M.P.S.I.2 2015 **Jeudi novembre** 2016 Charlemagne א_1PHO01_א

MPSI 2/2015 **SUITES DE CUNNINGHAM** 1pho01

On appelle "*suite de Cunningham*" toute liste d'entiers naturels $[p_0,\ p_1, \dots p_N]$ dont tous les termes sont premiers et vérifiant $p_{n+1} = 2.p_n + 1$ pour tout n. On appelle "*graine*" le premier terme p_0.

I- 1) Donnez la suite de Cunningham de graine 2. [1 pt.]

I- 2) Donnez la suite de Cunningham de graine 41. [1 pt.]

I- 3) Donnez la suite de Cunningham de graine 89. [1 pt.]

I- 4) Trouvez une formule générale pour le n^{ieme} terme d'une suite de Cunningham de premier terme p, du type $p_n = 2^n.p_0 - 1$ *(mais ce n'est pas ça...)*. [2 pt.]

I- 5) Pourquoi ne construit-on pas de suites de Cunningham suivant le critère $p_{n+1} = 3.p_n + 1$? [1 pt.]

Le but de cet exercice est de construire des suites de Cunningham, d'en mesurer la longueur, et de trouver des suites de Cunningham le plus longues possibles.

II- 1) On suppose dans un premier temps qu'on dispose déjà d'une fonction qui teste si un entier naturel n est premier : `est_prem(n)` retourne `True` ou `False` suivant que n est premier ou non. On suppose aussi qu'on a une liste `Liste_prem` des Nb nombres premiers dans l'ordre *(avec Nb bien grand)*.
Complétez ce script Python pour qu'il construise pour p donné *(a priori)* premier la suite de Cunningham de graine p : [2 pt.]

```
def Cunningham(p) :
....L =[p] :
....while est_premier(***) :
........p = ***
........L.append(***)
....
```

Modifiez le pour qu'il retourne une liste vide si p n'est pas premier. [1 pt.]

II- 2) En utilisant la procédure que vous venez de créer et la liste `Liste_prem`, affichez les suites de Cunningham de graine dans `Liste_prem`, dont la taille dépasse 4. [2 pt.]

En utilisant la procédure que vous venez de créer et la liste `Liste_prem`, affichez les suites de Cunningham de graine dans `Liste_prem`, dont la taille soit la plus grande possible. [2 pt.]

II- 3) Cadeau en passant : 63 439, 126 839, 253 679, 507 359, 1 014 719, 2 029 439. Question bonus : pourquoi tant de nombres se terminant par 9 dans les suites de Cunningham ? [2 pt.]

III- 1) Il est temps de crée nous même la fonction `test_prem` et de créer `Liste_prem`.
Corrigez le proposition suivante : [3 pt.]

```
def test_prem(n) :
....for k in range(1, n) :
........if int(n/k) == n/k :
............print(False)
....print(True)
```

Pour opitimiser ce script : montrez que si un entier n n'est pas premier, alors il a au moins un diviseur plus petit que $\sqrt{n}$. Optimisez le script. [3 pt.]

III- 2) Pour ce qui est de `Liste_prem`, un élève propose :

```
Liste_prem = [2]
p = 3
while len(Liste_prem)<1000 :
....if est_premier(p) :
........Liste_prem.append(p)
....p = p+2
```

Expliquez son programme. 2 pt.

Améliorez le pour que l'on saisisse d'abord le nombre de termes souhaité. 1 pt.

Améliorez le aussi pour faire patienter l'utilisateur en lui indiquant par un petit message chaque fois que vous avez trouvé deux cent nouveaux nombres premiers. 1 pt.

III- 3) Trouvez en un bien plus efficace qui profite du fait qu'un nombre p est premier si et seulement si il n'est divisible par aucun des nombres premiers qui le précèdent. 4 pt.

Un cadeau : la liste `Liste_prem`.

2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,107,109,113,127,131,137,139,149,151,157,
163,167,173,179,181,191,193,197,199,211,223,227,229,233,239,241,251,257,263,269,271,277,281,283,293,307,311,313,317,
331,337,347,349,353,359,367,373,379,383,389,397,401,409,419,421,431,433,439,443,449,457,461,463,467,479,487,491,499,
503,509,521,523,541,547,557,563,569,571,577,587,593,599,601,607,613,617,619,631,641,643,647,653,659,661,673,677,683,
691,701,709,719,727,733,739,743,751,757,761,769,773,787,797,809,811,821,823,827,829,839,853,857,859,863,877,881,883,
887,907,911,919,929,937,941,947,953,967,971,977,983,991,997,1009,1013,1019,1021,1031,1033,1039,1049,1051,1061,1063,
1069,1087,1091,1093,1097,1103,1109,1117,1123,1129,1151,1153,1163,1171,1181,1187,1193,1201,1213,1217,1223,1229,
1231,1237,1249,1259,1277,1279,1283,1289,1291,1297,1301,1303,1307,1319,1321,1327,1361,1367,1373,1381,1399,1409,
1423,1427,1429,1433,1439,1447,1451,1453,1459,1471,1481,1483,1487,1489,1493,1499,1511,1523,1531,1543,1549,1553,
1559,1567,1571,1579,1583,1597,1601,1607,1609,1613,1619,1621,1627,1637,1657,1663,1667,1669,1693,1697,1699,1709,
1721,1723,1733,1741,1747,1753,1759,1777,1783,1787,1789,1801,1811,1823,1831,1847,1861,1867,1871,1873,1877,1879,
1889,1901,1907,1913,1931,1933,1949,1951,1973,1979,1987,1993,1997,1999,2003,2011,2017,2027,2029,2039,2053,2063,
2069,2081,2083,2087,2089,2099,2111,2113,2129,2131,2137,2141,2143,2153,2161,2179,2203,2207,2213,2221,2237,2239,
2243,2251,2267,2269,2273,2281,2287,2293,2297,2309,2311,2333,2339,2341,2347,2351,2357,2371,2377,2381,2383,2389,
2393,2399,2411,2417,2423,2437,2441,2447,2459,2467,2473,2477,2503,2521,2531,2539,2543,2549,2551,2557,2579,2591,
2593,2609,2617,2621,2633,2647,2657,2659,2663,2671,2677,2683,2687,2689,2693,2699,2707,2711,2713,2719,2729,2731,
2741,2749,2753,2767,2777,2789,2791,2797,2801,2803,2819,2833,2837,2843,2851,2857,2861,2879,2887,2897,2903,2909,
2917,2927,2939,2953,2957,2963,2969,2971,2999,3001,3011,3019,3023,3037,3041,3049,3061,3067,3079,3083,3089,3109,
3119,3121,3137,3163,3167,3169,3181,3187,3191,3203,3209,3217,3221,3229,3251,3253,3257,3259,3271,3299,3301,3307,
3313,3319,3323,3329,3331,3343,3347,3359,3361,3371,3373,3389,3391,3407,3413,3433,3449,3457,3461,3463,3467,3469,
3491,3499,3511,3517,3527,3529,3533,3539,3541,3547,3557,3559,3571

Premiers exemples.

2015_1pho01_

En s'aidant de la liste des nombres premiers donnée en appendice :

2	5	11	23	47		95n'est pas premier
41	83	167				335n'est pas premier
89	179	359	719	2 879		5 759n'est pas premier

Comme la liste des nombres premiers s'arrête trop tôt, on doit tester nous même 5 759.
Ce n'est visiblement pas un multiple de 2, 3, 5, 11. Sa division par 7 ne tombe pas juste. En revanche, le calcul donne : $5\ 759 = 13 \times 443$.

Si on note p le premier terme, les suivants sont

$2.p+1$	$2.(2.p+1)+1 = 4.p+3$	$2.(4.p+3)+1 = 8.p+7$	$2.(8.p+7)+1 = 16.p+15$

On conjecture vite la formule $p_n = 2^n.p + (2^n - 1)$ qu'on va prouver par récurrence. On l'a initialisée. On la suppose vraie à rang n quelconque donné. On calcule alors :
$p_{n+1} = 2.p_n + 1 = 2.(2^n.p + 2^n - 1) + 1 = 2^{n+1}.p + 2^{n+1} - 2 + 1 = 2^{n+1}.p + 2^{n+1} - 1$

On pouvait aussi prouver directement que la suite $(p_n + 1)$ était une suite géométrique de raison 2 : $p_{n+1} + 1 = (2.p_n + 1) + 1 = 2.p_n + 2 = 2.(p_n + 1)$. Il ne restait plus qu'à préciser la valeur de son premier terme et à revenir à $p_n = 2^n.(p_0 + 1) - 1$.

Si on veut une suite de Cunningham répondant qau critère $p_{n+1} = 3.p_n + 1$, alors on a les termes suivants :

p	$3.p+1$	$9.p+4$	$27.p+13$

Comme p est premier, il est impair (sauf $p = 2$), alors $3.p + 1$ est pair et n'est déjà plus premier.

Procédure Cunningham.

2015_1pho01_

On prend en variable un entier p. On a le droit d'en modifier la valeur au sein de la procédure, ça ne le modifiera pas à la sortie. On peut donc remplacer p par $2.p + 1$ autant de fois qu'il faut, pour calculer donc les termes successifs. On place les nombres au fur et à mesure dans une liste qu'on avait initialisée à p. Combien de fois de suite remplace-t-on p par $2.p + 1$? Tant que $2.p + 1$ est un nombre premier.

```
def Cunningham(p) :
....L = [p] :
....while est_premier(***) :
........p = ***
........L.append(***)
....
```

```
def Cunningham(p) :
....L = [p] :
....while est_premier(2*p+1) :
........p = 2*p+1
........L.append(p)
....return(L)
```

En sortie, on retourne la liste qui contient donc la graine p et ses successeurs forcément premiers.

Ce serait une erreur de mettre

```
....while est_premier(2*p+1) :
........p = 2*p+1
........L.append(2*p+1)
```

puisque la valeur de p a déjà été modifiée.

Si on veut s'assurer qu'on travaille bien avec un nombre premier :

```
def Cunningham(p) :
....if est_premier(p) :
........L = [p] :
........while est_premier(2*p+1) :
```

```
............p = 2*p+1
............L.append(p)
....else :
........L = []
....return(L)
```
mais il y a d'autres structures de programme possibles.

Une fois qu'on a cette procédure et la liste :
```
for p in Liste_rem :
....Cunning = Cunningham(p)
....if len(Cunning) > 3 :
........print(Cunning)
```
Avec ceci, on va afficher les couples $(p,\ Cunningham(p))$, pour lesquels la longueur de la liste dépasse 4.
On évitera
```
for p in Liste_rem :
....if len(Cunningham(p)) > 3 :
........print(Cunningham(p))
```
qui va calculer deux fois `Cunningham(p)` quand la longueur de la liste dépasse 4. Et un programme informatique n'a pas plus de mémoire que ce qu'on lui indique ; ce n'est pas parce qu'il a déjà calculé `Cunningham(2431)` qu'il va dire "*je l'ai déjà, je ne le recalcule pas*" quand on lui redemande.
On pourra opter pour
```
LCunh = []
for p in Liste_rem :
....Cunning = Cunningham(p)
....if len(Cunning) > 3 :
........LCunh.append(Cunning)
print(LCunh)
```
qui met toutes les listes intéressantes dans une liste de listes et n'affiche ensuite ladite liste qu'à la fin.

Pour trouver une liste de longueur maximale parmi celles disponibles, la méthode est toujours la même : on parcourt la liste et chaque fois qu'on fait mieux, on mémorise le nouveau résultat à la place du précédent :
```
long_max = 0
Cunn_maximal = []
for p in Liste_prem :
....Cunning = Cunningham(p)
....LCunning = len(Cunning)
....if LCunning > long_max :
........Cunn_maximal = Cunning
........long_max = LCunning
print(Cunn_maximal)
```
Pour ceux qui n'ont pas peur de solliciter trop de fois les mêmes procédures *(on est en I.P.T., pas en informatique ?)* :
```
Cunn_maximal = []
for p in Liste_prem :
....if len(Cunningham(p)) >
len(Cunn_maximal) :
........Cunn_maximal = Cunningham(p)
print(Cunn_maximal)
```

Une question en passant : pourquoi tant de 9 ? _2015_1pho01_

On constate que de nombreuses suites de Cunningham contiennent des nombres se terminant par des 9. Pourquoi ? Si p se termine par un 9, alors $2.p+1$ se termine aussi par un 9. En revanche, si p se termine par un 7 par exemple, alors $2.p+1$ se termine par un 5 et ne peut donc pas être premier...
Déjà, on rappelle que p premier se termine par 1, 3, 5, 7 ou 9 *(certes, il reste à part le cas de $p=2$, mais c'est la seule graine paire autorisée).*

Au cas par cas :

p *mod* 10	$2.p+1$ *mod* 10	$4.p+3$ *mod* 10	$8.p+7$ *mod* 10	$16.p+15$ *mod* 10
1	3	7	5 fini	
3	7	5 fini		
5	1	3	7	5 fini
7	5 fini			
9	9	9	9	9

On voit donc que si un nombre premier se termine par autre chose que 9, il ne peut pas y avoir plus de quatre termes ensuite. Et encore, la ligne $p = 5$ *mod* 10 ne se trouve que quand la graine de la suite est 5 lui même.
Bref, pour qu'une suite de Cunningham autre que [2, 5, 11, 23, 47] ou sa descendante [5, 11, 23, 47] contienne plus que quatre termes, il faut que la graine p se termine par un 9.

Test de primalité et listes de nombres premiers. 2015_1pho01

Pour savoir si un nombre est premier, on teste si il a des diviseurs. On passe en revue les entiers k plus petits que lui *(un diviseur de n ne peut pas dépasser n)* et on regarde si l'un d'entre eux divise n.
Si ne serait ce qu'un k divise n, on sort tout de suite et on affirme sans crainte que n n'est pas premier (`return(False)`). Comme on vient de passer par un `return`, on oublie la suite de la procédure, qui ne sera pas lue ni exécutée.
Si aucun k n'a divisé n, alors on attaque enfin la dernière ligne de la procédure, et on sort en annonçant fièrement que n est premier (`return(True)`).

On propose à tort

```
def test_prem(n) :
....for k in range(1, n) :
........if int(n/k) == n/k :
............print(False)
....print(True)
```

On corrige :

```
def test_prem(n) :
....for k in range(2, n) :
........if int(n/k) == n/k :
............return(False)
....return(True)
```

Les erreurs sont multiples.
• On sort d'une procédure non pas en affichant quelquechose à l'écran, mais en rendant un résultat à exploiter (et à afficher si on y tient) : `return` et non `print`.
• Pour savoir si n est divisible par k, on oublie les bidouillages de calculette du type `int(n/k) == n/k`. On fait de l'informatique. On profite de ce que sait faire l'ordinateur proprement *(avec des entiers et pas des floats)* : `k%n == 0` *(en rappelant que* `n%k` *est le reste de la division euclidienne de n par k).*
• L'entier k est pris dans un `range` qui pourrait aller de 0 à $n-1$. Ce serait une erreur, puisque pour k nul, la division euclidienne n'a même pas de sens. Mais le `range(1, n)` n'est pas convenable non plus : on commence avec $k = 1$, et que n soit premier ou non, il est toujours divisible par 1. Avec un tel test, aucun nombre n'est premier. De même, il ne faut pas inclure n dans le range car sinon, on teste "est-ce-que n divise n ?" et c'est toujours vrai. Rappelons que `k in range(2, n)` fait parcourir à k les valeurs de 2 à $n-1$, ce qui est parfait.
On notera que pour n égal à 1, le range est vide. Le programme va prétendre à tort que 1 est premier. Mais dans notre contexte Cunninghamien, ce n'est pas grave, peu de $2.p+1$ sont égaux à 1.

Si un entier n n'est pas premier, il se décompose en produit $n = a.b$ avec a et b différents de n et 1. Nécessairement, un des deux est plus petit que $\sqrt{n}$ *(sinon, $a > \sqrt{n}$ et $b > \sqrt{n}$ entraîne $a.b > n$ et non pas $a.b = n$).*
Parmi tous les diviseurs de n, il y en a donc au moins un qui est plus petit que $\sqrt{n}$.
Pour tester si n est premier, il suffit donc d'explorer la liste des entiers de 2 à $\sqrt{n}$, ce qui réduit grandement le champ.
D'autre part, une fois qu'on a testé le diviseur possible 2, on ne teste que les diviseurs impairs, de la forme $k = 2.q+1$ avec $q \leqslant \dfrac{\sqrt{n}-1}{2}$. On propose donc le test optimisé :

```
def test_prem(n) :
....if n%2 == 0 :
```

```
.........return(n == 2)
....for q in range(int((sqrt(n)+1)/2)+1) : #entiers q de 0 à ⌊(√n−1)/2⌋ inclus
........if n%(2*q+1) == 0 :
............return(False)
....return(True)
```

Oui, on a écrit `if n%2 == 0 : return(n != 2)` pour ne pas commettre une erreur : 2 *est pair, mais est quand même premier...*
Donc, quand on tombe sur un nombre pair, on retourne le booléen n == 2 qui est automatiquement évalué. Si on a un nombre pair autre que 2, on retourne donc `False`, et dans le cas n=2, on retourne `True`.

Avec le petit script

```
Liste_prem = [2]
p = 3
while len(Liste_prem)<1000 :
....if est_premier(p) :
........Liste_prem.append(p)
....p = p+2
```

On crée petit à petit une liste de nombres premiers. On y met 2, seul nombre premier pair. Ensuite, on ne regarde que des nombres impairs *(on commence à 3 et on avance de 2 en 2)*. Si l'entier impair p est premier, on l'ajoute à la liste. Et dans tous les cas, on avance de 2 en 2.
Cette boucle d'incrémentation est effectuée tant que la liste ne contient pas encore mille termes.
On ne pouvait pas utiliser de boucle for, car on ne sait pas a priori jusqu'où il faut aller pour détecter le millième nombre premier...
Au fait, expliquer en commenter un prgramme, ce n'est pas écrire comme le font les élèves "gentillets"
on crée une liste qui s'appelle Liste_prem, dans laquelle on met 2 ; on crée une variable p qui vaut 3 ; on fait une boucle tant que la longueur de Liste_prem est plus petite que 1000 ; dans cette boucle, si p est premier, on allonge la liste Liste_prem en y mettant la valeur p ; on remplace p par $p+2$.
Une telle description (digne de la description d'un film pour les non-voyants sur un D.V.D.) ne fait que paraphraser ler programme mais n'explique en rien ce qu'il fait vraiment. Votre rôle de futur ingénieur est d'expliquer et décrire, et non pas d'être une version ancienne de Google-traductor tentant de traduire un poëme.

On va encore créer petit à petit une liste de nombres premiers qu'on va initialiser à 2 et 3. Ensuite, on va avancer de 2 en 2 pour ne tester que des nombres impairs. Pour chacun d'entre eux, on va partir de l'a priori favorable qu'il est premier, mais on va quand même regarder si il n'est pas divisible par un des nombres premiers de la liste déjà établie. Si aucun des nombres ne l'a divisé, il n'a aucun diviseur premier, il est premier, on l'ajoute à la liste. Sinon, on ne l'ajoute pas, tant pis.

```
Liste_prem = [2, 3] #on met déjà ceux là
p = 3 #on va avancer parmi les nombres impairs
while len(Liste_prem) < 1000 : #on le fait tant que la liste n'est pas assez longue
....p = p+2 #on avance donc parmi les nombres impairs
....Premier = True #on suppose a priori qu'il est premier
....for d in Liste_prem : #on va tester les diviseurs premiers possibles
........if p%d == 0 : #si il a un diviseur
............Premier = False #on bascule : il n'est pas premier
....if Premier : #si il n'a eu aucun diviseur premier
........Liste_premier.append(p) #on peut l'ajouter à la liste
```

On pourra insérer au bon endroit (au moment de l'`append`) :
`if len(Liste_prem)%200 == 0 : print(Liste_prem[-1], len(Liste_prem))`

MPSI 2/2015

DIGITS

1pho2

On rappelle que les nombres sont codés en binaire. On va juste regarder ici un chiffre [1], codé avec quatre bits : Unités, Deuxaines, Quatraines et Huitaines. Par exemple :

	U	D	Q	H
5	1	0	1	0
7	1	1	1	0

. Chaque bit vaut `0` ou `1`, que l'on assimile automatiquement à `False` et `True`.

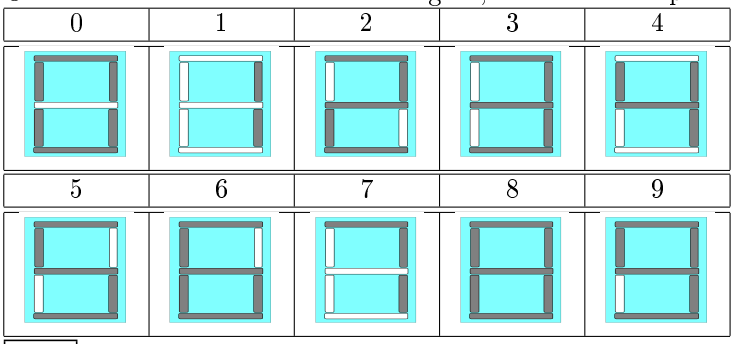
◇1 Quel chiffre est codé par la fonction logique `U and D and not(Q or H)` ? 1 pt.

◇2 Comment coder `6` en fonction logique ? 1 pt.

◇3 Qui est codé par `U×H×(1-D)×(1-Q)` ? 1 pt.

◇4 Ecrivez un script qui vérifie que la quadruplet `U`, `D`, `Q`, `H` code bien un chiffre décimal *(par exemple le binaire `1101` est le décimal 13, on doit répondre `False`)*. 1 pt.

On veut afficher alors le chiffre en digital, comme sur les pendules et les vieilles montres.



◇5 Ecrivez une fonction logique qui se charge d'allumer ou non la barre horizontale du haut en fonction des quatre entrées binaires `U`, `D`, `Q` et `H`. 3 pt.

◇6 Même question pour la barre du milieu. 2 pt.

MPSI 2/2015

EVACUATION D'URGENCE

1pho2

On se propose de modéliser et simuler l'évacuation d'urgence d'une salle de type "*salle des fêtes*", afin de voir comment disposer au mieux les issues de secours par exemple. Des études ont montré aussi que la présence d'un poteau à proximité d'une issue de secours permet de scinder le flux et d'améliorer la fluidité.

La salle sera donc un rectangle avec des coordonnées réelles X *(et Y)* de `0` à `Xmax` *(de `0` à `Ymax`)*, sur lequel seront placées initialement aléatoirement `NbP` personnes. On disposera de `NbS` sorties situées au niveau des murs de la salle. A partir de l'instant initial *(l'alerte)*, chaque individu va se diriger vers la sortie la plus proche à une vitesse qui lui est propre, à chaque étape de temps. On visualisera ensuite si il se crée des bouchons, si il faut que certaines personnes se réorientent vers des sorties moins bouchées.

Format des données :

nom de la variable	type	exemple
`personnes`	liste de couples, de longueur `NbP`	`[[10, 4], [2, 5], [6, 12]]`
`vitesse`	liste de nombres, de longueur `NbP`	`[2, 2, 1]`
`sorties`	liste de couples, de longueur `NbS`	`[[0, 8], [12, 0]]`

[1] rappel : les nombres en décimal de 0 à l'infini sont écrits avec des chiffres de 0 à 9, de même que les mots sont écrits avec des lettres

♠1 Ecrivez un script qui crée la liste personnes, placées aléatoirement dans la salle. Optimisez ce script pour qu'il n'y ait pas deux personnes au même endroit. 3 pt.

♠2 Ecrivez un script qui vérifie que les sorties de la liste sorties sont bien sur le bord de la salle. 2 pt.

♠3 Ecrivez un script qui crée pour chaque personne de la liste `personnes` sa vitesse, avec les probabilités suivantes :

0	1	2	3	4
impossible	20%	30%	40%	10%

2 pt.

♠4 Ecrivez un script qui pour chaque `personne[k]` de la liste `personnes` détermine la sortie la plus proche et la fait avancer d'un vecteur de longueur `vitesse[k]` en direction de cette sortie. 4 pt.

♠5 Optimisez votre script pour qu'une personne qui arrive à la sortie soit effacée de la liste `personnes` *(et de la liste `vitesse`)*. 1 pt.

♠6 Ecrivez un script qui vérifie à un instant donné combien de personnes sont à distance inférieure ou égale à `dist` de chaque sortie *(et indique combien de personnes sont encore "trop loin" des sorties)*. 3 pt.

♠7 On considère que le placement aléatoire des personnes dans la pièce n'est pas une bonne simulation. On dispose de bases de données dont les tables sont issues d'observations de situations réelles.

champ	individu	abscisse	ordonnée	vitesse
type	string	integer	integer	integer

Tapez une requête pour savoir combien il y a de personnes dont la vitesse est moindre que 4. 1 pt.

Tapez une requête pour obtenir la liste des individus qui sont dans le quart supérieur de la salle *(abscisse entre `0` et `Xmax/2` et ordonnée entre `0` et `Ymax/2`)*. 2 pt.

MPSI 2/2015 **HÖRNER** 1pho2

Un polynôme `P` est représenté par la liste de ses coefficients *(le coefficients `P[k]` est celui de X^k)*. L'algorithme de Hörner calcule les coefficients du polynôme "translaté" `P(X+a)`. Exemple `Horner([1, 2, 4], 2)` comprend $P = 1 + 2.X + 4.X^2$ et calcule $P = 1 + 2.(X + 2) + 4.(X + 2)^2$ et rend `[21, 18, 4]`.

```
def Horner(P, a) :
....for k in range(len(P)-1) :
........for i in range(len(P)-k-2, len(P-1) :
............P[i] = P[i]+a*P[i+1]
....return(P)
```

♣1 Justifiez que ce programme fait bien ce qu'il faut dans le cas `a=0`. 1 pt.

♣2 Justifiez que ce programme fait bien ce qu'il faut dans le cas `len(P)=4`. 3 pt.

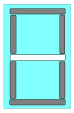
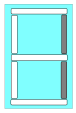
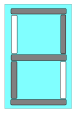
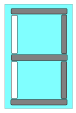
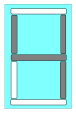
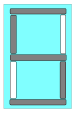
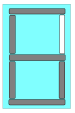
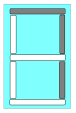
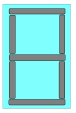
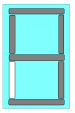
♣3 Justifiez le cas général. 6 pt.

Pour finir, que donne ceci : 3 pt.

```
import numpy as np
import mathplotlib.pyplot as plt
def f(t) :
....return(abs(np.sin(t*t)))
x = np.arange(0, 5, 0.1)
plt.plot(x, f(x))
plt.show()
```

Codage binaire. 2015_1pho2

On dresse un tableau plus précis :

	0	1	2	3	4	5	6	7	8	9
U	0	1	0	1	0	1	0	1	0	1
D	0	0	1	1	0	0	1	1	0	0
Q	0	0	0	0	1	1	1	1	0	0
H	0	0	0	0	0	0	0	0	1	1
										

On décode alors :

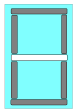
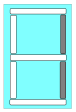
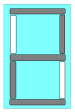
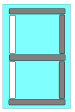
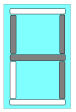
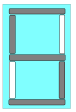
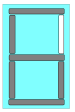
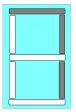
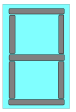
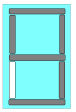
logique	binaire	décimal	
`U and D and not(Q or H)`	0011	3	$U \times D \times \overline{Q} \times \overline{H}$
`D and Q and not(U) and not(H)`	0110	6	$\overline{U} \times D \times Q \times \overline{H}$
`U and H and not(D or Q)`	1001	9	$U \times H \times \overline{D} \times \overline{Q}$

On veut détecter les nombres binaires supérieurs ou égaux à 10, puisque ce ne sont plus des chiffres.
Le plus simple :

```
def test(U, D, Q, H):
....nombre = U+2*D+4*Q+8*H
....return(nombre < 10)
```

Ce que l'on retourne est un booléen qui sera directement effectué et donnera donc `True` ou `False` , plutôt que de proposer `if nombre < 10 return(True) else return(False)`.

On poursuit en regardant quand on doit allumer la barre du haut :

	0	1	2	3	4	5	6	7	8	9
U	0	1	0	1	0	1	0	1	0	1
D	0	0	1	1	0	0	1	1	0	0
Q	0	0	0	0	1	1	1	1	0	0
H	0	0	0	0	0	0	0	0	1	1
										
barre du haut	oui	non	oui	oui	non	oui	oui	oui	oui	oui
barre du milieu	non	non	oui	oui	oui	oui	oui	non	oui	oui

Pour la barre du haut : on l'allume toujours, sauf pour le 1 et le 4.
Un test simple, mais c'est encore de la triche :

```
nombre = U+2*D+4*Q+8*H
test_14 = 1-(nombre-1)*(nombre-4)
```

Création des listes pour une évacuation d'urgence. _2015_1pho2_

On crée des couples `[x, y]` avec `x` entre `0` et `Xmax` et `y` entre `0` et `Ymax`, et on met en boucle avec `append`. Ne pas oublier de charger le module `random` [2].

```
personnes = []
for k in range(NbP) :
....x = randrange(1, Xmax) #on commence à 1 pour ne pas incruster des gens dans
les murs
....y = randrange(1, Ymax)
....personnes.append([x, y])
```

Attention, c'est bien une liste `[x, y]` qu'on doit `append`er, et non pas deux éléments `x` et `y`, sinon on crée `[10, 4, 2, 5, 6, 12]` au lieu de `[[10, 4], [2, 5], [6, 12]]`.

Pour s'assurer d'avoir des points tous distincts, on a plusieurs possibilités :

```
for k in range NbP :
....x = randrange(1, Xmax)
....y = randrange(1, Ymax)
....while [x, y] in personnes :
........x = randrange(1, Xmax)
........y = randrange(1, Ymax)
....personnes.append([x, y])
```

On tire des coordonnées au hasard, et si elles sont déjà dans la liste, on recommence.

On peut aussi créer une grande liste des possibles et en extraire `NbP` éléments :

`possibles = [[i, j] for i in range(1, Xmax) for j in range(1, Ymax)]` #double boucle for

```
for k in range(NbP) :
....indice = randrange(len(possibles)) #n'oublions pas que le liste des possibles fond peu à peu
....choix = possibles.pop(indice) #on le sort de la liste des possibles
....personnes.append(choix) #on le colle dans la liste des personnes
```

Pour vérifier si les sorties sont sur le bord ($abscisse \in \{0, Xmax\}, ordonnée \in \{0, Ymax\}$).

```
def test(sorties) :
....for sor in sorties :
........x, y = sor[0], sor[1]
........if (0<x<Xmax) or (0<y<Ymax) :
............return(False)
....return(true)
```

Si on a traversé toutes les boucles sans que le test ait détecté un problème, la liste est valide.

Si on détaille plus, par crainte de tests en `a<b<c` :

`........if ((0<x) and (x<Xmax)) or ((0<y) and (y<Ymax)) :`

D'autres tests sont possibles, avec `x*(x-Xmax)` et `y*(y-Ymax)`.

Pour les vitesses ou mobilités de chaque personne :

[2] rappel : pour un script Python avec Solène, le réflexe est `from math import *`, avec moi, c'est `from random import *`

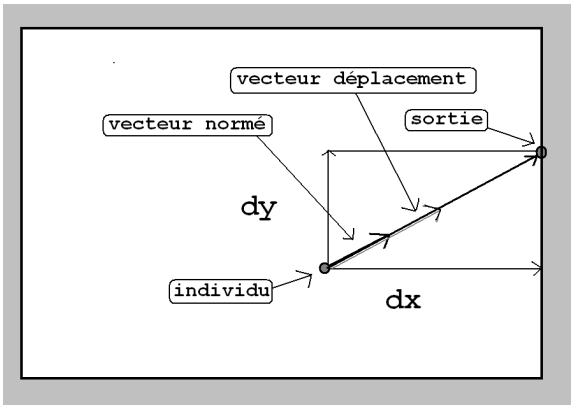
```
for k in range(len(personnes)) : #ou k in range(NbP) si tout va bien
....alea = randrange(10) : #on tire un entier entre 0 et 9
....if alea < 2 : #dans les cas 0 ou 1 (deux cas sur dix)
........vit = 1
....elif alea < 5 : #dans les cas 2, 3 et 4 (trois cas sur dix)
........vit = 2
....elif alea < 9 : #dans les cas 5, 6, 7 et 8 (quatre cas sur dix)
.........vit = 3
....else : #il ne reste que le cas 9 (un cas sur dix)
........vit = 4
....vitesse.append(vit)
```

On peut aussi créer une liste pondérée en probabilités :
`possibilites = [1, 1, 2, 2, 2, 3, 3, 3, 3, 4]`
puis tirer un des éléments de cette liste au hasard "*uniforme*"
`vit = possibilites(randrange(10))`

Déplacement vers la sortie. _2015_1pho2_

Pour chaque individu de `personnes`, on parcourt la liste des `sorties`, et pour chacune, on calcule la distance à laquelle l'individu se trouve de cette sortie. On regarde si cette distance est meilleure que le minimum possible. Si c'est le cas, on garde l'indice et on réactualise le minimum, sinon, on ne fait rien.
Pour ne pas se fatiguer, on cherche le minimum du carré de la distance, c'est pareil, et ça évite de faire appel à `sqrt` de manière totalement inutile.

```
sortants = [] #on verra plus loin à quoi va servir cette liste vide
for p in range(len(personnes)) :
....x, y = personnes[p][0], personnes[p][1] #on extrait les coordonnées
....DistanceMin = Xmin**Xmin+Ymin*Ymin #on initialise à la distance maximale (grande diagonale)
....index = 0 #on le mettra à jour plus tard
....for k in range(len(sorties)) :
........xs, ys = issue[k][0], issue[k][1] #on extrait les coordonnées
........Dist = (x-xs)*(x-xs)+(y-ys)*(y-yx) #on calcule le carré de la distance
........if Dist < DistanceMin : #on regarde si c'est mieux
............DistanceMin = Dist #on mémorise la nouvelle sortie
............index = k
```



Une fois qu'on connaît la sortie la plus proche d'indice `index`, on s'en rapproche justement. Il faut faire un peu de géométrie :
• on a un vecteur de direction `[dx, dy]` avec `dx = xs-x` et `dy = ys-y`
• on doit le renormer par `norme = sqrt(dx*dx+dy*dy)`
• on doit ensuite se déplacer avec la norme `vitesse[k]` suivant ce vecteur
• Il ne reste plus qu'à réactualiser les coordonnées de l'individu

```
....xs = sorties[index][0]
....ys = sorties[index][1]
....dx, dy = xs-x, ys-y
....norme = sqrt(dx*dx+dy*dy)
```

```
....dx, dy = dx/norme, dy/norme
....dx, dy = dx*vitesse[p], dy*vitesse[p]
....x, y = x+dx, y+dy
....personnes[k] = [x, y]
```

C'est ici qu'il va aussi falloir faire attention. Si la distance individu-sortie est plus courte que la distance parcourue par unité de temps *(appelée ici vitesse)*, il faut laisser sortir l'individu de la salle. On va donc reprendre

```
....norme = sqrt(dx*dx+dy*dy)
....if norme > vitesse[k] :
........dx, dy = dx/norme, dy/norme
........dx, dy = dx*vitesse[p], dy*vitesse[p]
........x, y = x+dx, y+dy
........personnes[k] = [x, y]
....else :
........sortants.append(k)
```

Une fois qu'on a étudié tous les individus, on efface ceux qui sont sortis de la salle des fêtes :

```
for k in sortants :
....personnes.pop(k)
....vitesses.pop(k)
```

Pourquoi ne pas avoir tapé

```
....norme = sqrt(dx*dx+dy*dy)
....if norme > vitesse[k] :
........dx, dy = dx/norme, dy/norme
........dx, dy = dx*vitesse[p], dy*vitesse[p]
........x, y = x+dx, y+dy
........personnes[k] = [x, y]
....else :
........personnes.pop(k)
........vitesses.pop(k)
```

Parce que ceci perturbe le parcours de la liste personnes. Si on détruit des éléments de la liste quand on la parcourt, on se perd, on saute au dessus des éléments situés juste après un élément éliminé, puisque le compteur s'incrémente ; et à la fin, on sollicite des éléments hors de la liste.

Et même, mon script

```
for k in sortants :
....personnes.pop(k)
....vitesses.pop(k)
```

va poser un problème, puisque une fois qu'on a effacé un terme, les indices qui suivent sont décallés. Il faut donc commencer par effacer les termes dont l'indice est le plus grand :

```
sortants.reverse() #pour renverser la liste
for k in sortants :
....personnes.pop(k)
....vitesses.pop(k)
```

Pour comprendre, imaginez une liste `[a, b, c, d, e, f]` *dans laquelle on doit éliminer les termes d'indices pythonien* `1`, `4` *et* `5` (c'est à dire `b`, `e` et `f`). *Avec une mise en boucle de* `pop` *d'indices croissants, on obtient*

méthode	`pop(1)`	`pop(4)`	`pop(5)`
liste	`[a, c, d, e, f]`	`[a, c, d, e]`	`erreur`

Si on effectue la mise en boucle des `pop` *en sens inverse :*

méthode	`pop(5)`	`pop(4)`	`pop(1)`
liste	`[a, b, c, d, e]`	`[a, b, c, d]`	`[a, c, d]`

Pour savoir combien de personnes sont dans le rayon de chaque sortie. _2015_1pho2_

On va prendre les sorties une à une, puis les personnes une à une *(double boucle `for`)*. On va mesurer la distance de la sortie à la personne, la comparer à `dist`.
Attention, il serait lourd de tester `sqrt((x-xs)*(x-xs)+(y-ys)*(y-ys))<dist`.
Un tel test fait appel à des calculs de produits *(pas trop lourds)*, mais aussi à un calcul de racine *(opération longue mettant en jeu des produits en cascade et des tests)*.
On préfèrera `((x-xs)*(x-xs)+(y-ys)*(y-ys))<dist*dist`. Comme toujours, il faut doubler son point de vue "*calcul mathématique*" du recul "*estimation algorithmique*" [3]

```
Liste_sorties = [] #on va l'agrandir peu à peu
distcarre = dist*dist #autant le faire une fois pour toutes
for sort in sorties :
....xs, ys = sort[0], sort[1] #on extrait les coordonnées de la sortie
....compteur = 0
........for individu in personnes :
............xi, yi = individu[0], individu[1] #on extrait les coordonnées de l'individu
............distancecarre = (xi-xs)*(xi-xs)+(yi-ys)*(yi-ys) #petit cacul
............if distance < distcarre :
................compteur = compteur+1
....Liste_sorties.append(compteur)
sum(Liste_sorties) #pour avoir le nombre d'individus presque sortis
```

Il y a quand même un problème dans ce script, dans l'estimation du nombre d'individus proches d'une sortie. En effet, un individu proche à la fois de deux sorties sera compté deux fois. Il faudrait donc faire d'abord la boucle sur individu dans personnes, puis sur les sorties ; et si le test est validé, on n'incrémente pas un compteur d'une unité, mais on le met à 1. Ainsi, on saura juste si individu est proche d'une ou plusieurs sorties, sans forcément savoir si c'est uune ou plusieurs.

Requètes S.Q.L. _2015_1pho2_

La base de données est ouverte, et on a a priori une seule table à consulter, appelée `table`. Les requètes seront en `SELECT ... FROM table`.
On va avoir des conditions du type `WHERE vitesse<4`.
La première requète ne demande pas de connaître le contenu des fiches/enregistrements sélectionnés, mais juste de les compter. On va donc utiliser `SELECT count(*)`.

```
SELECT COUNT(*) FROM table WHERE vitesse<4
```

Pour la seconde, la condition sera un booléen à deux tests : `(0<2*abscisse<Xmax) AND (0<2*ordinnee<Ymax)`

```
SELECT individu FROM table WHERE ((0<2*abscisse<Xmax) AND
(0<2*ordinnee<Ymax))
```

Pour l'instant, on a fait simple.

Algorithme de Hörner. _2015_1pho2_

On travaille donc sur une liste qu'on modifie et remodifie (boucles imbriquées).
La cas `a=0` doit transformer `P(X)` en `P(X+0)`, et donc ne pas le modifier.
La seule instruction qui modifie P est `P[i] = P[i]+a*P[i+1]` ; elle ne le modifie donc pas.

Dans le cas `len(P)=4`, on prend donc un polynôme à quatre coefficients `P=[P[0], P[1], P[2], P[3]]`.
C'est un polynôme de degré inférieur ou égal à `3`. Comme cette longueur est petite, on va pouvoir simuler sur le papier le comportement de l'algorithme.
Pour se simplifier la vie, on a nommer plutôt qu'indexer les quatre coefficients : `P = [A, B, C, D]`.
On part donc de `P(X)=A+B.X+C.X^2+D.X^3`.

[3] tout à coup, je m'interroge sur l'appellation "calcul mathématique" ; pour moi c'est presque une contradiction

On doit aboutir à `P(X+a)=A+B.(X+a)+C.(X+a)^2+D.(X+a)^3`, et, en développant, on doit obtenir la liste

`[A+a.B+a^2.C+a^3.D, b+2.C.a+3.D.a^2, c+3.D.a, D]`

On va suivre pas à pas les modifications de la liste dans la double boucle

```
....for k in range(3) :
........for i in range(2-k, 3) :
............P[i] = P[i]+a*P[i+1]
```

k	range de i		
k=0	range(2, 3) =[2]	P[2]=P[2]+a.P[3]	P=[A, B, C+a.D, D]
k=1	range(1, 3) =[1, 2]	P[1]=P[1]+a.P[2]	P=[A, B+a.C+a^2.D, C+a.D, D]
		P[2]=P[2]+a.P[3]	P=[A, B+a.C+a^2.D, C+2.a.D, D]
k=2	range(0, 3) =[0, 1, 2]	P[0]=P[0]+a.P[1]	P=[A+a.B+a^2.C+a^3.D, B+a.C+a^2.D, C+2.a.D, D]
		P[1]=P[1]+a.P[2]	P=[A+a.B+a^2.C+a^3.D, B+2.a.C+3.a^2.D, C+2.a.D, D]
		P[2]=P[2]+a.P[3]	P=[A+a.B+a^2.C+a^3.D, B+2.a.C+3.a^2.D, C+3.a.D, D]

Les coefficients binomiaux sont apparus comme par miracle...
...ou par récurrence.

Je vous laisse réfléchir à la démonstration générale.

Pour n plus grand que 5, on sait que l'écriture décimale de $n!$ se termine par pas mal de 0 *(exemple :* $25! = 15\ 511\ 210\ 043\ 330\ 985\ 984\ 000\ 000$*).*
Ecrivez un *(ou des)* script(s) Python qui prend en entrée n et retourne le nombre de 0 et le dernier chiffre avant tous les 0 *(exemple* 25 *donne* $(6,\ 4)$*).* 5 pt.

Pourquoi **des** *scripts ? Parce qu'il y en a des plus ou moins coûteux en temps, en taille mémoire... Par exemple, vous pourrez envisager le cas où vous avez une vieille version de Python qui ne manipule pas les entiers longs ; dans ce cas vous aurez un bonus.*

MPSI 2/2015 **LOGISTIQUE** 1pho3

La logistique n'est pas de la logique pour médecin légiste, mais la gestion des flux et la mise en adéquation des ressources et requêtes *(une chaine de magasin parle de logistique du dernier kilomètre pour ses livraisons à domicile...).* C'est aussi ainsi que le mathématicien Pierre-François Verhulst a dénommé l'étude d'évolution des populations au sens large *(populations humaines, animales, végétales, biologiques...).* On considère les évolutions soit régies par des équations différentielles *(temps "continu")*, soit par des suites *(temps "discret").* Bref, on va résoudre des équations différentielles, mais pas pour la physique.

I- 1) Le premier modèle est celui de Malthus [4] : $\boxed{y'_t = a.y_t}$ avec y_0 donné.
Cette proportionnalité traduit : plus il y a d'individus, plus il y a de naissances.
Donnez la vraie valeur de y_T. 1 pt.

I- 2) On considère qu'on ne sait pas résoudre l'équation. On applique alors la méthode approximative d'Euler : on découpe l'intervalle $[0,\ T]$ en n intervalles $(t_k = k.T/n)$, et à chaque pas, on calcule $Y_{t_{k+1}} - Y_{t_k} = a.Y_{t_k}.dt$ avec $dt = T/n$. Donnez alors la formule trouvée pour Y_T. 1 pt.

I- 3) Donnez le développement limité de $Y_T - y_T$ sous la forme $\dfrac{\alpha}{n} + \dfrac{\beta}{n^2} + o\Big(\dfrac{1}{n^2}\Big)_{n\to+\infty}$. 4 pt.

I- 4) On améliore la méthode d'Euler. $y'_t = a.y_t$ donc $y"_t = a.y'_t = a^2.y_t$, et on utilise le développement limité d'ordre 2 : $y_{t+dt} = y_t + y'_t.dt + \dfrac{y"_t}{2}.(dt)^2 + o(dt^2)$, on pose donc cette fois $Y_{t_{k+1}} = Y_{t_k} + a.Y_{t_k}.dt + \dfrac{a^2.Y_{t_k}}{2}.(dt)^2$. Montrez alors que la différence à l'instant T entre la solution obtenue et la vraie solution est équivalente à $\dfrac{a^3.T^3}{6.n^2}.e^{T.a}$ quand n tend vers l'infini. 5 pt.

I- 5) Ecrivez une procédure `Python` qui prend en entrée y_0, a, T et n et donne la liste des $n+1$ couples `[abscisse, ordonnée]` par la seconde méthode. 3 pt.

II- 1) Le second modèle est celui de Verhulst : $\boxed{y'_t = a.y_t - b.(y_t)^2}$ avec y_0 donné.
Le facteur $b.(y_t)^2$ *correspond à la probabilité que deux individus se rencontrent, le signe moins exprime une concurrence pour les ressources : une rencontre entre individus se traduit par un combat pouvant s'achever de manière négative.*
Montrez qu'il existe une solution constante non nulle. 1 pt.

II- 2) Donnez une primitive de $u \longmapsto \dfrac{1}{a.u - b.u^2}$, que l'on notera F *(pensez aux éléments simples).* 2 pt.

II- 3) Montrez que $t \longmapsto F(y_t) - t$ est une "***intégrale***" de notre équation différentielle *(c'est à dire une quantité qui reste constante au fil du temps quand y est une solution de l'équation différentielle).* 2 pt.

---

[4]

Thomas Robert Malthus	1766-1834	économiste anglais	Warder Clyde Allee	1894-1980	zoologiste américain
Pierre-François Verhulst	1804-1849	mathématicien belge	Voto Volterra	1860-1940	mathématicien italien

II- 4) Exprimez alors y_t en fonction de t, et montrez qu'il tend vers a/b quand t tend vers l'infini. [4 pt.]

II- 5) On donne $a = 2$, $b = 1$. Ecrivez un script `Python` *(pouvant utiliser les modules* ***math*** *,* ***numpy*** *,* ***pyplot*** *)* qui trace sur un même graphe les solutions pour y_0 valant 1, 2, 3 et 4. [4 pt.]

III- 1) Le troisième modèle est celui de ALLEE : $\boxed{y'_t = y_t.\left(1 - \frac{y_t}{K}\right).\left(\frac{y_t - c}{K}\right)}$ avec y_0 donné.

La formule $y'_t = y_t.(1 - \frac{y_t}{K})$ correspondrait au modèle de Verhulst, mais le correctif $(\frac{y_t - c}{K})$ indique que si la population est sous un seuil critique c, alors il y a trop peu de rencontres pour qu'elle survive, et elle doit encore décroître (et s'éteindre ?).

Ecrivez un programme qui prend en entrée y_0, K, c, T et n et retourne la liste des $n+1$ valeurs aux instants $k.T/n$ *(k de* 0 *à n)*, en suivant la méthode de résolution approximative d'Euler. [4 pt.]

IV- 1) Le quatrième modèle est celui de VOLTERRA : $\boxed{\begin{array}{rcrcl} x'_t & = & a.x_t & - & b.x_t.y_t \\ y'_t & = & -c.y_t & + & d.x_t.y_t \end{array}}$

Les proies de loi x_t se reproduisent $a.y_t$, mais pâtissent des rencontrent avec les prédateurs $-b.x_t.y_t$; les prédateurs de loi y_t se morfondent et meurent $-c.y_t$ et ce n'est que par rencontres avec des proies qu'ils se développent $+d.x_t.y_t$).

Déterminez la solution constante non nulle $(x_\infty,\ y_\infty)$ *(la notation n'induit pas que ce soit la limite des solutions quand t tend vers l'infini).* [1 pt.]

IV- 2) Montrez que $t \longmapsto d.x_t + b.y_t - c.\ln(x_t) - a.\ln(y_t)$ est une intégrale du système *(on admettra que x_t et y_t restent strictement positifs).* [2 pt.] *La définition de "intégrale de l'équation différentielle" a été donnée plus haut, relisez que diable !*

IV- 3) On décide de traiter le cas particulier $(a,\ b,\ c,\ d) = (2/3,\ 4/3,\ 1,\ 1)$ et $(x_0,\ y_0) = (1,\ 1)$. Montrez pour tout x : $x - \ln(x) \geqslant 1$. Déduisez que y reste bornée. [3 pt.]

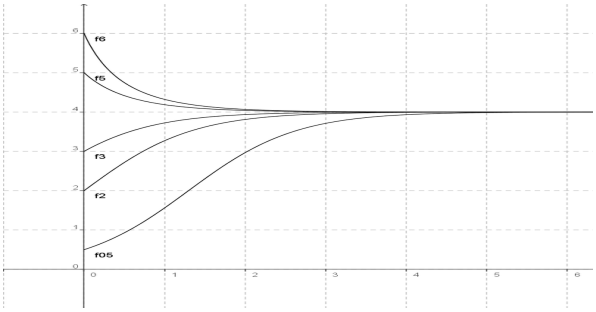
IV- 4) Déterminez le minimum sur $\mathbb{R}^{+*}$ de $y \longmapsto \dfrac{4.y - 2.\ln(y)}{3}$. Déduisez que x est bornée. [2 pt.]

IV- 5) Montrez que pour tout x, il y a au plus deux valeurs de y possibles. Montrez que pour tout y il y a au plus deux valeurs de x possibles. [3 pt.]

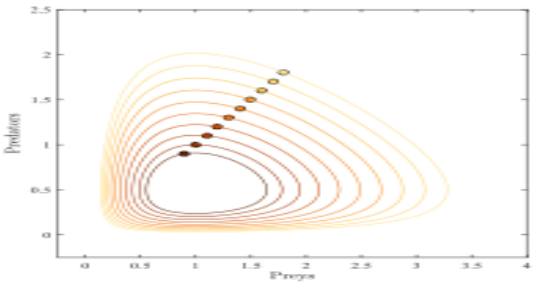
IV- 6) Montrez que quand y est maximum, alors x vaut 1 et y est solution de $2.y - \ln(y) = 2$. [2 pt.]

Ecrivez un script `Python` qui détermine par dichotomie une valeur approchée à 10^{-2} près de chacune de ces solutions. [3 pt.]

IV- 7) On peut montrer que les solutions $t \longmapsto (x_t,\ y_t)$ sont périodiques. Montrez en intégrant $\dfrac{y'_t}{y_t} + \dfrac{c}{d}$, déduisez que la valeur moyenne de x sur une période est x_∞ définie plus haut. [3 pt.]



Verhulst



Volterra

Factorielle. 2015_1pho3_

La première méthode, peut être couteuse, va consister à calculer d'abord $n!$, puis à le faire "fondre" en comptant les 0, et extraire ensuite modulo 10 le chiffre des unités.

```
def factorielle(n) : #du classique
....fact = 1
....for k in range(n) :
........fact = fact*(k+1)
....return(fact)
def bout_de_factorielle(n) :
....f = factorielle(n) #on la calcule
....compte_zero = 0 #on initialise un compteur
....while (f%10) == 0 : #tant que f se termine par un 0
........f = f/10 #on l'efface
........compte_zero = compte_zero+1 #on compte le 0
....chiffre = f%10 #c'est fini, on regarde le dernier chiffre
....return(compte_zero, chiffre) #on donne les deux réponses
```

Si on ne dispose pas d'entiers longs pour calculer explicitement la factorielle, on doit se débrouiller avec la décomposition en produit de facteurs premiers allégée.
On décompose $n! = 2^a.3^b.5^c.7^d.11^e \dots$ *(exemple :* $12! = 2^{10}.3^5.5^2.7.11$*)*.
Comment trouver le nombre de 0 à la fin ? Avec les facteurs 10. L'exposant de 5 étant plus petit que celui de 2, c'est lui qui donne le nombre de 0 : $n = 10^c.(2^{a-c}.3^b.7^d.11^e \dots)$ *(exemple :* $12! = (2^8.3^5.7.11).100 = (4790016).100$*)*.
On va donc devoir déterminer l'exposant de 5 dans $n!$ et en fait dans chaque facteur k de n. Ensuite, on regarde juste le chiffre des unités de $(2^{a-c}.3^b.7^d.11^e \dots)$, qui ne peut pas être un 0. Il suffit donc de calculer ce nombre modulo 10 *(exemple : dans* 12! *le chiffre avant les* 0 *est* 4790016 *modulo* 10*)*.
On va donc créer une petite procédure qui prend un entier k et donne l'exposant de 2, l'exposant de 5 et le chiffre des unités de ce qu'il reste. Il suffira ensuite de l'utiliser pour k de 1 à n *(inclus)* et de sommer les exposants, étape par étape.
Exemple : $1.(2).(3).(2^2).(5).(2.3).(7).(2^3).(3^2).(2.5).(11).(2^2.3)$.

```
def decompose_deux_cinq(k) :
....expo2 = 0 #on initialise deux compteurs
....while (k%2) != 0 : #tant qu'il y a des facteurs 2
........expo2 += 1 #on les compte
........k = k/2
....expo5 = 0
....while (k%5) != 0 : #tant qu'il y a des facteurs 5
........expo5 += 1
........k = k/5
....chiffre = k%10 #on regarde le chiffre de ce qu'il reste
```

On met ensuite cette procédure en boucle pour chaque facteur k de $n!$:

```
def decfact(n) :
....expofact2 = 0
....expofact5 = 0
....unitefact = 1
....for k in range(1, n+1) :
........expok2, expok5, unitek = decompose_deux_cinq(k)
........expofact2 += expok2
........expofact5 += expok5
........unitefact = (unitefact*unitek)%10
```

A ce stade, on a la pseudo-décomposition en facteurs premiers $2^{expofact2}.5^{expofact5}.K$ avec $K \ mod \ 10 = unitefact$.
On connaît alors l'exposant de 10, il reste à récupérer le chiffre des unités :
$unitefact * 2^{expofact2-expofact5} \ mod \ 10$:

```
....for i in range(unitefact2-unitefact5) :
........unitefact = (2*unitefact)%10
....return(expofact5, unitefact)
```

Modèle de Malthus. _2015_1pho3_

On résout l'équation différentielle en appliquant directement le résultat du cours (et même du cours de Terminale, c'est dire) : $y_t = y_0.e^{a.t}$.
S'il vous plait, ne m'emmerdez pas avec des $y_t = C^{te}.e^{a.t}$ puis la détermination de C^{te} par la valeur en 0. On calcule $y_t = y_0.e^{A_t}$ où A est une primitive de l'application constante $t \longmapsto a$.

Avec la méthode d'Euler, on a $y_{t_{k+1}} = y_{t_k} + y_{t_k}.a.dt$. On a une suite géométrique de raison $1 + a.dt$. Le k^{ieme} terme est donc $y_0.\Big(1 + \dfrac{a.T}{n}\Big)^k$, et le dernier (quand on a atteint T) est $y_0.\Big(1 + \dfrac{a.T}{n}\Big)^n$

L'écart avec la vraie solution est donc $\left|\Big(1 + \dfrac{a.T}{n}\Big)^n - e^{a.T}\right|$
Quand n tend vers l'infini, cet écart tend vers 0, en utilisant notre célèbre "$\Big(1 + \dfrac{x}{n}\Big)^n$ converge vers e^x quand n tend vers l'infini.
On va affiner avec des développements limités :

$(1 + x/n)^n = e^{n.\ln(1+x/n)}$	$\ln(1+u) = u - \dfrac{u^2}{2} + \dfrac{u^3}{3} + o(u^3)_{u\to 0}$	$e^u = 1 + u + \dfrac{u^2}{2} + o(u^2)_{u\to 0}$

$n.\ln\Big(1 + \dfrac{a.T}{n}\Big) = n.\Big(\dfrac{a.T}{n} - \dfrac{a^2.T^2}{2.n^2} + \dfrac{a^3.T^3}{3.n^3} + o\Big(\dfrac{1}{n^3}\Big)\Big) = a.T - \dfrac{a^2.T^2}{2.n} + \dfrac{a^3.T^3}{3.n^2} + o\Big(\dfrac{1}{n^2}\Big)$
On passe à l'exponentielle, en rappelant $e^{u+v+w} = e^u.e^v.e^w$:
$\Big(1 + \dfrac{a.T}{n}\Big)^n = e^{a.T}.e^{-a^2.T^2/2.n}.e^{a^3.T^3/3.n^2}.e^{o(1/n^2)}$
On développe chaque terme jusqu'à $o(1/n^2)$

$e^{a.T}$	$1 - \dfrac{a^2.T^2}{2.n} + \dfrac{a^4.T^4}{8.n^2} + o\Big(\dfrac{1}{n^2}\Big)$	$1 + \dfrac{a^3.T^3}{3.n^2} + o\Big(\dfrac{1}{n^2}\Big)$	$1 + o\Big(\frac{1}{n^2}\Big)$

On multiplie tout en incorporant au $o(1/n^2)$ tout ce qui est en $1/n^3$ et pire

	1	$-\dfrac{a^2.T^2}{2.n}$	$\dfrac{a^4.T^4}{8.n^2}$	$o\Big(\dfrac{1}{n^2}\Big)$
1	1	$-\dfrac{a^2.T^2}{2.n}$	$\dfrac{a^4.T^4}{8.n^2}$	o
$\dfrac{a^3.T^3}{3.n^2}$	$\dfrac{a^3.T^3}{3.n^2}$	o	o	o
$o\Big(\dfrac{1}{n^2}\Big)$	o	o	o	o

On termine en multipliant par $e^{a.T}$ et en soustrayant :

$$\Big(1+\frac{a.T}{n}\Big)^n - e^{a.T} = y_0.\Big(-\frac{a^2.T^2.e^{a.T}}{2.n} + \frac{(3.a^4.T^4+8.a^3.T^3).e^{a.T}}{24.n^2} + o\Big(\frac{1}{n^2}\Big)_{n\to+\infty}\Big)$$

La méthode améliorée avec un développement d'ordre 2 donne $y_{t_{k+1}} = y_{t_k}.\Big(1+\frac{a.T}{n}+\frac{a^2.T^2}{2.n^2}\Big)$. La raison a changé, mais la suite est toujours géométrique : $Y_T = y_0.\Big(1+\frac{a.T}{n}+\frac{a^2.T^2}{2.n^2}\Big)^n$

On suite la même méthode, mais déjà, pour $\ln(1+u) = u - \frac{u^2}{2} + \frac{u^3}{3} + o(u^3)$ il faut être prudent car ici u n'est autre que $\frac{a.T}{n}+\frac{a^2.T^2}{2.n^2}$. On a alors $u^2 = \frac{a^2.T^2}{n^2} + \frac{a^3.T^3}{n^3} + \frac{a^4.T^4}{4.n^4}$, que l'on coupe à $u^2 = \frac{a^2.T^2}{n^2} + \frac{a^3.T^3}{n^3} + o\Big(\frac{1}{n^3}\Big)$.
On fait de même avec $u^3 = \frac{a^3.T^3}{n^3} + o\Big(\frac{1}{n^3}\Big)$.
On somme, on regroupe les termes de même exposant :
$\ln\Big(1+\frac{a.T}{n}+\frac{a^2.T^2}{2.n^2}\Big) = \frac{a.T}{n} + \frac{0}{n^2} - \frac{a^3.T^3}{6.n^3} + o\Big(\frac{1}{n^3}\Big)$
On multiplie par n et on passe à l'exponentielle :
$\Big(1+\frac{a.T}{n}+\frac{a^2.T^2}{2.n^2}\Big)^n = e^{a.T}.e^{-a^3.T^3/6.n^2}.e^{o(1/n^2)}$
On ne veut que des termes jusqu'à l'ordre 2 : seul $e^{-a^3.T^3/6.n^2}$ en apporte :

$$\Big(1+\frac{a.T}{n}+\frac{a^2.T^2}{2.n^2}\Big)^n - e^{a.T} = e^{a.T}.\frac{a^3.T^3}{6.n^2} + o\Big(\frac{1}{n^2}\Big)_{n\to+\infty}$$

Une conclusion : on est passe d'une erreur en $\frac{1}{n}$ à une erreur en $\frac{1}{n^2}$. Il y a un progrès certain.

Pour créer une liste abscisse/ordonnée, on peut aller très vite. La suite des abscisses est arithémtique de raison T/n et la suite des ordonnées est géométrique de raison $1+a.T/n+a^2.T^2/2.n^2$.

```
def Euler_en_mieux(y0, a, T, n) :
....pas = T/n #devrait on s'assurer que n est non nul
....abscisse = 0
....raison = 1+a*pas+a*a*pas*pas/2 #on la calcule une fois pour toutes
....ordonnee = y0
....L = [] #on crée une liste vide qui va grandir
....for k in range(n) :
........L.append([abscisse, ordonnee]) #on ajoute à chaque fois un couple
........abscisse += pas #on avance d'un pas sur Ox
........ordonnee *= raison #on monte sur Oy

....return(L)
```

Bien sûr, vous pouvez aussi prendre

```
....for k in range(n) :
........L.append([abscisse, ordonnee])
........abscisse = k*pas
........ordonnee *= y0*(raison**k)
```

mais c'est du gaspillage, on programme dynamique pas à pas et non statique.

Modèle de Verhulst. _2015_1pho3_

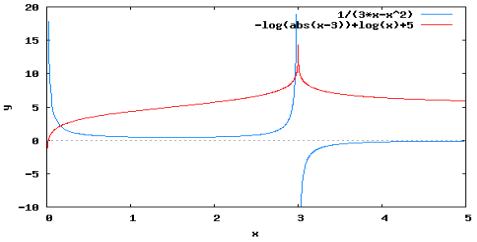
Une solution de $y'_t = a.y_t - b.(y_t)^2$ est constante si et seulement si y'_t est identiquement nulle. On résout donc $a.c - b.c^2 = 0$ d'inconnue c : $c.(a-b.c) = 0$. La solution non nulle est a/b.
Remarque : dans le cas du modèle de Malthus, b est nul, et la solution est "infinie".

On décompose $u \longmapsto \dfrac{1}{a.u-b.u^2}$ en éléments simples : $\dfrac{1}{a.u-b.u^2} = \dfrac{}{u} + \dfrac{}{a-b.u}$. Les valeurs des constantes sont obtenues par méthode des pôles ou par un système : $\dfrac{1}{a.u-b.u^2} = \dfrac{1}{a}.\dfrac{1}{u} + \dfrac{b}{a}.\dfrac{1}{a-b.u}$.
Quand il s'agit d'intégrer, il faut se placer sur un intervalle convenable (ne contenant ni 0 ni la solution fixe) :

$]-\infty,\ 0[$	$]0,\ a/b[$	$]a/b,\ +\infty[$
$u \longmapsto \dfrac{1}{a}.\ln(-u) - \dfrac{1}{a}.\ln(a-b.u)$	$u \longmapsto \dfrac{1}{a}.\ln(u) - \dfrac{1}{a}.\ln(a-b.u)$	$u \longmapsto \dfrac{1}{a}.\ln(u) - \dfrac{1}{a}.\ln(b.u-a)$

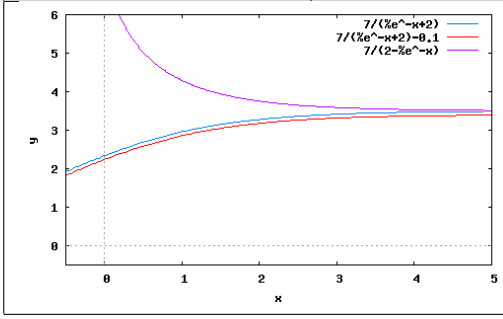
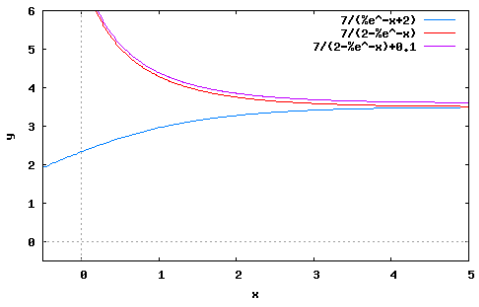
Si on veut une solution utilisable sur chaque intervalle $\boxed{u \longmapsto \dfrac{1}{a}.\ln(|u|) - \dfrac{1}{a}.\ln(|a-b.u|)}$

Mais attention, il faut préciser le domaine. Et ce n'est surtout pas $\mathbb{R}$. Ce n'est pas non plus $\mathbb{R} - \{0,\ b/a\}$, ni même $]-\infty,\ 0[\cup]0,\ b/a[\cup]b/a,\ +\infty[$. Le domaine ne peut être qu'un seul intervalle, et jamais une réunion d'intervalles. Sinon, rappelons que l'on pourrait changer de constante quand on change d'intervalle. Et pour les calculs d'intégrales, on ne peut pas passer au dessus de 0 ou de b/a.



On étudie l'application $t \longmapsto F(y_t) - t$ que l'on dérive : $t \longmapsto y'_t.F'(y_t) - 1$ (composée). On remplace y'_t par $a.y_t - b.(y_t)^2$ et $F'(y_t)$ par $\dfrac{1}{a.y_t - b.(y_t)^2}$. Tout se simplifie et on trouve 0.
L'application a une dérivée nulle sur l'intervalle d'étude. Elle est constante.

On a donc : $F(y_t) - t = F(y_0) - 0$ pour tout t. On note α le réel $F(y_0)$.
On a donc $\frac{1}{a}.\ln(|y_t|) - \frac{1}{a}.\ln(|a-b.y_t|) = \alpha + t$.
On fusionne $\ln\left(\dfrac{y_t}{a-b.y_t}\right) = a.(\alpha+t)$ et on isole : $\left|\dfrac{y_t}{a-b.y_t}\right| = e^{a.(t+\alpha)}$.
Après, il y a une histoire de signe. Pas pour y_t mais pour $a - b.y_t$.
On notera que $a - b.y_t$ ne peut pas s'annuler. En effet, si y_t prend la valeur a/b, alors il coïncide à cet instant avec la solution constante. Mais avec une dérivée qui est alors nulle. Elle continue donc alors à coïncider avec la solution constante. C'est donc un cas à garder à part, qui n'est pas l'objet de notre étude.
Comme $a - b.y_t$ ne s'annule pas, il reste de signe constant.
Il ne reste plus qu'à résoudre pour isoler y_t.

$0 < y_0 < a/b$	$y_0 = a/b$	$y_0 > a/b$
$\dfrac{a.e^{a.(t+\alpha)}}{b.e^{a.(t+\alpha)}+1}$	$\dfrac{a}{b}$	$\dfrac{a.e^{a.(t+\alpha)}}{b.e^{a.(t+\alpha)}-1}$
		

L'équation n'est pas linéaire. Elle n'entre pas dans le cadre du programme actuel de Classes Préparatoires (elle y entrait avant une réforme). On sait tout de même la résoudre, car elle est "à variables séparables, simple". Sachez quand même qu'il n'y a que peu d'équations que l'on sache résoudre explicitement...

Dans tous les cas, en divisant numérateur et dénominateur par $e^{a.(t+\alpha)}$, on trouve que la solution converge vers a/b quand t tend vers l'infini.
Au contraire du modèle de Malthus qui "explose à l'infini", le modèle de Verhulst se stabilise.

Modèle de Allee. _2015_1pho3_

Cette fois, ce n'est pas la peine de chercher à résoudre explicitement l'équation. Alors tant pis, on utilise la méthode d'Euler en découpant l'intervalle $[0,\ T]$ en n intervalles de longueur T/n.

```
def Allee_Euler(y0, K, c, T, n) :
....pas = T/n #s'assurer qu'on travaille en float
....y = y0
....L = [y0]
....for k in range(n) :
........yprime = y*(1-y/K)*(y-c)/K
........y = y+yprime*pas
........L.append(y)
....return(L)
```

Si besoin, on fera aussi intervenir la liste des abscisses.
Graphiquement, je ne vois pas de grande différence avec le modèle de Verhulst.

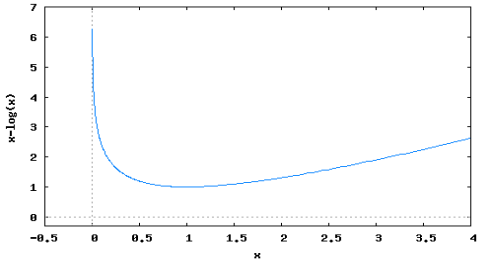
Modèle de Volterra. _2015_1pho3_

Pour trouver une solution constante, il convient d'annuler à la fois y' et x'. La solution constante *(non nulle)* est $x_\infty = c/d$ et $y_\infty = a/b$ puisqu'on a $x' = x.(a - b.y)$ et $y' = y.(-c + d.x)$.

On dérive donc $t \longmapsto d.x_t + b.y_t - c.\ln(x_t) - a.\ln(y_t)$. On trouve $d.x'_t + b.y'_t - c.\dfrac{x'_t}{x_t} - a.\dfrac{y'_t}{y_t}$.
On remplace : $d.x_t.(a - b.y_t) + b.y_t.(-c + d.x_t) - c.(a - b.y_t) - a.(-c + d.x_t)$.
On développe et on trouve bien 0.
Sur l'intervalle de résolution, cette quantité est donc constante.

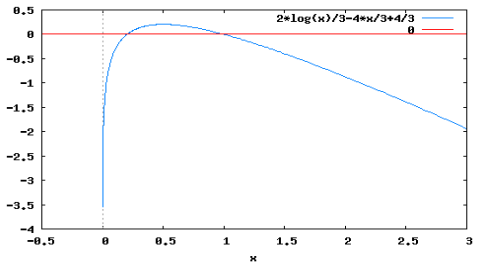
Dans le cas particulier $(a,\ b,\ c,\ d) = (2/3,\ 4/3,\ 1,\ 1)$ et $(x_0,\ y_0) = (1,\ 1)$, cette constante se calcule à $t = 0$: $d.x_0 + b.y_0 - c.\ln(x_0) - a.\ln(y_0) = 7/3$.
On a donc pour tout t : $x_t + \dfrac{4}{3}.y_t - \ln(x_t) - \dfrac{2}{3}.\ln(y_t) = \dfrac{7}{3}$.

Il pourra m'arriver dans la suite d'appeler "énergie du système" cette quantité constante.
On étudie l'application $u \longmapsto u - \ln(u)$ sur l'intervalle $]0,\ +\infty[$. Elle est décroissante, puis croissante *(minimum en 1 où sa dérivée s'annule)*. Son minimum, atteint en 1, est 1.



On a donc pour tout t : $x_t + \frac{4}{3}.y_t - \ln(x_t) - \frac{2}{3}.\ln(y_t) = \frac{7}{3}$ et $x_t - \ln(x_t) \geqslant 0$.
On isole $\dfrac{7}{3} - \dfrac{4}{3}.y_t + \dfrac{2}{3}.\ln(y_t) = x_t - \ln(x_t) \geqslant 1$.

On étudie alors l'autre application auxilliaire $y \longmapsto \frac{4}{3} - \frac{4}{3}.y + \frac{2}{3}.\ln(y)$.
Elle admet un maximum en 1/2 et est de la forme suivante :
Elle s'annule et change de signe en deux points *(valeurs intermédiaires, sens de variations...)*. On les note r_1 et r_2.
Comme la quantité $\frac{4}{3} - \frac{4}{3}.y + \frac{2}{3}.\ln(y)$ est tenue de rester positive, c'est que y_t reste entre les racines. Il est positif.

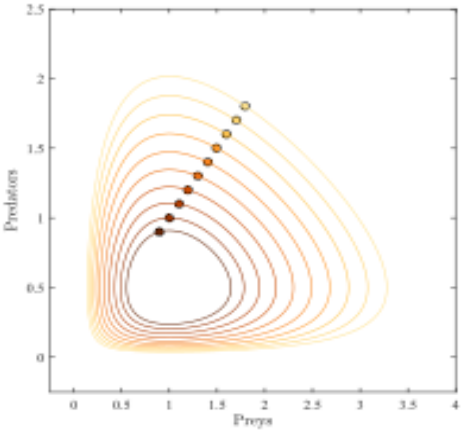


y_t reste bornée *(existence des bornes prouvée, mais valeur précise non déterminée)*.

On repasse dans l'autre sens : pour tout t et tout y_t, la quantité $\frac{4}{3}.y_t - \frac{2}{3}.\ln(y_t)$ est plus grande que $\frac{2+\ln(4)}{3}$ *(sens de variations de fonction, minimum atteint en 1/2)*.
On reporte : $\ln(x_t) - x_t$ doit rester plus grand qu'une constante qu'on peut déterminer.
Par variation de fonctions là encore, x reste entre deux racines qu'on peut nommer sans les calculer.

Les deux fonctions $t \longmapsto x_t$ et $t \longmapsto y_t$ sont bornées. C'est ce que l'on voit sur le graphe quand on le trace. On voit aussi que les deux espèces ne vont pas s'éteindre, ni même tendre vers 0 à l'infini.

On fait mieux. Pour y connu, on cherche x qui est donné par l'équation $x_t + \frac{4}{3}.y_t - \ln(x_t) - \frac{2}{3}.\ln(y_t) = \frac{7}{3}$ que l'on met sous la forme $x - \ln(x) = \frac{2}{3}.\ln(y) - \frac{4}{3}.y + \frac{7}{3}$.
Or, l'application $x \longmapsto x - \ln(x)$ est décroissante puis croissante et passe donc au plus deux fois par la valeur imposée $\frac{2}{3}.\ln(y) - \frac{4}{3}.y + \frac{7}{3}$ et même une seule si on est en un "sommet").
Pour chaque valeur de y, il n'y a que deux x possibles, et pour chaque valeur de x, il n'y aura aussi que deux y possibles. C'est ce qui justifie la forme "patatoïdale" des trajectoires x et y fonctions du temps. Les "trajectoires" ne peuvent être "spiralées".



On n'est plus très loin de la conclusion "le phénomène est périodique".

Profitions en pour noter T la période. La valeur moyenne de x sur une période est donc $\frac{1}{T}\int_0^T x_t.dt$.
Mais on peut extraire de la formule $y'_t = -c.y_t + d.x_t.y_t$: $d.x_t.y_t = y'_t + c.y_t$ puis $x_t = \frac{y'_t}{d.y_t} + \frac{c}{d}$.
On intègre donc $\frac{1}{T}\int_0^T \left(\frac{y'_t}{d.y_t} + \frac{c}{d}\right).dt$.
Le premier terme s'intègre en logarithme : $\ln(y_T) - \ln(y_0)$. Il est nul, justement parce que l'application est périodique.
Le second terme s'intègre comme une constante et donne pour valeur moyenne c/d.
On pourrait faire de même avec la valeur moyenne de y_t qui vaut a/b.
On peut conclure qu'on tourne autour des valeurs $(x_\infty,\ y_\infty)$ correspondant au système statique. Mais en plus, la valeur moyenne est ce centre.
Les populations varient, mais au fil du temps, il y a en moyenne toujours autant de proies que si le système était stationnaire.
Là où il se passe des choses étonnantes, c'est que quand intervient une troisième espèce de super-prédateurs qui s'en prennent tant aux proies qu'aux prédateurs, ce point médian $(x_\infty,\ y_\infty)$ se déplace. Mais pas forcément

comme on s'y attent. Et Volterra l'a observé avec le système réel des poissons et des requins. Les populations fluctuent (périodiquement) au fil du temps, mais globalement, le nombre de poissons reste stable, de même que le nombre de requins. Mais si les humains interviennent avec leurs filets et pêchent indifféremment poissons et requins, le nombre moyen de poissons agmente.

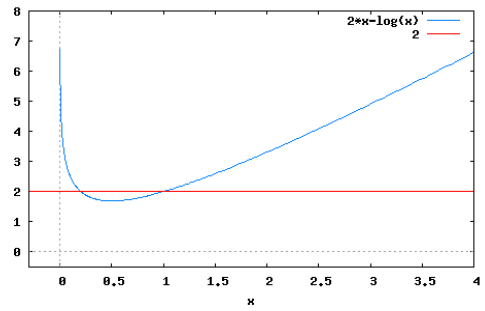
Il manque une question sur les extrema. Que se passe-t-il quand y est maximum ? Classiquement, la dérivée y'_t s'annule et change de signe. Or, cette dérivée, c'est $-c.y_t + d.x_t.y_t$. On trouve alors $y_t = 0$ *(impossible)* ou $x_t = c/d$ *(tiens, encore notre valeur moyenne x_∞).*

Application numérique avec $(a,\ b,\ c,\ d) = (2/3,\ 4/3,\ 1,\ 1)$: $x_\mu = 1$.

On reporte alors dans la formule donnant l'"énergie" $x_\mu + \frac{4}{3}.y_\mu - \ln(x_\mu) - \frac{2}{3}.\ln(y_\mu) = \frac{7}{3}$: $1 + \frac{4}{3}.y_\mu - \ln(1) - \frac{2}{3}.\ln(y_\mu) = \frac{7}{3}$. On trouve alors $2.y - \ln(y) = 2$.

C'est une équation pas évidente. Sauf qu'une racine en est connue : 1.

A-t-on besoin de Python pour la trouver ? Non. Mais c'est peut être l'autre qu'il faut trouver par dichotomie.



La racine cherchée est entre 1/5 et 1/2 *(vérifiez en calculant les valeurs approchées en ces points).*

```
def f(x) :
....return(2*x-log(x)) #vous avez importé log du module math ?
a = 0.2
b = 0.5
while b-a > 0.01 :
....c = (a+b)/2
....if f(c) > 0 :
........a = c
....else :
........b = c
print(a, b)
```