Programmation dynamique

I. Planning de TD

1. Si l'on pouvait choisir librement toutes les épreuves, on pourrait passer toutes les épreuves difficiles pour un score de $7+2+4+6=19$ points. Une solution qui respecte la contrainte (**ii**) consiste à passer les exercices difficiles des TDs pairs et faciles sinon pour un score de $7+1+4+5=17$. Pour obtenir un score supérieur, il est nécessaire de passer sur au moins trois exercices difficiles lors des 4 TDs, ce qui contredirait (**ii**). Par conséquent,

   L'expérience maximale que l'on peut obtenir lorsque `f = [2;1;3;5]` et `d = [7;2;4;6]` est égale à 17.

   A noter qu'il n'y a pas unicité du planning permettant d'obtenir cette expérience, puisqu'on peut également passer les épreuves difficiles des TDs 1 et 4 uniquement pour aboutir au même score.

2. (a) Il suffit de parcourir le tableau `p` en mettant à jour le total des points d'expérience engrengés. On récupère ces points soit dans la liste `f`, soit dans la liste `d` en fonction de la valeur de `p[i]` qui indique quel sujet a été travaillé.

```
def score(p,f,d):
    s=0
    for i in range(len(p)):
        if p[i]:
            s+=d[i]
        else:
            s+=f[i]
    return s
```

   (b) Pour cette fonction, on parcourt à nouveau le tableau en vérifiant qu'il n'y a pas deux valeurs consécutives égales à `True`, ce qui signiferait que le planning prévoit deux exercices difficiles successifs, ce qui équivaut à contredire (**ii**).

```
def verifie(p):
    for i in range(len(p)-1):
        if p[i] and p[i+1]:
            return False
    return True
```

3. (a) S'il n'y a qu'un seul TD (le numéro 0), on peut librement choisir l'exercice facile ou difficile. Il y a donc deux plannings possibles. Pour deux TDs, le seul planning qui ne respecte pas (**ii**) est celui qui consiste à choisir les deux exercices difficiles. Il reste donc 3 plannings compatibles avec cette condition. Enfin, pour trois TDs, il y a $2^3=8$ plannings au total (sans condition). Les plannings incompatibles sont ceux où on prend les 3 exercices difficiles, deux exercices dificilles puis un facile, ou inversement un facile suivi de deux difficiles. Il reste alors 5 plannings compatibles.

$$\boxed{u_1=2 \qquad u_2=3 \qquad u_3=5}$$

   (b) Soit n un entier supérieur ou égal à 1. Pour créer un planning compatible sur $n+2$ TDs, on distingue deux cas :

   - On prend le sujet difficile au TD 0. On doit alors prendre le sujet facile au TD 1. Il reste alors n TDs à plannifier de façon à respecter (**ii**), et on a u_n façons de le faire.
   - On prend le sujet facile au TD 0 ce qui n'a aucune incidence sur les choix suivants. Il reste donc cette fois $n+1$ TDs à plannifier de façon à respecter (**ii**), et on a u_{n+1} façons de le faire.

   Finalement,
$$\boxed{\forall n\in\mathbb{N}, \qquad u_{n+2}=u_{n+1}+u_n}$$

   (c) La relation de récurrence précédente est linéaire d'ordre 2 à coefficients constants (c'est précisément celle de la suite de Fibonacci). Si l'on note $\lambda=(1+\sqrt{5})/2$ et $\mu=(1-\sqrt{5})/2$ les deux racines de l'équation caractéristique de la récurrence, on peut trouver deux réels α et β tels que

$$\forall n\in\mathbb{N}, \qquad u_n=\alpha\,\lambda^n+\beta\,\mu^n \underset{n\to+\infty}{\sim} \alpha\,\lambda^n$$

   Calculer le score d'un planning donné est de complexité linéaire (voir question 2.(a)). La méthode proposée est donc de complexité en $O(n\,\lambda^n)$ donc exponentielle. En théorie, cette méthode naïve est donc inexploitable pour

de grandes valeurs de n. Cependant, si l'on prend l'exemple de la PC*, il n'y a guère qu'une trentaine de séances de TDs dans l'année (22 à 23 semaines avant les écrits, puis 6-7 pendant la préparation aux oraux). Pour $n = 30$, $n\alpha^{30}$ est de l'ordre de $10^7 - 10^8$. Sachant que les machines actuelles sont capables d'effectuer de l'ordre de 10^{11} calculs à la seconde, le temps de calcul prendrait vraisemblablement un temps largement inférieur à une seconde, y compris avec cette méthode. Pour conclure,

> Même si la méthode naïve proposée est de complexité épouvantablement mauvaise, tant que le nombre de TDs reste raisonnable (de l'ordre de 3 à 4 dizaines), elle suffit pour répondre au problème avec les machines actuelles.

4. (a) Il s'agit essentiellement de respecter la condition (**ii**). Si lors du TD $n-2$ on a présenté un exercice difficile, on n'a pas le choix et il faut présenter l'exercice facile lors du dernier TD. Si en revanche on a présenté l'exercice facile, on n'a aucune contrainte pour le dernier TD, et on présentera alors celui qui donne le plus d'expérience (vraisemblablement l'exercice difficile, même si l'énoncé ne fait aucune hypothèse de comparaison entre les (f_i) et les (d_i)).

   (b) On peut proposer par exemple

$$\texttt{f = [1;1;1]} \qquad \text{et} \qquad \texttt{d = [1;10;100]}$$

Puisque $10 + 1 > 1 + 1$, l'algorithme propose de présenter l'exercice facile, puis l'exercice difficile. On sera alors contraint de prendre l'exercice facile au dernier TD pour un score total de $1+10+1 = 12$. Alors que si on prend les deux exercices faciles, puis l'exercice difficile, on termine avec un score de $1+1+100 = 102$ strictement supérieur.

> L'algorithme proposé ne fournit pas nécessairement un planning optimal.

5. Soit $j \in [\![1; n-2]\!]$. Considérons l'ensemble des plannings possibles sur les $j+2$ premiers TDs (du TD 0 au TD $j+1$) et qui respectent (**ii**). On sépare ces plannings en deux sous-ensembles :

   - Ceux pour lesquels on présente l'exercice difficile au TD $j+1$. Pour ces plannings, on présente nécessairement l'exercice facile au TD j et un planning respectant (**ii**) sur les TDs précédents (du TD 0 jusqu'au TD $j-1$). Le score maximal envisageable est donc égal à $m_{j-1} + f_j + d_{j+1}$.
   - Ceux pour lesquels on présente l'exercice facile au TD $j+1$. Pour ces plannings, on a pour seule contrainte un planning respectant (**ii**) sur les TDs précédents (du TD 0 jusqu'au TD j). Le score maximal envisageable est donc égal cette fois à $m_j + f_{j+1}$.

   Le score maximal sur les $j+2$ premiers TDs est donc obtenu en prenant le maximum de ces deux valeurs. En d'autres termes,

$$\forall j \in [\![1; n-2]\!], \qquad m_{j+1} = \max\{m_j + f_{j+1}, m_{j-1} + f_j + d_{j+1}\}$$

6. Il suffit d'implémenter le calcul de la suite récurrente précédente. Un tableau `m` est utilisé pour stocker les résultats au fur et à mesure. A noter qu'il faut deux cas particulier pour les deux premières valeurs, car la formule de récurrence précédente ne permet que le calcul de m_2 à partir du m_1 et m_0 (sauf à prendre la convention $m_{-1} = 0$).

```
def score_max(f,d):
    n=len(f)
    m=[0 for i in range(n)]
    m[0]=max(f[0],d[0])
    m[1]=max(m[0]+f[1],f[0]+d[1])
    for j in range(1,n-1):
        m[j+1]=max(m[j]+f[j+1],m[j-1]+f[j]+d[j+1])
    return m[n-1]
```

   On pourrait envisager une fonction purement récursive à condition d'utiliser le principe de mémoïzation, mais c'est un peu plus lourd.

7. Pour cette question, pour tout entier $i \in [\![0; n-1]\!]$, on définit un boléen c_i qui vaut `True` s'il faut absolument choisir un exercice difficile au TD i pour maximiser le score sur les TDs de 0 à i et `False` sinon. La formule de récurrence de la question 5 s'adapte pour la suite $(c_i)_{i\in[\![0;n-1]\!]}$ de la façon suivante :

$$\forall j \in [\![1; n-1]\!], \qquad c_{j+1} = \begin{cases} \texttt{True} & \text{si } m_{j+1} = m_{j-1} + f_j + d_{j+1} \\ \texttt{False} & \text{sinon} \end{cases}$$

   Calculer la suite $(c_i)_{i\in[\![0;n-1]\!]}$ se fait alors en parallèle de celui des $(m_i)_{i\in[\![0;n-1]\!]}$.

Remarque 1

En cas d'égalité entre $m_{j-1}+f_j+d_{j+1}$ et m_j+f_{j+1}, on peut librement choisir entre l'exercice facile et l'exercice difficile au TD $j+1$ pour maximiser le score (il n'y a pas unicité d'un planning optimal). Ici, on a choisit de privilégier dans tous les cas l'exercice difficile lorsque c'est possible.

```
def calcul_ci(f,d):
    n=len(f)
    m=[0 for i in range(n)]
    c=[False for i in range(n)]
    m[0]=max(f[0],d[0])
    if m[0]==d[0]:
        c[0]=True
    m[1]=max(m[0]+f[1],f[0]+d[1])
    if m[1]==f[0]+d[1]:
        c[1]=True
    for j in range(1,n-1):
        m[j+1]=max(m[j]+f[j+1],m[j-1]+f[j]+d[j+1])
        if m[j+1]==m[j-1]+f[j]+d[j+1]:
            c[j+1]=True
    return c
```

Une fois ces valeurs calculées, on construit le planning par récurrence décroissante : pour tout entier $k \geq n-1$, une fois choisis $(p_{k+1},\dots,p_{n-1})$, on consulte la valeur c_k pour savoir s'il faut prendre l'épreuve difficile au TD k. Si oui, on prend l'épreuve facile au TD $k-1$ et on recommence le procédé à partir du rang $k-2$. Sinon, on prend l'épreuve facile au TD k et on recommence le procédé à partir du rang $k-1$.

```
def planning_optimal(f,d):
    c=calcul_ci(f,d)
    k = len(f)-1; p=[]
    while k>=0:
        if c[k] and k>0:
            p.append(True); p.append(False); k-=2
        elif c[k]:
            p.append(True); k-=1
        else:
            p.append(False); k-=1
    p.reverse()
    return p
```

II. Plus long chemin dans un graphe ordonné

1. On peut par exemple proposer le chemin suivant, de longueur 6 :

$$\boxed{\text{Le chemin } (0,1,2,5,6,7) \text{ est un plus long chemin du sommet 0 vers le sommet 7.}}$$

2. Comme suggéré dans l'énoncé, on définit le prédicat :

$$\mathcal{P}(i) : \text{« Il existe un chemin depuis le sommet 0 vers le sommet } i. \text{ »}$$

Justifions que $\mathcal{P}(i)$ est vrai pour tout $i \in [\![0; n-1]\!]$ en raisonnant par récurrence forte sur i.

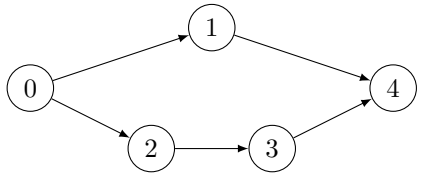
- $\mathcal{P}(0)$ est vraie puisque la suite (0) réduite à un élément est un chemin de 0 vers lui-même.
- Soit $i \leq n-1$. Supposons $\mathcal{P}(k)$ vraie pour tout $k < i$. Par hypothèse sur le graphe, le sommet i admet un arc entrant depuis un sommet $j < i$. Par hypothèse de récurrence, il existe un chemin depuis le sommet 0 jusqu'au sommet j. En ajoutant i à ce chemin, on obtient bien un chemin depuis 0 jusqu'à i, ce qui prouve $\mathcal{P}(i)$.

Par principe de récurrence, $\mathcal{P}(i)$ est vraie pour tout entier i ce qui prouve que

$$\boxed{\text{Pour tout } i \in [\![0; n-1]\!], \text{ il existe un chemin depuis le sommet 0 jusqu'au sommet } i \text{ du graphe.}}$$

On montre de même qu'il existe pour tout sommet i du graphe un chemin depuis i jusqu'au sommet $n-1$ en raisonnant cette fois par récurrence décroissante sur i. On utilise cette fois l'hypothèse assurant que tout sommet i du graphe possède un arc sortant vers un sommet d'indice $j > i$.

3. On peut proposer l'exemple suivant :



Sur cet exemple, l'algorithme proposé renvoie le chemin $(0,1,4)$ car 1 est le voisin de 0 d'incide minimal. Pourtant, le chemin $(0,2,3,4)$ est strictement plus long.

$$\boxed{\text{L'algorithme proposé ne fournit pas nécessairement un plus long chemin.}}$$

4. Par définition, la la représentation du graphe de l'énoncé par listes d'ajacences est la liste L suivante :

```
L = [[1,2],[2,4,5],[3,4],[5,6],[7],[6,7],[7],[]]
```

5. Dans cette fonction, on parcourt toutes les listes d'adjacences du graphe. Si on trouve dans la liste d'adjacence d'un sommet i un sommet $j \leq i$, le graphe n'est pas ordonnée et on peut interrompre la recherche et renvoyer `False`. Si la recherche se termine sans tomber sur ce cas de figure, alors le graphe est ordonné et on peut renvoyer `True` en fin de boucle.

```
def est_ordonne(L):
    for i in range(len(L)):
        for j in L[i]:
            if j<=i:
                return False
    return True
```

6. On crée un tableau de n listes vides qui vont servir à stocker les antécédents de chaque sommet, puis on parcourt à nouveau les listes d'adjacences. Si un sommet j est dans la liste d'adjance du sommet i, cela signifie que i est un antécédent de j donc on l'ajoute à la liste des antécédents de ce sommet.

```
def listes_antecedents(L):
    n=len(L)
    L2=[[] for i in range(n)]
    for i in range(n):
        for j in L[i]:
            L2[j].append(i)
    return L2
```

7. Fixons $i \geq 1$. Considérons l'ensemble E_i des chemins depuis le sommet 0 jusqu'au sommet i. On sépare l'ensemble de ces chemins en fonction du dernier sommet avant i sur ce chemin (qui est nécessairement un élément de $a(i)$), ce que l'on peut noter

$$E_i = \bigcup_{j \in a(i)} E_{i,j} \qquad \text{où} \qquad E_{i,j} = \{\text{chemins de 0 vers } i \text{ terminant par } (j,i)\}$$

Si l'on note $m_{i,j}$ le maximum des longueurs des chemins de $E_{i,j}$, on en déduit que

$$m_i = \max_{j \in a(i)} m_{i,j}$$

Pour finir, étant donné j appartenant à $a(i)$, la longueur maximale d'un chemin vers i terminant par (j,i) est nécessairement celle d'un plus long chemin jusqu'à j, auquel on rajoute le sommet i. En d'autres termes, $m_{i,j} = 1 + m_j$. Finalement,

$$m_i = \max_{j \in a(i)} m_{i,j} = \max_{j \in a(i)} (1 + m_j) = 1 + \max_{j \in a(i)} m_j$$

$$\boxed{\forall i \geq 1, \qquad m_i = 1 + \max_{j \in a(i)} m_j}$$

8. Il suffit de calculer tous les $(m_i)_{i \in [\![0;n-1]\!]}$ en utilisant la formule de récurrence de la question précédente. On utilise un tableau `m` de longueur n pour stocker ces valeurs au fur et à mesure du calcul. La fonction `listes_antecedents` est utilisée en début de code pour calculer les antécédents de tous les sommets.

```
1  def longueur_max(L):
2      n=len(L)
3      m=[0 for i in range(n)]
4      L2=listes_antecedents(L)
5      for i in range(1,n):
6          m[i]=1+max([m[j] for j in L2[i]])
7      return m[n-1]
```

Pour l'analyse de complexité, on part du principe que la fonction `append` est de coût constant.

- La fonction `listes_antecedents` (ligne 4) effectue un nombre constant d'opérations élémentaires pour chaque élément de chaque liste d'adjacence. Le cout total de cet appel à `listes_antecedents` est donc proportionnel à la somme des longueurs des listes d'adjacences soit en $O(|A|)$.
- Dans les lignes 5 et 6, on réalise cette fois pour chaque sommet un caclul de maximum sur un nombre d'éléments égal au nombre d'éléments de la liste des antécédents de ce sommet. La somme des coûts de ces calculs est de l'ordre à nouveau de la somme des longueurs des listes d'antécédents, soit à nouveau en $O(|A|)$.

Pour conclure,

La fonction `longueur_max` est de complexité $O(|A|)$ soit linéaire en le nombre d'arêtes du graphe.

9. Dans un premier temps, on modifie la fonction `longueur_max` en une fonction `antecedents_pgc` : l'objectif est de garder en mémoire, pour chaque entier i, l'indice j appartenant à $a(i)$ pour lequel le maximum est atteint dans la formule de récurrence de la question 7. Cet entier j est alors noté p_i (`p` signifiant « père »).

```
1  def antecedents_pgc(L):
2      n=len(L)
3      m=[0 for i in range(n)]
4      p=[(-1) for i in range(n)]
5      L2=listes_antecedents(L)
6      for i in range(1,n):
7          m[i]=1+max([m[j] for j in L2[i]])
8          for j in L2[i]:
9              if m[i]==1+m[j]:
10                 p[i]=j
11     return p
```

La preuve de cette formule assure également que pour réaliser un plus long chemin depuis le sommet 0 jusqu'au sommet i, il faut nécessairement trouver un plus long chemin depuis 0 jusqu'à p_i, auquel on rajoute le sommet i. On peut alors reconstruire un plus long chemin depuis le sommet 0 jusqu'au $n-1$: il suffit de partir du sommet $n-1$ puis de remonter de père en père jusqu'au sommet 0.

```python
def chemin_max(L):
    n=len(L)
    p=antecedents_pgc(L)
    k=n-1; c=[k]
    while k>0:
        k=p[k]; c.append(k)
    c.reverse()
    return c
```