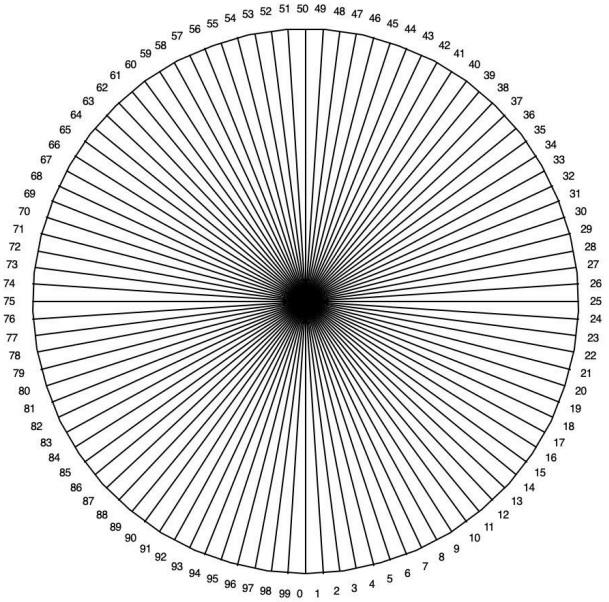
Exercice2

Comme mes écureuils s'ennuyaient, je leur ai fabriqué une roue avec des barreaux pour qu'ils puissent faire de l'exercice.

La roue fait 100 barreaux. Pour m'y retrouver, je les numérote de 0 à 99.



Roue de mes écureuils

Très vite je me suis rendu compte que chacun utilisait la roue en faisant des sauts réguliers (Up fait des sauts de 7 barreaux à chaque fois, Py des sauts de 9, LaB des sauts de 13, …).

Je mets une noisette sur un des barreaux de la roue. Aidez-moi à savoir si un de mes écureuils va l'attraper sachant que je vais mettre l'écureuil qui fait le test, par exemple Up, sur le barreau 0 et que je connais son comportement (Up fait toujours des sauts de 7 barreaux) et le numéro de barreau, différent de 0, où se trouve la noisette (Up n'aura donc pas la noisette au départ).

Écrire un programme qui teste si, pour une configuration donnée, un écureuil va ou non atteindre la noisette. Il reçoit deux valeurs entières en

entrée, une valeur **saut** et une valeur **position_cible**, toutes deux entre 1 et 99.

Le programme va calculer une valeur **position_courante**, initialement la valeur 0, et vérifier si en calculant de façon répétitive la valeur **position_courante**, celle-ci aboutira un moment à la valeur **position_cible**.

Notez que pour calculer la valeur suivante de **position_courante** (initialement mise à 0), il faut incrémenter la valeur actuelle de **position_courante** de la valeur **saut** et ensuite, si le résultat est plus grand ou égal à 100, calculer la position en faisant un modulo 100 de la valeur obtenue (ce qui donne à chaque fois une valeur **position_courante** entre 0 et 99). (Notez que l'on peut systématiquement faire le modulo 100 du résultat sans tester si **position_courante** est ou non supérieure à 100 pour obtenir sa bonne valeur).

Notez également, pour ne pas épuiser mon écureuil sans fin, que s'il atteint à nouveau le barreau `0`, j'arrête l'expérience sachant qu'il prendra toujours les mêmes barreaux sans jamais atteindre **position_cible** (la noisette).

À la fin, votre programme dira si oui ou non la noisette a été atteinte ou non.

En pratique, après avoir lu les deux valeurs **saut** et **position_cible**, votre programme affichera chaque valeur de **position_courante** sur une ligne différente à partir de la

seconde valeur (sauf la `position_courante` initiale qui vaut toujours 0). La dernière `position_courante` affichée sera soit 0 soit la dernière valeur de `position_courante` avant qu'elle n'ait la valeur de `position_cible`, si l'écureuil trouve la noisette. Votre programme terminera en affichant, sur une nouvelle ligne, le message donnant le résultat :

- `"Cible atteinte"` si l'écureuil a trouvé la noisette,
- `"Pas trouvée"` si l'écureuil est revenu en position 0 sans trouver la noisette.

Vous pouvez supposer que les valeurs lues sont bien des entiers qui respectent les consignes.

Exercice

Écrivez un code qui lit un nombre entier strictement positif `n` et affiche sur `n` lignes une table de multiplication de taille `n x n`, avec, pour *i* entre 1 et `n`, les n premières valeurs multiples de i strictement positives sur la i*ème* ligne.
Ainsi, les `n` premiers multiples de 1 strictement positifs (0 non compris) sont affichés sur la première ligne, les `n` premiers multiples de 2 sur la deuxième, et cætera.

Exemple 1

Avec la valeur lue suivante :

```
3
```

le résultat à afficher sera :

```
1   2   3

2   4   6

3   6   9
```

Exemple 2

Avec la valeur lue suivante :

```
10
```

le résultat à afficher sera :

```
 1   2   3   4   5   6   7   8   9  10

 2   4   6   8  10  12  14  16  18  20

 3   6   9  12  15  18  21  24  27  30

 4   8  12  16  20  24  28  32  36  40

 5  10  15  20  25  30  35  40  45  50

 6  12  18  24  30  36  42  48  54  60

 7  14  21  28  35  42  49  56  63  70

 8  16  24  32  40  48  56  64  72  80

 9  18  27  36  45  54  63  72  81  90

10  20  30  40  50  60  70  80  90 100
```

Exercice 3

On peut calculer approximativement le sinus d'un nombre x en effectuant la sommation des premiers termes de la série (une série est une somme infinie) :

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

où x est exprimé en radians et 3! désigne la factorielle de 3.

Écrire un programme qui lit une valeur flottante x en entrée et imprime une approximation de $sin(x)$

Cette approximation sera obtenue en additionnant successivement les différents termes de la série jusqu'à ce que la valeur du terme devienne inférieure (en valeur absolue) à une constante ϵ que l'on fixera à 10^{-6}.

Exemple 1

Avec les données lues suivantes :

```
0.8
```

le résultat à imprimer vaudra :

```
0.7173557231746032
```

Remarque : Du fait du manque de précision dans les calculs avec les nombres de type float, votre résultat pourra différer de celui indiqué ci-dessus. Ce n'est pas un problème, car UpyLaB acceptera toute réponse suffisamment proche du résultat attendu, avec une tolérance d'environ 1.0e-5.

Exercice 4

Écrire un programme qui lit un nombre entier strictement positif `n` et imprime une pyramide de chiffres de hauteur `n` (sur `n` lignes complètes, c'est-à-dire toutes terminées par une fin de ligne).

- La première ligne imprime un "1" (au milieu de la pyramide).
- La ligne i commence par le chiffre i % 10 et tant que l'on n'est pas au milieu, le chiffre suivant a la valeur suivante ((i+1) % 10).
- Après le milieu de la ligne, les chiffres vont en décroissant modulo 10 (symétriquement au début de la ligne).

Notons qu'à la dernière ligne, aucune espace n'est imprimée avant d'écrire les chiffres `0123...`

Exemple 1

Avec la donnée lue suivante :

```
1
```

le résultat à imprimer vaudra :

```
1
```

Exemple 2

Avec la donnée lue suivante :

```
2
```

le résultat à imprimer vaudra :

```
1
232
```

Exemple 3

Avec la donnée lue suivante :

```
10

```

le résultat à imprimer vaudra :

```
         1
        232
       34543
      4567654
     567898765
    67890109876
   7890123210987
  890123454321098
 90123456765432109

0123456789876543210
```

EXERCICE 6

Le Petit Prince vient de débarquer sur la planète U357, et il apprend qu'il peut y voir de belles aurores boréales !

La planète U357 a deux soleils : les étoiles E1515 et E666. C'est pour cela que les tempêtes magnétiques sont permanentes, ce qui est excellent pour avoir des aurores boréales.

Par contre, il y fait souvent jour, sauf bien évidemment quand les deux soleils sont couchés en même temps.

Heureusement pour nous, une journée U357 s'écoule sur 24 heures comme sur notre Terre, et pour simplifier, nous ne prendrons pas en

compte les minutes (on ne donne que les heures avec des valeurs entières entre 0 et 23).

Nous vous demandons d'aider le Petit Prince à déterminer les périodes de jour et de nuit.

Pour cela, vous allez dans un premier temps **écrire une fonction** `soleil_leve` qui, pour un soleil particulier, reçoit trois valeurs entières en argument pour la planète :

- l'heure de lever du soleil (entre 0 et 23)
- l'heure du coucher du soleil (entre 0 et 23)
- l'heure actuelle (entre 0 et 23)

et qui renvoie une valeur booléenne vraie si le soleil est levé sur la planète à l'heure donnée en troisième argument et fausse, s'il est couché. On considérera que si l'heure actuelle est égale à l'heure de lever, il fait déjà jour, et si elle est égale à l'heure de coucher, il fait déjà nuit.

On supposera que chacun des soleils ne se lève et ne se couche au plus qu'une seule fois par jour.
Il est toutefois possible que le lever ait lieu après l'heure du coucher, ce qui signifie dans ce cas que le soleil est levé au début de la journée, puis qu'il se couche, puis qu'il se lève à nouveau plus tard dans la journée.
Enfin, si l'heure du lever est la même que l'heure du coucher :

- soit toutes deux valent 12, cela signifie que le soleil ne se lève pas de la journée,
- soit toutes les deux valent 0, cela signifie que le soleil ne se couche pas de la journée.

Exemple 1

L'appel suivant de la fonction :

```
soleil_leve(6, 18, 10)
```

doit retourner

```
True
```

L'exercice 4.3 d'UpyLaB vous propose une fonction assez classique à rédiger, qui teste si un nombre `n` , reçu en paramètre, est premier.

Par définition, un nombre entier naturel est premier s'il est divisible (entièrement) uniquement par deux nombres différents, par 1 et par lui-même. 1 n'est pas premier. Le plus petit nombre premier est 2. Tous les nombres premiers suivants sont impairs puisque tous les nombres pairs sont divisibles par 2.

Il faut rédiger une fonction booléenne (dont le résultat est `True` ou `False`), qui reçoit un entier positif et renvoie `True` si et seulement si n est un nombre `premier.`

MODULO

Dans la fonction `premier,` il faut vérifier que `n` n'est divisible par aucun des nombres entiers à partir de 2 et strictement inférieurs à lui-même. Pour cela, notons que l'opérateur modulo peut être utile : le test `n % i == 0` est vrai si `i` divise `n` entièrement.

NB : Il est toutefois intéressant de constater qu'il n'est en fait nécessaire de vérifier cette condition que pour les nombres entiers qui sont inférieurs ou égaux à la racine carrée du nombre `n`. Si vous souhaitez rendre votre code plus efficace en réduisant le nombre d'instructions exécutées, la fonction `sqrt` du module `math` s'avèrera utile ici.

EXERCICE 7

Écrire une fonction booléenne `premier(n)` qui reçoit un nombre entier positif `n` et qui renvoie `True` si `n` est un nombre premier, et `False` sinon.

Ensuite, **écrire un programme** qui lit une valeur entière `x` et affiche, grâce à des appels à la fonction `premier`, tous les nombres premiers **strictement inférieurs** à `x`, chacun sur sa propre ligne.

Exemple 1

Avec la donnée lue suivante :

```
7
```

le résultat à imprimer vaudra donc

```
2
3
5
```

Écrire une fonction `alea_dice(s)` qui génère trois nombres (pseudo) aléatoires à l'aide de la fonction `randint` du module `random`, représentant trois dés (à six faces avec les valeurs de 1 à 6), et qui renvoie la valeur booléenne `True` si les dés forment un 421, et la valeur booléenne `False` sinon.

Le paramètre `s` de la fonction est un nombre entier, qui sera passé en argument à la fonction `random.seed` au début du code de la fonction. Cela permettra de générer la même suite de nombres aléatoires à chaque appel de la fonction, et ainsi de pouvoir tester son fonctionnement.

Exemple 1

L'appel suivant de la fonction :

```
alea_dice(1)
```

doit retourner :

```
False
```

EXERCICE 8

Écrire une fonction `rendre_monnaie` qui reçoit en paramètre un entier `prix` et cinq valeurs entières `x20, x10, x5, x2` **et** `x1`, qui représentent le nombre de billets ou de pièces de chaque valeur que donne un client pour l'achat d'un objet dont le prix est mentionné.

La fonction doit renvoyer cinq valeurs, représentant le nombre de billets et pièces de chaque sorte qu'il faut rendre au client, dans le même ordre que précédemment. Cette décomposition doit être faite en rendant le plus possible de billets et pièces de grosses valeurs.

Si la somme d'argent avancée par le client n'est pas suffisante pour effectuer l'achat, la fonction retournera cinq valeurs `None`.

Exemple 1

L'appel suivant de la fonction :

```
rendre_monnaie(38, 1, 1, 1, 1, 1)
```

doit retourner :

```
(0, 0, 0, 0, 0)
```

EXERCICE 8

Écrire une fonction `somme(a, b)` qui retourne la somme de deux valeurs entières `a` et `b`. Par défaut, la valeur de `a` est 0 et la valeur de `b` est 1.

`rac_eq_2nd_deg(a, b, c)` qui reçoit trois paramètres de type float correspondant aux trois coefficients de l'équation du second degré $ax^2 + bx + c = 0$, avec *a* différent de 0, et qui renvoie la ou les solutions s'il y en a, sous forme d'un tuple.

Exemple 1

L'appel suivant de la fonction :

```
rac_eq_2nd_deg(1.0,-4.0,4.0)
```

```

```

doit retourner :

```
(2.0,)
```