Rappelons que les commandes `a/b` et `mod(a,b)` données dans l'énoncé s'écrivent respectivement en Python `a//b` et `a%b`.

I. Nombres premiers

Question 1

On peut se demander ici s'il faut déjà utiliser la remarque classique et très rentable selon laquelle il suffit de tester la divisibilité de n par 2 puis par les entiers impairs d tels que $3 \leq d \leq \sqrt{n}$. En effet :

- si n n'admet pas 2 comme diviseur, il n'admettra aucun diviseur pair
- si n admet un diviseur d tel que $\sqrt{n} < d < n$, il en admet aussi un plus petit que n, à savoir n/d.

Ce qui est certain, c'est qu"il faut renvoyer 0 dès qu'on trouve un diviseur. Dans certains langages, cette situation impose une boucle `while`, mais en Python la commande `return` permet d'utiliser une boucle `for`...

```
from math import sqrt

def estPremier(n):
    if n%2==0: return 0
    for d in range(3,1+int(sqrt(n)),2):
        if n%d==0: return 0
    return 1
```

Je teste avant tout la parité de n, puis je teste les éventuels diviseurs impairs d tels que $3 \leq d \leq \sqrt{n}$
Noter que les paramètres de `range` doivent être entiers (d'où le `int`) ; le troisième paramètre (optionnel) de `range` définit le pas d'incrémentation.
Si je sors de la boucle `for`, c'est que n est premier d'après les remarques précédentes et il est donc temps de renvoyer 1 (avec les langages "modernes", on choisit plutôt de faire renvoyer `True` ou `False` à ce type de fonction...).

Question 2

Il faut sans doute ici comprendre dans l'énoncé qu'il est plus "simple" d'utiliser la fonction précédente comme sous-programme.
Après avoir placé 2 dans le tableau global `premier` (en fait une "liste Python"), je teste ainsi la primalité des entiers impairs (économie appréciable, mais correspondant à un facteur constant, c'est moins intéressant que de remplacer n par $\sqrt{n}$) de 3 à n et j'incrémente le compteur nb lorsque je trouve un nombre premier, tout en ajoutant ce dernier à la fin de `premier`.

```
def petitsPremiers(n):
    global premier
    premier=[2]
    for p in range(3,n+1,2):
        if estPremier(p)==1:
            premier.append(p)
    return len(premier)
```

Question 3

Et voici l'astuce de la racine carrée, accompagnée de l'utilisation des nombres premiers déjà calculés.
Le plus "modulaire" est d'écrire une fonction `estPremier2`, le code précédent pourra alors être repris tel quel pour la fonction `petitsPremiers2`, en remplaçant `estPremier` par `estPremier2` ! Pour cela, je mets en œuvre la remarque de l'énoncé, en ne testant comme diviseurs éventuels que les valeurs de la liste globale `premier`, qui se remplit progressivement durant l'exécution de `petitPremier2` (d'où l'intérêt de la liste **globale** !).

```
def estPremier2(p):
    global premier
    r=int(sqrt(p))
    for d in premier:
       if p%d==0: return 0
       if d>r: return 1
    return 1
```

Le résultat est bien correct car, lorsque j'appellerai `estPremier2(p)`, la liste `premier` contiendra les nombres premiers strictement inférieurs à p, or $\sqrt{p} < p$ pour tout $p \geq 2$, donc j'aurai bien examiné tous les nombres premiers utiles. Le dernier `return 1` est là pour le cas où il n'y aurait dans `premier` aucune valeur supérieure à $\sqrt{p}$...

Noter que `estPremier2` n'est utilisable que si `premier` a été suffisamment rempli... Il serait sans doute plus sage de l'encapsuler dans `petitPremier2`, ce qui permettrait en outre de laisser locale la liste `premier` :

```
def petitsPremiers2bis(n):
    def estPremier2bis(p):
        r=int(sqrt(p))
        for d in premier:
            if p%d==0: return 0
            if d>r: return 1
        return 1
    premier=[2]
    for p in range(3,n+1,2):
        if estPremier2bis(p)==1:
            premier.append(p)
    return len(premier)
```

Question 4

J'essaye de coller au plus près de la description de l'énoncé. J'utilise un tableau `coche` de booléens, `coche[i]` contiendra `True` ou `False` selon que la case correspondant à l'entier `i` a été cochée ou pas. Les variables `c1` (*resp.* `c2`) contiendront l'entier correspondant à la case où se trouve le premier (*resp.* second) caillou. La variable `fini` sert à déterminer si l'on doit sortir à l'étape 4 de l'énoncé. À la sortie, il n'y a plus qu'à collecter les nombres premiers...

```
def petitsPremiers3(n):
    global premier
    coche=[False]*(n+1)
    c1=1;fini=False
    while not fini:
        c1+=1
        while c1<=n and coche[c1]: c1+=1
        c2=c1+c1;fini=True
        while c2<=n:
            if not coche[c2]:
                coche[c2]=True;fini=False
            c2+=c1
    premier=[]
    for i in range(2,n+1):
        if not coche[i]:premier.append(i)
    return len(premier)
```

Je n'ai utilisé que des additions et des comparaisons d'entiers.

Question 5

Là encore je colle à l'énoncé. Comme le laisse entendre la question 6, je commence ici par déterminer les nombres premiers inférieurs ou égaux à n. On pourrait s'arrêter à $\sqrt{n}$, mais à condition de tester si $m = 1$ après avoir épuisé les facteurs premiers entre 2 et $\sqrt{n}$, pour le cas où il n'y en aurait pas (cas où n est premier lui-même !).

```
def factoriser(n):
    global premier,facteur
    i=petitsPremiers3(n)
    m=n;i=0;facteur=[]
    while m>1:
        p=premier[i]
        while m%p==0:
            facteur.append(p)
            m=m//p
        i+=1
    return len(facteur)
```

Question 6

Toujours selon les indications de l'énoncé... Ne pas oublier de commencer par épuiser le facteur 2 avant d'examiner les nombres impairs, ni d'ajouter m à la liste des diviseurs lorsqu'on arrive à $q^2 > m$ (dans ce cas je sors de la boucle).

```
def factoriser2(n):
    global facteur
    m=n;facteur=[]
    while m%2==0:
        facteur.append(2)
        m=m//2
    q=3
    while m>1:
        if q**2>m:
            facteur.append(m)
            return len(facteur)
        else:
            while m%q==0:
                facteur.append(q)
                m=m//q
        q+=2
    return len(facteur)
```

II. Reconnaître les puissances

Question 7

Il suffit de reprendre l'algorithme précédent, en l'adaptant pour stocker les multiplicités au lieu des facteurs premiers eux-mêmes. Si l'algorithme se termine avec $q^2 > m$, c'est que m est le plus grand facteur premier et qu'il est de multiplicité 1.

```
def calculerAlpha(n):
    global alpha
    m=n;alpha=[]
    a=0
    while m%2==0:
        a+=1
        m=m//2
    if a!=0: alpha.append(a)
    q=3
    while m>1:
        if q**2>m:
            alpha.append(1)
            return len(alpha)
        else:
            a=0
            while m%q==0:
                a+=1
                m=m//q
            if a!=0: alpha.append(a)
        q+=2
    return len(alpha)
```

Question 8

Il suffit ici de tester si toutes les multiplicités sont multiples de b. Je pourrais tester lesdites multiplicités au fur et à mesure que je les calcule selon le principe de la question 7, mais l'esprit du sujet semble plutôt d'appeler le programme précédent dans un souci de modularité. C'est moins efficace mais le code est plus limpide.

```
def estPuissance(n,b):
    global alpha
    nb=calculerAlpha(n)
    for a in alpha:
        if a%b!=0: return 0
    return 1
```

Question 9

Il suffit par exemple de tester si le nombre est égal au carré de la partie entière de sa racine carrée.

Remarque (un peu) hors sujet : attention toutefois, au nombre de chiffres significatifs utilisés pour calculer une valeur approchée de la racine carrée d'un "grand" entier !
Exemple avec Python :

```
In [2]:  N=12345678901234567**2;floor(sqrt(N))
Out[2]:  12345678901234568
```

Il existe toutes sortes d'autres méthodes plus ou moins sophistiquées, mais le sujet ne donne aucun cadre... On peut toutefois se sentir autorisé à utiliser `sqrt`, fonction fournie dans le préambule...

III. Nombres de Carmichael

Question 10

Comme ici le traitement de chaque facteur est assez différent de ceux précédents, je choisis de reprendre tous les calculs, en vérifiant au fur et à mesure les conditions, pour chaque facteur premier détecté selon le principe de la question 6. Je renvoie tout de suite 0 si m est pair. Ensuite, j'examine les valeurs successives de q :

- dans le cas où $q^2 > m$, je dois tester si $m = c$ (dans ce cas c est premier et je renvoie 0), sinon m est le plus grand facteur premier et j'ai juste à tester la condition "$m-1$ divise $c-1$" (si j'arrive là c'est que tous les facteurs premiers précédents vérifient les conditions imposées)
- dans le cas $q^2 \leq m$, rien à faire si q ne divise pas m ; s'il le divise, c'est un facteur premier de m, donc de c puisque m n'est autre que c divisé par tous ses facteurs premiers inférieurs à q. Il reste donc à voir si q est bien de multiplicité 1 et vérifie "$q-1$ divise $c-1$", sinon je renvoie 0.

Si je sors de la boucle, c'est que c vérifie bien toutes les conditions.

```
def estCarmichael(c):
    m=c
    if m%2==0: return 0
    q=3
    while m>1:
        if q**2>m:
            if m==c:
                return 0
            elif (c-1)%(m-1)!=0:
                return 0
            else:
                return 1
        elif m%q==0:
            m=m//q
            if m%q==0 or (c-1)%(q-1)!=0: return 0
        q+=2
    return 1
```

On retrouve le fait (bien connu en arithmétique !) que le premier nombre de Carmichael est

$$561 = 3 \cdot 11 \cdot 17.$$

Question 11

Pour calculer les nombres premiers inférieurs à n (strictement, puisque si n est premier il ne conviendra pas), j'appelle le programme de la question 4. Ensuite une triple boucle permet d'examiner les produits de trois facteurs premiers distincts. Ne pas utiliser `premier[0]` qui vaut 2 ! On pourrait ajouter des tests pour éviter certains calculs lorsque `p1` ou `p2` sont devenus trop grands, mais l'énoncé dit "tous les produits possibles de trois de ces nombres". Noter que cette triple boucle deviendra vite très coûteuse quand n augmentera...

```
def calculerCarmichael3(n):
    global premier,carmichael
    nb=petitsPremiers3(n-1)
    carmichael=[]
    for i1 in range(1,nb-2):
        p1=premier[i1]
        for i2 in range(i1+1,nb-1):
            p2=premier[i2]
            c2=p1*p2
            for i3 in range(i2+1,nb):
                p3=premier[i3]
                c=c2*p3
                if c<=n and (c-1)%(p1-1)==0 and (c-1)%(p2-1)==0 and (c-1)%(p3-1)==0:
                    carmichael.append(c)
    return len(carmichael)
```

Question 11

J'applique bêtement l'algorithme décrit dans l'énoncé, algorithme assez mystérieux au demeurant...

```
def calculerCarmichael3(n):
    global premier,carmichael
    nb=petitsPremiers3(n-1)
    carmichael=[]
    for i1 in range(1,nb-2):
        p1=premier[i1]
        for i2 in range(i1+1,nb-1):
            p2=premier[i2]
            c2=p1*p2
            for i3 in range(i2+1,nb):
                p3=premier[i3]
                c=c2*p3
                if c<=n and (c-1)%(p1-1)==0 and (c-1)%(p2-1)==0 and (c-1)%(p3-1)==0:
                    carmichael.append(c)
    return len(carmichael)
```

Algorithme mystérieux mais diaboliquement efficace : sur mon ordinateur, pour $n = 5\,000$, `carMichael3` prend environ 8,5s et `carMichael` moins d'un millième de seconde !