Correction TD Info - Programmation dynamique

I - Exercices de mise en jambe

1 Suite récurrente simple

Q1. Effectuer une programmation itérative d'une fonction `u_iter` d'argument `n:int` permettant de calculer u_n. Vérifier que vous obtenez $u_{10} = 3\,628\,800$.

Cette question ne doit normalement pas vous poser de problèmes. Il s'agit du genre de questions que vous aviez lors des premiers TDs d'informatique l'année dernière. Il est important de savoir traiter cet exemple.

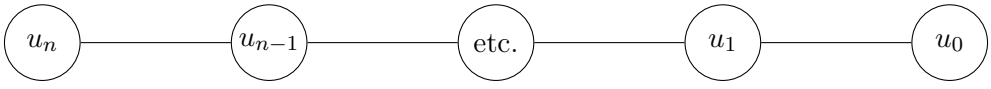
```
1  def u_iter(n:int)->int:
2      u=1 # Initialisation u0
3      for i in range(1,n+1): # Boucle de n itérations commençant à 1 car u1=1*u0
4          u=i*u # U_n=n*U_(n-1)
5      return u
6
7  >>> print(u_iter(10))
8  3628800
```

Q2. Effectuer une programmation récursive d'une fonction `u_rec` d'argument `n:int` permettant de calculer u_n. Vérifier que vous obtenez $u_{10} = 3\,628\,800$.

```
1  def u_rec(n:int)->int:
2      # Cas de base
3      if n==0:
4          return 1
5      # Cas récursif
6      else:
7          return n*u_rec(n-1)
8  >>> print(u_rec(10))
9  3628800
```

Q3. Justifiez que le principe de programmation dynamique ne s'applique pas dans ce cas d'étude (par exemple pas besoin de memoïsation).

La programmation récursive effectuée n'effectue pas de calculs redondants. En effet, la relation de récurrence est « simple ». Le calcul de u_n nécessite de connaître u_{n-1} etc. Ces calculs étant liés de manière simples, il n'y a pas de redondances dans ceux-ci. En construisant l'arbre d'appels récursifs, on a :



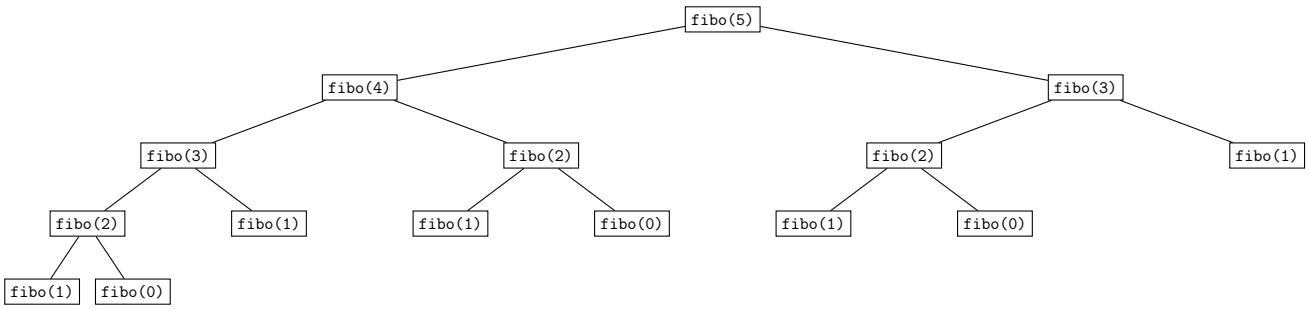
On constate alors l'inutilité de mettre en place une mémoïsation pour résoudre ce problème.

2 Suite de Fibonacci

Q4. Effectuer une programmation récursive d'une fonction `fibo` d'argument `n:int` permettant de calculer u_n. Vérifier que vous obtenez $u_{10} = 55$.

```
def fibo(n:int)->int :
    # Cas de base
    if n==0:
        return 0
    elif n==1:
        return 1    # Cas récursif
    else:
        return fibo(n-1)+fibo(n-2)
```

Q5. Construire l'arbre d'appels récursifs de `fibo(5)`. Justifier alors que la programmation dynamique est tout à fait indiqué pour résoudre plus efficacement ce problème.



On constate qu'il y a une redondance importante de calculs. Par exemple, on calcule deux fois `fibo(3)`. On peut donc utiliser le principe de mémoïsation pour éviter d'avoir à refaire ce calcul. Une approche bottom-up est également envisageable.

Q6. Effectuer une programmation dynamique `top-down` (principe de mémoïsation) en vous appuyant sur la structure de code proposée. **Dans le cadre d'une mémoïsation, cette structure est à privilégier**. Vérifier que `fibonacci(10)` renvoie bien 55.

```
def fibonacci(n:int)->int:
    tab_fab=[0 for i in range(n+1)] # Initialisation du tableau de mémoïsation
    tab_fab[1]= ..... # Initialisation utile pour u_1, u_0 est initialisé à 0
    resultat=memoisation(n,tab_fab) # renvoie le résultat de u_n
    return resultat

def memoisation(n,tab_fab):
    # tab_fab est une liste qui est mise à jour d'appels en appels tout en conservant les valeurs renseignées grâce à l'effet de bord.
    if n==0:
        return .....
    elif n==1:
        return .....
    else :
        if tab_fab[n]!=0:
            return .....
        else:
            resultat= ..... + .....
```

```
            tab_fab[n]=.............
            return .............
```

```
def fibonacci(n:int)->int:
    tab_fab=[0 for i in range(n+1)] # Initialisation du tableau de mémoïsation
    tab_fab[1]=1 # Initialisation utile pour u_1, u_0 est initialisé à 0
    resultat=memoisation(n,tab_fab) # renvoie le résultat de u_n
    return resultat

def memoisation(n,tab_fab):
    # tab_fab est une liste qui est mise à jour d'appels en appels tout en
        conservant les valeurs renseignées grâce à l'effet de bord.
    if n==0:
        return tab_fab[0]
    elif n==1:
        return tab_fab[1]
    else :
        if tab_fab[n]!=0:
            return tab_fab[n]
        else:
            resultat=memoisation(n-1,tab_fab)+memoisation(n-2,tab_fab)
            tab_fab[n]=resultat
            return resultat

>>> print(fibonacci(10))
55
```

Q7. Reprendre cette question mais en utilisant un dictionnaire pour réaliser la mémoïsation.

Ici, on crée un dictionnaire `tab_fab` qu'on initialise seulement avec les deux premières valeurs connues de la suite de Fibonacci à savoir u_0 et u_1. On le complétera au fur et à

```
def fibonacci(n:int)->int:
    tab_fab={0:0,1:1}# Initialisation du tableau de mémoïsation,
    resultat=memoisation(n,tab_fab) # renvoie le résultat de u_n
    return resultat

def memoisation(n,tab_fab):
    # tab_fab est un dictionnaire qui est mis à jour d'appels en appels tout en
        conservant les valeurs renseignées grâce à l'effet de bord
    if n==0:
        return tab_fab[0]
    elif n==1:
        return tab_fab[1]
    else :
        if n in tab_fab: # Si n est une clé de la dictionnaire tab_fab...
            return tab_fab[n]
        else:
            resultat=memoisation(n-1,tab_fab)+memoisation(n-2,tab_fab)
            tab_fab[n]=resultat
            return resultat

>>> print(fibonacci(10))
55
```

Q8. Effectuer maintenant une programmation bottom-up `fibo_bottom_up` afin de calculer u_n. Celle-ci se fera en débutant par le calcul de u_2, puis de u_3 etc. Vous serez attentif à ne pas utiliser un tableau pour stocker les résultats.

```python
def fibo_bottom_up(n:int)->int:
    u_old_old=0 # u_old_old représente u(i-2)
    u_old=1 # u_old représente u(i-1)
    for i in range(2,n+1) : # Calcul de u_2 jusque u_n
        u=u_old_old+u_old # Calcul de u(i) en fonction de u(i-2) et u(i-1)
        # Mise à jour des données. ATTENTION à l'ordre dans lequel la mise à
            jour est faite.
        u_old_old=u_old
        u_old=u
    return u
```

Dans ce code, il faut faire attention de ne pas écrire :

```
u_old=u
u_old_old=u_old
```

Car dans ce cas, vous obtenez la même valeur de `u`, de `u_old` et de `u_old_old`.

II - Découpe de barres d'acier

1 Approche par force brute

Q1. En considérant les doublons possibles, quel est le nombre de possibilités de découpes de la barre de longueur n ?

On considère une barre de longueur n. On va utiliser un code binaire pour spécifier si la découpe est réalisée. Pour la valeur 0, la barre n'est pas coupée, pour la valeur 1, la barre est coupée. Par exemple, pour une barre de longueur 4, le code suivant `011` signifie qu'après 1 cm la barre n'est pas coupée, après 2 cm la barre est coupée et après 3 cm la barre est coupée. Ainsi, la découpe obtenue est : un morceau de 2 cm et deux morceaux de 1 cm. De ce fait, pour une barre de longueur n, le mot binaire associé aux découpes possibles a une taille de $n-1$. C'est-à-dire que le nombre de découpes possibles en tenant compte des doublons est $\boxed{2^{n-1}}$.

Q2. Quand bien même il y a des découpes équivalentes possibles, une approche par force brute vous paraît-elle possible pour un nombre n conséquent ?

La complexité du problème étant exponentielle avec les doublons, quand bien même ceux-ci sont éliminés, celle-ci restera de toute façon importante (elle restera exponentielle en réalité).
Une approche par force brute est donc irréaliste.

2 Analyse du problème

Q3. En supposant que $r_0 = 0$, écrire alors le problème r_n sous sa nouvelle forme.

De cette façon, on pose le problème sous la forme :

$$\boxed{r_n = \max_{1 \leqslant i \leqslant n} (p_i + r_{n-i})}$$

Q4. On pose la liste `p=[0,1,5,8,9,10,17,17,20,24,30]`, le tableau des prix. Établir la programmation d'une fonction `couper_barre` d'arguments `p` et d'un entier `n`, longueur de la barre avec `n`$\leqslant$`len(p)-1` permettant de résoudre le problème de découpe proposé.

```
p=[0,1,5,8,9,10,17,17,20,24,30]
def couper_barre(p:[int],n:int)->int:
    ''' Fonction permettant de trouver le revenu maximal possible pour une barre
        de longueur n
    Entrée :
        - liste p : tableau des prix
        - n : longueur de la barre
    La longueur de la liste p est plus grande que n d'une unité.
    Sortie :
        - rev : revenu maximal possible pour la découpe des barres '''
    # On traite le cas de base
    if n==0:
        return 0
    # Cas récursif
    else:
        rev=0
        for i in range(1,n+1):
            rev=max(rev,p[i]+couper_barre(p,n-i))
        return rev
```

Q5. Modifier votre fonction, pour compter le nombre d'appels récursifs. Tester que votre code fonctionne en vérifiant que vous obtenez le bon résultat de revenu maximal pour des barres de longueur différentes. Pour $n = 4$ et $n = 10$, combien d'appels sont effectués ? Effectuer une conjecture sur le nombre d'appels à la fonction à réaliser pour obtenir la solution du problème.

On va utiliser un compteur que l'on va incrémenter au début de la fonction. Dans chaque `return`, on va retourner la valeur du compteur pour pouvoir l'exploiter lors du prochain appel récursif. On aurait également pu utiliser une variable globale.

```
p=[0,1,5,8,9,10,17,17,20,24,30]
def couper_barre(p:[int],n:int,compt=0)->int:
    # Par défaut, la valeur du compteur vaut 0
    compt+=1
    # On traite le cas de base
    if n==0:
        return 0,compt
    # Cas récursif
    else:
        rev=0
        for i in range(1,n+1):
            temp=couper_barre(p,n-i,compt)
            rev=max(rev,p[i]+temp[0])
            compt=temp[1]
        return rev,compt
```

ou

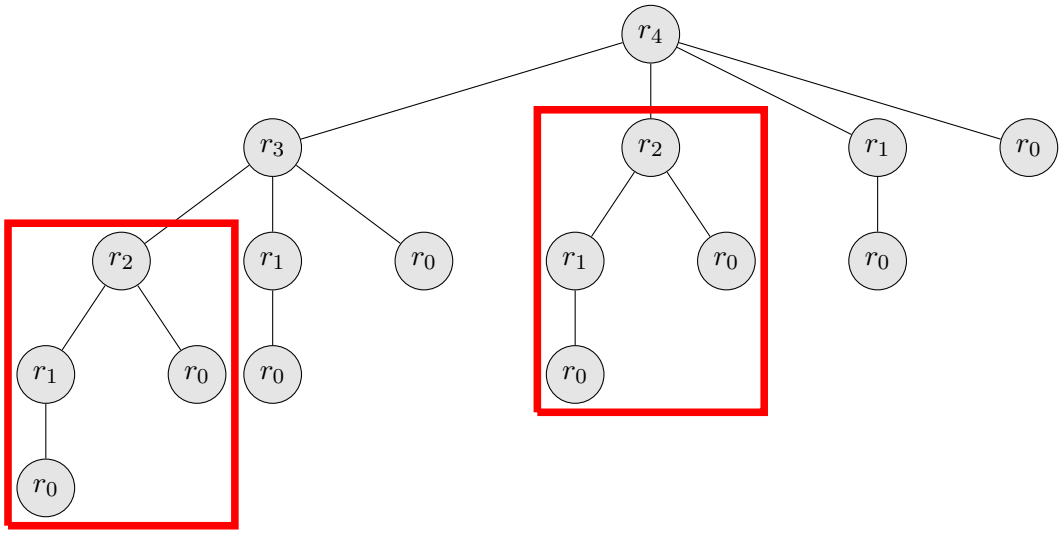
```
p=[0,1,5,8,9,10,17,17,20,24,30]
compt=0
def couper_barre(p:[int],n:int)->int:
    global compt
    compt+=1
    # On traite le cas de base
    if n==0:
        return 0,compt
    # Cas récursif
    else:
```

```
        rev=0
        for i in range(1,n+1):
            rev=max(rev,p[i]+couper_barre(p,n-i))
        return rev
```

```
>>> print(couper_barre(p,4))
(10,16)
>>> print(couper_barre(p,10))
(30,1024)
```

Pour $n = 4$, on trouve qu'il faut 16 appels à la fonction `couper_barre` pour résoudre le problème et pour $n = 10$, il faut 1024 appels à cette même fonction.
On peut conjecturer que le nombre d'appels à la fonction est de 2^n.

Q6. Pour $n = 4$, dessiner l'arbre d'appels récursifs et justifier qu'une programmation dynamique est envisageable.



On constate, par exemple, sur ce graphe des appels de la fonction que le sous-problème r_2 est appelé deux fois. Celui-ci faisant appels à d'autres sous-problèmes, on retrouve une propriété de sous-structure optimale. La programmation dynamique est donc toute indiquée pour résoudre ce problème.

3 Programmation dynamique top-down : mémoïsation

Q7. Compléter le code permettant de résoudre le problème. Que contient la liste `r` ?

```
p=[0,1,5,8,9,10,17,17,20,24,30]
def couper_barre(p:[int],n:int)->int:
    r=[-1 for i in range(len(p))] # Initialisation du tableau qui contiendra la
        résolution du problème pour r[i]
    r[0]=0 # Pour r_0=0.
    return memoisation_coupe_barre(p,n,r)

def memoisation_coupe_barre(p,n,r):
    # Si la solution est connue
    if r[n]>=0:
        return r[n]
# Cas de base du problème récursif
    if n==0:
        return 0
```

```
    # Cas récursif
    else:
        rev=-1 # Initialisation du revenu possible pour une barre de longueur n
        for i in range(1,n+1):
            rev=max(rev,p[i]+memoisation_coupe_barre(p,n-i,r))
    r[n]=rev # Mise à jour de r
    return rev

>>> solution,tableau_r=couper_barre(p,10)
>>> print(solution,tableau_r)
30 [0, 1, 5, 8, 10, 13, 17, 18, 22, 25, 30]
```

La liste `r` contient donc l'ensemble des solutions au problème r_i avec $i \in [\![0, n]\!]$.

Q8. Modifier votre code pour savoir combien d'appels à la fonction `memoisation_coupe_barre`.

On utilise une variable locale.

```
p=[0,1,5,8,9,10,17,17,20,24,30]
def couper_barre(p:[int],n:int)->int:
    r=[-1 for i in range(len(p))] # Initialisation du tableau qui contiendra la
        résolution du problème pour r[i]
    r[0]=0 # Pour r_0=0.
    compteur=0
    solution,compteur=memoisation_coupe_barre(p,n,r,compteur)
    return solution,compteur,r

def memoisation_coupe_barre(p,n,r,compteur):
    compteur+=1
    # Si la solution est connue
    if r[n]>=0:
        return r[n]
# Cas de base du problème récursif
    if n==0:
        return 0
    # Cas récursif
    else:
        rev=-1 # Initialisation du revenu possible pour une barre de longueur n
        for i in range(1,n+1):
            temp=memoisation_coupe_barre(p,n-i,r,compteur)
            rev=max(rev,p[i]+temp[0])
            compteur=temp[1]
    r[n]=rev # Mise à jour de r
    return rev,compteur

>>> solution,compteur,tableau_r=couper_barre(p,10)
>>> print(compteur)
56
```

On trouve alors que le nombre d'appels réalisé est de $\boxed{56}$. Ce qui réduit considérablement le temps de calculs par rapport à la solution récursive naïve précédente.

4 Programmation dynamique bottom-up

Q9. Compléter le code. Vérifier les résultats obtenus.

```
p=[0,1,5,8,9,10,17,17,20,24,30]
def couper_barre(p:[int],n:int)->int:
```

```
    r=[-1 for i in range(len(p))] # Initialisation du tableau qui contiendra la
        résolution du problème pour r[i]
    r[0]=0 # Pour r_0=0.
    for j in range(1,n+1):
        rev=-1 # Initialisation
        for i in range(1,j+1):
            rev=max(rev,p[i]+r[j-i])
        r[j]=rev
    return r[n]

>>>print(couper_barre(p,10))
30
```