corrigé

Coupes d'un tableau

Question 1.

```
n = 10000
u = 13
t = [u % 2]
for i in range(1, n):
    u = (u * 16365) % 65521
    t.append(u % 2)
    if i == 1000 or i == 5000:
        print("u({}) = {}".format(i, u))
```

Question 2.

```
s = 0
for i in range(n):
    if t[i] == 1:
        s += 1
        if s == 1000:
            i0 = i
print("nb d'éléments égaux à 1 :", s)
print("indice du 1000e :", i0)
```

Question 3. Je commence par définir une fonction qui détermine si la coupe $t[i : j]$ est un palindrome :

```
def est_un_palindrome(i, j):
    for k in range(i, (i + j) // 2):
        if t[k] != t[i + j - 1 - k]:
            return False
    return True
```

```
s = 0
for i in range(n - 7):
    if est_un_palindrome(i, i + 7):
        s += 1
        if s == 2:
            i2 = i
print("nb de palindromes de longueur 7 :", s)
print("indice du 2e :", i2)
```

Question 4. Je commence par définir une fonction qui détermine s'il existe un palindrome de longueur ℓ dans t. Si c'est le cas, la fonction retourne le booléen `True` associé à l'indice du premier de ces palindromes ; s'il n'en existe pas la fonction retourne le booléen `False` associé à la valeur -1.

```
def cherche_palindrome(ell):
    for i in range(n - ell + 1):
        if est_un_palindrome(i, i + ell):
            return True, i
    return False, -1
```

Pour une valeur de ℓ fixée, cette fonction est de coût linéaire, mais si on l'applique pour tout entier ℓ compris entre 1 et $n = 10\,000$ à la recherche d'un palindrome de longueur maximale, on obtient un algorithme de coût quadratique rédhibitoire. Il faut donc trouver un moyen de stopper la recherche quand on est certain de ne plus trouver de palindrome. Pour cela, on utilise la remarque suivante : s'il existe dans t un palindrome de longueur $\ell > 2$, il en existe aussi de longueur

$\ell - 2$. *A contrario*, s'il n'existe pas de palindrome de longueur ℓ et $\ell - 1$, on peut affirmer que la recherche est terminée. Cette remarque conduit au script suivant :

```
lmax = 1
for ell in range(2, n):
    r, i = cherche_palindrome(ell)
    if r:
        lmax, imax = ell, i
    elif ell == lmax + 2:
        break
print("taille maximale d'un palindrome :", lmax)
print("indice du premier :", imax)
```

Notons que cette fonction reste de coût quadratique dans le pire des cas, par exemple lorsque le tableau ne contient que des 0 (et où toute coupe est un palindrome). Mais si on admet que la distribution des 0 et des 1 dans le tableau est uniforme, cet évènement est très peu probable.

Question 5.

```
lmax = ell = 0
for i in range(n):
    if t[i] == 0:
        ell += 1
        if ell >= lmax:
            lmax, imax = ell, i - ell + 1
    else:
        ell = 0
print("taille maximale d'un plateau :", lmax)
print("indice de début du dernier :", imax)
```

Question 6. Je commence par définir une fonction qui calcule l'interprétation d'une coupe :

```
def bintodec(i, j):
    x = 0
    for k in range(i, j):
        x = 2 * x + t[k]
    return x
```

```
print("interprétation de la coupe t[1000: 1020] :", bintodec(1000, 1020))
```

Question 7. J'écris cette fois une fonction qui convertit en base 2 un entier :

```
def dectobin(k):
    x = []
    while k > 0:
        x.append(k % 2)
        k //= 2
    return x[::-1]
```

puis une fonction qui détermine si un tableau x est un sous-tableau de t :

```python
def est_dans(x):
    p = len(x)
    for i in range(n - p + 1):
        b = True
        for j in range(p):
            if x[j] != t[i + j]:
                b = False
                break
        if b:
            return True
    return False
```

```python
k = 0
while est_dans(dectobin(k)):
    k += 1
print("plus petit entier absent dans t :", k)
```

Question 8. Je commence par définir une fonction qui teste la primalité d'un entier $p \geqslant 2$ (il ne doit pas être divisible par un entier $2 \leqslant k \leqslant \sqrt{p}$) :

```python
def is_prime(p):
    k = 2
    while True:
        if k * k > p:
            return True
        if p % k == 0:
            return False
        k += 1
```

```python
pmax = 0
for i in range(len(t) - 19):
    p = bintodec(i, i + 20)
    if is_prime(p) and p > pmax:
        pmax = p
print("coupe première maximale :", pmax)
```

Question 9. Déterminer si une coupe est équilibrée a un coût linéaire vis-à-vis de la taille de cette coupe, donc passer en revue toutes les coupes (il y en a $n(n-1)/2$) à la recherche de celles qui sont équilibrées a un coût en $O(n^3)$. Une telle recherche est bien évidemment exclue pour $n = 10\,000$.

Appelons *déséquilibre* de $t[:i]$ la différence entre le nombre de 1 et le nombre de 0 dans la coupe $t[:i]$. On peut observer qu'une coupe $t[i:j]$ est équilibrée lorsque les déséquilibres de $t[:i]$ et $t[:j]$ sont égaux. Or calculer le déséquilibre de chaque coupe $t[:i]$ peut être réalisé en un seul parcours de t, donc en coût linéaire. Une fois ceci effectué, il restera à trouver le couple (i,j) le plus écarté possible possédant le même déséquilibre.

Je commence par remplir un dictionnaire dont les clefs sont les déséquilibres et les valeurs la liste des indices ayant ce déséquilibre :

```
d = {0: [0]}
s = 0
for i in range(n):
    if t[i] == 0:
        s -= 1
    else:
        s += 1
    if s not in d:
        d[s] = [i + 1]
    else:
        d[s].append(i + 1)
```

Il reste alors à trouver la longueur maximale d'une coupe équilibrée :

```
lmax = 0
for s in d:
    if d[s][-1] - d[s][0] > lmax:
        lmax = d[s][-1] - d[s][0]
print("taille maximale d'une coupe équilibrée :", lmax)
```