B) Programmation dynamique

Définition : La programmation dynamique est une **classe d'algorithmes** qui résolvent un problème complexe en le **divisant en sous-problèmes** en conservant les résultats des sous-problèmes pour éviter de calculer à nouveau les mêmes résultats.
Cette méthode a été introduite au début des années 1950 par Richard Bellman. La programmation dynamique s'applique généralement **aux problèmes d'optimisation**. Nous avons déjà évoqué les problèmes d'optimisation lorsque nous avons étudié les algorithmes gloutons l'année dernière.
La programmation dynamique **s'applique quand les sous-problèmes se recoupent**, c'est-à-dire lorsque les sous-problèmes ont des problèmes communs (dans le cas du calcul de fib(6) on doit calculer 2 fois fib(4). Pour calculer fib(4), on doit calculer 4 fois fib(2)...).

Un algorithme de programmation dynamique résout chaque sous-problème une seule fois et mémorise sa réponse dans un tableau, évitant ainsi le recalcul de la solution chaque fois qu'il résout chaque sous-problème

C) Cas du rendu de monnaie

Rappel du problème : Vous avez à votre disposition un nombre illimité de pièces de 2 cts, 5 cts, 10 cts, 50 cts et 1 euro
Vous devez rendre une certaine somme (rendu de monnaie).

Le problème est le suivant : "**Quel est le nombre minimum de pièces qui doivent être utilisées pour rendre la monnaie**"

❖ **1ère Stratégie** : une méthode "gloutonne" de ce problème peut être la suivante :

- On prend la pièce de plus grande valeur (valeur de cette pièce doit être inférieure ou égale à la somme à rendre)
- On recommence l'opération ci-dessus jusqu'au moment où la somme à rendre est *égale à zéro.*

Appliquer cette méthode pour une somme de 1€ 77 (177 cts) à rendre : …………………………………………………………

……

Appliquer cette méthode à la somme de 11 centimes : ……………………………………………………………

……

Donner la solution pour 11 centimes…………………………………………………………………………..

❖ **2ème stratégie : Mise au point d'un algorithme récursif**

Soit X la somme à rendre, on notera nb(X) le nombre minimal de pièces nécessaires pour rendre la somme X.

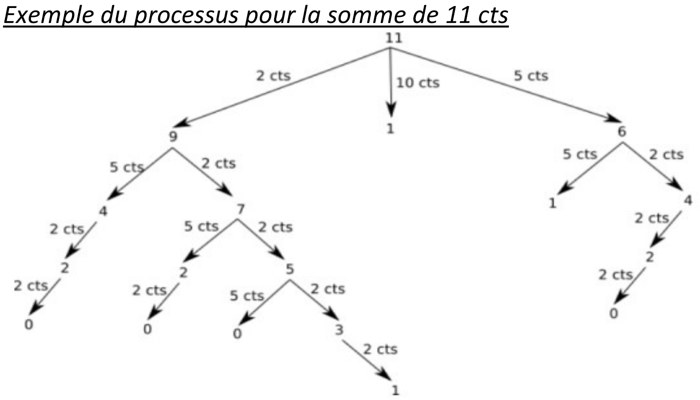
On note $p_1, p_2, \ldots, p_n$ les pièces à notre disposition

Principe de l'algorithme : Si on est capable de rendre la somme de X cts avec nb(X) pièces, on est alors capable de rendre la somme de X-p_i avec nb(X-p_i) (sachant la valeur de p_i est inférieure à X), soit :

- Si X=0 alors nb(X)=0
- Si X>0 alors nb(X)=nb(X-p_i) +1 avec $1 \le i \le n$ et $p_i \le X$

  Plus précisément c'est : Si X > 0 alors nb(X)=min(nb(X-p_i)) +1 avec $1 \le i \le n$ et $p_i \le X$
  Le « min » présent dans la formule de récurrence exprime le fait que le nombre de pièces à rendre pour une somme X-p_i doit être le plus petit possible

Exemple du processus pour la somme de 11 cts



Remarques :
- Tous les cas sont traités, quand un algorithme traite tous les cas on parle de méthode de **"force brute"**
- Pour certains cas, on se retrouve dans une impasse (cas où on termine par "1")
- La profondeur minimum de l'arbre (avec une feuille "0") est de 4, qui est donc la solution au problème
  (il existe plusieurs parcours (5,2,2,2) ou (2,5,2,2),…..)

Compléter et coder la fonction rendu_monnaie_rec(P,X) :

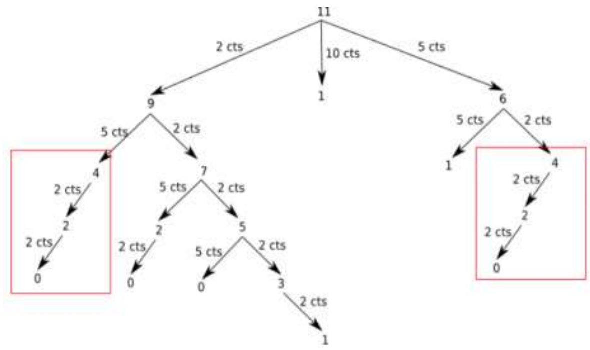
```python
Pieces = [100,50,20,10,5,2] # tableau des pièces disponibles

# rendu de monnaie en version recursif
def rendu_monnaie_rec(X,P):
    """ X est la somme à rendre de type entier et P est la
    liste des pièces disponibles trié dans l'ordre décroissar
    if X==0:  # cas de base
        return ......
    else:
        mini = 1000 # on utilisera au pire 1000 pièces !
        for piece in .....:
            if .......<= ....:
                nb = ... + rendu_monnaie_rec(..........,P) #
                if nb < .....:
                    .... = nb
    return ......
```

<u>Remarque</u> : Pour être sûr de renvoyer le plus petit nombre de pièces, on attribue dans un premier temps la valeur 1000 à la variable mini

- *Faire fonctionner ce programme pour la somme de 11 cts*
- *Puis évaluer le nb de pièces à rendre pour 25, 42 , 67, ……*
- *A partir de quelle somme le programme devient-il lent ?*

❖ 3ème <u>stratégie :</u> la programmation dynamique :

Comme vous l'avez sans doute constaté le programme ne permet pas toujours d'obtenir une solution, pourquoi ?
Parce que les appels récursifs sont trop nombreux, on dépasse la capacité de la pile. En y regardant de plus près, on s'aperçoit que certains calculs se font plusieurs fois (le rendu de 4 cts par exemple cf schéma ci-contre)
On va pouvoir appliquer la même méthode que pour Fibonacci en stockant dans un tableau les résultats des appels récursifs déjà calculés



<u>L'algorithme</u> :

<u>Entrées</u>: X un entier (somme à rendre en cts), m tableau contenant X+1 élément nul (None), P tableau des pieces

<u>Sortie</u> : un entier mini qui est le nombre minimal de pièces à rendre sur la monnaie X

Fonction rendu_monnaie_mem_rec (X,m,P):

Début

```
    Si X = …… alors
        renvoyer …..
    Sinon si m[X] ≠ ⋯ …            # valeur déjà calculée et placée dans le tableau
        renvoyer …..
    Sinon
        mini ← 1000
        Pour  p  dans Pieces faire
            Si …… <= ….. alors
                nb ← 1 + rendu_monnaie_mem_rec (…………………………)
                Si …….. < ……….. alors
                    …….. ← …..
                    m[X] ← ……….
                fin Si
            fin si
        fin pour
    fin si
    renvoyer ……
```

Fin

Implémenter cette fonction en python, on définit une fonction qui initialisera le tableau m et appellera notre fonction.
Le tableau **Pieces** sera définit à l'extérieur des fonctions
Vérifier que cette fois-ci, on trouve bien les solutions pour des sommes à rendre comme 177cts, 289cts.

```python
def rendu_monnaie_dynamique(X,P):
    """ fonction qui initialise le tableau
    mem=[None]*(X+1)
    return rendu_monnaie_mem_rec(X,mem,P)
```

<u>Bonus</u> : modifier le code pour qu'il renvoie la liste des pièces utilisées pour le rendu minimal