Chapitre 2

Programmation dynamique

Table des matières

2 Programmation dynamique **1**
2.1 Introduction . . . 1
2.2 Autour de la suite de Fibonacci . . . 2
2.2.1 Un problème posé par la programmation récursive . . . 2
2.2.2 Amélioration de l'algorithme par programmation ascendante . . . 3
2.2.3 Amélioration de l'algorithme par mémoïsation dans un tableau . . . 3
2.2.4 Amélioration de l'algorithme par mémoïsation dans un dictionnaire . . . 3
2.2.5 Amélioration par programmation sans mémoïsation . . . 4
2.3 Un second exemple . . . 4
2.3.1 Un calcul naïf des coefficients binomiaux . . . 4
2.3.2 Evaluation de la complexité . . . 4
2.3.3 Origine du problème . . . 5
2.3.4 Solution proposée par la programmation dynamique . . . 5
2.4 Problèmes d'optimisation . . . 7
2.5 Le problème du sac à dos . . . 8
2.5.1 Algorithme glouton et heuristique . . . 8
2.5.2 Algorithme dynamique par programmation ascendante . . . 8
2.5.3 Algorithme dynamique avec mémoïsation . . . 9
2.6 Plus longue sous-suite commune . . . 10
2.6.1 Position du problème . . . 10
2.6.2 Résolution par force brute . . . 10
2.6.3 Mise en évidence de sous-problème vérifiant la propriété de sous-structure optimale. . . . 11
2.6.4 Mise en place de l'algorithme naïf et complexité . . . 11
2.6.5 Chevauchement des solutions . . . 12
2.7 Algorithme de Floyd-Warshall . . . 12
2.7.1 Description de l'algorithme . . . 12
2.7.2 Mise en œuvre pratique . . . 13
2.7.3 Application au calcul de la fermeture transitive d'un graphe . . . 13

2.1 Introduction

La programmation dynamique s'applique à des problèmes pour lesquels on doit faire un ensemble de choix et arriver à une solution optimale.
A mesure que de nouvelles options sont choisies, des sous-problèmes de la même forme apparaissent.

Le principe est de stocker la solution à chaque sous-problème au cas où il réapparaîtrait.
On procède par mémoïsation : ce terme désigne une technique d'optimisation de code consistant à réduire le temps d'exécution d'une fonction en mémorisant (dans un dictionnaire ou un tableau...) ses résultats d'une fois sur l'autre afin de ne pas les recalculer.
En pratique :

— Décomposer le problème en des sous-problèmes plus petits ;
— Calculer les solutions optimales de tous ces sous-problèmes et les garder en mémoire.
— Calculer la solution optimale à partir des solutions optimales des sous-problèmes

2.2 Autour de la suite de Fibonacci

2.2.1 Un problème posé par la programmation récursive

La suite de Fibonacci est donnée par la récurrence :

$$\begin{cases} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2}, \quad \forall n \geqslant 2 \end{cases}$$

Il est naturel de l'implémenter par une fonction récursive :

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

Observons en détail les appels récursifs requis pour le calcul de F(8)

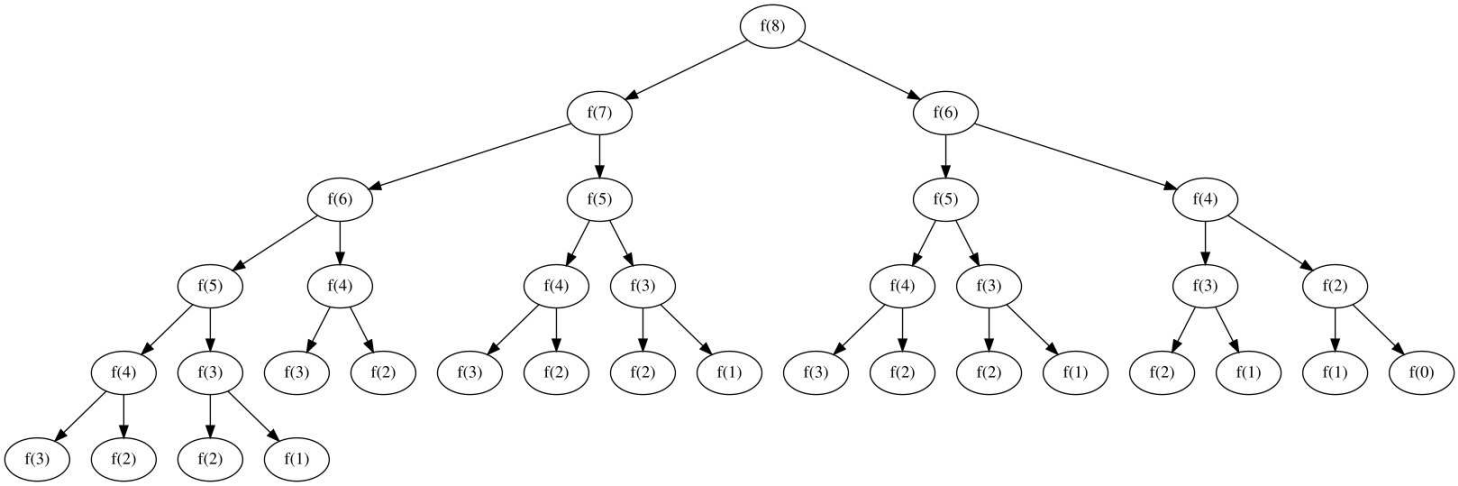


FIGURE 2.1 – Décomposition de F(8) en graphe

On observe que certaines parties du graphe se chevauchent, ce qui est une perte de temps. Plus précisément, la complexité temporelle de l'algorithme est exponentielle de type $O(\varphi^n)$ où φ est le nombre d'or [1].
La programmation dynamique va nous aider à gérer les problèmes de chevauchement précédents ce qui permettra d'améliorer grandement le programme précédent.
Il y a deux approches possibles :

1. Si $T(n)$ désigne le nombre d'additions nécessaires au calcul de $F(n)$ on observe que $(T(n))$ satisfait la récurrence :

$$\begin{cases} T(0) &= T(1) = 0 \\ T(n+2) &= T(n+1) + T(n) + 1, \quad \forall n \geqslant 2 \end{cases}.$$

Une solution particulière de cette récurrence linéaire non homogène est $\alpha(n) = -1$. ALors $T(n) - \alpha_n$ satisfait la récurrence double $u(n+2) = u(n+1) + u(n)$. On a alors $u(n) = \alpha\varphi^n + \beta\psi^n$ avec $\alpha, \beta \in \mathbb{R}$ et $\varphi = \frac{1+\sqrt{5}}{2}$ et $\psi = \frac{1-\sqrt{5}}{2}$. On en déduit que $T(n) \underset{n\to+\infty}{\sim} \alpha\varphi^n$ avec $\varphi > 1$.

— La première consiste à ordonner les calculs en partant des cas limites et en respectant l'ordonnancement imposé par la relation de récurrence, cela revient à parcourir le graphe précédent du bas vers le haut. Elle utilise en général un tableau pour mémoïser les valeurs calculées. Elle est en générale plus efficace que l'algoritme récursif naïf mais donne des algorithmes plus difficiles à lire
— La seconde est consiste à associer à la fonction récursive un dictionnaire ou un tableau qui stockera les valeurs déjà calculées. Ainsi, à chaque fois que le programme aura besoin de calculer une valeur, il ira voir dans le dictionnaire si la valeur dont il a besoin a déjà été calculée, et ne réalisera le calcul que dans le cas contraire, en ajoutant ensuite la nouvelle valeur calculée au dictionnaire.

2.2.2 Amélioration de l'algorithme par programmation ascendante

Commmençons par l'algorithme ascendant.

```
#De bas en haut
def fibo_ascendant(n):
    L = [None for _ in range(n+1)]
    L[0] , L[1] = 1 , 1
    for i in range(2,n+1):
        L[i] = L[i-1]+L[i-2]
    return(L[n])
```

2.2.3 Amélioration de l'algorithme par mémoïsation dans un tableau

Voici l'algorithme de mémoïsation avec un tableau :

```
#mémoïsation par tableau
def fibo(n):
    L=[None for _ in range(n+1)]
    L[0] = 1
    L[1] = 1
    def f(m):
        if L[m] != None:
            return(L[m])
        else:
            L[m] = f(m-1)+f(m-2)
            return(L[m])
    return(f(n))

print(fibo(10))
```

La mémoïsation permet d'élaguer l'arbre de récursivité. Les noeuds de cet arbre ne sont calculés qu'une fois.
Le nombre d'additions à effectuer revient à dénombrer le nombre de cases à remplir dans le tableau. On a donc une complexité temporelle en O(n).
La complexité spatiale est aussi en O(n) (taille du tableau n).

2.2.4 Amélioration de l'algorithme par mémoïsation dans un dictionnaire

Voici maintenant l'algorithme de mémoïsation avec un dictionnaire :

```
#mémoïsation par dictionnaire
def fibo(n):
    dico = {}
    dico[0] = 1
    dico[1] = 1
    def f(m):
        if m in dico:
            return(dico[m])
        else:
            dico[m] = f(m-1)+f(m-2)
            return(dico[m])
    return(f(n))
```

L'intérêt du dictionnaire est qu'on n'a pas besoin de générer un tableau de taille $n+1$, seuls les calculs effectués sont stockés en mémoire d'où un gain en complexité spatiale.

2.2.5 Amélioration par programmation sans mémoïsation

```
#Programmation ascendante
def fibo(n):
    if n in [0,1]:
        return(1)
    a,b = 1, 1
    for _ in range(n-1):
        a,b = b,a+b
    return(b)
```

On calcule les termes de la suite de proche en proche. La complexité temporelle est en O(n) et la complexité spatiale est en O(1).
C'est l'algorithme le plus efficace.

2.3 Un second exemple

2.3.1 Un calcul naïf des coefficients binomiaux

On calcule le coefficient binomial $\binom{n}{p}$ en utilisant la formule de Pascal et un algorithme récursif :

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

Pour calculer $\binom{30}{15}$, il faut près de 35s à mon ordinateur. Quand on regarde l'arbre des appels récursifs pour un calul de $\binom{5}{2}$, on comprend vite la raison :

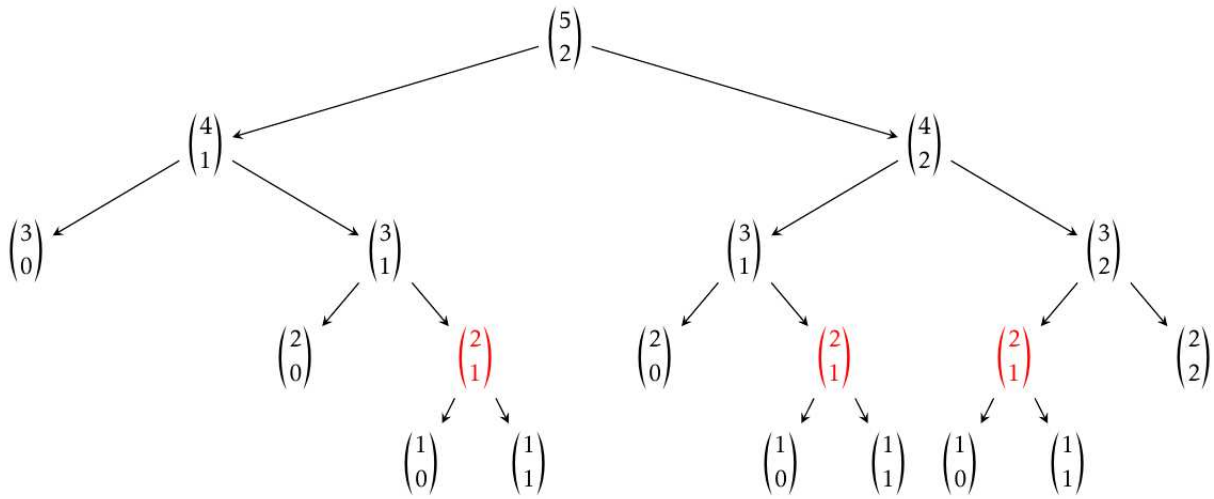


FIGURE 2.2 – Le calcul de $\binom{5}{2}$ fait appel à 3 fois le calcul de $\binom{2}{1}$

Il y a de nombreux calculs répétés ! Dans le calcul de $\binom{30}{15}$, on fait appel 40 116 600 fois au calcul de $\binom{1}{2}$...

2.3.2 Evaluation de la complexité

On évalue la complexité temporelle du calcul de $\binom{n}{p}$:

On note $C(n,p)$ le nombre d'additions réalisées dans le calcul de $\binom{n}{p}$ avec le programme récursif précédent.
On a :

$$C(n,0) = C(n,p) = 0 \quad \text{et} \quad \forall p \in [\![1, n-1]\!], \quad C(n,p) = C(n-1,p-1) + C(n-1,p) + 1$$

On montre alors que, par une récurrence sur $n \in \mathbb{N}$, pour $p \in [\![1,n]\!]$, on a $C(n,p) = \binom{n}{p} - 1$.
Par la formule de Stirling, on obtient que $\binom{2n}{n} \underset{n\to+\infty}{\sim} \frac{4^n}{\sqrt{\pi n}}$ et donc le calcul de $\binom{2n}{n}$ par cette fonction est de complexité exponentielle.

2.3.3 Origine du problème

Le calcul de $\binom{n}{p}$ fait appel au calcul de $\binom{n-1}{p-1}$ et $\binom{n-1}{p}$ qui sont deux sous-problèmes en interaction, cette dernière se faisant par chevauchement.
C'est exactement ce qui fait croitre la complexité de la fonction de faction rédhibitoire.

2.3.4 Solution proposée par la programmation dynamique

L'idée de la programmation dynamique est de résoudre en premier les plus petits sous-problèmes puis de combiner leurs solutions pour résoudre les problèmes de plus en plus grand.
Pour le calcul de $\binom{5}{2}$, on suit ainsi le schéma suivant :

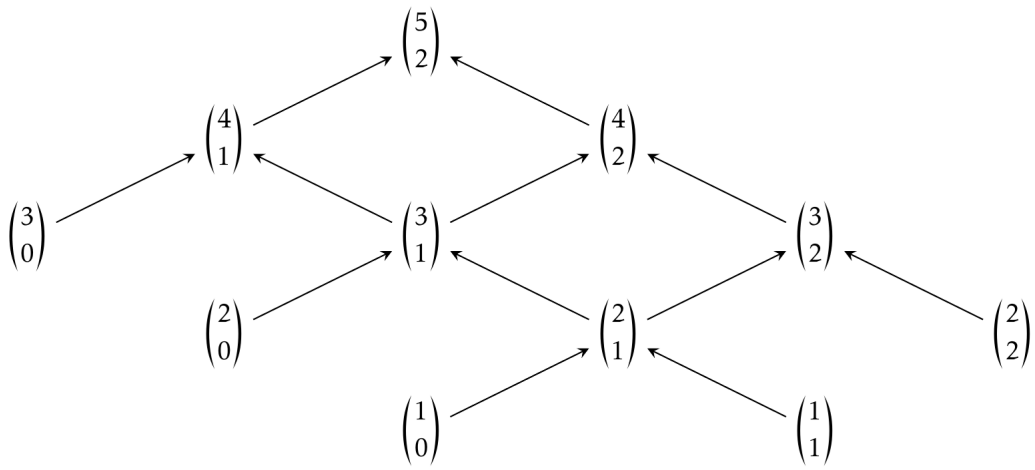


FIGURE 2.3 – Le calcul de $\binom{5}{2}$ en programmation dynamique

Ce type de programmation se fait en utilisant par exemple un tableau qui à deux dimensions qui va être rempli progressivement par les valeurs des coefficients binomiaux en commençant par les plus petits.
On obtient le programme suivant :

```
def binom(n, p):
    t = np.zeros((n + 1, p + 1), dtype=np.int64)
    for i in range(0, n + 1):
        t[i, 0] = 1
    for i in range(1, p + 1):
        t[i, i] = 1
    for i in range(2, n + 1):
        for j in range(1, min(p, i) + 1):
            t[i, j] = t[i - 1, j - 1] + t[i - 1, j]
    return t[n, p]
```

Le calcul de $\binom{30}{15}$ s'effectue alors en 1.8 ms sur mon ordinateur...
De façon plus général, au prix d'un coup spatial (la création du tableau), la complexité temporelle de cet algorithme est en $O(np)$.

Remarque 2.1 Cette solution n'est pas optimale car de nombreux coefficients non nécessaires sont calculés. On n'a besoin que de ceux coloriés sur le dessin ci-dessous.

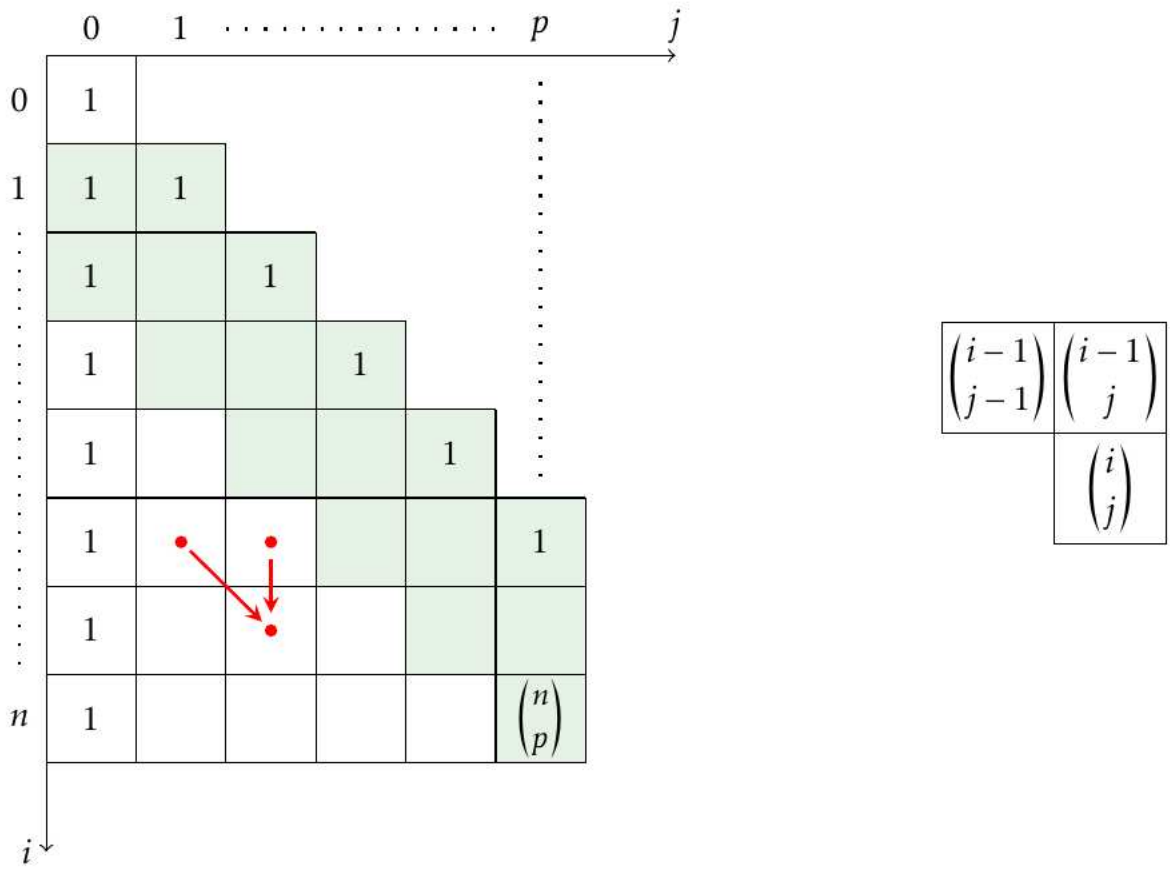


FIGURE 2.4 – Le calcul de $\binom{5}{2}$ avec le programme précédent

Remarque 2.2 Un autre problème, plus important encore, consiste en la perte de lisibilité de l'algorithme. L'idéal serait d'arriver à combiner l'élégance de la programmation récursive à l'efficacité de la programmation dynamique.

La solution est la *mémoïsation*. On va à nouveau utiliser un dictionnaire pour mémoriser les résultats du calculs déjà effectués. A chaque étape de calcul, le programme vérifie s'il n'a pas déjà calculé la valeur demandée et n'effectuera le calcul que dans le cas contraire.
On obtient l'algorithme suivant :

```
binom_dict = {}
def binom(n, p):
    if (n, p) not in binom_dict:
        if p == 0 or n == p:
            b = 1
        else:
            b = binom(n - 1, p - 1) + binom(n - 1, p)
        binom_dict[(n, p)] = b
    return binom_dict[(n, p)]
```

On effectue le calcul de $\binom{5}{2}$ avec cet algorithme :

```
In [1]: binom(5, 2)
Out[1]: 10
In [2]: binom_dict
Out[2]: {(3, 0): 1, (2, 0): 1, (1, 0): 1, (1, 1): 1, (2, 1): 2, (3, 1): 3, (4, 1): 4,
```

```
(2, 2): 1, (3, 2): 3, (4, 2): 6, (5, 2): 10}
```

On retrouve les 10 valeurs nécessaires au calul et rien d'autre.
Pour finir, on peut calculer les coefficients binomiaux avec un algorithme récursif sans mémoïsation, avec un programme à la fois plus court, plus élégant et bien plus rapide... Comment ?

```
def binom(n,p):
    if p==0:
        return(1)
    else:
        return(n/p*binom(n-1,p-1))
```

Le calcul de $\binom{35}{15}$ s'effectue alors en 1.12×10^{-3} ms ! Il y a néanmoins un problème, le résultat renvoyé est un flottant.
Proposons un algorithme qui évite cela :

```
def binom(n,p):
    if p == 0:
        return(1)
    num , den = 1 , 1
    for k in range(p):
        num *= (n-k)
        den *= (p-k)
    return(num//den)
```

Remarque 2.3 La mémoïsation est tellement utile pour résoudre les problèmes de programmation dynamique que cette fonctionnalité est présente dans la librairie standart de Python via le module `lru_cache`. Ce décorateur associe automatiquement un dictionnaire à la définition d'une fonction récursive.

```
from functools import lru_cache

@lru_cache(maxsize = 100)

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

Avec ce programme, le calcul de $\binom{30}{15}$ s'effectue en $1.12ms$.

2.4 Problèmes d'optimisation

La programmation dynamique est fréquemment employée pour résoudre des problèmes d'optimisation : elle s'applique dès lors que la solution optimale peut être déduite des solutions optimales des sous-problèmes (c'est le principe d'optimalité de Bellman, du nom de son concepteur). Cette méthode garantit d'obtenir la meilleure solution au problème étudié, mais dans un certain nombre de cas sa complexité temporelle reste trop importante pour pouvoir être utilisée dans la pratique.
Dans ce type de situation, on se résout à utiliser un autre paradigme de programmation, la programmation gloutonne. Alors que la programmation dynamique se caractérise par la résolution par taille croissante de tous les problèmes locaux, la stratégie gloutonne consiste à choisir à partir du problème global un problème local et un seul en suivant une heuristique (c'est à dire une stratégie permettant de faire un choix rapide mais pas nécessairement optimal).
On ne peut en général garantir que la stratégie gloutonne détermine la solution optimale, mais lorsque l'heuristique est bien choisie on peut espérer obtenir une solution proche de celle-ci. Pour apprécier la différence entre programmation dynamique et programmation gloutonne, nous allons maintenant étudier le problème du sac à dos.

2.5 Le problème du sac à dos

Le problème du sac à dos se pose en les termes suivants :

étant donné n objets de valeurs $c_1,\dots,c_n$ et de poids respectifs $\omega_1,\dots,\omega_n$, comment remplir un sac à dos maximisant la valeur emportée $\sum_{i\in I} c_i$ tout en respectant la contrainte $\sum_{i\in I}\omega_i \leqslant W_{max}$?

2.5.1 Algorithme glouton et heuristique

Pour élaborer un algorithme glouton résolvant le problème, il faut définir une heuristique, ici un critère de priorité pour le choix des objets à prendre. Nous pouvons par exemple choisir en priorité les objets dont la valeur $\dfrac{\text{valeur}}{\text{poids}} = \dfrac{c_i}{\omega_i}$ est maximale, et remplir le sac tant que c'est possible.
Cela donne le programme glouton suivant :

```python
#par algo glouton avec heuristique
print('################\nsac à dos glouton avec heuristique\n################\n')
c=[7,9,3,3]
w=[13,12,8,10]
Wmax = 30

def sac_a_dos_glouton(c, w, Wmax):
    r = sorted(
        [(c[k], w[k]) for k in range(len(c))], key=lambda t: t[0] / t[1], reverse=True
)
    s, p = 0, Wmax
    for k in range(len(c)):
        if r[k][1] <= p:
            s += r[k][0]
            p -= r[k][1]
    return s
A=sac_a_dos_glouton(c,w,Wmax)
print('sac a dos glouton',A)
```

spe_exos_info/hachage/hachage.py

Sa complexité temporelle est un $O(n\ln n)$ à cause du tri.
Pour évaluer la qualité de cette heuristique, j'ai réalisé 1 000 expériences pour chacune desquelles j'ai :

- pris au hasard 100 objets de valeurs comprises entre 1 et 20 et de poids compris entre 1 et 30 ;
- calculé la valeur optimale obtenue pour un poids maximal égal à 300 à la fois par l'algorithme glouton ci-dessus et par l'algorithme dynamique que nous étudierons plus loin.

Sur les 1 000 expériences, 421 ont donné le même résultat pour chacun des deux algorithmes, et dans le cas des 579 autres, l'algorithme glouton a toujours rendu un résultat au moins égal à 98% du résultat de l'algorithme dynamique.
On peut donc considérer ici qu'à défaut de donner un résultat toujours exact, l'algorithme glouton donne un résultat acceptable tout en ayant une complexité moindre que l'algorithme dynamique.

2.5.2 Algorithme dynamique par programmation ascendante

Pour résoudre ce problème, nous allons noter $f(k,W)$ la valeur maximale qu'il est possible d'atteindre avec les k premiers objets pour un poids total égal à W.
Si l'objet d'indice k est dans la solution optimale, alors $w_k \leqslant W$ et $f(k,W) = c_k + f(k-1,W-\omega_k)$; s'il n'y est pas alors $f(k,W) = f(k-1,W)$. On en déduit :

$$f(k,W) = \begin{cases} \max\left(c_k + f(k-1,W-\omega_k), f(k-1,W)\right) & \text{si } \omega_k \leqslant W \\ f(k-1,W) & \text{sinon} \end{cases}$$

Pour calculer cette valeur, nous allons utitiliser un tableau bi-dimensionnel de taille $(n+1)\times(W_{\max}+1)$ destiné à contenir les valeurs de $f(k,\omega)$ pour $k \in [\![0,n]\!]$ et $W \in [\![0,W_{\max}]\!]$.
Nous prendrons comme valeurs initiales $f(0,W) = f(k,0) = 0$, et notre but est de calculer $(n,W_{\max})$.
Pour remplir ce tableau, il est primordial de respecter l'ordre de dépendance des cases de ce tableau : la case $f(k,W)$ ne peut être calculée que lorsque les cases $f(k-1,W)$ et $f(k-1,W-\omega_k)$ auront été remplies.

L'algorithme correspondant est le suivant :

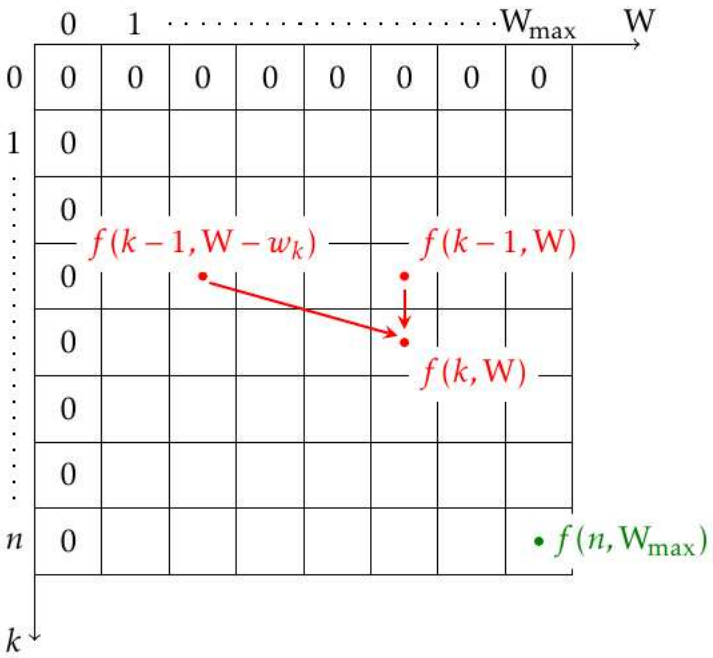


FIGURE 2.5 – Ordre de dépendance du sac à dos.

```python
print('################\nsac à dos dynamique avec prog ascendante\n################\n')
import numpy as np
def sacAdos(c, w, Wmax):
    n = len(c)
    f = np.zeros((n + 1, Wmax + 1), dtype=int)
    for k in range(n):
        for W in range(0, Wmax + 1):
            if w[k] <= W:
                f[k + 1, W] = max(c[k] + f[k, W - w[k]], f[k, W])
            else:
                f[k + 1, W] = f[k, W]
        # print(f)
    return f[n, Wmax]
```

spe_exos_info/hachage/hachage.py

Sa complexité est en $O(nW_{max})$.

2.5.3 Algorithme dynamique avec mémoïsation

On reprend maintenant l'algorithme précédent en utilisant la mémoïsation.

```python
#programmation dynamique avec mémoÃ¯sation
print('################\nsac à dos dynamique avec prog ascendante et mémoÃ¯sation\n################\
    n')
def sac_a_dos(c,w,Wmax):
    dico={}
    def f(k,W):
        if (k,W) not in dico:
            if k == 0 or W == 0:
                x=0
            elif w[k-1] <=W:
                x=max((c[k - 1] + f(k - 1, W - w[k - 1]), f(k - 1, W)))
            else:
                x=f(k-1,W)
            dico[(k,W)]=x
        return(dico[(k,W)])
    return(f(len(c),Wmax))
```

spe_exos_info/hachage/hachage.py

2.6 Plus longue sous-suite commune

2.6.1 Position du problème

Pour une suite finie $x = (x_0, \ldots, x_{m-1})$ formée d'éléments d'un méme ensemble (cela pourrait étre une chaîne de caractéres ou une liste), et une suite $y = (y_0, ..., y_{n-1})$, on cherche la plus longue sous-suite commune (PSLC) à x et y, ie la plus longue suite extraite commune à x et y.

Exemple 2.1
X = 'q**ue j**'aime à fai**re** a**pp**rendre'

Z = 'uejrepp' est une sous-suite de X.

Exemple 2.2 PLSC
X = 'q**ue** j'ai**me** à f**a**ire **appre**ndre'
Y = '**u**n nombre ut**ile** **a**ux s**age**s'
Z = 'uieaae' est une sous-suite commune à X et Y.

Exemple 2.3
X = [1,2,3,4,5,7]

Y = [2,3,7,9,10]
Z = [2,3,7] est la plus longue sous suite commune à X et Y.

2.6.2 Résolution par force brute

On pourrait construire la liste de toutes les sous-suites de X, puis celle de toutes les sous-suite de Y et enfin déterminer l'élément maximal de l'intersection de ces deux listes.
Commençons par créer la liste de toutes les sous-séquence d'une liste donnée :

```
def liste_sous_sequences(X:list, L:list=[],l:list=[])->list:
    """
    X une liste
    renvoie la liste de toutes les sous-séquence de X
    """
    if X == []:
        L.append(l)
        return(L)
    else:
        liste_sous_sequences(X[:-1],L,l)
        return(liste_sous_sequences(X[:-1],L,[X[-1]]+l))
```

Pour calculer l'intersection, on peut procéder ainsi :

```
def intersection(K:list,L:list)->list:
    """
    K,L: listes de listes
    renvoie l'intersection de deux listes
    """
    I = []
    for l in L:
        if l in K:
            I.append(l)
    return(I)
```

La PSLC se calcule alors ainsi :

```
def PLSC(K:list,L:list)->list:
    """
    K,L deux listes
    renvoie la PLSC de K et L
    """
    sol = []
```

```
    for l in L:
        if l in K and len(l) > len(sol):
            sol = l
    return(sol)
```

2.6.3 Mise en évidence de sous-problème vérifiant la propriété de sous-structure optimale.

On va mettre en évidence la nature récursive de la résolution du problème.

✎ *Notation 2.4* Pour une séquence $X = (x_0, \dots, x_n)$ et $i \in [\![1, n]\!]$, on note X_i la séquence construite en conservant les i premiers éléments de X :

$$X_i = (x_0, \dots, x_{i-1}).$$

On posera par ailleurs $X_0 = \varnothing$.

Remarque 2.4 La longueur de X_i est i.

THÉORÈME 2.1 ★ Sous-structure optimale d'une PLSC
Soit $X = (x_0, \dots, x_{m-1})$ et $Y = (y_0, \dots, y_{n-1})$. Soit encore $Z = (z_0, \dots, z_{p-1})$ une PLSC de X et Y . Alors :

1. Si $x_{m-1} = y_{n-1}$ alors $z_{p-1} = x_{m-1} = x_{n-1}$ et Z_{p-1} est une PLSC de X_{m-1} et Y_{n-1}.
2. Si $x_{m-1} \neq y_{n-1}$. Alors $z_{p-1} \neq x_{m-1}$ implique que Z est une PLSC de X_{m-1} et Y .
3. Si $x_{m-1} \neq y_{n-1}$. Alors $z_{p-1} \neq y_{n-1}$ implique que Z est une PLSC de X et Y_{n-1} .

Démonstration

1. *Si $x_{m-1} = y_{n-1}$. Alors x_{m-1} et y_{n-1} sont communs à X et Y . Par suite $z_{p-1} = x_{m-1} = x_{n-1}$. Comme $Z = (z_0, \dots, z_{p-1})$ est une SSC à X et Y alors $Z_{p-1} = (z_0, \dots, z_{p-2})$ est une SSC à X_{m-1} et Y_{n-1} . Si, par l'absurde, Z_{p-1} n'était pas une PLSC à X_{m-1} et Y_{n-1} alors il existerait a tel que $Z' = (z_0, \dots, z_{p-2}, a)$ soit une SSC à X_{m-1} et Y_{n-1}. Ainsi $Z'' = (z_1, \dots, z_{p-1}, a, z_p)$ serait une SSC à X et Y . Ceci est absurde car Z'' est strictement plus grand que Z : cela contredit la maximalité de Z. Par suite Z_{p-1} est une PLSC à X_{m-1} et Y_{n-1}.*
2. *Si $z_{p-1} \neq x_{m-1}$. Alors pour tout $i \in [\![0, p-1]\!]$, $z_i \neq x_{m-1}$ (sinon $\exists i_0 \in [\![0, p-1]\!]$ tel que $z_{i_0} = x_{m-1}$; nécessairement, comme x_{m-1} est le dernier élément de X alors $i_0 = p-1$ et $z_{p-1} = x_{m-1}$: absurde). Comme Z est une sous-séquence de X, alors Z est une sous-séquence de X_{m-1}. Ainsi Z est une SSC à X_{m-1} et Y. Il reste à montrer que Z est maximale comme SSC à X_{m-1} et Y. Si, par l'absurde, ce n'était pas le cas alors il existerait a tel que $Z' = (z_0, \dots, z_{p-1}, a)$ soit une SSC à X_{m-1} et Y . En particulier, Z serait une SSC à X et Y strictement plus grande que Z. Cela contredit la maximalité de Z. Ainsi Z est une PLSC à X_{m-1} et Y .*
3. *Raisonner comme en 2 en permutant les rôles de X et Y.*

Cela conduit à la récurrence suivante :
Soit Z(X,Y) la PLSC des suites X et Y et X′ et Y′ les suites X et Y privées de leur dernier élément, respectivement.

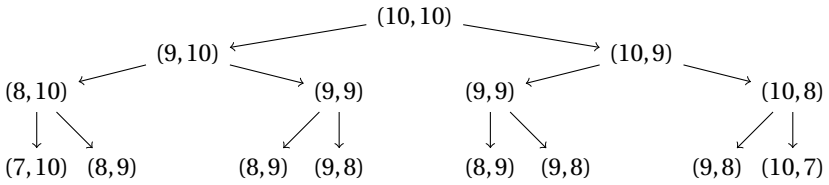
$$Z(X,Y) = \begin{cases} \varnothing & \text{si } X = \varnothing \text{ ou } Y = \varnothing \\ Z(X',Y') \bigcup \{x_N\} & \text{si } x_N = y_M \\ PLS(Z(X',Y), Z(X,Y')) & \text{sinon.} \end{cases}$$

2.6.4 Mise en place de l'algorithme naïf et complexité

On écrit la fonction Python selon la récurrence précédente.

```
def Z(X,Y):
    if len(X)==0 or len(Y)==0:
        return ''
    elif X[-1]==Y[-1]:
        return Z(X[:-1],Y[:-1])+X[-1]
    else:
        A,B=Z(X,Y[:-1]),Z(X[:-1],Y)
        if len(A)>len(B):
            return A
        else:
            return B
```

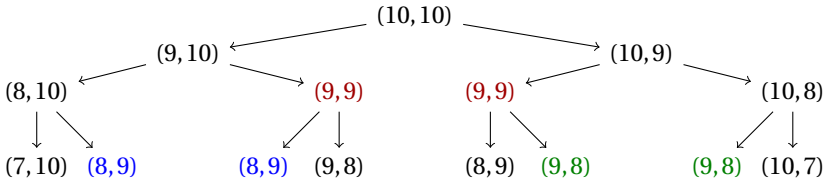
Prenons le cas N = len(X) = 10 et M = len(Y) = 10. Dans le pire des cas,



Posons C(N,M) la complexité en temps, dans le pire des cas.
D'après la relation de récurrence, ⟦C(N,M)=C(N-1,M)+C(N,M-1).⟧
On vérifie que $2^{N+M} = 2^{N-1+M} + 2^{N+M-1}$ d'où ⟦$\boxed{C(N,M) = 2^{N+M}}$⟧ soit une **complexité exponentielle**.

2.6.5 Chevauchement des solutions

Il apparaît dans l'arbre des positions identiques, dont le résultat doit être calculé plusieurs fois : à nouveau on a un chevauchements de sous-problèmes.



On va y remédier par la mémoïsation, c'est-à-dire le stockage de la valeur calculée.

```python
def ZZ(X,Y):
    n,m=len(X),len(Y)
    dic={}
    for i in range(n+1):
        dic[(i,0)]=''
    for j in range(m+1):
        dic[(0,j)]=''
    for i in range(1,n+1):
        for j in range(1,m+1):
            if X[i-1]==Y[j-1]:
                dic[(i,j)]=dic[(i-1,j-1)]+X[i-1]
            else:
                if len(dic[(i-1,j)])>len(dic[(i,j-1)]):
                    dic[(i,j)]=dic[(i-1,j)]
                else:
                    dic[(i,j)]=dic[(i,j-1)]
    return dic[(n,m)]
```

2.7 Algorithme de Floyd-Warshall

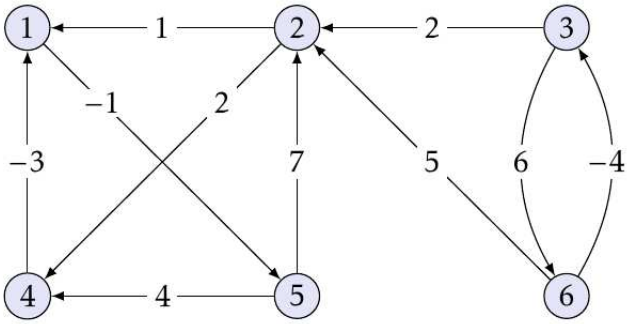
2.7.1 Description de l'algorithme

Le dernier algorithme de programmation dynamique que nous allons étudier concerne les graphes. Nous allons considérer ici un graphe orienté et pondéré : autrement dit, les arêtes sont orientées, et chaque arête possède un poids, ici un entier relatif. Une façon de représenter en machine un tel graphe consiste à numéroter les sommets et à considérer une matrice $M \in \mathfrak{M}_n(\mathbb{Z})$, pour laquelle le coefficient $m_{i,j}$ sera égal au poids de l'arête reliant les sommets v_i à v_j , avec la convention que ce poids est égal à $+\infty$ s'il n'y a pas d'arête reliant v_i à v_j (illustration figure **??**.)

fw

L'algorithme de Floyd-Warshall a pour objet de calculer, pour chacun des couples de points (v_i, v_j), le chemin de poids minimal reliant v_i à v_j, s'il existe. Pour ce faire, cet algorithme calcule la suite finie de matrices $M^{(k)}$, $0 \leqslant k \leqslant n$ avec $M^{(0)} = M$ et :

$$\forall k < n, \quad \forall (i,j) \in [\![1,n]\!]^2, \quad m_{ij}^{(k+1)} = \min\left(m_{ij}^{(k)}, m_{ik+1}^{(k)} + m_{k+1j}^{(k)}\right).$$



$$M = \begin{pmatrix} 0 & +\infty & +\infty & +\infty & -1 & +\infty \\ 1 & 0 & +\infty & 2 & +\infty & +\infty \\ +\infty & 2 & 0 & +\infty & +\infty & 6 \\ -3 & +\infty & +\infty & 0 & +\infty & +\infty \\ +\infty & 7 & +\infty & 4 & 0 & +\infty \\ +\infty & 5 & -4 & +\infty & +\infty & 0 \end{pmatrix}$$

FIGURE 2.6 – Un exemple de graphe pondéré et sa matrice d'adjacence.

THÉORÈME 2.2 ★
Si G ne contient pas de cycle de poids strictement négatif, alors $m_{ij}^{(k)}$ est égal au poids du chemin minimal reliant v_i à v_j en ne s'autorisant de passer que par des sommets parmi $v_1, v_2, \ldots, v_k$.
De ceci, il découle immédiatement que $m_{ij}^{(n)}$ est le poids minimal d'un chemin reliant v_i à v_j.

Démonstration *Raisonnons par récurrence sur k.*

- *Si $k = 0$, $m_{ij}^{(0)} = m_{ij}$ est bien entendu le poids du chemin minimal reliant v_i à v_j sans passer par aucun autre sommet.*
- *Si $k < n$, supposons le résultat acquis au rang k et considérons un chemin $v_i \rightsquigarrow v_j$ ne passant que par les sommets $v_1, \ldots, v_{k+1}$ et de poids minimal.*
  Si ce chemin ne passe pas par v_{k+1}, son poids total est par hypothèse de récurrence égal à $m_{ij}^{(k)}$.
  Si ce chemin passe par v_{k+1} , alors par principe de sous-optimalité les chemins $v_i \rightsquigarrow v_{k+1}$ et $v_{k+1} \rightsquigarrow v_j$ sont minimaux et ne passent que par des sommets de la liste v_1 ,… , v_k donc par hypothèse de récurrence son poids total est égal à $m_{ik+1}^{(k)} + m_{k+1j}^{(k)}$ (car il n'existe pas de cycle de poids négatif reliant v_{k+1} à lui-même).
  De ceci, il résulte que $\min\left(m_{ij}^{(k)}, m_{ik+1}^{(k)} + m_{k+1j}^{(k)}\right) = m_{ij}^{(k+1)}$ est le poids minimal d'un plus court chemin reliant v_i et v_j et ne passant que par des sommets de la liste $v_1, \ldots, v_{k+1}$.

2.7.2 Mise en œuvre pratique

On écrira en TP l'algorithme de Floyd-Warshall.
Appliqué à la matrice précédente, on obtient la matrice :

$$\begin{pmatrix} 0 & 6 & +\infty & 3 & -1 & +\infty \\ -1 & 0 & +\infty & 2 & -2 & +\infty \\ 1 & 2 & 0 & 4 & 0 & 6 \\ -3 & 3 & +\infty & 0 & -4 & +\infty \\ 1 & 7 & +\infty & 4 & 0 & +\infty \\ -3 & -2 & -4 & 0 & -4 & 0 \end{pmatrix}$$

On observe que les sommets 3 et 6 ne sont pas accessibles à partir des sommets 1, 2, 4 et 5. En revanche, on peut aller du sommet 6 au sommet 1 pour un coût total égal à -3 (il n'est pas difficile de deviner qu'il faut passer par les sommets 3, 2 et 4) ou du sommet 3 au sommet 5 pour un coût total nul (en passant par les sommets 2, 4 et 1).

Remarque 2.5 De manière évidente la complexité temporelle de l'algorithme de Floyd-Warshall est en O(n3), où n est l'ordre du graphe et la complexité spatiale en O(n2).

2.7.3 Application au calcul de la fermeture transitive d'un graphe

Considérons de nouveau un graphe non pondéré, orienté ou non. Le problème de la fermeture transitive consiste à déterminer si deux sommets a et b peuvent être reliés par un chemin allant de a à b. Pour le résoudre, nous allons utiliser la matrice d'adjacence associée à ce graphe, mais cette fois en utilisant les valeurs booléennes `True` pour dénoter l'existence d'une arête et `False` pour en marquer l'absence. Remplaçons maintenant dans l'algorithme de Floyd-Warshall la relation de récurrence sur les coefficients des matrices M(k) par :

$$m_{ij}^{(k+1)} = m_{ij}^{(k)} \quad \text{ou} \quad \left(m_{ik+1}^{(k)} \quad \text{et} \quad m_{k+1j}^{(k)}\right)$$

Il n'est pas difficile de prouver que le booléen $m_{ij}^{(k)}$ dénote l'existence d'un chemin reliant les sommets v_i et v_j en ne passant que par les sommets v_1 , v_2, … , v_k et que par voie de conséquence la matrice $M^{(n)}$ résout le problème de la fermeture transitive.
L'algorithme ainsi modifié est connu sous le nom d'algorithme de Warshall.