+1/1/60+

Bases de l'informatique 1 — SPUF100

Année 2024-2025 — Seconde session

Nom : .....

Prénom : .....

Numéro d'étudiant : .....

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6 ☐7 ☐8 ☐9
☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6 ☐7 ☐8 ☐9
☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6 ☐7 ☐8 ☐9

Durée : 2 heures.

Aucun document n'est autorisé. L'usage de la calculatrice ou de tout autre appareil électronique est interdit.

Les exercices sont indépendants. Au sein d'un même exercice, vous pouvez utiliser les variables et fonctions des questions précédentes, même si vous n'avez pas su les faire ; chaque question est donc indépendante.

À part les méthodes et fonctions de base, vous n'avez pas le droit d'utiliser les fonctions et les méthodes « avancées », sauf si l'énoncé vous conseille l'utilisation de certaines d'entre elles.

```
# Fonctions autorisées
range(…) len(…) print(…)

# Boucles autorisées
Boucles while
Boucles for avec un range
Boucles for sans range (for x in L)

# Vous pouvez utiliser les compréhensions ou les slices
[ x for x in range(L) ]
chaine[début:fin:pas]

# Les méthodes suivantes sont autorisées :
L.append(…)
```

```
# Par exemple les méthodes et fonctions suivantes sont entre autres interdites
max(…) min(…) sum(…) abs(…)
s.split(…) s.index(…) L.extend(…)
```

Exercice 1 Boucles while . . . . . 4 points

☐0 ☐0,5 ☐1 ☐1,5 ☐2 ☐2,5 ☐3 ☐3,5 ☐4

1. Écrire une fonction `sr(n)` qui renvoie le plus petit nombre entier k vérifiant $n < \sqrt{1} + \sqrt{2} + ... + \sqrt{k}$. On utilisera une boucle `while` et la fonction `sqrt` de la biblothèque `math`. C'est à vous de faire l'import.

```
>>> sqrt(1) + sqrt(2) +  sqrt(3) + sqrt(4) +  sqrt(5)
8.382332347441762
>>> sqrt(1) + sqrt(2) +  sqrt(3) + sqrt(4) +  sqrt(5) + sqrt(6)
10.83182209022494
>>> sr(10)
6
```

```
from math import sqrt


def sr(n):
    somme = 0
    i = 0
    while somme <= n:
        i = i+1
        somme = somme + sqrt(i)
    return i
```

2. Écrire une fonction `intervalle(a,b)` qui à partir de deux entiers `a` $\leq$ `b` renvoie le tableau `[a, …, b]`. Il est évidemment interdit d'utiliser la fonction `range` et il faudra créer le tableau de la bonne taille avant de le remplir (pas de append ou de concatenation).

```
>>> intervalle(5,10)
[5, 6, 7, 8, 9, 10]
>>> intervalle(2019,2025)
[2019, 2020, 2021, 2022, 2023, 2024, 2025]
```

```


def intervalle(a,b):
    taille = (b-a) + 1
    tableau = taille * [0]
    i = 0
    while i+a<b:
        tableau[i] = i+a
        i = i+1
    return tableau
```

Exercice 2 Calculs binaires . . . . . 4,5 points

☐0 ☐0,5 ☐1 ☐1,5 ☐2 ☐2,5 ☐3 ☐3,5 ☐4 ☐4,5

1. En utilisant l'algorithme vu en cours, écrire 75 en binaire.

```
75 = 2 × 37 + 1
37 = 2 × 18 + 1 # J'enleve des points  pour ceux
18 = 2 ×  9 + 0 # utilisant une autre présentation que :
 9 = 2 ×  4 + 1 # … = 2 × … + …
 4 = 2 ×  2 + 0
 2 = 2 ×  1 + 0
 1 = 2 ×  0 + 1
75 est donc représenté par 100 1011 en binaire
```

2. En déduire la représentation binaire signée sur 8 bits de -75.

```
On part de 75 sur 8 bits   : 0100 1011
On prend le complément à 2 : 1011 0100
et on ajoute 1             : 1011 0101

-75 est donc représenté par 1011 0101  en binaire sur 8 bits
```

3. En utilisant l'algorithme vu en cours, donnez l'écriture décimale du nombre positif dont la représentation binaire est `101 1001`.

```
Sur 8 bits 0100 1011 commence par 0, c'est un nombre positif
1        = 1
10       = 2 ×  1 + 0 = 2
101      = 2 ×  2 + 1 = 5
101 1    = 2 ×  5 + 1 = 11
101 10   = 2 × 11 + 0 = 22
101 100  = 2 × 22 + 0 = 44
101 1001 = 2 × 44 + 1 = 89

101 1001 vaut donc 89
```

4. Donnez l'écriture décimale du nombre dont la représentation signée sur 8 bits binaire est `1111 0111`.

```
Le nombre commençant par 1 sur 8 bits, c'est un négatif
On a la représentation sur 8 bits : 1111 0111
On prend le complément à 2        : 0000 1000
et on ajoute 1                    : 0000 1001
On reconnaît 8 + 1 = 9, donc le nombre de départ est -9.
```

Exercice 3 Des points sur des courbes . . . . . 4,5 points

☐0 ☐0,5 ☐1 ☐1,5 ☐2 ☐2,5 ☐3 ☐3,5 ☐4 ☐4,5

Dans cet exercice, on appelle courbe une liste **non vide** de points `(x,y)`. Par exemple `C = [ (2,7), (10,10), (-3,-4) ]` représente une courbe de trois points. Rappel de vocabulaire : dans le couple `(x,y)`, x est l'abscisse et y l'ordonnée.

1. Une courbe est dite fonctionnelle si les différentes valeurs de x sont strictement croissantes. Écrire une fonction `est_fonctionnelle(courbe)` qui renvoie `True` si la courbe est fonctionnelle et `False` sinon.

```
>>> courbe_1 = [(2,3), (4,4), (7,-2), (8,0), (10, -33)]
>>> est_fonctionnelle(courbe_1) # car 2 < 4 < 7 < 8 < 10
True
>>> courbe_2 = [(2,3), (4,4), (7,-2), (7,0), (10, -33)]
>>> est_fonctionnelle(courbe_2) # car 7 = 7
False
>>> courbe_3 = [(2,3), (4,4), (8,-2), (7,0), (10, -33)]
>>> est_fonctionnelle(courbe_3) # car 8 < 7
False
```

```
def est_fonctionnelle(courbe):
    for i in range(1, len(courbe)):
        (x0,y0) = courbe[i-1]
        (x1,y1) = courbe[i]
        if x0 >= x1:
            return False
    return True
```

2. Écrire une fonction `maximum(courbe)` qui renvoie le point de la courbe avec la plus grande ordonnée (si plusieurs points ont la même ordonnée, on renverra le point avec le plus petit abscisse).

```
>>> maximum([(2,3), (4,4), (7,-2), (8,0), (10, -33), (11,4)])
(4, 4)
```

```
def maximum(courbe):
    p_max = courbe[0]
    for point in courbe:
        if p_max[0] < point[0]:
            p_max = point
    return p_max
```

3. Écrire une fonction `nombre_positif(courbe)` qui renvoie le nombre de fois où l'ordonnée est positive (0 est considéré comme positif).

```
>>> nombre_positif([(2,3), (4,4), (7,-2), (8,0), (10, -33)])
3
```

```
def nombre_positif(courbe):
    n=0
    for point in courbe:
        (x,y) = point
        if y>=0:
            n = n+1
    return n
```

4. Écrire une fonction `flouter(courbe)` qui construit une liste de points à partir du paramètre `courbe` en sélectionnant un point sur deux.

```
>>> flouter([(-11,1), (-8,0), (0,3), (2,-3), (5,1), (11,12)])
[(-11, 1), (0, 3), (5, 1)]
```

```
def flouter(courbe):
    c = []
    for i in range(0,len(courbe),2):
        c.append(courbe[i])
    return c
```

Exercice 4 Un peu de biologie . . . . . 7 points

☐0 ☐0,5 ☐1 ☐1,5 ☐2 ☐2,5 ☐3 ☐3,5 ☐4 ☐4,5 ☐5 ☐5,5 ☐6 ☐6,5 ☐7

Le monde du vivant n'utilise pas le langage Python, mais le langage de l'ADN pour coder des protéines. L'objectif de ce problème est de convertir une séquence d'ADN en séquence de protéines.

1. Écrire une fonction `est_adn(chaîne)` qui prend une chaîne de caractères en argument et qui renvoie `True` si la chaîne ne contient que les caractères majuscules `'A'`, `'C'`, `'G'`, `'T'` et `False` sinon.

```
>>> est_adn("gattaca")
False
>>> est_adn("GATTACA")
True
```

```
>>> est_adn("GAT C")
False
>>> est_adn("Choucroute")
False
```

```
def est_adn(chaîne):
    for lettre in chaîne:
        if lettre!='A' and lettre!='G' and lettre!='T' and lettre!='C':
            return False
    return True
```

2. En pratique, les séquences d'ADN, comme les chaussettes, vont toujours par paire. Le complémentaire d'une séquence est obtenue en remplaçant `A` par `T` et `T` par `A` ainsi que `C` par `G` et `G` par `C`. Écrire la fonction `complémentaire(adn)` qui prend une chaîne `adn` et renvoie son complémentaire.

```
>>> complémentaire("AAGGTTCCAGTC")
'TTCCAAGGTCAG'
```

```
def complémentaire(adn):
    s=''
    for e in adn:
        if e=='A':
            s = s + 'T'
        elif e=='T':
            s = s + 'A'
        elif e=='G':
            s = s + 'C'
        elif e=='C':
            s = s + 'G'
    return s
```

Un brin d'ADN est une séquence de nucléotides (A, C, T, G). Pour traduire l'ADN en protéines, on utilise le code génétique : à un codon (séquence de trois nucléotides) on associe une protéine. Chaque protéine sera représentée par une lettre. On se donne une liste `traducteur` qui donne les associations (codon, protéine) sous la forme d'un couple de chaînes de caractères. C'est le fameux code génétique.

```
>>> traducteur['CGT']
'R'
>>> traducteur['ATC']
'I'
```

```
>>> traducteur['TCT']
'S'
>>> traducteur['CCT']
'P'
```

Le dictionnaire `traducteur` est définie de manière globale. On pourra donc l'utiliser dans toutes les fonctions.

3. Écrire une fonction `traduire(codon)` qui prend en entrée une chaîne `codon` et qui renvoie la protéine associée dans le dictionnaire `traducteur`. Si la chaîne `codon` ne se trouve pas dans ce dictionnaire, la fonction renverra le caractère `'_'`.

```
>>> traduire("CGT")
'R'
>>> traduire("CFDT")
'_'
```

```
def traduire(codon):
    if codon in traducteur:
        return traducteur[codon]
    else:
        return codon
```

4. On se donne une chaîne `adn` dont la longueur est un multiple de trois. Écrire la fonction `protéines(adn)` qui renvoie la chaîne des protéines associées à chacun des codons.

```
>>> protéines("TCTCCTATCAAAGAA") # TCT CCT ATC AAA GAA
'SPIKE'
```

```
def protéines(adn):
    n=len(adn)//3
    protéine=''
    for i in range(n):
        codon = adn[3*i] + adn[3*i+1] + adn[3*i+2]
        protéine = protéine + traduire_codon(codon)
    return protéine
```

5. Certains codons ne sont associés à aucune protéine. Dans ce cas, le symbole correspondant est donné par le caractère `'_'`. Ainsi une séquence d'ADN code plusieurs séquences de protéines.

Écrire une fonction `liste_protéines(adn)` qui prend une séquence ADN et renvoie la liste des gènes associés.

```
>>> adn
'CTGACGTCAGTGAGTTAGTCTCCTATCAAAGAATAACCATATTAAAGGTCTTAGGCGCTTGAA'
>>> protéines(adn) # Écrite à la question précédente.
'LTSVS_SPIKE_PY_RS_ALE'
>>> liste_protéines(adn)
['LTSVS', 'SPIKE', 'PY', 'RS', 'ALE']
```

```
def liste_protéines(adn):
    chaîne = protéines(adn)
    prot =''
    L=[]
    for p in chaîne:
        if p=='_' and prot !='':
            L.append(prot)
            prot=''
        elif p!='_':
            prot = prot + p
    if prot != '':
        L.append(prot)
    return L
```