Corrigé

Question 1

```
def random_list (n, k) :
    t = [] # crée un tableau vide
    for i in range (n) : # rajoute n fois un élémént àt
        t.append( randrange (k) ) # cet élément est pris au hasard
                                  # entre 0 et k-1
    return (t)
```

Question 2

```
def comptage (k, t) :
    """ si t est un tableau àvaleurs dans range(k), comptage renvoie
    un tableau u de longueur k tel que pour tout i, u[i] vaut le nombre
    d'occurences de i dans t """
    u = [0]*k
    for i in t : # on parcourt t
        u[i] += 1 # on ajoute 1 àu[i], car on a trouvé une occurence
                  # supplémentaire de i
    return (u)

def counting_sort (k, t) :
    u = comptage (k,t)
    b = [] # b contiendra le tableau trié
    for i in range (k) : # k est la longueur de u
        b += [i] * u[i] # l'élément i est ajouté u[i] fois
    return b

# autre méthode en utilisant comptage_cumulé, je vous laisse finir

def comptage_cumule (k, t) :
    """ si t est un tableau àvaleurs dans range(k), comptage_cumule renvoie
    un tableau u de longueur k tel que pour tout i, u[i] vaut le nombre
    d'occurences d'éléments inférieurs ou égaux ài dans t """
    u = comptage (k, t)
    for i in range(1,k) : # invariant : pour j de 0 ài, u[j] vaut le nombre
                # d'occurences d'éléments inférieurs ou égaux àj dans t
        u[i] = u[i] + u[i-1]
    return (u)
```

Question 3

```
def bucket_sort (f,k,t) :
    baquets = [None]*k # on crée un tableau àk éléments bidons
    for i in range(k) :
        baquets[i] = [] # on crée k baquets vides
    for x in t :
        baquets[f(x)] += [x] # on ajoute l'élément x dans le baquet f(x)
                # on remarque que les éléments d'un même baquet sont dans
                # le même ordre que dans t
    b = [] # b sera le tableau trié
    for baquet in baquets :
        b += baquet # on rajoute les éléments des baquets les uns
                    # après les autres
    return b # b contient les éléments de t, triés suivantleur image
        # et deux éléments de même image sont dans le même ordre que
        # dans t

# test
```

```
k = 10
n = 16
# créons une fonction aléatoire f de range(k) dans range(k)
T = random_list (k,k) # ce tableau T contient les images de notre
    # future f dans l'ordre : T = [f(0),f(1),...,f(k-1)]
def f (x) :
    return T[x] # il suffit d'envoyer x sur T[x]

t = random_list (n,k)
print(T)
print(t)
print(bucket_sort (f,k,t))
```

Question 4 Nous avons rp(0) = 0, rp(1) = 1, rp(2) = 2, rp(4) = 4, rp(5) = 0, rp(6) = 1, rp(7) = 2, rp(9) = 4, rp(10) = 0, rp(12) = 2 et rp(17) = 2. Donc `bucket_sort(rp, 5, t)` doit renvoyer `[0,5,10,1,6,6,12,17,2,7,4,9]`.
Puis, lp(0) = 0, lp(1) = 0, lp(2) = 0, lp(4) = 0, lp(5) = 1, lp(6) = 1, lp(7) = 1, lp(9) = 1, lp(10) = 2, lp(12) = 2 et lp(17) = 3.
Donc `radix_sort(5, t)` renvoie `[0,1,2,4,5,6,6,7,9,10,12,17]`.
On peut conjecturer que `radix_sort` est un algorithme qui trie une liste d'entiers par ordre croissant.

Question 5 Le programme `random_list(n, k)` ne contient qu'une boucle, de n tours. Or à chaque tour une seule instruction est éxécutée : l'ajout d'un élément à la fin d'un tableau.
Sous l'hypothèse que l'ajout d'un élément à la fin d'un tableau se fait à coût constant, alors la complexité de `random_list(n, k)` est bien un $\Theta(n)$.

Question 6 `counting_sort` utilise ici le programme `comptage`. Il faut donc d'abord évaluer la complexité de ce dernier programme.
La première instruction de `comptage` est la création d'un tableau de longueur k : cette opération a une complexité en $\Theta(k)$. Ensuite, la boucle du programme a n tours, et dans chaque tour de boucle on suppose que l'opération effectué a un coût constant. Ainsi, le temps d'éxécution de `comptage` est encadré par deux fonctions de la forme $an+bk+c$, où a, b, c sont des constante. C'est donc la plus grande des valeurs de n et k qui est prépondérante : la complexité de `comptage` est donc un $\Theta(\max(n,k))$.
Les instructions de `counting_sort` sont les suivantes :

- un appel à `comptage (k,t)` en $\Theta(\max(n,k))$;
- une boucle de k tours dans laquelle la totalité des opérations (en prenant en compte tous les tours de boucle en une fois) consiste à recopier les éléments des éléments de `u` dans un tableau : or il y a n éléments à recopier, donc une complexité $\Theta(n)$.

Globalement, la complexité est donc bien $\Theta(\max(n,k))$.

Question 7 Les instructions de `bucket_sort` sont les suivantes :

- création d'un tableau de k éléments, soit une complexité $\Theta(k)$;
- une boucle de n tours contenant chacun une opération unitaire, soit une complexité $\Theta(n)$;
- une troisième boucle de k tours dans laquelle à chaque tour on accède à un élément d'un tableau : ces accès ont une complexité $\Theta(k)$.
  De plus, la totalité des autres opérations (en prenant en compte tous les tours de boucle en une fois) consiste à recopier les éléments des éléments de `baquets` dans un tableau : or il y a n éléments à recopier, donc une complexité $\Theta(n)$.

Globalement, la complexité est donc bien $\Theta(\max(n,k))$.

Question 8 Notons `n` la longueur de `t`.
Remarquons d'abord que si `p` est un entier compris dans `range(k**2)`, alors `rp(p)` et`lp(p)` sont dans `range(k)`.
`radix_sort` effectue un premier appel à `bucket_sort`, sur une liste de longueur `n`, et en utilisant une fonction à valeurs dans `range(k)`. Ainsi, cet appel à une complexité en $\Theta(\max(n,k))$. Il renvoie `u`, qui est encore de longueur `n`.
Ensuite, un second appel à `bucket_sort` est effectué, toujours sur une liste de longueur `n`, et en utilisant une fonction à valeurs dans `range(k)`. La complexité est encore $\Theta(\max(n,k))$.
Globalement, la complexité de `radix_sort` est $\Theta(\max(n,k))$.
Si n est "grand" et k constant (i.e. si la taille du tableau est plus grande que les entiers qu'il contient), ce tri est de complexité linéaire, dans tous les cas. Ceci est à comparer à la complexité des tris pour lesquels aucune hypothèse n'est faite sur les éléments du tableau à trier, et qui est en moyenne $\Theta(n\log n)$ pour les tris les plus performants.

Question 9 Une méthode naturelle pour trier des entiers et d'utiliser l'ordre lexicographique, c'est-à-dire que l'on trie ces entiers en regardant d'abord leurs chiffres de poids le plus fort, et en finissant par les unités.
On pourrait ainsi mettre dans un même baquet tous les entiers commençant par 9 (ou $k-1$ si l'on compte en base

k), puis dans un autre tous ceux commençant par 8 ... etc ... jusqu'à ceux commençant par 0 (on suppose que tous les entiers manipulés sont écrits avec le même nombre de chiffres).
Puis on reprend chaque baquet, et dans chaque baquet on refait la même opréation, cette fois en regardant le deuxième chiffre. Puis on réitère en regardant le troisième chiffre et ainsi de suite. On se rend compte que le nombre de baquets utilisés pour ce tri est considérable (combien précisément ?).
Le tri `radix_sort` trie lui les entiers en commençant par comparer les chiffres des unités : on met dans un même baquet tous les éléments finissant par 0, puis dans un autre finissant par 1 et ainsi de suite. Si l'on compte en base k, k baquets sont ainsi utilisés. Puis on remet tous les éléments ensemble, en respectant l'ordre des baquets. Et on recommence en regardant le deuxième chiffre, etc.

Lorsque l'on compte en base k et que les éléments du tableau sont dans `range(k**2)` comme dans ce TP, alors chaque élément s'écrit sur deux bits. Après la première boucle for de `radix_sort`, les éléments sont triés suivant leur chiffre des unités. Après la seconde boucle for, ces entiers sont triés suivant le bit de poids fort. Mais puisque le tri `bucket_sort` est stable, les entiers ayant le même bit de poids fort sont encore triés suivant leur chiffre des unités : le tableau est donc bien trié après l'éxécution de ces deux boucles.