Programmation dynamique

Cours et correction des activités

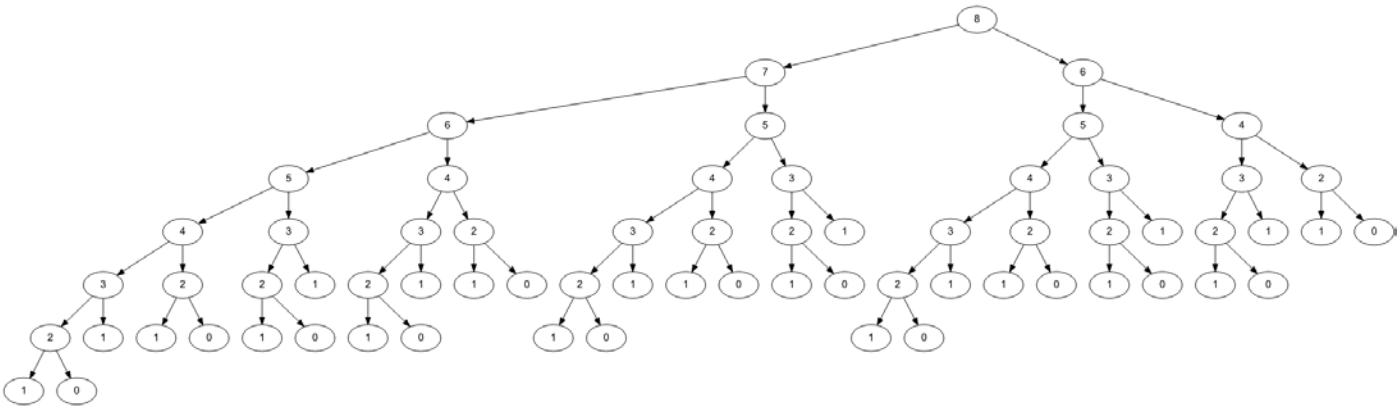
1. Une introduction avec la suite de Fibonacci

La suite de Fibonacci est définie par $F_0 = 0$, $F_1 = 1$ et pour tout entier naturel $n \geq 2$, $F_n = F_{n-1} + F_{n-2}$.
Cette définition se traduit naturellement en la fonction fibo qui calcule les termes de cette suite :

```
def fibo(n):
    if n < 2:
        return n
    else:
        return fibo(n-1) + fibo(n-2)
```

Le problème des appels récursifs redondants

Les appels augmentent de manière exponentielle comme on peut le voir dans l'arbre des appels de fibo(8).



Chaque appel est en temps constant, et la complexité de la fonction est un grand O du nombre d'appels.
En notant a_n le nombre d'appels récursifs effectués par fibo(n), on démontre (niveau licence ou prépa) que $a_n = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$. C'est bien une complexité exponentielle.

La mémoïsation : approche top-down

Une technique efficace, dite mémoïsation, consiste à stocker le résultat des calculs ce qui évite tous les appels redondants qui découlent des recalculs.

```
F = [-1]*(100)

def fibo_mem(n):
    if n < 2:
        return n
    else:
        if F[n] < 0:
            F[n] = fibo_mem(n-1) + fibo_mem(n-2)
        return F[n]
```

On peut bien sûr encapsuler la création de la liste dans une fonction fibo.

```
def fibo(n):
    def fibo_mem(n):
        if n < 2:
            return n
        else:
            if F[n] < 0:
                F[n] = fibo_mem(n-1) + fibo_mem(n-2)
            return F[n]

    F = [-1]*(n+1)
    return fibo_mem(n)
```

Certains langages comme Python peuvent faire automatiquement cette mémoïsation (hors-programme).

```
from functools import lru_cache
@lru_cache(maxsize=None)
def fibo(n):
    if n < 2: return n
    return fibo(n-1) + fibo(n-2)
```

@lru_cache est un décorateur, une fonction s'appliquant à une autre fonction pour modifier son comportement. La fonction lru_cache exécute la fonction appelée, ici fibo, mais en mémorisant les calculs déjà effectués afin de ne pas les refaire. La bibliothèque functools contient des fonctions d'ordre supérieur, c'est-à-dire des fonctions s'appliquant à d'autres fonctions.

L'approche bottom-up

Une stratégie très similaire consiste à remplacer les appels récursifs par une boucle et à stocker les valeurs calculées dans un tableau.

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

En réalité, la suite de Fibonacci n'est pas un vrai problème de programmation dynamique puisqu'il suffit de conserver les deux dernières valeurs calculées :

```
def fibo(n):
    f0 = 0
    f1 = 1
    for i in range(1, n+1):
        f0, f1 = f1, f0 + f1
    return f0
```

On peut imaginer une suite un peu plus complexe où la programmation dynamique se justifierait davantage, une suite définie par $F_0 = 0$, $F_1 = 1$ et pour tout entier naturel $n \geq 2$, $F_n = F_{n-1} + F_{n-2} + F_{n//2}$.

```
def fibo2022(n):
    F = [0]*(n+2)
    F[1] = 1
    for i in range(2, n+1):
        F[i] = F[i-1] + F[i-2] + F[i//2]
    return F[n]
```

2. Principes de la programmation dynamique

Le concept de programmation dynamique a été introduit en 1950 par Richard Bellman. À l'époque, le terme programmation signifiait planification et ordonnancement. Ainsi, la programmation dynamique n'est pas une méthode de programmation (comme la programmation objet, la programmation fonctionnelle…) mais une méthode algorithmique.

Certains problèmes, principalement des problèmes d'optimisation, admettent une résolution récursive qui engendre un grand nombre de calculs répétés et inutiles. On opte alors pour un algorithme de programmation dynamique, qui consiste à conserver les résultats des calculs dans un tableau ou un dictionnaire. On reconnait ces problèmes à la propriété de sous-structure optimale et au caractère de chevauchement de sous-problèmes.

2.1. Propriété de sous-structure optimale et principe d'optimalité de Bellman

La programmation dynamique s'appuie sur le principe d'optimalité de Bellman : on peut construire une solution optimale du problème à partir de solutions optimales de sous-problèmes. On dit alors que le problème présente une sous-structure optimale.

La recherche du plus court chemin dans un graphe a cette propriété. En effet, si on considère le plus court chemin C allant d'un sommet D à un sommet A, alors pour tous sommets S et T de C, le chemin C contient un plus court chemin de S à T.

Lorsqu'un problème présente cette propriété, on peut généralement le résoudre par programmation dynamique ou avec un algorithme glouton.

Un algorithme glouton est un algorithme construisant une solution à un problème étape par étape en choisissant à chaque étape la solution localement optimale. Si la solution optimale à un problème s'obtient à partir d'une solution optimale à un sous-problème et d'un choix glouton, alors on optera pour l'algorithme glouton, plus léger que la programmation dynamique.

<u>Exemple du problème du rendu de monnaie</u>
Étant donné un système de monnaies, comment rendre une somme de manière optimale, c'est-à-dire avec un minimum de pièces ?
Un algorithme glouton consiste à choisir à chaque fois la pièce (ou billet) la plus grande inférieure à la somme restante. Avec les grands systèmes de monnaie en vigueur dans le monde, dits systèmes canoniques, cet algorithme renvoie la solution optimale, mais ce n'est pas le cas avec l'ancien système monétaire britannique par exemple.
On recourt alors à la programmation dynamique. Voir exercices.

L'algorithme de programmation dynamique s'appuie sur les formules de récurrence permettant de résoudre le problème à *i* variables à partir des solutions des problèmes à *i* - 1 variables.
On résout successivement les différentes instances du problème en faisant croitre le nombre de variables.

2.2. Chevauchement de sous-problèmes

La programmation dynamique est pertinente si l'algorithme récursif rencontre souvent les mêmes sous-problèmes. La programmation dynamique réduit alors le temps de calcul en évitant de refaire les mêmes calculs plusieurs fois.

2.3. Deux approches

Approche top-down, récursive

On part du problème principal et on rappelle la fonction sur les sous-problèmes, en mémoïsant les résultats, généralement dans un tableau ou un dictionnaire, afin de ne pas effectuer les calculs plusieurs fois. Cette approche permet de n'effectuer que les calculs nécessaires à la résolution du problème principal mais peut être plus lente à cause de la gestion de la récursivité.

Approche bottom-up, itérative

On commence par résoudre tous les sous-problèmes les plus simples et on « monte » vers le problème principal. Organiser l'algorithme de programmation dynamique nécessite de déterminer les relations de récurrence et l'ordre des calculs.

2.4. Reconstruction d'une solution optimale à partir de l'information calculée

Afin de reconstruire la solution optimale à partir des solutions des sous-problèmes, on doit parfois enregistrer les solutions de chaque sous-problème, mais ce n'est pas obligatoire dans toutes les situations. Dans le problème du sac à dos, on dispose d'un sac à dos à contenance limitée, et on doit sélectionner des objets dans une liste afin de maximiser la valeur des objets dans le sac. Une solution par programmation dynamique consiste à résoudre tous les problèmes similaires en faisant croitre la liste d'objets et la contenance du sac à dos. On pourrait conserver pour chaque problème la liste des objets à sélectionner, mais ce n'est pas nécessaire. Il suffit de stocker à chaque étape la valeur maximale dans le sac à dos. On peut ensuite reconstruire la solution optimale à partir de ces informations.

2.5. Complexité en mémoire

Une résolution par programmation dynamique nécessite généralement de mémoïser un grand nombre de résultats. On pensera à estimer les besoins en mémoire de ses codes.

2.6. La programmation dynamique en pratique

1) On reconnait un problème que l'on peut résoudre par programmation dynamique et on cherche le lien entre la solution d'un problème et celles de sous-problèmes (principe récursif, relation de récurrence)
2) On choisit l'approche, top-down ou bottom-up
3) On écrit l'algorithme ou le code qui détermine la valeur optimale
4) On reconstitue si besoin la solution correspondant à cette valeur

3. Résolution du problème du sac à dos

On dispose de n objets assimilables à des couples (valeur, poids) et d'un sac à dos qui peut porter un poids maximum w. L'objectif est de maximiser la valeur des objets contenus dans le sac.

Données : n couples (v_i, p_i), un nombre w
Problème : sélectionner un sous-ensemble de couples tels que $\Sigma p_i \leqslant w$ et Σv_i est maximale.

Pour tout entier naturel i strictement inférieur à n, et tout entier j inférieur ou égal à w, on note $t_{i,j}$ la valeur maximale réalisable en utilisant les i premiers objets avec un poids inférieur ou égal à j.

Le principe récursif à la base de la résolution par programmation dynamique est le suivant.
Pour construire une solution optimale à $i + 1$ objets, on prend le meilleur des deux cas suivants :

- La solution optimale n'inclut pas le dernier objet, auquel cas $t_{i,j} = t_{i-1,j}$
- La solution optimale inclut le dernier objet (v_{i-1}, p_{i-1}) et les objets précédents sont à choisir avec un poids maximal de $j - p_{i-1}$. Alors, $t_{i,j} = v_{i-1} + t_{i-1,j-p_{i-1}}$

3.1. Approche top-down

On cherche dans un premier temps une solution renvoyant la valeur maximale, sans la liste d'objets.
On commence par la **résolution récursive, sans mémoïsation**.

```
def sac_a_dos_rec(objets, w, n=None):
    if n == None:        # facultatif, permet juste d'éviter de passer
        n = len(objets)  # la valeur de n au premier appel
    if n == 0:
        return 0
    v, p = objets[n-1]   # dernier objet, qu'on prend ou pas
    val_sans_dernier_objet = sac_a_dos_rec(objets, w, n-1)
    if p > w:
        return val_sans_dernier_objet
    val_avec_dernier_objet = sac_a_dos_rec(objets, w - p, n-1) + v
    return max(val_avec_dernier_objet, val_sans_dernier_objet)
```

Comme dans le cas de Fibonacci, les temps de calcul augmentent de manière exponentielle avec *n*.
D'où la nécessité de **mémoïser** les résultats $t_{i,j}$ pour ne pas effectuer plusieurs fois les mêmes calculs. On peut utiliser un **tableau** ou un **dictionnaire**.
Puisque les différents problèmes à résoudre sont identifiées par un couple d'entiers (*i*, *j*), on a préféré l'utilisation d'un tableau à deux dimensions, structure plus légère qu'un dictionnaire.

```
def sac_a_dos_mem_val(objets, w, n=None, tab=None):
    if n == None:                                # facultatif : n et
        n = len(objets)                          # tab peuvent être
    if tab == None:                              # définis en dehors
        tab = [[-1]*(w+1) for i in range(n+1)]   # de la fonction.
    if n==0:
        return 0
    if tab[n][w] >= 0:
        return tab[n][w]
    v, p = objets[n-1]   # dernier objet, qu'on prend ou pas
    val_sans_dernier_objet  = sac_a_dos_mem_val(objets, w, n-1, tab)
    if p > w:
        tab[n][w] = val_sans_dernier_objet
        return val_sans_dernier_objet
    val_pour_ajout_dernier_objet = sac_a_dos_mem_val(objets, w - p, n-1, tab)
    val_avec_dernier_objet = val_pour_ajout_dernier_objet + v
    if val_avec_dernier_objet > val_sans_dernier_objet:
        tab[n][w] = val_avec_dernier_objet
        return val_avec_dernier_objet
    else:
        tab[n][w] = val_sans_dernier_objet
        return val_sans_dernier_objet
```

Récupération de la liste des objets

Première méthode

On stocke dans le tableau tab, en plus de chaque valeur, la liste des objets correspondants.
Le code a alors un plus grand besoin en mémoire et la recopie de la liste des objets à chaque étape ralentit le code.

Deuxième méthode, plus efficace

On reconstitue la liste des objets après avoir complété le tableau tab. Cette opération est en temps linéaire par rapport au nombre d'objets et la complexité globale est inchangée.

```
def sac_a_dos_mem_final(objets, w):
    n = len(objets)
    tab = [[-1]*(w+1) for i in range(n+1)]
    tab[0] = [0]*(w+1)
    vmax = sac_a_dos_mem_val(objets, w, n, tab)
    j = w
    contenuSac = []
    for i in range(n, 0, -1):
        if tab[i - 1][j] != tab[i][j]:
            v, p = objets[i - 1]
            contenuSac.append((v, p))
            j = j - p
    return vmax, contenuSac
```

3.2. Approche bottom-up

On crée un tableau tab à (n + 1) lignes et (w + 1) colonnes que l'on va remplir ligne par ligne de haut en bas, et de gauche à droite, à l'aide des formules de récurrences.

```
def sac_a_dos_dyn_tab(objets,w):
    n = len(objets)
    tab = [[0]*(w + 1) for i in range(n + 1)]
    for i in range(n):
        for j in range(w + 1):
            v, p = objets[i]
            if j >= p:
                tab[i + 1][j] = max(tab[i][j], tab[i][j-p] + v)
            else:
                tab[i + 1][j] = tab[i][j]
    return tab
```

Récupération de la liste des objets

Comme dans la version top-down, on peut mémoriser les listes d'objets réalisant chaque valeur du tableau tab ou reconstituer la liste d'objets optimale à la fin.

```
def sac_a_dos_dyn(objets, w):
    tab = sac_a_dos_dyn_tab(objets, w)
    n = len(tab) - 1      # égal à len(objets)
    j = len(tab[0]) - 1
    contenuSac = []
    valeur = tab[n][j]
    for i in range(n, 0, -1):
        if tab[i - 1][j] != tab[i][j]:
            v, p = objets[i - 1]
            contenuSac.append((v, p))
            j = j - p
    return valeur, contenuSac
```

Complexité

La création du tableau, et son remplissage, ont une complexité en temps en O(*nw*) car il s'agit de deux boucles imbriquées de longueurs respectives *n* et *w* avec des opérations en temps constant. La reconstitution de la solution est en O(*n*). Au final, l'algorithme de résolution est en O(*nw*).

La complexité en mémoire est également en O(*nw*) puisqu'on utilise un tableau à *nw* cases, ou plus précisément (*n* + 1)(*w* + 1) cases pour mémoriser les résultats. On peut négliger les autres variables et la liste objets.

Solution de l'exercice papier

objets = [(5, 3), (9, 2), (10, 5), (6, 4), (7, 1), (9, 3)] et *w* = 10

objets\poids	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	5	5	5	5	5	5	5	5
2	0	0	9	9	9	14	14	14	14	14	14
3	0	0	9	9	9	14	14	19	19	19	24
4	0	0	9	9	9	14	15	19	19	20	24
5	0	7	9	16	16	16	21	22	26	26	27
6	0	7	9	16	16	18	25	25	26	30	31

La valeur maximale est 31, avec un poids de 10 kg. On prend :

- l'objet 6 et 7 kg avec les 5 autres objets
- les objets 6, 5 et 6 kg avec les 4 autres objets
- les objets 6, 5, 4 et 2 kg avec les 3 autres objets
- les objets 6, 5, 4 et 2.

On obtient donc la valeur optimale de 31 avec les objets 2, 4, 5, 6.