+1/1/60+

Bases de l'informatique 1 — SPUF100

Année 2024-2025 — Partiel

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
```

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

# Les boucles itérants directement sur les séquences sont interdites
for x in L:

# Vous n'avez pas le droit d'utiliser des compréhensions ou des slices
# À la place vous devez utiliser des boucles.
x in L
[ x for x in range(L) ]
chaine[début:fin:pas]
```

Exercice 1 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 98 en binaire.

```
98 = 2 × 49 + 0
49 = 2 × 24 + 1 # Au prochains examens, j'enleverai des points pour ceux
24 = 2 × 12 + 0 # utilisant une autre présentation que :
12 = 2 ×  6 + 0 # … = 2 × … + …
 6 = 2 ×  3 + 0
 3 = 2 ×  1 + 1
 1 = 2 ×  0 + 1
98 est donc représenté par 110 0010 en binaire
```

2. En déduire la représentation binaire signée sur 8 bits de -98.

```
On part de 98 sur 8 bits    : 0110 0010
On prend le complément à 2 : 1001 1101
et on ajoute 1              : 1001 1110

-98 est donc représenté par 1001 1110  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 100 1011.

```
Sur 8 bits 0100 1011 commence par 0, c'est un nombre positif
1        = 1
10       = 2 ×  1 + 0 = 2
100      = 2 ×  2 + 0 = 4
100 1    = 2 ×  4 + 1 = 9
100 10   = 2 ×  9 + 0 = 18
100 101  = 2 × 18 + 1 = 37
100 1011 = 2 × 36 + 1 = 75

100 1011 vaut donc 75
```

4. Donnez l'écriture décimale du nombre dont la représentation signée sur 8 bits binaire est 1011 0100.

```
Le nombre commençant par 1 sur 8 bits, c'est un négatif
On a la représentation sur 8 bits : 1011 0100
On prend le complément à 2        : 0100 1011 (tiens ? mais c'est 75 !)
et on ajoute 1                    : 0100 1100
On reconnaît 75 + 1 = 76, donc le nombre de départ est -76.
```

Exercice 2 Questions de cours . . . . . 3,5 points

☐0 ☐0,5 ☐1 ☐1,5 ☐2 ☐2,5 ☐3 ☐3,5

1. Écrire la fonction `nono(a,b,c)` qui est la négation de la fonction `toto`. On ne pourra pas utiliser l'opérateur `not` ni la structure `if`. Ainsi, `nono(a,b,c)` devra toujours être égale à `not toto(a,b,c)`

```
1 def toto(a,b,c):
2     return (a>b and  b!=c) or c+c < a
```

```
def nono(a,b,c):
    return (a<=b or b==c) and c+c >= a
```

2. Écrire une fonction `oui_non_bof(n)` qui renvoie `True` si n est divisible par deux ou par trois mais `False` s'il est divisible à la fois par deux et trois ou s'il n'est divisible par aucun des deux.

```
def oui_non_bof(n):
    if n%2==0 and n%3==0:
        return False
    elif n%2==0 or n%3==0:
        return True
    else:
        return False
```

3. Écrire une fonction `combien_de_vides(liste)` qui renvoie le nombre de chaînes vides dans une liste. Par exemple `combien_de_vides(["Salut", "", "à", "toi", ""])` vaudra 2

```
def combien_de_vides(liste):
    n=0
    for i in range(len(liste)):
        if liste[i] == "":
            n = n+1
    return n
```

Exercice 3 Devenir un vrai hacker . . . . . 3,5 points

☐0 ☐0,5 ☐1 ☐1,5 ☐2 ☐2,5 ☐3 ☐3,5

Dans les années 80/90, les adolescents avaient développé une écriture ~~débile~~ cryptique pour l'« élite » : le *leetscript*. L'idée était de remplacer certaines lettres par des symboles afin de prouver au monde que l'on était un vrai hacker ! Par exemple en *leetscript*, `"Valrose"` se traduit en `"V41r0$3"`.

On se donne un tableau de correspondance indiquant comment chiffrer un message : le couple `("e","3")`. signifiant qu'il faut remplacer les « e » par « 3 ».

```
dico = [ ("e","3"), ("x","><"), ("l", "1"), ("o", "0"), ("a", "4"), ("s","$") ]
```

1. Écrire une fonction `l3ttr3(c)`, où `c` est un caractère, qui renvoie la chaîne `s` si le couple `(c,s)` est dans la variable `dico`. Sinon, la fonction renverra `c` sans modification.

```
>>> l3ttr3("x")
'><'
>>> l3ttr3("V")
'V'
```

```
>>> l3ttr3("e")
'3'
>>> l3ttr3("k")
'k'
```

```
def l3ttr3(lettre):
    for i in range(len(dico)):
        (a,b) = dico[i]
        if  a == lettre:
            return b
    return lettre
```

2. En déduire une fonction `l33t(message)` qui traduit la chaîne de caractères `message` en *leetscript*.

```
>>> l33t("Je suis Grobelix le hacker qui fait trembler la cia! lol")
'J3 $ui$ Gr0b31i>< 13 h4ck3r qui f4it tr3mb13r 14 ci4! 101'
>>> l33t("Seule l'elite des hackers peut me comprendre... ha ha ha ha ha")
'S3u13 1'31it3 d3$ h4ck3r$ p3ut m3 c0mpr3ndr3... h4 h4 h4 h4 h4'
```

```
def l33t(message):
    kikoo = ""
    for i in range(len(message)):
        kikoo = kikoo + l3ttr3(message[i])
    return kikoo
```

Exercice 4 Déplacements sur une grille . . . . . 4,5 points

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

On représente la position d'un joueur sur un écran par un tuple de taille 2 (un couple) représentant ses coordonnées, par exemple `(2,3)`. Le point `(0,0)` étant en haut à gauche.

1. Écrire une fonction `déplacement(position, dx, dy)` qui renvoie une nouvelle position obtenue en ajoutant `dx` et `dy` aux deux coordonnées de `position`.

```
>>> pos = (10,10)
>>> déplacement(pos, 4, 7)
(14, 17)
>>> déplacement(pos, -20, 33)
(-10, 43)
```

```
def déplacement(position, dx, dy):
    (x,y) = position
    return (x+dx, y+dy)
```

2. Écrire une fonction `déplacement_restreint(position, dx, dy, taille)` qui déplace la position de la même façon que précédemment, mais cette fois-ci les coordonnées doivent rester positives et ne pas dépasser (strictement) `taille`. Si une coordonnée atteint la valeur `taille`, elle ne plus augmenter davantage, et si elle atteint zéro, elle ne peut plus diminuer. On pourra utiliser les fonctions `max` et `min`.

```
>>> pos = (10,10)
>>> déplacement_restreint(pos, 4, 7, 15)   # et non pas (14,17) car 17>15
(14, 15)
>>> déplacement_restreint(pos, -20, -2, 15) # et non pas (-10, 8) car -10<0
(0, 8)
```

```
def déplacement_restreint(position, dx, dy,taille):
    (x,y) = déplacement(position)
    x = min(x, taille)
    y = min(y, taille)
    x = max(x, 0)
    y = max(y, 0)
    return (x, y)
```

3. On encode une suite de mouvements par une chaîne de caractères (exemple : `"HHGGBBDDHGGD"`) représentant les différentes directions (**H**aut, **B**as, **G**auche, **D**roite). Les directions `"H"` et `"G"` diminue par 1 les coordonnées correspondantes, alors que `"B"` et `"D"` les augmentent de 1. Écrire une fonction `voyage(position, chemin, taille)` qui calcule la position après les déplacements décrits dans chemin. Exemple, en prenant `p = (0,0)`

```
>>> voyage(p, "BDDDBDB", 5)
(4, 3)
>>> voyage(p, "HHHHHHHHHHBB", 5) # malgré les "H", y ne descend jamais sous 0
(0, 2)
>>> voyage(p, "DDDDDDDDDDGG", 5) # malgré les "D", x ne dépasse jamais 5
(3, 0)
```

```
def voyage(position, chemin, taille):
    p = position
    for i in range(len(chemin)):
        if chemin[i] == "G":
            p = déplacement_restreint(p, -1, 0, taille)
        elif chemin[i] == "D":
            p = déplacement_restreint(p,  1, 0, taille)
        elif chemin[i] == "H":
            p = déplacement_restreint(p, 0,  1, taille)
        else:
            p = déplacement_restreint(p, 0, -1, taille)
    return p
```

Exercice 5 Permutations . . . . . 5 points

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

1. Écrire une fonction `nouveau(n)` qui renvoie un nouveau tableau (c'est-à-dire une liste) de taille `n` contenant comme valeurs les entiers de `0` à `n-1`. Rappel : l'usage de la méthode append est interdit.

```
>>> nouveau(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
def nouveau(n):
    tab = [0] * n
    for i in range(n):
        tab[i] = i
    return tab
```

2. Écrire une procédure `échange(tab,i,j)` qui échange la valeur des cases d'indices `i` et `j`. On supposera que les indices sont bien valides. La fonction ne renverra rien mais modifiera directement le tableau.

```
>>> tab = [11,22,33,44]
>>> échange(tab, 1, 2) # on échange 22 avec 33
>>> tab
[11, 33, 22, 44]
>>> échange(tab, 0, 0) # on échange 11 avec 11 (donc aucun changement)
>>> tab
[11, 33, 22, 44]
```

```
def échange(tab,i,j):
    tmp = tab[i]
    tab[i] = tab[j]
    tab[j] = tmp
```

On utilisera pour la question suivante la fonction `randint(a,b)` qui renvoie un nombre entier aléatoire `k` compris entre `a` et `b` inclus ($a \leq k \leq b$).

```
>>> randint(0,5)
1
>>> randint(0,5)
5
```

```
>>> randint(0,5)
0
>>> randint(0,5)
2
```

3. On souhaite mélanger aléatoirement notre tableau. Supposons que son indice maximal soit 10. On commence par échanger la case d'indice 10 avec une case d'indice aléatoire entre 0 et 10. Puis on recommence en échangeant la valeur de la case d'indice 9 avec une case d'indice aléatoire entre 0 et 9. Exemple :

- Au début on prend `tab = [0, 1, 2, 3, 4, 5]`
- On tire un nombre aléatoire `a` vérifiant $0 \leq a \leq 5$ puis on échange `t[a]` avec `t[5]`
- Avec `a=2`, on obtient `tab =[0, 1, 5, 3, 4, 2]`
- On tire un nombre aléatoire `a` vérifiant $0 \leq a \leq 4$ puis on échange `t[a]` avec `t[4]`
- Avec `a=0`, on obtient `tab =[4, 1, 5, 3, 0, 2]`
- On recommence avec `t[3]`, `t[2]`, `t[1]` puis `t[0]`.
- À la fin, la liste est bien permutée de manière aléatoire.

Écrire la procédure `mélanger(tableau)` qui implémente l'algorithme précédent en modifiant le tableau et sans rien renvoyer.

```
>>> tab = [0,1,2,3,4]
>>> mélanger(tab)
>>> tab
[0, 1, 4, 2, 3]
>>> mélanger(tab)
>>> tab
[3, 4, 1, 0, 2]
```

```
def mélanger(tab):
    n = len(tab)
    for i in range(len(tab)-1, -1, -1):
        a = randint(0, i)
        échange(tab, a, i)
```

4. Considérons le tableau `tab = [5,2,0,4,3,1]`. On a `tab[0]=5`, puis `tab[5]=1`, `tab[1]=2` et enfin `tab[2]=0`. On dit que $0 \to 5 \to 1 \to 2 \to 0$ forme un cycle.

Écrire une fonction `cycle(tab,i)` qui affiche le cycle obtenu à partir de l'indice `i`. On considère que la liste `tab` contient tous les entiers de 0 à n-1 (où n est la taille de `tab`).

```
>>> cycle([5,2,0,4,3,1], 0)
0 -> 5 -> 1 -> 2 -> 0
>>> cycle([5,2,0,4,3,1], 1)
1 -> 2 -> 0 -> 5 -> 1
>>> cycle([5,2,0,4,3,1], 3)
3 -> 4 -> 3
```

```
def cycle(tab,i):
    print(i, end=" -> ")
    k = tab[i]
    while k != i:
        print(k, end=" -> ")
        k = tab[k]
    print(k)
```