Programmation dynamique - stage 2/2

Eulerismes, lecture de fichiers

Vendredi 7 (MP) et 14 (MP) octobre 2022*

Buts du TP

— Mettre en place de nouveaux schémas de programmation dynamique pour résoudre entre autres quelques problèmes du Project Euler.
— Observer et un peu pratiquer la lecture (voire l'écriture) de données dans un fichier texte.

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 des très mystérieux fichier* `pe18.txt`, `pe67.txt`, `matrices1.txt` *et* `matrices2.txt`.
Lancer Spyder/Pyzo/Idle, sauvegarder immédiatement au bon endroit un nouveau fichier (`tp_prog_dyn2.py` *par exemple) ; écrire une commande absurde, de type* `print(5*3)` *dans l'éditeur, sauvegarder et exécuter.*

1 Partitions d'entier (PE 76)

Voici un énoncé du Project Euler :
Problem 76 :
It is possible to write five as a sum in exactly six different ways :
4 + 1
3 + 2
3 + 1 + 1
2 + 2 + 1
2 + 1 + 1 + 1
1 + 1 + 1 + 1 + 1
How many different ways can one hundred be written as a sum of at least two positive integers ?

Le calcul des « partitions d'un entier » est un problème classique en maths comme en informatique. Maintenant habitués à ce type de situations attaquées par programmation dynamique, on réalise immédiatement qu'on peut s'en sortir en calculant (pour $n, k \in \mathbb{N}^*$) $\varphi(n,k)$ le nombre de partitions de n en somme d'entiers inférieurs ou égaux à k. Il s'agira alors de calculer $\varphi(100,100) - 1$ (la partition triviale étant exclue d'après l'exemple donné et le « at least two... »).
Toujours en pensant aux exemples vus dans le cours et le premier TP, il vient naturellement :

$$\forall k, n \geqslant 1 \qquad \varphi(n,k) = \begin{cases} 1 & \text{si } k = 1 \\ \varphi(n, k-1) & \text{si } n < k \\ 1 + \varphi(n, k-1) & \text{si } 2 \leqslant k = n \\ \varphi(n, m-1) + \varphi(n-m, m) & \text{si } 2 \leqslant k < n \end{cases}$$

En étendant la définition de φ on pourrait avoir un cas de moins, mais ce ne serait pas forcément pertinent...
On va pouvoir calculer cette quantité comme toujours :
— par récursivité bourrine, en se doutant que le temps de calcul va vite exploser ;
— de haut en bas par mémoïzation.

Exercice 2. *Écrire une fonction calculant φ avec le principe précédent (full récursif), puis revenir au problème initial avec $n = 5$ puis $n = 100$.*

Normalement, ça a du prendre de l'ordre de la minute pour $n = 100$: c'est mal ! Les développeurs de problèmes ne laissent rien au hasard : pour $n = 50$ ça passe, mais pour $n = 100$ le temps commence à devenir à la fois pénible et presque acceptable !

Exercice 3. *Reprendre ce calcul en mémoïzant, sur un principe déjà vu :*

```
def partitions2(n0: int) -> int:
    remember = {}
    def part_rec(n: int, k: int) -> int:
        ...
    return part_rec(n0, n0) -1
```

Normalement le résultat doit être obtenu en une fraction de seconde pour $n = 100$.

2 Chemins descendant dans un triangle (PE 18 et 67)

Voici deux énoncés Project Euler :
Problem 18 :
By starting at the top of the triangle below and moving to adjacent numbers on the row below, the maximum total from top to bottom is 23.

```
   3
  7 4
 2 4 6
8 5 9 3
```

That is, $3 + 7 + 4 + 9 = 23$.
Find the maximum total from top to bottom of the triangle below :

```
                75
               95 64
              17 47 82
             18 35 87 10
            20 04 82 47 65
           19 01 23 75 03 34
          88 02 77 73 07 63 67
         99 65 04 28 06 16 70 92
        41 41 26 56 83 40 80 70 33
       41 48 72 33 47 32 37 16 94 29
      53 71 44 65 25 43 91 52 97 51 14
     70 11 33 28 77 73 17 78 39 68 17 57
    91 71 52 38 17 14 91 43 58 50 27 29 48
   63 66 04 68 89 53 67 30 73 16 69 87 40 31
  04 62 98 27 23 09 70 98 73 93 38 53 60 04 23
```

NOTE : As there are only 16384 *routes, it is possible to solve this problem by trying every route. However, Problem* 67*, is the same challenge with a triangle containing one-hundred rows ; it cannot be solved by brute force, and requires a clever method !*

Problem 67 :
By starting at the top of the triangle below and moving to adjacent numbers on the row below, the maximum total from top to bottom is 23.

```
   3
  7 4
 2 4 6
8 5 9 3
```

That is, $3 + 7 + 4 + 9 = 23$.
Find the maximum total from top to bottom in `triangle.txt`*, a 15K text file containing a triangle with one-hundred rows.*

Le premier exemple portera donc sur :

```
exple1 = [[3], [7, 4], [2, 4, 6], [8, 5, 9, 3]]
```

Voici un script qui permet de lire en une ligne le contenu d'un fichier (élégant, non ?) :

```
exple2 = [[int(x) for x in ligne.split(',')] for ligne in open('pe18.txt')]
```

Exercice 4. *Écrire une commande permettant de récupérer le triangle de problème 67.*

Si on note $\varphi(i,j)$ la somme optimale qu'on peut obtenir en descendant depuis l'entier en position j de la ligne numéro i (c'est-à-dire 35 si $i=3$ et $j=1$ dans l'exemple du problème 18), alors on a (en notant n le nombre de lignes du tableau T indexé à la Python) :

$$\varphi(i,j) = \begin{cases} 0 & \text{si } i = n \\ T[i][j] + \text{Max}\left(\varphi(i+1,j), \varphi(i+1,j+1)\right) & \text{sinon} \end{cases}$$

On va vérifier que la formule donnée plus haut fonctionne.

Exercice 5. *Résoudre le problème 18 en écrivant une fonction calculant φ de façon récursive.*

```
def d_m_depuis(T:list, i: int, j: int):  # full récursif
    if ...
    else:
        ...

def descente_max1(T: list) -> int:
    return d_m_depuis(T, 0, 0)
```

La complexité du programme précédente est limpide : il y a 2^n descentes dans l'arbre (avec n le nombre de lignes, à une vache près). Ça passe pour $n = 14$ mais certainement pas pour $n = 100$ (problème 67). Il s'agit donc maintenant de reprendre ça en mémoïzant.

Exercice 6. *Sur un principe maintenant bien connu, écrire un programme calculant la valeur de la somme optimale par récursion mémoïzée.*

Vous pouvez maintenant résoudre le problème numéro 67 ; félicitations ! Les joueurs pourront même attaquer la chose au tableur !

Exercice 7 (Très optionnel)**.** *Depuis votre tableur préféré (WhateverOffice, ou même Google sheet !), ouvrir le fichier* `pe18.txt` *et résoudre le problème 18 en écrivant les formules qui vont bien... et en les faisant glisser !*

Comme toujours en programmation dynamique, on peut s'intéresser au chemin réalisant la somme optimale.

Exercice 8 (Optionnel : reconstruction du chemin)**.**
Écrire un programme fournissant le chemin optimal dans un triangle.

Personnellement j'ai calculé, en plus de la somme maximale, la position du (d'un) successeur permettant d'obtenir $\varphi(i,j)$, ceci pour chaque (i,j).

```
>>> descente_max3(exple1)
(23, [3, 7, 4, 9])

>>> descente_max3(exple2)
(1074, [75, 64, 82, 87, 82, 75, 73, 28, 83, 32, 91, 78, 58, 73, 93])
```

3 Multiplication de matrices

Si $A \in \mathcal{M}_{n,1}(\mathbb{K})$, $B \in \mathcal{M}_{1,n}(\mathbb{K})$ et $C \in \mathcal{M}_{n,1}(\mathbb{K})$ alors on peut calculer ABC de deux façons différentes : $(AB)C$ ou $A(BC)$. La première méthode demandera $n^2 + n^2$ multiplications, et l'autre $n + n$.
On cherche ici la façon optimale de parenthéser le calcul d'un produit de matrices de dimensions connues.

Exercice 9. *Expliciter une formule permettant (lorsque $i < j$) de calculer $\varphi(i, j)$ le nombre minimal de multiplications à effectuer pour calculer $M_i M_{i+1} ... M_{j-1}$*

On veillera en particulier aux cas limites...

Exercice 10. *Écrire un programme mettant en place le calcul précédent par mémoïzation. Tester !*

```
>>> matrices1 = [[10, 1], [1, 10], [10, 1]]

>>> produit_optimal(matrices1)
20
```

On peut évidemment souhaiter connaître, au delà de son coût, la façon optimale de calculer le produit !

Exercice 11 (Très optionnel : la reconstuction)**.**
Améliorer le programme précédent en reconstruisant le parenthésage optimal.

```
>>> produit_optimal_bis(matrices1)
(20, '(0)((1)(2))')
```

On peut enfin faire des tests sur des grosses suites de matrices. Pour cela on ira les chercher dans un fichier texte.

Exercice 12 (Optionnel : lire et écrire dans des fichiers)**.**
Écrire une fonction permettant de récupérer une liste de tailles de matrices dans un fichier du type `matrices1.txt` *ou* `matrices2.txt`. *Tester les programmes de calcul optimal (sans et avec reconstruction) sur ces exemples.*

```
>>> lire('matrices1.txt')
[[10, 1], [1, 10], [10, 1]]
```

Vous pouvez aussi jouer à *écrire* un fichier contenant une centaine de matrices à multiplier et le proposer aux autres sur le dossier partagé de la classe. Vous pouvez aussi calculer, pour une liste donnée de dimensions de matrices, quel aurait été le coût d'un produit « de gauche à droite » par exemple.

```
>>> lire('matrices2.txt')
[[19, 47], [47, 76],
 ...
 [64, 37], [37, 84]]

>>> produit_optimal_bis(lire('matrices2.txt'))
(238618,
 '((0)((1)((2)((3)((4)((5)  ...   (96))(97))(98))(99))')

>>> cout_gd(matrices1)
200

>>> cout_gd(lire('matrices2.txt'))
4488598
```