TD 1 – Programmation dynamique

Exercice 1. *Implantations*

1. On s'intéresse au problème du rendu de monnaie.
   - **i.** Programmer l'algorithme RenduProgDyn. L'entrée doit être une liste `P` de valeurs de pièces, et une somme `s`, et la sortie le nombre minimal de pièces de `P` dont la somme est `s`.
     Attention, on ne peut pas utiliser la valeur $+\infty$, et la remplacer par un très grand entier est une mauvaise pratique : trouver une solution pour contourner la difficulté.
   - **ii.** Modifier le programme précédent, pour qu'il prenne un paramètre optionnel `tab` qui est `False` par défaut, et qu'il renvoie le tableau en entier si `tab` est `True`.
   - **iii.** Programmer l'algorithme RenduReconstruction.
   - **iv.** Programmer une version efficace en mémoire de RenduProgDyn, qui utilise un tableau aussi petit que possible.
2. On s'intéresse maintenant au problème de l'alignement de séquences. On suppose que les mots sont des chaînes de caractères Python.
   - **i.** Programmer l'algorithme Edition. De manière similaire au cas du rendu de monnaie, utiliser un paramètre optionnel pour renvoyer soit la distance, soit le tableau en entier.
     Rappel. On peut déclarer un tableau bidimensionnel de taille $m \times n$ initialisé à 0 avec l'instruction `E = [[0] * n for _ in range(m)]` [1].
   - **ii.** Programmer l'algorithme EditionMinMemoire.
   - **iii.** Programmer l'algorithme Alignement. La sortie est un couple de chaînes de caractères, de mêmes longueurs, qui réalise l'alignement.
   - **iv.** *Test :* comparer (un extrait de) la protéine CTCF [2] chez différentes espèces :
     - humain (*H. sapiens*) : EDSSDSENAEPDLDDNEDEEEPAVEIEPEPEPQPVTPA
     - chimpanzé (*P. troglodytes*) : EDSSDSENAEPDLDDNEDEEEPAVEIEPEPEPQPVTPA
     - loup (*C. lupus*) : EDSSDSENAEPDLDDNEDEEEPAVEIEPEPEPQPVTPA
     - vache (*B. taurus*) : EDSSDSENAEPDLDDNEDEEEPAVEIEPEPEPQPVTPA
     - souris grise (*M. musculus*) : EDSSDSEENAEPDLDDNEEEEEPAVEIEPEPEPQPQPPPPPQPVAPA
     - rat brun (*R. norvegicus*) : EDSSDSENAEPDLDDNEEEEEPAVEIEPEPEPQPQPQPQPQPQPVAPA
     - coq sauvage (*G. gallus*) : EDSSDSEENAEPDLDDNEDEEETAVEIEAEPEVSAEAPA
     - poisson zèbre (*D. rerio*) : DDDDDDSDEHGEPDLDDIDEEDEDDLLDEDQMGLLDQAPPSVPIPAPA
   - **v.** À partir des génomes fournis, justifier pourquoi on considère que le coronavirus ayant provoqué le Covid-19 provient plutôt d'une chauve-souris que d'une évolution du coronavirus humain ayant provoqué le SRAS [3] :
     - `2019-nCoV_WH01.fa` , `2019-nCoV_WH03.fa` et `2019-nCoV_WH04.fa` sont des virus prélevés chez des patients chinois lors de l'épidémie actuelle de Covid-19 ;
     - `bat-CoV-RaTG13.fa` est un virus de chauve-souris, séquencé en février 2020 ;
     - `SARS-CoV.fa` est un virus prélevé chez des patients lors de l'épidémie de SRAS (Asie, 2003).

     Attention, les calculs peuvent être longs, de l'ordre de 10 minutes par comparaisons sur mon ordinateur. On peut utiliser le code suivant qui écrit dans la variable `s` la chaîne contenue dans le fichier `"Fichier.ext"`, en supprimant les passages à la ligne :

```
with open("NomDuFichier.ext") as f:
    s = f.read().replace('\n','')
```

1. Attention, `[[0] * n] * m` est incorrect. Il est intéressant de comprendre pourquoi (en posant la question si besoin !).
2. Demander aux collègues biologistes si vous voulez des détails !
3. Remarque : la *vraie* détermination de l'origine du coronavirus est bien entendu plus complexe que ce qui est suggéré ici, l'objectif est simplement de montrer une utilisation possible de la distance d'édition.

Exercice 2. *Plus longue sous-suite croissante*

Étant donné un tableau T d'entiers, une *sous-suite croissante de longueur k* de T est un ensemble d'indices $i_1 < i_2 < \cdots < i_k$ tels que $T_{i_1} \leq T_{i_2} \leq \cdots \leq T_{i_k}$. Une *plus longue sous-suite croissante* (PLSSC) est une sous-suite croissante de longueur k telle qu'il n'existe aucune sous-suite croissante de longueur $k+1$ dans T. Étant donné T, on cherche une PLSSC.

1. Quel est le coût d'un algorithme qui teste pour chaque sous-suite possible si elle est croissante, et garde la plus longue ?

On souhaite décrire un algorithme de programmation dynamique pour le problème.

2. On commence par établir la formule récursive. Pour tout $i < n$, on note $\ell_T(i)$ la longueur de la plus longue sous-suite croissante qui commence en case T_i.
   - **i.** Quelle est la valeur qu'on souhaite connaître ?
   - **ii.** Déterminer $\ell_T(n-1)$ où n est la longueur du tableau.
   - **iii.** Soit $i_1 < i_2 < \cdots < i_k$ les indices d'une PLSSC commençant en case T_{i_1}. Montrer que la plus longue sous-suite croissante commençant en case T_{i_2} est de longueur $k-1$.
   - **iv.** En déduire que pour tout $i < n-1$, $\ell_T(i) = 1 + \max\{\ell_T(j) : j > i \text{ et } T_j \geq T_i\}$.
3. On veut déduire de la formule récursive un algorithme.
   - **i.** Écrire un algorithme de programmation dynamique qui calcule la longueur d'une PLSSC du tableau T et analyser sa complexité.
   - **ii.** Écrire un algorithme de reconstruction qui calcule une PLSSC du tableau, et analyser sa complexité.

Exercice 3. *Cœfficients binomiaux*

Pour $k \leq n$ donnés, on veut calculer le cœfficient binomial $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ (correspondant au nombre de choix de k éléments parmi n). On rappelle pour cela les formules du triangle de Pascal :

— pour tout entier n, on a $\binom{n}{0} = \binom{n}{n} = 1$ et
— pour $0 < k < n$, on a $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.

1. Proposer un algorithme récursif COEF-BIN(n,k) pour le calcul de $\binom{n}{k}$. Borner le nombre total d'appels récursifs pour calculer $\binom{n}{k}$.
2. On souhaite s'inspirer d'une procédure de type programmation dynamique en stockant pour chaque couple (i,j) avec $i \leq n$ et $j \leq k$ la valeur de $\binom{i}{j}$. Écrire l'algorithme correspondant et calculer sa complexité.

Exercice 4. *Stratégie étudiante*

Un étudiant veut établir son programme de travail du semestre en fonction des points qu'il espère gagner sur sa moyenne. Chaque semaine i du semestre, il doit décider s'il fait une *semaine tranquille*, à l'issue de laquelle il pense gagner $t(i)$ points sur sa moyenne, ou une *semaine harassante*, qui lui permettra, d'après lui, de gagner $h(i)$ points. Son énergie étant limitée, avant chaque semaine harassante, il doit se reposer la semaine précédente, ce qui lui rapporte 0 point. Le but est d'obtenir un planning qui maximise le nombre de points espérés.
Par exemple sur les semaines suivantes, sa stratégie optimale consiste à prendre une semaine de repos, puis faire une semaine harassante en Semaine 2 et enfin deux semaines tranquilles, ce qui lui rapportera 2,2 points.

	Semaine 1	Semaine 2	Semaine 3	Semaine 4
t	0,3	0,5	0,4	0,6
h	0,5	1,2	1	0,8

1. Montrer que l'algorithme suivant n'est pas optimal (n correspond au nombre de semaines dans le semestre).

```
i ← 1
Tant que i ≤ n :
  Si i ≤ n − 1 et h(i + 1) > t(i) + t(i + 1) :
      Se reposer semaine i et faire une semaine harassante semaine i + 1
      i ← i + 2
  Sinon :
      Faire une semaine tranquille semaine i
      i ← i + 1
```

2. Proposer une formule récursive pour calculer le nombre $p(i)$ maximal de points qu'on peut obtenir en travaillant jusqu'à la semaine i.
3. Donner un algorithme de type programmation dynamique pour calculer $p(n)$ et aussi produire la stratégie de travail associée.

Exercice 5. *Le retour du sac-à-dos*

On revient sur le problème du sac-à-dos[4]. Pour une instance $(T, (t_0, v_0), \ldots, (t_{n-1}, v_{n-1}))$ de ce problème, où toutes les valeurs sont entières, on dispose d'un sac-à-dos ayant une taille T et d'un ensemble d'objets $O_0, \ldots, O_{n-1}$, chaque objet O_i ayant une valeur v_i et une taille t_i. Le but est toujours de charger le sac-à-dos en maximisant la valeur emportée, mais cette fois, les objets ne sont fractionnables: pour tout $i = 1, \ldots, n$, soit on prend l'objet O_i, soit on ne le prend pas.
Pour résoudre cette version du problème, on va proposer deux solutions basées sur la programmation dynamique. Pour la première, on note $V(m,t)$ la valeur maximale que l'on peut atteindre en ne prenant que des objets parmi $O_0, \ldots, O_m$, pour une taille totale de t, pour $m < n$ et $t \leq T$.

1. Que vaut $V(m,t)$ si $t < t_m$? Et si $m = 0$?
2. Établir une relation de récurrence exprimant $V(m,t)$ en fonction de valeurs $V(m',t')$ avec $m' < m$ et $t' < t$. *On pourra étudier deux cas, selon que l'on choisisse l'objet O_m ou non.*
3. À l'aide de la formule précédente, écrire un algorithme résolvant le problème. Estimer la complexité de votre algorithme. Cette complexité est-elle polynomiale en la taille de l'entrée?
4. On souhaite maintenant fournir, en plus de la valeur optimale, le choix d'objets permettant d'obtenir cette valeur optimale. À l'aide d'une valeur $C_{[m,t]}$, on encodera le choix fait lors du calcul de $V(m,t)$.
   i. Compléter votre algorithme afin de calculer les valeurs $C_{[m,t]}$.
   ii. Répondre ensuite à la question en donnant un algorithme pour calculer un choix optimal d'objets.
5. On peut faire une variante de l'algorithme précédent, en définissant $t(m,v)$ comme étant la taille minimale d'un sac-à-dos qui atteindrait la valeur v à l'aide des objets O_0 à O_m. Décrire la solution du problème (valeur maximale) à partir des $t(m,v)$, une formule récursive pour $t(m,v)$ (avec les cas de base) et un algorithme correspondant (version sans et avec reconstruction de la solution). Quelle est la complexité obtenue ? Est-elle meilleure ou moins bonne que l'algorithme précédent ?

Exercice 6. ***Exercice à rendre.***

Au cours d'un jeu, il faut choisir un emplacement pour construire une ville. Le plateau de jeu est formé d'une grille carrée divisée en cellules et une ville occupe un ensemble de cellules libres formant un carré. Le but est de construire la plus grande ville possible et on souhaite mettre au point un algorithme pour décider de l'emplacement optimal. Le problème se modélise ainsi comme suit.

Entrée: une matrice A de taille $n \times n$ dont les entrées sont des `1` (occupée) et des `0` (libre).
Sortie 1: la taille de la plus grande sous-matrice carrée de A ne contenant que des `0`.
Sortie 2: l'emplacement de cette sous-matrice.

On note T_A la taille de la plus grande sous-matrice carrée ne contenant que des `0`. Pour $0 \leq i \leq n-1$ et $0 \leq j \leq n-1$, on note aussi $t_A(i,j)$ la taille k de la plus grande sous-matrice matrice carré $k \times k$ de A ne contenant que des `0` et dont le coin inférieur droit est la cellule $A_{[i,j]}$ de A. On dit qu'un tel carré est *basé en* $A_{[i,j]}$.

4. Vu dans le cours 2 du bloc 2.

$$A = \begin{array}{|c|c|c|c|c|}\hline 1&0&1&1&0\\\hline 0&0&0&1&0\\\hline 0&0&0&0&0\\\hline 1&1&0&0&0\\\hline 0&0&0&0&0\\\hline\end{array}$$

1. Sur l'exemple proposé, que vaut $t_A(0,0)$, $t_A(0,4)$, $t_A(2,2)$? Et que vaut T_A?
2. On cherche à établir un premier algorithme pour le calcul de T_A, pas forcément optimal en complexité.
   - **i.** Écrire un algorithme EXTENSIBLE(A,i,j,k) qui prend en entrée i, j et k tels que $t_A(i,j) \geq k$ et teste si $t_A(i,j) \geq k+1$. *En entrée de l'algorithme, on suppose (sans le vérifier) que* $t_A(i,j) \geq k$. Évaluer sa complexité.
   - **i.** En déduire un algorithme CARRÉ(A,i,j) qui étant donné i et j calcule la taille du plus grand carré de `0` basé en $A_{[i,j]}$. Évaluer sa complexité.
   - **i.** En déduire un algorithme pour calculer T_A. Quel est la complexité de votre algorithme ?
3. On va fournir maintenant un algorithme plus rapide, basé sur la programmation dynamique. On donne la relation de récurrence suivante. Pour $i \geq 1$ et $j \geq 1$, $t_A(i,j) = 0$ si $A_{[i,j]} = 1$ et $t_A(i,j) = 1 + \min\{t_A(i,j-1), t_A(i-1,j), t_A(i-1,j-1)\}$ sinon.
   - **i.** Donner les cas de base $t_A(0,j)$ et $t_A(i,0)$ et exprimer T_A en fonction des $t_A(i,j)$.
   - **i.** Sur l'exemple proposé, calculer T_A à l'aide de la formule proposée.
   - **i.** Prouver la relation de récurrence donnée.
   - **i.** Écrire un algorithme implantant la formule ci-dessus (sans preuve) et établir sa complexité.
   - **i.** Comment peut-on modifier l'algorithme pour renvoyer également la position du plus grand carré libre ?

Exercice 7. *Faire l'appoint*

Dans le problème du rendu de monnaie, on veut minimiser le nombre de pièces en rendant la monnaie. Dans la vraie vie, avant de lui rendre la monnaie, le client doit payer. On s'intéresse aux problèmes algorithmiques qui se posent du point de vue du client.

1. On suppose que le client possède autant de pièces qu'il souhaite de chaque type. Décrire un algorithme qui permet pour le client de minimiser le nombre de pièces que le caissier lui rend.

Dans la suite, on suppose que le client n'a qu'une somme finie dans son portefeuille, c'est-à-dire qu'il n'a qu'un nombre fini (éventuellement nul) de pièces de chaque valeur. Le caissier, lui, dispose de toutes les pièces en autant d'exemplaires qu'il veut. On suppose que le client doit payer un prix p inférieur ou égal à la somme totale s présente dans son portefeuille.

2. Le client souhaite, après avoir eu la monnaie, avoir le nombre minimum de pièces dans son portefeuille. Quel algorithme extrêmement simple peut-il utiliser ? *Indication : il laisse tout le travail au caissier.*
3. Le client redevient plus sympa avec le caissier. Il cherche à payer la somme la plus proche possible de p à l'aide de ses pièces. Décrire un algorithme pour résoudre ce problème.
4. En plus de payer la somme la plus proche possible, le client veut également utiliser le moins de pièces possibles. Comment faire ?
5. En fait, le mieux pour le caissier est d'avoir le moins de pièces à rendre. Que peut faire le client ?
6. Le client veut toujours être sympa avec le caissier, mais son but est tout de même de minimiser le nombre de pièces qu'il a dans son portefeuille après la transaction. Pour éviter de faire travailler le caissier de manière inutile, il s'impose la règle suivante : il ne doit donner au caissier aucune pièce que celui-ci lui rend ensuite. Décrire un algorithme pour ce problème.
7. Finalement, le caissier a été remplacé par un automate. Or l'automate est programmé pour ne rendre que certaines pièces [5] : il accepte les pièces d'un ensemble P mais ne rend que les pièces d'un sous-ensemble $R \subset P$. Décrire un algorithme pour que le client ait le moins de pièces possibles dans son

5. C'est le cas des machines à café des bâtiments d'enseignement où ont eu lieu les trois premiers blocs : ils ne rendent pas de pièce de 1€ ni de 2€, bien qu'ils les acceptent.

portefeuille après transaction. Essayer également de minimiser le nombre de pièces insérées par le client dans l'automate.

Exercice 8. *Autres distances*

1. Généraliser l'algorithme de calcul de la distance d'édition si on associe différents *coûts* aux désaccords : par exemple, une insertion de lettre coûte 1, une suppression 2 et un remplacement 3. *L'algorithme prend en paramètre le coût pour chaque type de désaccords.*
2. La distance de Damerau-Levenshtein est similaire à la distance d'édition avec un type de désaccord supplémentaire : on peut échanger deux lettres adjacentes. *Par exemple,* `CHINE` *est* `CHIEN` *se trouvent à distance de Damerau-Levenshtein* 1 *car il suffit d'inverser le* `N` *et le* `E` *pour passer de l'un à l'autre.*
   Généraliser l'algorithme de distance d'édition pour qu'il calcule la distance de Damerau-Levenshtein, ainsi que l'algorithme d'alignement.