1. Algorithmes de recherche

1. Pour une complexité constante dans le meilleur des cas, il faut interrompre la fonction dès qu'on trouve `n`

```
def present(n,L) :
    trouve = False
    i = 0
    while not trouve and i < len(L) :
        if L[i] == n :
            trouve = True
        i += 1
    return trouve
```

2. Même chose avec deux indices :

```
def present2D(n,T) :
    trouve = False
    i = 0
    n,p = len(L),len(L[0])
    while not trouve and i < n :
        j = 0
        while not trouve and j < p :
            if L[i][j] == n :
                trouve = True
            j += 1
        i += 1
    return trouve
```

2. Autour du maximum

1. Il faut initialiser la variable `M` avant de faire des comparaisons

```
def Max(L) :
    M = L[0]
    for k in L[1:] :
        if M < k :
            M = k
    return M
```

2. On distingue les cas selon que l'on retrouve la même valeur ou une nouvelle valeur strictement plus grande

```
def ToutSurMax(L) :
    M,nb,pos = L[0],1,[0]
    for k in range(1,len(L)) :
        if M == L[k] :
            nb += 1
            pos.append(k)
        elif M < L[k] :
            M,nb,po = L[k],1,[k]
    return [M,nb,pos]
```

3. On ne peut pas initialiser `M` avec le « premier » élément de `d` puisque les éléments de `d` ne sont pas ordonnés ; on contourne le problème avec `inf` du module `math` qui est plus grand que tous les réels (donc la « vraie » initialisation de `M` est faite au premier passage dans la boucle)

```
from math import inf

def MaxDico(d) :
    M = -inf
    for c in d :
        if d[c] > M :
            cleMax = c
            M = d[c]
    return cleMax
```

3. Utilisations de dictionnaires

1. On calcule la note au fur et à mesure en examinant les réponses données ; il n'y a rien à faire s'il n'y a pas de réponse à une question puisqu'elle ne figure pas dans `eleve` (et rapporte alors 0)

```
def note(eleve,reponses) :
    s = 0
    for c in eleve :
        if eleve[c] == reponses[c] :
            s += 2
        else :
            s -= 1
    return s
```

2. On fait une copie de `d1` à laquelle on rajoute les éléments de `d2` non présents

```
def union(d1,d2) :
    d = {c:d1[c] for c in d1}
    for c2 in d2 :
        if c2 not in d1 :
            d[c2] = d2[c2]
        elif d1[c2] != d2[c2] :
            d[c2] = (d1[c2],d2[c2])
        # rien à faire sinon : c'est un doublon
    return d
```

3. La comparaison des chaînes de caractères est possible en Python (donc le test `m in d`)

```
def mot(k,ch) :
    d = {}
    for i in range(len(ch)-k) :
        m = ch[i:i+k]
        if m in d :
            d[m] += 1
        else :
            d[m] = 1
    return d
```

Cette fonction à une complexité linéaire (en $O(N)$) puisque le test `m in d` a un coût constant avec un dictionnaire.

4. On fait un peu comme pour `MaxDico` (et on aurait pu initialiser `M` avec `inf` mais ici on est sûr qu'il y a au plus `len(ch)` mots, dans le cas de mots de longueur 1 sur une chaîne dont toutes les lettres sont identiques)

```
def MoinsFrequentMot(k,ch) :
    d = mot(k,ch)
    M = len(ch)+1
    for m in d :
        if d[m] < M :
            cleMin = m
            M = d[m]
    return cleMin
```

Comme `mot` est en $O(N)$, la boucle suivante aussi, la complexité totale est linéaire en N (ie $O(N)$).

5. Les lettres sont les mots de longueur 1 et on peut aussi comparer directement des dictionnaires

```
def anagramme(ch1,ch2) :
    d1 = mot(1,ch1)
    d2 = mot(1,ch2)
    return d1 == d2
```