```
def rendu_monnaie_glouton(s, pieces):
        """Renvoie la solution gloutonne du rendu de monnaie d'une somme s entière et positive.
        Le tableau pieces contient les valeurs des pièces à disposition dans l'ordre croissant."""
        solution = []
        i = len(pieces) - 1 # position de la première pièce à tester (la plus grande est à la fin)
        while s > 0 and i >= 0: # tant qu'il reste de l'argent à rendre et que toutes les pièces n'ont pas été testées
            valeur = pieces[i] # on prend la pièce d'indice i
            if valeur <= s: # s'il est possible de rendre la pièce
                solution.append(valeur) # on l'ajoute à solution
                s = s - valeur # et on déduit sa valeur de la somme à rendre
            else:
                i = i - 1 # sinon on passe à la pièce immédiatement inférieure
        return solution
```

Question 1 :

1) `rendu_monnaie_glouton(147, [1, 2, 5, 10, 20, 50, 100, 200, 500] )`
   `# [100, 20, 20, 5, 2]`
2) `rendu_monnaie_glouton(8, [1, 4, 6])`
   `# [6, 1, 1]`
3) `rendu_monnaie_glouton(8, [2, 5])`
   `# [5, 2]`

Question 2 :

Elles ne sont pas optimales.
On ne renvoie pas forcement le nombre minimal de pièces (2).
On ne renvoie pas toujours une solution satisfaisante alors qu’il en existe une (3).

Question 3 :

```
def rendu_monnaie_glouton(s, pieces):
        """Renvoie la solution gloutonne du rendu de monnaie d'une somme s entière et positive.
        Le tableau pieces contient les valeurs des pièces à disposition dans l'ordre croissant."""
        solution = []
        i = len(pieces) - 1 # position de la première pièce à tester (la plus grande est à la fin)
        while s > 0 and i >= 0: # tant qu'il reste de l'argent à rendre et que toutes les pièces n'ont pas été testées
            valeur = pieces[i] # on prend la pièce d'indice i
            if valeur <= s: # s'il est possible de rendre la pièce
                solution.append(valeur) # on l'ajoute à solution
                s = s - valeur # et on déduit sa valeur de la somme à rendre
            else:
                i = i - 1 # sinon on passe à la pièce immédiatement inférieure
        return len(solution)
```

Question 4 :

```
                                        nb_pieces(100)
                                         |          |
                              nb_pieces(99)          nb_pieces(98)
                  |             |                      |           |
          nb_pieces(98)     nb_pieces(97)          nb_pieces(97)     nb_pieces(96)
         |          |    |          |            |          |     |          |
nb_pieces(97) nb_pieces(96) nb_pieces(96) nb_pieces(95)  nb_pieces(96)  nb_pieces(95)  nb_pieces(95)  nb_pieces(94)
```

Non, nous n'avons pas envie de continuer sur les 96 niveaux suivants

Question 5 :

$$nb\_pieces(s) = \begin{cases} 0 & \text{si } s == 0 \\ 1 + nb\_pieces(s-p) & \text{si } s \geq p \end{cases}$$

Question 6 :

```
def rendu_monnaie_naif(s, pieces):
    """Renvoie le nombre minimal de pièces pour rendre la somme s avec les pièces stockées
dans le tableau pieces."""
    if s == 0:
        return 0
    nb_pieces = s     # nb_pieces = 1 + 1 + ... + 1 dans le pire des cas
    for p in pieces:
        if p <= s:   # inutile de tester une pièce dont la valeur dépasse la somme s à
rendre
            nb_pieces_bis = 1 + rendu_monnaie_naif(s-p, pieces)
            nb_pieces = min(nb_pieces_bis, nb_pieces)
    return nb_pieces
```

Question 7 :

```
rendu_monnaie_naif(8, [1, 4, 6])
# 2

rendu_monnaie_naif(8, [2,5])
# 3 On tombe dans le pire des cas avec rendu_monnaie_naif(1, [2,5])
```

Question 8 :

Ça plante, pas de surprise.

Question 9 :

```
import time

start = time.time()
rendu_monnaie_naif(35, [1, 2])
end = time.time()
print(end - start)
```

Me donne environ 39 secondes.

Question 10 :

```
def rendu_monnaie_naif_bis(s, pieces, dico):
    """Renvoie le nombre minimal de pièces pour rendre la somme s avec les
pièces stockées dans le tableau pieces."""
    if s == 0:
        return 0
    nb_pieces = s
    for p in pieces:
        if p <= s:
            dico[s-p] = 1 if s-p not in dico else dico[s-p] + 1
            nb_pieces_bis = 1 + rendu_monnaie_naif_bis(s-p, pieces, dico)
            nb_pieces = min(nb_pieces_bis, nb_pieces)
    return nb_pieces
```

Question 11 :

```
>>> nombre_appels(8, [1, 4, 6])
{8: 1, 7: 1, 6: 1, 5: 1, 4: 2, 3: 3, 2: 5, 1: 7, 0: 10}

8
2 4 7
1 0 3 1 3 6
0 2 0 2 0 2 5
1 1 1 1 4
0 0 0 0 0 3
2
1
0

>>> nombre_appels(35, [1, 2])
{35: 1, 34: 1, 33: 2, 32: 3, 31: 5, 30: 8, 29: 13, 28: 21, 27: 34, 26: 55,
25: 89, 24: 144, 23: 233, 22: 377, 21: 610, 20: 987, 19: 1597, 18: 2584, 17:
4181, 16: 6765, 15: 10946, 14: 17711, 13: 28657, 12: 46368, 11: 75025, 10:
121393, 9: 196418, 8: 317811, 7: 514229, 6: 832040, 5: 1346269, 4: 2178309,
3: 3524578, 2: 5702887, 1: 9227465, 0: 14930352}
```

Les valeurs correspondent à la suite de Fibonacci de (n-1) + 1

Question 12 :

```
def rendu_monnaie_memo(s, pieces, memo):
    """Renvoie le nombre minimal de pièces pour rendre la somme s avec les
pièces stockées dans le tableau pieces."""
    if s == 0:
        return 0
    nb_pieces = s     # nb_pieces = 1 + 1 + ... + 1 dans le pire des cas
    for p in pieces:
        if p <= s:   #inutile de tester une pièce dont la valeur dépasse s
            if s-p not in memo:
                memo[s-p] = 1 + rendu_monnaie_memo(s-p, pieces, memo)
            nb_pieces = min(memo[s-p], nb_pieces)
    return nb_pieces
```

```
>>> rendu_monnaie_memo(35, [1, 2], {})
18
```

Question 13 :

```
def rendu_monnaie_memoisation(s, pieces):
    return rendu_monnaie_memo(s, pieces, {})
```

Question 14 :

```
import time

start = time.time()
rendu_monnaie_memoisation(35, [1, 2])
end = time.time()
print(end – start)

import time

start = time.time()
rendu_monnaie_memoisation(100, [1, 2])
end = time.time()
print(end - start)
```

On obtient des résultats instantanés (0.0 secondes) pour les 2 cas.

Question 15 :

```
def rendu_monnaie_memo_liste(s, pieces, memo):
    """Renvoie le nombre minimal de pièces pour rendre la somme s avec les
pièces stockées dans le tableau pieces."""
    if s == 0:
        return 0
    nb_pieces = s     # nb_pieces = 1 + 1 + ... + 1 dans le pire des cas
    for p in pieces:
        if p <= s:   # inutile de tester une pièce dont la valeur dépasse la
somme s à rendre
            if memo[s-p] is None:
                memo[s-p] = 1 + rendu_monnaie_memo_liste(s-p, pieces, memo)
            nb_pieces = min(memo[s-p], nb_pieces)
    return nb_pieces

def rendu_monnaie_memoisation_liste(s, pieces):
    return rendu_monnaie_memo_liste(s, pieces, [None]*s)
```

Question 16 :

1) On a 101 une valeur de 0 à 100 inclus, donc 101 appels.
Exemple avec une variable compteur.

```python
cpt = 0

def rendu_monnaie_memo_liste(s, pieces, memo):
    """Renvoie le nombre minimal de pièces pour rendre la somme s avec les pièces stockées dans le tableau pieces."""
    global cpt
    cpt += 1
    if s == 0:
        return 0
    nb_pieces = s     # nb_pieces = 1 + 1 + ... + 1 dans le pire des cas
    for p in pieces:
        if p <= s:   # inutile de tester une pièce dont la valeur dépasse la somme s à rendre
            if memo[s-p] is None:
                memo[s-p] = 1 + rendu_monnaie_memo_liste(s-p, pieces, memo)
            nb_pieces = min(memo[s-p], nb_pieces)
    return nb_pieces

def rendu_monnaie_memoisation_liste(s, pieces):
    return rendu_monnaie_memo_liste(s, pieces, [None]*s)

rendu_monnaie_memoisation_liste(100, [1, 2])
print(cpt) # affiche 101
```

2) il y a 2 passages dans la boucle for, un par pièce

Question 17 :

Pour chaque valeur entre 0 et s+1, on effectue une iteration par pièce. On est en O(sp).
On souhaite faire tourner cet algorithme sur l'exemple s=8 et pieces=[1,2,5].

Le tableau nb est initialisé à [0, 0, 0, 0, 0, 0, 0, 0, 0] et on va le remplir indice par indice.

- Pour n=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 n=2:
  - on initialise nb[2] à 2 puisque dans le pire des cas, on peut rendre la somme 1 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 n=3:
  - on initialise nb[3] à 3
  - on regarde ensuite si on peut faire mieux en analysant tous les cas possibles :
    - 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 n=4:
    - on initialise nb[4] à 4
    - on regarde ensuite si on peut faire mieux en analysant tous les cas possibles :
        - 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 n=5:
    - on initialise nb[5] à 5
    - on regarde ensuite si on peut faire mieux en analysant tous les cas possibles :
        - 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 n=6:
    - on initialise nb[6] à 6
    - on regarde ensuite si on peut faire mieux en analysant tous les cas possibles :
        - 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 n=7:
    - on initialise nb[7] à 7
    - on regarde ensuite si on peut faire mieux en analysant tous les cas possibles :
        - 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 n=8:
    - on initialise nb[8] à 8
    - on regarde ensuite si on peut faire mieux en analysant tous les cas possibles :
        - 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]

On a le résultat dans nb[8].

Question 19 :

```
def rendu_monnaie_ascendante(s, pieces):
    # On initialise directement nb en le créant
    nb = [i for i in range(s+1)]
    for i in range(s+1):
        for p in pieces:
            if i-p >=0:
                nb[i] = min(nb[i], 1 + nb[i-p])
    return nb[s]
```

Question 20 :

```
print(rendu_monnaie_ascendante(8, [1, 2, 5])) # 3
```

Question 21 :

```
def rendu_monnaie_ascendante_solution(s, pieces):
    # On initialise directement nb en le créant
    nb = [i for i in range(s+1)]
    sol = [[] for _ in range(s+1)]
    for i in range(s+1):
        for p in pieces:
            if i-p >=0:
                if nb[i] > 1 + nb[i-p]:
                    nb[i] = 1 + nb[i-p]
                    sol[i] = sol[i-p].copy() + [p]
    return sol[s]

print(rendu_monnaie_ascendante_solution(13, [1, 2, 5]))
```