Correction du TP n°4 du 3/10/2023

Programmation dynamique, mémoïsation

Un grand classique de la programmation dynamique : les coefficients binomiaux. Le second exercice est un bon exercice d'entraînement semblable aux exemples déjà traités.

Exercice 4.1. Coefficients binomiaux

La fonction `comb` de la bibliothèque standard `math` renvoie les coefficients binomiaux pour vérifier les réponses obtenues. On suppose $0 \leq p \leq n$ comme l'indique l'énoncé.

1) Il est primordial de dessiner le triangle de Pascal dans sa version exploitable pour l'informatique : les lignes indicées par n et les colonnes par p.

n \ p	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	**10**	**10**	5	1	
6	1	6	15	**20**	15	6	1

ce qui fournit, si on l'avait oublié, sur l'exemple des trois cases en gras :

$$\binom{n}{p} = \binom{n-1}{p-1} + \binom{n-1}{p} \tag{1}$$

Cette règle ne peut pas être appliquée pour la colonne $p = 0$ car il n'y a pas de colonne $p-1$. De même la formule ne peut pas être utilisée pour $p = n$ car il n'y a pas de case (n-1,p) au dessus. Finalement ce sont tous les 1 du triangle de Pascal qu'il nous faut considérer comme cas de base :

```
def binom1(n, p):
    if p == 0 or n == p:
        return 1
    return binom1(n-1, p-1) + binom1(n-1, p)
```

On vérifie des petites valeurs comme $\binom{4}{3}$ ou $\binom{6}{3}$, mais on constate assez vite que l'on n'obtient pas $\binom{32}{16}$ par exemple.

2) Il n'y a effectivement aucune addition réalisée lorsque $p = 0$ ou $p = n$. Sinon la fonction écrite vérifie bien $\forall p \in [\![1, n-1]\!],\ C(n,p) = C(n-1,p-1) + C(n-1,p) + 1$. On procède par récurrence sur $n \in \mathbb{N}$ pour montrer $\mathcal{P}(n) : \forall p \in [\![0, n]\!],\ C(n,p) = \binom{n}{p} - 1$.

$\mathcal{P}(0)$ est vraie car la seule valeur possible pour p est zéro : $C(0,p) = C(0,0) = 0 = \binom{0}{0} - 1$.

On suppose $\mathcal{P}(n-1)$ pour un $n \in \mathbb{N}^*$ fixé. On distingue alors deux cas :

— Soit $p = 0$ ou $p = n$ et dans ce cas $C(n,p) = 0$ d'après la fonction écrite. Or dans ces deux cas, on a bien $\binom{n}{p} = 1$. Donc $C(n,p) = \binom{n}{p} - 1$.

— Si $p \in [\![1, n-1]\!]$, alors $C(n,p) = C(n-1,p-1) + C(n-1,p) + 1 = \binom{n-1}{p-1} + \binom{n-1}{p} - 1$ par hypothèse de récurrence. Donc $C(n,p) = \binom{n}{p} - 1$ d'après la formule de Pascal. Donc $\mathcal{P}(n)$.

On a donc montré $\mathcal{P}(n)$ pour tout $n \in \mathbb{N}$ d'après le principe de récurrence.

3) La formule de Stirling $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ permet de trouver :

$$\binom{2n}{n} = \frac{(2n)!}{n!\,n!} \sim \frac{\sqrt{2\pi 2n}\left(\frac{2n}{e}\right)^{2n}}{2\pi n\left(\frac{n}{e}\right)^{2n}} = \frac{1}{\sqrt{\pi n}}\,2^{2n} = \frac{4^n}{\sqrt{\pi n}} \tag{2}$$

Le nombre d'additions à réaliser croît exponentiellement avec n : il n'est pas réaliste d'utiliser la fonction de la question 1) pour de grandes valeurs de n.

Ce n'est pas du tout ce que nous faisons à la main en construisant le triangle de Pascal, où manifestement le nombre d'additions pour calculer $\binom{2n}{n}$ augmente comme $n^2/2$. On connait bien ce phénomène avec la suite de Fibonacci du cours : les sous-problèmes se chevauchent fortement et l'on calcule plusieurs fois les mêmes $\binom{n}{p}$ avec cette méthode.

4) Ce que nous faisons à la main en représentant un triangle de Pascal n'est rien d'autre que de la **programmation dynamique bottom-up impérative** : partir des problèmes les plus élémentaires et les combiner jusqu'à obtenir la solution au problème global : la valeur de $\binom{n}{p}$. Le papier nous sert de tableau informatique pour stocker les résultats intermédiaires.

Nous allons même pouvoir réaliser un peu moins d'opérations qu'on en réalise à la main ordinairement (on peut aussi optimiser à la main), en remarquant que certains coefficients binomiaux calculés ne sont jamais réutilisés, ou alors pour calculer des coefficients qui, eux, ne sont pas utilisés. Deux simplfications :

— Il est inutile d'aller au-delà de la colonne p pour calculer $\binom{n}{p}$,

— Les coefficients au dessous de la diagonale $i - j = n - p$ ne sont pas utiles, où i est l'indice de ligne et j celui des colonnes (vérifier en coloriant les coefficients utiles sur le tableau précédent).

Bien sûr il ne faut pas déborder du tableau triangulaire pour l'indice j, d'où l'utilisation des fonctions `min` et `max` dans la fonction suivante pour que l'indice j reste entre 1 et $i-1$. Comme l'indique l'énoncé, un tableau de taille $(n+1)\times(p+1)$ convient parfaitement.

```
def binom2(n, p):
    B = [[1] * (p+1) for _ in range(n+1)]
    for i in range(2, n+1):
        for j in range(max(1,i-n+p), min(i,p+1)):
            B[i][j] = B[i-1][j-1] + B[i-1][j]
    return B[n][p]
```

On a utilisé des listes pour simplifier, mais il serait plus judicieux d'utiliser un tableau `numpy` de type « unsigned int » de taille adaptée, plutôt que des entiers standards, très gourmands en mémoire.

On obtient instantanément $\binom{32}{16} = 601080390$ avec cette fonction.

5) Avec la fonction précédente, le calcul de $\binom{2n}{n}$ nécessite exactement n^2 additions. Cette complexité quadratique permet de calculer tous les coefficients binomiaux $\binom{n}{p}$ dont on peut avoir besoin avec moins de $n^2/4$ additions (on utilise la propriété $\binom{n}{n-p} = \binom{n}{p}$ éventuellement).

6) Seule la ligne $n-1$ est utile pour calculer la ligne n, il est donc possible de travailler avec un tableau à une seule dimension, de taille $p+1$. Comme deux coefficients sont utilisés pour en calculer un seul, il est astucieux de remplir la ligne de droite à gauche pour ne jamais écraser à tort un coefficient : la boucle sur j procède donc par valeurs décroissantes.

```
def binom2_bis(n, p):
    B = [1] * (p+1)
    for i in range(2, n+1):
        for j in range(min(i-1, p), max(0, i-n+p-1), -1):
            B[j] += B[j-1]
    return B[p]
```

Outre le gain en complexité spatiale ($O(p)$ au lieu de $O(np)$), la manipulation d'un tableau à une seule dimension est forcément plus rapide. Noter que si l'on ne pense pas à balayer le tableau de droite à gauche, il suffit de mettre à jour une variable temporaire dans la boucle car la valeur écrasée ne servirait que dans l'itération suivante.

7) Il nous a fallu réfléchir sérieusement pour ne pas calculer inutilement certains coefficients avec la programmation dynamique bottom-up. L'intérêt de la **mémoïsation** et de la **programmation top-down** réside dans la possibilité de faire exactement les mêmes calculs mais sans y réfléchir préalablement !

```
def binom3(n, p):
    if p == 0 or p == n: return 1
    if (n,p) not in d:
        d[n,p] = binom3(n-1, p-1) + binom3(n-1, p)
    return d[n, p]
```

Pour ne pas écrire de fonction locale, on a supposé qu'un dictionnaire vide a été créé de manière globale. Voici un exemple d'exécution :

```
d = {}
print(binom3(32,16))
```

```
601080390
```

On peut observer le contenu du dictionnaire à la fin des calculs : il contient exactement 256 entrées avec cet exemple, c'est-à-dire que le nombre d'additions réalisées pour calculer $\binom{2n}{n}$ est toujours exactement n^2 : **les simplifications auxquelles nous avions réfléchi sont automatiquement réalisées car la progression top-down ne résout que les sous-problèmes vraiment utiles !**
