NSI *Première* -> Terminale

Le programme de terminale est très chargé avec de nombreuses notions à voir.

Nous vous proposons donc de reprendre et chercher quelques exercices de programmation en Python (fin août), afin d'être opérationnel dès la reprise.

- ➔ La correction de quelques exercices sera donnée à partir du lundi 19 août 2024 sur le site du lycée
- ➔ D'autres seront corrigés lors des premières séances de NSI
- ➔ **Les exercices 10 et 11** sont à rendre lors de la première séance de NSI

Ces notions seront évaluées, notamment à l'occasion d'une épreuve pratique ou questions flash dès la semaine de la rentrée.

Bonnes vacances à toutes et à tous !

Exercice 1

Objectif : *print | input | boucle | spécification, assertion*

1. Pour aider le fils de votre voisin qui entre en CE1, compléter le script ci-dessous.

```
def table(n):
    # à compléter

n = input("Saisir un entier compris entre 1 et 10")
n = ...
table(n)
```

La fonction table prend en paramètre un entier n compris entre 1 et 10 (compris) et affiche en console la table de multiplication de n.

➔ Ne pas oublier la docstring et d'éventuelles assertions

Exemple : si l'utilisateur saisit 6, on obtient :

```
>>>
Table de 6
6 * 0 = 0
6 * 1 = 6
6 * 2 = 12
6 * 3 = 18
6 * 4 = 24
6 * 5 = 30
6 * 6 = 36
6 * 7 = 42
6 * 8 = 48
6 * 9 = 54
6 * 10 = 60
```

Un clavier après quelques semaines de vacances

2. Finalement, le petit voisin a besoin de toutes les tables de multiplication de 1 jusqu'à 10. Modifier le script.

3. Apprendre par cœur les tables de multiplication 😊

Exercice 2

Objectif : *boucle | spécification, assertion | jeu de tests*

Écrire la fonction nb_de_div_par_2 qui prend en paramètre un entier non nul et qui renvoie combien de fois de suite cet entier est divisible par 2.

➔ On créera la docstring, des assertions en précondition et postcondition et un jeu de tests.

Exemples

```
>>> nb_de_div_par_2(24)
3
>>> nb_de_div_par_2(5)
0
```

Exercice 3

Objectif : *chaîne de caractères | Recherche d'un élément*

Il s'agit d'écrire un programme qui demande de saisir une chaîne d'ADN valide et une séquence d'ADN valide (*valide* signifie qu'elles ne sont pas vides et sont formées exclusivement d'une combinaison arbitraire de "a", "t", "g" ou "c") et de dénombrer le nombre de séquences dans la chaîne.

▪ Écrire la fonction valide qui prend en paramètre une chaine de caractère et qui renvoie True si la saisie est valide, False sinon.

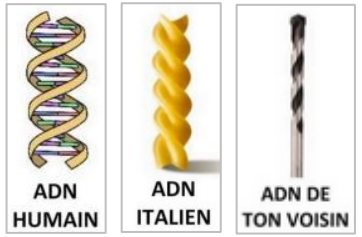
Exemples

```
>>> valide("agtcc")
True
>>> valide("accbtg")
False
```

▪ Écrire la fonction nombre qui reçoit deux paramètres, la *chaîne* et la *séquence* et qui retourne le nombre de *séquence* dans la *chaîne* [ne pas oublier la documentation et les assertions].

Exemple :

```
>>> chaine = "attgcaatcgtggtacatgc"
>>> sequence = "ca"
>>> nombre(chaine, sequence)
2
```



Exercice 4

Objectif : *listes | pop | append*

1. Préliminaire. On rappelle que :

- l'expression `T1 = list(T)` fait une copie de T indépendante de T,
- l'expression `x = T.pop()` enlève le sommet de la liste T et le place dans la variable x
- l'expression `T.append(v)` place la valeur v au sommet de la liste T.

Tester, éventuellement, en console les instructions suivantes ➡

Console

```
>>> T = [1, 2, 3]
>>> T2 = list(T)
>>> x = T2.pop()
>>> x
...
>>> T2.append(4)
>>> T2
...
>>> T
...
```

2. Compléter le code Python de la fonction positif ci-dessous ⬇ qui prend une liste T de nombres entiers en paramètre et qui renvoie la liste des entiers positifs dans le même ordre, sans modifier la variable T.

```
def positif(T):
    T2 = ...(T)      # copie de T
    T3 = ...         # liste vide
    while T2 != []:
        x = ...      # on retire le dernier élément de T2
        if ... >= 0:
            T3.append(...)
    # on replace les éléments de T3 dans l'ordre dans T2
    while T3 != ... :
        x = T3.pop()
        ...
    print('T =', T) # on vérifie que T est inchangée
    return T2
```

Exemple

```
>>> positif([-1, 0, 5, -3, 4, -6, 10, 9, -8])
T = [-1, 0, 5, -3, 4, -6, 10, 9, -8]
[0, 5, 4, 10, 9]
```

Exercice 5

Objectif : *dictionnaires*

Partie 1

On considère le dictionnaire suivant :

```
mondict = {"device": "laptop" , "constructeur": "acer" , "ram": "8G" ,
           "processeur": "Intel core i5", "stockage": "1 T"}
```

Écrire les inscriptions permettant :

- de corriger l'erreur `"stockage": "2 T"`
- d'afficher ➡ la liste des clés,
  ➡ la liste des valeurs
  ➡ la liste des paires de clés et valeurs
- d'ajouter la pair clé-valeur : `"Système d'exploitation" : "Windows 11"`
- à l'aide d'une boucle, d'afficher l'ensemble des couples.

```
*** Console ***
>>>
processeur : Intel core i5
constructeur : acer
device : laptop
stockage : 2 T
Système d'exploitation : windows 11
ram : 8G
>>>
```

Partie 2

Écrire une fonction Python qui prend en paramètre un entier n et qui renvoie un dictionnaire formé des entiers de 1 à n et de leurs carrées.

Exemple

Pour n = 7 le dictionnaire sera : `{1: 1, 2: 4, 3: 9, 4: 16, 5: 25 , 6: 36 , 7: 49}`

Exercice 6
[Finir le sujet 25, exercice 2]

EXERCICE 2
On considère la fonction `separe` ci-dessous qui prend en argument un tableau `tab` dont les éléments sont des 0 et des 1 et qui sépare les 0 des 1 en plaçant les 0 en début de tableau et les 1 à la suite.

```
def separe(tab):
    '''Separe les 0 et les 1 dans le tableau tab'''
    gauche = 0
    droite = ...
    while gauche < droite:
        if tab[gauche] == 0 :
            gauche = ...
        else :
            tab[gauche] = ...
            tab[droite] = ...
            droite = ...
    return tab
```

Compléter la fonction `separe` ci-dessus.

Exemples

```
>>> separe([1, 0, 1, 0, 1, 0, 1, 0])
[0, 0, 0, 0, 1, 1, 1, 1]
>>> separe([1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0])
[0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Description d'étapes effectuées par la fonction `separe` sur le tableau ci-dessous, les caractères ^ indiquent les cases pointées par les indices gauche et droite :

```
tab = [1, 0, 1, 0, 1, 0, 1, 0]
       ^                    ^
```

• Étape 1 : on regarde la première case, qui contient un 1 : ce 1 va aller dans la seconde partie du tableau final et on l'échange avec la dernière case. Il est à présent bien positionné : on ne prend plus la dernière case en compte.

```
tab = [0, 0, 1, 0, 1, 0, 1, 1]
       ^                 ^
```

• Étape 2 : on regarde à nouveau la première case, qui contient maintenant un 0 : ce 0 va aller dans la première partie du tableau final et est bien positionné : on ne prend plus la première case en compte.

```
tab = [0, 0, 1, 0, 1, 0, 1, 1]
          ^              ^
```

• Étape 3 : on regarde la seconde case, qui contient un 0 : ce 0 va aller dans la première partie du tableau final et est bien positionné : on ne prend plus la seconde case en compte.

```
tab = [0, 0, 1, 0, 1, 0, 1, 1]
             ^           ^
```

• Étape 4 : on regarde la troisième case, qui contient un 1 : ce 1 va aller dans la seconde partie du tableau final et on l'échange avec l'avant-dernière case. Il est à présent bien positionné : on ne prend plus l'avant-dernière case en compte.

```
tab = [0, 0, 1, 0, 1, 0, 1, 1]
             ^        ^
```

Et ainsi de suite…

```
tab = [0, 0, 0, 0, 1, 1, 1, 1]
```

Exercice 7

Objectif : *écriture binaire / décimale d'un entier*

1. On modélise la représentation binaire d'un entier non signé par un tableau d'entiers dont les éléments sont 0 ou 1.

Par exemple, le tableau `[1, 0, 1, 0, 0, 1, 1]` représente l'écriture binaire de l'entier 1010011_2 dont l'écriture décimale est `2**6 + 2**4 + 2**1 + 2**0 = 83`.

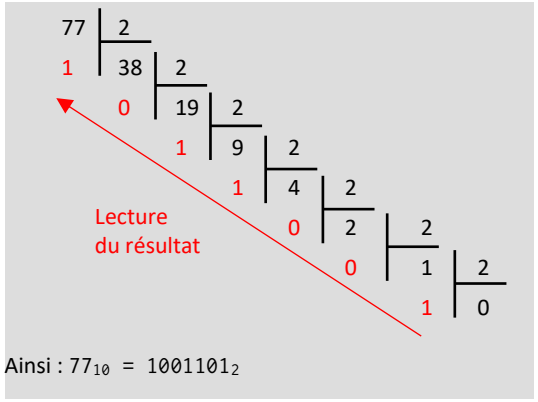
À l'aide d'un parcours séquentiel, écrire la fonction `convertir` répondant aux spécifications suivantes ⬇

```
def convertir(T):
    """T est un tableau d'entiers, dont les éléments sont 0 ou 1 et
    représentant un entier écrit en binaire.
    Renvoie l'écriture décimale de l'entier positif dont la représentation
    binaire est donnée par le tableau T"""
```

Exemple

```
>>> convertir([1, 0, 1, 0, 0, 1, 1])
83
>>> convertir([1, 0, 0, 0, 0, 0, 1, 0])
130
```

2. Pour rappel, la conversion d'un nombre entier positif en binaire peut s'effectuer à l'aide des divisions successives comme illustré ici ⬇



Ainsi : $77_{10} = 1001101_2$

```
def binaire(a):
    bin_a = [...]
    a = a//2
    while a ... :
        bin_a = ... + ...
        a = ...
    return bin_a
```

Voici une fonction python basée sur la méthode des divisions successives ⬆ permettant de convertir un nombre entier positif en binaire.

Compléter la fonction `binaire`.

Exemple

```
>>> binaire(77)
[1, 0, 0, 1, 1, 0, 1]
```

Exercice 8

■ Le problème

Pour simplifier, on suppose qu'on ne dispose que de pièces (en quantité illimitée).
On considère donc un système de pièces sous la forme d'un n-uplet :

$p = (p_0, p_1, \dots p_{n-1})$ où p_i représente la valeur de la pièce d'indice i

Les valeurs sont classées par ordre croissant.
Remarque : si $p_0 > 1$, alors certaines sommes ne pourront pas être rendues.

Il s'agit de trouver une liste d'entiers positifs $[x_0, x_1, \dots, x_{n-1}]$ qui vérifie

$x_0p_0 + x_1p_1 + \cdots + x_{n-1}p_{n-1} = r$ où r est la somme à rendre

en **minimisant** le nombre de pièces utilisé

Exemple 1 avec le système : $p = (2, 5, 10)$ -> $p_0 = 2, p_1 = 5, p_2 = 10$

- $r = 3$: on ne peut pas rendre cette somme
- $r = 12$:
  - ➔ $12 = 10 + 2$ ($x_0 = 1, x_1 = 0, x_2 = 1$)-> 2 pièces
  - ➔ $12 = 5 + 5 + 2 = 5 \times 2 + 2$ ($x_0 = 1, x_1 = 2, x_2 = 0$) -> 3 pièces
  - ➔ $12 = 2 + 2 + 2 + 2 + 2 + 2 = 2 \times 6$ ($x_0 = 6, x_1 = 0, x_2 = 0$) -> 6 pièces

Pour 12, **la solution** est 2 pièces (une pièce de 10 et une pièce de 2).

■ Première solution : par **force brute** -> tester de manière exhaustive toutes les éventualités

Exemple 2 avec le système : $p = (1, 2, 5)$ et $r = 8$

La fct `liste_solution` permet de déterminer tous les listes $[x_0, x_1, x_2]$ tel que : $x_0p_0 + x_1p_1 + x_2p_2 = 8$

Compléter le programme.

```
1  def liste_solutions(r):
2      liste = ...
3      for i in range(9):
4          for j in range(5):
5              for k in range(2):
6                  if ... :
7                      liste.append(...)
8      return liste
```

```
*** Console ***
>>> liste_solutions(8)
[[0, 4, 0], [1, 1, 1], [2, 3, 0], [3, 0, 1], [4, 2, 0], [6, 1, 0], [8, 0, 0]]
```

Question : Combien de fois est exécuter la ligne 6 ?

Conclusion : ce type d'algorithme est simple, mais souvent inutilisable en raison de son coût.

■ Algorithme glouton

Un algorithme glouton permet de résoudre un problème d'optimisation : on obtient souvent une solution intéressante mais pas toujours la meilleure.

Un choix optimal peut être globalement optimal, c'est le meilleur de tous, ou localement optimal, c'est le meilleur parmi un ensemble restreint de choix.

À chaque étape exécutée par un algorithme, se présente un ensemble de choix et un algorithme glouton fait le meilleur choix parmi les propositions. Un choix glouton est donc un choix localement optimal. La question est de savoir si en faisant une série de choix localement optimaux, on finit par aboutir à une solution optimale.

C'est parfois le cas mais pas toujours.

Appliquons l'algorithme glouton à la situation du rendu de monnaie :

- On cherche, par valeur décroissante en partant de la pièce qui a la plus forte valeur, la première pièce qui a une valeur inférieure ou égale à la somme à rendre r.
- On prend cette pièce, on retranche sa valeur v à r.
- On recommence en partant de la pièce prise en cherchant celle qui a une valeur inférieure ou égale à la nouvelle somme à rendre $r - v$. On prend cette pièce, et ainsi de suite jusqu'à arriver à une somme à rendre nulle (si possible).

➔ Les entrées sont p pour le système pièces et r pour la somme à rendre.
➔ La sortie est la liste $[x_0, x_1, \dots, x_{n-1}]$

Exemple 3

```
>>> p = [1, 2, 5, 10]
>>> r = 23
[1, 1, 0, 2]
```

En effet :
23 -> on peut utiliser une pièce de 10 -> 23−10=13
13 -> on peut utiliser une autre pièce de 10 -> 13−10=3
3 -> on ne peut pas utiliser une pièce de 10
3 -> on ne peut pas utiliser une pièce de 5
3 -> on peut utiliser une pièce de 2 -> 3−1=1
1 -> on ne peut pas utiliser une pièce de 2
1 -> on peut utiliser une pièce de 1 -> 1−1=0
0 -> on ne peut pas une pièce de 1
Il n'y a plus de pièce disponible et il reste 0 à rendre
d'où la solution optimale : 2 pièces de 10, 1 pièce de 2 et une pièce de 1

Exemple 4

```
>>> p = [2, 5, 10]
>>> r = 3
n'aboutit pas
```

Compléter le programme.

```
def rendu_glouton(p, r):
    n = len(p)
    liste = [0]*n   # initialisation de [x0, x1,..., x(n-1)]
    # À compléter

p = [1, 3, 4, 6, 12]
r = 8
print(rendu_glouton(p, r))
```

Dernier exemple

```
>>> p = [1, 3, 4, 6, 12]
>>> r = 8
[2, 0, 0, 1, 0]
```

Et pourtant, le choix optimal est [0, 0, 2, 0, 0]
-> L'algorithme glouton ne donne pas toujours la solution optimale !

Exercices à rendre

Exercice 9

Objectif : *implémenter en Python un algorithme*

Connaître les quelques algorithmes du cours :

- Déterminer le minimum / maximum d'une liste
- Tri par sélection, tri par insertion
- Recherche d'un élément dans une liste -> par parcours et par dichotomie

Exercice 10

Objectif : *dictionnaire*

Le but de l'exercice consiste à écrire une fonction en Python `nb_occurrences` qui prend en paramètre une chaine de caractère `chaine`, et qui renvoie un dictionnaire dont les clés sont les caractères de la chaine et les valeurs sont les nombres d'occurrences des caractères dans la chaine.

Exemple

```
>>> chaine = "vacances"
>>> nb_occurrences(chaine)
{'n': 1, 'e': 1, 'v': 1, 'c': 2, 'a': 2, 's': 1}
```

Compléter le code de cette fonction.

```
def nb_occurrences(chaine):
    dico = ...
    for caractere in ... :
        if dico.get(caractere) != ...
            dico[caractere] = ...
        else:
            ...
    return dico
```

Exercice 11

Objectif : *Tri à bulle*

Une première approche à donner cette implémentation du tri à bulle rappelé ci-dessous auquel on a ajouté des affichages (`print`) pour visualiser l'évolution en console.

```
1  def tri_bulles(T):
2      n = len(T)
3      for i in range(n-1):
4          for j in range(n-1-i):
5              if T[j] > T[j+1]:
6                  T[j], T[j+1] = T[j+1], T[j]
7              print(T)
8          print(" Fin passage", i+1, ":", T)
9
10 T = [3, 7, 1, 2, 5, 4, 6]
11
12 tri_bulles(T)
13 print(T)
```

```
*** Console ***
>>>
[3, 7, 1, 2, 5, 4, 6]
[3, 1, 7, 2, 5, 4, 6]
[3, 1, 2, 7, 5, 4, 6]
[3, 1, 2, 5, 7, 4, 6]
[3, 1, 2, 5, 4, 7, 6]
[3, 1, 2, 5, 4, 6, 7]
Fin passage 1 : [3, 1, 2, 5, 4, 6, 7]
[1, 3, 2, 5, 4, 6, 7]
[1, 2, 3, 5, 4, 6, 7]
[1, 2, 3, 5, 4, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
Fin passage 2 : [1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
Fin passage 3 : [1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
Fin passage 4 : [1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
Fin passage 5 : [1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
Fin passage 6 : [1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
>>>
```

On remarque qu'à la fin du 2ème passage, la liste est triée :

↳ on pourrait donc s'épargner les 4, 5 et 6ème passages.

■ Proposer une amélioration du programme qui permettrait de s'arrêter lorsque la liste est triée.

Indication : on utilisera une variable booléenne qui change de valeur s'il n'y a pas de permutations (ligne 6).