Part 1

Fonctions pair/ impair

1.4. Script impair

```
def impair(N):
    if N==0 :
        return False
    elif N==1:
        return True
    else :
        return impair(N-2)
```

Part 2

PGCD Euclide

Traité à la page : https://allophysique.com/docs/nsi/langages/page2/#application-recherche-du-pgcd

Exemple de tracé de la fonction :

division de A	par B	q	r
264	228	1	36
228	36		12
36	12	3	0

Part 3

Fonction Mc Carthy

Script de la fonction Mc Carthy :

```
def f(n):
    """fonction de Mc Carthy
    retourne 91 quel que soit n<102
    """
    if n > 100:
        return n-10
    else:
        return f(f(n+11))
```

Tracé de f(99) : voir la page : https://fr.wikipedia.org/wiki/Fonction_91_de_McCarthy

```
f(99)  = f(f(110))  car 99   100
       = f(100)     car 110 > 100
       = f(f(111))  car 100   100
       = f(101)     car 111 > 100
       = 91         car 101 > 100
```

Part 4

Fonction x + 1/x

Script de l'algorithme 1 :

```
def u_rec(n):
    """calcule le terme de rang n de la suite
    recurente : u_n+1 = u_n + 1/u_n
    Param:
    n: int
    Return:
    float
    """
    if n == 0: return 1
    else:
        return u_rec(n-1) + 1/u_rec(n-1)
```

Avec les conventions de comptage des opérations, on a la relation de recurence sur T(n) :

$$T(n) = 5 + 2 \times T(n-1)$$

Ce qui fait une complexité en $O(2^n)$. Cet algorithme n'est pas très efficace...

Script de l'algorithme 2 :

```
def u_rec(n):
    """calcule le terme de rang n de la suite
    recurente : u_n+1 = u_n + 1/u_n
    avec une variable locale x
    Param:
    n: int
    Return:
    float
    """
    if n == 0: return 1
    else:
            x = u_rec(n-1)
        return x + 1/x
```

Ici :

$$T(n) = 4 + T(n-1)$$

Ce qui fait cette fois une complexité en O(n). C'est bien plus efficace.

Part 5

Exponentiation

1. Il faut n produits : x * x * x ... * 1
2. On a : $u_{n+1} = u_n + 1$
3. T(n+1) = T(n) + 1 donc une complexité asymptotique O(n)

Part 6

Hanoi

Calcul du nombre de déplacements Pour 2 disques, on a :

```
deplacement de 1 vers 2
deplacement de 1 vers 3
deplacement de 2 vers 3
```

Donc 3 deplacements.

Pour 3 disques (exemple proposé), on a 7 deplacements.

Pour 5 disques, on a 15 deplacements. (à tester avec la page de programmation des exercices sur la recursivité).

Loi de recurence

On a visiblement :

$$T(n) = 1 + 2 \times T(n-1)$$

Verfication pour l'algorithme recursif de Hanoï On prendra comme opérations significatives, les comparaisons de N avec 1.

Ligne 16 : 1 comparaison (1 unité)

Ligne 19 : appel de hanoi(N-1,d,a,i) => T(n-1) unités

Ligne 20 : appel de hanoi(1,d,i,a) => 1 unité T(1) vaut 1

Ligne 21 : appel de hanoi(N-1,i,d,a) => T(n-1) unités

Total :

$$T(n) = 1 + 2 \times T(n-1)$$

Part 7

Correction des travaux pratiques en ligne

Exercice 1 : Impair

```python
def impair(N):
    if N==0 :
        return False
    elif N==1:
        return True
    else :
        return impair(N-2)
```

Exercice 2 : Suite de Fibonacci

```python
def fibo(n):
    """
    algo recursif
    """
    if (n==0) : return 0
    if (n==1) : return 1
    return fibo(n-1) + fibo(n-2)

for i in range(11):
    print('rang {} => {}'.format(i,fibo(i)))
```

Exercice 4 : nombre d'occurences

```
def nombre_r(lettre, phrase):
    """nombre d_occurences d_une lettre
    dans une phrase
    :Params:
    lettre: str, un caractere a trouver
    phrase: str, une chaine de caracteres
    :Returns:
    au final, la fonction renvoie un entier
    :Exemple:
    >>> nombre_r('u','lustucru')
    3
    """
    if len(phrase)==0:
        return 0
    elif phrase[0] == lettre:
        return 1 + nombre_r(lettre,phrase[1:])
    else:
        return nombre_r(lettre,phrase[1:])
```

Exercice 5 : Retournement de liste

```
def reverse_r(seq,i=0):
    """algo recursif de retournement de chaine
    """
    seq=list(seq)
    a=i
    b=len(seq)-i-1
    if b-a <=0:
        return seq
    else:
        seq[a], seq[b] = seq[b],seq[a]
        return reverse_r(seq,i+1)
>>> reverse_r('lustucrus')
['s', 'u', 'r', 'c', 'u', 't', 's', 'u', 'l']
```

autre script :

```
def reverse_r(s):
    """algo recursif de retournement de chaine
    """
    if len(s) <= 1:
        return s
    m = len(s) - 1
    return s[m] + reverse_r(s[1:m]) + s[0]
```

La condition de base : On traite le retournement des caractères aux extremités d'une chaine dont la taille diminue au fur et à mesure des appels recursifs. * si la longueur de chaine est un nombre impair : Au moment où la longueur de chaine est égale à 1 : Le dernier caractère non retourné est au milieu de la chaine. On renvoie ce dernier caractère : `return s` * si la longueur de chaine est un nombre pair : Au moment où la longueur de chaine est nulle, on termine l'execution de la fonction en renvoyant s qui vaut `''`.

dans tous les cas, on pourra exprimer cette condition de base avec :

```
 if len(s) <= 1:
        return s
```

L'hérédité : On cherche un traitement valable à chaque profondeur de la pile d'appels recursifs. Ici, on veut echanger la place du premier caractère, s[0], avec le dernier, s[-1]. Et concatener ces 2 caractères avec le traitement recursif de la chaine restante s[1 :-1] :

```
return s[0] + s[1:-1] + s[-1]
```

Part 8

Compléments de cours sur l'hérédité

8.1 Le problème des permutations

Une permutation d'objets distincts rangés dans un certain ordre correspond à un changement de l'ordre de succession de ces objets. https://fr.wikipedia.org/wiki/Permutation

Comment construire les permutations de `abcd` ? Il faut d'abord reserver `a` en première position. Puis réaliser **toutes** les permutations sur la chaine `bcd`. Comment fait-on cela ? Il faut d'abord reserver `b` puis réaliser **toutes** les permutations sur la chaine `cd`... On voit clairement apparaitre l'aspect recursif de cet algorithme.

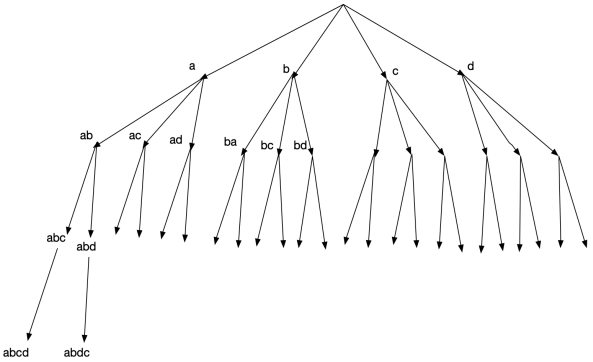

FIGURE 1 – permutations

```
def permut(mot):
    if len(mot) == 1: return mot

    L = []
    for i in range(len(mot)):
        for x in permut(mot[0:i] + mot[i + 1:]):
            L.append(mot[i] + x)
    return L
```

- au 1er appel de la fonction : il faut reserver chacune des lettres du mot comme premiere lettre. On commence par la premiere, "a".
- Il faut ensuite associer "a" avec TOUS les mots issus de la permutation de "bcd". D'où les 3 lignes de la partie *hérédité* du script :

```
    for i in range(len(mot)):
        for x in permut(mot[0:i] + mot[i + 1:]):
            L.append(mot[i] + x)
```