DS 1

Exercice 1. Écrire les fonctions `factorielle(n)` (itératif) et `factorielle_rec(n)` (récursif). Quelle est la complexité de ces fonctions ? Donner un inconvénient de la version récursive : en python, pour quelles valeurs de n aura-t-on un problème ?

Exercice 2. Écrire une fonction en python permettant de calculer le pgcd de deux entiers positifs a et b.

Écrire une fonction récursive permettant de calculer le plus grand diviseur commun d'un ensemble de nombres (donnés dans un tableau) : `pgcd_tableau(t)`. On ne fera pas attention à la complexité.

Exercice 3. Écrire une fonction permettant de déterminer si un entier n est premier. De quel type est la valeur renvoyée ? Expliquer les optimisations que vous avez faites.

Exercice 4. On considère deux listes d'entiers (type `list` en Python), `u` et `v`, déjà triées dans l'ordre croissant. Écrire une fonction `fusion(u, v)` qui renvoie la fusion des deux listes u et v (triée aussi). Quelle est la complexité de votre fonction ?

`fusion([1, 7, 7, 12], [-1, 2, 11, 12, 13])` doit renvoyer `[-1, 1, 2, 7, 7, 11, 12, 12, 13]`

Exercice 5. Soit un point M du plan de coordonnées (x, y) dans un repère orthonormé direct (O, I, J). Écrire un une fonction `trigo(x, y)` permettant de calculer $cos((\vec{OI}, \vec{OM}))$. On pourra utiliser la fonction `sqrt` du module `math` qui renvoie la racine carrée d'un flottant.

Exercice 6. [Récursivité] Écrire une fonction `dichotomie(t, x)` qui cherche si l'élément x est présent dans un tableau (trié) t (type `list`) par dichotomie en utilisant une fonction récursive. Quelle est la complexité de votre fonction : attention le slicing $t[a:b]$ a pour complexité $\Theta(b-a)$.

Exercice 7. `import matplotlib.pyplot as plt`

Écrire un bloc de code permettant d'afficher une courbe représentative de la fonction $f : x \to x^3 + 1$ sur $[0;1]$ (avec la précision que vous voulez). Commentez votre code.

Exercice 8. La fonction `random()` du module `random` renvoie un flottant aléatoire dans $[0;1[$. En *déduire* une fonction python `alea(n)` qui renvoie un entier tiré au hasard dans $[\![1;n]\!]$ et écrire un bloc de code qui demande un entier k à l'utilisateur (avec `input`) et affiche (avec `print`) un entier tiré au hasard dans $[\![1;k]\!]$.

Exercice 9. SQL : On considère un nuage de points associés à des prénoms stockés dans une base de données. Un prénom peut apparaître plusieurs fois.

Le schéma relationnel associé est

$$points\{\underline{id : integer}; x : integer; y : integer; prenom : text\}$$

1. Expliquer à quoi sert le champ id et ce que signifie le fait qu'il soit souligné.
2. Écrire une requête SQL qui donne la plus grande valeur de x et la plus petite valeur de y qui apparaissent dans la base de donnée.
3. Écrire une requête SQL qui compte combien de points sont associés à chaque prénom.
4. Écrire une requête SQL qui donne les coordonnées du point moyen (moyenne des x_i, moyenne des y_i). Remarque : ce point n'appartient pas forcément à l'ensemble des points dans la base de donnée.
5. Écrire une requête SQL qui donne les noms associés aux points les plus à droite du nuage de points.
6. Écrire une requête SQL qui donne les coordonnées du point le plus éloigné de l'origine.

Exercice 10. Expliquer succintement le schéma d'Euler explicite.

Écrire une fonction python `euler(F, y0, temps)` qui applique le schéma d'Euler explicite à l'équation différentielle d'ordre 1 : $y' = F(y, t)$ aux temps $t_0, t_1, \ldots t_{n-1}$ (liste `temps`) avec $y(t_0) = y_0$

Écrire la fonction python `F(y, t)` qui correspond à l'équation différentielle $y''(t) - 3y'(t) + 4t = 5$

Exercice 11. [Exponentiation rapide]

```
def expo_rapide(x, n)
    resultat, p, m = 1, x, n
    while m > 0:
        if m % 2 == 0:
            resultat, p, m = resultat, p * p, m // 2
        else:
            resultat, p, m = resultat * p, p * p, (m - 1) // 2
    return resultat
```

Montrer que l'égalité $x^n = resultat \times p^m$ est un invariant de boucle (pour la seule boucle $while$ du programme), c'est à dire qu'elle est vraie avant la première itération, et que si elle est vraie au début d'une itération alors elle est aussi vraie à la fin. On pourra noter $resultat, p, m$ les variables en début d'itération et $résultat', p', m'$ les variables en fin d'itération. Conclure.

Quelle est la complexité de cette fonction ?

Exercice 12. [Projet Euler 12] The sequence of triangle numbers is generated by adding the natural numbers. So the 7th triangle number would be 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28. The first ten terms would be:

1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...

Let us list the *factors* of the first seven triangle numbers:

```
 1: 1
 3: 1,3
 6: 1,2,3,6
10: 1,2,5,10
15: 1,3,5,15
21: 1,3,7,21
28: 1,2,4,7,14,28
```

We can see that 28 is the first triangle number to have over five divisors.

What is the value of the first triangle number to have over five hundred divisors?

Résoudre ce problème à l'aide de python en expliquant les étapes.

Exercice 13. Écrire une fonction permettant de tester si deux piles sont *égales* (mêmes éléments dans le même ordre). On supposera qu'on sait manipuler les piles avec les fonctions `creer_pile()`, `est_vide(p)`, `empiler(p, x)`, `depiler(p)`, `sommet(p)`.

Exercice 14. Écrire une fonction `partage(p)` prenant en entrée une pile p et renvoyant une pile contenant les mêmes éléments, mais où tous les éléments strictement positifs sont "au dessus" de tous les éléments négatifs. On ne demande pas de laisser p intacte.

Exercice 15. [Project Euler 5] 2520 is the smallest number that can be divided by each of the numbers from 1 to 10 without any remainder.

What is the smallest positive number that is evenly divisible by all of the numbers from 1 to 20?

Écrire un programme python qui permet de résoudre ce problème.

Exercice 16. Reprendre l'exercice sur les nombres triangulaires (Project Euler 12) en remarquant que $\frac{n(n+1)}{2}$ est le produit de deux nombres premiers entre eux (distinguer les cas)

Exercice 17. Reprendre tous les exercices précédents en Scilab ;-)

DS 01 - Algorithmes / Récursivité

Exercice 1

```
def factorielle_rec(n):
    if n == 0: # cas de base
        return 1
    else:
        return n * factorielle_rec(n - 1)

def factorielle_it(n):
    produit = 1
    for i in range(2, n + 1): # attention aux bornes : de 2 à n inclus
        produit *= i
    return produit
```

Ces deux fonctions sont de complexité $\Theta(n)$. Avec la version récursive, il peut se produire un dépassement de la pile d'exécution (n > 1000 environ)

Exercice 2

```
def pgcd(a, b):
    """Algorithme d'Euclide pour calculer le pgcd de deux entiers."""
    if b == 0:
        return a
    else:
        return pgcd(b, a % b)

def pgcd_it(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

On peut aussi utiliser des soustractions, mais il faut faire attention à échanger `a` et `b` si `a < b`...

```
def pgcd_general(tabl):
    assert len(tabl) > 0, "On attend un tableau avec au moins un élément."
    if len(tabl) == 1:
        return tabl[0]
    else:
        return pgcd(tabl[0], pgcd_general(tab[1:])
```

La version récursive ci-dessus n'est pas terrible en termes de complexité à cause du slicing. On pourrait écrire une fonction qui prend en entrée le tableau et un indice...

```
def pgcd_aux(t, i, res):
    if i == len(t):
        return res
    return pgcd_aux(t, i + 1, pgcd(t[i], res))

def pgcd_general(t):
    return pgcd_aux(t, 0, 0)                    # On initisalise le pgcd à 0
```


ou, plus simple, une version itérative (pas demandé ici).

```
def pgcd_general(tabl):   # Version itérative, pas demandée
    p = 0
    for x in tabl:
        p = pgcd(p, x)
    return p
```

Exercice 3

```
import math
def est_premier(n):
    assert n >= 0, "On suppose n positif"
    if n <= 1: # Cas particulier : à vérifier !
        return False
    for d in range(2, int(math.sqrt(n)) + 1):
        if n % d == 0:
            return False
    return True
```

- On renvoie un booléen (True ou False)
- On peut se limiter à rechercher des diviseurs potentiels inférieurs (au sens large) à sqrt(n)
- Dès qu'on trouve un diviseur de n, on peut renvoyer directement False

Exercice 4

Les listes sont déjà triées dans l'ordre croissant. Remarque : il suffit de comparer le premier élément de chaque liste :

- si ils sont égaux, cette valeur fait partie de l'intersection
- sinon on supprime le plus petit de deux et on recommence
- il ne faut pas supprimer l'élément dans les deux listes !
- Ici le fait d'utiliser le slicing fait exploser la complexité : cf. exercice sur la dichotomie.

```
def fusion(a, b):
    if a == [] or b == []:
        return []
    elif a[0] < b[0]:
        return [a[0]] + fusion(a[1:], b)
    else:
        return [b[0]] + fusion(a, b[1:])
```

Idem, ici la complexité est en $O(|a|^2 + |b|^2)$. On peut faire mieux : cf. la "fusion" itérative du tri fusion vue en cours.

Exercice 5

```
def trigo(x, y):
    assert not (x == 0 and y == 0)
    return x / sqrt(x ** 2 + y ** 2)
```

(Faire un schéma)

Exercice 6

```python
def dichotomie(t, x):
    if len(t) == 0: return False
    m = len(t) // 2
    if x == t[m]:
        return True
    elif x < t[m]:
        return dichotomie(t[:m], x) # jusqu'à m exclu
    else:
        return dichotomie(t[m + 1:], x) # à partir de m exclu
```

Cette version fonctionne bien, mais le slicing est très couteux : pour un tableau de taille $n = 2^k$, on recopie : $2^{k-1}+2^{k-2}+...+2^0 = 2^k - 1 = O(n)$ éléments. Donc la complexité de cette fonction est en $O(n)$ contre $O(ln(n))$ attendu :

```python
def dicho(x, t, i, j): # oui, vous pouvez l'appler 'aux' :s
    """Recherche dichotomique de x dans t
    entre l'indice i inclus et j exclu (convention)"""
    if i == j:
        return False
    m = (i + j) // 2
    if x == t[m]:
        return True
    elif x < t[m]:
        return dicho(x, t, i, m)
    else:
        return dicho(x, t, m + 1, j)

def dichotomie(t, x):
    return dicho(t, x, 0, len(t)) # jusqu'à len(t) exclu
```

On ne recopie jamais t. Chaque appel à dichotomie fait un nombre borné d'opérations, plus un appel récursif. pour $n = 2^k$, on a $C(n) = 1 + C(\frac{n}{2})$ d'ou $C(n) = O(ln(n))$

Exercice 7

```python
import matplotlib.pyplot as plt, numpy as np

def f(x):
    return (x ** 3 + 1) / x

LX = (np.linspace(0, 1, 100))
LY = [f(x) for x in LX]          # plus simple que d'autred méthodes
plt.plot(LX, LY) # à connaitre !
plt.show()
```

Exercice 8

```python
import random

def alea(n):
    return int(random.random() * n) + 1     # n valeurs possibles

k = int(input("Entrez un entier k :"))      # à connaître !
print(alea(k))
```


Exercice 9

C'est un identifiant qui permet de référencer chaque ligne de manière unique : c'est une *clé primaire*.

```sql
SELECT MAX(x), MIN(y) FROM base

-- C'est important de garder le prénom !
SELECT COUNT(*), prenom FROM base GROUP BY prenom

SELECT AVG(x), AVG(y) FROM base

-- On peut aussi utiliser HAVING
SELECT prenom FROM base WHERE x = (SELECT MAX(x) FROM base)

-- Même principe, c'est un peu galère...
SELECT x, y, x * x + y * y AS dist_carre FROM base WHERE dist_carre = (SELECT MAX(x * x + y * y) FROM base)
```

Exercice 10

Le schéma d'Euler explicite permet de résoudre numériquement une équation différentielle donnée par $y' = F(y,t)$ en calculant des valeurs y_k aux temps t_k de proche en proche en utilisant l'approximation : $y_{k+1} = y_k + (t_{k+1} - t_k) \times F(y_k, t_k)$ (c'est ça le schéma d'Euler explicite)

```python
def euler(F, y0, temps):
    LY = [y_0]
    for i in range(1, len(temps)): # on ne prend pas le premier élément
        LY.append(LY[-1] + (temps[i] - temps[i - 1]) * F(LY[-1], temps[i - 1]))
    return LY
```

Attention : pour que ça fonctionne à un ordre supérieur à 1, il faut que, dans `LY.append(... + ... * ...)`, le produit et la somme se fassent bien terme à terme... par exemple en utilisant numpy

Après, il faut définir la fonction $F(Y,t)$ pour que ça corresponde au problème à résoudre.

```python
def F(Y, t):                # y''= 3*y' - 4*t + 5
    y, yp = Y
    return (yp, 3*yp-4*t+5)
```

Exercice 11

Avant la première itération, on a : $resulat \times p^m = 1 \times x^n = x^n$

Si on suppose qu'au début d'une itération on bien $resultat \times p^m = x^n$, alors à la fin de l'itération, on a :

- Si m est pair : $resultat' \times p'^{m'} = resultat \times (p * p)^{m/2} = resultat \times p^m = x^n$
- Si m est impair : $resultat' \times p'^{m'} = (resultat * p) \times (p * p)^{(m-1)/2} = resultat \times p^m = x^n$

On peut en déduire qu'à la fin de la boucle (i.e. pour $m = 0$), on a $resultat = resultat \times p^0 = x^n$

Exercice 12

On peut utiliser une méthode bruteforce en calculant tous les nombres triangulaires et pour chacun son nombre de diviseurs par exemple avec :

```python
def nbdiv(n):
    if n == 1:
        return 1
    nb, racine = 0, int(math.sqrt(n))
    for d in range(racine + 1):
        if n % d == 0:
            nb += 2       # on a trouvé 2 diviseurs : d et (n // d)
    if racine * racine = n:
            nb -= 1 # Si c'est un carré parfait, on a compté deux fois la racine
    return nb
```

ou faire mieux (Ex 16)

Exercice 13

```python
def egalite(pile1, pile2):
    while not est_vide(pile1) and not est_vide(pile2):
        if depiler(pile1) != depiler(pile2):
            return False
    return est_vide(pile1) and est_vide(pile2)
```

Exercice 14

```python
def partage(pile):
    positif = creer_pile()
    negatif = creer_pile()
    while not est_vide(pile):
        element = depiler(pile)
        if element > 0:
            empiler(positif, element)
        else:
            empile(negatif, element)
    resultat = creer_pile()
    while not est_vide(negatif):
        empiler(resultat, depiler(negatif))
    while not est_vide(positif):
        empiler(resultat, depiler(positif))
    return resultat
```

Exercice 15

```python
def euler15(): # Plusieurs méthodes, en particulier on peut calculer le ppcm...
    p = 1
    for i in range(1, 20 + 1):
        p *= i // pgcd(p, i)
    return p
```

Exercice 16

Selon la parité de n, comme (n et $\frac{n+1}{2}$) ou ($\frac{n}{2}$ et $n+1$) sont premiers entre eux, on peut faire le produit de leur nombre de diviseurs respectifs en utilisant la fonction `nbdiv` précédente. C'est *beaucoup* plus rapide que la méthode directe !