La programmation dynamique

La programmation dynamique est une technique algorithmique qui consiste à mémoriser le résultat d'une recherche pour ne plus avoir à la refaire ultérieurement si le programme en a besoin.

I- Prenons par exemple le calcul de la célèbre suite de Fibonacci (paniquez pas, c'est facile) :

$$u_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ u_{n-1} + u_{n-2} & \text{si } n \geq 2 \end{cases}$$

Les premiers termes de la suite sont :
0, 1, 1, 2, 3, 5, 8, 13, 21
Chaque terme, à partir du 3ème, se calcule en additionnant les deux termes précédents

L'écriture mathématique $U_n = U_{n-1} + U_{n-2}$ signifie simplement que l'on additionne les 2 termes précédents pour obtenir le suivant, n représentant la position ou le numéro dans la suite de nombres.

Compléter les termes de la suite pour U_9 et U_{10} :

Valeur	0	1	1	2	3	5	8	13	21		
Un	U_0	U_1	U_2	U_3	U_4	U_5	U_6	U_7	U_8	U_9	U_{10}

Par exemple $U_4 = U_3 + U_2 = 1+2=3$ ou encore $U_5 = U_4 + U_3 = 3 + 5 = 8$

Pour calculer U_9 et U_{10} vous pouvez soit partir de U_0 et U_1 et faire tous les calculs jusqu'à U_9 puis U_{10} , soit « voir » dans le tableau que U_7 et U_8 ont déjà été calculés, donc pas la peine de tout refaire depuis le début.

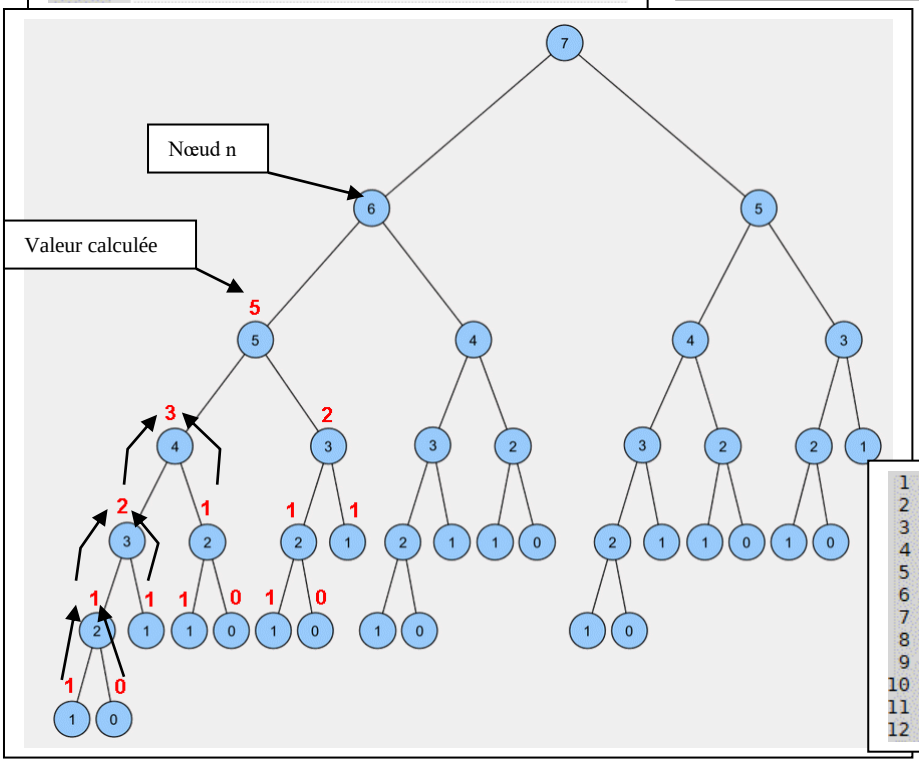
Et bien la programmation dynamique c'est juste pour traduire la deuxième méthode (instinctive pour un humain, pas pour la machine …). L'idée est de sauvegarder les résultats intermédiaires dans une mémoire (souvent un tableau ou une liste) afin de pouvoir les utiliser si nécessaire pour ne pas refaire tous les calculs.

Exercice à faire à la maison : faire le programme en version itérative du calcul de fibonacci.

Nous allons voir la version récursive :

```
1  # Fibonacci
2  # Version récursive
3
4  def fibo(n):
5      if n==0 or n==1:
6          return n
7      else:
8          return fibo(n-1) + fibo(n-2)
9
10
11 print (fibo(7))
```

Ce qui est beau et apprécié en récursif c'est que finalement cela ressemble fortement à la définition mathématique.
Le seul problème c'est que le programme va faire tout un paquet de calculs redondants !
Taper le programme et **vérifier** qu'il fonctionne correctement.



Compléter l'arbre de calculs.
Le calcul se fait en « remontant ».
Le parcours est en profondeur en visitant en priorité les fibo(n-1) .
La réécriture du programme ci-dessous vous permettra avec des print de visualiser les valeurs renvoyées gauche et droite.

```
1  # Fibonacci
2  # Version récursive
3
4  def fibo(n):
5      if n==0 or n==1:
6          return n
7      else:
8          gauche = fibo(n-1)
9          droite = fibo(n-2)
10         return (gauche + droite)
11
12 print (fibo(7))
```

Comme vous pouvez le constater, le programme fait de nombreuses fois les mêmes calculs (fibo(4) est calculé plusieurs fois, idem fibo(3) etc ...
Pour améliorer le programme, on va mémoriser les calculs de chaque nœud n dans un tableau et avant d'appeler la fonction récursive, on vérifie si le calcul n'a pas déjà été fait (et dans ce cas on renvoie la valeur mémorisée) :

<u>Taper et tester</u> le programme

```
1  # Fibonacci
2  # Version récursive
3
4  def fibo(n):
5      global t
6      print (n)
7
8      if n==0 or n==1:
9          t[n] = n
10         return n
11
12     if t[n] > 0:
13         return t[n]
14
15     t[n] = fibo(n-1) + fibo(n-2)
16     return (t[n])
17
18 t = [0]*8
19 print (t)
20 print ("resultat = ",fibo(7))
21 print (t)
```

Pour visualiser les nœuds parcourus.

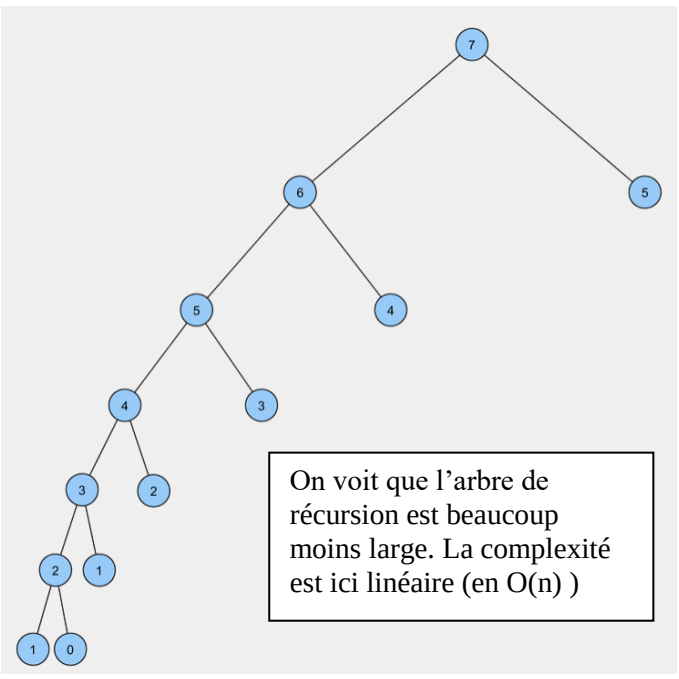
La valeur t[n] est renvoyée si elle est non nulle

Chaque valeur calculée de fibo(n) est enregistrée une case du tableau t : t[n].

```
[0, 0, 0, 0, 0, 0, 0, 0]
7
6
5
4
3
2
1
0
1
2
3
4
5
resultat =  13
[0, 1, 1, 2, 3, 5, 8, 13]
```

L'utilisation de variables globales dans une fonction est d'une façon générale à proscrire. Cependant la maîtrise de la récursivité est déjà très complexe et quand on débute l'introduction d'une petite variable globale permet souvent de se faciliter un peu la vie mais uniquement si vous bloquez avec que du local !
Ci-dessous, une solution équivalent plus « propre » (on passe t en paramètre) :

```
1  # Fibonacci
2  # Version récursive
3
4  def fibo(n,t):
5
6      print (n)
7
8      if n==0 or n==1:
9          t[n] = n
10         return n
11
12     if t[n] > 0:
13         return t[n]
14
15     t[n] = fibo(n-1,t) + fibo(n-2,t)
16     return (t[n])
17
18 t = [0]*8
19 print (t)
20 print ("resultat = ",fibo(7,t))
21 print (t)
```



On voit que l'arbre de récursion est beaucoup moins large. La complexité est ici linéaire (en O(n))

Corriger votre programme (sous nouveau nom) et vérifier qu'il fonctionne à l'identique

II- Algorithme du rendu de monnaie

Problématique : comment rendre la monnaie avec le moins de pièces possible ?

Exemple :
Vous avez à votre disposition un nombre illimité de pièces de 2 cts, 5 cts, 10 cts, 50 cts et 1 euro (100 cts). 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" ?

La résolution "gloutonne" de ce problème peut être la suivante :

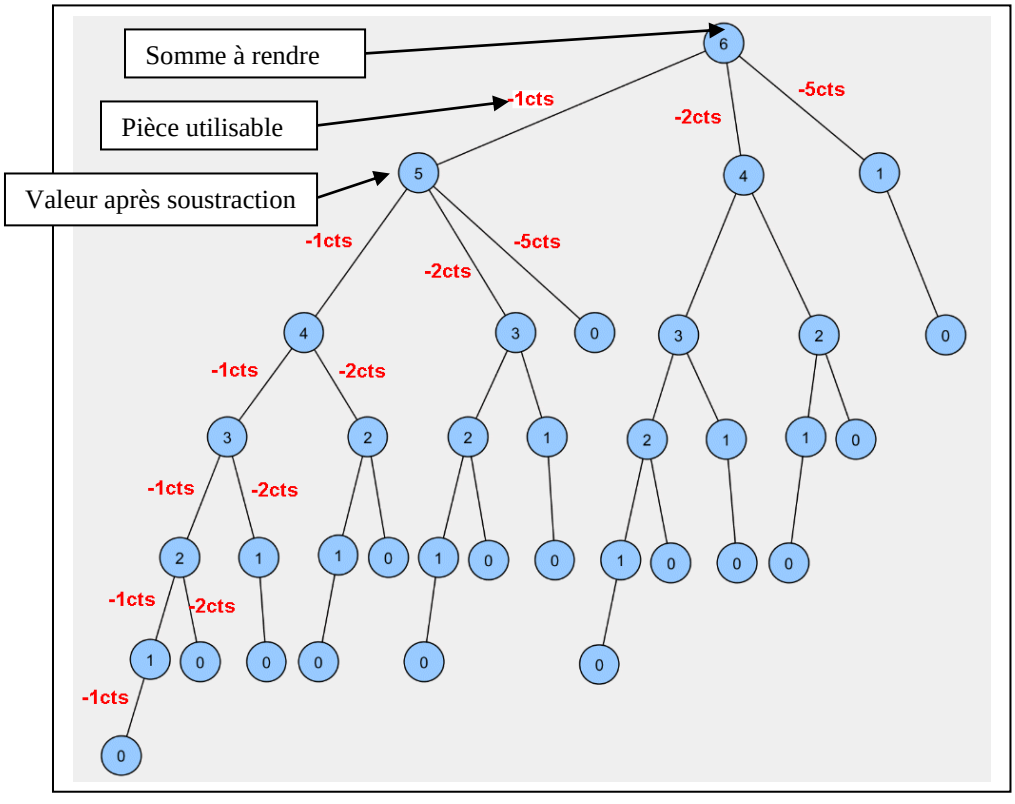
On prend la pièce qui a la plus grande valeur (il faut que la valeur de cette pièce soit inférieure ou égale à la somme restant à rendre)
On recommence l'opération ci-dessus jusqu'au moment où la somme à rendre est égale à zéro.

A faire : programmez la solution gloutonne !

Prenons un exemple :
Nous avons X = 1 euro 77 cts à rendre :
On utilise une pièce de **1 euro** (plus grande valeur de pièce inférieure à 1,77 euro), il reste 77 cts à rendre
On utilise une pièce de **50 cts** (plus grande valeur de pièce inférieure à 0,77 euro), il reste 27 cts à rendre
On utilise une pièce de **10 cts** (plus grande valeur de pièce inférieure à 0,27 euro), il reste 17 cts à rendre
On utilise une pièce de **10 cts** (plus grande valeur de pièce inférieure à 0,17 euro), il reste 7 cts à rendre
On utilise une pièce de **5 cts** (plus grande valeur de pièce inférieure à 0,07 euro), il reste 2 cts à rendre
On utilise une pièce de **2 cts** (plus grande valeur de pièce inférieure à 0,02 euro), il reste 0 cts à rendre

Supposons maintenant que l'on veuille rendre 11cts avec le même algorithme « glouton » :
On utilise une pièce de **10 cts** (plus grande valeur de pièce inférieure à 11 cts), il reste 1 ct à rendre
Il n'y a pas de pièce de 1cts, on est coincé, il faudrait revenir en arrière et prendre une pièce de 5cts et 6 pièces de 2cts. Sauf que mon algo glouton ne revient jamais en arrière.
Un algo glouton rendra donc 10cts, ce n'est pas une solution optimale

Résolution par force brute : on teste toutes les possibilités
Exemple ci-dessous, on traite pour une petite valeur avec X = 6 et pièces = 1cent 2 cents 5 cents



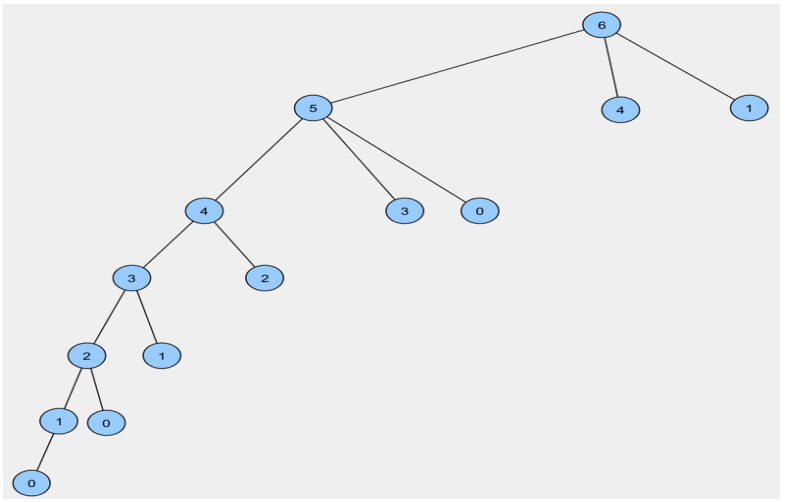
Principe :
On part de la somme totale à rendre et on soustrait successivement toutes les valeurs possibles des pièces.

Compléter le graphe avec les calculs effectués.
Repérer les redondances

Que peut-on dire du nombre de calculs à effectuer ?

Comment déduire de cet arbre le nombre de pièces minimum à utiliser ?

On voit que le nombre de calculs augmente exponentiellement avec la valeur de la somme à rendre (et du nombre de pièces utilisables)
Pour diminuer le nombre de calcul, il faut **mémoriser les calculs** déjà effectués pour ne pas avoir à les refaire. C'est donc le principe de la **programmation dynamique** qu'il faut appliquer :



Voici l'arbre obtenu si on enlève les redondances.

Exercice à faire :
On veut rendre 11cts en utilisant des pièces de 10cts, 5cts et 2cts.

Construire l'arbre des possibilités dans le cas force brute et dans le cas programmation dynamique.

Ce qu'il faut retenir de ce cours ce sont les grands principes qui caractérisent la programmation dynamique à travers les deux exemples (fibonacci et rendu de monnaie).
La construction du programme n'est pas vraiment de votre niveau…

Annexe : Programmation en python

```python
def recursiveChange(S,X):
    if X==0:
        return 0
    else:
        mini = X+1
        for i in range(len(S)):
            if S[i]<=X:
                nb = 1 + recursiveChange(S,X-S[i])
                if nb<mini:
                    mini = nb
        return mini

S=(1,2,5,10,20)
print(recursiveChange(S,30))
```

Programme en force brute, on explore toutes les possibilités.
Vous noterez un appel récursif à l'intérieure d'une boucle for …nécessite une bonne maîtrise de ce qu'on fait !
Le programme fait « remonter » le nombre minimum de pièces nécessaires pour chaque sous arbre

```python

# Rendu monnaie dynamique
# en version récursive

def rendu_monnaie_mem(P,X):
    mem = [0]*(X+1)
    return rendu_monnaie_mem_c(P,X,mem)

def rendu_monnaie_mem_c(P,X,m):
    if X==0:
        return 0
    elif m[X]>0:
        return m[X]
    else:
        mini = X+1
        for i in range(len(P)):
            if P[i]<=X:
                nb=1+rendu_monnaie_mem_c(P,X-P[i],m)

                if nb<mini:
                    mini = nb
                    m[X] = mini
    return mini


somme = 1000001
liste_pieces=[1545,574,56,45,40,10,2]

print (rendu_monnaie_mem(liste_pieces ,somme))
```

La version 'dynamique' consiste à mémoriser dans un tableau m les valeurs calculées de chaque nœud X.
Avant chaque appel de la fonction rendu_monnaie, on teste si la valeur n'a pas déjà été calculée (et dans ce cas, on la renvoie directement)

Mémorisation de la valeur minimale du nombre de pièces au nœud X de l'arbre.

A faire : Tester et comparer les vitesses d'exécution des deux programmes.