Informatique

TP noté 1

Semaine du 14 avril 2025
Durée : 1 heure 00

1 Les Shadoks se mettent à l'informatique

29 points

Les *Shadoks* possèdent pour tout vocabulaire quatre mots monosyllabiques : « Ga, Bu, Zo, Meu ». Ces mots leur servent aussi de chiffres pour compter : Ga pour 0, Bu pour 1, etc.

Dans leur conquête sans relâche de la planète voisine Buga, ils mettent en place un système informatique utilisant tout naturellement la base 4. Or, les occupants de la planète Buga utilisent comme nous la base 10...

1. Les *Shadoks* choisissent d'encoder, en base 4, des entiers naturels sur 4 « *bi-bits* ». Ils seront pour nous représentés par des tableaux numpy. Par exemple,

198 est encodé par `np.array([3, 0, 1, 2])` .

3 pts **a.** Créer une fonction `int_to_t4` qui prend pour paramètre un entier `n` compris entre 0 et 255 et qui rend le tableau numpy de son écriture en base 4.

Le code est le suivant. Il est déjà complété pour la question 1.b. ▶ 0,5 pt initialiser ▶ 0,5 pt boucle for ▶ 1,5 pt calculs dans la boucle ▶ 0,5 pt return

```
""" ***** Exercice Shadok ***** """

# Questions 1a et 1b

import numpy as np # pour les tableaux

def int_to_t4(n):
    # n doit être un entier
    assert (type(n) == int)
    # compris entre 0 et 255
    assert (n >= 0 and n <= 255)

    t4 = np.zeros(4, dtype = int) # initialiser le tableau
    val = n # valeur à traiter, pour ne pas modifier l'entrée n
    for i in range(len(t4)): # parcourir le tableau
        p = 3-i # puissance de 4 à considérer
        q = val // 4**p # valeur à mettre
        t4[i] = q # mise à jour du tableau
        val = val - q * 4**p # traiter le reste

    assert (val == 0) # vérifier que tout est encodé
    return t4

# test de la fonction avec l'exemple de l'énoncé
print("198 est encodé par", int_to_t4(198), "d'après mon code")
```

✎ *Ligne 28, ce simple test vous permet de vérifier votre code sur un exemple connu.*

✎ $255 = 4^4 - 1$, *c'est bien la valeur maximale pouvant être encodée par 4* bi-bits.

1 pt **b.** Modifiez votre code afin de spécifier `int_to_t4` à l'aide de commentaires et de `assert`. Si

vous avez sauté la question précédente, vous pouvez traiter celle-ci en utilisant `res = np.zeros(4)` pour définir le résultat que le code devrait fournir normalement (il rend alors systématiquement la représentation de 0).

Voir le code de la question 1.a. ▶ 0,5 pt type de n ▶ 0,5 pt bornes de n Si elle a été sautée, il est possible d'écrire

```
1 def int_to_t4(n):
2     # n doit être un entier
3     assert (type(n) == int)
4     # compris entre 0 et 255
5     assert (n >= 0 and n <= 255)
6
7     res = np.zeros(4, dtype = int) # initialiser le tableau
8     # mais le calcul n'est pas implémenté
9     return res
```

4 pts **c.** Créer une fonction `t4_to_int` qui effectue l'opération réciproque de `int_to_t4`. La spécifier. *Attention :* lors des calculs avec `numpy`, le type d'un entier peut être `numpy.int64`, donc `np.int64` selon votre import du module, au lieu de `int`.

Le code est le suivant. ▶ 0,5 pt assert type de t ▶ 0,5 pt initialiser ▶ 0,5 pt boucle for ▶ 1,5 pt calculs dans la boucle ▶ 0,5 pt assert valeurs de c ▶ 0,5 pt return

```
30 # Question 1c
31
32 def t4_to_int(t4):
33     # t4 doit être un tableau d'une écriture en base 4
34     assert (type(t4) == np.ndarray)
35
36     n = 0 # initialiser le résultat
37     for p in range(len(t4)): # itérer sur les puissances de 4
38         # ici, en remontant de droite à gauche
39         c = t4[len(t4)-(p+1)] # récupérer le coefficient
40         assert (c in [0, 1, 2, 3]) # seules possibilités pour la valeur de c
41         n = n + c * 4**p # calculer le terme à ajouter
42     assert (type(n) == np.int64) # le résultat doit être un entier
43     return n
44
45 # test de la fonction avec l'exemple de l'énoncé
46 t4_198 = np.array([3, 0, 1, 2])
47 assert (t4_to_int(t4_198) == 198)
```

✎ *Lignes 45-47, ce simple test vous permet de vérifier votre code sur un exemple connu.*

2. En explorant la planète Buga, les Shadoks découvrent que ses occupants utilisent plus que des entiers naturels, par exemple le décimal « 56,75 ». Afin de l'encoder, ils décident d'utiliser plus de *bi-bits* et une méthode analogue à celle utilisée pour les flottants en base 2. Pour $x \in \mathbb{R}^*$,

$$x = \pm m \times 4^k, \quad m \in [1;4[, \quad k \in \mathbb{Z} \quad \text{en utilisant :}$$

— 1 *bi-bit* pour coder le signe de m : 0 pour + et 1 pour − ;
— 1 *bi-bit* pour coder la partie entière de m ;
— 10 *bi-bit* pour coder la partie décimale de m ;
— 4 *bi-bit* pour coder k.

1 pt **a.** Pourquoi faut-il ici utiliser une case mémoire (bit ou *bi-bit*) pour la partie entière de m alors que ce n'est pas le cas en base 2 ?

En base 2, en écrivant $x = \pm m' \times 2^{k'}$, la partie entière de m' est nécessairement 1, il n'est donc pas nécessaire de l'encoder. En base 4 en revanche, en écrivant $x = \pm m \times 4^k$, la partie entière de m appartient à $[\![1;3]\!]$, il est donc nécessaire de l'encoder. ▶ 1 pt

6 pts **b.** Compléter le code fourni pour qu'il permette de donner le tableau numpy correspondant à un

décimal. Comment est alors encodé 56,75 ?

Le code est le suivant. ▶ 0,5 pt assert ▶ 0,5 pt signe, bi-bit mis à 1 ▶ 1 pt indice de t4 ▶ 0,5 pt indice de t4_pour_k ▶ 0,5 pt indice 1 pour partie entière ▶ 0,5 pt multiplier par 4 ▶ 1 pt int(dec) ▶ 1 pt retrancher int(dec)

```
# Question 2b

def float_to_t4(d):
    # d doit être un entier ou un float
    assert (type(d) == int or type(d) == float)

    t4 = np.zeros(16, dtype = int) # initialiser le tableau
    m = d # valeur qui devra correspondre à la mantisse

    if d < 0: # premier bi-bit pour le signe
        t4[0] = 1 # stocker l'information du signe
        m = -m # on se ramène à un positif

    # trouver l'exposant
    k = 0
    while m >= 4: # cas d supérieur à 4
        m = m/4
        k = k + 1
    while m < 1: # cas d inférieur à 1
        m = m*4
        k = k - 1
    t4_pour_k = int_to_t4(k) # 4 derniers bi-bits pour k
    for i in range(len(t4_pour_k)):
        t4[len(t4)-len(t4_pour_k)+i] = t4_pour_k[i]

    # pour la mantisse
    # 1 bi-bit pour sa partie entière
    partie_entiere_m = int(m)
    t4[1] = partie_entiere_m

    # 10 bi-bits pour la partie décimale
    dec = m - partie_entiere_m
    for i in range(10):
        dec = dec * 4 # se décaler d'une 'quadrimale' (décimale base 4)
        t4[2+i] = int(dec) # remplir le tableau
        dec = dec - int(dec) # retirer ce qui est encodé dans le tableau

    return t4

# déterminer la représentation de 56.75
print("56.75 en encodé par", float_to_t4(56.75), "d'après mon code")
```

Ainsi, 56,75 est encodé par `[0 3 2 0 3 0 0 0 0 0 0 0 0 0 0 2]`. ▶ 0,5 pt

5,5 pts **c.** Compléter le code fourni pour que `t4_to_float` permette d'effectuer l'opération réciproque de `float_to_t4`.

Le code est le suivant. ▶ 0,5 pt assert type ▶ 0,5 pt assert len ▶ 1 pt indice de t4 ▶ 1 pt assert ▶ 1,5 pt ajout du terme ▶ 1 pt extraction de l'exposant

```
# Question 2c
def t4_to_float(t4):
    # t4 doit être un tableau d'une écriture en base 4
    assert (type(t4) == np.ndarray)
    # contenant 16 bi-bits
    assert (len(t4) == 16)

```

```
98      # déterminer la mantisse
99      m = 0
100     for p in range(12): # itérer sur 10 + 1 bi-bits
101         c = t4[p+1] # récupérer le coefficient
102         assert (c == 0 or c == 1 or c == 2 or c == 3) # seules possibilités pour la
            ↪ valeur de c
103         m = m + c * 4**(-p) # ajouter ce terme
104
105     # déterminer l'exposant
106     t_k = t4[12:] # extraire la partie encodant l'exposant
107     k = t4_to_int(t_k) # utiliser la fonction déjà codée
108
109     # resultat = m * 4**k
110     res = m * 4.**k # ne pas oublier le . avec 4. !!!
111
112     # cas du signe négatif
113     if t4[0] == 1:
114         res = -res
115     return res
116
117 assert(t4_to_float(float_to_t4(56.75)) == 56.75)
```

✎ *Ligne 117, ce simple test vous permet de vérifier votre code sur un exemple connu.*

1,5 pt **d.** En découvrant le décimal « 12,8 », les Shadoks paniquent. Pourquoi ?

Avec les fonctions précédentes, il est possible de déterminer la représentation de 12,8 (ligne 119) et tenter de la retraduire en flottant (ligne 120) ▶ 0,5 pt idée de tester

```
119 print("La représentation de 12.8 est", float_to_t4(12.8))
120 print("En revenant en float, cela donne", t4_to_float(float_to_t4(12.8)))
```

Cela donne :

```
La représentation de 12.8 est [0 3 0 3 0 3 0 3 0 3 0 3 0 0 0 1]
En revenant en float, cela donne 12.799999237060547
```

La représentation de 12,8 sur un nombre fini de « bits » en base 4 (ici nommés *bi-bits*) ne peut pas être exacte, ce devrait être une suite infinie de `030303...`. ▶ 0,5 pt constat Si la fonction réciproque est utilisée, comme `float_to_t4(12.8)` n'est pas exactement la représentation de 12,8 mais seulement celle d'une valeur proche, alors `t4_to_float(float_to_t4(12.8))` ne rend pas 12,8. ▶ 0,5 pt conclusion

2 pts **3.** Quelle est la plus grande valeur encodable par cette méthode ? Proposer un calcul théorique simple puis une vérification numérique.

Le calcul théorique est ▶ 1 pt

$$(4-4^{-11})\times 4^{4^4-1} = (4-4^{-11})\times 4^{255} = 4^{256}-4^{244} \simeq 1{,}340\,780\,713\,077\,5\times 10^{+154}$$

Le code pour le vérifier est ▶ 1 pt vérification même sommaire

```
122 # Question 3 : plus grande valeur encodable par cette méthode
123
124 M = 3*np.ones(16, dtype = int)
125 M[0] = 0
126 print("La représentation de la plus grande valeur est :", M)
127 print("D'après t4_to_float, cela donne :", t4_to_float(M))
128 print("Le calcul théorique est :", (4 - 4**(-11)) * 4**(4**4 - 1))
129 print("Vérification :",  t4_to_float(M) == (4 - 4**(-11)) * 4**(4**4 - 1))
```

Cela donne :

```
La représentation de la plus grande valeur est : [0 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3]
D'après t4_to_float, cela donne : 1.3407807130774968e+154
Le calcul théorique est : 1.3407807130774968e+154
Vérification : True
```

La valeur exacte de $4^{256} - 4^{244}$ est :

134078071307749682186800137645169553004287915489512688554909468237775823659783818389418755473685410434360872412707916837736584215772647960486326105957990 40

5 pts **4.** En réalité, les Shadoks ne disent pas « 0, 1, 2, 3 » mais bien « ga, bu, zo, meu ». Écrire la fonction de traduction `t4_to_Shadok` qui prend pour paramètre `t` le tableau numpy de l'écriture en base 4 d'un nombre **entier** et qui rend son écriture Shadok. Chaque mot est séparé par une espace et, bien entendu, les zéros précédant le premier *bi-bit* non nul ne sont pas prononcés. Vérifier que le tableau représentant 25 se dit bien `"bu zo bu"`.

Le code est le suivant. ▶ 1 pt utilisation d'un dictionnaire ▶ 0,5 pt ou d'une liste ▶ 0,5 pt initialiser ▶ 0,5 pt boucle for ▶ 1 pt condition sur le premier mot ▶ 1 pt construction avec espace ▶ 0,5 pt return ▶ 0,5 pt cas zéro

```
131 # Question 4
132
133 dic_to_Sha = { # utiliser un dictionnaire permet de 'faire le lien'
134     0 : "ga",
135     1 : "bu",
136     2 : "zo",
137     3 : "meu",
138 }
139
140 def t4_to_Shadok(t4):
141     mot = ""
142     for i in range(len(t4)):
143         bbit = t4[i]
144         if mot != "" or bbit != 0: # ne pas prononcer ga avant le premier bu/zo/meu
145             if mot != "":
146                 mot = mot + " " # mettre une espace entre les mots, mais pas au début
147             mot = mot + dic_to_Sha[bbit]
148     if mot == "": # t4 représente alors 0
149         return dic_to_Sha[0]
150     return mot
151
152 print(int_to_t4(25))
153 print(t4_to_Shadok(int_to_t4(25)))
```

✎ *Lignes 152-153, la vérification demandée par l'énoncé est réalisée.*

2 Chargement d'un camion

23 points

Marcel est bien embêté : le groupe de musique *Rock the Factory* a besoin de son camion pour transporter son matériel en vue d'un concert, mais les boites de matériel à charger dans la remorque du camion ont des dimensions très variables. Son ami Fred lui suggère alors de mettre au point un code pour tenter d'optimiser l'agencement des boites.

La remorque de Marcel possède une surface utile de 2,5 m × 5 m. Pour visualiser les résultats du code, il décide de la représenter par une image dont un pixel correspond à 10 cm × 10 cm. Si le pixel est blanc, la zone est libre ; si le pixel est noir, la zone est occupée. Pour simplifier, la hauteur n'est pas prise en compte dans cet exercice.

1,5 pt **1.** Créer une fonction `remorque_vide` qui prend comme paramètres `Lo` la longueur de la remorque et

`La` sa largeur renseignées en mètre à l'aide de flottants et qui rend le tableau numpy correspondant aux pixels qui, traduits en image, montrent la remorque vide. Marcel accède à sa remorque par l'arrière qui doit correspondre au bas de l'image.
Indication : un pixel blanc se code par « 255 », un pixel noir par « 0 ».

Le code est le suivant. ▶ 0,5 pt conversion de m à pxl ▶ 0,5 pt dimensions ▶ 0,5 pt type int

```
4  """ ***** Exercice chargement d'un camion ***** """
5
6  import numpy as np # pour les tableaux
7  import matplotlib.pyplot as plt # pour les images
8
9  # Question 1
10
11 def remorque_vide(Lo, La):
12     """
13     conversion de m à pxl.
14     Mieux vaut prendre la partie entière,
15     cela revient à sous-estimer éventuellement la surface
16     disponible
17     """
18     cm_par_pxl = 10
19     pxl_Lo = int(Lo * 100 / cm_par_pxl)
20     pxl_La = int(La * 100 / cm_par_pxl)
21
22     return 255 * np.ones((pxl_Lo, pxl_La), dtype = int)
```

1,5 pt **2.** Créer une fonction `show_save_remorque` qui prend comme paramètres `t` un tableau numpy représentant la remorque et `nom` le nom du fichier de sauvegarde de l'image et qui affiche l'image de cette remorque *et* la sauvegarde au format png.
Indication : si `plt.imshow(t) ; plt.show()` permet d'*afficher* l'image,
`plt.imsave(f"{nom}.png", t)` permet de *sauvegarder* l'image au format png.
Remarques :

- Avec `show_save_remorque(t, "test")`, vous pourrez afficher (et sauvegarder dans `test.png`) une représentation de la remorque au cours de la rédaction de votre code ;
- Si la remorque est complètement vide, Python peut l'afficher comme pleine car tous les pixels ont la même valeur. Dans ce cas, mettre un pixel à l'autre valeur extrême (comme s'il y avait un petit colis dans la remorque) permet de tester correctement le code.

Le code est le suivant. ▶ 0,5 pt show ▶ 1 pt save (cmap non exigée)

```
28 # Question 2
29
30 def show_save_remorque(t, nom):
31     # montrer
32     plt.imshow(t) ; plt.show()
33     # sauvegarder, cmap="Greys_r" permet d'avoir le bon nuancier de couleurs
34     plt.imsave(f"{nom}.png", t, cmap="Greys_r")
35
36 t = remorque_vide(5.0, 2.5)
37 t[1,1] = 0 # mettre quelque chose dans la remorque
38 show_save_remorque(t, "test")
```

✎ *Lignes 36-38, il s'agit du test du code.*

6,5 pts **3.** La liste des boites de matériel à ranger est construite de la manière suivante : chaque élément est lui-même une liste de deux entiers, la longueur et la largeur de la boite, par exemple :

```
boites = [ [12, 8], [6, 3] ]
```

correspond à une boite de 120 cm × 80 cm et une de 60 cm × 30 cm.

Marcel accède à sa remorque par l'arrière qui doit correspondre au bas de l'image. Il souhaite pouvoir mettre chaque boite le plus au fond de la remorque possible, c'est-à-dire le plus en haut sur l'image, puis le plus à gauche possible dans la remorque c'est-à-dire le plus en gauche sur l'image. Encore faut-il qu'il y ait de la place !

Modifier la fonction `range_1_boite` fournie pour qu'elle « range » une boite avec la méthode de Marcel. Elle prend comme paramètres `t` un tableau numpy représentant la remorque avant ajout d'une boite et `boite` la boite à ranger dans cette remorque et qui rend `True` et un tableau numpy représentant la remorque après ajout d'une boite s'il est possible de l'y ranger, `False` et le tableau initial sinon.

Le code est le suivant. ▶ 1 pt while x_pos ▶ 1 pt while y_pos ▶ 0,5 pt range de xi, deux fois ▶ 0,5 pt range de yi, deux fois ▶ 0,5 pt boite détectée ==0 ▶ 0,5 pt zone libre = False ▶ 0,5 pt colorier ▶ 0,5 pt test de la position suivante, deux fois

```
40 # Question 3
41
42 def range_1_boite(t, boite):
43     Lo_b, La_b = boite # longueur et largeur de la boite
44
45     # trouver la position où mettre le coin
46     # en haut à gauche de la boite sur l'image
47     x_pos = 0 # position selon la longueur de la remorque
48     # tester jusqu'à ce que la boite ne puisse plus rentrer selon x
49     while x_pos < t.shape[0] - Lo_b :
50         y_pos = 0 # position selon la largeur de la remorque
51         # tester jusqu'à ce que la boite ne puisse plus rentrer selon y
52         while y_pos < t.shape[1] - La_b :
53             # tester si la position est déjà occupée
54             zone_libre = True # initialiser la variable
55             for xi in range(Lo_b):
56                 for yi in range(La_b):
57                     if t[x_pos+xi, y_pos+yi] == 0: # boite détectée
58                         zone_libre = False # zone non libre
59             # si la zone est libre, alors la boite peut être placée
60             if zone_libre:
61                 for xi in range(Lo_b):
62                     for yi in range(La_b):
63                         t[x_pos+xi, y_pos+yi] = 0 # colorier en noir
64                 return (True, t)
65             y_pos = y_pos + 1 # tester la position suivante selon y sinon
66         x_pos = x_pos + 1 # tester la position suivante selon x
67     return (False, t)
68
69 t = remorque_vide(5.0, 2.5)
70 boites = [ [12, 8], [6, 3] ]
71 reussi, t = range_1_boite(t, boites[0])
72 show_save_remorque(t, "1_boite")
73 reussi, t = range_1_boite(t, boites[1])
74 show_save_remorque(t, "2_boites")
75 reussi, t = range_1_boite(t, [100, 100]) # vérifier qu'une boite trop grande ne rentre
   ↪ pas
76 show_save_remorque(t, "2_boites_toujours")
```

✎ *Lignes 69-76, ces tests vous permettent de vérifier votre code.*

1 pt **4.** Marcel doit ranger n boites dans sa remorque. Il se dit qu'il suffit de prendre la liste des boites, de ranger l'une d'entre elles selon la procédure de `range_1_boite`, puis de répéter la procédure sans se soucier de l'ordre des boites. Comment s'appelle un tel algorithme ?

Il s'agit d'un algorithme *glouton*. ▶ 1 pt Son principe est de faire à chaque étape le choix *localement* optimal (ici, positionner une boite au fond à gauche de la remorque). Cela ne permet pas, en général, d'arriver à l'optimum global (par exemple, des espaces non remplis peuvent apparaitre ou non selon l'ordre d'arrivée des boites).

3,5 pts **5.** Créer une fonction **récursive** `remplir_remorque` qui prend comme paramètres `t` un tableau numpy représentant la remorque `boites` la liste des boites de matériel à ranger et `nom` une chaine de caractères (utilisée seulement à partir de la question 6) et qui rend `t` un tableau numpy représentant la remorque remplie des boites rangées et `restantes` la liste des boites de matériel qui ne sont pas rentrées. La fonction `range_1_boite` y sera judicieusement utilisée.

Le code est le suivant. ▶ 1 pt condition initiale, return explicite ▶ 2 pts appel récursif avec pop ou équivalent ▶ 0,5 pt return Il est déjà complété pour la question 6.

```
78 # Questions 5 et 6
79
80 def remplir_remorque(t, boites, nom):
81     n_boites = len(boites) # pour la q6
82
83     if len(boites) == 0: # rien à faire !
84         return t, []
85
86     boite = boites.pop() # récupérer la derniere boite et la retirer des boites
       ↪  restantes
87     t, restantes = remplir_remorque(t, boites, nom) # récursivité
88
89     reussi, t = range_1_boite(t, boite) # traiter le cas présent
90     if not reussi:
91         restantes.append(boite)
92
93     show_save_remorque(t, f"{nom}-etape_{n_boites}") # pour la q6
94
95     return t, restantes
```

1 pt **6.** Mettre à jour le code de `remplir_remorque` en utilisant `show_save_remorque` afin de sauvegarder *toutes* les étapes de remplissage de la remorque sous la forme d'images dont le nom est de la forme `nom-etape_n` avec `nom` la variable `nom` évoquée à la question 5 et `n` le nombre de boites traitées (mais pas forcément *rangées*).

Voir le code de la question 5. ▶ 1 pt modifications

7. Tester votre code sur les exemples suivants :

```
test1 = [ [12, 8], [6, 3], [30, 10], [16, 10], [6, 4] ] ,
test2 = [ [12, 8], [6, 3], [10, 30], [16, 10], [6, 4] ]
```

et `test3 = [ [12, 8], [6, 3], [30, 10], [ 2, 20], [6, 4] ]` .

Le code pour réaliser ces tests est le suivant.

```
97 # Question 7
98
99 test1 = [ [12, 8], [6, 3], [30, 10], [16, 10], [6, 4] ]
100 test2 = [ [12, 8], [6, 3], [10, 30], [16, 10], [6, 4] ]
101 test3 = [ [12, 8], [6, 3], [30, 10], [ 2, 20], [6, 4] ]
102
103 t = remorque_vide(5.0, 2.5)
104 t, restantes = remplir_remorque(t, test1, "test1")
105
106 t = remorque_vide(5.0, 2.5)
107 t, restantes = remplir_remorque(t, test2, "test2")
108
```

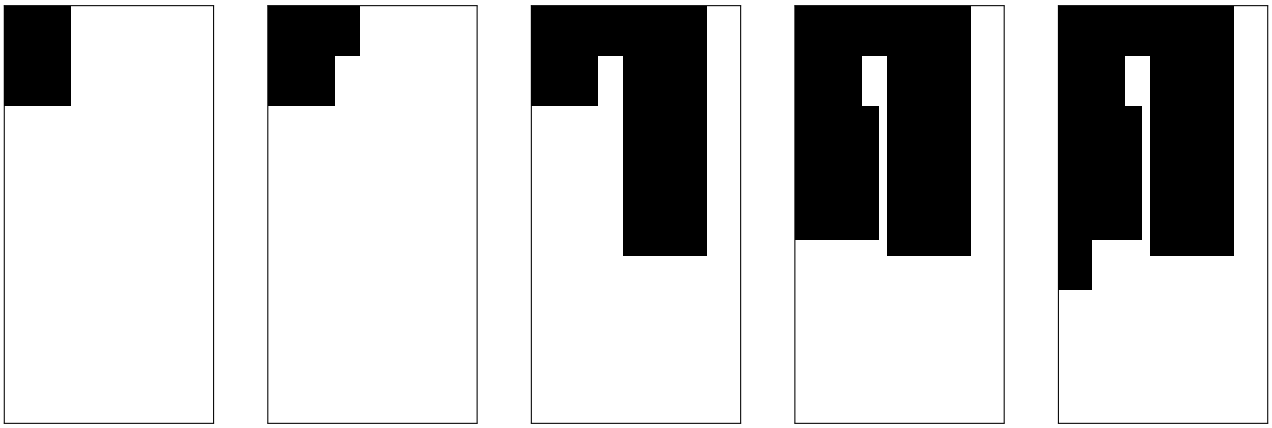
```
109 t = remorque_vide(5.0, 2.5)
110 t, restantes = remplir_remorque(t, test3, "test3")
```

4 pts **a.** Les listes `test1` et `test2` comportent les « mêmes » boites, pourquoi une telle différence ? Quel est le problème avec `test3` ?

Les images obtenues sont données en figures C1, C2 et C3. ▶ 2 pts production des images de test

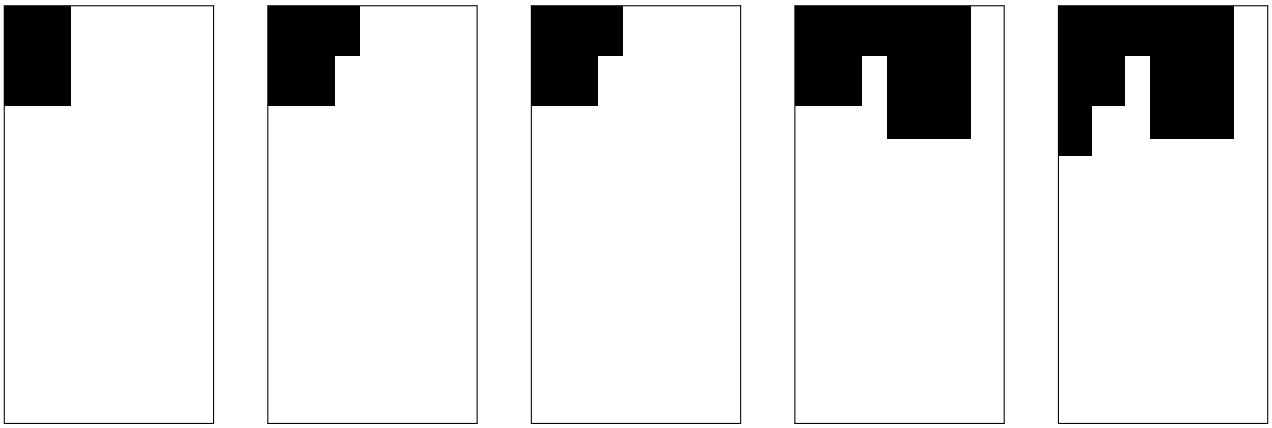
Dans le test 2, la troisième boite de dimensions 10×30 est trop large pour la remorque, alors que dans le test 1 elle est convenablement tournée (30×10) pour pouvoir rentrer. ▶ 1 pt

Dans le test 3, la boite très fine ajoutée à l'étape 4 devrait empêcher l'ajout de la dernière boite, à l'étape 5, dans l'espace vide à gauche de la remorque (le chemin est bloqué en pratique). ▶ 1 pt



(a) *Étape 1.* **(b)** *Étape 2.* **(c)** *Étape 3.* **(d)** *Étape 4.* **(e)** *Étape 5.*

Figure C1 – *Images issues du test 1.*



(a) *Étape 1.* **(b)** *Étape 2.* **(c)** *Étape 3.* **(d)** *Étape 4.* **(e)** *Étape 5.*

Figure C2 – *Images issues du test 2.*

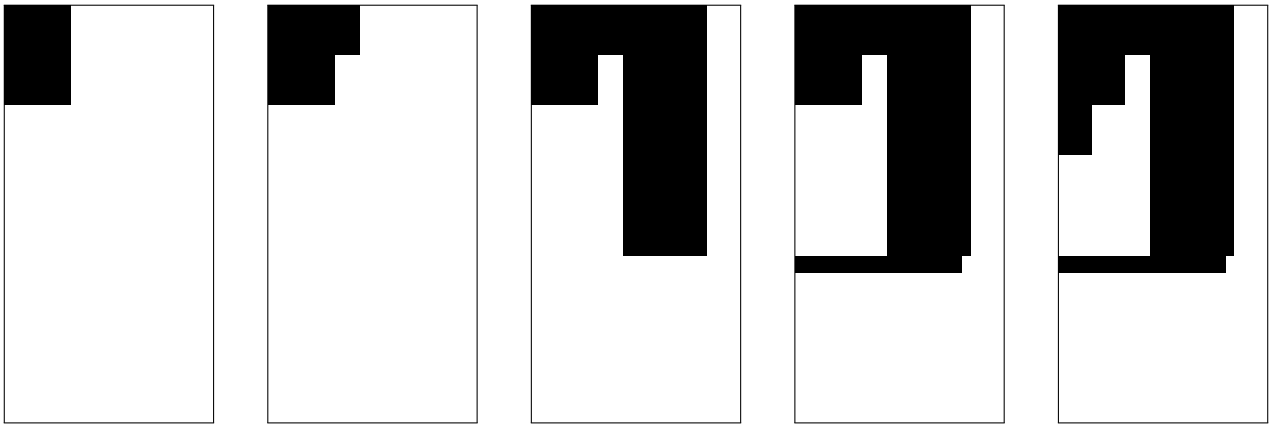
4 pts **b.** Proposez des pistes d'amélioration du code, voire une réécriture partielle du code pour mieux traiter le problème de Marcel.

Compte-tenu de la différence entre les tests 1 et 2, une possibilité est de faire tourner les boites qui ne peuvent pas rentrer en l'état.

Vu le test 3, il faudrait vérifier qu'une boite puisse bien se faufiler jusqu'à la position choisie. Sinon, il suffit de signifier que l'ordre de placement des boites en réalité ne correspond pas à celui de traitement par le code.

De plus, la boite ajoutée à l'étape 3 du test 1 (figure C1c) ou du test 3 (figure C3c) pourrait être mise plus sur la côté droite, contre la paroi de la remorque. Cela permettrait de libérer plus d'espace central.

Enfin, un algorithme glouton ne permet pas, en général, d'arriver à l'optimum global. Par exemple, des espaces non remplis peuvent apparaitre ou non selon l'ordre d'arrivée des boites, il pourrait être judicieux d'ajouter une étape de tri des boites selon un critère donné (leur surface ?) afin de remplir plus efficacement la remorque. ▶ 1 pt pour toute proposition sensée, max 4 fois



(a) *Étape 1.* **(b)** *Étape 2.* **(c)** *Étape 3.* **(d)** *Étape 4.* **(e)** *Étape 5.*

Figure C3 – *Images issues du test 3.*

FIN