# Corrigé de l'épreuve de travaux pratiques

**Remarque**. Nous avons pu constater lors de TP consacré aux coupes minimales que pour un tableau de taille 10 000, un algorithme de coût quadratique peut prendre de nombreuses minutes avant de retourner un résultat. la difficulté relative de certaines questions réside dans l'obtention d'un algorithme de coût linéaire, seul à même de retourner un résultat en un temps raisonnable.

**Question 1.**

```
u = 13
t = [u % 2]
for i in range(1, 10000):
    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, b in enumerate(t):
    if b == 1:
        s += 1
        if s == 1000:
            i0 = i
print("nb d'éléments égaux à 1 :", s)
print('indice du 1000e : ', i0)
```

**Question 3.** Je choisis de définir une fonction qui détermine si un tableau est un palindrome, en remarquant que `s[::-1]` retourne l'image miroir d'une liste `s`.

```
def est_un_palindrome(s):
    return s == s[::-1]
```

Cette fonction n'est pas optimale (elle effectue dans le pire des cas $n = |s|$ comparaisons alors que $\lfloor n/2 \rfloor$ suffiraient) mais reste de coût linéaire vis-à-vis de la longueur du tableau.
On répond ensuite à la question du problème en utilisant le script :

```
s = 0
for i in range(len(t)-6):
    if est_un_palindrome(t[i:i+7]):
        s += 1
        if s == 2:
            i2 = i
print('nb de palindromes de longueur 7 :', s)
print('indice du 2e :', i2)
```

La fonction `est_un_palindrome` appliquée à des mots de longueur 7 est de coût constant donc le coût de ce script est linéaire vis-à-vis de la taille de $t$.

**Question 4.** Commençons par définir une fonction qui détermine s'il existe un palindrome de longueur $\ell$ 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(l):
    for i in range(len(t)-l+1):
        if est_un_palindrome(t[i:i+l]):
            return True, i
    return False, -1
```

Pour une valeur de $\ell$ fixée, cette fonction est de coût linéaire, mais si on l'applique pour tout entier $\ell$ 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 $\ell \geqslant 2$, il en existe aussi de longueur $\ell - 2$. *A contrario*, s'il n'existe pas de palindrome de longueur $\ell$ et $\ell - 1$, on peut affirmer que la recherche est terminée. Cette remarque conduit au script suivant :

```
lmax = 1
for l in range(2, 10001):
    r, i = cherche_palindrome(l)
    if r:
        lmax, imax = l, i
    elif l == 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 : la probabilité qu'une coupe de taille $\ell$ soit un palindrome est égale à $2^{-\lfloor \ell/2 \rfloor}$, et la probabilité qu'il existe une coupe palindrome de longueur $\ell$ dans le tableau $t$ inférieure à $1 - \left(1 - 2^{-\lfloor \ell/2 \rfloor}\right)^{10001-\ell}$.
Pour $\ell \geqslant 47$ cette probabilité tombe en dessous de 1‰ (moins d'une chance sur 1 000). Vous ne courrez quasiment aucun risque à limiter la recherche aux palindromes de longueur inférieure à 50, ce qui rend celle-ci de coût linéaire. On obtiendra donc très vraisemblablement le même résultat avec le script plus simple suivant :

```
lmax = 1
for l in range(2, 50):
    r, i = cherche_palindrome(l)
    if r:
        lmax, imax = l, i
print("taille maximale d'un palindrome :", lmax)
print('indice du premier :', imax)
```

**Question 5.** Le script qui suit utilise deux invariants : `lmax` désigne la longueur du plus long plateau contenu dans $t[:i]$ et `l` la longueur du plus long plateau de la forme $t[k:i]$ où $k \leqslant i$.

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

**Question 6.** Déterminer si une coupe est équilibrée prend 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 $O(n^3)$, où $n$ désigne la taille du tableau. Une telle recherche est bien évidemment exclue pour $n = 10\,000$ et ici, contrairement à la question 4 il ne peut être question d'un argument probabiliste pour espérer que la recherche se termine rapidement.

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.
Sachant que le déséquilibre est compris entre $-10\,000$ et $10\,000$, nous allons utiliser deux tableaux $d$ et $f$ de taille 20 001. Pour tout $x \in [\![-10\,000, 10\,000]\!]$, $d[x]$ désignera le plus petit des entiers $i$ (s'il en existe) pour lequel le déséquilibre de $t[:i]$ est égal à $x$ et $f[x]$ le plus grand de ces entiers $i$. Pour toute valeur $x$ pour laquelle $i = d[x]$ et $j = f[x]$ sont définies la coupe $t[i:j]$ est équilibrée, et cette recherche prend un temps proportionnel au nombre de valeurs de $x$ à tester, donc un coût linéaire.

**Remarque.** Notez la manière adroite d'utiliser la possibilité d'indexer par des entiers négatifs les items d'une liste Python. Dans un tableau de taille $2n+1$, les $n+1$ premières cases sont indexées entre 0 et $n$, les $n$ suivantes entre $-n$ et $-1$ (ce qui n'est pas sans rappeler la représentation par complément à deux des entiers relatifs).

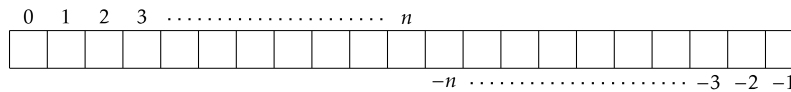


FIGURE 1 – Indexation par les entiers de $[\![-n, n]\!]$ d'un tableau de taille $2n+1$.

```python
d = [None] * 20001
f = [None] * 20001
s = 0
for i, b in enumerate(t):
    if b == 0:
        s -= 1
    else:
        s += 1
    if d[s] is None:
        d[s] = i
    f[s] = i
lmax = 0
for x in range(-10000, 10001):
    if d[x] is not None and f[x] is not None:
        if f[x] - d[x] > lmax:
            lmax = f[x] - d[x]
print("taille maximale d'une coupe équilibrée :", lmax)
```

**Question 7.** On définit une fonction qui convertit une coupe en nombre décimal :

```python
def bintodec(t):
    x = 0
    for b in t:
        x = 2 * x + b
    return x
```

puis on utilise le script :

```python
print('interprétation de la coupe t[1000:1020] :', bintodec(t[1000:1020]))
```

**Question 8.** On définit la fonction réciproque :

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

puis une fonction qui détermine si un tableau $x$ est une coupe de $t$ :

```python
def est_dans(x, t):
    for i in range(len(t)-len(x)+1):
        if x == t[i:i+len(x)]:
            return True
    return False
```

On utilise alors le script :

```python
n = 0
while est_dans(dectobin(n), t):
    n += 1
print('plus petit entier absent dans t :', n)
```

**Question 9.** On définit une fonction qui teste la primalité d'un nombre. On se contente du critère naïf : est premier un nombre qui ne possède pas de diviseur dans l'intervalle $[\![2, \sqrt{n}]\!]$.

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

puis on utilise le script :

```
pmax = 0
for i in range(len(t)-19):
    p = bintodec(t[i:i+20])
    if is_prime(p) and p > pmax:
        pmax = p
print('coupe première maximale :', pmax)
```