Récursivité, mémoïsation & programmation dynamique

1 Révisions : effet de bord

```python
def ordre1(M):
    N=sorted(M)
    return(N)

L=[2,1,4,3]
print(ordre1(L))
print(L)
```

```python
def ordre2(M):
    M.sort()
    N=M
    return(N)
L=[2,1,4,3]
print(ordre2(L))
print(L)
```

```python
def ajout1(M):
    M=M+[1]
    return(M)
L=[0]
print(ajout1(L))
print(L)
```

```python
def ajout2(M):
    M.append(1)
    return(M)
L=[0]
print(ajout2(L))
print(L)
```

Application : quel est l'affichage du programme :

```
M=[0,1,2]
def f(x):
   x[0]=x[1]*2
   return(x[0])
print(f(M),M)
```

2 Récurisivté

Une fonction récursive présente un comportement alternatif :

- soit la fonction s'appelle elle-même, mais avec des données qui ont été transformées, pour s'approcher de la solution du problème ;
- soit la fonction ne s'appelle pas elle-même, car les arguements sont suffisamment simples pour être traitées directement. On parle du « cas trivial », c'est la condition d'arrêt de récursivité .

2.1 Application : découper en 2 ou 3 m

Soit n un entier positif. On cherche à déterminer de combien de manière il est possible de découper une barre de n mètres en morceaux de 2 ou 3 mètres en tenant compte de l'ordre (il ne peut pas y avoir une chute de 1m).

Une barre de 8 mètre pour par exemple se découper de quatre manières différentes :

$$8 = 2+2+2+2 = 2+3+3 = 3+2+3 = 3+3+2$$

.

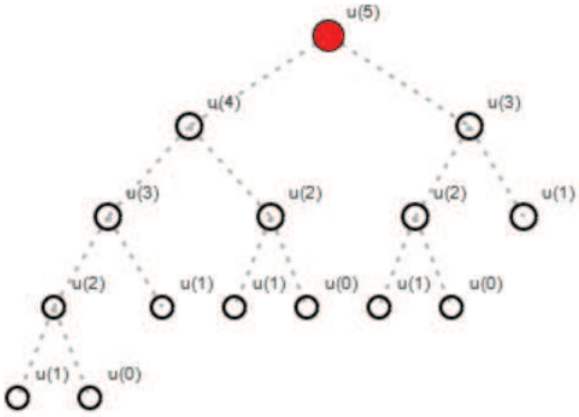
Soit $d(n)$ le nombre de découpes possibles pour un morceau de longueur n. On a $d(0) = d(1) = 0$ et $d(2) = d(3) = 1$. Si $n \geq 4$, on peut séparer la situation en deux cas : soit le premier morceau est de taille 2, et il y a $d(n-2)$ façon de découper le morceau restant, soit il est de taille 3, et il y a $d(n-3)$ façon de découper le morceau restant. On en déduit que pour tout $n \geq 3$, $d(n) = d(n-2) + d(n-3)$.

Écrire une fonction récursive prenant en entrée un entier n et retournant $d(n)$.

3 Mémoïsation

Certaines executions récursives de programme utilisent plusieurs fois les mêmes arguments lors d'appel différents de fonction. C'est par exemple le cas lors d'un calcul récursif de la suite de Fibonacci :

```python
def Fib(n) :
    if n ==1 :
        return(1) #F1
    elif n <=0 :
        return(0) #F0
    else:
        return(Fib(n-1) + Fib(n - 2))
print(Fib(5))
```



On peut alors améliorer la rapidité du programme en conservant les résultats des différents appels récursifs. Ces résultats sont conservés dans une variable globale qui sera une liste, ou pour plus d'ergonomie un dictionnaire.

```python
def fastFib(n):
    if n not in dic:
        dic[n] = fastFib(n-1) + fastFib(n-2)
    return dic[n]
dic = {0:0, 1:1} #conditions initiales
print(fastFib(5))
```

3.1 Application

Reprendre l'exercice 2.1 avec mémoïsation.

4 Programmation dynamique

La programmation dynamique est :
- une famille de méthodes de résolution utilisant récursivité et mémoïsation ;
- pour résoudre des problème de programmation, au sens planification d'opérations ;
- pour résoudre des problèmes d'optimisation combinatoire.

On rappelle qu'un problème d'optimisation peut se mettre sous la forme : $\begin{cases} \max_{x\in\mathcal{E}} f(x) \\ g(x) \leq 0 \end{cases}$.

Un problème d'optimisation combinatoire s'exprime : $\begin{cases} \max_{x_i\in[\![0;1]\!]^n} \sum x_i v_i \\ \sum x_i p_i \leq C \end{cases}$.

4.1 Exemple : rendu de monnaie ou partition de barres

Le problème du rendu de monnaie vise à minimiser le nombre de pièces nécessaires à réaliser une certaine somme S, dans un système monétaire σ défini par certaines valeurs $\sigma = \{v_1, v_2, \ldots, v_n\}$, où $S = \sum x_i v_i$. On cherche à minimiser $\sum x_i$.

4.1.1 Algorithme glouton

Une méthode non optimale est la méthode gloutonne (les valeurs v_i sont rangées par ordre décroissant) :
- tant que vous le pouvez, donnez des pièces de plus grande valeur ;
- lorsque ce n'est plus possible, passez à la valeur suivante, et recommencez ;
- arrêtez lorsque vous avez rendu la somme S.

Exemple : quel est le rendu de 8€ avec :
- $\sigma = 10, 5, 2, 1$;
- $\sigma = 6, 4, 1$;

4.1.2 Méthode récursive

Pour déterminer l'optimum, il est nécessaire de déterminer toutes les combinaisons possibles, et de choisir la ou les meilleure(s). On peut parcourir toutes ces combinaisons en utilisant une méthode récursive. Il faut alors exprimer le problème sous forme récursive. En notant `S` la somme à rendre, `V` la liste des valeurs monaitaire rangées par ordre décroissant, et `L` la liste des pièces déjà rendues :

Algorithme

Algorithme 1 : Rendu(S, L)

```
1 si S=0 alors
2 |  afficher L
3 sinon
4 |  pour chaque p de V faire
5 |  |  si p ≤S et p n'est pas plus grande que la dernière valeur de L alors
6 |  |  |  Rendu(S-p, L+[p])
```

python

```
def Rendu(S,L):
    #print(S)
    if S==0:
        print(L)
    else:
        for p in V:
            if p<=S and
            (len(L)==0 or len(L)>0 and p<=L[-1]) :
                Rendu(S-p,L+[p])

V=[6,4,1]
Rendu(8,[])
```