Programmation dynamique - stage 1/2

Vendredi 23 et 30 septembre 2022

Buts du TP

- Utiliser des dictionnaires de façon élémentaire.
- Tester sur la suite de Fibonacci les calculs de bas en haut et de haut en bas.
- Mettre en place des schémas de programmation dynamique.

Exercice 1. *Créer (à un endroit raisonnable, donc pas à la racine de votre répertoire personnel...) un dossier associé à ce TP (et uniquement à lui, donc probablement dans un dossier qui contiendra tous vos TP de seconde année). Y placer une copie du fichier* `cadeau_prog_dyn.py` *récupéré dans le dossier partagé de travail de la classe, ainsi que le très mystérieux fichier* `dico0.txt`.
Lancer Spyder/Pyzo/Idle, sauvegarder immédiatement au bon endroit un nouveau fichier (`tp_prog_dyn1.py` *par exemple) ; écrire une commande absurde, de type* `print(5*3)` *dans l'éditeur, sauvegarder et exécuter.*

À cet instant, vous devez avoir deux fichiers d'ouverts dans votre éditeur : votre fichier de travail, et le fichier-cadeau (d'où vous copierez des fragments de code vers votre fichier de travail).

1 Dictionnaires : manipulations élémentaires

On commence par créer et comparer deux structures « semblables »...

Exercice 2. Listes vs dictionnaires
- *Créer une liste* `L` *telle que* `L[i]` *contient* i^2+1, *pour tout* $i \in [\![0, 10]\!]$.
- *Créer un dictionnaire* `D` *telle que* `D[i]` *contient* i^2+1, *pour tout* $i \in [\![0, 10]\!]$.
- *Créer un bidule* `B` *telle que* `B[x]` *contient* x^2+1, *pour tout réel* x *de la forme* $i/10$ *avec* $0 \leqslant i \leqslant 100$.
- *Quelles sont les « longueurs » de ce trois machins ?*
- *Au fait, que vaut* `B[0.4]` *?*

Dans le fichier-cadeau on trouvera une fonction permettant de lire un dictionnaire codé dans un fichier texte. Copiez-le et utilisez-le pour définir un dictionnaire, disons `dico0`, codé dans le fichier `dico0.txt`.

Exercice 3. Observons `dico0`

1. *Donner les valeurs de* `dico0[12], dico0['12'], dico0['douze'], dico0['plouf'], dico0['paf']`
2. *Combien d'items sont dans ce dictionnaire ?*
3. *Combien de clés sont des entiers ? Combien de valeurs sont des entiers ?*

2 Fibonacci : full récursif vs bottom-up vs top-down

On va calculer de trois façons les termes de la suite de Fibonacci, définie par $f_0 = 0$, $f_1 = 1$, et $f_{n+2} = f_n + f_{n+1}$ pour tout $n \in \mathbb{N}$.

Exercice 4. *Écrire une fonction récursive se contentant d'appliquer la définition de* (f_n). *Vérifier que* f_{10} *vaut bien* 55.

Vous trouverez dans le corrigé une fonction évaluant les temps d'exécution des différents programmes. (On anticipe sur les deux programmes qui suivront ; le premier temps est celui avec récursion brutale). Les résultats sont sans appels !

```
>>> for n in [10, 20, 30, 35, 40]:
       print(n, chrono_fibo(n))

10 [1.3589859008789062e-05, 3.0994415283203125e-06, 5.4836273193359375e-06]
20 [0.0013709068298339844, 3.0994415283203125e-06, 6.67572021484375e-06]
30 [0.16498970985412598, 7.62939453125e-06, 1.3828277587890625e-05]
35 [1.92362380027771, 9.059906005859375e-06, 1.52587890625e-05]
40 [21.89234471321106, 1.1205673217773438e-05, 2.0265579223632812e-05]
```

On constate que f_{50} ne pourra pas être calculé avec la version récursive naïve. Plus précisément, les temps d'exécution sont exponentiels... au sens des maths, pas au sens journalistique !

```
>>> vals = [chrono_fibo(n)[0] for n in range(29, 36)]
>>> [vals[i+1]/vals[i] for i in range(6)]

[1.6413406480447938, 1.6246678971370871, 1.6164893815738977, 1.6262669768463052,
 1.6286777447009309, 1.6303310843799739]

>>> (1+5**0.5)/2
1.618033988749895
```

On peut calculer f_n en créant une liste de $n+1$ zéros (`[0] * (n+1)` doit faire le job), en donnant la bonne valeur pour l'indice 1, puis en calculant de proche en proche les valeurs dans le tableau. On obtient alors un algorithme effectuant n additions : c'est l'approche « bottom-up ».

Exercice 5. *Écrire un tel programme !*

On vérifiera que f_{10} vaut toujours 55.
On termine cette partie avec une première implémentation de calcul « top-down », en créant un dictionnaire qu'on consulte en espérant trouver une valeur déjà calculée, et qu'on alimente avec une nouvelle valeur (après calcul et avant de la renvoyer) si elle n'était pas présente. Voici le schéma type de ce genre de programme :

```
def fibo3(n: int) -> int:
    remember = {0 : 0, 1 : 1}
    def fibo_rem(k: int) -> int:
        if not(k in remember):
            remember[k] = fibo_rem(k-1)+fibo_rem(k-2)
        return remember[k]
    return fibo_rem(n)
```

Exercice 6. *Comprendre le programme précédent, et déclarer le procédé admirable.*

Ça vaut vraiment le coup de passer un petit moment pour comprendre le code précédent : le principe sera copié à l'identique dans chaque calcul top-down de programmation dynamique par memoïzation.
Les performances vues plus haut nous montrent que ce calcul a un temps d'exécution du même ordre que la version tabulée, avec certes un facteur de l'ordre de 2, ce qui est très acceptable si on prend en compte le fait qu'avec la version bottom-up, on n'a pas à s'occuper du stockage des valeurs calculées (c'est facile ici, mais plus difficile ou long dans d'autres schémas de programmation dynamique).
Différence sensible quand même : l'appel `fibo3(10000)` donnera lieu à une erreur liée au trop grand nombre d'appels récursifs imbriqués.
Il n'y a pas unicité de l'approche top-down mémoïzée : on peut préférer une utilisation d'un paramètre optionnel pour transporter le dictionnaire (des valeurs calculées). C'est à la fois élégant et casse-gueule !

3 Plus long sous-suite croissante

Trouver la plus grande sous-suite croissante d'une suite (finie ; bref : une liste) de nombres n'est pas évident. Un algorithme naïf énumératif de toutes les sous-suites est exponentiel en la longueur. Il existe

un algorithme « quasi-linéaire », id est en $O(n \ln n)$, qui est en fait une (belle) amélioration de l'algorithme présenté ici, qui est lui en $O(n^2)$, avec n la longueur de la liste.
L'idée est de calculer, pour chaque i, la longueur de la plus grande sous-suite croissante **qui termine** en position i. Si on arrive à faire ça, il n'y aura plus qu'à déterminer le maximum de ces valeurs.
Si on note $\varphi(i)$ cette grandeur, alors :

$$\varphi(i) = 1 + \mathrm{Max}\{\varphi(j) \mid j < i \text{ et } L[j] \leqslant L[i]\}$$

avec L la liste en entrée. Il faut déjà s'en convaincre !

Exercice 7. *Avec cette belle idée, écrire un programme calculant* ***la longueur*** *de la plus grande sous-liste croissante d'une liste donnée.*

Spoiler : mon programme fait 9 lignes !

```
>>> plsscroi([10, 1, 5, 3, 4, 2, 3, 12, 1])
4
```

Avec un petit effort on doit pouvoir **expliciter** une sous-liste croissante de longueur maximale : il suffit de calculer de proche en proche les couples $\varphi(i) = (\ell, d)$, où ℓ est la longueur de la (d'une) plus grande sous-suite croissante terminant en position i, et d vaut (lorsque $\ell > 1$) l'avant-dernier indice d'une telle sous-suite (ce qui permettra de proche en proche de reconstruire une sous-suite optimale). On peut aussi (pour rendre plus simple la reconstruction, mais avec un petit coût en espace) prendre comme deuxième information une sous-suite de longueur maximale terminant en i : c'est ce que j'ai fait par paresse dans le corrigé.

Exercice 8. *Faites-le !*

```
>>> plsscroi_expl([10, 1, 5, 3, 4, 2, 3, 12, 1])
[1, 2, 3, 12]
```

Pour les matheux : si le tableau en entrée est une permutation aléatoire (au sens uniforme) de $[\![1, n]\!]$ alors l'espérance de cette longueur est équivalente à $2\sqrt{n}$ (googler/wikipédier pour avoir plus précis !)
La librairie `numpy.random` fournit une fonction `permutation` prenant en entrée une liste et renvoyant une permutation aléatoire (au sens uniforme) de cette liste.

```
>>> permutation(list(range(5)))
array([4, 3, 0, 2, 1])
```

Exercice 9 (Optionnel)**.** *Écrire une fonction prenant en entrée un nombre* `n` *(la taille des permutations), un nombre* `Ne` *d'expérience à réaliser, et renvoyant la moyenne des plus grandes sous-suites croissantes de permutations de* $[\![1, n]\!]$*, calculée sur* N_e *expériences.*

On peut se convaincre que « c'est bien en $O(\sqrt{n})$ » mais douter de la constante !

```
>>> esperance_longueur(25, 10**3)
7.535

>>> esperance_longueur(100, 10**3)
16.739

>>> esperance_longueur(400, 10**3)
35.546

>>> esperance_longueur(1600, 10**3)
74.468
```

Si vous googlez, vous verrez que « la convergence vers l'équivalent » (on se comprend n'est-ce pas ?) est assez lente, ce qui est rassurant quant à nos résultats expérimentaux.

4 Plus long sous-suite commune

On se donne deux listes `L1` et `L2`, et il s'agit de déterminer la longueur de la plus grande sous-suite commune. Par exemple pour les listes

$$[\mathbf{5}, 3, 11, \mathbf{18}, \mathbf{19}, 11, 1, \mathbf{7}, 16, 14] \quad \text{et} \quad [6, 16, 14, \mathbf{5}, 13, 12, 16, \mathbf{18}, \mathbf{19}, \mathbf{7}]$$

la plus grande sous-suite commune est de longueur 4.
Une telle longueur maximale peut être calculée par programmation dynamique, en répondant plutôt au sous-problème :

Quelle est la longueur maximale des sous-suites communes aux listes tronquées en i_1 et i_2 inclus ?

On note $\varphi(i_1, i_2)$ cette valeur, et le résultat recherché sera alors $\varphi(n_1 - 1, n_2 - 1)$, avec n_1 et n_2 les longueurs des deux listes. Il reste à se convaincre des formules suivantes :

$$\varphi(i_1, i_2) = \begin{cases} 0 & \text{si } i_1 < 0 \text{ ou } i_2 < 0 \\ 1 + \varphi(i_1 - 1, i_2 - 1) & \text{si } L_1[i_1] = L_2[i_2] \\ \text{Max}\left(\varphi(i_1 - 1, i_2), \varphi(i_1, i_2 - 1)\right) & \text{sinon} \end{cases}$$

Exercice 10. *Écrire une fonction calculant la longueur de la plus longue sous-suite croissante en suivant ce principe. On écrira donc une fonction globale très simple appelant le calcul récursif de φ.*

On pourra partir du modèle suivant :

```
def plsscfins1(L1, L2, i1, i2):
    ...

def plssc1(L1, L2):
    return plsscfins1(L1, L2, len(L1)-1, len(L2)-1)
```

On testera cette fonction sur l'exemple vu plus haut.
On passe maintenant à l'écriture d'un programme implémentant cette idée, mais en tabulant les valeurs : un calcul « bottom-up » qui va créer une liste de listes, disons `vals`, telle que `vals[i1][i2]` contiendra ultimement $\varphi(i_1, i_2)$. On peut initialiser une telle double-liste par exemple ainsi :

```
vals = [[0] * len(L1) for _ in range(len(L2))]
```

Exercice 11. *Écrire une fonction calculant la longueur de la plus grande sous-suite commune en tabulant les valeurs de φ.*
On commencera par réfléchir à la façon d'écrire les boucles, se dire que gérer les cas aux bords est très pénible, aller voir le corrigé, plisser les yeux, et se dire qu'on aurait certainement codé ça.

Quand je vous disais que le calcul bottom-up pouvait être pénible...

On termine par la version mémoïzée top-down, avec un dictionnaire. On va voir que c'est bien plus simple à écrire que la version tabulée.

Exercice 12. *S'inspirer du principe vu avec la suite de Fibonacci pour calculer la longueur de la plus grande sous-suite commune par memoïzation top-down.*

Sans surprise, les temps de calculent explosent pour la version récursive ; les deux autres versions ont des temps d'exécution quadratiques, la version top-down ne passant pas le cap de listes de tailles sensiblement plus grandes que 1000.

```
10 [0.028853178024291992, 3.24249267578125e-05, 6.079673767089844e-05]
11 [0.07154965400695801, 4.458427429199219e-05, 7.772445678710938e-05]
12 [0.6073739528656006, 5.030632019042969e-05, 9.751319885253906e-05]
13 [1.648728370666504, 5.793571472167969e-05, 0.000118255615234375]
14 [6.344505071640015, 6.604194641113281e-05, 0.00013971328735351562]
15 [25.344244718551636, 7.414817810058594e-05, 0.0001614093780517578]
```

Pour de plus grandes longueurs on exclut le premier algorithme.

```
>>> for n in [100, 200, 400, 800]:
        print(n, chrono_plssc(randlist(n), randlist(n))[1:])
```

Personnellement, je trouve que la dernière version est la meilleure : certes un facteur de 4-5 par rapport à la tabulée, mais un code beaucoup plus simple/limpide, donc plus fiable !
Le dernier exercice est optionnel mais intéressant. Il est réservé aux courageux/rapides pendant la séance, mais tout le monde est invité à y réfléchir a posteriori.

Exercice 13. *Écrire un programme calculant la (une) plus grande sous-suite commune à deux listes (et non la longueur !).*

On adaptera la méthode vue pour la plus grande sous-suite croissante.

5 Le problème du sac à dos

Il s'agit d'un problème classique, et important en informatique théorique : on dispose d'objets ayant un poids et une valeur, et on veut optimiser le remplissage d'un sac à dos qui ne peut supporter qu'un poids limité. Selon les contextes, on imaginera un randonneur optimisant son alimentation, ou bien un voleur optimisant son forfait.
Par exemple si les objets ont pour valeurs et poids respectifs :

$$(100, 10), (60, 5), (50, 5)$$

et que le poids total maximal est de 10, on pourra constituer un sac à dos de valeur 110 (mieux que si on avait gloutonnement commencé par prendre l'objet de plus grande valeur). Une fausse bonne idée (mais pas absurde !) consisterait à se dire qu'on appliquer un (autre) algorithme glouton en prenant l'objet « de plus grande valeur massique » possible, et continuer ainsi. Dans le cas précédent on obtiendrait le même sac optimal, mais on peut facilement construire un contre-exemple :

$$(100, 10), (90, 6)$$

L'algorithme glouton (basé sur la meilleure valeur massique) présenté plus haut conduirait à un sac de valeur 90, alors qu'un sac de valeur 100 est possible.
Une solution classique consiste à calculer $\varphi(i, P)$ la valeur maximale du sac à dos qu'on peut constituer en utilisant des objets parmi les i premiers, pour un poids maximal de P. Selon le bon vieux principe « je prends tel objet ou je ne le prends pas », on obtient :

$$\varphi(i, P) = \text{Max}\left(v_i + \varphi(i-1, P-p_i), \varphi(i-1, P)\right)$$

Exercice 14. *Écrire un programme mettant en place ce principe pour calculer un sac à dos optimal.*

On prendra garde aux effets de bord non décrit par la formule précédente (si $v_i > P$ il se passe des choses par exemple).
On pourra tester le programme sur les deux premiers exemples.

```
>>> knapsack([(100,10), (60,5), (60,5)], 10)
120

>>> knapsack([(100,10), (60,9), (60,9)], 10)
100
```

On peut bien entendu souhaiter obtenir non pas la valeur mais le contenu du sac à dos optimal !

Exercice 15. *Les courageux voudront bien, comme dans les deux parties précédentes, écrire un tel programme !*

```
>>> knapsack_contenu([(100,10), (60,5), (60,5)], 10)
(120, [(60, 5), (60, 5)])

>>> knapsack_contenu([(100,10), (60,9), (60,9)], 10)
(100, [(100, 10)])
```