Exercice 4 : listes chaînées

Cet exercice utilise le type de données abstrait *liste chaînée*. Il est implémenté en utilisant la classe `Cellule`

```
1  class Cellule():
2      def __init__(self, valeur, suivante = None):
3          self.valeur = valeur
4          self.suivante = suivante # type Cellule, None par défaut
```

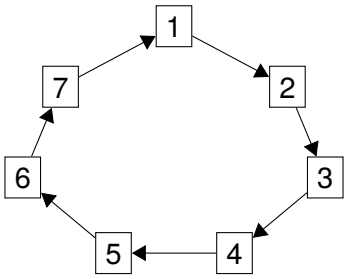
Dans cet exercice, une liste chaînée est représentée par sa première cellule.

Le problème de Josephus *L'historien Falvius Josephus raconte que pendant la guerre contre les Romains, lui et ses soldats furent piégés dans une grotte par leurs ennemis. Plutôt que de se rendre, ils décidèrent de se donner la mort les uns les autres par tirage au sort selon une règle que nous allons étudier ici.*

N personnes forment un cercle en se tenant par la main. Elles sont numérotées de 1 à n. On choisit un nombre m entre 1 et n inclus. La personne n° m est éliminée, et le cercle se reforme sans elle. Ensuite, la m-ième personne en comptant à partir de la suivante de celle qui vient d'être éliminée, est à son tour éliminée, et ainsi de suite, jusqu'à ce qu'il ne reste qu'une seule personne, qui sera épargnée.

On appelle *permutation de Josephus* la suite des numéros des personnes éliminées, et le *problème de Josephus* consiste à déterminer, étant donné la personne n°1, et m, quel est le numéro de la gagnante.

Exemple



On prend $m = 3$. C'est donc la personne n° 3 qui est éliminée en premier. Ensuite, on compte jusqu'à 3 en partant de la personne n° 4 : cela tombe sur la personne n° 6, qui est donc, à son tour, éliminée. On compte encore jusqu'à 3 en partant de la personne n° 7, et cela tombe sur la personne n° 2, qui est donc à son tour éliminée. Les 3 premiers numéros de la permutation sont donc : 3, 6, 2. À la fin, la permutation est : 3, 6, 2, 7, 5, 1, et la gagnante est 4.

Question 1 Dans l'exemple précédent, où $(n = 7; m = 3)$,

1. dessinez la liste chaînée sur le modèle du schéma proposé, mais après que 4 personnes auront été éliminées ;
2. puis, lorsqu'il ne reste qu'une seule personne ;
3. si la dernière personne est représentée par la cellule **x**, que vaut **x.suivante** ?

Dans la suite de l'exercice, chaque personne n°x est représentée par une cellule de liste chaînée dont la valeur est x. L'élimination d'une personne consistera à *supprimer* la cellule représentant cette personne.

Question 2 Écrivez la fonction python **cercle (n)** qui renvoie la première cellule d'une liste chaînée représentant le cercle des n personnes. Avec $n = 7$, elle devra retourner la liste représentée par le schéma ci-dessus. La première cellule est la **suivante** de la dernière.

Question 3 Écrivez la fonction python **nieme (prem, n)** qui renvoie la **n**-ième cellule (en comptant de sorte que la cellule **prem** a le n° 1).

Question 4 Écrivez la fonction python **eliminer (cell)** qui renvoie la cellule suivant **cell** et supprime **cell**.

Question 5 Écrivez la fonction JOSEPHUS (cercle, m) qui affiche la suite des n° des personnes éliminées dans le bon ordre, suivies du n° de la gagnante. Cette fonction prend comme arguments les variables :

- **cercle** représente le cercle des personnes, une liste chaînée renvoyée par la fonction de la question 1.
- **m** le numéro de la personne qui est éliminée en premier.

Par exemple :

```
>>> c = cercle(7)
>>> josephus(c, 3)
3
6
2
7
5
1
4
```

Exercice 5 : diviser pour régner

Dans cet exercice, on se propose d'appliquer le principe « Diviser pour régner » pour multiplier deux entiers, avec la méthode de Karatsuba.

Le principe est le suivant. Supposons deux entiers x et y ayant chacun $2n$ chiffres en base 2.

On peut les écrire sous la forme

$x = 2^n a + b$

$y = 2^n c + d$

avec $0 < a, b, c, d < 2^n$,

c'est-à-dire avec quatre entiers a, b, c, d qui s'écrivent chacun sur n chiffres en base 2.

Dès lors, on peut calculer le produit de x et y de la façon suivante :

$$\begin{aligned} x \times y &= (2^n a + b)(2^n c + d) \\ &= 2^{2n} ac + 2^n (ad + bc) + bd \\ &= 2^{2n} ac + 2^n (ac + bd - (a - b)(c - d)) + bd \end{aligned}$$

Cette dernière forme, d'apparence inutilement compliquée, fait apparaître seulement **trois produits**, à savoir ac, bd et $(a - b)(c - d)$.

Ainsi, on a ramené la multiplication de deux entiers de $2n$ chiffres à trois multiplications d'entiers de n chiffres.

Pour faire chacune de ces trois multiplications, on peut appliquer le même principe, et ainsi de suite jusqu'à obtenir de petits entiers dont la multiplication est immédiate.

Au final, cela permet d'effectuer la multiplication en un temps proportionnel à $n^{1,58}$ (environ) au lieu de n^2, ce qui est un gain significatif lorsque le nombre de chiffres n est très grand.

1. Écrire une fonction `taille(x)` qui renvoie le nombre de chiffres de l'entier x lorsqu'il est écrit en base 2.
2. Écrire une fonction `karatsuba(x, y, n)` qui calcule le produit de x et y par la méthode de Karatsuba, en supposant que x et y s'écrivent sur n chiffres en base 2.

   Indication : On peut calculer 2^n en Python avec l'expression python

   ```
   1 << n
   ```

   . On peut décomposer x sous la forme $2^n \times a + b$ avec

   ```
   a, b = x >> n, x % (1 << n)}.
   ```

3. En déduire une fonction `mult(x, y)` qui calcule le produit de x et y.

Exercice 6 : Programmation en python : le problème du rendu de monnaie

Le problème consiste à rendre une somme **s** avec le moins de pièces possible, chaque pièce étant un exemplaire d'une liste de pièces de monnaie autorisées.

Le but du devoir est de construire une fonction **rendre_monnaie(P, s)**.

Entrées :

— un entier positif **s** qui représente la somme à rendre ;

— un tuple **P** qui contient toutes les valeurs des pièces possibles, de la plus petite à la plus grande. Par ex, **P = (1, 2, 5)** s'il existe des pièces de 1, 2, 5 euros. appelle **P** le système de monnaie.

On dispose d'une quantité illimitée de chaque pièce possible.

Sortie : la liste **R** des pièces rendues. Par ex, si l'algorithme rend 2 pièces de 1 euro et 3 pièces de 2 euros, **R = [1, 1, 2, 2, 2]**. Le but est qu'il y ait le moins possible de pièces rendues. Ainsi, pour rendre 8 euros, il faut retourner **[5, 2, 1]** et non par ex. **[2, 2, 2, 2]**. La première solution est dite *optimale*.

On se propose de remplir une liste **nb_pieces** de sorte que, pour chaque somme **0 <= j <= s**, **T[j]** est le plus petit nombre de pièces pour rendre la somme **j**.

L'algorithme proposé consiste à la remplir d'abord en utilisant seulement des pièces de la plus petite valeur possible, puis, seulement les deux pièces de la plus petite valeur, etc. jusqu'à utiliser toutes les pièces possibles.

Question 1 : étude d'un exemple d'itération de l'algorithme

Étudiez bien l'exécution de l'algorithme sur l'exemple **s = 8 ; P = (1, 4, 5)**.

Après la 1re itération :

pièces utilisées : de **1 à 1** inclus.

j	0	1	2	3	4	5	6	7	8
nb_pieces	0	1	2	3	4	5	6	7	8

après la 2e itération :

pièces utilisées : de **1 à 4** inclus.

j	0	1	2	3	4	5	6	7	8
nb_pieces	0	1	2	3	1	2	3	4	2

après la 3e itération :

pièces utilisées : de **1 à 5** inclus.

j	0	1	2	3	4	5	6	7	8
nb_pieces	0	1	2	3	1	1	2	3	2

(a) Finalement, quelle est la solution trouvée par cet algorithme et où se trouve-t-elle dans la liste **nb_pieces** ?

(b) Comparez le tableau **nb_pieces** après la 2e itération et après la première itération. Quelles sont les cases qui **n'ont pas** été mises à jour lors de la 2e itération, et pourquoi ?

(c) Quelles sont les cases qui ont été mises à jour après la 3e itération ? expliquez pourquoi.

(d) Appelons **p** est la plus grosse pièce disponible lors d'une itération, et soit **0 < j <= s**. Comment calculer **nb_pieces[j]** à partir de **nb_pieces[j - p]** ?

Question 2 : initialisation de la liste nb_pieces

Écrire le code pour initialiser la liste **nb_pieces** avec le cas où il n'y a que les pièces de 1 (1ère itération).

Question 3 : programmer la boucle interne

Écrivez la fonction **mettre_a_jour (nb_pieces, max_p)** où **max_p** est la valeur de pièce non encore utilisée.
Cette fonction met à jour la liste **nb_pieces** avec une nouvelle pièce utilisable.

```
pour chaque somme j de 0 à s incluse :
  si on peut rendre j avec moins de pieces que nb_pieces[ j ]:
    modifier nb_pieces[ j ] avec la nouvelle valeur
```

Exemples :

```
>>> nb_pieces = [0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> mettre_a_jour(nb_pieces, 4)
>>> nb_pieces
[0, 1, 2, 3, 1, 2, 3, 4, 2]
>>> mettre_a_jour(nb_pieces, 5)
>>> nb_pieces
[0, 1, 2, 3, 1, 1, 2, 3, 2]
```

Question 4 : programmer la boucle principale

À chaque itération, on ajoute une pièce de valeur supérieure au jeu de pièces de l'itération précédente, jusqu'à atteindre la plus grande pièce de **P**.
Programmer la fonction **rendre_monnaie(P, s)** qui retourne le plus petit nombre de pièces permettant de rendre la somme **s** avec le système de monnaie **P**.

Question 5 : analyse du temps d'exécution

Étant données **s** la somme à rendre, et **n** le nombre de pièces dans **P** , par quelle quantité estimer le temps d'exécution de cet algorithme (justifiez) ?

Question 6 : construire et retourner une solution

C'est bien joli, de savoir combien de pièces au miminum on utilisera pour rendre **s**, mais on voudrait savoir quelles sont ces pièces.
Pour le savoir, il suffit de remplir et de mettre à jour une deuxième liste **pièce_max**.
À chaque itération, on ajoute la pièce de valeur **i**. Pour chaque somme $j \leq s$, si l'on met à jour **nb_pieces[j]**, alors, c'est qu'une solution utilisant au moins une pièce de valeur **i** est meilleure. On inscrit **i** dans la liste **pièce_max**.
Modifier la fonction de la question précédente pour initialiser, puis mettre à jour à chaque itération, la liste **pièce_max** en même temps que la liste **nb_pieces**.
Voici une trace de l'algorithme pour l'exemple avec **s = 8 ; P = (1, 4, 5)**.

j	0	1	2	3	4	5	6	7	8
pièce_max	0	1	1	1	1	1	1	1	1

j	0	1	2	3	4	5	6	7	8
pièce_max	0	1	1	1	4	4	4	4	4

j	0	1	2	3	4	5	6	7	8
pièce_max	0	1	1	1	4	5	5	5	4

Modifiez le code précédent pour intégrer un parcours du tableau **pièce_max** permettant de reconstruire la solution optimale. Par ex, avec **s = 8 ; P = (1, 4, 5)**, on devrait retourner **[4, 4]**.