naturel p, on dit que p est un point fixe si `L[p]=p` (à condition que `L[p]` ait un sens). On souhaite construire un algorithme qui recherche un tel point fixe, sachant qu'il n'est pas nécessairement unique. On commence par un algorithme "naïf".

/1,5 **1)** Écrire une fonction `pointFixe` qui prend en argument une liste `L` et renvoie l'indice `p` d'un point fixe s'il existe, et dans le cas contraire renvoie `None`.

```
1 def pointFixe(L):
2     n = len(L)
3     for k in range(n):
4         if L[k]==k:
5             return k
6     return None
```

/1,5 **2)** Quelle est la complexité de `pointFixe` ?

- Il n'y a qu'une opération élémentaire : le == ligne 4.
- Le pire cas est lorsqu'il n'y a pas de point fixe dans `L`.
- Dans ce pire cas, cette opération est répétée n fois, donc un total de n opérations élémentaires.
- La complexité est donc d'ordre n, c'est-à-dire linéaire.

À présent, on considère que L est une liste *d'entiers relatifs tous distincts triés par ordre croissant.*

/1,5 **3)** Si on constate que `L[k]>k`, que peut-on en déduire sur l'éventuel point fixe ? Même question si `L[k]<k`.

Comme la liste `L` ne contient que des entiers relatifs et est triée par ordre croissant, si on a $L[k] > k$, alors par récurrence immédiate on a pour tout $m \geq k$: $L[m] > m$. Ainsi, un point fixe ne peut pas avoir un indice supérieur ou égal à k.

De même, si $L[k] < k$, alors un point fixe ne peut pas avoir un indice inférieur ou égal à k.

/3,5 **4)** S'inspirer de la méthode de dichotomie pour concevoir une fonction `PFtri` qui prend en argument une liste `L` qui correspond aux conditions ci-dessus et qui renvoie l'indice `p` d'un point fixe s'il existe, et dans le cas contraire renvoie `None`.

```
1  def PFtri(L):
2      g,d = 0,len(L)-1   # indices extrêmes de L
3
4      while g <= d:      # tant qu'il y a un élément dans L[g:d+1]
5          m = (g+d)//2   # on regarde le "milieu" de L[g:d+1]
6          if L[m]==m:
7              return m   # si ce milieu est un point fixe : c'est fini
8          elif L[m]<m:
9              g = m+1    # si L[m]<m, alors le point fixe ne peut
                             être que dans L[m+1:d+1]  (de m+1 à d inclus)
10         else:
11             d = m-1    # si L[m]>m, alors le point fixe ne peut
                             être que dans L[g:m]   (de g à m-1 inclus)
12
13     return None
```

Tournez la page S.V.P.

Questions diverses

/2 **1)** On considère l'algorithme suivant :

```
1  def mystere(S):  # S est une chaine de caractères
2      if S=="":
3          return ""
4      else:
5          return S[-1] + mystere(S[0:-1])
```

Que renvoie `mystere("X")` ? et `mystere("abc")` ? Quelle opération réalise finalement la fonction `mystere` ?

- `mystere("X")` renvoie `"X"`
- `mystere("abc")` renvoie `"cba"`
- `mystere` retourne la chaine de caractères qui correspond à la chaine initiale mais dont les caractères apparaissent dans l'ordre inverse.

/2,25

2) Expliquer le principe général du tri fusion. Quelle est sa complexité en fonction de la taille de la liste à trier ?

Le tri fusion sert à trier les éléments d'une liste L par ordre croissant (par exemple). Le principe est de séparer la liste L en deux sous-listes de tailles à peu près égales, et de les trier avec la méthode de tri fusion (c'est une fonction récursive, on continue jusqu'à avoir des listes de taille 1, qui sont déjà triées). Une fois que les deux sous-listes ont été triées, si on les note $L1$ et $L2$, liste L de départ sera triée en fusionnant les deux listes $L1$ et $L2$ par un interclassement des leurs éléments.

Si n est la taille de la liste à trier, la complexité est d'ordre $n\log_2(n)$, ou encore log-linéaire.

/2,25

3) Écrire une fonction `sansDoublon` qui à une liste `L` retourne une liste qui contient les valeurs contenues dans `L`, sans doublon.

```
def sansDoublon(L):
    M = []       # la liste qu'on va retourner
    for v in L: # boucle sur les valeurs
        if v not in M:
            M.append(v)  # on ajoute v à M s'il n'y est pas déjà
    return M
```

/20

4) (bonus très difficile, bon courage !!) On considère l'algorithme suivant :

```
def calc(n):
    if n>100:
        return n-9
    else:
        return calc(calc(n+10))
```

Montrer que pour tout $n \in [\![0, 100]\!]$, l'instruction `calc(n)` renvoie 92.

On résout le problème par une récurrence forte couplée à une récurrence descendante, donc par une double récurrence forte descendante (oui oui). Pour tout $k \in \mathbb{N}$, on pose :

$$I_k = [\![91 - 10k, 100 - 10k]\!]$$

On va démontrer (par récurrence forte) la véracité, pour tout $k \in \mathbb{N}$, de l'assertion $\mathcal{P}_k$ suivante : pour tout $n \in I_k$, l'instruction `calc(n)` renvoie 92.

- Montrons $\mathcal{P}_0$, i.e. que pour tout $n \in I_0 = [\![91, 100]\!]$, `calc(n)` renvoie 92. (On procède encore par une récurrence, sur n ici, et de manière descendante).
  - Si $n = 100$, alors `calc(n)` renvoie le résultat de `calc(calc(110))`, donc de `calc(101)`, i.e. 92.

- On suppose que pour un $n \in [\![91, 99]\!]$, l'instruction `calc(n+1)` renvoie 92. Montrons que c'est aussi le cas pour `calc(n)`. Pour un tel n, `calc(n)` renvoie `calc(calc(n+10))`. Or $n + 10 > 100$, donc `calc(n+10)` retourne $n + 1$. Ainsi, `calc(n)` renvoie `calc(n+1)`, qui renvoie 92 par hypothèse de récurrence.
- Ainsi, l'instruction `calc(n)` renvoie 92 pour tout $n \geq 91$.

- On suppose qu'il existe $k \in \mathbb{N}$ tel que pour tout $N \in I_0 \cup I_1 \cup \ldots \cup I_k = [\![91 - 10k, 100]\!]$, l'instruction `calc(N)` renvoie 92 (on l'a montré plus haut pour $k = 0$). Montrons que pour tout $n \in I_{k+1}$, l'instruction `calc(n)` renvoie 92.

  Soit donc $n \in I_{k+1}$. L'instruction `calc(n)` renvoie `calc(calc(n+10))`. Or $n + 10 \in I_k$, donc par hypothèse de récurrence, `calc(n+10)` renvoie 92. On en déduit que l'instruction renvoie `calc(92)`, ce qui, comme $92 \in I_0$, renvoie encore 92 par hypothèse de récurrence.
- Finalement, pour tout $k \in \mathbb{N}$, l'assertion $\mathcal{P}_k$ est vraie. Ainsi, pour tout entier $n \leq 100$, l'instruction `calc(n)` renvoie 92 (donc en particulier si $n \in [\![0, 100]\!]$).