Devoir Surveillé d'ITC N°1
le 19/12/2025
MPSI & PCSI

Consignes

- Les codes doivent être présentés en mentionnant explicitement l'indentation, au moyen par exemple de barres verticales sur la gauche.
- Pour qu'un code soit compréhensible, il convient de choisir des noms de variables les plus explicites possibles.
- La numérotation des exercices (et des questions) doit être respectée et mise en évidence. Les résultats (hors questions purement informatiques) doivent être encadrés proprement.
- Il est important de numéroter correctement les pages des copies qui seront données à la correction. Chaque candidat est responsable de la vérification de son sujet d'épreuve : pagination et impression de chaque page.
- Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il convient de le signaler sur la copie et de poursuivre la composition en expliquant les raisons des initiatives qui ont été prises.
- Les candidats ne doivent avoir aucune communication entre eux ou avec l'extérieur durant l'épreuve. Aussi, l'utilisation des téléphones portables et, plus largement, de tout appareil permettant des échanges ou la consultation d'informations, est interdite.
- À l'issue de la durée prévue pour cette épreuve, les candidats doivent déposer le stylo et ne sont plus autorisés à écrire quoi que ce soit sur leur copie. Tout retard donne lieu à une pénalité sur la note finale.
- **L'usage de la calculatrice est interdit.**

- Les signatures des fonctions devront toutes être écrites.
- Les codes devront être les plus « optimisés » possibles, lorsqu'une telle optimisation semble évidente. Par exemple, *via* l'utilisation de boucles ayant un nombre minimal d'itérations.

Problème **Autour du séquençage du génome** Dans ce sujet, on s'intéresse à la recherche d'un motif dans une molécule d'ADN. Une molécule d'ADN est constituée de deux brins complémentaires, qui sont un long enchaînement de nucléotides de quatre types différents désignés par les lettres A, T, C et G. Pour simplifier le sujet, on va considérer qu'une *molécule d'ADN* est une chaîne de caractères sur l'alphabet A, C, G, T (on s'intéresse donc seulement à un des deux brins). On parlera de séquence d'ADN.

Partie I — Génération d'une séquence d'ADN

On considère la chaîne de caractères : `seq = 'ATCGTACGTACG'`.

1. Que renvoie la commande `seq[3]` ? Que renvoie la commande `seq[2:6]` ?

Les fonctions que nous allons construire par la suite devront prendre en paramètre une chaîne de caractères ne contenant que les lettres A, C, G, T (ceci correspond à une séquence d'ADN). Nous allons commencer par construire aléatoirement une séquence d'ADN.

Pour générer aléatoirement une séquence d'ADN composée de n caractères, on utilisera le principe suivant :

- On commence par créer une chaîne de caractères vide.
- On tire alors aléatoirement n chiffres entiers compris entre 0 et 3 et :
  - si on obtient 0, alors, on ajoute 'A' à notre chaîne de caractères ;
  - si on obtient 1, alors, on ajoute 'C' à notre chaîne de caractères ;
  - si on obtient 2, alors, on ajoute 'G' à notre chaîne de caractères ;
  - si on obtient 3, alors, on ajoute 'T' à notre chaîne de caractères.
- On renvoie la chaîne de caractères ainsi construite.

2. Écrire une fonction `generation` qui prend en paramètre un entier n et qui renvoie une chaîne de caractères aléatoires de longueur n ne contenant que des 'A', 'C', 'G' et 'T'. On supposera que le module `numpy.random` est importé et on rappelle que la commande `randint(a, b)` renvoie alors un entier aléatoirement entre a et b-1.

Partie II — Recherche d'un motif

Considérons une chaîne de caractères `S = 'ACTGGTCACT'`. On appelle *sous-chaîne de caractères de* S une suite de caractères incluse dans S. Par exemple, `'TGG'` est une sous-chaîne de S mais `'TAG'` n'en est pas une.

L'objectif de cette partie et des suivantes est de rechercher une sous-chaîne de caractères `M` de longueur m appelée *motif* dans une chaîne de caractères `S` de longueur n.

Il s'agit d'une problématique classique en informatique, qui répond aux besoins de nombreuses applications. On trouve plus de 100 algorithmes différents pour cette même tâche, les plus célèbres datant des années 1970, mais plus de la moitié ont moins de 10 ans.

Dans la partie suivante, nous allons d'abord nous intéresser à l'algorithme naïf, puis dans les suivantes, nous nous pencherons sur plusieurs autres algorithmes. Les différentes parties sont indépendantes.

II — 1) ALGORITHME NAÏF

Le principe de l'algorithme naïf est le suivant :

- on parcourt la chaîne `S`, disons que l'on se trouve en indice i.
  - ◇ On regarde si le motif `M` est présent dans `S` à partir de l'indice `i`.
  - ◇ Si ce n'est pas le cas, on recommence avec l'élément suivant de la chaîne de caractères.

Par exemple, si on recherche le motif `'FOR'` dans la chaine `'INFORMATIQUE'`, on compare d'abord le motif `'FOR'` avec `'INF'` (les motifs ne correspondent pas!), puis le motif `'FOR'` avec `'NFO'` (les motifs ne correspondent toujours pas!), puis le motif `'FOR'` avec `'FOR'` (les motifs sont les mêmes).

3. Soit `S` une chaîne de caractères. Rappeler la commande, faisant appel à du slicing, permettant d'obtenir une sous-chaîne de S, constituée des éléments de S dont les indices sont compris entre les indices `debut` (inclus) et `fin` (inclus).

4. Écrire une fonction `recherche(S:str,M:str)->int`, faisant appel à du slicing, qui à une sous-chaîne de caractères S et une chaîne de caractères M renvoie -1 si M n'est pas dans S, et la position de la première lettre de la chaîne de caractères M si M est présente dans S. Par exemple, `recherche('INFORMATIQUE','FOR')` renvoie 2, tandis que `recherche('INFORMATIQUE','FORA')` renvoie -1.

On pourrait démontrer que cette première méthode naïve est complètement inefficace lorsqu'on l'applique à de très longues chaines de caractères.

II — 2) ALGORITHME DE KARP-RABIN (1987)

L'algorithme de KARP-RABIN a pour objectif d'améliorer l'algorithme naïf, mais en éliminant des comparaisons de chaînes (qui peuvent être coûteuses). L'idée est de remplacer les chaînes `S` par un entier `f(S)` où f est une fonction à valeurs entières sur l'ensemble des chaînes, appelée *fonction de hachage*. On s'intéresse tout d'abord à un premier exemple simple de telle fonction, puis on met en oeuvre son utilisation pour la recherche de motifs. Enfin on étudie une deuxième telle fonction appelée *empreinte de Rabin*.

5. [1er exemple de fonction f et algorithme] On décide par exemple d'associer à une chaîne `S` le nombre de fois que la lettre 'A' apparait dedans, noté ici `f(S)`.

5.1) Écrire une fonction itérative `nbA(ch:str)->int` à qui on fournit une chaîne `ch` et qui renvoie le nombre de 'A' présents dans `ch`.

5.2) L'algorithme de RABIN-KARP consiste à, dans l'algorithme naïf, effectuer la comparaison entre `M` et une sous-chaine de `S` à l'indice `i` uniquement si les images par f de ces deux sous-chaînes coïncident.
Recopier et compléter la fonction ci-après pour qu'elle implémente ce principe.

```
def recherche_RK1(S:str,M:str)->int:
    m = len(M)
    n = len(S)
    f_M = nbA(M)
    for i in range(_____________):
        S_i = _____________ # chaîne extraite de S à \
        ↪ l'indice i de longueur m
        if nbA(S_i) == f_M :
            if S_i == M:
                return _____________
    return _____________
```

5.3) Une amélioration simple de la fonction précédente peut être obtenue. En effet, à chaque itération `i`, `nbA(S_i)` est recalculé alors que l'on pourrait utiliser la valeur de l'itération précédente. En effet, `S_i` à l'itération suivante partage $m-1$ lettres en commun avec `S_i` de l'itération précédente. On se sert alors de la fonction `nbA` uniquement pour l'initialisation.
Écrire une nouvelle fonction `recherche_RK2(S:str,M:str)->int` exploitant ce principe. En quoi cet algorithme est-il meilleur que l'algorithme naïf codé dans la précédente sous-partie ?

6. [2ème exemple de fonction f] Voici un autre exemple de conversion d'une chaine sur l'alphabet A,T,G,C en entier :

- à chaque caractère de l'alphabet, on associe une valeur. Ici, on va associer à 'A' la valeur 0, à 'C' la valeur 1, à 'G' la valeur 2 et à 'T' la valeur 3. Pour un motif de taille n, on obtient donc une suite de chiffre a_{n-1}, ..., a_1, a_0. Par exemple, à la chaîne `'TAGC'`, on lui associe la suite de chiffre 3021 ;
- cette suite de chiffres peut être considérée comme l'écriture d'un entier en base 4. On convertit ensuite cet entier en base 10 en calculant
  $$N = a_{n-1} \times 4^{n-1} + a_{n-2} \times 4^{n-2} + \ldots + a_1 \times 4 + a_0 ;$$
  pour 3021 : $n = 4$, $a_3 = 3$, $a_2 = 0$, $a_1 = 2$, $a_0 = 1$.
- on calcule enfin le reste de la division euclidienne de N par 13.

Dans cette question, on s'intéresse uniquement à la fonction de hachage, et pas à son utilisation dans l'algorithme de KARP-RABIN.

6.1) On ne considère dans cette question que des motifs de taille 3. Quels nombres obtient-on en convertissant les motifs `'CCC'`, `'ACG'` et `'GAG'` ?
Pour cela, recopier le tableau suivant sur la copie puis le compléter :

Motif	Suite de chiffres	Entier en base 10	Reste de la division par 13
'CCC'			
'ACG'			
'GAG'			

6.2) On souhaite maintenant écrire une fonction à qui on fournit une chaîne de caractères (qui représente une séquence d'ADN) et qui renvoie son codage avec la méthode précédente.

i) Écrire une fonction itérative `sommeit(L:list)->int` à qui on fournit une liste `L` $= [a_{n-1}, a_{n-2}, ..., a_1, a_0]$ constituée de nombres entiers compris entre 0 et 3, et qui renvoie la somme : $\sum_{k=0}^{n-1} a_k 4^k$.

ii) On se propose maintenant d'écrire une fonction récursive `sommerec(L:list)->int` qui agit comme la fonction précédente. On pourra par exemple s'appuyer sur l'algorithme de HORNER et utiliser l'écriture suivante :

$$\sum_{k=0}^{n-1} a_k 4^k = a_0 + 4 \times [a_1 + 4 \times (a_2 + 4 \times (a_{n-2} + 4a_{n-1}))].$$

iii) Recopier la fonction suivante sur votre copie et la compléter afin qu'à une chaîne de caractères `ch`, elle renvoie le résultat de sa conversion décrite précédemment.

```
def f(ch:str)->int:
    seq = []
    codage = {'A': 0, 'C': 1, 'G': 2, 'T': 3} # dictionnaire
    for car in ch:
        seq.append(.....)
    return sommerec(.....) % 13
```

II — 3) ALGORITHME DICHOTOMIQUE UTILISANT LA STRUCTURE DE LISTE

Une autre possibilité pour chercher un motif dans une chaîne de caractères (ou séquence d'ADN) est de construire une liste contenant tous les sous-motifs de notre chaîne, triés par ordre alphabétique, puis de faire la recherche dans cette liste.

Par exemple, à la chaîne `'CATCG'`, on peut associer la liste :

```
['C', 'A', 'T', 'G', 'CA', 'AT', 'TC', 'CG', 'CAT', 'ATC', 'TCG',
                    'CATC', 'ATCG', 'CATCG']
```

que l'on peut ensuite trier pour obtenir la liste :

```
['A', 'AT', 'ATC', 'ATCG', 'C', 'CA', 'CAT', 'CATC', 'CATCG', 'CG',
                    'G', 'T', 'TC', 'TCG']
```

La première étape de cette méthode est donc de trier une liste.

7. Écrire une fonction `position_max` qui prend pour paramètre une liste `L` non vide de nombres et qui renvoie l'indice de la première occurrence du maximum de la liste `L`.

8. Recopier la fonction suivante sur votre copie et compléter les pointillés afin qu'elle trie une liste `L` fournie en paramètre.

```
def tri(L:list)->None:
    n = len(L)
    for i in range(n-1):
        p = position_max(L[.....])
        L[.....], L[.....] = L[.....], L[.....]
```

Vous noterez que cette fonction agit donc sur `L` par effets de bord, et ne renvoie rien.

9. Quel est le nom de cette méthode de tri déjà étudiée en TP ?

10. La méthode de tri précédemment écrite, prévue pour trier une liste de nombres fonctionne-t-elle encore si on l'utilise pour trier une liste de chaînes de caractères ? Justifier votre réponse.

11. On propose alors la fonction suivante, incomplète, à qui on fournit :

- une liste triée `L` de chaînes de caractères constituée de tous les sous-motifs d'une séquence d'ADN.
- un motif `M`

et qui renvoie -1 si `M` n'est pas dans `L`, et la position de `M` dans `L` sinon :

```
def recherche2(M:str, L:list)->int:
    """
    la liste L est triée et on y recherche le motif M
    """
    debut = .....
    fin = .....
    while debut < fin:
        m = (debut + fin) // 2
        if L[m] < M:
            debut = .....
        else:
            fin = ...
    if L[debut] == M:
        return .....
```

```
    else:
        return -1
```

11.1) Quel nom porte la méthode proposée dans cette dernière fonction ?

11.2) Recopier la fonction précédente sur votre copie et compléter les pointillés afin qu'elle réalise l'opération attendue.

II — 4) Algorithme de Knuth-Morris-Pratt (1970)

Lorsqu'un échec a lieu dans l'algorithme naïf, c'est-à-dire lorsqu'un caractère du motif est différent du caractère correspondant dans la séquence d'ADN, la recherche reprend à la position suivante en repartant au début du motif. Pour améliorer la recherche, nous allons introduire les notions de *préfixe* et de *suffixe*.

- Un *préfixe* d'un motif M est un motif, **différent de M**, qui est un début de M. Par exemple, 'mo' et 'm' sont des préfixes de 'mot', mais 'mot' n'est pas un préfixe de 'mot'.
- Un *suffixe* d'un motif M est un motif, **différent de M**, qui est une fin de M. Par exemple, 'ot' et 't' sont des suffixes de 'mot', mais 'mot' n'est pas un suffixe de 'mot'.

12. Donner tous les préfixes et les suffixes du motif `'ACGTAC'`.

13. Quel est le plus long préfixe de `'ACGTAC'` qui soit aussi un suffixe ? Quel est le plus long préfixe de `'ACAACA'` qui soit aussi un suffixe ?

L'algorithme de Knuth-Morris-Pratt (KMP) est un algorithme reposant essentiellement sur une fonction annexe appelée `KMP_aux` dans la suite. On ne s'intéresse pas dans ce sujet à l'algorithme KMP complet, mais uniquement à plusieurs méthodes d'implémentation de `KMP_aux`, qui prend en argument un motif `M`.

L'objectif de cette dernière est de renvoyer une liste `F`, telle que `F[i]` contienne, pour chaque lettre de `M` à la position `i`, la longueur du plus grand sous-mot de `M` qui finit par la lettre `M[i]` (donc le plus grand suffixe de `M[:i+1]`) qui soit aussi un préfixe de M.

14. On considère que le motif `M` est `'ACGTAC'` et on note `F = KMP_aux(M)`.

14.1) Quelle est la taille de la liste `F` ?

14.2) Quelle information sur `F` donne le résultat de la question 13 ?

15. On considère que le motif `M` est `'ACAACA'` et on note `F = KMP_aux(M)`.

15.1) Quelle est la taille de la liste `F` ?

15.2) Quelle information sur `F` donne le résultat de la question 13 ?

16. On souhaite proposer ici une première version de `KMP_aux`.

16.1) Écrire une fonction `calc_pref(M:str)->list` qui renvoie une liste des préfixes de `M` triés dans l'ordre croissant de longueur. par exemple, `calc_pref("ACCT")` renverra : `["A", "AC", "ACC"]`.

16.2) Écrire une fonction récursive `renverse(M:str)->str` qui renvoie la chaîne miroir de `M`. Par exemple, `renverse("ACCT")` renverra `"TCCA"`.

16.3) Écrire une fonction `calc_suff(M:str)->list` qui renvoie une liste des suffixes de `M` triés dans l'ordre croissant de longueur. par exemple, `calc_suff("ACCT")` renverra : `["T", "CT", "CCT"]`.

16.4) Écrire enfin une première version de la fonction `KMP_aux`.

17. On propose le code suivant :

```
def KMP_aux(M:str)->list:
    F = [0]
    i = 1
    j = 0
    m = len(M)
    while i < m :
        if M[i] == M[j]:
            F.append(j+1)
            i = i+1
            j = j+1
        else:
            if j > 0:
                j = F[j-1]
            else:
                F.append(0)
                i = i+1
    return F
```

17.1) Décrire l'exécution de la fonction `fonctionannexe` lorsque `M = 'ACAACA'` en recopiant sur votre copie et complétant le tableau suivant, qui décrit l'évolution du contenu des variables `i`, `j` et `F`.

	i	j	F
Fin du premier passage dans la boucle while			
Fin du deuxième passage dans la boucle while			
Fin du troisième passage dans la boucle while			
Fin du quatrième passage dans la boucle while			
Fin du cinquième passage dans la boucle while			
Fin du sixième passage dans la boucle while			

17.2) Quel est l'avantage de cette nouvelle version par rapport à l'ancienne ?

Correction
du Devoir Surveillé d'ITC N°1 (le 19/12/2025)
MPSI & PCSI

Solution

PARTIE I — GÉNÉRATION D'UNE SÉQUENCE D'ADN

1. `seq[3]` renvoie 'G' et `seq[2:6]` renvoie 'CGTA'.
2. Fonction `generation` :

```
def generation(n:int)->str:
    seq = ''
    for _ in range(n):
        x = randint(0,4)
        if x == 0:
            seq += 'A'
        elif x == 1:
            seq += 'C'
        elif x == 2:
            seq += 'G'
        else:
            seq += 'T'
    return seq
```

ou plus simplement :

```
def generation(n):
    seq = ''
    codage = ['A','C','G','T']
    for _ in range(n):
        seq += codage[randint(0,4)]
    return seq
```

PARTIE II — RECHERCHE D'UN MOTIF

3. Il s'agit de `S[debut:fin+1]`.
4. Fonction `recherche` :

```
def recherche(S:str,M:str)->int:
    m = len(M)
    n = len(S)
    for i in range(n-m+1):
        if S[i:i+m] == M:
            return i
    return -1
```

5. 5.1)

```
def nbA(ch:str)->int:
    a = 0
    for x in ch:
        if x == "A":
            a += 1
    return a
>>> nbA("AAAGTC")
3
```

5.2)

```
def recherche_RK1(S:str,M:str)->int:
    m = len(M)
    n = len(S)
    f_M = nbA(M)
    for i in range(n-m+1):
        S_i = S[i:i+m] # chaîne extraite de S à l'indice i de longueur m
        if nbA(S_i) == f_M :
            if S_i == M:
                return i
    return -1
>>> recherche_RK1("ATGCCCCGTACG", "TAG")
-1
>>> recherche_RK1("ATGCCCCGTACG", "TAC")
8
```

5.3)

```
def recherche_RK2(S:str,M:str)->int:
    m = len(M)
    n = len(S)
    f_M, f_S = nbA(M), nbA(S[0:m])
    for i in range(n-m+1):
        S_i = S[i:i+m] # chaîne extraite de S à l'indice i de longueur m
        if f_S == f_M :
            if S_i == M:
```

```
                return i
            # Actualisation de f_S
            if S_i[0] == "A":
                f_S -= 1
            if i+m < n and S[i+m] == "A": # 1er caractère \
            ↪ dans S à droite de S_i
                f_S += 1
        return -1
>>> recherche_RK2("ATGCCCCGTACG", "TAG")
-1
>>> recherche_RK2("ATGCCCCGTACG", "TAC")
8
```

L'utilisation d'une fonction f permet d'éliminer un nombre important de comparaisons de chaînes ; l'actualisation de `nbA(S_i)` se faisant quant à elle avec seulement quelques opérations simples.

6. **6.1)** On obtient :

Motif	Suite de chiffres	Entier en base 10	Reste de la division par 13
'CCC'	111	21	8
'ACG'	012	6	6
'GAG'	202	34	8

6.2) **i)** Voici une première fonction `sommeit`, peu efficace, car pour le calcul de 4^k, on repart à chaque fois du début :

```
def sommeit(L:list)->int:
    S = 0
    n = len(L)
    for k in range(n):
        S = S+L[n-1-k]*4**k
    return S
```

La fonction suivante est meilleure car on garde en mémoire la dernière puissance de 4 calculée.

```
def sommeit(L:list)->int:
    S = 0
    n = len(L)
    puiss = 1
    for k in range(n):
        S = S+L[n-1-k]*puiss
        puiss = puiss*4
    return S
```

ii) Fonction `sommerec` :

```
def sommerec(L:list)->int:
    if len(L) == 1:
        return L[0]
    else:
        return L[-1] + 4*sommerec(L[:-1])
```

iii) Fonction `conversion` :

```
def conversion(ch:str)->int:
    seq = []
    codage = { 'A': 0, 'C': 1, 'G': 2, 'T': 3 } # \
    ↪ dictionnaire
    for car in ch:
        seq.append(codage[car])
    return sommerec(seq) % 13
>>> conversion("CCC")
8
>>> conversion("ACG")
6
>>> conversion("GAG")
8
```

7. Fonction `position_max` :

```
def position_max(L:list)->int:
    max = L[0]
    pos = 0
    n = len(L)
    for i in range(1,n):
        if L[i] > max:
            max, pos = L[i], i
    return pos
```

8. Fonction `tri` :

```
def tri(L:list)->None:
    n = len(L)
    for i in range(n-1):
        p = position_max(L[:n-i])
        L[p], L[n-i-1] = L[n-i-1], L[p]
```

9. Il s'agit du tri par sélection.

10. Oui, la méthode précédente fonctionnera encore. Lors du test, `if L[i] > max`, on comparera deux chaînes de caractères à l'aide de l'ordre lexicographique.

11. **11.1)** Il s'agit d'une méthode de recherche par dichotomie.

11.2)

```
def recherche2(M:str, L:list)->int:
```

```
    debut = 0
    fin = len(L)-1
    while debut < fin:
        m = (debut + fin) // 2
        if L[m] < M:
            debut = m+1
        else:
            fin = m
    if L[debut] == M:
        return debut
    else:
        return -1
```

12. Les préfixes de `'ACGTAC'` sont 'A', 'AC', 'ACG', 'ACGT' et 'ACGTA', et ses suffixes sont 'C', 'AC', 'TAC', 'GTAC' et 'CGTAC'.

13. Le plus long préfixe de `'ACGTAC'` qui soit aussi un suffixe est `'AC'`. Le plus long préfixe de `'ACAACA'` qui soit aussi un suffixe est `'ACA'`.

14. On considère que le motif `M` est `'ACGTAC'` et on note `F = KMP_aux(M)`.

14.1) `F` possède autant d'éléments que de lettres dans `M`, donc est de longueur 6 pour ce motif.

14.2) La question 13 fournit alors la dernière coordonnée de `F`, qui est donc 2.

15. On considère que le motif `M` est `'ACGTAC'` et on note `F = KMP_aux(M)`.

15.1) `F` possède autant d'éléments que de lettres dans `M`, donc est de longueur 6 pour ce motif.

15.2) La question 13 fournit alors la dernière coordonnée de `F`, qui est donc 3.

16. 16.1)

```
def calc_pref(M:str)->list:
    L, pref = [], ""
    for c in M[:-1]:
        pref += c # à l'itération k, pref contient M[:k]
        L.append(pref)
    return L
```

```
>>> calc_pref("ACCT")
['A', 'AC', 'ACC']
```

16.2)

```
def renverse(M:str)->str:
    if len(M) == 0:
        return ""
    else:
        return M[-1] + renverse(M[:-1])
```

```
>>> renverse("ACCT")
'TCCA'
```

16.3)

```
def calc_suff(M:str)->list:
    M_r = renverse(M)
    L_suff = calc_pref(M_r) #les suffixes, mais renversés
    return [renverse(suff) for suff in L_suff]
```

```
>>> calc_suff("ACCT")
['T', 'CT', 'CCT']
```

16.4)

```
def KMP_aux(M:str)->list:
    F = [0]
    m = len(M)
    L_pref = calc_pref(M)
    for i in range(1, m):
        L_suff = calc_suff(M[:i+1])
        # on parcourt les suffixes à rebours, et on \
        ↪ regarde si c'est un préfixe
        j = i-1 # dernier indice de M[:i+1]
        while j >= 0 and L_suff[j] != L_pref[j]:
            j -= 1
        F.append(j+1)
    return F
```

```
>>> KMP_aux("ACAACA")
[0, 0, 1, 1, 2, 3]
>>> KMP_aux("ACGTAC")
[0, 0, 0, 0, 1, 2]
>>> KMP_aux("ACAACA")
[0, 0, 1, 1, 2, 3]
```

17. 17.1) Voici le tableau complété :

	i	j	F
Fin du premier passage dans la boucle while	2	0	[0,0]
Fin du deuxième passage dans la boucle while	3	1	[0,0,1]
Fin du troisième passage dans la boucle while	3	0	[0,0,1]
Fin du quatrième passage dans la boucle while	4	1	[0,0,1,1]
Fin du cinquième passage dans la boucle while	5	2	[0,0,1,1,2]
Fin du sixième passage dans la boucle while	6	3	[0,0,1,1,2,3]

17.2) La fonction `KMP_aux` sera beaucoup plus rapide, car elle ne nécessite qu'une seule boucle de parcours de `M`.