Écrit blanc

Master MEEF NSI 2023/2024

Convention Dans ce sujet, l'écriture « `machine à écrire` » représente du code Python.

Problème 1. La spirale d'Ulam

1.1 Dessin d'une spirale

Le langage Python dispose d'un module `turtle` permettant de réaliser des figures en déplaçant une « tortue » symbolisée par un triangle. Au départ, la tortue est tournée vers l'Est. Trois instructions seront utiles (x et a pouvant être de type entier ou flottant) :

— `forward(x)` fait avancer la tortue de x pas ;

— `left(a)` et `right(a)` font respectivement tourner la tortue de a degrés sur sa gauche et sur sa droite (la tortue n'avance pas).

La figure 1 illustre le fonctionnement du module.

```
from turtle import *
from math import sqrt

forward(100)
left(45)
forward(100*sqrt(2))
right(135)
forward(100)
```

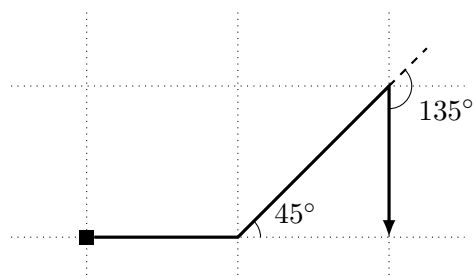


FIGURE 1 – Exemple d'utilisation du module `turtle`. Le point de départ est indiqué par le carré ■ et la position et l'orientation finales de la tortue par la flèche ➔.

Question 1.1. On veut dessiner à l'aide du module `turtle` des spirales comme celles représentées sur la figure 2.

(a) Donner la suite d'instructions utilisant le module `turtle` pour dessiner la spirale 3×3.

Note : On supposera que les carrés du quadrillage ont côté 20, et on ne dessinera pas le quadrillage.

(b) Montrer que pour tout $n \in \mathbb{N}^*$, $n + \sum_{i=1}^{n}(i+i) = (n+1)^2 - 1$. Donner une interprétation graphique de ce résultat.

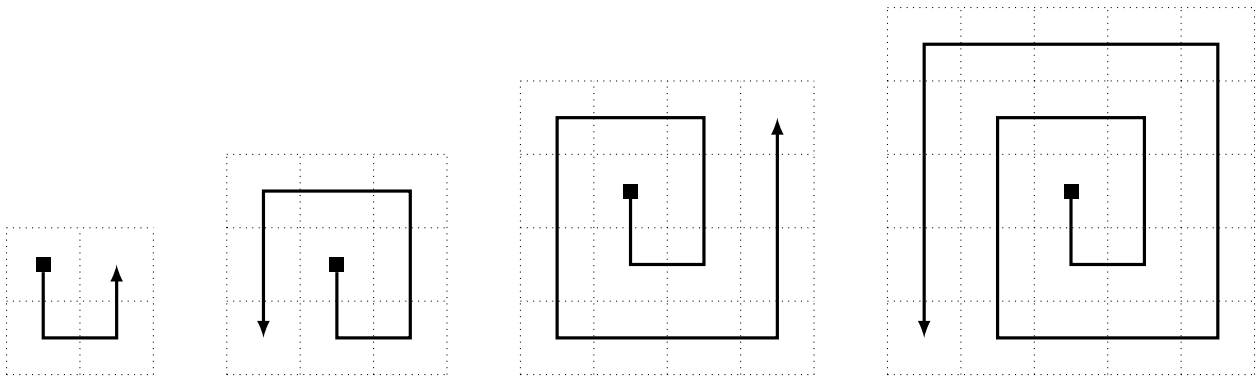


FIGURE 2 – Spirales 2×2, 3×3, 4×4 et 5×5.

(c) Écrire en python une fonction `spirale` prenant un entier $n \geq 2$ en entrée et dessinant avec le module `turtle` la spirale $n \times n$.

Question 1.2. On donne maintenant des coordonnées au quadrillage utilisé pour la spirale : le carré en haut à gauche a pour coordonnées $(0, 0)$, et celui en bas à droite a pour coordonnées $(n-1, n-1)$.

(a) Donner sans justification les coordonnées du point de départ de la spirale en fonction de n.

La fonction `coordonnees_spirale` de la figure 3 renvoie la liste des coordonnées des cases traversées par la tortue. La table donne un exemple pour $n = 7$ des positions successives de la tortue, en commençant par 0.

```python
def coordonnees_spirale(n):
    dx, dy = 0, 1
    a = b = (n-1)//2
    w = [(a,b)]
    for i in range(1,n):
        t = 3 if i == n-1 else 2
        for j in range(t):
            for k in range(i):
                a, b = a+dx, b+dy
                w.append((a,b))
            dx, dy = dy, -dx
    return w
```

42	41	40	39	38	37	36
43	20	19	18	17	16	35
44	21	6	5	4	15	34
45	22	7	0	3	14	33
46	23	8	1	2	13	32
47	24	9	10	11	12	31
48	25	26	27	28	29	30

FIGURE 3 – Fonction `coordonnees_spirale`, et résultat (mis en forme) pour la valeur 7.

(b) À l'aide de la fonction `coordonnees_spirale`, écrire une fonction `table_spirale` qui prend en entrée un entier $n > 0$ et qui renvoie le tableau T de dimension $n \times n$ contenant les entiers de 0 à $n^2 - 1$ dans l'ordre de parcours de la spirale, comme dans l'exemple donné.

Rappel : on peut initialiser un tableau $n \times n$ à 0 en utilisant la syntaxe
`T=[[0]*NbColonnes for _ in range(NbLignes)]`.

Question 1.3. Nous allons voir que la répartition des nombres premiers dans la spirale construite précédemment est loin d'être quelconque.

(a) Donner une liste de six nombres premiers alignés dans l'exemple de la spirale de taille 7×7.

(b) Écrire une fonction en python qui prend 3 couples de coordonnées en entrée et qui teste si les points correspondant sont alignés (réponse `True`) ou non (réponse `False`).

(c) Généraliser le programme précédent pour qu'il accepte une liste de points de taille quelconque.

1.2 Nombre de diviseurs

Dans cette partie, on s'intéresse au calcul du nombre des diviseurs d'un nombre premier. On appelle *diviseur strict* d'un entier n tout nombre positif $k < n$ qui divise n.

Question 1.4.

(a) Quels sont les diviseurs stricts de 0 et de 1 ? Comment appelle-t-on un nombre ayant un seul diviseur strict ?

La fonction suivante compte le nombre de diviseurs stricts de l'entier donné en entrée.

```
def nombre_diviseurs(n):
    c = 0
    k = 1
    while k < n:
        if n % k == 0:
            c += 1
        k += 1
    return c
```

(b) Quelle est la complexité de la fonction `nombre_diviseurs` ?

(c) Comment modifier la quatrième ligne pour améliorer cette complexité ?

(d) Écrire une fonction `est_parfait` testant si un nombre est parfait, c'est-à-dire s'il est égal à la somme de ses diviseurs stricts.

Question 1.5. Crible d'Eratosthène

Dans cette question on utilise la technique du crible d'Eratosthène pour calculer le nombre de diviseurs stricts de tous les entiers entre 2 et une borne n donnée. La version originale du crible permet de calculer la liste des nombres premiers jusqu'à n. Partant de la liste de tous les entiers entre 2 et n, on commence par barrer tous les entiers pairs (sauf 2), puis tous les multiples de 3 (sauf 3), etc. Plus généralement, pour tout entier $k < n$ non encore barré, on barre tous ses multiples stricts. La liste des nombres non barrés à l'issue du crible est la liste des nombres premiers.

(a) Quel est le plus grand entier k qu'il est nécessaire de considérer dans l'algorithme décrit ci-dessus ?

(b) Écrire une fonction `crible` qui prend en entrée un entier n et renvoie la liste des nombres premiers inférieurs ou égaux à n, en utilisant la technique du crible.

(c) Adapter la méthode du crible pour écrire une fonction `crible_etendu` qui prend en entrée un entier n et renvoie une liste `L` de taille $(n+1)$ telle que `L[k]` contient le nombre de diviseurs stricts de k, pour $1 \leq k \leq n$ (par convention, `L[0]` contiendra 0).

(d) Estimer la complexité de votre algorithme, et la comparer à la complexité de l'algorithme consistant à utiliser la fonction `nombre_diviseurs` sur chaque entier entre 1 et n.

1.3 Spirale d'Ulam

La spirale d'Ulam est obtenue à partir de la spirale calculée par la fonction `table_spirale` en remplaçant chaque entier par son nombre de diviseurs stricts.

7	1	7	3	3	1	8
1	5	1	5	1	4	3
5	3	3	1	2	3	3
5	3	1	0	1	3	3
3	1	3	0	1	1	5
1	7	2	3	1	5	1
8	2	3	3	5	1	7

FIGURE 4 – Spirale d'Ulam 7×7.

Question 1.6. Écrire une fonction `ulam` qui prend un entier $n > 0$ en entrée et qui renvoie un tableau de dimension $n \times n$ représentant la spirale d'Ulam $n \times n$.

À l'aide de `matplotlib`, on peut alors tracer la spirale, en représentant les entiers par des couleurs.

```
import matplotlib.pyplot as plt

plt.imshow(ulam(300),
           cmap="ocean_r",
           interpolation="nearest")
```

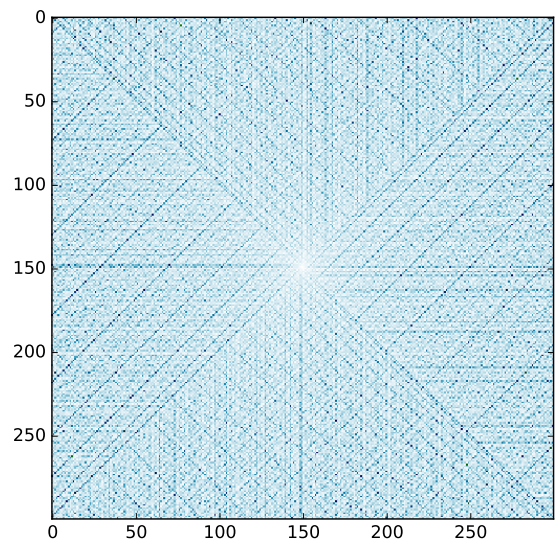


FIGURE 5 – Représentation de la spirale d'Ulam 300×300, produite par `matplotlib`.

D'après un problème de E. Volte (U. Cergy-Pontoise).

Problème 2. Quelques systèmes de chiffrement

L'objectif de cet exercice est de voir quelques systèmes de chiffrement, et les algorithmes associés.

Vocabulaire. Un système de chiffrement permet de transformer un *message clair* (c'est-à-dire compréhensible) en *message chiffré* (incompréhensible). La fonction qui permet de passer du message clair au message chiffré est appelée *fonction de chiffrement*, et la fonction qui permet de faire l'opération inverse est appelée *fonction de déchiffrement*.

2.1 Chiffrement à clef secrète

Dans cette partie, les messages (clairs et chiffres) sont écrits sur l'alphabet à 26 lettres constitué des lettres majuscules latines (sans espace ni ponctuation). On utilisera les fonctions suivantes pour passer d'une lettre à son numéro (de 0 pour `A` à 25 pour `Z`) et inversement.

```python
def numero(c):
    "Renvoie le numero de la lettre c"
    return ord(lettre)-65

def lettre(i):
    "Renvoie la lettre dont le numero est i"
    return chr(i + 65)
```

Question 2.1. Chiffrement par décalage Pour ce chiffrement, on choisit un nombre entier k ; le chiffrement par décalage consiste alors à remplacer chaque lettre par la lettre située k rangs plus loin, en bouclant : par exemple pour $k = 3$, $A \to D$, $B \to E$, ..., $W \to Z$, $X \to A$, $Y \to B$, $Z \to C$. Ainsi, le mot `BONJOUR` devient `ERQMRXU`. On fait l'opération inverse pour déchiffrer.

La valeur k du décalage est appelée *clef (secrète)* du chiffrement. Pour chiffrer ou déchiffrer un message, il faut disposer de cette clef.

(a) Chiffrer le mot `ALGORITHME` par décalage de 5 lettres. Déchiffrer le mot `RFYMX` avec un décalage de 5 lettres.

(b) Combien y a-t-il de clefs possibles dans un chiffrement par décalage ?

(c) Écrire en Python les fonctions de chiffrement et de déchiffrement du chiffrement par décalage.

Question 2.2. Chiffrement de Vigenère Afin d'augmenter le nombre de clefs possibles, le chiffrement de Vigenère utilise un mot comme clef. Chaque lettre de la clef est vue comme un décalage, égal au numéro de la lettre. Ainsi `A` correspond au décalage 0, `B` au décalage 1, ..., `Z` au décalage 25. Pour chiffrer un message, on écrit la clef sous le message, en la répétant autant de fois que nécessaire pour atteindre la longueur du message. Chaque lettre du message clair est alors chiffrée en appliquant le décalage correspondant à la lettre de la clef écrite dessous. Le schéma ci-dessous montre le chiffrement du message `ALGORITHME` avec la clef `CAFE` : la lettre `A` est décalée de 2 pour donner `C`, la lettre `L` est décalée de 0, la lettre `G` de 5 pour donner `L`, etc.

```
ALGORITHME
CAFECAFECA
CLLSTIYLOE
```

(a) Chiffrer le mot `RECURSIVITE` avec la clef `CAFE`. Déchiffrer le message chiffré `KTJVCTNSP` avec la même clef `CAFE`.

(b) Écrire en Python les algorithmes de chiffrement et déchiffrement pour le système de Vigenère.

(c) Combien le chiffrement de Vigenère admet-il de clefs de longueur comprise entre 4 et 10 (inclus) ? Estimer le nombre de chiffres en base 10 du résultat en utilisant la courbe du logarithme en base 10 fournie en annexe.

On cherche maintenant à concevoir des algorithmes de cryptanalyse des systèmes de chiffrement étudiés précédemment, c'est-à-dire des algorithmes qui étant donné un message chiffré, renvoient le message clair *le plus probable*. Ces algorithmes ne sont pas infaillibles, et le message clair renvoyé n'est pas forcément le bon.

Pour le chiffrement par décalage, on remarque d'une lettre est toujours chiffrée par la même lettre (par exemple `T` en `C` si le décalage est 9). En utilisant le fait qu'en français la lettre `E` est la plus fréquente, on adopte la stratégie suivante : on calcule la lettre la plus fréquente du message chiffré, puis en déduit le décalage en faisant l'hypothèse que cette lettre est bien la version chiffrée du `E`.

Question 2.3. Cryptanalyse du chiffrement par décalage Écrire en Python l'algorithme de cryptanalyse du chiffrement par décalage décrit ci-dessus. *L'algorithme prend en entrée un message chiffré (mais pas sa clef) et renvoie le message clair correspondant le plus probable.*

Pour le chiffrement de Vigenère, la même stratégie ne fonctionne pas directement, car une lettre n'est pas toujours chiffrée de la même manière. La première étape sert à retrouver la longueur de la clef. L'idée est de se baser sur des motifs fréquents de lettres consécutives, comme `ION` ou `QUE` en français. Si un tel motif se trouve deux fois dans le message en clair à une distance qui est un multiple de la longueur de la clef, le motif sera chiffré les deux fois de la même manière. Par exemple, considérons le chiffrement suivant où l'on chiffre les quatre mots concaténés `NATION`, `AVION`, `POSITION` et `PORTION` avec la clef `CAFE`.

```
NATIONAVIONPOSITIONPORTION
CAFECAFECAFECAFECAFECAFECA
PAYMQNFZKOSTQSNXKOSTQRYMQN
```

On remarque que deux occurences de `ION` ont été chiffrées en `MQN`, et les deux autres en `KOS`. Les deux occurences de `MQN` dans le message chiffré sont à distance 20, et les deux occurrences de `KOS` à distance 8. Le PGCD de ces distances est 4, la longueur de la clef. La deuxième étape de l'attaque consiste à utiliser la même analyse de fréquence que pour le chiffrement par décalage, autant de fois que la longueur de la clef. Dans notre exemple, en considérant les lettres d'indices 0, 4, 8, ..., on retrouve par l'analyse de fréquence la première lettre `C` de la clef. Puis on effectue le même travail pour retrouver les trois autres lettres afin de déchiffrer totalement le message. *On remarque que cette attaque ne peut fonctionner que sur un texte bien plus long que celui proposé en exemple. En effet, le message clair chiffré dans l'exemple n'a pas du tout la lettre `E` comme plus fréquente, elle n'apparaît même pas !*

Question 2.4. Cryptanalyse du système de Vigenère

(a) Écrire un programme Python qui calcule les motifs de longueur 3 qui apparaissent au moins deux fois dans un texte. La sortie doit être un `dict` (un dictionnaire) qui associe à chacun de ces motifs la liste des positions auxquelles il apparaît. *Dans l'exemple, la sortie serait*

```
{'KOS': [8, 16], 'MQN': [3, 23], 'OST': [9, 17],
  'STQ': [10, 18], 'YMQ': [2, 22]}
```

(b) Programmer la cryptanalyse du chiffrement de Vigenère décrite ci-dessus. *On pourra utiliser la fonction* `gcd` *du module* `math` *qui calcule le PGCD de deux entiers fournis en entrée.*

2.2 Chiffrement à clef publique

Une difficulté du chiffrement à clef secrète est que les deux parties qui veulent communiquer doivent d'abord s'entendre sur une même clef. Le chiffrement à clef publique permet de contourner cette difficulté.

On se place dans le contexte où un certain Bob veut envoyer un message à une certaine Alice. Dans un système de chiffrement à clef publique, la personne qui souhaite recevoir un message, Alice, diffuse publiquement une clef, dite *clef publique*, dans un annuaire (ou sur une page web, etc.). Elle garde pour elle une deuxième clef, dite *clef secrète*. Pour communiquer avec Alice, Bob chiffre le message clair à l'aide de la clef publique, puis Alice déchiffre le message chiffré à l'aide de la clef secrète (*cf.* Figure 6).

FIGURE 6 – Pour communiquer le message clair m à Alice, Bob commence par calculer le chiffré c de m à l'aide de la clef publique $\mathtt{pk}_A$ d'Alice. Il envoie le message chiffré c à Alice. À la réception, Alice déchiffre c à l'aide de sa clé secrète $\mathtt{sk}_A$. La sécurité d'un tel système repose sur le fait qu'il est difficile de calculer m à partir de c si on ne dispose que de $\mathtt{pk}_A$, et difficile de retrouver $\mathtt{sk}_A$ à partir de $\mathtt{pk}_A$.

On présente dans cette partie le système de chiffrement à clef publique de Merkle et Hellman, qui repose sur la difficulté d'un cas particulier du problème dit du « sac-à-dos ». Le problème est le suivant : étant donné un n-uplet d'entiers $(a_1, \ldots, a_n)$ et un entier S, il faut trouver un n-uplet $(x_1, \ldots, x_n) \in \{0, 1\}^n$ tel que $S = \sum_{i=1}^{n} x_i a_i$. Autrement dit, il s'agit d'identifier un sous-ensemble des a_i dont la somme vaut S.

Question 2.5. Problème du sac-à-dos

(a) Écrire un algorithme récursif pour résoudre le problème du sac-à-dos décrit ci-dessus. L'algorithme renverra le n-uplet $(x_1, \ldots, x_n)$ s'il y a une solution, et $(0, \ldots, 0)$ sinon.

(b) Quelle est la complexité de votre algorithme, en fonction de n ? Pourquoi peut-on dire que ce problème est *difficile* ?

Le système de chiffrement de Merkle et Hellman permet de chiffrer des messages binaires d'une longueur n donnée. La clef publique est un n-uplet d'entiers $(k_1, \ldots, k_n)$. Étant donné un message

clair $m = m_1 \cdots m_n \in \{0,1\}^n$, Bob calcule le message chiffré $c = \sum_{i=1}^n k_i m_i$. Pour retrouver le message clair si on ne connaît pas la clef privée, il faut résoudre le problème du sac-à-dos avec comme entrée la clef publique $(k_1, \ldots, k_n)$ et la somme c. Ce système de chiffrement repose sur le fait que le problème du sac-à-dos est en général difficile, mais facile sur certaines instances particulières.

Question 2.6. n-uplets super-croissants On dit qu'un n-uplet d'entiers $(a_1, \ldots, a_n)$ est *super-croissant* si pour $2 \leq i \leq n$, $a_i > \sum_{j=1}^{i-1} a_j$.

(a) Donner un exemple de 10-uplet d'entiers super-croissant.

(b) Écrire un algorithme qui teste si un n-uplet d'entiers est super-croissant.

(c) Soit $(a_1, \ldots, a_n)$ un n-uplet super-croissant et S un entier. Montrer que s'il existe un sous-ensemble $I \subset \{1, \ldots, n\}$ tel que $\sum_{i \in I} a_i = S$, alors l'entier $i_{\max} = \max\{i : a_i \leq S\}$ appartient à I.

(d) En déduire un algorithme de type glouton qui résout le problème du sac-à-dos lorsque l'entrée est un n-uplet super-croissant. Comparer sa complexité avec la complexité de l'algorithme dans le cas général.

Afin de créer une clef publique et une clef secrète, on part d'un n-uplet super-croissant (aléatoire) qu'on *camoufle*. Pour cela, étant donné un n-uplet super-croissant $(a_1, \ldots, a_n)$, on définit la clef publique de la manière suivante :

— on choisit un entier $u > \sum_{i=1}^n a_i$;
— on choisit un entier $v \leq u$ tel que $\mathrm{pgcd}(u, v) = 1$;
— on définit $k_i \leq u$ par $k_i = v \times a_i \bmod u$;
— la clef publique est $(k_1, \ldots, k_n)$ et la clef secrète est le couple (u, v).

Question 2.7. Chiffrement de Merkle et Hellman

(a) Écrire un algorithme qui prend en entrée deux entiers n et M et renvoie un n-uplet $(a_1, \ldots, a_n)$ super-croissant aléatoire tel que $a_1 \in \{1, \ldots, M\}$ et pour tout $i > 1$,

$$1 < \frac{a_i}{\sum_{j<i} a_j} \leq M.$$

On pourra utiliser la fonction `randint` *du module* `random` *qui prend en entrée deux entiers a et b et renvoie un entier aléatoire compris entre a et b, inclus.*

(b) Écrire un algorithme `clefs_MH` qui prend en entrée deux entiers n et M et renvoie un couple $(\mathtt{pk}, \mathtt{sk})$ où $\mathtt{pk} = (k_1, \ldots, k_n)$ est la clef publique et $\mathtt{sk} = (u, v)$ la clef secrète : elles sont créées à partir d'un n-uplet super-croissant aléatoire, en choisissant u aléatoire entre $1 + \sum_i a_i$ et $M \times \sum_i a_i$, et en choisissant v aléatoirement parmi les éléments premiers avec u.

(c) Écrire un algorithme de chiffrement `chiffre_MH` pour le système de Merkle et Hellman, qui prend en entrée une clef publique et un message clair, et renvoie le message chiffré. *Le message clair sera un n-uplet de* 0 *et de* 1*, et le message chiffré sera un entier Python.*

Question 2.8. Inverse modulaire Afin de pouvoir déchiffrer un message chiffré, il faut inverser v *modulo* u, c'est-à-dire calculer l'unique entier $w \in \{1, \ldots, u-1\}$ tel que $vw = 1 \bmod u$. On utilise pour cela l'algorithme suivant.

```python
def euclide_etendu(u, v):
    if v == 0:
        return (u, 1, 0)
    q = u // v
    r = u % v
    (d, s1, t1) = euclide_etendu(v, r)
    return (d, t1, s1 - t1*q)
```

Soit $u, v \in \mathbb{N}$ tels que $u \geq v$ et (d, s, t) le triplet renvoyé par `euclide_etendu(u,v)`.

(a) Montrer que $d = \text{pgcd}(u, v)$. *On pourra montrer que pour tout* $x, y \in \mathbb{N}$, $\text{pgcd}(x, y) = \text{pgcd}(y, x \bmod y)$ *où* $x \bmod y$ *désigne le reste dans la division euclidienne de* x *par* y.

(b) Montrer que $d = su + tv$.

(c) Comment s'appellent les entiers s et t ?

(d) Écrire un algorithme qui étant donné deux entiers u et v premiers entre eux, $u \geq v$, calcule l'inverse de v *modulo* u.

Question 2.9. Déchiffrement du système de Merkle et Hellman On utiliser l'algorithme de déchiffrement suivant : étant donné un message chiffré c, la clef publique $\texttt{pk} = (k_1, \ldots, k_n)$ et la clef privée $\texttt{sk} = (u, v)$,

— calculer l'inverse w de v *modulo* u, puis $c' = wc \bmod u$;
— pour tout i, calculer $a_i = wk_i \bmod u$;
— résoudre le problème du sac-à-dos d'entrées $(a_1, \ldots, a_n)$ et c' ;
— renvoyer la solution trouvée du problème de sac-à-dos.

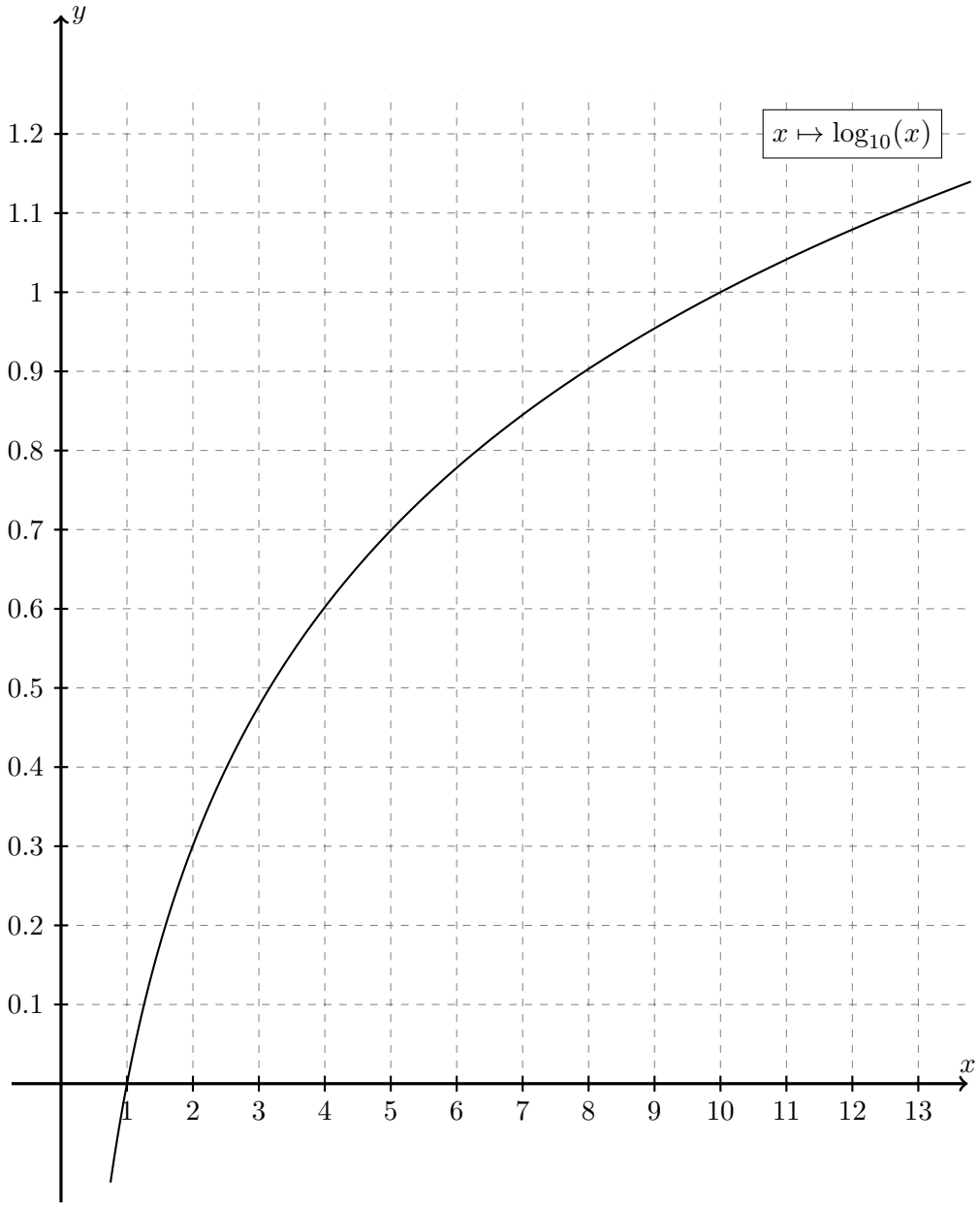
(a) Montrer que l'algorithme est correct : si c est le message chiffré obtenu en chiffrant un message m avec la clef `pk`, alors l'algorithme décrit renvoie bien m.

(b) Soit $\texttt{pk} = (9, 18, 36, 72, 13, 26)$ la clef publique, et $\texttt{sk} = (131, 70)$ la clef privée. Après avoir vérifié que 73 est l'inverse de 70 *modulo* 131, déchiffrer le message chiffré 67.

(c) Écrire l'algorithme `dechiffre_MH` de déchiffrement du système de Merkle et Hellman.

Annexes

Courbe de la fonction logarithme en base 10



Listes Python

```
>>> maListe = [42, 17, 21, 172, 13]    # Creation
>>> maListe[2]              # Element d'indice 2
21
>>> len(maListe)            # Longueur
5
>>> maListe + [4, 5]        # Concatenation
[42, 17, 21, 172, 13, 4, 5]
>>> maListe.pop()           # Suppression de l'element de queue
13
>>> maListe.append(4)       # Ajout en queue de liste
>>> maListe
[42, 17, 21, 172, 4]
>>> for x in maListe: ... # Parcours
>>> maListe[1:4]            # Sous-liste
[17, 21, 172]
```

Dictionnaires Python

```
>>> monDico = { 5:'b', 1:'a', 3:'c'}     # Creation
>>> 2 in monDico                         # Presence d'une cle
False
>>> monDico[3]                           # Valeur associee a 3
'c'
>>> monDico[42] = 'd'                    # Ajout d'une valeur
>>> monDico[5] += 'ut'                   # Modif d'une valeur
>>> for k in monDico:                    # Parcours (par cle)
        monDico[k] += '.'
>>> del monDico[3]                       # Suppression
>>> monDico
{1:'a.', 5:'but.', 42:'d.'}
```