Interrogation d'informatique n° 3
MP
mardi 25 mars 2025, 2 heures

Lycée Camille PISSARRO

Exercice 1

On note ζ la fonction zêta de Riemann définie sur $]1,+\infty[$ par : $\zeta(x)=\sum_{n=1}^{+\infty}\frac{1}{n^x}$. La suite des nombres de Bernoulli notée $(b_n)_{n\in\mathbb{N}}$ est définie par :

$$b_0=1,\quad \forall n\geqslant 1,\quad b_n=\frac{-1}{n+1}\sum_{k=0}^{n-1}\binom{n+1}{k}b_k$$

Leonhard Euler (1707-1783) a démontré la formule suivante qui exprime les nombres $\zeta(2k)$ à l'aide des nombres de Bernoulli :

$$\forall k\in\mathbb{N}^*,\zeta(2k)=\frac{(-1)^{k-1}2^{2k-1}\pi^{2k}b_{2k}}{(2k)!}$$

On se propose de programmer le calcul des nombres de Bernoulli b_n afin d'obtenir des valeurs exactes de $\zeta(2k)$.

1. Écrire une fonction `factorielle(n)` qui renvoie la factorielle d'un entier $n\in\mathbb{N}$.
2. On considère la fonction Python suivante `binom(n,p)` qui renvoie le coefficient binomial $\binom{n}{p}$:

```
def binom(n, p):
    if not(0 <= p <= n):
        return 0
    return factorielle(n) // (factorielle(p)*factorielle(n-p))
```

   Combien de multiplications sont effectuées lorsque l'on exécute `binom(30,10)` ? Expliquer pourquoi il est possible de réduire ce nombre de multiplications à 20.
   Quel serait le type du résultat renvoyé si l'on remplaçait la dernière ligne de la fonction `binom` par `return factorielle(n)/(factorielle(p)*factorielle (n-p))` ?
3. Démontrer que, pour $n\geqslant p\geqslant 1$, on a

$$\binom{n}{p}=\frac{n}{p}\binom{n-1}{p-1}$$

   En déduire une fonction récursive `binom_rec(n,p)` qui renvoie le coefficient binomial $\binom{n}{p}$.
4. Écrire une fonction non récursive `bernoulli(n)` qui renvoie une valeur approchée du nombre rationnel b_n. On pourra utiliser librement une fonction `binomial(n, p)` qui renvoie le coefficient binomial $\binom{n}{p}$. Par exemple, `bernoulli(10)` renvoie 0,07575757575757576 qui est une valeur approchée de $b_{10}=\frac{5}{66}$.

Exercice 2

On dit qu'un entier naturel n est premier si, et seulement si, il admet exactement deux diviseurs dans $\mathbb{N}$: 1 et lui-même. 0 et 1 ne sont donc pas des nombres premiers. Par contre, 3 est un nombre premier puisque l'ensemble des diviseurs de 3 dans $\mathbb{N}$ est $\{1, 3\}$.
Toutes les fonctions demandées sont à écrire en langage Python. On pourra bien entendu, au fil des questions, utiliser les fonctions construites dans les questions précédentes.

1. Écrire une fonction `divise(p, q)` d'argument deux entiers naturels non nuls p et q, renvoyant `True` si p divise q et `False` sinon.
2. Écrire une fonction `estpremier(p)` d'argument un entier naturel p, renvoyant 1 si p est premier et 0 sinon.
3. Écrire une fonction `phi(p)` d'argument un entier naturel p, renvoyant le nombre de nombres premiers inférieurs ou égaux à p.

Pour $n \in \mathbb{N}$, on désigne par $\varphi(n)$ le nombre de nombres premiers inférieurs ou égaux à n. Dans la suite de l'exercice, on admet le théorème des nombres premiers :

$$\varphi(n) \underset{n\to+\infty}{\sim} \frac{n}{\ln(n)}$$

Pour $n \in \mathbb{N}^*$, on pose $\Theta(n) = \left|\frac{\varphi(n)\ln(n)}{n} - 1\right|$.

4. Quelle est la limite de $\Theta(n)$ quand n tend vers l'infini ?
5. En utilisant le théorème des nombres premiers, montrer qu'il existe une infinité de nombres premiers.
6. Écrire une fonction `test(epsilon)` d'argument un réel ε strictement positif, renvoyant le premier entier naturel $N \geqslant 50$ tel que $\Theta(N) \leqslant \varepsilon$.
7. Question bonus.
   Donner une suite d'instructions permettant de tracer le graphe de la fonction Θ sur $[\![50, 5000]\!]$.

Exercice 3

On considère la suite donnée par $t_0 = 12$, $t_1 = 13$, $t_2 = 14$ et pour $n \in \mathbb{N}$,

$$t_{n+3} = t_{n+2} + t_{n+1} - 2t_n$$

Écrire un programme utilisant une fonction récursive et un dictionnaire de mémoïsation, pour le calcul de t_p, $p \in \mathbb{N}$.

Exercice 4

On se donne une liste `L`, contenant des réels quelconques, de longueur n. On cherche à répondre à deux questions :

- Existe-t-il une valeur apparaissant strictement plus de $n/2$ fois dans la liste ? (une telle valeur est un élément majoritaire de la liste).
- Si ce n'est pas le cas, quelle est la valeur apparaissant avec la plus grande fréquence ?

De telles questions peuvent se poser dans des domaines aussi variés que le traitement du signal ou le dépouillement d'élections.

1. Écrire une fonction `Test_majoritaire(L, x)` permettant de déterminer si x est majoritaire dans L ou non. Cette fonction doit renvoyer un booléen.
2. On considère le programme suivant :

```
def Majoritaire(L):
    for x in L:
        if Test_majoritaire(L, x):
            return x
    return False
```

   Expliquer ce que renvoie le programme, et donner sa complexité. Toutes les affirmations doivent être justifiées, de manière *concise* mais *précise*.

3. Dans cette question, on suppose que la liste `L` est triée par ordre croissant :

$$\mathrm{L}[0] \leqslant \mathrm{L}[1] \leqslant \cdots \leqslant \mathrm{L}[n-1]$$

   Écrire une fonction `Majoritaire_trie(L)`, dont la complexité sera linéaire par rapport à la longueur n de `L`, renvoyant :
   - la valeur de l'élément majoritaire de `L` s'il en existe un,
   - le booléen `False` sinon.

4. Le langage Python permet de trier une liste : l'instruction `sorted(L)` renvoie une nouvelle liste comportant tous les éléments de `L`, triés par ordre croissant. En admettant que cette instruction est de complexité $\Theta(n \ln n)$, donner la complexité de la fonction suivante :

```
def Majoritaire_optimise(L):
  T = sorted(L)
  return Majoritaire_trie(T)
```

5. Dans le cas où aucun élément n'est majoritaire, on souhaite déterminer quel élément apparaît le plus souvent – ou quels éléments, en cas d'ex-aequo. On appellera *score* d'un réel le nombre de fois où il apparaît dans la liste.
   Écrire une fonction `Meilleur_Score(L)` prenant une liste de réels en argument et retournant le couple `(podium, score)` où :
   - `score` est le score maximal atteint par les éléments de la liste ;
   - `podium` est la liste de tous les éléments ayant obtenu le score maximal.

   Donner la complexité de votre algorithme.
