SÉANCE 6

PROGRAMMATION DYNAMIQUE

I - L'exemple du calcul des coefficients binomiaux

◇ La formule du triangle de Pascal permet le calcul de $\binom{n}{p}$ à l'aide de deux calculs plus simples :

$$\forall (n,p) \in (\mathbb{N}^*)^2, \quad \binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}.$$

Cette formule suggère une programmation récursive du calcul :

```
def binomial(n, p):
    """ entrée : deux entiers n et p
        sortie : entier égal au coefficient p parmi n
    """
    if p < 0 or p > n:
        return 0
    if p == 0 or p == n:
        return 1
    return binomial(n-1, p-1) + binomial(n-1, p)
```

On constate un problème de vitesse de calcul :

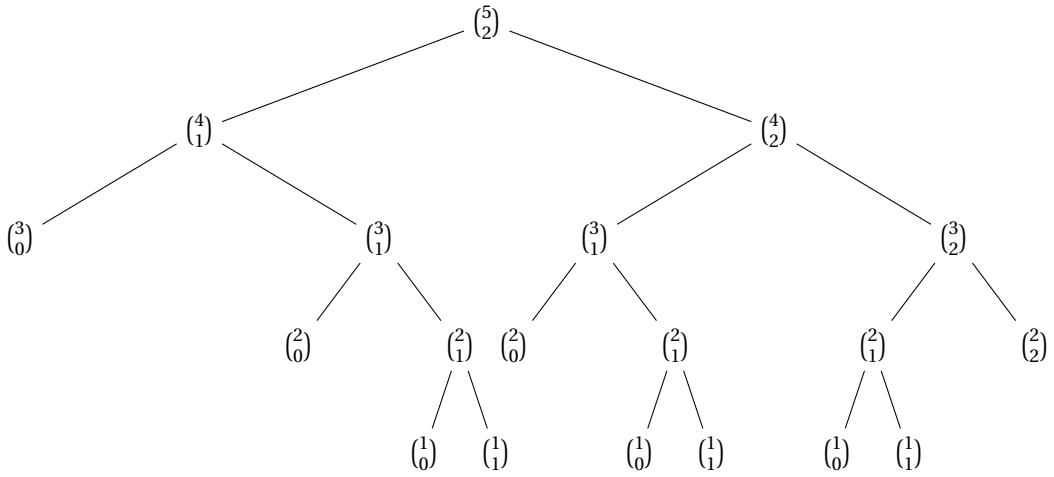
```
>>> from time import time
>>> t = time(); binomial(22, 11); time()-t
705432
0.3385660648345947
>>> t = time(); binomial(25, 13); time()-t
5200300
2.622504711151123
```

En fait, comme le résultat est obtenu en additionnant des 1, la complexité pour le calcul de $\binom{n}{p}$ est en $O\left(\binom{n}{p}\right)$.

Exercice

Justifier que, par cette méthode, le calcul de $\binom{2n}{n}$ est de complexité exponentielle en n.

L'arbre suivant illustre le fait que l'on calcule plusieurs fois les mêmes coefficients.



Pour calculer $\binom{5}{2}$, on va donc évaluer $\binom{4}{1}$ et $\binom{4}{2}$; et pour trouver ces deux termes, on va à chaque fois avoir besoin du coefficient $\binom{3}{1}$.

La fonction ramène le calcul d'un coefficient binomial au calcul de deux coefficients inférieurs cependant, et contrairement à ce qu'il se passe dans une approche du type *diviser pour régner* (cf. tri fusion par exemple), ici les deux sous-problèmes ne sont pas indépendants.

Ces calculs redondants ne sont pas anecdotiques : pour calculer $\binom{20}{10}$, le coefficient $\binom{2}{1}$ va être évalué presque 50 000 fois ! Il faut donc mettre en place un système pour ne pas perdre du temps à recalculer ce qui a déjà été déterminé.

◇ On va palier ce problème de temps avec une autre stratégie. On commence par résoudre les «plus petits» problèmes et on exploite leurs solutions pour résoudre des problèmes de plus en plus grands. Concrètement, on a va utiliser un tableau de nombres initialisé avec des zéros et on va calculer les coefficients de proche en proche.

```
def binomial_iter(n, p):
    t = [[0 for _ in range(p+1)] for _ in range(n+1)] # T comme triangle !
    for k in range(n+1):
        t[k][0] = 1
        if k <= p:
            t[k][k] = 1
    for i in range(2, n+1):
        for j in range(1, p+1):
            t[i][j] = t[i-1][j-1] + t[i-1][j]
    return t[n][p]
```

La complexité pour le calcul de $\binom{n}{p}$ est désormais en $O(np)$ ce qui améliore considérablement la situation ! Il y a également un coût en espace puisque l'on crée un tableau de taille $(n+1)\times(p+1)$. Ce dernier point pourrait néanmoins être amélioré en ne conservant qu'une ligne à chaque étape.

Il y a deux autres inconvénients :

- on calcule de nombreux coefficients pour rien ;
- on perd en lisibilité par rapport à la version récursive.

On peut concilier les deux aspects (la concision de la programmation récursive et l'efficacité du procédé itératif) en utilisant une méthode de ***mémoïsation*** : on va stocker les valeurs déjà calculées et ne faire un appel récursif que lorsque le calcul est utile.
Pour stocker les valeurs déjà calculées, on peut utiliser à nouveau une liste de listes mais la structure de dictionnaire est particulièrement adaptée puisqu'elle permet d'utiliser des clés de la forme `(n,p)`.

```
d = {}

def binomial_memo(n, p):
    if (n, p) not in d:
        if p < 0 or p > n:
            res = 0
        elif p == 0 or p == n:
            res = 1
        else:
            res = binomial_memo(n-1, p-1) + binomial_memo(n-1, p)
        d[(n, p)] = res
    return d[(n, p)]
```

Voici une autre version pour éviter la variable `d` en dehors de la fonction :

```
def binomial_memo_bis(n, p):
    d = {}
    def aux(m, q):
        if (m, q) not in d:
            if q < 0 or q > m:
                res = 0
            elif q == 0 or q == m:
                res = 1
            else:
                res = binomial_memo(m-1, q-1) + binomial_memo(m-1, q)
            d[(m, q)] = res
        return d[(m, q)]
    return aux(n, p)
```

II - Les principes de la programmation dynamique

L'idée essentielle est que toute étape d'une solution optimale est elle-même optimale, c'est le *principe d'optimalité de Bellman*.

La résolution d'un problème par programmation dynamique consiste donc déterminer un algorithme fondé sur une ***relation de récurrence*** puis à programmer en stockant les résultats des sous-problèmes afin d'éviter des calculs inutiles.

Pour cette programmation, il y a deux stratégies.

- Une démarche itérative.
  Il s'agit de :
  - trouver une relation de récurrence reliant la solution du problème aux solutions des sous-problèmes ;
  - définir un tableau (ou un dictionnaire) de taille adéquate et l'initialiser avec les valeurs triviales ;
  - résoudre les sous-problèmes de taille de plus en plus grande (boucles utilisant les formules de récurrence) ;
  - lire la solution dans le tableau (ou la récupérer si la lecture n'est pas directe).
- Une démarche récursive avec mémoïsation.
  À chaque appel récursif, on vérifie si la valeur est déjà connue et, sinon, on la calcule et on la stocke.

III - L'exemple du problème du sac à dos

On dispose d'un sac à dos dont la charge utile est limitée à un poids maximal $p_{\max}$ et de n objets $x_0, x_1, \dots, x_{n-1}$ possédant chacun un poids p_i et une valeur v_i. Le but est de remplir le sac en emportant la valeur maximale sans dépasser le poids limite.

On a déjà évoqué lors du cours précédent l'idée de procéder par «force brute» : on liste toutes les possibilités et on prend la meilleure. S'il y a n objets, il y a alors 2^n possibilité. C'est inutilisable en pratique. On a également évoqué l'idée d'utiliser une stratégie «glouton», par exemple en classant les objets selon le rapport $\frac{\text{valeur}}{\text{poids}}$ et en donnant systématiquement la priorité à l'objet présentant ce meilleur rapport.

On s'intéresse dans ce qui suit à la mise en place d'une stratégie de programmation dynamique.

On note $f(k,p)$ la valeur maximale obtenue avec les objets $x_0, \dots, x_k$ ne dépassant pas le poids p (on recherche donc ici $f(n-1, p_{\max})$. On cherche une relation de récurrence vérifiée par la fonction f.

- Si $k = 0$ alors il y a deux cas à distinguer pour $f(0,p)$:
  - si $p_0 > p$ alors $f(0,p) = 0$,
  - si $p_0 \leqslant p$ alors $f(0,p) = v_0$;
- pour $k \neq 0$, l'idée est de relier $f(k,p)$ à $f(k-1,p')$:
  - si $p_k > p$ alors $f(k,p) = f(k-1,p)$,
  - si $p_k \leqslant p$ alors $f(k,p) = \max\big(f(k-1,p), v_k + f(k-1,p-p_k)\big)$.

III.1 - Stratégie itérative (ascendante)

```
def sac_a_dos(objets, pMax):
    """ entrées :
            objets liste de 2 listes d'entiers (poids et valeurs) de même longueur
            pMax entier > 0
        sortie :
            valeur maximale pour un poids total <= pMax
    """
    p, v = objets  # poids et valeurs
    n = len(p)
    tab = [[0 for _ in range(pMax+1)] for _ in range(n)]
    for j in range(pMax+1):
        if j < p[0]:
            tab[0][j] = 0
        else:
            tab[0][j] = v[0]
    for i in range(1, n):
        for j in range(pMax+1):
            if j < p[i]:
                tab[i][j] = tab[i-1][j]
            else:
                x = tab[i-1][j]
                y = v[i] + tab[i-1][j-p[i]]
                tab[i][j] = max(x, y)

    return tab[n-1][pMax]
```

Par exemple, avec les listes de poids et valeurs données dans la variable `test` et `pMax=14`, on obtient une valeur maximale de 24.

```
>>> test = [[3, 8, 5, 1, 6, 1, 2, 6, 6], [1, 2, 6, 3, 7, 8, 2, 3, 4]]
>>> sac_a_dos(test, 14)
24
```

Exercice

Comment obtenir, non seulement la valeur maximale, mais également la liste des objets concernés ?