CHAPITREIII
Programation dynamique

1 Rappels

1.1 Complexité

Pour résoudre un problème il existe toujours plusieurs algorithmes. Souvent le choix, d'un algorithme, est conditionné par les caractéristiques de l'objet étudier et aussi les capacités de la machine. On peut privilégier la stabilité sur la vitesse ou la consommation de la mémoire.
Pour commencer il faut qu'il soit stable : c'est à dire qu'il fonctionne sans trop d'erreur en évitant des opérations entre nombre trop différents, en particulier les divisions souvent très approximatives qui peuvent faire apparaitre des phénomènes de petits diviseurs. Le premier problème est d'ordre mathématique et donc ... compliqué.

Deux problèmes cruciaux nous intéressent :

- l'utilisation de la mémoire (éviter de consommer trop de mémoire) appelé complexité en espace (mais ce problème est plutôt technique, lié à la machine, et assez facile à évaluer bien que très caché dans le cas récursif). On retiendra qu'un nombre occupe 64 bits (8 octets)
- Le temps de calcul, complexité en temps.

On se propose dévaluer ce temps en indiquant l'ordre de grandeur, du nombre d'opérations nécessaires en fonction d'un entier n fixé correspondant à la taille de la donnée traitée ou de la précision attendue. Ca peut être le nombre de valeurs, par exemple n^2 pour une matrice carré de taille n quand l'algorithme est fini, ou lié à la précision du résultat $\frac{1}{n}$.
Évidement entre le coût théorique d'un algorithme et celui d'un programme, il y a une petite différence car les différents programmes prennent plus ou moins de temps pour effectuer certaines opérations. On se contentera de comptabiliser les opérations simples (addition division multiplication affectation de variable...). Souvent on obtient un ordre de grandeur du nombre minimal ou maximal d'opérations nécessaires. Ce qui nous amènera aussi à choisir un algorithme plutôt qu'un autre et à bien préciser sa performance.

Nous allons d'abord nous intéresser aux algorithmes finis (à distinguer des algorithmes itératifs donnant une valeur approchée). Pour noter cette complexité on utilise les grand O : $O(n)$ signifie que la grandeur est proportionnelle à n c'est à dire que $\frac{O(n)}{n}$ est bornée. Ils se manipulent comme les petits o des développements limités que vous aimez tant !

Techniquement, dans un programme en boucle (récursif) on obtient une relation de récurrence (liée au choix de l'algorithme) qui permet d'obtenir, grâce aux théorie sur les suites, notre ordre de grandeur (je sents que ça va vous plaire...). Quand on a un $O(1)$ on est en temps constant (instantané), en $O(\log n)$ ou $O(\ln(n))$ on est en logarithme donc très rapide, en $0(n)$ on est linéaire proportionnel à la taille n ce qui reste acceptable, $0(n^2)$ quadratique donc lent, $O(2^k)$ exponentiel donc trop lent !
Par exemple, si nous prenons la suite définie par récurrence :

$$\begin{cases} u_{n+1} = \frac{u_n^2}{2} \\ u_0 = 1 \end{cases}$$

Proposons un premier algorithme qui donne la somme :

$$S_n = \sum_{k=0}^{n} u_k.$$

```
    def u(n,u0):
         u=u0
         if n==0:
           return u
         for i in range(n):
            u=(u**2)/2
         return u
     def somme(n,u,u0):
           S=0
           for i in range(n+1):
                  S=S+u(i,u0)
            return S
```

La complexité $U(n)$ de la fonction $u(n)$ est un $O(n)$ et celui de la somme est $C(n) \leq \sum_{i\leq n} 1+U(i)$ ce qui donne $O(n^2)$... pas terrible. Une simple modification améliore le calcul :

```
    def u(n,u0):
         u=[None]*(n+1)
         u[0]=u_0
         if n==0:
           return u
         for i in range(n):
            u[i+1]=(u[i]**2)/2
         return u
     def somme(n,u,u0):
           S=0
           W=u(n,u0)
           for i in range(n+1):
                  S=S+W[i]
            return S
```

On a désormais un $O(n)$ en complexité en temps et ...un $O(n)$ en complexité en espace contre un $O(1)$ précédemment.

1.2 Fonctions récursives

La récursivité est un mode naturel de programmation d'un algorithme : très puissant et gourmand en mémoire.

Des algorithmes définis par récurrence, a priori compliqués à programmer deviennent très simple (comme on le verra avec les exemples qui suivent). Il consiste a appeler dans la fonction la même fonction appliquer sur des données de taille inférieur (c'est une récurrence descendante). L'ordinateur "descend" jusqu'à obtenir une valeur et remonte progressivement pour construire la réponse. Ainsi l'étude de la complexité s'obtient souvent par une relation de récurrence : c'est magique ! Attention suivant le choix de l'écriture on peut aussi dangereusement augmenter les calculs. On remarquera que le choix récursif ne permet pas d'accélérer un algorithme mais permet de le formaliser et de le mettre en place très simplement.

Par exemple une suite récurrente de type

$$u_{n+2} = u_n + u_{n+1} \text{ et } u_0 = u_1 = 1$$

s'écrit très naturellement :

```
>>> def u(n):
          if n==1:
            return 1
          if n==0:
            return 1
         else:
           return u(n-1)+u(n-2)
```

Attention ici on calcule deux fois $u(n-1)$ et $u(n-2)$ ce qui donne une complexité $C(n) \leq C(n-1)+C(n-2)+1$, soit un $O(2^n)$.... bof. Pour y remédier encore une liste :

```
>>> def u(n):
        if n==1:
          return [1,1]

        else:
          w=u(n-1)
          return [w[1],w[1]+w[0]]
```

Qui donne une complexité en $O(n)$ et retourne un vecteur de dimension 2 :

$$[u_{n-1}, u_n].$$

Le factoriel se programme aussi très facilement en récursif :

```
>>> def factoriel(n)
""" calcul factoriel n"""
      if n==0:
         return 1
      else:
         return n*factoriel(n-1)
```

Pour calculr `factoriel(n)` l'algorithme va chercher à obtenir `factoriel(n-1)` qui l'amène à chercher `factoriel(n-2)`.... jusqu'à tomber sur le calcul de factoriel de 1 et remonter `factoriel(2),...factoriel(1)`.

Jusque la .... rien d'exceptionnel. Mais on peut aussi bien utiliser des récurrences multiples en prenant des matrices comme indice et non plus seulement un entier n :

```
>>> def g(a)
     if len(a)==1:
          return 1
     else:
          b=a[0:len(a)-1]
          c=a[1,len(a)]     # la longueur des vecteurs diminue
                                # jusqu'a devenir de taille 1
          return g(b)+2*g(c)
```

Ici on calcule une valeur à partir de vecteurs de taille toujours plus petite.
Attention la complexité en temps peut être bien cacher, en reprenant l'exemple précédant :

```
   def u(n,u0):
        if n==0:
            return u0
       return (u(n-1,u0)**2)/2
    def somme(n,u0):
          if n==0:
             return u0
          return somme(n-1,u0)+u(n,u0)
```

Ici la complexité est $C(n) \leq C(n-1) + 1 + U(n) \leq C(n-1) + O(n)$ soit $C(n) = O(n^2)$. Par contre si on utilise intelligemment les listes :

```
    def u(n,u0):
        if n==0:
            return [u0]
        v=u[n-1,u0]
        w= v+[(v[n-1])**2)/2]
        return w
     def somme(n,u,u0):
          if n==0:
             return u0
           w=u(n,u0)
           S=0
           for i in range(n+1):
           S=S+w[i]
          return S
```

La complexité est alors un $O(n)$. Nous reviendront plus en détail sur les applications à l'étude de programme "classique" dans le chapitre suivant.

2 Algorithme Glouton

Un algorithme glouton est un algorithme qui construit une solution à un problème d'optimisation en faisant une succession de choix localement optimum. Le choix reste subjectif (intuitif) et n'est pas forcement optimum. Il renvoie souvent une solution presque optimum :

rendu de monnaie, chemin de poids minimum, ensemble disjoint de cardinal maximum, coloration de graphe....

Prenons l'exemple du rendu de monnaies :

On se donne un ensemble $\mathcal{P} = \{p_0, \dots, p_{n-1}\}$ (les valeurs des différentes pièces et billets) , avec $p_0 = 1 < p_1 < \cdots < p_{n-1}$, (ces valeurs étant entières). On se donne également une somme d'argent $s \in \mathbb{N}$ que l'on souhaite décomposer à l'aide des p_i, de manière à minimiser le nombre de pièces et billets.

On cherche donc une liste `[m0, . . . , m(n-1)]` d'entier vérifiant $\sum_{i=0}^{n-1} m_i p_i = s$, telle que la somme $\sum_i m_i$ soit minimale. *Comme $p_0 = 1$, le problème admet toujours une solution (et le nombre de pièces totales est inférieur à s).*

Le choix glouton consiste à choisir étape par étape une solution optimale :

- On commence par choisir le plus gros billet, puis on détermine le nombre maximum de fois ou on peut utiliser ce billet pour approcher au plus près s : On calcule le quotient de s par p_{n-1} et on pose s_1 le reste.
- On recommence avec p_{n-1} et le reste s_1
- Et ainsi de suite.

Voici donc une proposition de programme de type glouton adapté au problème du rendu de monnaie :

```
def rendu_monnaie(s,P):
""" P[0]=1, P trie dans l'ordre croissant."""
     i=len(P)-1
     L=[]
     while s>0:
           if P[i]<=s:
               L.append(P[i])
               s-=P[i]
          else:
                i-=1
     return L
```

Le résultat n'est pourtant pas toujours optimal, par exemple si $p_0 = 1, p_1 = 3, p_2 = 4$, pour $s = 6$ nous aurons $s = 2p_0 + p_2$ alors que $s = 2 \times p_1$ serait la solution optimale.

Exercice 2.1 *L'objectif est d'afficher un paragraphe de manière harmonieuse en le justifiant, c'est-à-dire en alignant les mots sur les bords gauche et droit de la zone d'écriture de la page. Le paragraphe à justifier est constitué d'une liste de mots (chaînes de caractères). La difficulté est de placer des espaces entre les mots et de découper la liste en sous-listes pour que les lignes soient équilibrées (pas de ligne avec beaucoup d'espaces à la fin ou entre les mots par exemple). Pour simplifier le problème, on considère que le paragraphe est constitué d'une liste `lmots` d'entiers correspondant aux longueurs de chaque mot du paragraphe dans l'ordre d'apparition des mots (`lmots[i]` correspond au nombre de caractères du mot d'indice `i`). On note `L` le nombre de caractères et espaces que peut contenir une ligne au maximum. Il faut au minimum un espace entre deux mots d'une même ligne, on suppose que cet espace minimal correspond à un caractère. Compléter*

le squellete suivant pour que la fonction `Arangertexte(lmots,L)` *renvoie la liste* `lignes` *du texte correctement arrangé, en proposant un algorithme de type glouton*

```
def Arangertexte(Lmots,L)
     nligne=[]
     lignes=[]
    l,n=0,len(Lmots)
    for i in range(n) :
         if (Lmots[i] + l) > L:
                  ........
                  ........
                  .........
        else:
                ............
                ..............
  lignes.append(nligne)
  return lignes
```

3 Programmation dynamique

Le problème avec les algorithmes glouton, c'est que bien qu'ils soient souvent rapide, ils ne donnent pas, en général, le meilleur résultat attendu et donc la solution optimale.

3.1 Principe de la mémmoisation

Prenons l'exemple des $\binom{n}{p}$ que l'on souhaite calculer à l'aide de la relation de Pascal :

$$\begin{cases} \forall n & \begin{pmatrix} n \\ 0 \end{pmatrix} = 1 \\ \forall n & \begin{pmatrix} n \\ n \end{pmatrix} = 1 \\ \forall p \leq n & \begin{pmatrix} n+1 \\ p \end{pmatrix} = \begin{pmatrix} n \\ p \end{pmatrix} + \begin{pmatrix} n \\ p-1 \end{pmatrix} \end{cases}$$

Une fonction récursive $binome(n,p)$, prenant pour paramètres n et p et retournant le nombre $\begin{pmatrix} n \\ p \end{pmatrix}$ peut être

```
def binome(n,p):
    if p==0     or p==n:
        return 1

     return binome(n-1,p-1)+binome(n-1,p)
```

Dans ce programme simple on répète plusieurs fois les mêmes calculs ; et la complexité est affolante (exponentielle), sans compter que la programmation récursive consomme beaucoup de mémoire.

On peut améliorer le programme en proposant une fonction $tablebinome(n)$ retournant la liste bin des listes des coefficients binomiaux, de telle sorte que $bin(n,p)$ soit le coefficient $\begin{pmatrix} n \\ p-1 \end{pmatrix}$. On parle de memoisation (on mémorise certains calculs pour ne pas les répéter) :

```
def tablebinome(n):
    if n==1:
        return [[1],[1,1]]
     bin=tablebinome(n-1)+[[None]*(n+1)]
    bin[n][0]=1
    for i in range(1,n):
          bin[n][i]=bin[n-1][i-1]+bin[n-1][i]
    bin[n][n]=1
    return  bin
```

Pour compléter ce tableau il faut $n*p$ opérations. Encore une fois ça fait beaucoup pour un simple calcul, une proposition "dynamique" serait, en conservant la même idée, de ne calculer que ce qui nous ait vraiment utile :

```
def binome(n,p,B):
    if n==0 or p==0 or n==p:
        B[n][p]=1
        return B[n][p]
    if B[n][p] !=0:
        return B[n][p]
    if B[n-1][p-1]!=0:
        if   B[n-1][p]!=0:
            B[n][p]=B[n-1][p-1]+B[n-1][p]
        else:
            B[n-1][p]=binome(n-1,p,B)
            B[n][p]=B[n-1][p-1]+B[n-1][p]
    else:
            B[n-1][p-1]=binome(n-1,p-1,B)
            if   B[n-1][p]!=0:
                 B[n][p]=B[n-1][p-1]+B[n-1][p]
            else:
                 B[n-1][p]=binome(n-1,p,B)
                 B[n][p]=B[n-1][p-1]+B[n-1][p]
    return B[n][p]
# pour utiliser le programme il faut initialiser B comme une liste de 0 bonne taille
def coeffbin(n,p):
    B=[[0 for k in range(p+1)] for i in range(n+1)]
    return binome(n,p,B)
```

Si on propose l'exécution et l'affichage de l'algorithme on verra qu'une partie seulement des valeurs du tableau ont été nécéssaire pour terminer le calcul :

```
    coeffbin(20,7)
Out[27]:
(77520,
 [[0, 0, 0, 0, 0, 0, 0, 0],
  [1, 1, 0, 0, 0, 0, 0, 0],
  [1, 2, 1, 0, 0, 0, 0, 0],
  [1, 3, 3, 1, 0, 0, 0, 0],
  [1, 4, 6, 4, 1, 0, 0, 0],
  [1, 5, 10, 10, 5, 1, 0, 0],
  [1, 6, 15, 20, 15, 6, 1, 0],
  [1, 7, 21, 35, 35, 21, 7, 1],
  [1, 8, 28, 56, 70, 56, 28, 8],
  [1, 9, 36, 84, 126, 126, 84, 36],
  [1, 10, 45, 120, 210, 252, 210, 120],
  [1, 11, 55, 165, 330, 462, 462, 330],
  [1, 12, 66, 220, 495, 792, 924, 792],
  [1, 13, 78, 286, 715, 1287, 1716, 1716],
  [0, 14, 91, 364, 1001, 2002, 3003, 3432],
  [0, 0, 105, 455, 1365, 3003, 5005, 6435],
  [0, 0, 0, 560, 1820, 4368, 8008, 11440],
  [0, 0, 0, 0, 2380, 6188, 12376, 19448],
  [0, 0, 0, 0, 0, 8568, 18564, 31824],
  [0, 0, 0, 0, 0, 0, 27132, 50388],
  [0, 0, 0, 0, 0, 0, 0, 77520]])
```

Exercice 3.1 *En appliquant le principe de la mémoisation, on se propose de programmer une fonction* `Rech_prof_rec(G,S,u)` *prenant comme paramètre le dictionnaire G de la liste d'adjacence du graphe G, un noeud u et le dictionnaire de la liste des noeuds reliés au noeud v, c-a-d que S[v] renvoie la liste des noeud relier à ce noeud v. La fonction* `Rech_prof_rec(G,S,u)` *retourne la liste des noeuds relier à u :*

```
def Rech_prof_rec(G,S,u):
    if u in S:
        return S[u]
    if len(u)==0:
        return []
    L=[] # liste des noeuds relies a u
    ....................................
     return L
```

3.2 Principes de la programmation dynamique

Le principe générale de la programmation dynamique repose sur les grandes lignes suivantes :
On se donne $f : \mathcal{U} \to \mathbb{Z}$ et on cherche à déterminer x pour que $f(x)$ soit un extremum de f (maximum ou minimum même combat).

— Décomposer en sous problèmes $\mathcal{U}_i \subset \mathcal{U}$, trouver les x_i nécéssaire, avec

$$f(x_i) = \max_{u \in \mathcal{U}_i}(f(u)).$$

Il faut évidement que le problème soit plus simple (Comme $\mathcal{U}_i$ est plus petit normalement ça se passe bien)
— Déterminer une relation de récurrence qui permet de trouver x à partir des x_i et des $\mathcal{U}_i$
— Définir, souvent de manière récursive la valeur optimal. Attention les $\mathcal{U}_i$ ne sont pas forcément disjoints, donc pour éviter de se répéter, on aura donc tendance à stocker les résultats dans un tableau (qui peut lui même être "dynamique", c-a-d évoluer autant en taille qu'n valeurs), on parle alors de mémoisation.
— Calculer les valeurs
— Construire la solution

Nous allons l'illustrer sur deux exemples :

3.3 Exemple, dynamique, du rendu de monnaie

Reprenons le cas du rendu de monnaie, et proposons une programmation de type dynamique.
Elle repose sur l'idée de chercher pour une somme s, la solution de sous problème :
On découpe en sous problème en s'intéressant au rendu de monnaie pour les sommes $s - p_1$, $s - p_2 \ldots s - p_n$, on cherche donc à optimiser le nombre de pièces pour toutes les valeurs inférieures à s possible $s - p_i$ puis de comparer. La construction s'effectue de façon récursive et comme le risque de répéter les calculs est important, nous allons les stocker dans un dictionnaire S (la mémoisation), contenant les $s+1$ couple $S[i] = [\sum_j m_j(i), [m_1(i), \ldots, m_n(i)]]$, où les $[m_1, \ldots, m_i]$ sont les solutions optimum du cas $s = i$, pour rendre la monnaie sur la somme i, il faut $n(i) = \sum_j m_j(i)$ billets, avec une décomposition de $m_j(i)$ billets de valeur p_j et ainsi de suite.
Le point de départ repose sur le fait que pour rendre la monnaie sur la somme p_i un seul billet de valeur p_i suffit et c'est la solution optimale.
Progressivement l'algorithme complète les cases utiles, c-a-d les `S[i]`, correspondant au résultat optimal :

Etapa 1 : Si s a déjà été traiter on renvoie la valeur :

```
if s in S:
        return S[s]
```

Etape 2 : Sinon $S[s]$ est initialiser comme non traité :

```
S[s]=[np.infty,[0]*n]
```

Etape 3 : On teste tous les billets possible, donc tous les $p_i \leq s$. Si $s = p_i$ le cas optimum est $S[s] = [1, [0, \ldots, 1, \ldots, 0]]$ (un billet de valeur p suffit pour rendre la monnaie)

```
if p==s:
            S[p]=[1,[0]*n]
            S[p][1][i]=1
```

Etapae 4 : Sinon, on recherche le cas optimum pour les $s1 = s - p_i$ (avec $p_i < s$, si la valeur n'est pas connue on la calcule récursivement et on la stocke dans S :

```
    if s1 not in S:
                    S[s1]=rendumonnaie(s-p,P,S)
```

Etape 5 : Ensuite on compare et on stocke la nouvelle valeur dans S :

```
            if S[s][0]> S[s1][0]+1:
                    S[s][0]=S[s1][0]+1
                    S[s][1]=S[s1][1].copy()
                    S[s][1][i]+=1
```

On a donc la relation de récurrence suivante :

$$\texttt{S[s]} = (n(s), m(s)) = \begin{cases} (n(s) = 1, m_i(s) = 1, \quad \forall k \neq i \quad m_k(s) = 0) & \text{si } s = p_i \\ (n(s) = \min_i(1 + n(s - p_i)), \quad m(s) = m(s - p_k) + 1) & \text{avec } k \text{ vérifiant } m(s - p_k) + 1 = n(s) \end{cases}$$

Une proposition de programme serait :

```
import numpy as np
def rendumonnaie(s,P,S):
    n=len(P)
    if s in S:
          return S[s]
    S[s]=[np.infty,[0]*n]

    for i in range(n):
             p=P[i]
             #print(S)
             if p==s:
                S[p]=[1,[0]*n]
                S[p][1][i]=1

                return S[p]
             if p< s:
                s1=s-p # pour la nouvelle somme s1-p on recherche la valeur optimale

               if s1 not in S: # si pas encore calculer on la determine recursivement
                    S[s1]=rendumonnaie(s-p,P,S)
               if S[s][0]> S[s1][0]+1: # la decomposition est meilleure
                    S[s][1]=S[s1][1].copy()# remplacer la decomposition
                    S[s][0]=S[s1][0]+1
                    S[s][1][i]+=1# un billet de plus de valeur p_i
    return S[s]
def renduM(s,P):
    n=len(P)
    S={0:[0,[0]*n]}
    return rendumonnaie(s,P,S)
```

Exercice 3.2 *On reprend l'exercice précédant :*
On introduit une mesure de la qualité de la ligne qui mesure le carré du nombre d'espace en trop sur une ligne composé des mots de i à j :

$$cout(i,j) = \left(L - (i-j) - \sum_{k=i}^{j} lmots[k]\right)^2$$

Nous disposons donc de la fonction :

```
  def cout(i,j,lmots,L):
        res=np.sum(lmots[i:j+1])+(j-i)
        if res>L:
            return float("inf")
       else:
           return (L-res)**2
```

Si on pose i_k les indices du dernier mots de chaque ligne, on cherche à minimise :

$$\sum_{k\geq 0} cout(i_k, i_{k+1})$$

et accessoirement déterminer le bon découpage.

1. *Pour commencer on proposera un programme dynamique renvoyant la valeur minimale $d(i)$ coût du placement optimum à partir du i ème mot, qui repose sur la relation de récurrence suivante :*

$$d(i) = \min_{i<j\leq i+L} (d(j) + cout(i, j-1))$$

On se donne le dictionnaire `D`, initialement donné par : `D={n:0}`, c-a-d $d(n) = 0$, on stockera pour la clé i la valeur $d(i)$. Compléter :

```
def progd_memo(i,lmots,L,D):
        if i in D:
            return D[i]
        D[i]=np.infty
        ....................
        return D[i]
 def Bellman(i,lmots,L):
        D={n:0}
        return progd_memo(0,lmots,L,D)
```

2. *Une petite amélioration, consisterait à renvoyer le découpage optimum et intuitivement commencer par la dernière ligne (programmation de bas en haut). `t`, représente un dictionnaire où `t[i]` renvoie le passage optimal à la ligne quand on commence au mot i, c-a-d si i est le premier mot de la ligne, $t[i]$ est celui de la ligne suivante dans le cas optimal.*

```
def progd_bashaut(lmots,L,t):
      M=[0]*(len(lmots)+1)
      for i in range(len(lmots)-1,-1,-1):
            mini,indi=float("inf"),-1
            ............................
           t[i]=indi
           M[i]=mini
       return M[-1]
```

3. *Donner un ordre de grandeur de la complexité de ces deux algorithmes.*