Chapitre 2

Programmation dynamique - Mémoïsation

Sommaire

Si le fait qu'un sous-problème quelconque possède une meilleure solution implique l'existence d'une meilleure solution globale, alors la meilleure solution globale correspond à la meilleure solution de tous les sous-problèmes.
La programmation dynamique, c'est la formulation des problèmes d'optimisation par une formule de récurrence.

2.1 Décomposition en sous-problèmes

2.1.1 Programmation dynamique

Ce concept a été introduit vers 1950 par Richard Bellman.

Diviser pour régner versus programmation dynamique

La méthode diviser pour régner consiste à décomposer un problème en problèmes élémentaires en partitionnant la recherche en branches indépendantes : la méthode diviser pour régner s'applique à des **sous-problèmes disjoints**. La programmation dynamique est une technique pour résoudre un problème qui se divise naturellement en **sous-problèmes interdépendants** (en ce sens qu'ils possèdent des sous-sous-problèmes en commun). Dans les deux méthodes, on formule le problème par une **relation de récurrence**.

La dépendance entre les sous problèmes provient généralement d'une recherche de **solution optimale**[1] parmi de nombreux chemins interdépendants : recherche d'un parcours optimal dans un graphe, de la meilleure concordance entre deux séquences, d'une répartition optimale, etc. **La programmation dynamique offre l'avantage de ne pas recalculer de nombreuses fois les parties communes à plusieurs parcours ou à plusieurs séquences.**

1. Une technique d'optimisation suppose l'évaluation d'une fonction de coût que l'on cherche à maximiser ou minimiser. La programmation dynamique est notamment appliquée en économie et en chimie des procédés.

Principe

La programmation dynamique offre une solution efficace à certains problèmes d'optimisation pour lesquels des solutions locales optimales peuvent conduire à une solution globale optimale (principe d'optimalité de Bellman). La programmation dynamique est d'autant plus efficace que les sous-problèmes se chevauchent fortement (on dit aussi qu'ils se superposent), c'est-à-dire que les mêmes sous-sous-problèmes réapparaissent souvent.

Le terme de « programmation » doit être compris comme signifiant « tableau de planification », il ne s'agit pas de programmation dans le sens d'écrire du code informatique. Lorsque la solution d'un niveau de sous-problèmes ne dépend que du niveau précédent, on parle de programmation dynamique série.
La programmation dynamique fournit facilement l'évaluation optimale, mais nécessite parfois une **phase de reconstruction d'une solution optimale** *a posteriori*, à l'aide d'informations supplémentaires mémorisées lors de la recherche.

2.1.2 Programmation dynamique bottom-up

La programmation dynamique bottom-up utilise des tableaux pour stocker les résultats communs à plusieurs sous-problèmes épargnant de les recalculer plusieurs fois, elle est associée à un paradigme impératif. On part des conditions initiales de la relation de récurrence et l'on remplit le tableau (structure mutable) en résolvant les sous-problèmes par taille croissante. **La solution globale optimale doit pouvoir se déduire des solutions optimales des sous-problèmes, on dit qu'elle possède une sous-structure optimale.**

2.1.3 Programmation dynamique top-down : mémoïsation

De manière générale, la mémoïsation consiste à garder en mémoire des résultats intermédiaires utilisés plusieurs fois dans un algorithme.
La mémoïsation se rencontre en programmation récursive de type top-down (comme celle qui est utilisée dans la stratégie diviser pour régner lorsqu'on la programme de manière récursive). La programmation dynamique top-down par mémoïsation est souvent utilisée. La gestion du recouvrement des sous-problèmes permet de **ne pas provoquer d'appel récursif lorsqu'un sous-problème a déjà été résolu**.

Mémoïsation

La programmation dynamique par mémoïsation [a] **(ou recensement) est une technique de programmation récursive top-down.** Dans la phase de descente, les appels récursifs ne sont réalisés que si un appel similaire n'a pas déjà été effectué. On tient donc à jour une **table des résultats des sous problèmes déjà résolus**, que l'on consulte afin de ne procéder à l'appel récursif que si le sous problème est nouveau. Un test précède généralement un appel récursif qui porte sur la consultation de la table de résultats déjà obtenus.

a. De l'anglais memoization ; terme proposé en 1968 par Donald Michie, chercheur en intelligence artificielle.

La table est réalisée à l'aide d'une structure mutable, comme une table d'association (dictionnaire), réalisée par une table de hachage par exemple, ou une liste associative.

Le principal avantage de la mémoïsation est l'approche récursive descendante, notamment si l'arbre de recherche s'élargit très vite : il est plus facile de partir du problème global et de le décomposer (mémoïsation) que de partir des problèmes élémentaires (programmation dynamique bottom-up). L'inconvénient de la mémoïsation par rapport à la programmation dynamique bottom-up est la lourdeur de la gestion des appels récursifs, ainsi que la complexité accrue de la structure de stockage (clé + valeur au lieu de la seule valeur), mais c'est souvent une approche intéressante.

2.1.4 Rappel : la programmation gloutonne

Lorsque la programmation dynamique est trop lourde à cause de la multiplication des options possibles à chaque niveau de sous-problème, on peut se limiter à faire un seul choix local qui semble être le meilleur en suivant une heuristique, ce qui élimine beaucoup de sous-problèmes. On quitte les modèles formels de recherche exhaustive, et la solution trouvée n'est pas nécessairement la meilleure. Cette stratégie de recherche fournit parfois la solution optimale. Cette stratégie est de type top-down (penser au sac à dos ou au rendu de monnaie). Si la solution optimale peut être obtenu par un choix glouton à la première étape et que l'on peut propager cette propriété par récurrence à chaque étape, sur des problèmes plus petits, alors la stratégie gloutonne est un bon choix.

Programmation dynamique versus algorithme glouton

La **notion de sous-structure optimale** de la programmation dynamique est partagée par les algorithmes gloutons. La recherche gloutonne n'est pas exhaustive, elle est purement locale.

2.1.5 Conclusion

Conventions

Certains informaticiens opposent programmation dynamique et mémoïsation. Nous faisons un choix différent dans le cours en parlant systématiquement de programmation dynamique, mais en distinguant l'approche montante (plutôt impérative, mais on peut toujours trouver une formulation récursive terminale à une programmation dynamique montante.) et l'approche descendante récursive (mémoïsation).

La programmation dynamique est une méthode consistant à décomposer un problème d'optimisation en sous-problèmes qui se chevauchent, pour lesquels on peut propager la notion d'optimalité. La mémoïsation apporte le bénéfice habituel de la récursivité top-down pour ne pas avoir à gérer la reconstruction de la solution globale à partir des sous-problèmes les plus simples (penser aux tours de Hanoï pour visualiser l'intérêt du top-down !).

Programmation dynamique

La programmation dynamique est invoquée dans un problème d'optimisation qui peut se résoudre par la propriété de sous-structure optimale.
Formulation impérative : on commence par résoudre les problèmes de petite taille ; on combine les solutions obtenues pour résoudre des sous-problèmes de plus grande taille qui se chevauchent, et l'on remonte jusqu'au problème global : la progression est de type bottom-up, il faut réfléchir à organiser les calculs.
Mémoïsation : écriture récursive top-down souvent plus simple à concevoir, il faut gérer le stockage des sous-problèmes résolus (dans un dictionnaire typiquement).

La mémoïsation est utilisée si l'on ne parvient pas facilement à décortiquer la relation de récurrence qui caractérise la décomposition en sous-problèmes pour deviner l'ordre des calculs par programmation dynamique impérative (notamment si le nombre de problèmes élémentaires est très grand et que l'on suspecte qu'il n'est pas nécessaire de tous les résoudre ou qu'on ne sait pas comment les organiser). Le problème du sac à dos est un bon exemple où la mémoïsation limite le nombre de sous-problèmes considérés. En revanche si l'on sait résoudre facilement la récurrence la programmation dynamique bottum-up est plus légère (suite récurrente d'ordre 2, coefficients binomiaux par exemple).
On peut faire le parallèle avec les fonctions récursives déjà rencontrées : la récursivité terminale bottom-up où le cas de base résout le problème élémentaire qui est finalement remonté sans traitement ou la récursivité non terminale top-down où l'on plonge dans les profondeurs sans trop se préoccuper de l'ordre des calculs avec une reconstruction de la solution à la remontée.

2.2 Suite de Fibonacci

Soit la suite récurrente linéaire d'ordre 2 $(F_n)_{n\in\mathbb{N}}$ définie par

$$F_0 = F_1 = 1 \qquad \text{et pour } n \geqslant 2, \quad F_n = F_{n-1} + F_{n-2} \tag{2.1}$$

L'objectif est de procéder au calcul du terme général F_n pour un n donné.

2.2.1 Calcul flottant

Une idée qui peut venir à l'esprit est de résoudre la récurrence linéaire ; on trouve facilement :

$$F_n = \frac{1}{\sqrt{5}}\left(\phi^n - \phi'^n\right) \qquad \text{avec} \qquad \phi = \frac{1+\sqrt{5}}{2} \quad \text{et} \quad \phi' = \frac{1-\sqrt{5}}{2} \tag{2.2}$$

Le second terme devient vite négligeable et F_n peut se calculer comme l'entier le plus proche de $\frac{\phi^n}{\sqrt{5}}$. Comme $\frac{F_{n+1}}{F_n}$ tend vers $\phi \approx 1,618$, et que les flottants Python suivent la norme IEEE754 avec 53 bits significatifs, on peut espérer mener le calcul tant que $\phi^n < 2^{53}$ (en gros pour $n \leqslant 76$).

```python
from math import sqrt
def fibo1(n):
    r = sqrt(5.)
    phi = (1. + r) * 0.5
    return round(phi ** n / r)
```

On obtient effectivement la bonne valeur jusqu'à $n = 70$ avec la fonction précédente, dont la complexité temporelle et spatiale est $\Theta(1)$. Mais l'idée de passer par des calculs en virgule flottante pour calculer une suite entière ne s'avère pas féconde très longtemps.

2.2.2 Calcul impératif simple

La méthode la plus sérieuse est une implémentation impérative élémentaire de la suite récurrente avec mémorisation des deux dernières valeurs calculées.

```python
def fibo2(n):
    a, b = 0, 1
    for i in range(n-1):
        a, b = b, a + b
    return 0 if n == 0 else b
```

Les entiers Python ne sont pas limités : cette fonction est la bonne méthode pour programmer la suite de Fibonacci, comme toute suite récurrente d'ordre 2. La complexité temporelle est $\Theta(n)$ et la complexité spatiale $\Theta(1)$. En effet, le calcul de F_n nécessite $n-1$ additions.

Attention

La complexité temporelle n'est pas réellement en $\Theta(n)$ car les additions portent sur des nombres dont la taille augmente avec n. Comme la seule opération à réaliser est une addition entre deux nombres de $\Theta(n)$ bits. La complexité temporelle globale sera en $\Theta(n^2)$ et la complexité spatiale en $\Theta(n)$.

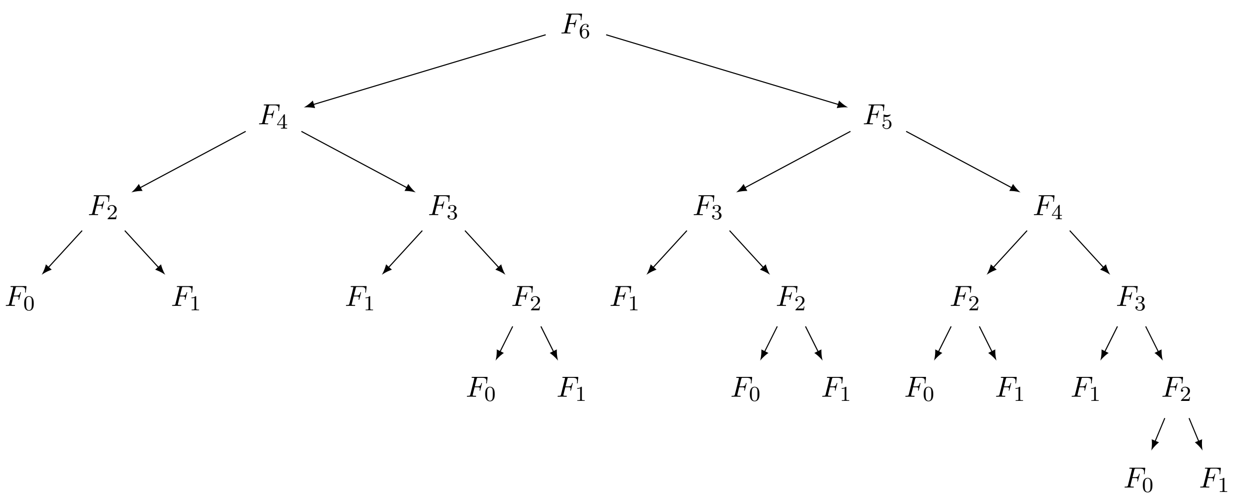
On trouve instantanément $F_{100} = 354224848179261915075$.

2.2.3 Implémentation récursive top-down

Si l'on recherche une implémentation fonctionnelle n'utilisant que des variables immuables, l'approche naïve conduit à une écriture très élégante, parlante et synthétique.

```python
def fibo3(n):
    if n < 2:
        return n
    return fibo3(n-1) + fibo3(n-2)
```

On constate que le calcul de F_{40} prend plusieurs dizaines de secondes et le calcul de F_{50} n'aboutit pas ! Il suffit de représenter l'arbre des appels récursifs pour constater que pour calculer F_n, on calcule une fois F_{n-1}, deux fois F_{n-2}, 3 fois F_{n-3}, 5 fois F_{n-4}, 8 fois F_{n-5}, etc. Finalement F_{n-k} est calculé F_{k+1} fois. Le nombre d'appels récursifs pour calculer F_n est $2(F_{n+1} - 1) \sim \frac{2}{\sqrt{5}}\phi^{n+1} = \Theta(\phi^n)$. **La complexité temporelle de l'algorithme récursif est exponentielle**, ce qui explique l'impasse dans laquelle on se retrouve rapidement.



Le schéma montre les 24 appels récursifs réalisés pour calculer F_6, qui se calcule pourtant simplement avec 5 additions (il suffit de suivre la branche la plus à droite).

2.2.4 Implémentation récursive bottom-up

Une manière de rendre efficace l'approche récursive est de partir du cas de base par une approche bottom-up et d'utiliser des accumulateurs pour les deux termes à mettre à jour.

```python
def fibo4(n, a = 0, b = 1):
    if n == 1:
        return b
    return 0 if n == 0 else fibo4(n-1, b, a+b)
```

Nous retrouvons une complexité linéaire en temps, avec un seul appel récursif par valeur de n. En revanche la complexité spatiale n'est plus constante, mais linéaire. L'appel récursif est terminal et un bon compilateur doit aisément simplifier la gestion des contextes (ce n'est pas le cas de l'interpréteur Python).
L'inconvénient de cette méthode est de ne plus mettre en évidence dans le code la relation de récurrence $F_n = F_{n-1} + F_{n-2}$. La programmation récursive terminale n'est qu'un maquillage d'une programmation impérative naturelle. On obtient instantanément $F_{100} = 354224848179261915075$.

2.2.5 Programmation dynamique impérative bottom-up

Attention

L'exemple de la suite de Fibonacci n'a qu'une vocation d'illustration pédagogique, personne ne devrait penser à la programmation dynamique pour une suite récurrente : la bonne solution a été fournie précédemment (calcul impératif simple).

Nous avons utilisé une astuce dans les solutions linéaires précédentes : supprimer les évaluations redondantes de F_k en remplaçant la récurrence d'ordre 2 par une récurrence d'ordre 1. L'idée est de se placer dans un espace de dimension 2 et de remarquer que si $u_n = (F_{n-1}, F_n)$ alors $u_{n+1} = (F_n, F_{n+1}) = (u_n[1], u_n[0] + u_n[1])$. Ainsi $u_{n+1} = f(u_n)$ avec $f((a, b)) = (b, a + b)$, ce que traduit précisément la solution récursive simple.
Certains problèmes ne permettent pas cette astuce et la programmation dynamique impérative bottom-up permet d'utiliser la relation de récurrence de manière apparente en conservant la complexité temporelle linéaire.

```
def fibo5(n):
    F = [0,1] + [0] * (n-1)
    for k in range(2, n+1):
        F[k] = F[k-1] + F[k-2]
    return F[n]
```

Un tableau est alloué dans la fonction afin de sauvegarder les valeurs intermédiaires. Les contraintes des méthodes précédentes (résolution de la récurrence ou passage en dimension supérieure) ont disparu. On comprend que cette approche est généralisable à de nombreux autres problèmes où l'onpeut avoir besoin de valeurs déjà calculées. Il faut cependant être en mesure de remplir le tableau dans le bon ordre, c'est-à-dire traiter les sous-problèmes dans le sens ascendant (bottom-up). Les complexités temporelle et spatiale sont en $\Theta(n)$. Dans le cas présent le tableau est bien inutile puisque seules les deux dernières valeurs sont à conserver (d'où l'aspect factice de cet exemple).
On obtient instantanément $F_{100} = 354224848179261915075$.

2.2.6 Programmation dynamique top-down par mémoïsation

Si l'on privilégie l'approche récursive, la mémoïsation permet de procéder dans le sens naturel descendant et l'on implémente la suite récurrente d'une manière proche de sa définition mathématique.

```
def fibo6(n):
    F = {0:0, 1:1}
    def f(n):
        if n not in F:
            F[n] = f(n-1) + f(n-2)
        return F[n]
    return f(n)
```

L'écriture de la fonction suit sa définition : on cherche F_n dans la table d'association ; si la valeur est trouvée elle est simplement retournée, sinon elle est calculée par la relation de récurrence, sauvegardée dans la table et retournée.
Les différentes valeurs de F_n n'étant calculées qu'une fois, la complexité temporelle est linéaire. L'usage d'une table de hachage est assez classique pour la mémoïsation, mais ici les clés sont simplement les valeurs de n, si bien qu'un simple tableau peut suffire. La dernière implémentation illustre l'avantage de la mémoïsation pour la simplicité du code, mais l'exemple de la suite de Fibonacci est artificiel. On obtient instantanément $F_{100} = 354224848179261915075$.

2.2.7 Complexité sous-linéaire

Pour de grandes valeurs de n, on rappelle qu'il serait plus efficace d'utiliser l'algorithme d'exponentiation rapide pour calculer

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} \tag{2.3}$$

Vérifier que vous savez programmer une fonction `fibo7` qui réalise le calcul de F_n par cette méthode.

2.3 Rendu de monnaie

2.3.1 Présentation, approche gloutonne

Algorithme glouton

La stratégie gloutonne est une heuristique visant à fournir efficacement une solution à un problème d'optimisation, que l'on espère la meilleure possible, sans qu'elle soit nécessairement optimale. Elle procède par une suite de choix en sélectionnant à chaque étape la solution qui semble être localement la meilleure.

Le problème du rendu de monnaie avec un minimum de pièces a été traité en première année à l'aide d'un algorithme glouton : on rend systématiquement la pièce de valeur maximale (mais inférieure à ce qu'il reste à rendre!) tant qu'il reste quelque chose à rendre. On suppose les pièces classées par ordre de valeurs décroissantes pour l'approche gloutonne.

```python
def monnaie0(pieces, S):
    T = []
    for i, v in enumerate(pieces):
        while v <= S:
            S -= v
            T.append(v)
    return T
```

On obtient `[4,1,1]` pour `monnaie0([4,3,1],6)`. C'est bien la solution attendue, mais elle n'est pas optimale : la solution optimale est `[3,3]`
Le rendu est optimal pour un système de pièces canonique avec l'approche gloutonne, mais un système quelconque de pièces n'est pas canonique, comme $\{1, 3, 4\}$.

2.3.2 Approche naïve par force brute

Rendu de monnaie : une stratégie simple

Pour rendre la somme S, si on rend une pièce p il reste à rendre la somme $S - p$. Si l'on sait rendre toute somme inférieure à S de manière optimale, il suffit de tester toutes les possibilités de rendre $S - p$, pour toute pièce p et de choisir la meilleure solution.

Contrairement à la suite de Fibonacci, il est peu intuitif d'utiliser une approche bottom-up simple pour construire la solution car le nombre de problèmes élémentaires peut être gigantesque. On privilégie spontanément l'approche top-down, mais avec la même difficulté par force brute : une explosion combinatoire liée à une complexité temporelle exponentielle.
Pour considérer toutes les manières de parvenir à la somme S, on considère chaque pièce p dans la liste L de pièces et l'on appelle récursivement la fonction pour obtenir le nombre minimal de pièces pour parvenir à la somme $S-p$. À chacune des valeurs obtenues, on ajoute 1 et l'on prend le minimum pour l'ensemble des pièces p possibles. La fonction suivante renvoie le nombre de pièces minimal (la liste de pièces n'a pas besoin d'être triée).

```python
def monnaie1(pieces, S):
    if S == 0:
        return 0
    nbre = S
    for p in pieces:
        if p <= S:
            nbre = min(nbre, 1 + monnaie1(pieces, S-p))
    return nbre
```

On a supposé que 1 appartient à l'ensemble des pièces (ce qui est nécessaire pour pouvoir rendre $S = 1$), pour initialiser `nbre`. On obtient la bonne réponse 2 pour `monnaie1([1,3,4],6)`.
On s'aperçoit très vite de l'impasse : on n'obtient pas la réponse pour rendre $S = 100$ avec $\{1, 3, 4\}$, alors que la solution est évidente à trouver de tête. Le problème est exactement le même qu'avec la suite de Fibonacci : les mêmes sous-problèmes sont résolus plusieurs fois, la complexité temporelle est exponentielle en S.
Un codeur ignorant serait pourtant fier d'une solution comme celle-ci :

```python
def monnaie1(pieces, S):
    return 1 + min(monnaie1(pieces, S-p) for p in pieces if p <= S) if S else 0
```

2.3.3 Formalisation comme problème d'optimisation

Soit $S \in \mathbb{N}$ la somme à rendre et $(p_i)_{0 \leqslant i \leqslant N-1}$ le système de pièces. On suppose que dans ce système la pièce p_0 a pour valeur 1.
On note $n_k(s)$ le nombre de pièces minimal (solution d'un sous-problème) pour rendre la somme s avec les pièces $(p_i)_{0 \leqslant i \leqslant k-1}$. La solution au problème du rendu de monnaie est donc $n_N(S)$.
On remarque que :

$$n_k(s) = \begin{cases} n_{k-1}(s) & \text{si la pièce k n'est pas utilisée dans la solution optimale} \\ n_k(s - p_k) + 1 & \text{si la pièce k est utilisée} \end{cases} \tag{2.4}$$

On a la condition initiale $n_k(0) = 0$ pour tout $k > 0$ et $n_0(s) = s$ pour tout s afin de résoudre par itérations sur $k \geqslant 1$ et s. La programmation dynamique consiste à résoudre le problème par la récurrence suivante :

$$n_k(s) = \min\{n_{k-1}(s), n_k(s - p_k) + 1\} \tag{2.5}$$

en imbriquant la boucle sur k dans une boucle sur s, de manière à pouvoir rendre plusieurs fois la même pièce. La récurrence est simple sur k (seul $k - 1$ est utilisé) mais pas sur l'argument s.

2.3.4 Programmation dynamique bottom-up

Si l'on procède par valeurs croissantes (bottom-up) de s, la valeur de $n_k(s - p_k)$ a déjà été calculée lorsqu'on en a besoin. Comme dans le cas de la suite de Fibonacci, un simple tableau peut suffire pour stocker les résultats des sous-problèmes. La programmation dynamique impérative permet d'écrire un programme aussi simple que le précédent et dont la complexité temporelle est de l'ordre de $n \times S$ si n est le nombre de pièces.

```python
def monnaie2(pieces, S):
    nbre = [0] * (S+1)
    for s in range(1, S+1):
        nbre[s] = s
        for p in pieces:
            if p <= s:
                nbre[s] = min(nbre[s], 1 + nbre[s-p])
    return nbre[S]
```

L'appel à `monnaie2([1, 3, 4],100)` retourne immédiatement la réponse évidente 25. Il et clairement plus efficace de consulter le tableau `nbre` que de réaliser un appel récursif, qui en réalise d'autres, etc.
Il paraît souhaitable de connaître, outre le nombre de pièces, la solution au rendu de monnaie, sous la forme d'une liste de pièces à rendre.

```
def monnaie3(pieces, S):
    nbre = [0] * (S+1)
    sol = [[]] * (S+1)
    for s in range(1, S+1):
        nbre[s] = s
        sol[s] = [1] * s
        for p in pieces:
            if p <= s:
                n = nbre[s-p] + 1
                if nbre[s] > n:
                    nbre[s] = n
                    sol[s] = sol[s-p] + [p]
    return sol[S]
```

L'appel `monnaie3([1,3,4],6)` renvoie `[3,3]`

2.3.5 Programmation dynamique top-down par mémoïsation

On inverse le sens de la recherche, on part de S et l'on descend (top-down) récursivement vers les sommes plus petites en stockant les résultats dans un dictionnaire.

```
def monnaie4(pieces, S):
    nbre = {0:0}
    def rend(s):
        if s not in nbre:
            nbre[s] = s
            for p in pieces:
                if p <= s:
                    nbre[s] = min(nbre[s], 1 + rend(s-p))
        return nbre[s]
    return rend(S)
```

Le principe de la fonction récursive qui utilise la mémoïsation est toujours de commencer par consulter le dictionnaire pour directement retourner la valeur si elle a déjà été calculée.
Si l'on veut la liste des pièces à rendre, on complète l'information contenue dans le dictionnaire.

```
def monnaie5(pieces, S):
    dico = {0:(0,[])}
    def rend(s):
        if s not in dico:
            dico[s] = s, [1] * s
            for p in pieces:
                if p <= s:
                    n, L = rend(s-p)
                    if dico[s][0] > n + 1:
                        dico[s] = n + 1, L + [p]
        return dico[s]
    return rend(S)
```

L'avantage de la mémoïsation par rapport à la programmation bottom-up est de ne pas traiter de sous-problème inutile, puisque le dictionnaire n'est enrichi que si le sous-problème est effectivement rencontré.

2.4 PROBLÈME DU SAC À DOS

2.4.1 Présentation de l'optimisation

Le problème du sac à dos (KP pour *Knapsack Problem)* est une référence algorithmique incontournable concernant l'optimisation combinatoire. Wikipedia (28/09/2022) :

> « Le problème du sac à dos est l'un des 21 problèmes NP-complets de Richard Karp, exposés dans son article de 1972. Il est intensivement étudié depuis le milieu du XXe siècle et on trouve des références dès 1897, dans un article de George Ballard Mathews. La formulation du problème est fort simple, mais sa résolution est plus complexe. Les algorithmes existants peuvent résoudre des instances pratiques de taille importante. Cependant, la structure singulière du problème, et le fait qu'il soit présent en tant que sous-problème d'autres problèmes plus généraux, en font un sujet de choix pour la recherche. »

Problème du sac à dos

Soit un ensemble de n objets, $n \in \mathbb{N}^*$, numérotés par $i \in [\![1, n]\!]$, de poids w_i et de valeurs v_i. Le **problème du sac à dos** consiste à trouver un sous-ensemble de ces objets, de valeur totale maximale (pour emporter le meilleur butin) et telle que le poids total soit inférieur à une valeur donnée $w_{\max}$ (ce que supporte le sac à dos sans craquer).

L'objectif est de déterminer un n-uplet $(x_1, \cdots x_n) \in \{0,1\}^n$ tel que

$$\sum_{i=1}^{n} x_i\, v_i \quad \text{maximal} \qquad \text{et} \qquad \sum_{i=1}^{n} x_i\, w_i \leqslant w_{\max} \tag{2.6}$$

La solution par force brute consistant à générer les 2^n n-uplets possibles est de complexité exponentielle est inefficace pour n et $w_{\max}$ grands.
On note $g_i(w)$ la valeur maximale qu'il est possible d'atteindre à l'aide des i premiers objets pour que le poids total soit inférieur ou égal à $w > 0$. La réponse au problème est $g_n(w_{\max})$.
On distingue deux cas :

- Si $w_i > w$, l'objet i ne peut pas être choisi, ainsi $g_i(w) = g_{i-1}(w)$.
- Si $w_i \leqslant w$, deux sous-cas se présentent :
  — Soit l'objet i permet d'optimiser la valeur totale et on doit placer cet objet dans le sac. La masse encore disponible est $w - w_i$ et l'on en déduit $g_i(w) = v_i + g_{i-1}(w - w_i)$.
  — Soit il est préférable de ne pas prendre cet objet i dans le sac car la valeur totale peut être maximale en choisissant un autre objet parmi les $(i-1)$ précédents. Ainsi $g_i(w) = g_{i-1}(w)$.

  La programmation dynamique peut donc être utilisée avec la récurrence suivante :

$$g_i(w) = \max(g_{i-1}(w), v_i + g_{i-1}(w - w_i)) \tag{2.7}$$

Du point de vue algorithmique, le problème du rendu de monnaie n'est qu'un cas particulier du problème du sac à dos (min au lieu de max, 1 au lieu de v_i, égalité au lieu d'inégalité).
L'initialisation de la récurrence se réalise avec $g_0(w) = 0$ pour tout $W > 0$ car c'est la valeur de zéro objet et $g_i(0) = 0$ pour tout i car tout objet a un poids supérieur au maximum 0.

2.4.2 Approche naïve récursive

Même si la solution est inefficace, il est important de savoir la programmer en Python. L'usage d'une fonction locale est naturelle et la fonction englobante renvoie la réponse $g_n(w_{\max})$.

```python
def sac1(objets, wmax):
    def g(i, w):
        if i == 0 or w == 0:
            return 0
        vi, wi = objets[i-1]
        if wi > w:
            return g(i-1, w)
        return max(g(i-1, w), vi + g(i-1, w-wi))
    return g(len(objets), wmax)
```

On peut facilement comprendre que les deux appels récursifs conduiront encore à une complexité exponentielle à cause du recouvrement des sous-problèmes, comme pour les nombres de Fibonacci ou le rendu de monnaie. Cette solution n'est pas réaliste pour des problèmes de grande taille.

2.4.3 Programmation dynamique bottom-up

Les exemples précédents montrent que l'usage d'un tableau est naturel pour l'écriture impérative de la programmation dynamique (car la récurrence n'est que du premier ordre sur i). L'itération externe porte sur i, et pour chaque i on considère tous les w possibles. On prend garde à écraser g_{i-1} par g_i seulement lorsque g est mis à jour pour tous les w, d'où la copie de g.

```python
def sac2(objets, wmax):
    g = [0] * (wmax+1)
    for i in range(len(objets)):
        t = g[:]
        for w in range(1, wmax+1):
            vi, wi = objets[i]
            if wi <= w:
                g[w] = max(t[w], vi + t[w-wi])
    return g[wmax]
```

Ce programme réalise $\Theta(n \cdot w_{\max})$ opérations ; cette complexité temporelle devient praticable. Noter la bonne complexité spatiale en $O(n + w_{\max})$ avec cette approche.
On peut même éviter la copie de tableau si l'on procède par valeurs décroissantes de w. Cela permet en outre de ne pas écrire la valeur dans `g[w]` lorsqu'elle demeure inchangée !

```python
def sac2(objets, wmax):
    g = [0] * (wmax+1)
    for i in range(len(objets)):
        for w in range(wmax,0,-1):
            vi, wi = objets[i]
            if wi <= w:
                a = vi + g[w - wi]
                if a > g[w]:
                    g[w] = a
    return g[wmax]
```

Et Python permet d'obtenir la solution complète de manière très élégante :

```python
def sac4(objets, wmax):
    g = [(0,[])] * (wmax + 1)
    for i in range(len(objets)):
        for w in range(wmax, 0, -1):
            vi, wi = objets[i]
            if wi <= w:
                a, b = g[w - wi]
                if a + vi > g[w][0]:
                    g[w] = a + vi, [(vi,wi)] + b
    return g[wmax]
```

2.4.4 Programmation dynamique top-down par mémoïsation

Cette approche utilise une fonction récursive pour traduire la récurrence, mais avec un stockage global qui est mis à jour. Avec l'approche top-down, il n'est plus possible d'écraser les résultats précédents : un stockage à deux dimensions est nécessaire, ou un dictionnaire prenant pour clés les couples (i, w). L'usage d'un dictionnaire est encore un peu trop luxueux ici, mais c'est l'esprit de la démarche.

Complexités

La programmation dynamique résulte souvent d'un compromis entre complexité temporelle et complexité spatiale. L'intérêt de la mémoïsation est de ne pas explorer de sous-problème inutilement.

```
def sac3(objets, wmax):
    dico = {}
    def g(i, w):
        if (i,w) not in dico:
            if i == 0 or w == 0:
                dico[0,w] = 0
                return 0
            vi, wi = objets[i-1]
            if wi > w:
                dico[i,w] = g(i-1, w)
            else:
                dico[i,w] = max(g(i-1, w), vi + g(i-1, w-wi))
        return dico[i,w]
    return g(len(objets), wmax), len(dico)
```

La complexité temporelle est toujours en $\Theta(n \cdot w_{\max})$ mais la complexité spatiale dans le pire des cas est dégradée par rapport à l'approche bottom-up, elle est en $O(n \times w_{\max})$. Le dictionnaire permet d'obtenir la liste des objets.

```
def sac3(objets, wmax):
    dico = {}
    def g(i, w):
        if (i,w) not in dico:
            if i == 0:
                dico[0,w] = 0
                return 0
            vi, wi = objets[i-1]
            if wi > w:
                dico[i,w] = g(i-1, w)
            else:
                dico[i,w] = max(g(i-1, w), vi + g(i-1, w-wi))
        return dico[i,w]
    val = g(len(objets), wmax)
    w, L = wmax, []
    for i in range(len(objets), 0, -1):
        if dico[i, w] != dico[i-1, w]:
            vi, wi = objets[i-1]
            L.append((vi,wi))
            w -= wi
    return L, val, wmax - w
```

Ainsi `print(sac3([(7,13),(4,11),(3,8),(3,10)] ,30))` renvoie ([(4, 11), (7, 13)], 11, 24), ce que ne permet pas la programmation gloutonne.

Table des matières