Chapitre 3

Programmation dynamique et mémoïsation

En informatique, la programmation dynamique est une méthode algorithmique pour résoudre des problèmes d'optimisation. Le concept a été introduit au début des années 1950 par Richard Bellman. À l'époque, le terme « programmation » signifie planification et ordonnancement. La programmation dynamique consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires. Elle a d'emblée connu un grand succès, car de nombreuses fonctions économiques de l'industrie étaient de ce type, comme la conduite et l'optimisation de procédés chimiques, ou la gestion de stocks.

I. Programmation dynamique

1. Etude de la suite de Fibonacci

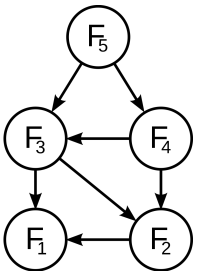
Exercice 3.1

- Ecrire une fonction `FiboNaif(n)` utilisant la récursivité permettant de calculer le n-ème terme de la suite de Fibonacci
- Quel est le nombre total d'appel à la fonction `FiboNaif` pour le calcul de `FiboNaif(4)`. Et plus généralement pour le calcul de `FiboNaif(n)` ?

Ce programme a donc une complexité exponentielle. Cela s'explique par le très grand nombre d'appels redondants : ainsi `FiboNaif(4)` appelle 2 fois `FiboNaif(2)`, 3 fois `FiboNaif(1)` et 2 fois `FiboNaif(0)` alors que leurs résultats respectifs seront toujours les mêmes.
Le calcul de `FiboNaif(10)` nécessite 177 appels et celui de `FiboNaif(50)`, 21891.

2. Principe de la programmation dynamique

La programmation dynamique s'appuie sur le principe d'optimalité de Bellman : une solution optimale d'un problème s'obtient en combinant des solutions optimales à des sous-problèmes. Sur l'exemple de la suite de Fibonacci, la solution F_n s'obtient en additionnant F_{n-1} et F_{n-2}.
La méthode de programmation dynamique, comme la méthode *diviser pour régner*, résout des problèmes en combinant des solutions de sous-problèmes. Les algorithmes diviser pour régner partitionnent le problème en sous-problèmes indépendants qu'ils résolvent récursivement, puis combinent leurs solutions pour résoudre le problème initial. La méthode diviser pour régner est inefficace si on doit résoudre plusieurs fois le même sous-problème. On a vu par exemple que l'algorithme ci-dessus est inefficace.



Le graphe de dépendance ci-dessus n'est pas un arbre : cela illustre que les sous-problèmes se chevauchent.
Il existe alors deux méthodes pour calculer effectivement une solution :

1. **la méthode ascendante : (bottom-up)** : on commence par calculer des solutions pour les sous-problèmes les plus petits, puis, de proche en proche, on calcule les solutions des problèmes en utilisant le principe d'optimalité et on mémorise les résultats dans un tableau.

```python
def fibo_asc(n):
    mem = [0]*(n+1)
    mem[1] = 1
    for i in range(2, n+1):
        mem[i] = mem[i-1] + mem[i-2]
    return mem[n]
```

2. **la méthode descendante (top-down)** : Dans la méthode descendante, on écrit un algorithme récursif mais on utilise la mémoïsation pour ne pas résoudre plusieurs fois le même problème, c'est-à-dire que l'on stocke dans un tableau ou dictionnaire les résultats des appels récursifs :

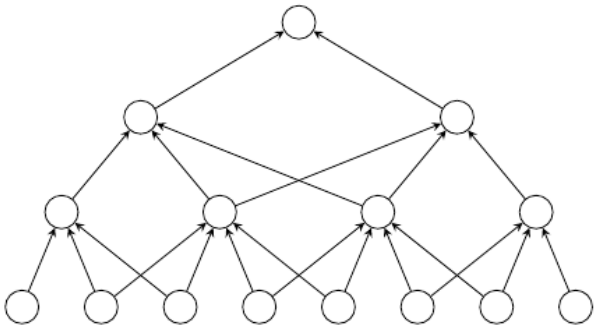
```python
d = {}
def fibo_desc(n):
    if n not in d:
        if n == 0:
            d[n] = 0
        elif n == 1:
            d[n] = 1
        else :
            d[n] = fibo_desc(n-1) + fibo_desc(n-2)
    return d[n]
```

Exercice 3.2

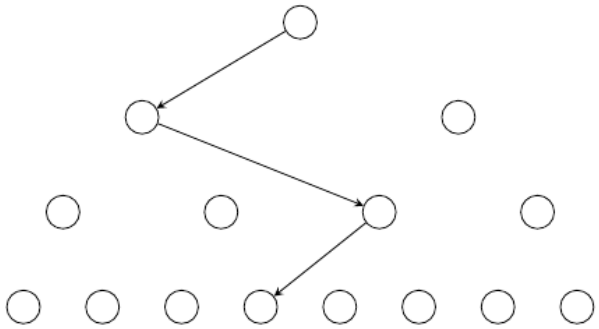
1. Écrire une fonction permettant le calcul $\binom{n}{p}$, en utilisant la programmation récursive et la formule de Pascal.
2. Améliorer ce programme en utilisant l'une des deux méthodes précédentes.

II. Algorithmes gloutons

Tout comme les problèmes que l'on résout par une méthode « diviser pour régner », les problèmes que l'on résout par la programmation dynamique se ramènent à la résolution de sous-problèmes de tailles inférieures. Mais à la différence de la méthode « diviser pour régner », ces sous-problèmes ne sont pas indépendants, ce qui impose d'accompagner la programmation récursive par une analyse fine des relations de dépendance, ou beaucoup plus simplement par l'utilisation de la mémoïsation qui gère les relations de dépendance à notre place.



programmation dynamique



programmation gloutonne

1. 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 (on a vu que 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 suivant.

2. Problème du sac à dos

Problème :
Étant donnés n objets de valeurs $c_1, c_2, ..., c_n$ et de poids respectifs $w_1, ..., w_n$, comment remplir un sac à dos maximisant la valeur emportée $\sum_{i\in \mathrm{I}} c_i$ tout en respectant la contrainte $\sum_{i\in \mathrm{I}} w_i \leqslant \mathrm{W}_{max}$?

3. Solution gloutonne

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 le rapport $\frac{\text{valeur}}{\text{poids}}$ est maximal, et remplir le sac tant que c'est possible.

> **Exercice 3.3**
> Ecrire une fonction qui utilise un algorithme glouton pour résoudre ce problème.

4. Solution dynamique

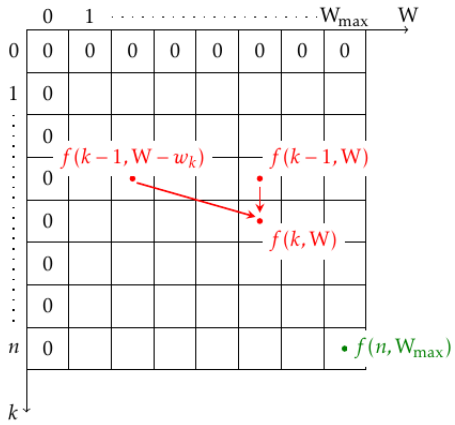
Pour résoudre ce problème, nous allons noter $f(k,\mathrm{W})$ la valeur maximale qu'il est possible d'atteindre avec les k premiers objets pour un poids total égal à W.

- Si l'objet d'indice k est dans la solution optimale, alors $w_k \leqslant \mathrm{W}$ et $f(k,\mathrm{W}) = c_k + f(k-1,\mathrm{W}-w_k)$;
- s'il n'y est pas alors $f(k,\mathrm{W}) = f(k-1,\mathrm{W})$

On en déduit :

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

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



> **Exercice 3.4**
> Construire le tableau correspondant aux objets définis par `c = [1, 6, 18, 22, 28]`, `w=[1, 2, 5, 6, 7]` et `Wmax = 11`

a. Solution Bottom-up

:

```
def sac_a_dos_dyn(c, w, Wmax):
    n = len(c)
    mem = [[0]*(Wmax+1) for i in range(len(c)+1)]
    for k in range(1, n+1):
        for W in range(Wmax+1):
            if w[k] <= W:
                mem[k][W] = max(c[k]+mem[k-1, W-w[k]], mem[k-1, w])
            else :
                mem[k][w] = mem[k-1, w]
    return mem[n, Wmax]
```

> **Remarque**
> Il apparaît clairement que la complexité temporelle de cet algorithme est proportionnel au

produit $n.\mathrm{W}max$, soit en $\mathrm{O}(n\mathrm{W}max)$. L'algorithme glouton quant à lui est en $\mathrm{O}(nln(n))$ (le coût du tri).

b. Solution par mémoïsation

: La technique de mémoïsation nous permet de moins nous préoccuper de l'ordre de dépendance qui est géré par la récursivité :

```
def sac_a_dos_dyn(c, w, Wmax):
    n = len(c)
    mem = [[0]*(Wmax+1) for i in range(len(c)+1)]
    for k in range(1, n+1):
        for W in range(Wmax+1):
            if w[k] <= W:
                mem[k][W] = max(c[k]+mem[k-1, W-w[k]], mem[k-1, w])
            else :
                mem[k][w] = mem[k-1, w]
    return mem[n, Wmax]
```

c. Reconstruction d'une solution optimale

Cet algorithme calcule la valeur maximale qui peut être emportée dans le sac, mais pas la façon d'y parvenir. Pour la connaître il faut utiliser le tableau (ou le dictionnaire) calculé par la fonction précédente, et retrouver le chemin qui mène de la case initiale à la case finale.
Par exemple, si on modifie la fonction non récursive (la première) pour qu'elle renvoie le tableau f qui a été calculé au lieu de la valeur `f[len(c), Wmax]`, la fonction qui détermine les objets à choisir s'écrira :

```
def objetsAchoisir(c, w, Wmax):
    f = sacAdos(c, w, Wmax)
    sac = []
    k, W = len(c), Wmax
    while k > 0:
        if f[k, W] > f[k-1, W]:
            sac.append((c[k-1], w[k-1]))
            W -= w[k-1]
        k -= 1
    return sac
```