COURS : Dictionnaires et programmation dynamique

I] Le type dictionnaire

A. Rappel sur les dictionnaires et leur manipulation

On accède aux éléments d'une liste via ses indices qui sont des entier compris entre 0 et $n-1$ (où n est le nombre d'éléments dans la liste), le type *dictionnaire* est une collection non ordonnée d'éléments qu'on accède par une *clé*, cette clé n'est pas nécessairement un entier, cela doit être un object python qui doit être *hachable* (ce qui correspond aux objets qui ne sont pas mutable, par exemple de type `int`, `float`, `bool`, `str` ainsi que les tuples d'objets de ce type, par contre cela ne peut pas être une liste). Cette notion sera détaillée plus loin.

Pour créer un dictionnaire en Python on utilise la syntaxe : `D={`c_1`:`v_1`, ..., `c_n`:`v_n`}`, où $c_1, \dots, c_n$ sont les clés et $v_1, \dots, v_n$ sont les valeurs associées à ces clés. Pour créer un dictionnaire vide on utilise donc : `D={}` (on peut aussi utiliser `D=dict()`).

Si `D` est un dictionnaire et `c` une clé :

- `D[c]` est la valeur de la clé `c` (erreur si elle n'existe pas) ;
- `D[c]=v` permet de modifier (ou de créer si elle n'existait pas) la valeur de la clé `c` ;
- `c in D` est le booléen qui vaut `True` si `c` est une clé de `D` et `False` sinon ;
- `len(D)` est le nombre de clés (donc de valeurs) dans le dictionnaires ;
- `D2=D.copy()` pour effectuer une copie du dictionnaire (attention c'est comme pour les listes, ...), si certaines valeurs sont des liste, il faudra utiliser `deepcopy` ;
- `for c in D` permet de parcourir les clefs du dictionnaire.

Il en existe d'autres (`del D[c]` pour enlever un élément, `D.keys()` pour avoir la liste des clés, `D.values()` pour avoir la liste des valeurs , `D.items()` pour avoir la liste des couples clé/valeur), mais elles ne sont pas exigible.

Remarque : Les dictionnaires permettent de représenter des ensembles, il suffit de ne pas considérer les valeurs associées aux clés.

Q1. On peut créer un dictionnaire Anglais/Français à l'aide .... d'un dictionnaire, par exemple avec `dicoAF={}` et on rajoute quelques traductions : `dicoAF['computer']='ordinateur'`, `dicoAF['keyboard']='clavier'`, etc.
Rajouter quelques mots, par exemple `'screen'` et `'souris'`.
Écrire une fonction qui prend un dictionnaire `dico` en paramètre dont on suppose que les valeurs sont des objets mutables tous distinct et qui créé un nouveau dictionnaire en inversant les clés et les valeurs, ainsi appliqué à `dicoAF` il créera un dictionnaire Français/Anglais

B. Principe derrière les dictionnaires

1/ Rappel sur le principe derrière les listes et les tableaux

En générale [1] la représentation en mémoire d'une liste utilise le principe des « listes chainées »

Pour les tableaux c'est plus simple, en effet un tableau ne contient que des éléments de même type (donc qui occupent le même espace mémoire) et que le nombre d'éléments et défini à la création on connait l'occupation mémoire à la création, ainsi :

2/ Dictionnaires

Une première idée pour implémenter un dictionnaire serait d'avoir une liste `lst_c=[`c_1`, ..., `c_n`]` des clés et une liste `lst_v=[`v_1`, ..., `v_n`]` des valeurs, pour obtenir la valeur de la clé c_k il faut trouver l'indice correspondant dans `lst_c`, disons i_k pour ensuite obtenir la valeur avec `lst_v[`i_k`]`. Cette première méthode n'est pas terrible au niveau de la complexité temporelle, il s'avère que les dictionnaires sont implémentés pour que les opérations de base aient (en vrai ce n'est pas tout a fait cela mais on fait tout comme) une complexité temporelle constante (ie $O(1)$).

Si les clés étaient des entiers entre 0 et $m-1$ il suffirait d'utiliser un tableau de taille m, pour se ramener à cette situation on utilise une fonction f qui à une clé $c \in C$ associe un entier compris entre 0 et $m-1$, comme le cardinal de C peut être grand, une telle fonction f ne peut pas être injective. Un coupe (c_1, c_2) tel que $f(c_1) = f(c_2)$ (avec $c_1 \neq c_2$) s'appelle une *collision*, c'est un problème qu'il faudra résoudre.

Pour créer une telle fonction f on utilise une fonction de hachage (par exemple la fonction `hasch` de Python, qui à un objet de type nom mutable associe un entier) et on prend le résultat modulo m. Le choix de m à une certaine importance, un m trop petit provoque beaucoup de collisions, un m trop grand risque de gâcher de l'espace mémoire.

Pour résoudre le problème de collisions on peut (par exemple) utiliser un tableau indexé par les entiers entre 0 et $m-1$ et l'élément à l'indice k est la liste des couples (c, v) (clé/valeurs) tels que $f(c) = k$. C'est ce qu'on appelle la *solution par chainage*, il en existe d'autres, par exemple celle par *adressage ouvert* [2].

Remarque 1 : Des collisions peuvent arriver plus souvent qu'on y pense, vous l'avez déjà montré en maths au lycée avec le paradoxe des anniversaire, pour un groupe d'individus on peut considérer la date d'anniversaire comme

1. ce n'est pas le cas en `Python` où le type `list` ne correspond pas à des listes, mais plutôt à des tableaux dynamiques
2. si on veut rajouter (c, v) mais qu'il y a déjà une clé $\tilde{c}$ tel que $f(\tilde{c}) = f(c)$ alors on va mettre (c, v) ailleurs que dans la case $f(c)$.

"fonction de hachage", il y a donc 365 possibilités, et vous avez vu que, dés que vous avez un groupe de plus de 23 individus il y a plus qu'une chance sur deux d'avoir une collusion de date d'anniversaire (et pour 59 individus on dépasse les 99% de chance de collusion).

Remarque 2 : Les fonctions de hachage ont d'autres applications en informatique, par exemple un serveur ne va pas stocker les mots de passe, mais uniquement l'application d'une fonction de hachage au mot de passe, en effet il est très couteux de trouver des antécédents par ces fonctions de hachage, ainsi si le serveur se fait pirater, les mots de passe ne seront pas compromis. Une autre utilisation classique est le `md5` (pour vérifier que le fichier qu'on a téléchargé n'a pas subit d'erreurs de transmissions.

Remarque 3 : La fonction `hash` de Python est assez compliquée, pour un entier n, elle retourne n modulo N, où $N = 2^{61} - 1$ (qui est un nombre premier), pour un négatif n, elle retourne $-\texttt{hash}(-n)$ (sauf -1 où la fonction retourne -2, la valeur -1 étant réservé pour signaler un problème), pour un rationnel $\frac{p}{q}$ elle retourne $\texttt{hash}(p)\texttt{hash}(q)^{-1}$ (où l'inverse est l'inverse dans $\mathbb{Z}/N\mathbb{Z}$ qui est un corps puisque N est premier). Pour les chaines de caractères c'est plus compliqué (en particulier pour un même mot les valeurs sont différents à chaque foi que l'on relance python).

II] Préliminaires à la programmation dynamique

A. Préliminaire sur la méthode "diviser pour régner"

Ce paradigme implique trois étapes à chaque niveau de la récursivité :
Diviser : le problème en 2 (ou plus) sous-problèmes ;
Régner : sur les sous-problème en les résolvant de manière récursive (ou directement s'il est de petite taille) ;
Combiner : les solutions des sous-problèmes pour avoir celle du problème initial.
C'est exactement la méthodologie adoptée par le *tri fusion* : on divise la suite des n éléments à trier en deux sous-suites de $n//2$ éléments, on trie les deux sous-suites de manière récursive en utilisant le tri fusion (s'il n'y a qu'un élément c'est déjà trié) et on fusionne les deux listes triées pour avoir notre solution, la partie importante de cet algorithme est la fonction auxiliaire qui permet de fusionner deux listes triées.

Q2. Pour un réel x et un entier n si on veut calculer x^n en n'utilisant que des multiplications et des additions, l'algorithme naïf consiste à faire $n-1$ produits (que ce soit de manière itérative ou de manière récursive), l'algorithme d'exponentiation rapide consiste juste à remarquer que $x^n = (x^2)^{n/2}$ si n est pair et que $x^n = x(x^2)^{(n-1)/2}$ si n est impaire

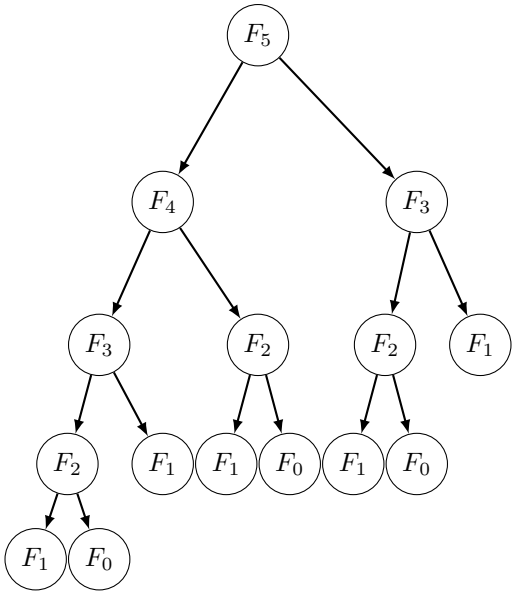
a) Écrire une fonction `puissance(x,n)` qui retourne la valeur de x^n en utilisant un algorithme naïf récursif.
b) Écrire une fonction `puissanceR(x,n)` qui retourne la valeur de x^n en utilisant l'exponentiation rapide.
c) Donner le nombre de multiplication dans le meilleur et le pire des cas.

B. Préliminaire sur la mémoïsation

Avant d'étudier la programmation dynamique on va considérer le calcul des termes de la suite de Fibonacci : (F_n), ie $F_0 = 0$, $F_1 = 1$ et pour tout $n \in \mathbb{N}$, $F_{n+2} = F_{n+1} + F_n$, un algorithme récursif naïf serait :

```
def fibo(n):
    if n==0 or n==1:
        return n
    else:
        return fibo(n-1)+fibo(n-2)
```

Le problème de cet algorithme, comme déjà vu, est sa complexité qui exponentielle (elle est en $O(\phi^n)$ où $\phi = \frac{1+\sqrt{5}}{2}$) voici les différents appels quand on veut calculer F_5



Ainsi pour calculer F_5 on a calculé F_2 trois fois, on veut donc éviter ces calculs superflus, pour cela on va présenter deux solutions.

Une première solution consiste à procéder au *calcul de bas en haut*, ie de calculer F_k pour k allant de 0 à n, ce qui se fait de manière itérative :

```
def fibo_iter(n):
    F=np.zeros(n+1)
    F[1]=1
    for k in range(n-1):
        F[k+2]=F[k+1]+F[k]
    return F[n]
```

On peut aussi utiliser ce qu'on appelle de la *mémoïsation* (ou méthode de haut en bas), cela consiste à garder les appels récursifs tout en utilisant une structure pour garder en mémoire ce qu'on a déjà calculé, un dictionnaire est tout indiqué (on peut aussi utiliser un tableau).

Q3. Écrire une fonction récursive `fibo(n)` qui retourne la valeur du `n`-ième terme de la suite de Fibonacci, cette fonction utilisera un dictionnaire `fibo_dic` créé avant la définition de la fonction et qui ne calculera que ce qui n'a pas était déjà calculé (et les nouvelles valeurs calculées seront stockées dans le dictionnaire).

Remarque : On a le même problème pour le calcul de $\binom{n}{p}$ avec la formule de Pascal, la même solution marche encore (une couple (n,p) peut être une clé d'un dictionnaire).

III] Programmation dynamique

La programmation dynamique[3] à été inventée par Bellman (vers 1954) pour résoudre des problèmes de chemins optimaux, c'est une méthode pour construire des algorithmes pour rechercher des solutions optimales dans un ensemble de solutions qui est certes fini mais très grand (et donc il n'est pas raisonnable de toutes les tester).

La programmation dynamique concerne des problèmes d'optimisation, la solution du problème de départ doit posséder une *propriété de sous-structure optimale* (qu'on verra sur des exemples) et un *chevauchement de sous problèmes* (contrairement à la méthode "diviser pour régner" qui s'applique plutôt quand les sous-problèmes sont indépendant). L'algorithme de Dijkstra est une amélioration d'un exemple de programmation dynamique, dans ce cadre la propriété de sous-structure optimale correspond au fait que si $(A_0,\dots,A_n)$ est un plus court chemin (donc optimal) pour aller de A_0 à A_n, alors $(A_0,\dots,A_k)$ et $(A_k,\dots,A_n)$ sont les plus cours chemins pour aller de A_0 à A_k et pour aller de A_k à A_n respectivement. Ainsi, $\text{len}(A,B) = \min_C(\text{len}(A,C)+\text{len}(C,B))$.

On va maintenant regarder la programmation dynamique sur deux exemples : la recherche d'une plus longue sous-séquence commune (plsc) de deux chaines de caractères et sur l'algorithme de Floyd-Warshall de recherche de plus court chemin dans un graphe qui ne contient pas de cycle de poids strictement négatif.

Il y a beaucoup de problèmes qui relèvent de la programmation dynamique, par exemple :
— Rendu de monnaie ;
— Problème du sac à dos ;
— Ordonnance de tâches pondérées ;
— Parenthésage d'un produit matriciel (pour avoir le moins de multiplications) ;
— Distance d'édition (Levenshtein) ;
— ...

A. Premier exemple : plus longue sous-séquence commune (plsc)

On considère deux chaines de caractères, par exemple `X="ATCTGAT"` et `Y="TGCATA"`, et on cherche une sous-séquence commune, par exemple `TA` en est une, l'objectif est d'en trouver une de longueur maximale.

Une solution naïve serait de déterminer toutes les sous-séquences de `X` et de déterminer pour chacune si elles sont des sous-séquences de `Y`, le problème est que si `X` possède m caractères alors il a 2^m sous-séquences, ce qui fait que notre algorithme naïf aurait une complexité non utilisable en pratique ...

On va commencer par introduire quelques notations, notons m le nombre de caractères de X et n celui de Y. On écrit aussi $X=(x_1,\dots,x_m)$ (où les x_i sont les caractères de la chaine X), idem pour Y. Pour $i\in[\![0,m]\!]$ on écrit $X_i=(x_1,\dots,x_i)$ avec la convention que X_0 est la chaine vide (et on remarquera que $X_m=X$).

3. qu'il aurait peut-être été préférable d'appeler méthode ou planification dynamique