CORRIGÉ DU DS 1

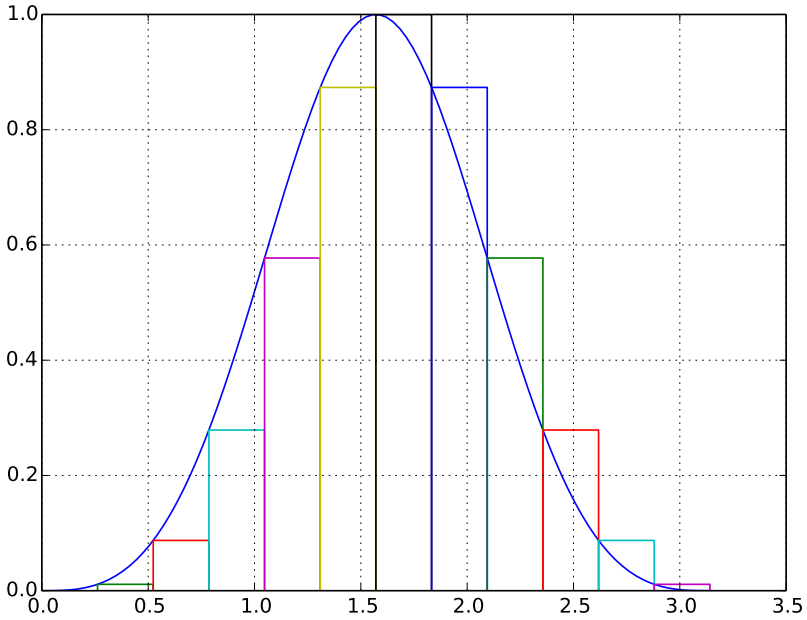
le 27/09/2025 – Durée : 2h

- L'usage de la calculatrice n'est pas autorisé.
- La clarté et la précision des raisonnements interviendront pour une grande part dans la notation. La présentation (en particulier les indentations) est également essentielle.
- Le résultat d'une question peut être admis afin de traiter une question suivante ; une fonction peut être utilisée dans la suite d'un exercice même si elle n'a pas été écrite.
- L'exercice 3 est à aborder en dernier et sera compté en bonus.

Exercice 1

D'APRÈS E3A MODÉLISATION PSI 2015

Question 1 –



$$\lambda_n = \pi/n$$

Question 2 –

```
def f(x):
    return ((cos(pi*cos(x))/2)**2/sin(x))
```

Question 3 –

```
def rectangle1(n):
    S=0
    for k in range(1,n):
        S+=f(k*pi/n)
    return pi*S/n
```

Question 4 – Pour une valeur donnée de n, cette fonction nécessite $n-1$ évaluations de la fonction f.

Question 5 –

```
def integrale1(eps):
    n = 2
    R_prec = rectangle1(n-1) # R_{n-1}
    R = rectangle1(n) # R_n
    while abs(R-R_prec) >= eps :
        n += 1
        R_prec = R
        R = rectangle1(n)
    return R
```

Question 6 – Au passage numéro n dans la boucle, on calcule `rectangle1(n)`, ce qui nécessite $n-1$ évaluations de la fonction `f`. On aura donc en tout $1+2+\cdots+33$ évaluations de la fonction `f` ; c'est à dire un nombre d'évaluations N égal à :

$$N = \frac{34 \times 33}{2} = 17 \times 33 = 561 > 500 \ .$$

Question 7 –

```
def rectangle2(m):
    if m == 0:
        return 0
    else :
        S = 0
        for k in  range(1,2**(m-1)+1):
            S += f((k-0.5)*pi/(2**(m-1)))
        R = rectangle2(m-1)
        return R/2+(pi/(2**m))*S
```

Question 8 – Pour passer, en utilisant la fonction récursive, du calcul de $R_{2^{m-1}}$ à celui de R_{2^m}, on a besoin de 2^{m-1} évaluations de la fonction `f`. En tout, pour calculer R_{2^m}, le nombre d'évaluations de la fonction `f` est donc :

$$1 + 2 + 2^2 + \cdots + 2^{m-1} = \frac{1-2^m}{1-2} = 2^m - 1 \ .$$

Remarquons que ce nombre est le même que celui nécessité par la fonction `rectangle1(2**m)`.

Question 9 –

```
def integrale2(eps):
    m = 1
    R_prec = 0 # R_{2**(m-1)}
    R = rectangle1(2) # R_{2**(m)}
    while abs(R-R_prec) >= eps:
        m += 1
        R_prec = R
        n = 2**(m-1)
        S = 0
        for k in  range(1,n+1):
            S += f((k-0.5)*pi/n)
        R = R_prec/2+(pi/(2*n))*S
    return R
```

Question 10 – Dans la fonction `integrale2(eps)`, lors du passage dans la boucle while qui fait passer de m à $m+1$, on effectue évaluations de la fonction `f`. Comme la dernière valeur de m est 7, le nombre total d'évaluations de f effectué est donc :

$$1+2+2^2+\cdots+2^6=\frac{1-2^7}{1-2}=2^7-1=127\ .$$

On a là nettement gagné par rapport à la fonction `integrale1(eps)`.

Question 11 – on a, pour $n\in\mathbb{N}^*$,

$$R_{2n}=\frac{\pi}{n}\sum_{k=1}^{2n-1}f\left(\frac{k\pi}{2n}\right)\ ;$$

ce qui donne en séparant les valeurs paires et impaires de k :

$$R_{2n}=\frac{\pi}{2n}\sum_{j=1}^{n-1}f\left(\frac{2j\pi}{2n}\right)+\frac{\pi}{2n}\sum_{j=1}^{n}f\left(\frac{(2j-1)\pi}{2n}\right)\ ;$$

d'où

$$R_{2n}=\frac{1}{2}\frac{\pi}{n}\sum_{j=1}^{n-1}f\left(\frac{j\pi}{n}\right)+\frac{\pi}{2n}\sum_{j=1}^{n}f\left(\frac{(2j-1)\pi}{2n}\right)=\frac{1}{2}R_n+\frac{\pi}{2n}\sum_{j=1}^{n}f\left(\frac{(j-1/2)\pi}{n}\right)\ .$$

Exercice 2

1. **Une première version de la recherche du nombre d'éléments distincts**

(a)
```
def Premier(L,i):
    m=L[i]
    for j in range(i):
        if L[j] == m:
            return False
    return True
```

(b)
```
def NombreDistincts(L):
    n=len(L)
    N=0
    for i in range(n):
        if Premier(L,i) :
            N=N+1
    return N
```

(c) Chaque appel à `Premier(L,i)` nécessite $i-1$ comparaisons, donc la fonction `NombreDistincts(L)` nécessite $1+2+\cdots+n-1$ comparaisons (le nombre d'affectations est négligeable). La complexité est donc :

$$C(n) = \frac{(n-1)n}{2} = \mathcal{O}(n^2) \ .$$

2. **Cas d'une liste triée**

(a)
```
def NombreDistinctsTri(L):
    n = len(L)
    i = 1
    N = 1
    while i < n :
        if L[i] != L[i-1]:
            N = N+1
        i = i+1
    return N
```

(b) On parcourt la liste une seule fois, avec à chaque étape au plus 3 opérations, donc l'ordre de grandeur de la complexité est $\mathcal{O}(n)$.

(c) L'invariant de la boucle constituant la fonction `NombreDistinctsTri(L)` est le suivant : `L[:i]` contient `N` éléments distincts.

- L'invariant est vérifié à l'initialisation car `L[:1]` contient 1 éléments.
- L'invariant est maintenu à chaque passage dans la boucle. En effet si `L[:i]` contient `N` éléments distincts alors :
  - Soit `L[i]==L[i-1]` et alors `L[:i+1]` contient `N` éléments distincts (ce qui se traduit dans l'algorithme par le fait qu'on ne modifie pas `N`).

– Soit `L[i]>L[i-1]`, et alors pour tout $0 \leq j < i$, `L[i]!=L[j]` car `L` est trié ; donc dans ce cas `L[:i+1]` contient `N+1` éléments distincts, qui est bien la nouvelle valeur de `N`.

- L'invariant et la condition d'arrêt impliquent que le programme renvoie le nombre d'éléments distincts de `L[:n]` et donc respecte son objectif : renvoyer le nombre d'éléments distincts de`L`.

(d)
```
def NombreDistinctsRec(L):
    n=len(L)
    if n == 0:
        return 0
    elif n == 1 :
        return 1
    else :
        if L[n-1]==L[n-2]:
            return NombreDistinctsRec(L[:n-1])
        else :
            return NombreDistinctsRec(L[:n-1])+1
```

(e)
```
def NombreDistinctsRec2(L):
    def NombreDistinctsRecAux(L,i):
        if i == 0:
            return 0
        elif i == 1:
            return 1
        else :
            if L[i-1] != L[i-2] :
                return NombreDistinctsRecAux(L,i-1) + 1
            else :
                return NombreDistinctsRecAux(L,i-1)

    return NombreDistinctsRecAux(L , len(L) )
```

3. **Tri d'une liste contenant au plus N éléments distincts connus**

(a)
```
def Comptage(L,N):
    n=len(L)
    C=N*[0]
    for k in range(n):
        C[L[k]]+=1
    return C
```

(b) Remarquons que pour obtenir une liste contenant les éléments de `L` triés, il suffit, à partir du résultat `C` de `Comptage(L,N)`, d'empiler `C[0]` fois 0 puis `C[1]` fois 1 ...

```
def tri(L,N):
    n = len(L)
    C = Comptage(L,N)
    M = [0 for i in range(n)]
    i = 0
    for k in range(N):
        for j in range(C[k]):
            M[i] = k
            i += 1
    return M
```

(c) La fonction comptage est clairement linéaire en fonction de n. Ensuite, pour remplir `M`, on fait exactement n affectations (si l'on a initialisé `M` à une liste vide et qu'on la fait évoluer par des appels à `append`, le raisonnement reste valable si l'on considère `append` comme une opération élémentaire). La complexité de ce tri est donc linéaire. Elle est donc meilleure que celle ($\mathcal{O}(n^2)$) des tris insertion ou sélection.
Notons bien que l'on est ici dans un cas particulier et que l'on perd un peu en complexité en espace car il faut créer deux nouvelles listes.

Exercice 3

1.
```
def points_fixes(t):
    l = []
    for i in range(n):
        if t[i] == i :
            l.append(i)
    return l
```

Remarque : vu l'énoncé, la définition de `n` n'était pas indispensable.

2.
```
def itere(t,x,k):
    a=x
    for i in range(k):
        a=t[a]
    return(a)
```

3.
```
def nb_points_fixes__iteres(t,k):
    N=0
    for i in range(len(t)):
        if itere(t,i,k)==i :
            N=N+1
    return(N)
```

4. On vérifie tout d'abord que f admet un seul point fixe.
Il faut également remarquer que si il existe un entier $k \geq 0$ tel que $f^k(x) = z$, alors il existe un tel entier $k \leq n$ et $f^n(x) = z$.

```
def admet_attracteur_principal(t):
    n=len(t)
    l=points_fixes(t)#liste des points fixes
    if len(l)!=1:
        return False
    else :#pas indispensable
        z=l[0]#unique point fixe
        for x in range(len(t)):
            if itere(t,x,n)!=z:
                return False
        return True
```

Cet algorithme n'est pas optimal.