**Question 1** :

```
def rendu_glouton(pieces, somme):
    i =  len(pieces) - 1  # Attention, les pièces sont classées dans
l'ordre croissant.
    nb_pieces = 0
    while somme > 0:
        if pieces[i] <= somme:
            nb_pieces = nb_pieces + 1
            somme = somme - pieces[i]
        else :
            i = i - 1
    return nb_pieces

assert rendu_glouton([1, 2, 5], 12) == 3
```

**Question 2** :

```
def rendu_recursif(pieces, somme):
    nb_pieces = somme  # nombre de pièces dans le pire des cas
    if somme == 0:
        return 0  # cas de base
    else:
        for p in pieces:
            if p <= somme  :  # peut-on rendre la pièce p ?
                nb_pieces = min(nb_pieces, 1 + rendu_recursif(pieces,
somme - p))
        return nb_pieces
```

**Question 3** :

```
assert rendu_recursif([1, 2, 5], 12) == 3
assert rendu_recursif([1, 4, 6], 8) == 2
```

**Question 4** :

L'arbre d'appel de rendu_recursif([1, 2], 100) est énorme. Chaque nœud représente une somme et possède deux branches, une pour chaque pièce. On aimera connaître la taille de cette arbre précisément. Mais le programme ne se termine pas, alors il va falloir réfléchir.

L'arbre est compliqué car la taille d'un sous arbre n'est pas le même si on utilise une pièce de 1 ou de 2. On va se simplifier le calcul en se disant qu'a chaque nœud, on soustrait deux à la somme. C'est le meilleur cas pour nous. Avec ce cas, on va pouvoir minorer le nombre d'appels récursifs, c'est à dire trouver une valeur pour laquelle le nombre d'appels est forcement plus grand.

En enlevant 2 a chaque fois, l'arbre à 50 étages, donc 2**50 appels.
On peut aussi majorer le nombre d'appels, trouver une valeur pour laquelle le nombre d'appel est forcement inférieur. C'est le cas pour lequel on enlève 1 à la somme. L'arbre a ainsi 100 étages, donc 2**100 appels. Le nombre réel d'appels se situe donc entre 2**50 et 2**100.

En modifiant le programme pour compter le nombre d'appel, on voit les résultats suivant :
Pour une somme égale à 25, le nombre d'appel est déjà 317810.
Pour une somme égale à 30, le nombre d'appel est déjà 3524577.
Pour une somme égale à 35, le nombre d'appel est déjà 39088168.
On multiplie par plus de 10 le nombre d'appels en ajoutant 5 euros.

**Question 5** :

```
def rendu_recursif_memo(pieces, somme, memo):
    if memo[somme] is not None:  # si on a déjà calculé le nombre optimal
de pièces pour rendre somme
        return memo[somme]      # on le renvoie directement
    elif somme == 0:
        memo[0] = 0
        return 0
    else:
        nb_pieces = somme     # nb_pieces = 1 + 1 + ... + 1 dans le pire
des cas
        for p in pieces:
            if p <= somme:   # inutile de tester les pièces p de valeur >
somme
                nb_pieces = min(nb_pieces, 1 + rendu_recursif_memo(pieces,
somme - p, memo))
        memo[somme] = nb_pieces
        return nb_pieces

assert rendu_recursif_memo([1, 2], 100, [None]*(100+1)) == 50
```

**Question 6** :

```
def rendu_recursif_memoise(pieces, somme):
    return rendu_recursif_memo(pieces, somme, [None]*(somme+1))
```

**Question 7** :

```
assert rendu_recursif_memoise([1, 2], 100) == 50
```

**Question 8** :

```
# La seul différence est dans la première condition
def rendu_recursif_dico(pieces, somme, memo):
    if somme in memo:  # si on a déjà calculé le nombre optimal de pièces
pour rendre somme
        return memo[somme]      # on le renvoie directement
    elif somme == 0:
        memo[0] = 0
        return 0
    else:
        nb_pieces = somme     # nb_pieces = 1 + 1 + ... + 1 dans le pire
des cas
        for p in pieces:
            if p <= somme:   # inutile de tester les pièces p de valeur >
somme
                nb_pieces = min(nb_pieces, 1 + rendu_recursif_dico(pieces,
somme - p, memo))
        memo[somme] = nb_pieces
        return nb_pieces

assert rendu_recursif_dico([1, 2], 100, {}) == 50

def rendu_recursif_memoise_dico(pieces, somme):
    return rendu_recursif_dico(pieces, somme, {})
```

**Question 9** :

Le tableau nb est initialisé à [0, 0, 0, 0, 0, 0, 0, 0, 0] et on va le remplir indice par indice.
- Pour s = 1 :
  - on initialise nb[1] à 1 puisque dans le pire des cas, on peut rendre la somme 1 avec une pièce ( 1=1 )
  - on regarde ensuite si on peut faire mieux en analysant tous les cas possibles :
    - si on rend la pièce 1, il faut encore rendre 0 et on sait que nb[0] = 0 donc cela ferait 1+0=1 pièce ;
    - on ne peut pas rendre la pièce 2;
    - on ne peut pas rendre la pièce 5;
  - à la fin de l'itération, on a donc nb[1] = 1 et donc nb = [0, 1, 0, 0, 0, 0, 0, 0, 0]
- Pour s = 2 :
  - on initialise nb[2] à 2 puisque dans le pire des cas, on peut rendre la somme 2 avec 2 pièces de 1 ( 2=1+1)
  - on regarde ensuite si on peut faire mieux en analysant tous les cas possibles :
    - si on rend la pièce 1, il faut encore rendre 1 et on sait que nb[1] = 1 donc cela ferait 1+1=2 pièces;
    - si on rend la pièce 2, il faut encore rendre 0 et on sait que nb[0] = 0 donc cela ferait 1+0=1 pièce;
    - on ne peut pas rendre la pièce 5;
  - à la fin de l'itération, on trouve que nb[2] = 1 et donc nb = [0, 1, 1, 0, 0, 0, 0, 0, 0]
- Pour s = 3 :
  - on initialise nb[3] à 3
    - si on rend la pièce 1, il faut encore rendre 2 et on sait que nb[2] = 1 donc cela ferait 1+1=2 pièces;
    - si on rend la pièce 2, il faut encore rendre 1 et on sait que nb[1] = 1 donc cela ferait 1+1=2 pièce;
    - on ne peut pas rendre la pièce 5;
  - à la fin de l'itération, on trouve que nb[3] = 2 et donc nb = [0, 1, 1, 2, 0, 0, 0, 0, 0]
- Pour s = 4 :
  - on initialise nb[4] à 4
    - si on rend la pièce 1, il faut encore rendre 3 et on sait que nb[3] = 2 donc cela ferait 1+2=3 pièces;
    - si on rend la pièce 2, il faut encore rendre 2 et on sait que nb[2] = 1 donc cela ferait 1+1=2 pièce;
    - on ne peut pas rendre la pièce 5;
  - à la fin de l'itération, on trouve que nb[4] = 2 et donc nb = [0, 1, 1, 2, 2, 0, 0, 0, 0]
- Pour s = 5 :
  - on initialise nb[5] à 5
    - si on rend la pièce 1, il faut encore rendre 4 et on sait que nb[4] = 2 donc cela ferait 1+2=3 pièces;
    - si on rend la pièce 2, il faut encore rendre 3 et on sait que nb[3] = 2 donc cela ferait 1+2=3 pièce;
    - si on rend la pièce 5, il faut encore rendre 0 et on sait que nb[0] = 0 donc cela ferait 1+0=1 pièce;
  - à la fin de l'itération, on trouve que nb[5] = 1 et donc nb = [0, 1, 1, 2, 2, 1, 0, 0, 0]
- Pour s = 6 :
  - on initialise nb[6] à 6
    - si on rend la pièce 1, il faut encore rendre 5 et on sait que nb[5] = 1 donc cela ferait 1+1=2 pièces;
    - si on rend la pièce 2, il faut encore rendre 4 et on sait que nb[4] = 2 donc cela ferait 1+2=3 pièce;
    - si on rend la pièce 5, il faut encore rendre 1 et on sait que nb[1] = 1 donc cela ferait 1+1=2 pièce;
  - à la fin de l'itération, on trouve que nb[6] = 2 et donc nb = [0, 1, 1, 2, 2, 1, 2, 0, 0]
- Pour s = 7 :
  - on initialise nb[7] à 7
    - si on rend la pièce 1, il faut encore rendre 6 et on sait que nb[6] = 2 donc cela ferait 1+2=3 pièces;
    - si on rend la pièce 2, il faut encore rendre 5 et on sait que nb[5] = 1 donc cela ferait 1+1=2 pièce;
    - si on rend la pièce 5, il faut encore rendre 2 et on sait que nb[2] = 1 donc cela ferait 1+1=2 pièce;
  - à la fin de l'itération, on trouve que nb[7] = 2 et donc nb = [0, 1, 1, 2, 2, 1, 2, 2, 0]
- Pour s = 8 :
  - on initialise nb[8] à 8
    - si on rend la pièce 1, il faut encore rendre 7 et on sait que nb[7] = 2 donc cela ferait 1+2=3 pièces;
    - si on rend la pièce 2, il faut encore rendre 6 et on sait que nb[6] = 2 donc cela ferait 1+2=3 pièce;
    - si on rend la pièce 5, il faut encore rendre 3 et on sait que nb[3] = 2 donc cela ferait 1+2=3 pièce;
  - à la fin de l'itération, on trouve que nb[8] = 3 et donc nb = [0, 1, 1, 2, 2, 1, 2, 2, 3]

**Question 10** :

```
def rendu_iteratif_ascendant(pieces, somme):
    """Version itérative ascendante.
    Utilisation d'un tableau pour stocker les valeurs calculées."""
    # ÉTAPE 1 : création et initialisation du tableau
    nb = [0] * (somme+1) # nb[0] est ainsi bien initialisé
    # ÉTAPE 2 : remplissage du reste du tableau par indice croissant
    for s in range(1, somme+1): # attention, il faut aller jusqu'à la
valeur somme.
        nb[s] = s # nombre de pièces dans le pire des cas.
        for p in pieces:
            if p <= s:
                nb[s] = min(nb[s], 1 + nb[s - p])
    # ÉTAPE 3 :
    return nb[somme] # ou nb[-1]

assert rendu_iteratif_ascendant([1, 2, 5], 8) == 3
```

**Question 11** :

```
assert rendu_iteratif_ascendant([1, 2], 100) == 50
```

**Question 12** :

```
# La seul modification est à l'étape 1, on initialise le dictionnaire avec
le couple 0:0
# ÉTAPE 1 : création et initialisation du tableau
    nb = {0:0}
```

**Question 13** :

```
def rendu_monnaie_ascendante_solution(pieces, somme):
    # ÉTAPE 1 : création et initialisation des tableaux
    nb = [0] * (somme + 1)
    sol = [[]] * (somme + 1)
    # ÉTAPE 2 : remplissage du reste des tableaux par indice croissant
    for s in range(1, somme + 1): # L'indice 0 est déjà correct
        nb[s] = s # nb[s] = 1 + 1 + ... + 1 dans le pire des cas
        sol[s] = [1] * s # on rend s fois la pièce 1 dans le pire des cas
        for p in pieces:
            if p <= s:
                if 1 + nb[s-p] < nb[s]:
                    nb[s] = 1 + nb[s-p]
                    sol[s] = sol[s-p].copy() # copie du tableau, pour ne
pas ajouter la pièce dans celui de départ
                    sol[s].append(p)
    # ÉTAPE 3 : le résultat est dans la dernière case
    return sol[somme]
assert rendu_monnaie_ascendante_solution([1, 2, 5], 38) == [5, 5, 5, 5, 5,
5, 5, 2, 1]
```