Le sujet ayant été posé avant l'avènement de Python comme langage "officiel", les tableaux sont supposés indexés à partir de 1. Pour simplifier, puisque je coderai en Python, conformément au programme, je considérerai qu'ils le sont à partir de 0. Il faudra toutefois prendre garde qu'avec Python $tab[a \dots b]$ s'écrira `tab[a:b+1]`, puisque b doit être compris.

Quant aux primitives `allouer(n)` et `taille(t)`, elles renverront respectivement `[0]*n` et `len(t)`.

Partie I. Méthode directe

Question 1

L'énoncé autorise à considérer que la précondition $0 \leq k < n$ est acquise, mais ne parle pas de $k+m \leq n$, qu'il faut donc contrôler. Je le fais d'emblée. Lorsque c'est le cas, je lance une boucle dont je sors à la première discordance.

```
def enTeteDeSuffixe(mot,tab,k):
    m=taille(mot)
    if k+m>taille(tab): return False
    for j in range(m):
        if mot[j]!=tab[k+j]: return False
    return True
```

Question 2

L'énoncé a dit "algorithme simple", utilisons donc la fonction précédente, en limitant la valeur de k aux cas où le mot à chercher n'est pas trop long !

```
def rechercherMot(mot,tab):
    for k in range(taille(tab)-taille(mot)+1):
        if enTeteDeSuffixe(mot,tab,k): return True
    return False
```

Question 3

Avec la définition de l'énoncé, il suffit de compter le nombre de fois où *mot* est en tête d'un suffixe de *tab*.

Pour cela je reprends le principe précédent et j'ajoute un compteur :

```
def compterOccurrences(mot,tab):
    nb=0;
    for k in range(taille(tab)-taille(mot)+1):
        if enTeteDeSuffixe(mot,tab,k): nb+=1
    return nb
```

Question 4

Je calcule d'abord le nombre d'occurrences de chaque lettre, puis je divise par le nombre de caractères dans *tab* pour obtenir les fréquences.

Côté complexité, il serait très maladroit ici d'appeler 26 fois la fonction précédente (qui balayera complètement t à chaque appel) alors qu'un seul parcours de *tab* suffit, dès l'instant que l'on a créé le tableau de 26 cases requis (il est bien initialisé à 0 par ma fonction `allouer` !). En effet il suffit, partant de 0 pour chaque case, d'incrémenter de 1 l'effectif correspondant à chacune des lettres de *tab* (1re boucle `for`).

Vu l'exemple de l'énoncé, il semble que le mot *fréquence* soit utilisé à tort, ce sont les nombres d'occurrences qui sont donnés…

```
def frequenceLettres(tab):
    n=taille(tab)
    f=allouer(26)
    for k in range(n): f[tab[k]]+=1
    return f
```

Question 5

Le plus rapide serait sans doute ici d'adapter l'algorithme précédent, en créant un tableau de 26×26 (une case par digramme), mais les tableaux à 2 dimensions ne sont pas évoqués dans l'énoncé. De plus, l'indication (fonction `afficherMot` qui ne servirait à rien avec cette première idée, puisque l'on ne sait pas à quel endroit apparaît tel ou tel digramme) laisse supposer que l'idée envisagée est la suivante : balayer *tab* pour chaque digramme afin de déterminer son nombre d'occurrences. Comme expliqué précédemment, cette solution est mauvaise pour n grand, mais puisque l'énoncé donne le feu vert... L'avantage est de pouvoir réutiliser la fonction de la question 3.

Je commence par écrire (hors sujet) la fonction `afficherMot` renvoyant une chaîne de caractères, à l'aide de la fonction Python `chr` : `chr(a)` renvoie le caractère de code ASCII `a`. Il suffit alors de savoir que les codes ASCII des minuscules `'a'` à `'z'` sont les entiers de 97 à 122.

```
def afficherMot(tab,i,k):
    s=''
    for j in range(k):
        s+=chr(96+tab[i+j])
    return s
```

Puis la fonction demandée, où je construis au fur et à mesure la liste des digrammes déjà rencontrés et en parallèle celle de leurs nombres d'occurrences ; pour éviter les redondances, je prends soin de vérifier pour chaque nouveau digramme s'il n'a pas déjà été rencontré (je triche un peu avec la commande `in` de Python ; il suffirait d'une boucle pour la reprogrammer...) :

```
def afficherFrequenceBigramme(tab):
    n=taille(tab)
    mot=allouer(2)
    bigrs=[];occs=[]
    for i in range(n-1):
        mot[0]=tab[i];mot[1]=tab[i+1]
        bigr=afficherMot(mot,0,2)
        if bigr not in bigrs:
            bigrs.append(bigr)
            occs.append(compterOccurrences(mot,tab))
    for j in range(len(bigrs)):
        print(bigrs[j],occs[j])
```

Partie II. Tableau des suffixes

Question 6

La comparaison lexicographique se prête bien à une programmation récursive, puisque par principe on compare les deux premières lettres de chaque chaîne et, en cas d'égalité, les chaînes obtenues en supprimant lesdites premières lettres. Attention toutefois à la gestion des cas d'arrêt ! Lorsque les deux suffixes sont vides ils sont égaux, si un seul des deux est vide il précède l'autre. Les trois dernières lignes traitent les suffixes non vides, où j'applique le principe précédent. L'appel récursif intervient lorsque les deux suffixes sont non vides et commencent par la même lettre. Le programme se terminera bien puisque `k1` et `k2` augmentent strictement à chaque appel récursif et qu'il y a arrêt dès que l'un des deux dépasse `n`.

```
def comparerSuffixes(tab,k1,k2):
    n=taille(tab)
    if k1>=n and k2>=n: return 0
    if k1>=n: return -1
    if k2>=n: return 1
    if tab[k1]<tab[k2]: return -1
    if tab[k1]>tab[k2]: return 1
    return comparerSuffixes(tab,k1+1,k2+1)
```

Plus naturelle pour certains cerveaux, voici une version itérative. J'avance dans les deux suffixes tant qu'ils sont non vides et que je ne peux pas conclure, puis je traite les cas où l'un des deux s'est vidé.

```
def comparerSuffixesIter(tab,k1,k2):
    n=taille(tab)
    while k1<n and k2<n:
        if tab[k1]<tab[k2]: return -1
        if tab[k1]>tab[k2]: return 1
        k1+=1;k2+=1
    if k1>=n and k2>=n: return 0
    if k1>=n: return -1
    return 1
```

Question 7

Il s'agit de trier le tableau $[\![0, n[\![$ suivant l'ordre des suffixes correspondants. L'énoncé ne réclame pas de complexité particulière, mais le tri rapide est si facile à écrire en Python... Un tri quadratique serait mal vu ! Prendre garde à utiliser la relation d'ordre définie à la question précédente !

```
def calculerSuffixes(tab):
    def quicksort(t):
        if len(t)<2: return t
        pivot=t[0];t1=[];t2=[]
        for x in t[1:]:
            if comparerSuffixes(tab,x,pivot)<0:
                t1.append(x)
            else:
                t2.append(x)
        return quicksort(t1) + [pivot] + quicksort(t2)
    return quicksort(list(range(len(tab))))
```

Partie III. Exploitation du tableau des suffixes

Question 8

J'adapte la version itérative de la question 6 en comparant les premiers caractères de `mot` à ceux du suffixe et en renvoyant 0 si `mot` a été épuisé. Si c'est le suffixe qui a été épuisé (dernière ligne), c'est que `mot` le suit dans l'ordre lexicographique.

```
def comparerMotSuffixe(mot,tab,k):
    m=taille(mot);n=taille(tab)
    i=0
    while i<m and k<n:
        if mot[i]<tab[k]: return -1
        if mot[i]>tab[k]: return 1
        i+=1;k+=1
    if i>=m: return 0
    return 1
```

Pour une version récursive efficace, il faudrait utiliser une fonction auxiliaire avec un 2$^{\text{e}}$ paramètre entier jouant le rôle du `i` ci-dessus.

Question 9

Recherche dichotomique classique, sauf que la relation d'ordre est particulière... On divise en deux les sous-tableaux successifs de `tabS` en faisant en sorte de garder `mot` entre les "bornes" desdits sous-tableaux. Je m'arrête dès que c=0 puisqu'alors j'ai trouvé `mot`. À la sortie de la boucle, il n'y a plus que la valeur `g` à tester.

```
def rechercherMot2(mot,tab,tabS):
    g=0;d=taille(tab)-1
    while g<d:
        m=(g+d)//2
        c=comparerMotSuffixe(mot,tab,tabS[m])
        if c==0: return True
        if c<=0: d=m
        else: g=m+1
    return comparerMotSuffixe(mot,tab,tabS[g])==0
```

Question 10

Il faut sans doute comprendre par "comparaison de mot" un appel à la fonction `enTeteDeSuffixe` dans le premier cas et `comparerMotSuffixe` dans le second. Alors en moyenne et dans le pire des cas, la complexité de `rechercherMot` est de l'ordre de n tandis que celle de `rechercherMot2` est de l'ordre de $\log_2 n$. Cette dernière semble donc avantageuse, mais il faut y ajouter le "coût" de la création du tableau `tabS`, qui est nécessairement supérieur à n...

Donc pour une seule recherche la version naïve est compétititive. Mais le coût de création de `tabS` n'est à payer qu'une fois pour toutes les recherches dans le même texte. C'est ce qui fait l'intérêt de cette méthode dans le cas de pages Internet, où le texte change peu souvent et où les recherches sont nombreuses.

Question 11

Puisque 0 est pour nous un indice possible, je choisis de renvoyer -1 dans le cas où `mot` n'apparaît pas dans `tab`.
J'adapte la recherche dichotomique précédente pour renvoyer le résultat souhaité. Il ne faut plus s'arrêter dès que `mot` a été trouvé, mais continuer à faire diminuer `d` tant que `c<=0`.

```
def rechercherPremierSuffixe(mot,tab,tabS):
    g=0;d=taille(tab)-1
    while g<d:
        m=(g+d)//2
        c=comparerMotSuffixe(mot,tab,tabS[m])
        if c<=0: d=m
        else: g=m+1
    if comparerMotSuffixe(mot,tab,tabS[g])==0: return g
    else: return -1
```

Pour la question suivante, j'écris la version "symétrique".

```
def rechercherDernierSuffixe(mot,tab,tabS):
    g=0;d=taille(tab)-1
    while g<d:
        m=1+(g+d)//2
        c=comparerMotSuffixe(mot,tab,tabS[m])
        if c>=0: g=m
        else: d=m-1
    if comparerMotSuffixe(mot,tab,tabS[g])==0: return g
    else: return -1
```

Question 12

Le nombre d'occurrences de `mot` dans `tab` est le nombre de suffixes de `tab` commençant par `mot`, la fonction demandée est donc triviale (on peut éviter le second calcul si le premier renvoie 0 !).

```
def compterOccurrences2(mot,tab,tabS):
    prem=rechercherPremierSuffixe(mot,tab,tabS)
    if prem<0: return 0
    else: return rechercherDernierSuffixe(mot,tab,tabS)-prem+1
```

Question 13

Je reprends le principe de la question 5, pour éviter d'afficher chaque mot de k lettres autant de fois que son nombre d'occurrences. Mais j'utilise bien sûr ici la version 2 pour compter les occurrences !

```
def afficherFrequenceKgramme(tab,tabS,k):
    n=taille(tab)
    mot=allouer(k)
    Kgrs=[];occs=[]
    for i in range(n-k-1):
        for j in range(k): mot[j]=tab[i+j]
        Kgr=afficherMot(mot,0,k)
        if Kgr not in Kgrs:
            Kgrs.append(Kgr)
            occs.append(compterOccurrences2(mot,tab,tabS))
    for j in range(len(Kgrs)):
        print(Kgrs[j],occs[j])
```