Correction du devoir surveillé 1

Applications directes du cours

Partie I : Lecture de programmes

1. • Décrivons l'application de la fonction `bonnet_blanc` suivante à un nombre x donné.

```
def bonnet_blanc(x):
    if x < 0:
        return "bonnet"
    elif x < 1:
        return "blanc"
```

   ⋆ Si $x < 0$, la fonction renvoie la chaine de caractères `"bonnet"`.
   ⋆ Sinon, et si $x < 1$, c'est-à-dire si $x \in [0; 1[$, la fonction renvoie la chaine `"blanc"`.
   ⋆ Sinon, c'est-à-dire si $x \geqslant 1$, la fonction ne renvoie rien.

   • Décrivons à présent l'application de la fonction `blanc_bonnet` suivante à un nombre x donné.

```
def blanc_bonnet(x):
    if x < 1:
        return "blanc"
    elif x < 0:
        return "bonnet"
```

   ⋆ Si $x < 1$, la fonction renvoie la chaine de caractères `"blanc"`.
   ⋆ La fonction renvoie la chaine `"bonnet"` lorsque x n'est pas strictement inférieur à 1, mais est strictement négatif, ce qui n'arrive jamais : en effet, les nombres négatifs sont strictement inférieurs à 1. Ainsi, la fonction n'exécute jamais le bloc du `elif`.
   ⋆ Si $x \geqslant 1$, la fonction ne renvoie rien.

> Finalement, les deux fonctions `bonnet_blanc` et `blanc_bonnet` ne font pas la même chose.
> La fonction `bonnet_blanc` renvoie `"bonnet"` sur $]-\infty; 0[$ et renvoie `"blanc"` sur $[0; 1[$, tandis que la seule valeur que peut renvoyer la fonction `blanc_bonnet` est `"blanc"`, sur $]-\infty; 1[$.

2. (a) Le programme suivant

```
for k in range(5):  # Pour k allant de 0 à 4,
    print(2 ** k)       # on affiche 2**k.
```

   affiche les valeurs de 2^0, puis de 2^1, puis de 2^2, puis de 2^3, puis de 2^4,
   c'est-à-dire que le programme affiche 1, puis 2, puis 4, puis 8, puis 16.

(b) Le programme suivant

```
for k in range(3, 4):    # Pour k allant de 3 inclus à 4 exclu,
                         # c'est-à-dire pour k valant 3,
    print(2 ** k)            # on affiche 2**k.
```

affiche la valeur de 2^3 (puisque la seule valeur prise par k est 3), c'est-à-dire que le programme affiche 8.

(c) Le programme suivant

```
for k in range(1, 7, 3):# Pour k allant de 1 inclus à 7 exclu
                        # par pas de 3, c'est-à-dire
                        # pour k valant 1, puis 4,
    print(2 ** k)           # on affiche 2**k.
```

affiche les valeurs de 2^1, puis de 2^4, c'est-à-dire que le programme affiche 2 puis 16.

(d) Au cours de l'exécution du programme suivant

```
ch = ""              # Initialement, la variable ch est
                     # la chaine vide.
for o in "Bonjour": # Pour o parcourant un à un
                     # tous les caractères de "Bonjour",
    ch = o + ch          # on concatène le caractère o
                         # à gauche de la chaine ch,
    print(ch)            # et on affiche ch à chaque tour.
```

la variable `ch` vaut successivement

- `""` avant le premier tour de boucle ;
- `"B"` après le tour de boucle où la variable o vaut le caractère `"B"` ;
- `"oB"` après le tour de boucle où la variable o vaut le caractère `"o"` ;
- `"noB"` après le tour de boucle où la variable o vaut le caractère `"n"` ;
- `"jnoB"` après le tour de boucle où la variable o vaut le caractère `"j"` ;
- `"ojnoB"` après le tour de boucle où la variable o vaut le caractère `"o"` ;
- `"uojnoB"` après le tour de boucle où la variable o vaut le caractère `"u"` ;
- `"ruojnoB"` après le tour de boucle où la variable o vaut le caractère `"r"`.

Le programme affiche chacune des valeurs de la variable `ch` prises dans la boucle : le programme affiche `"B"`, puis `"oB"`, puis `"noB"`, puis `"jnoB"`, puis `"ojnoB"`, puis `"uojnoB"`, puis `"ruojnoB"`.

Partie II : Écriture de programmes

3. Un entier n est pair si et seulement si le reste de la division euclidienne de n par 2 est nul.

```
def est_pair(n):
    """
    Entrée : un entier n.
    Sortie : True si n est pair, False sinon.
    """
    return (n % 2 == 0)
```

4. Considérons la suite $(u_k)_{k\in\mathbb{N}}$ définie par :

$$\begin{cases} u_0 = -2 \\ \forall k \in \mathbb{N}, u_{k+1} = 2u_k - 3. \end{cases}$$

Pour calculer son terme de rang n, on procède de la manière suivante.

- On commence par créer une variable `terme` qui vaut initialement u_0, c'est-à-dire -2.
- Puis on fait une boucle, à chaque tour de laquelle on applique à la variable `terme` la formule de récurrence qui définit la suite.
  La boucle fera n tours :
  - ⋆ après le premier tour de boucle, la variable `terme` vaut u_1 ;
  - ⋆ après le deuxième tour de boucle, la variable `terme` vaut u_2 ;
  - ⋆ …
  - ⋆ après n tours de boucle, la variable `terme` vaut u_n.
- À l'issue de la boucle, on renvoie la variable `terme`, qui vaut u_n.

```
def suite(n):
    """
    Entrée : un entier naturel n.
    Sortie : le terme u_n de rang n.
    """
    terme = -2                  # Avant la boucle, terme vaut u_0.
    for i in range(1, n + 1):   # Pour i allant de 1 à n,
                                # c'est-à-dire n fois,
        terme = 2 * terme - 3       # on applique la formule
                                    # de récurrence à terme,
                                    # donc terme prend pour valeur u_i.
    return terme                # Après la boucle, terme vaut u_n.
```

5. La fonction suivante prend en argument un entier n et renvoie la somme $\displaystyle\sum_{k=1}^{n}(k+1)^5$.

```
def somme_puissances_cinq(n):
    """
    Entrée : un entier n.
    Sortie : la somme pour k allant de 1 à n de (k + 1)**5.
    """
    S = 0                       # La variable S va enregistrer au fur
                                # et à mesure la valeur de la somme
                                # des termes calculés.
    for k in range(1, n + 1):   # Pour k allant de 1 à n,
        S = S + (k + 1) ** 5        # on ajoute à la variable S
                                    # le terme (k + 1)**5.
    return S
```

6. (a) Pour déterminer si un objet x est un élément d'une liste L, on parcourt un à un les éléments de la liste L.
Dès que l'on trouve l'élément x, on arrête immédiatement la recherche.

Si on arrive à la fin du parcours sans avoir trouvé x, alors la liste L ne contient pas x.

```
def recherche(L, x):
    """
    Entrées : une liste L et un objet x.
    Sortie : True si x appartient à L, False sinon.
    """
    for elem in L:       # Pour elem parcourant un à un
                         # les éléments de la liste L,
        if elem == x:        # si l'élément elem vaut x,
            return True          # alors on renvoie True
                                 # et la fonction s'arrête.
    return False         # Si on arrive à la fin du parcours,
                         # c'est que la fonction ne s'est pas
                         # arrêtée avant, donc que x n'est pas
                         # un élément de la liste L.
```

(b) Pour calculer le nombre d'occurrences d'un objet x dans une liste L, on crée une variable, initialisée à 0, qui comptera le nombre de fois où l'objet x a été rencontré dans la liste.
On parcourt alors un à un les éléments de la liste L. À chaque fois que l'élément parcouru est x, on incrémente le compteur de 1.
À la fin du parcours de la liste L, on renvoie le compteur.

```
def nombre_occurrences(L, x):
    """
    Entrées : une liste L et un objet x.
    Sortie : le nombre d'occurrences de x dans la liste L.
    """
    compteur = 0         # Nombre d'occurrences de x dans L.
    for elem in L:       # Pour elem parcourant un à un
                         # les éléments de la liste L,
        if elem == x:        # si l'élément elem vaut x,
            compteur += 1        # alors on ajoute 1 au nombre
                                 # d'occurrences de x.
    return compteur
```

(c) Pour construire la liste des indices des occurrences d'un objet x dans une liste L, on commence par créer une variable qui sera la liste des indices voulus, initialement vide.
On parcourt alors les indices de tous les éléments de la liste L. À chaque fois que l'élément d'indice k vaut x, on ajoute k à la liste d'indices d'occurrences de x.
À la fin du parcours de la liste L, la liste construite contient tous les indices des occurrences de l'objet x dans la liste L.

```
def liste_occurrences(L, x):
    """
    Entrées : une liste L et un objet x.
```

```
    Sortie : la liste des indices des occurrences de x
             dans la liste L.
    """
    Occ = []                  # Liste des indices des occurrences
                              # de l'objet x dans la liste L.
    for k in range(len(L)): # Pour k allant de 0 à len(L) - 1,
                              # càd pour k parcourant les indices
                              # des éléments de la liste L,
        if L[k] == x:             # si l'élément d'indice k
                                  # de la liste L vaut x,
            Occ.append(k)             # alors on ajoute l'indice k
                                      # à la fin de la liste Occ.
    return Occ
```

7. (a) La fonction suivante prend en argument une liste non vide de nombres et renvoie son minimum.

```
def minimum(L):
    """
    Entrée : une liste L de nombres, non vide.
    Sortie : le plus petit élément de la liste L.
    """
    mini = L[0]           # La variable mini contiendra
                          # le plus petit élément rencontré.
                          # Ce minimum courant est initialisé
                          # au premier élément de la liste L.
    for elem in L:        # Pour elem parcourant un à un
                          # les éléments de la liste L,
        if elem < mini:       # si l'élément elem est plus petit
                              # que le minimum courant,
            mini = elem           # alors il en prend la place.
    return mini           # À la fin du parcours, la variable mini
                          # est le plus petit élément de la liste L.
```

(b) Écrivons une fonction qui prend en argument une liste de nombres non vide et qui renvoie la liste des indices de toutes les occurrences du minimum dans la liste.

- **Première méthode : en appelant les fonctions précédentes**

  Une première méthode consiste à appliquer la fonction `liste_occurrences` de la question 6.(c) à la liste ainsi qu'à son minimum, calculé à l'aide de la fonction de la question précédente.

```
def occurences_min(L):
    """
    Entrée : une liste L de nombres.
    Sortie : la liste des indices des occurrences
             du minimum dans la liste.
    """
    mini = minimum(L)           # mini est le minimum de L.
    return liste_occurrences(L, mini)
```

- **Seconde méthode : en un seul parcours**

  Plutôt que d'utiliser la fonction `minimum`, qui fait un premier parcours de la liste pour en trouver le minimum, puis de parcourir à nouveau la liste pour en trouver toutes les occurrences, on peut faire les deux en parallèle, comme dans le cours.

  On parcourt alors les indices des éléments de la liste en gardant en mémoire le minimum courant, c'est-à-dire le plus petit élément rencontré lors du parcours, ainsi que la liste `Occ` de toutes les positions auxquelles on a rencontré ce minimum courant.

  ⋆ Si l'on rencontre une nouvelle occurrence du minimum courant, on ajoute à la liste `Occurrences` l'indice auquel on l'a rencontrée.

  ⋆ Si l'on rencontre un élément strictement inférieur au minimum courant, le minimum devient cet élément et on remplace la liste `Occ` par une liste dont le seul élément est l'indice auquel on vient de rencontrer le nouveau minimum courant.

  À la fin du parcours, le minimum courant est le plus petit élément de la liste, et la liste `Occ` contient bien tous les indices des occurrences du minimum dans la liste.

```
def occurrences_min_bis(L):
    mini = L[0]              # Le minimum courant, initialisé
                             # au premier élément de L.
    Occ = []                 # Liste des indices des occurrences
                             # du minimum courant, vide au départ
                             # car l'indice 0 lui sera ajouté
                             # au premier tour de boucle.
    for k in range(len(L)): # Pour k allant de 0 à len(L) - 1,
                             # càd pour k parcourant les indices
                             # des éléments de la liste L.
        if L[k] < mini:          # Si l'élément d'indice k est
                                 # plus petit que mini,
            mini = L[k]              # alors mini devient L[k].
            Occ = [k]                # Ce nouveau minimum courant
                                     # n'a été rencontré qu'à
                                     # l'indice k pour l'instant.
        elif L[k] == mini:       # Si l'élément d'indice k
                                 # vaut mini, alors
            Occ.append(k)            # on ajoute l'indice k
                                     # de cette occurrence
                                     # de mini à la liste Occ.
    return Occ               # On renvoie la liste Occ
                             # à la fin du parcours.
```

(c) La fonction suivante prend en entrée une liste L de nombres non vide et, en lui appliquant la fonction précédente, commence par trouver la liste `Occ` des indices de toutes les occurrences du minimum de la liste.

Pour chaque élément k de cette liste `Occ`, c'est-à-dire pour chaque indice k de la liste L auquel le minimum de la liste L apparait, on remplace l'élément $L[k]$ par la chaine `"J'ai cru voir un gros min"`.

```
def titi(L):
    """
    Entrée : une liste L de nombres.
    Remplace dans L toutes les occurrences du minimum
    par la chaine de caractères "J'ai cru voir un gros min.".
    Sortie : aucune, la liste L est directement modifiée.
    """
    Occ = occurrences_min(L)
    for k in Occ:           # Pour k parcourant un à un
                            # les éléments de la liste Occ,
                            # càd pour k parcourant
                            # les indices de la liste L
                            # auxquels se trouve le minimum,
        L[k] = "J'ai cru voir un gros min."
                              # l'élément d'indice k dans L
                              # prend la valeur de la chaine
                              # "J'ai cru voir un gros min."
```

Problème — Gattaca (d'après Agro-Véto 2022, filière TB)
Dans ce problème, on s'intéresse à la recherche d'une sous-chaine dans une chaine de caractères. On commencera par écrire un algorithme naïf de recherche de sous-chaine, puis on implémentera dans la dernière partieun algorithme plus efficace : celui de Knuth-Morris-Pratt.

On dira qu'une chaine de caractères `mot` de longueur p est une **sous-chaine** d'une chaine de caractères `texte` s'il existe un indice i tel que la chaine `texte[i : i + p]` soit égale à `mot`. Si i est un tel indice, alors on dira que la chaine `mot` **est contenue et commence** à l'indice i dans la chaine `texte`.

Partie I : Comparaison de chaines de caractères

1. Pour calculer la distance de Hamming entre deux chaines de caractères de même longueur, on commence par créer une variable qui comptera le nombre de positions auxquelles les deux chaines diffèrent.

Remarquons que comme les deux chaines de caractères ont la même longueur, elles ont les mêmes indices. On parcourt alors les indices des deux chaines, puis on compare dans les deux chaines les caractères de même indice et on ajoute 1 au compteur à chaque fois qu'ils sont différents.

```
def Hamming(ch, CH):
    """
    Entrées : deux chaines de caractères ch et CH.
    Sortie : la distance de Hamming entre ch et CH si ch et CH ont la même
             longueur, un message d'erreur sinon.
    """
    if len(ch) != len(CH):
        return "Ces deux chaines de caractères n'ont pas la même
        longueur ! Leur distance de Hamming n'est pas définie."
    else:
        distance = 0                # Compteur de positions auxquelles
                                    # les chaines ch et CH diffèrent.
        for k in range(len(ch)):    # Pour k allant de 0 à len(ch) - 1,
                                    # càd pour k parcourant les indices
                                    # des caractères des deux chaines,
            if ch[k] != CH[k]:          # si les caractères d'indice k
                                        # dans les deux chaines ch et CH
                                        # sont différents,
                distance = distance + 1     # alors on augmente de 1
                                            # la distance de Hamming.
        return distance
```

2. Deux chaines de caractères sont égales si et seulement si leur distance de Hamming est nulle.

```
def egalite(ch, CH):
    """
    Entrées : deux chaines de caractères ch et CH.
    Sortie : True si ch et CH sont égales, False sinon.
    """
    return Hamming(ch, CH) == 0
```

Partie II : Force brute

3. Pour trouver la liste des indices auquel une chaine `mot` de longueur p est contenue et commence dans une chaine `texte` de longueur n, on procède de la manière suivante.

   On crée tout d'abord une liste `L`, initialement vide, qui contiendra tous les indices recherchés.

   On parcourt ensuite tous les indices dans la chaine `texte` auquel la chaine `mot` est susceptible de commencer, c'est-à-dire tous les indices allant de 0 (le premier indice dans `texte`) à $n - p$, pour laisser de la place aux $p - 1$ caractères suivants (le dernier caractère de `texte` est d'indice $n - 1$).

Pour chacun de ces indices i, on teste si la chaine `mot` est contenue et commence à l'indice i dans `texte`, c'est-à-dire si la chaine `mot` est égale à la sous-chaine `texte[i : i + p]`, à l'aide de la fonction précédente. Si c'est le cas, on ajoute l'indice i à la fin de la liste `L`.

À la fin du parcours, on renvoie la liste `L`.

```
def liste_occ(mot, texte):
    """
    Entrées : deux chaines de caractères mot et texte.
    Sortie : la liste des indices auxquels la chaine mot
             commence et est contenue dans texte.
    """
    L = []                                  # Liste des indices auxquels
                                            # la chaine mot commence et
                                            # est contenue dans texte.
    n = len(texte)
    p = len(mot)
    for i in range(n - p + 1):              # Pour i allant de 0 à n - p,
        if egales(mot, texte[i : i + p]):   # si la chaine mot est
                                            # égale à la sous-chaine
                                            # texte[i : i + p], alors
            L.append(i)                         # on ajoute l'indice i
                                                # à la fin de la liste L.
    return L
```

4. Pour déterminer le plus petit indice auquel une chaine `mot` commence et est contenue dans une chaine `texte`, on commence, à l'aide de la fonction précédente, par trouver la liste de tous les indices auxquels `mot` commence et est contenue dans `texte`.

   - Si cette liste est vide, alors la chaine `mot` n'est pas contenue dans `texte`, et on renvoie un message d'erreur.
   - Sinon, comme ces indices sont obtenus dans l'ordre croissant, le plus petit indice auquel `mot` commence et est contenue dans `texte` est simplement le premier élément de la liste.

```
def sous_chaine(mot, texte):
    """
    Entrées : deux chaines de caractères mot et texte.
    Sortie : le plus petit indice auquel mot commence et est
             contenue dans texte, si mot est bien contenue
             dans texte, un message d'erreur sinon.
    """
    L = liste_occ(mot, texte)   # Liste de tous les indices où
                                # mot commence dans texte.
    if len(L) == 0:
        return "La première chaine n'est pas contenue
                dans la seconde."
    else:
        return L[0]             # Le premier élément de la liste L.
```

Partie III : Bord d'une chaine de caractères

On dit qu'une chaine de caractères `souschaine` est :

- un **préfixe** d'une chaine de caractères `ch` s'il existe un indice j tel que les chaines `ch[ : j]` et `souschaine` soient égales ;
- un **suffixe** d'une chaine de caractères `ch` s'il existe un indice j tel que les chaines `ch[j : ]` et `souschaine` soient égales ;
- le **bord** d'une chaine de caractères `ch` si `souschaine` est la plus longue chaine de caractères distincte de `ch` qui soit à la fois un préfixe et un suffixe de `ch`.

5. La fonction suivante prend en argument une chaine de caractères `ch` de longueur n et renvoie la longueur de son bord.

   Pour cela, on initialise à 0 une variable `longueur_max` qui contiendra la longueur de la plus grande sous-chaine stricte de `ch` qui en est à la fois un préfixe et un suffixe.

   Pour toute valeur j de $[\![1; n-1]\!]$, on détermine alors si le préfixe `ch[ : j]`, qui est constitué des j premiers caractères de la chaine, est également un suffixe de `ch`, c'est-à-dire s'il est égal au suffixe `ch[n - j : ]`, qui est constitué des j derniers caractères de la chaine (ceux dont l'indice va de $n-j$ à $n-1$).
   Si c'est le cas, la variable `longueur_max` prend la valeur de j.

   On ne teste que les valeurs de j allant jusqu'à $n-1$, c'est-à-dire les préfixes de longueur strictement inférieure à n, car le bord d'une chaine de caractères doit être distinct de la chaine.

   À la fin du parcours, on renvoie la variable `longueur_max`.

```
def bord(ch):
    """
    Entrée : une chaine de caractères ch.
    Sortie : la longueur du bord de ch.
    """
    n = len(ch)
    longueur_max = 0                    # Initialement, la longueur
                                        # du préfixe vide,
                                        # qui est aussi un suffixe.
    for j in range(1, n):               # Pour j allant de 1 à n - 1,
        if egales(ch[ : j], ch[n - j : ]):  # si le préfixe ch[ : j]
                                            # de longueur j est égal
                                            # au suffixe ch[n - j : ],
            longueur_max = j                    # longueur_max prend
                                                # pour valeur j.
    return longueur_max
```

6. La fonction suivante prend en argument une chaine de caractères `ch` et renvoie la liste `B` des longueurs du bord de chacun des préfixes non vides de `ch` : pour tout indice i de la chaine

`ch`, l'élément `B[i]` sera la longueur du bord du préfixe `ch[ : i + 1]` de `ch`, calculée à l'aide de la fonction précédente.

```
def longueurs_bords(ch):
    """
    Entrée : une chaine de caractères ch.
    Sortie : la liste des longueurs du bord
             de chaque préfixe non vide de ch.
    """
    B = []
    for i in range(len(ch)):          # Pour i allant de 0 à len(ch) - 1,
                                      # càd pour i parcourant les indices
                                      # des caractères de ch,
        B.append(bord(ch[ : i + 1]))      # on ajoute à la liste B
                                          # la longueur du bord du
                                          # préfixe ch[ : i + 1] de ch.
    return B
```

Partie IV : Algorithme de Knuth-Morris-Pratt

Dans cette partie, on présente l'algorithme de Knuth-Morris-Pratt, qui est une amélioration de l'algorithme naïf de recherche d'une sous-chaine dans une chaine de caractères écrit à la partie II.

On peut en effet remarquer que lorsque la recherche de la chaine `mot` dans la chaine `texte` échoue à l'indice i, il n'est pas toujours nécessaire de chercher `mot` à l'indice $i+1$: on peut parfois décaler la prochaine recherche de plusieurs caractères.

Le fonctionnement de l'algorithme de Knuth-Morris-Pratt, pour la recherche de la chaine `mot` dans la chaine `texte`, est le suivant.

- Tout d'abord, on détermine la liste `B` des longueurs des bords des préfixes de `mot`, comme à la question 6.
- On initialise alors à 0 une variable `i` qui représente l'indice dans la chaine `texte` auquel on commencera la recherche de `mot`.
- On initialise également à 0 une variable `j` qui représente l'indice d'un caractère dans la chaine `mot` : celui que l'on comparera avec les caractères de `texte`.
- Tant que `i` est un indice valide dans la chaine `texte` :
  - ⋆ on détermine le plus petit entier `j` tel que les caractères `texte[i + j]` et `mot[j]` soient différents (c'est-à-dire le plus grand entier `j` tel que les sous-chaines `texte[i : i + j]` et `mot[ : j]` soient égales) ;
  - ⋆ si `j` est égal à la longueur de `mot`, on a trouvé la chaine `mot` dans `texte` à l'indice `i` ;
  - ⋆ si `j` est égal à 0, on se décale d'un seul caractère (comme dans l'algorithme naïf) et on recommence la recherche de `mot` à l'indice `i + 1`, à partir de l'indice `j = 0` dans `mot` ;
  - ⋆ sinon, on se décale de `j - B[j - 1]` caractères et on cherche `mot` à l'indice `i + j - B[j - 1]`, en partant de l'indice `j = B[j - 1]` dans `mot`.

7. Implémentons l'algorithme de Knuth-Morris-Pratt pour trouver le plus petit indice auquel une chaine `mot` commence et est contenue dans une chaine `texte` (si un tel indice existe).

```
def Knuth_Morris_Pratt(mot, texte):
    """
    Entrées : deux chaines de caractères mot et texte.
    Sortie : le premier indice auquel la chaine mot commence
             et est contenue dans texte, si mot est contenue
             dans texte, un message d'erreur sinon.
    """
    n = len(texte)
    p = len(mot)
    B = longueurs_bords(mot)
    i = 0
    j = 0
    while i < n:
    # Tant que i est un indice valide dans texte :

        # On commence par chercher le plus petit entier j tel que
        # les caractères texte[i + j] et mot[j] soient différents.

        while (j < p) and (texte[i + j] == mot[j]):
        # Tant que j est un indice valide dans mot
        # et que les caractères texte[i + j] et mot[j]
        # sont différents :
            j = j + 1

        if j == p:
            # Si on a trouvé la chaine mot à l'indice i,
            return i
                # on retourne l'indice i, ce qui arrête la fonction.

        elif j == 0:
            # Si la recherche a échoué dès le premier caractère.
            i = i + 1
                # on incrémente i de 1, et j reste à 0.

        else:
            # Sinon,
            i = i + j - B[j - 1]
            j = B[j - 1]
                # on incrémente i de j - B[j - 1]
                # et j prend pour valeur B[j - 1].

    # Si l'on sort de la boucle sans avoir trouvé mot,
    # on renvoie un message d'erreur.
    return "La première chaine n'est pas contenue
            dans la seconde."
```