TP 3

Dans ce TP nous allons travailler la programmation dynamique, illustrée par trois algorithmes classiques :
`Le problème du sac à dos-La plus longue sous suite croissante- Algorithme de Floyd Warshall.`

1 Echauffement le sac à dos

En adaptant le programme sur le rendu de monnaie, nous allons nous interesser au problème dit du sac à dos.
Soit E un ensemble de n éléments à ranger dans un sac à dos de capacité maximale W (son volume) ; chaque élément e a une valeur v (on les numérote de 1 à n), chaque élément a un poids w (on les numérote de 1 à n).

On cherche à trouver un sous-ensemble $F \subset E$ tel que :

— La somme des poids des éléments dans F ne dépasse pas la capacité W.

— La somme des valeurs des éléments de F est maximale ($\sum_{e \in F} v(e)$).

On souhaite donc une valeur maximum sans dépasser le poids maximum autorisé.

La programmation dynamique permet de scinder les entrées du problème en autant de sous-ensemble que nécessaire. Le problème est résolu pour chacun des sous-ensembles en utilisant les solutions précédentes pour calculer la valeur du sous-ensemble courant. Il faut cependant qu'il n'y ait qu'un nombre polynomial de sous-problèmes.

`w` et `u` deux dictionnaires contenant , le poids et les valeurs de chaque objet.

`Opt(i, j)` la valeur optimal de poids maximum j constituer à partir des éléments de 1 à i.

Nous pouvons établir la relation de récurrence.

$$\begin{cases} \texttt{Opt(i,j)} = 0 & \text{si } i = 0 \\ \texttt{Opt(i,j)=Opt(i-1,j)} & \text{si } w_i > j \text{ et } j \leq W \\ \texttt{Opt(i,j)} = \max(\texttt{Opt(i-1,j)}, v_i + \texttt{Opt}(i-1, j-w_i) & \text{sinon} \end{cases}$$

Exercice 1.1 *1. Interpréter cette relation de récurrence*

2. On propose une programmation de type dynamique, qui va calculer la solution aux sous-problèmes une seule fois et le mémoriser dans une table pour que ce calcul puisse être réutilisé, en adaptant l'algorithme suivant :

Etape 1 : *On construit un tableau* `V` *de* $n+1$ *lignes et* $Poids + 1$ *colonnes. Pour* $1 \leq i \leq n$ *et* $0 \leq j \leq Poids$, *la case* `V[i][ j]` *va mémoriser la valeur maximale des sous-ensembles de taille au plus* i. *Si on arrive à calculer toutes les cases de ce tableau, la case* `V[n ][ Poids]` *contiendra la valeur maximale des éléments qui peuvent être rangés dans le sac à dos, i.e. la solution à notre problème.*

Etape 2 : *On construit par récurrence la valeur d'une solution optimale en termes de solutions à des sous-problèmes :*

Initialisation : *On pose :* `V[0, j] = 0` *pour*$0 \leq j \leq poids$ *(on n'a pas gardé d'élément)* `V[i][ j]` $= -\infty$ *pour* $j < 0$ *(entrée interdite)*

Récursion : *On utilise pour* $1 \leq i \leq n$, $0 \leq j \leq W$

$$\texttt{V[i][j]} = \max(\texttt{V[i - 1][j],v\_i + V[i-1][j -w\_i)}$$

Etape 3 : *On calcule les valeurs de* `V[i][ j]` *de façon itérative en partant de l'initialisation de* `V` *(cf. étape la valeur optimale.*

Programmer une fonction `sacados(v,w,Poids)` *prenant comme paramètre deux listes* `v` *et* `w`*, respectivement, la liste des valeurs des objets et elle des poids,* `Poids` *étant le poids maximum autorisé. La fonction devra renvoyer la valeur maximum autorisée.*

```
def sacados(v,w,n,Poids):
.     """ v la liste des valeurs des objets,w celle des poids, Poids le poids
      n=len(v)
      V=[x[:] for x in [[0]*(Poids+1)] * (n+1)] # intialisation de V
      # ou aussi V=np.zeros((n+1,Poids))
      for i in range(1,n+1):
          for j in range(Poids+1):
              if w[i]<=j: # on ajoute l'element i
                      V[i][j]=max(V[i-1][j],v[i]+V[i-1][j-w[i]])
               else: # l'element i n'est pas ajoute
                      V[i][j]=V[i-1][j]
      return V[n][Poids]
```

3. *Modifier le programme afin qu'il renvoie le contenu du sac à dos :*

```
def sad(v,w,Poids):
       n=len(n)
      V=[x[:] for x in [[0]*(Poids+1)] * (n+1)]
      memo=[x[:] for x in [[0]*(Poids+1)] * (n+1)]
      # Mais aussi
      #V,memo=np.zeros((n+1,Poids)),np.zeros((n+1,Poids))
      elements=[]
              for i in range(1,n+1):
                      for j in range(Poids+1):
                              if ((w[i]<= j) and (v[i]+V[i-1][j-w[i]]>V[i-1][j]
                                      V[i][j]=v[i]+V[i-1][j-w[i]]
                                      memo[i][j]=1
                              else:
                                      V[i][j]=V[i-1][j]
                                      memo[i][j]=0
        K=Poids
              for i in range(n,0,-1):
                      if memo[i][K]==1:
                      elements.append(i)
                      K=K-w[i-1]
       return (V[n][Poids],elements,Matrix(memo),Matrix(V))
```

2 La plus longue sous suite croissante

La recherche d'une plus longue sous-suite strictement croissante dans une suite finie est un problème classique en algorithmique. Ce problème peut être résolu en temps $O(n \log n)$ avec n la longueur de la suite.

L'entrée du problème est une suite finie $x_1, \ldots, x_n$. L'objectif est de trouver une sous-suite strictement croissante de la suite , pour la plus grande longueur L possible, c-a-d compsée du plus grand nombre possible de termes .

Par exemple, la suite $(6, 1, 4, 9, 5, 11)$ possède des sous-suites strictement croissantes de longueur 4, mais aucune de longueur 5. Une plus longue sous- suite strictement croissante est $(1, 4, 9, 11)$, obtenue en prenant les éléments en position $2, 3, 4$ et 6 de la suite initiale. En général, la solution n'est pas unique. Ici, une autre solution est $(1, 4, 5, 11)$.

Le principe géneral est le suivant :

On recherche une plus grande sous suite des n premier termes $Xn = [X_1, \ldots, X_n]$, à partir de celles extraites de $X(n-1) = [X_1, \ldots X_n]$ (notre sous problème), en s'interessant aux sous suites de $X(n-1)$ dont le dernier

terme est le plus petit possible. On cherche parmi ces sous suites, la plus grande des sous suites auxquelles on peut ajouter le terme X_n ou non et construire ainsi une nouvelles sous suite croissante dont le dernier terme est minimal.

Exercice 2.1 *Dans la suite* `M,X,P` *sont des listes* `L` *un entier.*

1. *Soit* `X` *une liste de* n *entiers,* `M` *une liste de* $L+1 \leq n+1$ *entiers distincts tel que* `X[M[1]],...X[M[L]]` *définissent une suite croissante. Programmer une fonction* `recherche_dich(X,M,i)` *renvoyant le plus grand entier* $1 \leq j \leq L$ *vérifiant* $X[M[j]] < X[i-1]$*, s'il existe,* 0 *sinon, en utilisant le principe de recherche dichotomique dans un tableau trié.*

```
def recherche_dich(X,M,i):
    """ X et M des liste X[M[1]],...,X[M[-1]]croissante"""
    assert M[0]==0 and i<len(X)+1 and len(M)<=len(X)+1
    L=len(M)-1
    s=1
    if  X[M[s]]>=X[i-1]:
        return 0
    if X[M[L]]<X[i-1]:
        return L
    else:
        while s<L-1:

            ..............
            ...........
    return s
```

```
def recherche_dich(X,M,i):
    """ X et M des liste X[M[1]],...,X[M[-1]]croissante"""
    assert M[0]==0 and i<len(X)+1 and len(M)<=len(X)+1
    L=len(M)-1
    print(L)

    s=1
    if  X[M[s]]>=X[i-1]:
        return 0
    if X[M[L]]<X[i-1]:
        return L


     while s<L-1:

            if X[M[L]]<X[i-1]:
                return L
            j=int((s+L)//2)

            if X[M[j]]<X[i-1]:
                s=j
            else:
                L=j
    return s
```

2. *Nous allons construire un algorithme dynamique, en recherchant les solutions au sous problème associé à la sous suite* $X_1, \ldots, X_j$.
   `X` *la liste de* n *entiers,* `M=[0]*(n+1)` *et* `P=[0]*n` *deux listes qui seront compléter pour contenir pour* `M[i]`*, l'indice* k *tel que* `X[k]` *soit la plus petite valeur possible du dernier élément d'une sous suite strictement croissante d'exactement* i *éléments ;* `P[k]` *l'indice du terme précédant de cette sous suite optimale.*

Soit L le plus petit entier vérifiant `M[L]==n`. *Justifier que la liste* `X[M[1]],...,X[M[L]` *est croissante et que les* `M[i]` *sont distincts pour $i \in \{1,\dots,L\}$.*

3. *La relation de recurrence sur* `M` *et* `P` *est la suivante :*
   - *Initialement en recherche* `M[1]`*, l'indice le plus petit du minimum des* `X[i]`
   - *L le nombre de terme de la plus grande sous suite croissante connue, initialement L est nulle*
   - *Pour chaque i de 1 à n on recherche le plus grand $j \in [|1, L|]$ tel que X[M[j]]<X[i-1]*
   - *S'il existe P[i-1]=M[j] et M[j+1]=i*
   - *L=max(L,j+1)*

   Programmer la fonction renvoyant L

```
def PGSSDynamique(X,M,P):
    L=0
    m=X[0]
    n=len(X)
    j=0
    .......
    ........  # determiner le plus petit indice d'un minimum des X[i]
    .........
    M[1]=j
    L=1

    for i in range(1,n+1):

        j=recherche_dich(X,M[:L+1],i)
        ..............
    return L
def Plusgrandesoussuite(X):
    n=len(X)
    M=[0]*(n+1)
    P=[0]*n
   return PGSSDynamique(X,M,P)
```

```
def PGSSDynamique(X,M,P):
    L=0
    m=X[0]
    n=len(X)
    j=0
    for i in range(n):
        if X[i]<m:
            j,m=i,X[i]
    M[1]=j
    L=1

    for i in range(1,n+1):

        j=recherche_dich(X,M[:L+1],i)

        if j!=0:
            P[i-1],M[j+1]=M[j],i-1
        L=max(L,j+1)

    return L
```

4. *Programmer la fonction renvoyant* `L` *et une sous suite de taille* `L` *croissante maximale (on utilisera* `P`*) pour la reconstituer.*

```
def Plusgrandesoussuite(X):
    n=len(X)
    M=[0]*n
    P=[0]*n
    L=PGSSDynamique(X,M,P)
    j=M[L]
    u=[X[j]]
.................
.................
    return u
```

```
def Plusgrandesoussuite(X):
    n=len(X)
    M=[0]*(n+1)
    P=[0]*n
    L=PGSSDynamique(X,M,P)

    j=M[L]
    u=[X[j]]

    while j>0:
        j=P[j]
        u.append(X[j])
    u.reverse()
    return u
```

5. *Nous allons finalement adapter ce programmer pour rechercher la plus grande sous sécance commune entre deux chaines (ici deux suites numérique)*

   Le problème de la plus longue sous-suite commune à deux suites `S[0], S[2],..., S[n-1]` *et* `T[0], T[2],..., T[m-1]` *peut être réduit au problème de la plus longue sous-suite croissante.*

   Pour cela, on note `A[x]` *la liste des indices des éléments de S valant x par ordre décroissant. Si i[1], i[2], ..., i[k] est une plus longue sous-suite strictement croissante de la suite obtenue en concaténant* `A[T[0]], ..., A[T[m-1]]`*, alors* `S[i[0]], ..., S[i[k-1]]` *est une plus longue sous-suite commune à* `S` *et* `T`*. La taille de la suite obtenue par concaténation est au plus nm, mais seulement m si la première suite ne contient pas d'élément en double. Ainsi, la réduction donne une méthode de résolution du problème de la plus longue sous-suite commune relativement efficace dans des cas particuliers courants . En appelant les programme précédant, proposer une fonction prenant comme paramètre deux liste et renvoyant une des plus grande sous sécances commune aux deux suites. Les listes* `S` *et* `T` *seront des liste d'entiers par forcément distincts.*

```
def Soussecance(S,T):
    A={}
    for k in len(S):
        if S[k] in A:
            A[k]=S[k]+A[k]
        else:
            A[k]=[S[k]]
    I=[]
    for u in T:
        if u in A:
            I+=A[u]
    i= Plusgrandesoussuite(I)
    return [S[j] for j in i]
```

3 Pour les $\frac{5}{2}$: *Correction à l'aide de l'algorithme de Viterbi, d'après Mines Pont 2024*

Un message " d'origine" , codé sur K caractères assimilés pour l'exercice à des nombres entiers de 0 à $K-1$, de longueur N, sera représenter comme une liste de longueur N composé de N entiers entre 0 et $K-1$. Ce message est transmit et modifier par des erreurs, ce nouveau message sera le message observé, qui sera toujours une liste de longueur N d'entiers entre 0 et $K-1$. Le but est de retrouver le message d'origine en corrigeant (le mieux possible) les erreurs.

Exercice 3.1 *On connait en partie le risque d'erreur stocké dans deux tableaux ou matrices de probabilité de taille $K\times K$, `P` et `E`. $E_{i,j}$ est la probabilité d'observer le symbol i sachant que le symbol j a été émis. $P_{i,j}$ est la probabilité que le symbole j soit présent dans le message initial sachant que i le précède dans ce même message (qui vient du fait que de la langue française par exemple certaines syllabes sont plus probable que d'autres).*

On se donne un message observe `obs` de N entiers `obs[i]`$\in [|0,K-1|]$ pour $i\in[|0,K-1|]$.

1. *On propose une première approche Gloutone.*

   On définit alors un graphe G orienté et pondéré, représentant tous les message initiaux, possible (connaissant le message observé `obs`). Le graphe G est constitué d'un premier noeud initiale, une racine arbitraire r, relier à K noeuds $S_{0,j}$ indiquant les symboles j possible du premier symbole du message intial. Chacune des arrêtes est pondérée par $\frac{1}{K}$. Les noeuds $S_{i,j}$, suivant, indiqueront que le i ème symbol du message initiale est la lettre j.

   On construit finalement le graphe de tel sorte que :

   Chaque noeuds $S_{i,j}$ admetent comme noeuds adjacents supérieur les $K-1$ noeuds $S_{i+1,k}$, pondéré par $P_{j,k}E_{obs[i],k}$ (probabilité que le i ème symbole soit k sachant que le précédant était j et que celui observé est `obs[i]`).

   Un chemin sera donc un message intial possible et le produit des pondérations la probabilité qu'il soit effectivement ce message.

   (a) *Programmer une fonction `Graphe(obs,P,E)` renvoyant le dictionnaire de la liste d'adjacence du graphe G.*

```
def Graphe(obs,P,E):
    G={}
    N=len(obs)
    K=len(P)
    G[r]=[ [(0,i),1/K] for i in range(K)]
    for i in range(N):
        for k in range(K):
            G[(i,k)]=[]
            for p in range(K):
                pond=P[k,p]*E[obs[i+1,p]
                G[(i,k)].append([(i+1,p),pond])
    return G
```

   (b) *On testera le programme dans le cas $K=3$, $N=8$ et*

$$P=\begin{pmatrix}0.3 & 0.2 & 0.5\\ 0.4 & 0.4 & 0.2\\ 0.2 & 0.3 & 0.5\end{pmatrix},\quad E=\begin{pmatrix}0.7 & 0.2 & 0.3\\ 0.2 & 0.7 & 0.1\\ 0.1 & 0.1 & 0.6\end{pmatrix}$$

   (c) *Proposer une fonction `glouton(G,obs)` renvoyant le message initiale en utilisant un algorithme glouton. On rappelle qu'un algorithme glouton consiste à chaque étape de faire le choix localement optimal. Ainsi si on se trouve au sommet $S_{i,j}$ on choisira l'arrête la plus probable.*

```
def Glouton(G,obs):
    N=len(obs)
    message=[None]*N
```

```
        A=G[r]
        Max=0
        v=(0,0)
        for u in A:
                if u[1]>Max:
                        v=u[0]
        message[0]=v[1]
        for i in range(1,N):
              A=G[v]
              Max=0
              v=(i,0)
              for u in A:
                    if u[1]>Max:
                        v=u[0]
               message[i]=v[1]
        return message
```

2. *On remarquera que l'algorithme consiste à trouver un chemin optimum, qui pourrait se résoudre avec l'algorithme Dijstra. On propose ici une approche dynamique.*
   On pose T la matrice des $T_{i,j}$ la valeur de probabilité maximum entre la racine et le noeud $S_{i,j}$:

$$\begin{cases} T_{i,j} = \max_{k\in[|0,K-1|]}\left(T_{k,j-1}\times P_{k,i}\times E_{obs[j],i}\right) & si \quad N-1\geq j>0 \\ T_{i,0} = E_{obs[0],i} \end{cases}$$

   Compléter le programme suivant :

```
def Viterbi(obs,P,E,K,N):
      T=[[0 for j in range(N)] for i in range(K)]
      for i in range(K):
            T[i][0]=E[obs[0]][i]


       ..........................
       return T
```

3. *Proposer une fonction renvoyant le message initial.*

4 L'algorithme de Floyd Warshall.

En informatique, l'algorithme de Floyd-Warshall est un algorithme pour déterminer les distances des plus courts chemins entre toutes les paires de sommets dans un graphe orienté et pondéré, en temps cubique au nombre de sommets.

L'algorithme de Floyd-Warshall prend en entrée un graphe orienté et valué, décrit par une matrice d'adjacence donnant le poids d'un arc lorsqu'il existe et la valeur $+\infty$ sinon. Le poids d'un chemin entre deux sommets est la somme des poids sur les arcs constituant ce chemin. Les arcs du graphe peuvent avoir des poids négatifs, mais le graphe ne doit pas posséder de cycle de poids strictement négatif. L'algorithme calcule, pour chaque paire de sommets, le poids minimal parmi tous les chemins entre ces deux sommets.

On suppose que les sommets de G sont $\{1,2,3,4,...,n\}$. Il résout successivement les sous-problèmes suivants :

$W^k_{i,j}$ est le poids minimal d'un chemin du sommet i au sommet j n'empruntant que des sommets intermédiaires dans $\{1,2,3,...,k\}$ s'il en existe un, et $+\infty$ sinon. On note W^k le tableau des $W^k_{i,j}$. Pour $k=0$, W^0 est la matrice d'adjacence définissant G. Maintenant, pour trouver une relation de récurrence, on considère un chemin p entre i et j de poids minimal dont les sommets intermédiaires sont dans $\{1,2,3,...,k\}$. De deux choses l'une :

— soit p n'emprunte pas le sommet k ;

— soit p emprunte exactement une fois le sommet k (car les circuits sont de poids positifs ou nuls) et p est donc la concaténation de deux chemins, entre i et k et k et j respectivement, dont les sommets intermédiaires sont dans $\{1, 2, 3, ..., k-1\}$.

L'observation ci-dessus donne la relation de récurrence :

$$W_{i,j}^k = \min_{(i,j)\in[|1,n|]^2}(W_{i,j}^{k-1}, W_{i,k}^{k-1} + W_{k,j}^{k-1})$$

pour tous i, j et$k$ dans $\{1, 2, 3, 4..., n\}$. Ainsi on résout les sous-problèmes par valeur de k croissante.

Exercice 4.1 *On supposera que le graphe G, nous est donné sous la forme d'une liste d'ajacence stocker dans un dictionnaire* `G`*, où* `G[i]` *renvoie la liste des uplets* (v_j, p_j) *de telle sorte que* p_j *soit le poids (relatif de l'arrête de* u_i *vers* v_j*.*

1. *Programmer une fonction* `Adj(G)` *retournant sous forme de tableau, la matrice d'ajacence évoquée ci-dessus et prenant comme paramètre le dictionnaire* `G` *de la liste d'ajacence du graphe (on prendra garde au fait que la ligne* $i = 0$ *est la première ligne).*
2. *Proposez une fonction* `cycle(A)`*, renvoyant true si la matrice d'ajdacence n'a aucun cocycle de longueur* 1 *de poids négatif et* `False` *sinon.*
3. *Programmer une fonction* `FW(A,k)` *prenant comme paramètre le tableau A de la matrice d'adjacence et renvoyant* W^k *et s'arrête si le graphe pocède un cocycle de poids négatif.*
4. *Terminer le programme de la fonction* `FloydWarshall(G)` *renvoyant la distance du plus cours chemin.*