TD: programmation dynamique:

Exercice 1: suite de Fibonacci: On programme la suite de Fibonacci $u_{n+2} = u_{n+1} + u_n$ à l'aide du code suivant:

```
def fibo_rec(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibo_rec(n-1) + fibo_rec(n-2)
```

fibo.py

1. Quelles sont les valeurs de u_0 et de u_1?
2. Quel type de programmation est utilisé ici?
3. Calculer u_{20}, u_{30} et u_{40}. Quel problème rencontre t-on? Expliquer l'orgine de ce problème.

   On améliore ce programme en enregistrant chaque valeur calculée afin de l'utliser ensuite si besoin.

4. Écrire une fonction utilisant un dictionnaire permettant le calcul des éléments de la suite de Fibonacci. Le programme commencera de la manière suivante:

```
def fibo_dic_rec(n, memo = {}):
    if n == 0 or n == 1:
        return n
```

fibo.py

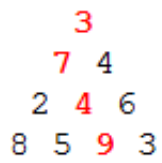
5. Même question avec l'utilisation d'un tableau. Le programme commencera de la manière suivante et renverra un tableau des éléments de la suite:

```
def fibo_tab_rec(n, tab = []):
    if n == 0:
        return [0]
    elif n == 1:
        return [0,1]
```

fibo.py

6. A l'aide de la bibliothèque time, comparer les temps d'éxécution de ces fonctions pour le calcul de u_{35}. Conclure.
7. Écrire une fonction itérative permettant le calcul des éléments de la suite de Fibonacci. Calculer u_{35} et comparer le temps d'exécution de ce calcul aux fonctions récursives utilisées dans les questions précédentes.

Exercice 2: somme maximale des éléments d'une pyramide:
On dispose d'une pyramide de la forme suivante. En partant du sommet, et en se dirigeant vers le bas à chaque étape, on cherche à maximiser le total des nombres traversés. Dans cet exemple, le chemin est le suivant et le total 23: On peut procéder de la manière suivante:



- si le nombre x est à la base de la pyramide: $m(x) = x$
- sinon, $m(x) = x + max(m(g(x)), m(d(x)))$

où m(x) est le maximum cherché, g(x) et d(x) sont les nombres à gauche et à droite sous x.

Une pyramide sera implémentée sous la forme d'un tableau de tableau de la manière suivante:

```
0 p = [[3],[7,4],[2,4,6],[8,5,9,3]]
```

pyramide.py

1. Écrire une fonction récursive permettant le calcul du maximum et la tester sur la pyramide p. Cette fonction sera de la forme:

```
0 def pyramide(p,i,j):
      n = len(p)
2     if i == n-1:
          return p[i][j]
4     else:
```

pyramide.py

2. On souhaite tester cette fonction sur une pyramide de hauteur plus importante.

   2.1 Écrire une fonction permettant de générer une pyramide aléatoire de hauteur fixée.

   2.2 Tester la fonction *pyramide()* sur une pyramide de hauteur 24. Que constate-on?

3. Afin de permettre un calcul plus efficace, on modifie la fonction pyramide afin de stocker les valeurs calculées. On utilise un dictionnaire avec des clés formées des deux indices des nombres correspondants:

```
0 def pyramide_dyn(p, i, j, m={}):
      if m.get((i,j)):
2         return m[(i,j)]
      if i == len(p) - 1:
4         m[(i,j)] = p[i][j]
          return p[i][j]
6     else:
```

pyramide.py

   3.1 Compléter le programme précédent.

3.2 Le tester sur la pyramide p et sur une pyramide de hauteur 24. Que constate-on?

3.3 Comparer alors les temps d'éxécution des fonctions *pyramide()* et *pyramide_dyn()* sur une pyramide de hauteur 24.

3.4 Modifier la fonction *pyramide_dyn()* afin de permettre l'affichage du dictionnaire.

4. On programme maintenant la fonction de la manière suivante:

```
0 def pyramide_iter(p):
      n = len(p)
2     q = [[p[i][j] for j in range(i+1)] for i in range(n)]
      for i in range(n-2, -1, -1):
4         for j in range(i+1):
              q[i][j] = q[i][j] + max(q[i+1][j], q[i+1][j+1])
6     return q[0][0]
```

pyramide.py

4.1 Expliquer le déroulement des calculs sur la pyramide p.

4.2 Comparer les temps d'éxécution des fonctions *pyramide_iter()* et *pyramide_dyn()* sur une pyramide de hauteur 24.

Exercice 3: le problème du rendu de monnaie:

Il s'agit de rendre une somme v avec un système de pièces $s = (p_1, p_2, p_3, ...)$ où les p_i sont les valeurs des pièces. On suppose $p_1 = 1$ (donc le problème a toujours une solution) et $p_1 < p_2 < p_3...$

Ce problème peut être résolu avec un algorithme glouton qui donne une solution optimale dans la pluspart des systèmes de pièces en vigueur (programme de première NSI).

La programmation dynamique permet d'envisager un autre type de résolution qui donne une solution optimale dans tous les cas.

1. **Méthode itérative:**
   Pour rendre une somme v, on peut rendre une pièce quelconque p_i. Il reste alors à rendre la somme $v - p_i$. Donc si on sait rendre toute somme inférieure à v de manière optimale, il suffit de tester toutes les différentes possibilités de rendre $v - p_i$ pour toute pièce p_i et de choisir la meilleure.

   1.1 On note $min(v, s)$ le nombre minimal de pièces pour rendre la somme v avec le système s. Donner le lien entre $min(v, s)$ et $min(v - p_i, s)$

   1.2 On propose le programme suivant:

```python
def monnaie_dyn(systeme, valeur):
    n = len(systeme)
    sol = [valeur] * (valeur + 1)
    sol[0] = 0
    for v in range(1, valeur+1):
        for i in range(n):
            if v - systeme[i] >= 0:
                sol[v] = min(sol[v-systeme[i]]+1,sol[v])
    return sol[-1]


s = [1, 2, 5, 10]
print(monnaie_dyn(s, 14))
```

rendu_monnaie.py

1.2.1 Le tester et expliquer ce qu'il renvoie.

1.2.2 Modifier ce programme afin d'afficher les différentes valeurs de la variable sol. Expliquer alors son fonctionnement.

1.2.3 La programmation est-elle réellement dynamique?

1.3 On propose de le modifier comme suit:

```python
def monnaie_dyn_pieces(systeme, valeur):
    n = len(systeme)
    sol = [[i, [i] + [0 for i in range(n-1)]] for i in range(valeur+1)]
    for v in range(1, valeur+1):
        for i in range(n):
            if v - systeme[i] >= 0:
                temp = sol[v-systeme[i]][0] + 1
                if sol[v][0] > temp:
                    sol[v][0] = temp
                    sol[v][1] = list(sol[v-systeme[i]][1])
                    sol[v][1][i] += 1
    return sol[-1]

s = [1, 2, 5, 10]
print(monnaie_dyn_pieces(s, 14))
```

rendu_monnaie.py

1.3.1 Le tester et expliquer ce qu'il renvoie.

1.3.2 Modifier ce programme afin d'afficher les différentes valeurs de la variable sol. Expliquer alors son fonctionnement.

2. **Méthode récursive:**
   On propose le programme récursif suivant pour le rendu de monnaie:

```python
def nb_min(dispos, rendre):
    from math import inf
    if dispos == []:
        if rendre == 0:
            return 0, []
        else:
            return inf, []
    elif dispos[0][1] > rendre:
        return nb_min(dispos[1:], rendre)
    else:
        piece = dispos[0]
        #branche gauche:
        nb_avec, rendre_avec = nb_min(dispos, rendre - piece[1])
        nb_avec += 1
        #branche droite:
        nb_sans, rendre_sans = nb_min(dispos[1:], rendre)
        #choix:
        if nb_avec < nb_sans:
            return nb_avec, rendre_avec + [piece[0]]
        else:
            return nb_sans, rendre_sans

pieces1 = [['p1', 1], ['p2', 2], ['p3', 5]]
print(nb_min(pieces1, 8))
```

rendu_monnaie.py

2.1 Identifier les différents types utilisés dans cette fonction.

2.2 Il s'agit d'une fonction récursive. Préciser le cas de base.

2.3 Expliquer le fonctionnement de cette fonction. On pourra utiliser un arbre binaire afin de modéliser son éxécution.

2.4 Cette fonction fait-elle appel à de la programmation dynamique?

2.5 On propose alors les modifications suivantes:

```python
def nb_min_dyn(dispos, rendre, memo = {}):
    from math import inf
    if (len(dispos), rendre) in memo:
        return memo[(len(dispos), rendre)]
    elif dispos == []:
        if rendre == 0:
            res = 0, ()
        else:
            res = inf, ()
    elif dispos[0][1] > rendre:
        return nb_min_dyn(dispos[1:], rendre, memo)
    else:
        piece = dispos[0]
        #branche gauche:
        nb_avec, rendre_avec = nb_min_dyn(dispos, rendre - piece[1], memo)
        nb_avec += 1
        #branche droite:
        nb_sans, rendre_sans = nb_min_dyn(dispos[1:], rendre, memo)
        #choix:
        if nb_avec < nb_sans:
            return nb_avec, rendre_avec + (piece[0],)
```

```
        else:
            return nb_sans, rendre_sans
    memo[(len(dispos), rendre)] = res
    return res


print(nb_min_dyn(pieces1, 8))
```

rendu_monnaie.py

Quel type est utilisé pour la mémoïsation? Expliquer laors le fonctionnement de cette nouvelle fonction.

Exercice 4: le problème du sac à dos:

Nous disposons d'un ensemble de n objets. Chaque objet o_i a une valeur n_i et un poids p_i. Il s'agit de déterminer l'ensemble d'objets qui a la plus grande valeur que nous pouvons emporter dans le sac sachant que le sac supporte un poids maximal P. Ici, le sac peut contenir 15 kg. Les poids sont en kg et les valeurs en euros:

objet	valeur	poids
1	126	14
2	32	2
3	20	5
4	5	1
5	18	6
6	80	8

Ces données seront implémentées de la manière suivante:

```
objets = [['objet 1', 126, 14], ['objet 2', 32, 2], ['objet 3', 20, 5], ['objet 4', 5,
    1], ['objet 5', 18, 6], ['objet 6', 80, 8]]
```

sac_dos.py

1. En vous aidant de la fonction *nb_min()* du rendu de monnaie, complétez les points d'interrogation de la fonction suivante. Cette fonction est-elle programmée dynamiquement?

```
def valeur_max(dispos, reste):
    if dispos == [] or reste == 0:
        return 0, []
    elif dispos[0][2] > reste:
        return valeur_max(dispos[1:], reste)
    else:
        element = dispos[0]
        #branche gauche:
```

```
        valeur_avec, reste_avec = valeur_max(???, ???)
        valeur_avec += element[1]
        #branche droite:
        valeur_sans, reste_sans = valeur_max(???, ???)
        #choix:
        if valeur_avec > valeur_sans:
            return ???, ???
        else:
            return ???, ???
```

sac_dos.py

2. On propose alors la fonction suivante. Quel type est utilisé pour la mémoïsation? Complétez alors les points d'interrogation.

```
def valeur_max_dyn(dispos, reste, memo = {}):
    if (len(dispos), reste) in memo:
        return memo[(len(dispos), reste)]
    elif dispos == [] or reste == 0:
        res = 0, ()
    elif dispos[0][2] > reste:
        res = valeur_max_dyn(dispos[1:], reste, memo)
    else:
        element = dispos[0]
        #branche gauche:
        valeur_avec, reste_avec = valeur_max_dyn(???, ???, ???)
        valeur_avec += element[1]
        #branche droite:
        valeur_sans, reste_sans = valeur_max_dyn(???, ???, ???)
        #choix:
        if valeur_avec > valeur_sans:
            res = valeur_avec, reste_avec + (element,)
        else:
            res = valeur_sans, reste_sans
    memo[???, ???)] = res
    return res
```

sac_dos.py

3. Il est aussi possible de programmer une telle fonction de la manière suivante. A quel paradigme appartient ce type de programmation? Est-elle dynamique?

```
def sacados(choix, taille):
    val = [(taille+1)*[0] for i in range(len(choix)+1)]
    for i in range(len(choix)):
        for t in range(1, taille + 1):
            if t - choix[i][2] < 0:
                val[i+1][t] = val[i][t]
            else:
                ti, di = choix[i][2], choix[i][1]
                val[i+1][t] = max(val[i][t], val[i][t-ti] + di)
    return val[-1][-1]
```

sac_dos.py