Correction TD 5

1. On commence par chercher la plus grosse pièce utilisable p_i (donc $\leqslant s$) puis on recommence avec la somme $s - p_i$:

```
def glouton(s,P) :
    if s == 0 :
        return []
    else :
        i = 0
        while s < P[i] :
            i += 1
        return [P[i]]+glouton(s-P[i],P)
```

2. `glouton(49,Pieces1)` renvoie `[20, 20, 5, 2, 2]`, qui est optimal.
`glouton(49,Pieces2)` renvoie `[30, 12, 6, 1]`, qui n'est pas optimal, car $49 = 24 + 24 + 1$.
Le système (européen) `Pieces1` est appelé « canonique », ce qui signifie que la démarche gloutonne fournit toujours le rendu optimal.

3. a) La liste `P_poss` contient la liste des pièces utilisable pour rendre la somme s, ie les pièces p_i pour lesquelles $p_i \leqslant s$.

b) C'est (très) long !

4. Avec deux fonctions (on peut aussi le faire avec une sous-fonction) :

```
def n(s,P,d) :
    if s not in d :
        if s == 0 :
            r = 0
        else :
            P_poss = []
            for k in P :
                if k <= s :
                    P_poss.append(k)
            r = 1+min([n(s-k,P_poss,d) for k in P_poss])
        d[s] = r
    return d[s]

def dynamique(s,P) :
    d = {}
    return n(s,P,d)
```

`dynamique(49,Pieces2)` renvoie `3`, et c'est (très) rapide.

5. Après avoir introduit `T`, avec la bonne taille, on la remplit :

```
def BasEnHaut(s,P) :
    T = [0]*(s+1)
    for n in range(1,s+1) :
        P_poss = []
        for k in P :
            if k <= n :
                P_poss.append(k)
        T[n] = 1+min(T[n-k] for k in P_poss)
    return T[s]
```

La complexité est $O(nk)$ car la construction de la liste `P_poss` est en $O(k)$, de même que la détermination du minimum (calculé sur une liste de longueur $\leqslant k$).

6. On a $4 = 1 + $ `T[20-12]` donc on commence par la pièce 12 et il reste $s = 8$ à rendre, ce qui peut se faire en `T[8]` $= 3$ pièces.

On a ensuite $3 = 1 +$ `T[12-6]` donc on continue par la pièce 6 et il reste $s = 2$ à rendre, ce qui peut se faire en `T[2]` $= 2$ pièces.

On a ensuite $2 = 1 +$ `T[2-1]` donc on continue par la pièce 1 et il reste $s = 1$ à rendre, ce qui peut se faire en `T[1]` $= 1$ pièces.

On termine avec $1 = 1 +$ `T[1-1]` donc on continue par la pièce 1 et il reste $s = 0$ à rendre, ce qui peut se faire en `T[0]` $= 0$ pièces donc on a terminé.

Il faut donc rendre `[12,6,1,1]`.

7. On suit la démarche précédente : on construit la tableau et on remonte depuis la dernière case en cherchant l'origine des minimums calculés pour trouver les pièces à utiliser jusqu'à tomber sur la somme $s = 0$:

```
def tableau(s,P) :
    T = [0]*(s+1)
    for n in range(1,s+1) :
        P_poss = []
        for k in P :
            if k <= n :
                P_poss.append(k)
        T[n] = 1+min(T[n-k] for k in P_poss)
    return T

def Rendu(s,P) :
    if s == 0 :
        return []
    else :
        L = []
        T = tableau(s,P)
        while s > 0 :
            for k in P : # il y a forcément un cas qui fonctionne
                if k <= s and T[s] == 1+T[s-k] :
                    p = k            # on utilise cette pièce
            L.append(p)
            s = s-p                  # on recommence sur la somme restante
        return L
```

8. On doit au moins, pour $s \geqslant 1$, utiliser une première pièce p_i (d'où le 1+) dont la valeur vérifie $p_i \leqslant s$; la somme à rendre est ensuite $s - p_i$, qui peut se rendre avec N_{s-p_i} pièces. Le rendu de s optimal est celui pour lequel la valeur de N_{s-p_i} est donc minimal (selon le système de pièces, il pourrait y avoir plusieurs possibilités).