CHAPITRE 2

Programmation dynamique

Jérôme VON BUHREN
http://vonbuhren.free.fr

CPGE - Filière Scientifique - 2e année

Dans ce chapitre, nous commencerons par décrire les principes de l'implémentation d'un dictionnaire avec une table de hachage. Dans un second temps, nous étudierons une méthode algorithmique permettant de résoudre de nombreux problèmes : la programmation dynamique. Ce concept a été introduit au début des années 1950 par le mathématicien américain Richard Bellman.

En première année, les dictionnaires ont été introduits et utilisés comme une boîte noire. Dans cette partie, nous allons étudier les principes généraux de leur fonctionnement. Nous les utiliserons dans la suite pour résoudre certains problèmes avec la programmation dynamique.

Un dictionnaire présente de nombreuses similitudes avec les listes, si ce n'est qu'au lieu d'accéder aux éléments individuels par le biais d'un indice, on y accède par le biais d'une clé. Les clés du dictionnaire doivent être des objets immuables : `str`, `int`, `tuple`, `float`,...

La création d'un dictionnaire se réalise en suivant la syntaxe

```
{c1: v1, ... , cn: vn}
```

où $c_1, \ldots, c_n$ sont des clés (nécessairement deux à deux distinctes) et $v_1, \ldots, v_n$ les valeurs qui leur sont associées.

```
1 # Création d'un dictionnaire vide
2 D = {}
3
4 # Création d'un dictionnaire avec quatre paires clé/valeur
5 D = {'cle1': 'valeur1', -8: [1,3], (3,5):'ptet', 0.5: 12}
```

Si `D` est un dictionnaire et `c` est une clé, alors

- l'expression `D[c] = v` crée une nouvelle association si la clé n'est pas présente dans le dictionnaire et modifie l'association précédente sinon ;
- l'expression `del D[c]` supprime une association si la clé est présente dans le dictionnaire et déclenche l'exception `KeyError` sinon ;
- l'expression `c in D` renvoie un booléen indiquant si la clé est présente ou non dans le dictionnaire ;
- l'expression `D[c]` renvoie la valeur associée à la clé si celle-ci est présente dans le dictionnaire et déclenche l'exception `KeyError` sinon.

Remarque 1

On considère que les quatre opérations de base décrites ci-dessus se réalisent avec une complexité temporelle constante (même si nous verrons que ce n'est pas tout-à-fait exact en réalité).

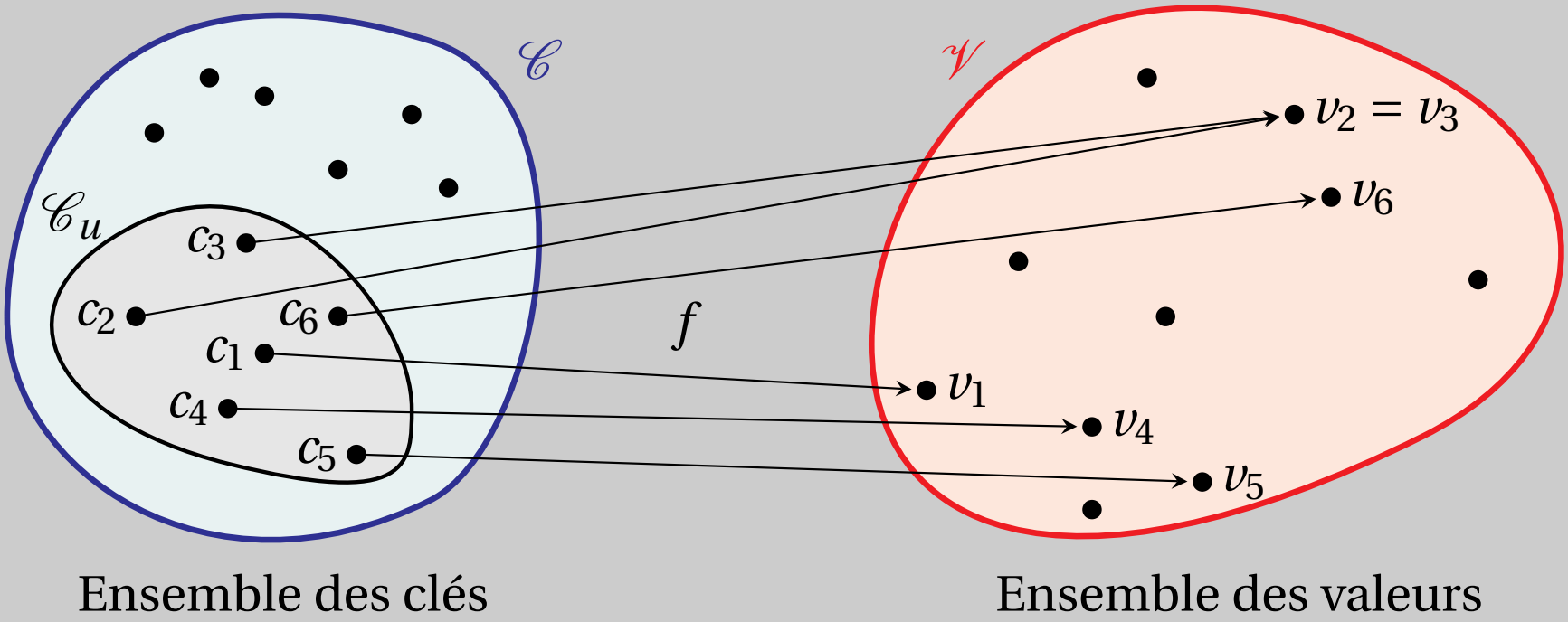
En complément des opérations décrites précédemment, il est également possible de récupérer le nombre de paires clé/valeur dans le dictionnaire et de parcourir toutes les clés du dictionnaire avec les syntaxes ci-dessous.

```
1 # Nombre de paires dans le dictionnaire
2 len(D)
3
4 # Parcours du dictionnaire
5 for cle in D:
```

D'un point de vue théorique, un **dictionnaire** (aussi appelé **table d'association**) est un type de données associant un ensemble de clés à un ensemble correspondant de valeurs. Plus formellement, si on désigne l'ensemble des clés par $\mathscr{C}$ et l'ensemble des valeurs par $\mathscr{V}$, un dictionnaire est une application $f : \mathscr{C}_u \to \mathscr{V}$ où $\mathscr{C}_u$ est une partie de $\mathscr{C}$ représentant les clés utilisées. Un couple de la forme $(c, f(c))$ avec $c \in \mathscr{C}_u$ est appelée une **association**.

Illustration

Le schéma ci-dessous représente de manière informelle un dictionnaire avec 6 associations.



Remarque 2

Dans l'implémentation en Python d'un dictionnaire, l'ensemble $\mathscr{C}$ est l'ensemble de tous les objets immuables : il est gigantesque.

Les opérations que l'on peut généralement effectuer sur un dictionnaire sont les suivantes :

- **ajouter** une nouvelle association $(c, v) \in \mathscr{C} \times \mathscr{V}$;
- **supprimer** une association existante ;
- **tester** l'existence d'une association avec une clé donnée $c \in \mathscr{C}$;
- **lire** la valeur associée à une clé utilisée $c \in \mathscr{C}_u$.

Dans le cas où l'ensemble des clés est $\mathscr{C} = [\![0, m-1]\!]$ où $m \in \mathbb{N}^*$, il est naturel d'utiliser un tableau $\mathscr{T}$ à m éléments pour implémenter notre dictionnaire. D'un point de vue théorique, si l'ensemble $\mathscr{C}$ n'est pas de la forme précédente, on pourrait s'y ramener en attribuant un indice à chacun de ses éléments.

Cependant, l'utilisation de cette démarche présenterait des inconvénients majeurs.

- L'ensemble des clés $\mathscr{C}$ étant très grand, la mémoire occupée par le tableau $\mathscr{T}$ serait très conséquente.
- L'ensemble des clés utilisées $\mathscr{C}_u$ est souvent petit par rapport à l'ensemble de toutes les clés possibles $\mathscr{C}$, donc une grande partie de l'espace mémoire alloué à $\mathscr{T}$ serait gaspillé.

Pour contourner cette difficulté, on fixe un entier $m \in \mathbb{N}^*$ et une **fonction de hachage** $h : \mathscr{C} \to [\![0, m-1]\!]$: il s'agit d'une fonction qui à chaque clé possible associe un entier dans $h : \mathscr{C} \to [\![0, m-1]\!]$. Chaque association (c, v) du dictionnaire est stockée dans la case d'indice $h(c)$ du tableau $\mathscr{T}$.

Cependant, un nouveau problème apparaît : le cardinal de $\mathscr{C}$ étant en général beaucoup plus grand que m, la fonction h ne peut pas être injective, donc il existe des couples $(c_1, c_2) \in \mathscr{C}^2$ avec $c_1 \neq c_2$ et $h(c_1) = h(c_2)$; un tel couple (c_1, c_2) est appelé une **collision**. Si notre dictionnaire contient deux associations (c_1, v_1) et (c_2, v_2), alors elles devraient être stockées dans la même case du tableau $\mathscr{T}$.

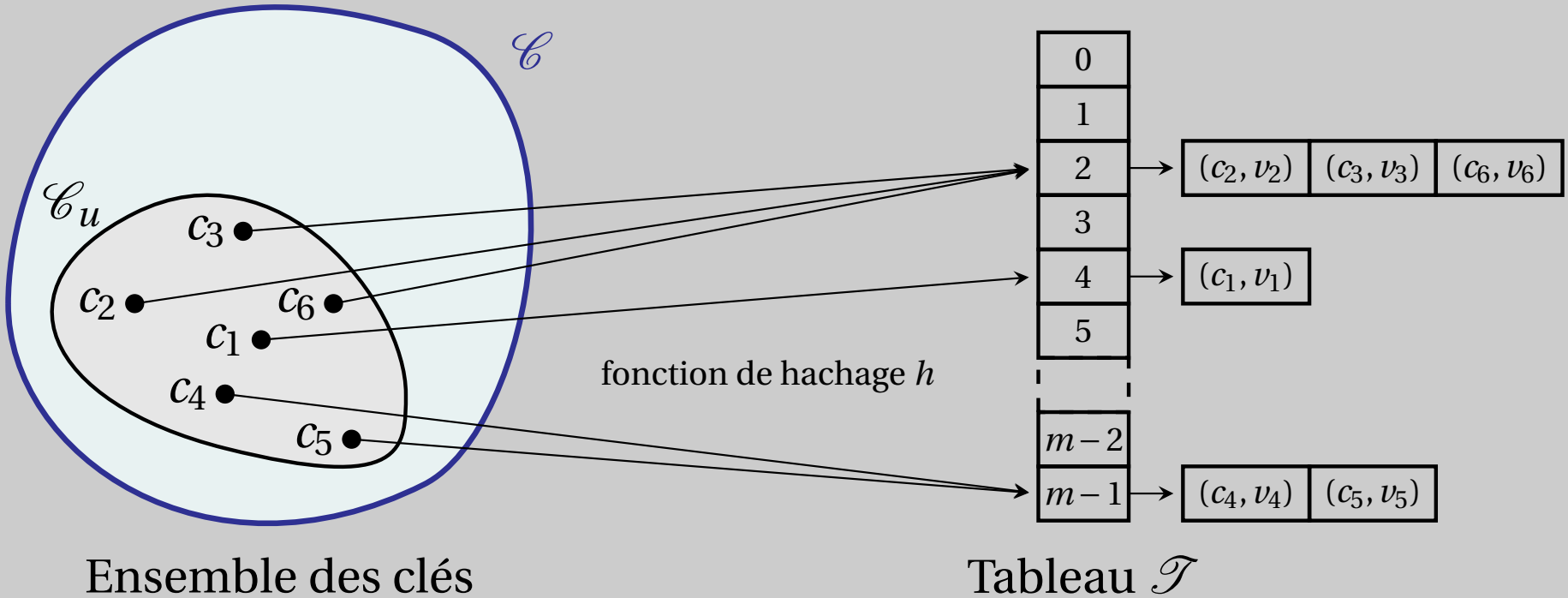
Il existe diverses méthodes, plus ou moins complexes pour gérer les collisions : on se contentera d'étudier les deux plus courantes.

I.C.1- Résolution des collisions par chainage

Lorsque deux clés c_1 et c_2 sont en collision, une solution consiste à stocker les associations (c_1, v_1) et (c_2, v_2) dans la même case du tableau $\mathscr{T}$, par exemple avec une liste.

Illustration

On peut représenter la résolution par chainage avec le schéma ci-dessous.



Avec ce choix, les opérations élémentaires se déroulent de la manière suivante.

- Pour ajouter une nouvelle association (c, v) avec $c \in \mathscr{C} \setminus \mathscr{C}_u$ et $v \in \mathscr{V}$, on ajoute le couple (c, v) dans la liste stockée à l'emplacement $h(c)$ du tableau $\mathscr{T}$.
- Pour supprimer une association existante (c, v), on la cherche dans la liste stockée à l'emplacement $h(c)$ du tableau $\mathscr{T}$ pour la supprimer de cette dernière.
- Pour tester l'existence d'une association avec une clé donnée $c \in \mathscr{C}$, on cherche dans la liste stockée à l'emplacement $h(c)$ du tableau $\mathscr{T}$ s'il existe un élément de la forme (c, v) avec $v \in \mathscr{V}$.
- Pour lire la valeur associée à une clé utilisée $c \in \mathscr{C}_u$, on cherche dans la liste stockée à l'emplacement $h(c)$ du tableau $\mathscr{T}$ l'unique élément de la forme (c, v) avec $v \in \mathscr{V}$, puis on renvoie la valeur v.

Remarques 3

a) On observe qu'avec cette méthode, le nombre d'associations n'est pas limité par la taille du tableau $\mathscr{T}$, car chaque case du tableau peut contenir autant d'éléments que nécessaire.

b) Cependant, nous verrons dans la partie suivante que si le nombre d'associations se rapproche trop de la taille du tableau $\mathscr{T}$, alors la complexité des quatre opérations ci-dessus n'est plus satisfaisante.

Exercice 1

On considère un tableau $\mathscr{T}$ de taille $m = 9$ et la fonction de hachage $h : k \mapsto k \bmod 9$. Réaliser l'insertion successive des clés

$$5,\ 28,\ 19,\ 15,\ 20,\ 33,\ 12,\ 17,\ 10$$

dans la table de hachage $\mathscr{T}$ dans le cas où les collisions sont résolues par chaînage.

0
1
2
3
4
5
6
7
8

I.C.2- Résolution des collisions par adressage ouvert

L'adressage ouvert consiste dans le cas d'une collision à stocker la nouvelle association dans une autre case du tableau $\mathscr{T}$. La recherche d'un emplacement libre est appelée **sondage**.

Plusieurs méthodes sont possibles.

- Le sondage linéaire : on fixe un pas $r \in [\![1, m-1]\!]$ et on cherche un emplacement libre dans le tableau $\mathscr{T}$ parmi les cases d'indice

$$h_k(c) = \big(h(c) + rk\big) \bmod m \quad \text{avec} \quad k = 0, \dots, m-1.$$

- Le sondage quadratique : on fixe deux constantes $(r, s) \in [\![0, m-1]\!] \times [\![1, m-1]\!]$ et on cherche un emplacement libre dans le tableau $\mathscr{T}$ parmi les cases d'indice

$$h_k(c) = \big(h(c) + rk + sk^2\big) \bmod m \quad \text{avec} \quad k = 0, \dots, m-1.$$

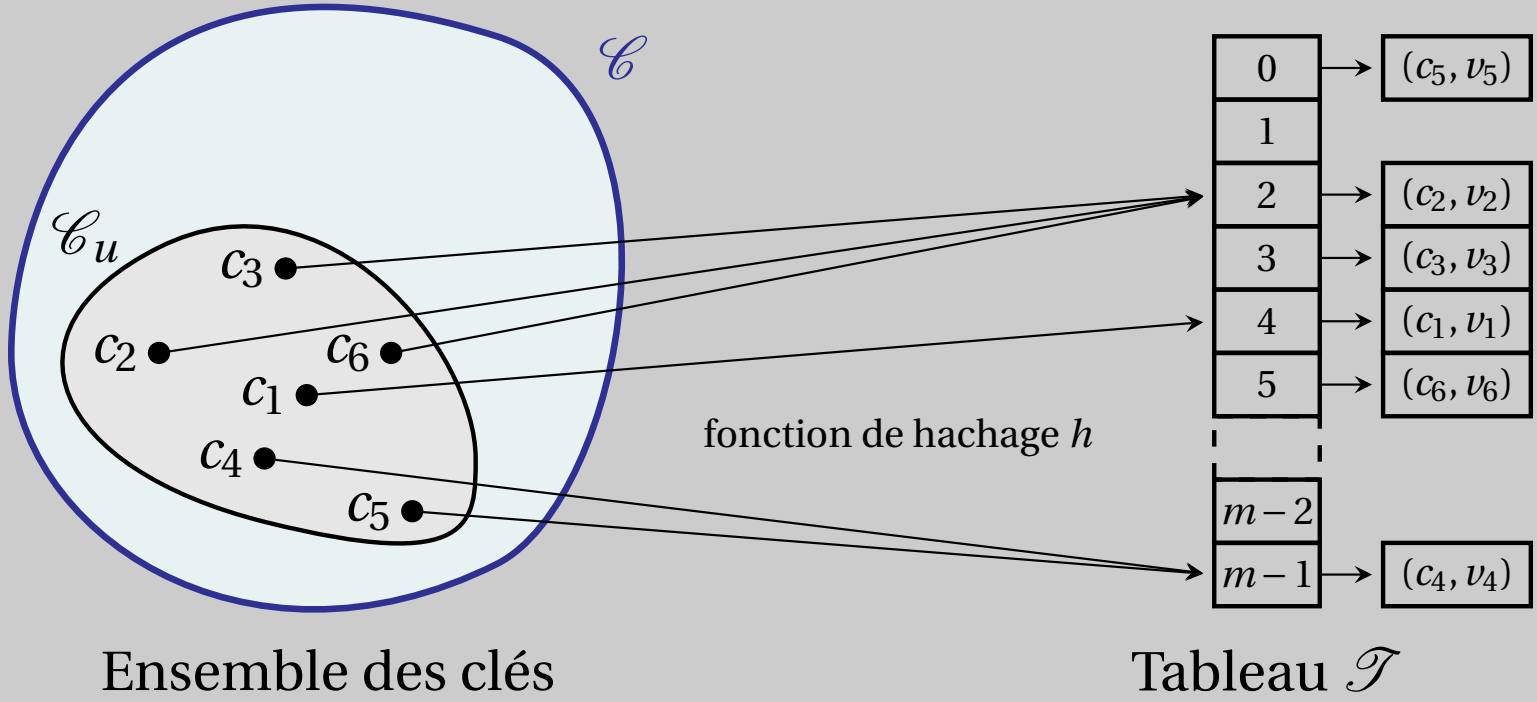
- Le double hachage : on considère une seconde fonction de hachage $h' : \mathscr{C} \to [\![0, m-1]\!]$ et on cherche un emplacement libre dans le tableau $\mathscr{T}$ parmi les cases d'indice

$$h_k(c) = \big(h(c) + kh'(c)\big) \bmod m \quad \text{avec} \quad k = 0, \dots, m-1.$$

Chacune des méthodes précédentes a des avantages et des inconvénients. Par exemple, le sondage linéaire est facile à implémenter, mais il favorise l'apparition de longues successions de cases occupées, augmentant ainsi le temps moyen de recherche d'une clé.

Illustration

On peut représenter la résolution par adressage ouvert en utilisant un sondage linéaire de pas $r = 1$ avec le schéma ci-dessous.



Remarque 4

Dans la résolution des collisions par adressage ouvert, le nombre d'associations est limité par la taille du tableau $\mathscr{T}$.

Exercice 2

On considère un tableau $\mathscr{T}$ de taille $m = 9$ et la fonction de hachage $h : k \mapsto k \bmod 9$. Réaliser l'insertion successive des clés

$$5,\ 28,\ 19,\ 15,\ 20,\ 33,\ 12,\ 17,\ 10$$

dans la table de hachage $\mathscr{T}$ dans le cas où les collisions sont résolues par adressage ouvert en utilisant un sondage linéaire de pas $r = 1$.

0
1
2
3
4
5
6
7
8

L'implémentation précédente dépend essentiellement de deux paramètres :

- l'entier $m \in \mathbb{N}^*$ donnant la taille du tableau $\mathscr{T}$;
- la fonction de hachage $h : \mathscr{C} \to [\![0, m-1]\!]$.

Dans cette partie, on étudie quelques-unes des contraintes que doivent vérifier ces paramètres afin que l'implémentation précédente soit efficace en termes de complexité.

Dans la suite, dans un souci de simplicité, on se limite à la résolution des collisions par chainage pour nos explications. Dans ce cas, on observe que les complexités des quatre opérations élémentaires se ramène essentiellement aux opérations analogues sur les listes. Pour les listes de Python, rappelons que l'on considère que la complexité dans le pire des cas pour ajouter un élément ou pour accéder à une valeur est constante ; tandis que celle pour supprimer un élément ou pour tester la présence d'une valeur est linéaire en la longueur de la liste.

On en déduit que les choix du paramètre m et de la fonction de hachage h doivent permettre de minimiser la taille des listes apparaissant dans le tableau, ce qui revient à minimiser le nombre de collisions.

I.D.1- La fonction de hachage

La construction d'une « bonne » fonction de hachage est un problème très complexe que nous n'aborderons pas. Précisons simplement que l'on souhaite que $h(c)$ se calcule en temps constant (pour des raisons d'efficacité) et que la fonction h ait une distribution la plus uniforme possible, i.e

$$\forall (k,\ell) \in [\![0, m-1]\!]^2, \quad \text{Card}\left(\{c \in \mathscr{C} \mid h(c) = k\}\right) \approx \text{Card}\left(\{c \in \mathscr{C} \mid h(c) = \ell\}\right).$$

On peut vérifier que cette dernière condition permet de minimiser la probabilité d'avoir une collision si les clés sont choisies aléatoirement et uniformément dans $\mathscr{C}$.

Remarque 5

Les fonctions de hachage ont d'autres applications. Par exemple, un site sécurisé ne stocke pas directement les mots de passe de ses utilisateurs, mais il conserve le résultat obtenu par l'application d'une fonction de hachage h_{crypt}. Si un pirate récupère cette information, il sera très difficile de l'exploiter, car la fonction h_{crypt} est construite de sorte qu'il soit presque impossible de déterminer les antécédents d'un entier par h_{crypt} (et en particulier le mot de passe).

I.D.2- La taille du tableau

Si l'on choisit la fonction de hachage avec la propriété décrite ci-dessus, on en déduit que chaque case du tableau contient une liste dont la longueur est en moyenne

$$\alpha = \frac{\text{Card}(\mathscr{C}_u)}{m} = \frac{\text{Nombre de clés utilisées}}{\text{Taille du tableau } \mathscr{T}}.$$

En pratique, on en déduit que la complexité en moyenne des quatre opérations sur un dictionnaire est un $O(\alpha)$.

Pour contrôler cette dernière complexité, on peut imposer une borne maximale $\alpha_{\max}$ sur la taille de α. Si le nombre de clés utilisées augmente de sorte que $\alpha > \alpha_{\max}$, on crée un nouveau tableau plus grand (la taille est doublée en général). On peut vérifier que ces choix permettent d'aboutir à une complexité que l'on peut considérer comme constante pour les quatre opérations de base sur un dictionnaire.

Remarque 6

En termes de complexité, le cas le plus défavorable se produit lorsque l'image de toutes les clés utilisées par la fonction de hachage est la même : tous les couples seront stockés dans la même case du tableau. Dans ce cas, tester la présence d'une association se fera dans le pire des cas avec une complexité linéaire (car il faut parcourir tous les éléments de la liste). Cependant, la probabilité que de « nombreuses clés utilisées » aient la même image par la fonction de hachage est extrêmement faible, ce qui nous amène à considérer que cet évènement est négligeable.

Le principe général de la programmation dynamique est de résoudre un problème en combinant des solutions de ses sous-problèmes. Cette idée directrice est proche de celle de la technique « diviser pour régner » vue en première année, mais avec une différence majeure : la programmation dynamique s'applique même si les sous-problèmes ne sont pas indépendants (ce que l'on appelle le **chevauchement des sous-problèmes**).

On considère un damier à m lignes et n colonnes. En partant du coin supérieur gauche, on souhaite déterminer le nombre $C(m,n)$ de chemins permettant de se rendre au coin inférieur droit en utilisant uniquement des déplacements d'une case vers le bas, d'une case vers la droite ou d'une case en diagonale vers le bas à droite.

Pour résoudre ce problème avec la programmation dynamique, on va étudier les trois sous-problèmes obtenus en distinguant les chemins étudiés en fonction de leur dernier déplacement.

- Cas 1 : le dernier déplacement est vers le bas ; il y a $C(m-1,n)$ chemins de cette forme.
- Cas 2 : le dernier déplacement est vers la droite ; il y a $C(m,n-1)$ chemins de cette forme.
- Cas 3 : le dernier déplacement est en diagonale ; il y a $C(m-1,n-1)$ chemins de cette forme.

On en déduit la formule de récurrence

$$\forall m, n \in \mathbb{N} \setminus \{0, 1\}, \quad C(m, n) = C(m-1, n) + C(m, n-1) + C(m-1, n-1).$$

De plus, si $m = 1$ ou $n = 1$, il n'y a qu'un seul chemin possible, donc on en déduit les valeurs initiales suivantes.

$$\forall (m, n) \in (\mathbb{N}^*)^2, \quad C(m, 1) = C(1, n) = 1.$$

Nous allons voir deux méthodes pour calculer ce nombre avec la programmation dynamique.

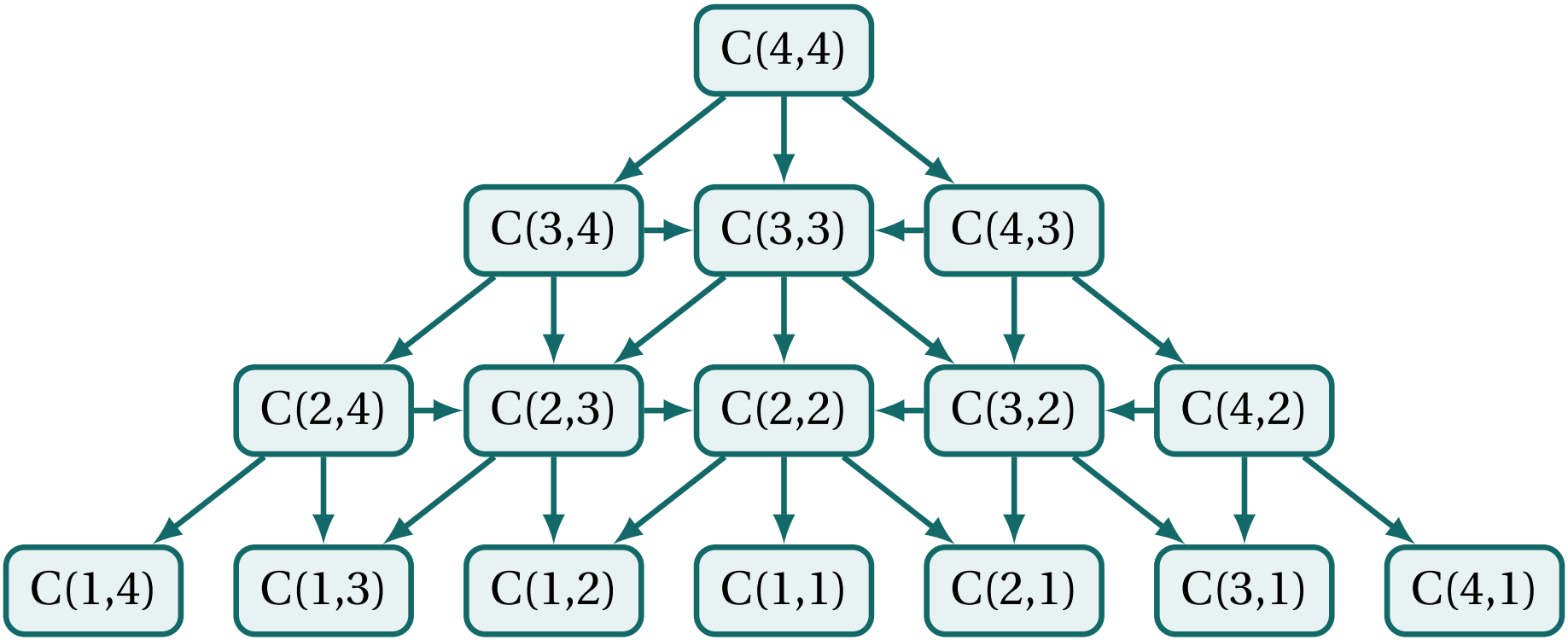
La **méthode descendante** (aussi appelée **de haut en bas**) consiste à partir directement du problème initial et à le découper en plusieurs sous-problèmes que l'on va résoudre par récursivité.

Une première idée serait d'utiliser la fonction ci-dessous.

```
1 def nb_chemin(m:int, n:int) -> int:
2     if m == 1 or n == 1:
3         return(1)
4     else:
5         return(nb_chemin(m-1,n) + nb_chemin(m,n-1) +
          ↪  nb_chemin(m-1,n-1))
```

En pratique, cette fonction est inutilisable si m et n ne sont pas suffisamment petits. Par exemple, mon ordinateur n'a pas réussi à terminer l'exécution de `nb_chemin(15,15)` au bout de plusieurs minutes.

Pour comprendre le problème avec la fonction récursive précédente, on peut représenter l'ensemble des appels récursifs à la fonction `nb_chemin` lorsque l'on essaye de calculer $C(4,4)$ avec le graphe de dépendance suivant.



On constate avec le graphe ci-dessus que le nombre $C(3,3)$ est calculé 3 fois, tandis que $C(2,2)$ est calculé 13 fois. On peut également vérifier avec l'ordinateur que le calcul $C(10,10)$ avec l'algorithme ci-dessus calculera 265 729 fois le nombre $C(2,2)$. En fait, on peut démontrer que le calcul de $C(n,n)$ via la fonction `nb_chemin` définie ci-dessus a une complexité temporelle qui est exponentielle.

Il est relativement simple de contourner le problème précédent en utilisant la **mémoïsation** : cette méthode consiste à associer un dictionnaire à notre fonction afin de mémoriser les résultats des calculs intermédiaires pour ne les faire qu'une fois.

Appliquons cette idée pour modifier notre fonction précédente.

```
1  dico = {}
2
3  def nb_chemin(m:int, n:int) -> int:
4      if (m,n) not in dico:
5          if m == 1 or n == 1:
6              dico[(m,n)] = 1
7          else:
8              dico[(m,n)] = nb_chemin(m-1,n) + nb_chemin(m,n-1) +
                ↪  nb_chemin(m-1,n-1)
9      return(dico[(m,n)])
```

La complexité temporelle et spatiale de cette nouvelle fonction est en $O(mn)$.

La mémoïsation permet d'effectuer un compromis entre complexité temporelle et complexité spatiale. En augmentant la complexité spatiale (mémorisation des résultats intermédiaires), nous avons réduit significativement la complexité en temps. Dans notre cas, l'optimisation est très intéressante : en augmentant la complexité spatiale de manière raisonnable, elle permet d'éviter une complexité en temps exponentielle qui rendait notre première fonction peu utile.

Bilan

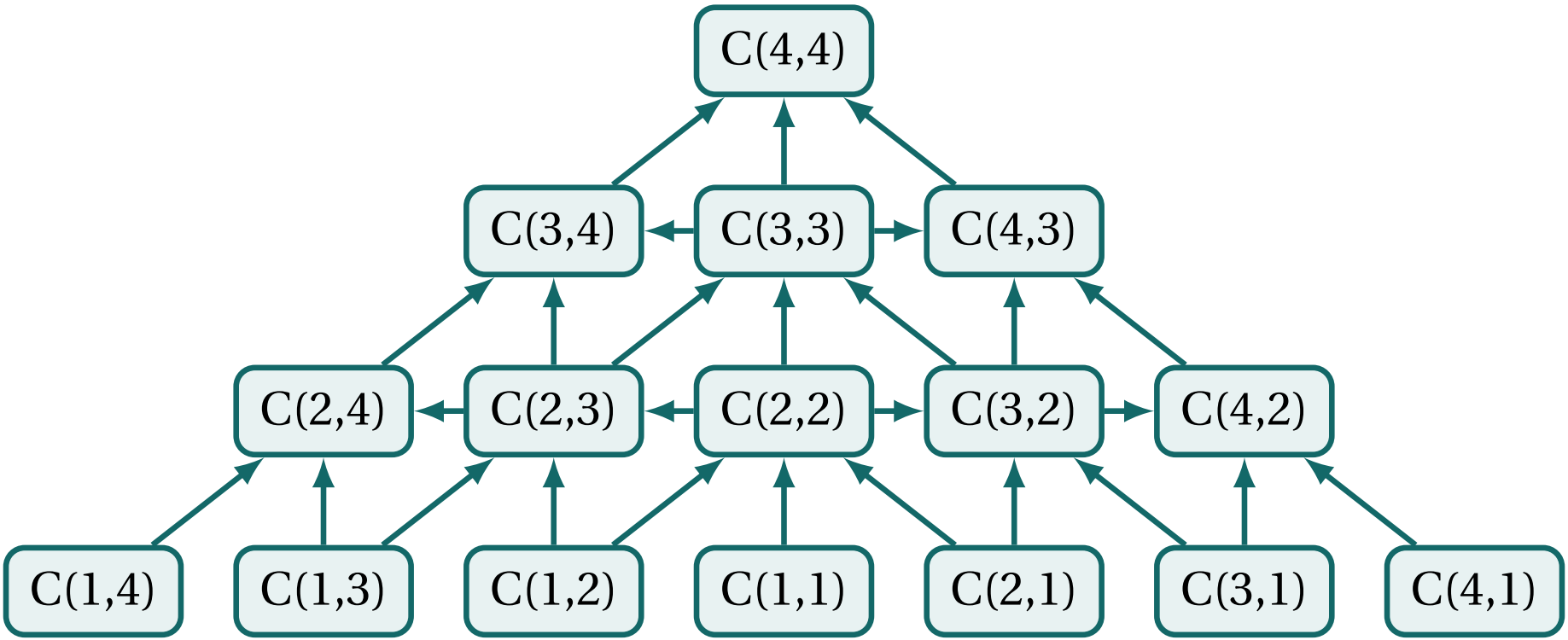
L'utilisation de la méthode descendante (avec mémoïsation) présente plusieurs avantages :

- l'algorithme est relativement simple à écrire et à comprendre en général ;
- l'algorithme gère automatiquement le choix des sous-problèmes à résoudre et leurs chevauchements.

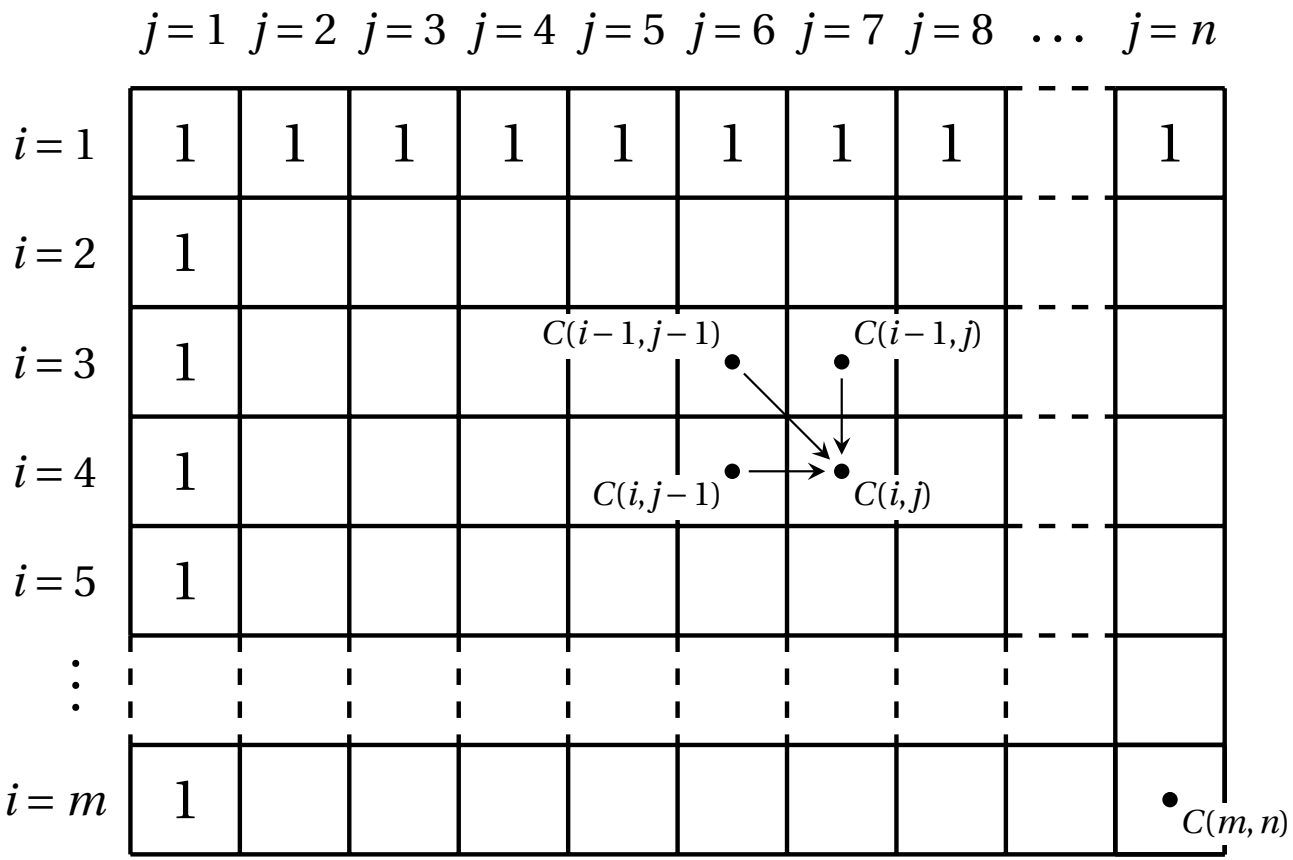
Elle présente néanmoins un inconvénient : les relations de dépendances étant gérées par l'algorithme directement, il est nécessaire de mémoriser les résultats de tous les sous-problèmes résolus durant l'intégralité de l'exécution, car l'algorithme ne peut pas détecter les valeurs sauvegardées en mémoire qui ne lui sont plus utiles. Ce défaut peut poser des difficultés lorsque la résolution avec la méthode descendante utilise une complexité spatiale trop importante.

La **méthode ascendante** (aussi appelée **de bas en haut**) consiste à utiliser un algorithme itératif pour résoudre les sous-problèmes en partant des plus petits et en remontant jusqu'au problème initial.

Pour le problème étudié, l'application de cette méthode pour calculer $C(4,4)$ est résumée par le graphe suivant.



Pour mettre en œuvre la méthode ascendante, on mémorise souvent les résultats intermédiaires dans un tableau. Dans notre cas, il est naturel d'utiliser un tableau bidimensionnel de taille $m \times n$ que nous allons remplir en utilisant les valeurs initiales et la relation de récurrence dont nous disposons.



Un point crucial est de choisir l'ordre de remplissage du tableau en tenant compte des relations de dépendance entre les différentes cases (représentées ci-dessus par les trois flèches) : on ne peut que calculer la valeur dans la case d'indice (i,j) une fois que l'on connait celles des cases d'indice $(i-1,j)$, $(i,j-1)$ et $(i-1,j-1)$.

Exercice 3

Déterminer la valeur de $C(4,7)$ en utilisant le tableau ci-dessous.

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
$i=1$							
$i=2$							
$i=3$							
$i=4$							

Dans notre cas, on peut par exemple choisir de le remplir ligne par ligne et de gauche à droite (d'autres choix sont possibles). On aboutit à l'algorithme suivant.

```python
import numpy as np

def nb_chemin(m:int, n:int) -> int:
    tab = np.ones((m+1,n+1), dtype=int)
    for i in range(2,m+1):
        for j in range(2,n+1):
            tab[i,j] = tab[i-1,j] + tab[i,j-1] + tab[i-1,j-1]
    return(tab[m,n])
```

La complexité temporelle et spatiale de cet algorithme est en $O(mn)$.

Sur cet exemple, on peut améliorer la complexité spatiale, car on constate que pour remplir une ligne du tableau, on n'a uniquement besoin de connaitre la ligne précédente. Il n'est donc pas nécessaire de sauvegarder l'intégralité du tableau, mais seulement la dernière ligne calculée, comme dans la fonction ci-dessous.

```python
import numpy as np

def nb_chemin(m:int, n:int) -> int:
    tab = np.ones((2,n+1), dtype=int)
    for i in range(2,m+1):
        for j in range(2,n+1):
            tab[1,j] = tab[0,j] + tab[1,j-1] + tab[0,j-1]
        tab[0] = tab[1]
    return(tab[0,n])
```

La complexité temporelle de cette dernière fonction est toujours en $O(mn)$, mais sa complexité spatiale est à présent en $O(n)$. On peut même encore réduire cette dernière complexité à $O(\min(m,n))$ en choisissant de remplir le tableau colonne par colonne dans le cas où $m < n$.

Bilan

L'utilisation de la méthode ascendante permet dans certains problèmes de diminuer la complexité en espace par rapport à la méthode descendante.

Par contre, elle peut s'avérer difficile à mettre en œuvre si les relations de dépendance entre les sous-problèmes sont complexes : choisir l'ordre de remplissage du tableau en mémoire risque d'être trop laborieux.

Lorsque l'on implémente un algorithme dynamique, il est difficile de faire un choix entre la méthode descendante ou la méthode ascendante. Néanmoins, il est important de retenir qu'aucune des deux approches n'est meilleure que l'autre : il faut s'adapter à la situation, au problème posé et aux contraintes à respecter.

Un problème d'optimisation consiste à trouver une solution optimale (selon un critère donné) parmi toutes les solutions d'un problème. Plus formellement, si $\mathscr{S}$ est un ensemble fini décrivant toutes les solutions d'un problème, alors on cherche à minimiser (ou maximiser) une fonction $\varphi : \mathscr{S} \to \mathbb{R}$ et à déterminer une solution $s \in \mathscr{S}$ permettant d'atteindre ce minimum (ou maximum). Par exemple, dans un graphe pondéré par des nombres positifs, déterminer un chemin de longueur minimale entre deux sommets donnés du graphe est un problème d'optimisation (que l'on peut résoudre avec l'algorithme de Dijkstra).

Une première solution élémentaire serait de parcourir exhaustivement les éléments $s \in \mathscr{S}$ pour déterminer la valeur minimale (ou maximale) de $\varphi(s)$. Cette méthode est inefficace en général : le cardinal de $\mathscr{S}$ étant souvent exponentiel par rapport aux paramètres du problème, il en est de même pour la complexité temporelle de l'algorithme de parcours exhaustif de $\mathscr{S}$.

La programmation dynamique s'applique notamment à tous les problèmes d'optimisation vérifiant **la propriété de sous-structure optimale** (aussi appelé **principe d'optimalité de Bellman**) : une solution optimale du problème étudié s'obtient en combinant des solutions optimales de ses sous-problèmes. Dans ce cadre, la conception d'un algorithme de programmation dynamique est typiquement découpée en trois étapes.

1. Définir (souvent de manière récursive) la valeur d'une solution optimale.
2. Calculer la valeur d'une solution optimale.
3. Construire une solution optimale à partir des informations calculées.

Dans la suite de cette partie, on étudie quelques exemples de résolution de problème d'optimisation avec la programmation dynamique.

Dans cette partie, on étudie le problème du rendu de monnaie. Étant donné un système de monnaie (pièces et billets), on cherche le nombre minimal de pièces et de billets nécessaire pour réaliser une somme donnée.

Plus formellement, le système monétaire est représenté par un ensemble $\mathscr{M} = \{m_1, \ldots, m_n\}$ où $m_1, \ldots, m_n$ sont des entiers naturels non nul distincts que l'on suppose rangés dans l'ordre croissant. Pour être certain que toute valeur entière s'écrit comme une somme d'éléments de $\mathscr{M}$, on impose également que $m_1 = 1$. L'ensemble des manières de rendre la valeur $V \in \mathbb{N}$ est décrit par

$$\mathscr{S} = \left\{ (x_1, \ldots, x_n) \in \mathbb{N}^n \;\middle|\; \sum_{k=1}^{n} x_k m_k = V \right\}.$$

Le problème d'optimisation revient ainsi à minimiser la fonction $\varphi : \mathscr{S} \to \mathbb{N}$ définie par

$$\varphi : (x_1, \ldots, x_n) \mapsto \sum_{k=1}^{n} x_k.$$

Exemple 1

Considérons le système monétaire $\mathscr{M} = \{m_1, m_2, m_3\} = \{1, 3, 4\}$ et la somme $V = 6$. Dans ce cas simple, on peut écrire explicitement l'ensemble de toutes les possibilités pour rendre la valeur 6 :

$$V = 6 \times m_1, \qquad V = 3 \times m_1 + 1 \times m_2, \qquad V = 2 \times m_1 + 1 \times m_3, \qquad V = 2 \times m_2.$$

On en déduit que

$$\mathscr{S} = \{(6,0,0), (1,3,0), (2,0,1), (0,2,0)\},$$

donc le nombre minimal de pièces dans ce cas est 2 et il y a une unique solution optimale qui est $(0,2,0)$.

III.A.1- Relation de récurrence pour la valeur d'une solution optimale

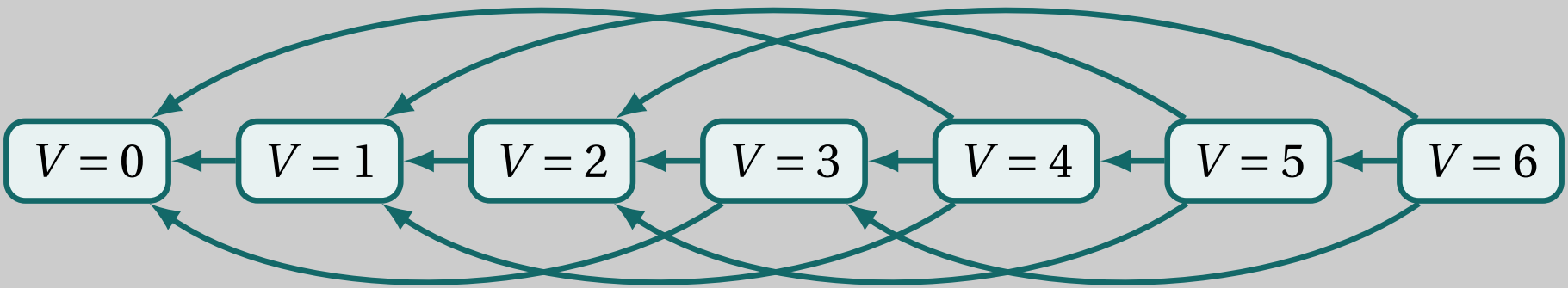
Pour résoudre ce problème avec la programmation dynamique, on note $r_{\mathscr{M}}(V)$ la valeur optimale pour rendre la valeur V avec le système monétaire $\mathscr{M}$. Pour rendre la valeur $V > 0$, il faut au moins une pièce m_i vérifiant la condition $m_i \leqslant V$. Une fois cette pièce choisie, la somme restante $V - m_i$ est strictement inférieure à V, donc la solution de ce sous-problème est $r_{\mathscr{M}}(V - m_i)$. On en déduit la relation

$$r_{\mathscr{M}}(V) = 1 + \min\left\{r_{\mathscr{M}}(V - m_i) \mid i \in [\![1, n]\!],\ \ m_i \leqslant V\right\}.$$

D'autre part, on a la valeur initiale $r_{\mathscr{M}}(0) = 0$.

Illustration

En reprenant l'exemple précédent et la relation ci-dessus, on peut réaliser le graphe de dépendance suivant entre les différents sous-problèmes.



III.A.2- Calcul de la valeur d'une solution optimale

On déduit de l'étude précédente les deux algorithmes ci-dessous.

```
1  # METHODE DESCENDANTE
2  dico = {0: 0}
3
4  def rendu_monnaie_desc(M:list, V:int) -> int:
5      if V not in dico:
6          res = V # Dans le pire des cas V = 1 + 1 + ... + 1
7          for m in M:
8              if m <= V:
9                  res = min(res, rendu_monnaie_desc(M, V-m))
10         dico[V] = 1 + res
11     return(dico[V])
```

```python
# METHODE ASCENDANTE
import numpy as np

def rendu_monnaie_asc(M:list, V:int) -> int:
    r = np.zeros(V+1, dtype=int)
    for k in range(1,V+1):
        r[k] = k
        for m in M:
            if m <= k:
                r[k] = min(r[k], r[k-m])
        r[k] += 1
    return(r[V])
```

Les deux fonctions ci-dessus ont une complexité temporelle en $O(nV)$ et une complexité spatiale en $O(V)$.

En remarquant que pour remplir une case du tableau, on n'a uniquement besoin des m_n cases précédentes, on peut réduire la complexité spatiale à $O(m_n)$ dans la fonction utilisant la méthode ascendante.

```python
import numpy as np

def rendu_monnaie_asc_bis(M:list, V:int) -> int:
    mod = M[-1] + 1
    r = np.zeros(mod, dtype=int)
    for k in range(1,V+1):
        r[k % mod] = k
        for m in M:
            if m <= k:
                r[k % mod] = min(r[k % mod], r[(k-m) % mod])
        r[k % mod] += 1
    return(r[V % mod])
```

Remarque 7

Cette fonction est plus économe en mémoire, mais nous verrons dans la partie suivante qu'elle ne permet pas de construire une solution optimale. Elle n'est donc intéressante que dans le cas où l'on cherche uniquement la valeur optimale (et pas une solution la réalisant).

III.A.3- Construction d'une solution optimale

Pour le moment, nous avons seulement déterminé la valeur (le nombre de pièces utilisées) d'une solution optimale pour tous les sous-problèmes de notre problème de départ, mais nous n'avons pas trouvé une solution optimale (les pièces utilisées). Pour en construire une à partir des informations calculées précédemment, il suffit de partir de la valeur optimale pour notre problème, puis de remonter le chemin emprunté en suivant notre relation de dépendance, jusqu'à arriver à une des cases initiales.

Exercice 4

Le tableau ci-dessous contient les informations calculées par la fonction `rendu_monnaie_asc` pour le système monétaire $\mathscr{M} = \{1,3,4\}$ et la valeur $V = 18$. Reconstruire à partir de celui-ci une solution optimale pour rendre la valeur $V = 18$.

V	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$r_{\mathscr{M}}(V)$	0	1	2	1	1	2	2	2	2	3	3	3	3	4	4	4	4	5	5

Pour implémenter cette démarche, on modifie la fonction `rendu_monnaie_asc` (ou `rendu_monnaie_desc`) pour qu'elle renvoie le tableau (ou le dictionnaire) contenant toute les informations calculées.

```python
import numpy as np

def rendu_monnaie(M:list, V:int):
    r = np.zeros(V+1, dtype=int)
    for k in range(1,V+1):
        r[k] = k
        for m in M:
            if m <= k:
                r[k] = min(r[k], r[k-m])
        r[k] += 1
    return(r)
```

On peut à présent mettre en œuvre la démarche décrite dans l'introduction de cette partie.

```
12 def pieces_opti(M:list, V:int) -> list:
13     r = rendu_monnaie(M, V)
14     sol_opt = [0] * len(M)
15     while V > 0:
16         for i in range(len(M)):
17             if M[i] <= V and r[V] == r[V-M[i]]+1:
18                 sol_opt[i] += 1
19                 V -= M[i]
20     return(sol_opt)
```

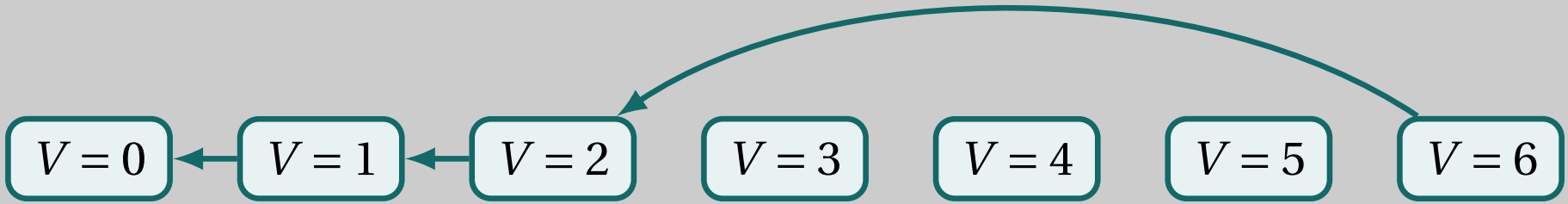
III.A.4- Comparaison avec la stratégie gloutonne

Lorsque les paramètres du problème sont trop grands, les algorithmes précédents ne sont plus assez efficaces. Dans ce cas, on peut utiliser une stratégie gloutonne. Tandis que l'utilisation de la programmation dynamique pour résoudre un problème d'optimisation s'appuie sur la résolution de tous les sous-problèmes, la stratégie gloutonne se contente d'en choisir un seul en suivant une heuristique (i.e. une stratégie permettant de faire un choix rapide, mais pas nécessairement optimal). En général, un algorithme glouton est plus rapide qu'un algorithme basé sur la programmation dynamique, mais en contre-partie, la solution qu'il construit n'est pas nécessairement optimale. Cependant, si l'heuristique est bien choisie, on peut espérer obtenir une solution « presque » optimale.

Dans le problème du rendu de monnaie, on peut s'appuyer sur la stratégie gloutonne consistant à utiliser la pièce de plus grande valeur inférieure à la somme à rendre.

Illustration

Le graphe ci-dessous représente l'application de la stratégie gloutonne pour rendre la valeur 6 avec le système monétaire $\mathscr{M} = \{m_1, m_2, m_3\} = \{1, 3, 4\}$.



Dans le cas ci-dessus, l'algorithme glouton donne un rendu de monnaie avec 3 pièces alors que la solution optimale n'en nécessite que 2.

On peut implémenter en Python la stratégie gloutonne décrite ci-dessus.

```python
def pieces_glouton(M:list, V:int) -> list:
    sol = [0] * len(M)
    for k in range(len(M)-1,-1,-1):
        sol[k] = V // M[k]
        V -= M[k] * sol[k]
    return(sol)
```

Cet algorithme a une complexité temporelle et spatiale en $O(n)$, ce qui est bien meilleure que les algorithmes basés sur la programmation dynamique (mais rappelons que la solution construite n'est pas nécessairement optimale).

Remarque 8

Pour le système monétaire des euros $\mathscr{M} = \{1,2,5,10,20,50,100,200\}$, on peut démontrer que l'algorithme glouton décrit ci-dessus fournit toujours la solution optimale.

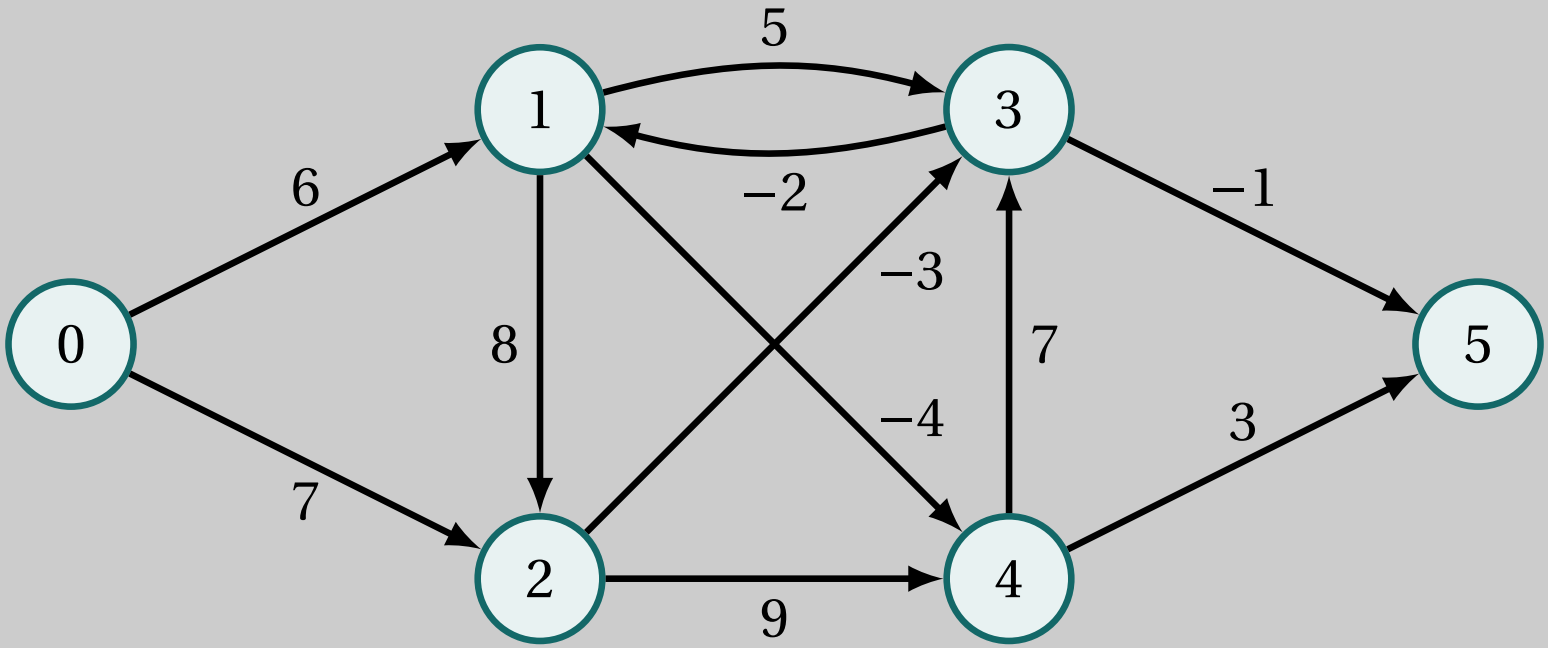
Dans cette partie, on étudie le problème consistant à déterminer un plus court chemin entre deux sommets d'un graphe orienté et pondéré par des entiers relatifs. Rappelons que l'algorithme de Dijkstra permet de résoudre ce problème lorsque le graphe est pondéré par des entiers positifs.

On suppose également que le graphe ne contient pas de circuits (chemins dont le nœud d'arrivée et le nœud de départ sont les mêmes) de poids strictement négatif. Sans cette condition, le problème serait mal posé : on pourrait fabriquer des chemins de poids arbitrairement petit en parcourant plusieurs fois un tel circuit.

Dans cette partie, nous allons décrire un graphe avec la liste de ses sommets S et la liste de ses arêtes A. Chaque arête est décrite par un triplet indiquant le sommet de départ, le sommet d'arrivée et le poids de l'arête.

Exemple 2

On peut par exemple considérer le graphe suivant.



En Python, le graphe précédent est représenté par les deux listes ci-dessous.

```
1 S = [0, 1 ,2 ,3 ,4 ,5]
2 A = [(0,1,6),(0,2,7),(1,2,8),(1,3,5),(1,4,-4),
   ↪  (2,3,-3),(2,4,9),(3,1,-2),(3,5,-1),(4,3,7),(4,5,3)]
```

III.B.1- Relation de récurrence pour la valeur d'une solution optimale

Pour résoudre ce problème avec la programmation dynamique, on fixe un sommet source s et on note $d_s(k,t)$ la distance de s jusqu'à un sommet t avec un chemin qui contient au plus k arêtes. On peut démontrer que l'absence de circuit de poids négatif implique que le plus court chemin de s à t est de poids $d_s(\text{Card}(S)-1,t)$.

Étant donné un entier $k>0$ et en considérant un chemin $a_1 \cdots a_\ell$ avec $\ell \leqslant k$ de longueur minimale parmi tous les chemins allant de s à t contenant au plus k arêtes, on a deux possibilités.

- Cas 1 : si $\ell < k$, alors $d_s(k,t) = d_s(k-1,t)$.
- Cas 2 : si $\ell = k$, alors en notant u le sommet de départ de l'arête a_ℓ, on a $d_s(k,t) = d_s(k-1,u) + \text{poids}(a_\ell)$.

On en déduit la formule de récurrence

$$d_s(k,t) = \min\Big(d_s(k-1,t), \min\big\{d_s(k-1,u) + \text{poids}(a) \mid a \in A \text{ est une arête de } u \text{ vers } t\big\}\Big).$$

En utilisant la convention que $d_s(k,t) = +\infty$ s'il n'existe pas de chemin de s à t contenant au plus k arêtes, on a les valeurs initiales

$$d_s(0,t) = \begin{cases} 0 & \text{si} \quad t = s \\ +\infty & \text{si} \quad t \neq s. \end{cases}$$

III.B.2- Calcul de la valeur d'une solution optimale

Les formules précédentes nous permettent de calculer les nombres $d_s(k,t)$ pour tout $k \in [\![0, \text{Card}(S)-1]\!]$ et tout sommet t.

Exemple 3

En reprenant le graphe donné dans l'exemple de l'introduction et en partant de $s = 0$, on obtient les valeurs ci-dessous pour le nombre $d_s(k, t)$. On en déduit que le plus court chemin de $s = 0$ à $t = 5$ est de poids 1.

	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
$k=0$	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$k=1$	0	6	7	$+\infty$	$+\infty$	$+\infty$
$k=2$	0	6	7	4	2	$+\infty$
$k=3$	0	2	7	4	2	3
$k=4$	0	2	7	4	-2	3
$k=5$	0	2	7	4	-2	1

En Python, on peut utiliser la fonction ci-dessous se basant sur la méthode ascendante.

```python
import numpy as np
from math import inf # Valeur infini en Python

def dist_bellman_ford(S:list, A:list, dep:int, arr:int) -> int:
    d = np.zeros((len(S), len(S)), dtype = float)

    for sommet in S:
        if sommet != dep:
            d[0,sommet] = inf

    for k in range(1,len(S)):
        d[k] = d[k-1]
        for arete in A:
            arete_dep, arete_arr, arete_poids = arete
            d[k,arete_arr] = min(d[k,arete_arr], d[k-1,arete_dep] +
            ↪  arete_poids)

    return(d[len(S)-1,arr])
```

Cette fonction a une complexité temporelle en $O(\text{Card}(S)\,\text{Card}(A))$ et une complexité spatiale en $O(\text{Card}(S)^2)$.

Remarque 9

En observant que pour remplir une ligne du tableau, on n'a uniquement besoin de la ligne précédente, on pourrait réduire la complexité spatiale de la fonction précédente à $O(\text{Card}(S))$.

III.B.3- Construction d'une solution optimale

En partant de la valeur optimale pour notre problème et en remontant dans le tableau en suivant notre relation de dépendance, on peut reconstruire un chemin de poids minimal.

Exemple 4

En utilisant les informations calculées dans le tableau précédent, on obtient que le chemin de longueur minimal du sommet d'indice 0 au sommet d'indice 5 est $0 \to 2 \to 3 \to 1 \to 4 \to 5$.

	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	
$k=0$	(0)	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	
$k=1$	0	6	(7)	$+\infty$	$+\infty$	$+\infty$	arête $0 \to 2$
$k=2$	0	6	7	(4)	2	$+\infty$	arête $2 \to 3$
$k=3$	0	(2)	7	4	2	3	arête $3 \to 1$
$k=4$	0	2	7	4	(−2)	3	arête $1 \to 4$
$k=5$	0	2	7	4	−2	(1)	arête $4 \to 5$

En Python, en modifiant la dernière ligne de la fonction précédente pour qu'elle retourne le tableau d entièrement, on peut déterminer un chemin de longueur minimale avec l'algorithme suivant.

```
18 def chemin_bellman_ford(S:list, A:list, dep:int, arr:int) -> list:
19     d = dist_bellman_ford(S, A, dep, arr)
20     chemin = [arr]
21 
22     k = len(S)-1
23     while chemin[-1] != dep:
24         if d[k,chemin[-1]] < d[k-1,chemin[-1]]:
25             for arete in A:
26                 arete_dep, arete_arr, arete_poids = arete
27                 if arete_arr == chemin[-1]:
28                     if d[k,arete_arr] == d[k-1,arete_dep] + arete_poids:
29                         chemin.append(arete_dep)
30         k -= 1
31 
32     chemin.reverse()
33     return(chemin)
```

La distance d'édition (ou distance de Levenshtein) est une mesure de la différence entre deux chaînes de caractères. Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne à une autre.

Exemple 5

On peut passer du mot `chat` au mot `chien` avec les opérations

$$\texttt{chat} \xrightarrow[\text{d'une lettre}]{\text{Insertion}} \texttt{chatn} \xrightarrow[\text{d'une lettre}]{\text{Remplacement}} \texttt{chitn} \xrightarrow[\text{d'une lettre}]{\text{Remplacement}} \texttt{chien}.$$

On en déduit que la distance d'édition entre ces deux mots est au plus 3 (et on vérifiera que 3 est la distance d'édition entre ces deux mots).

III.C.1- Relation de récurrence pour la valeur d'une solution optimale

Nous allons calculer la distance d'édition entre deux mots $a = a_1 \cdots a_p$ et $b = b_1 \cdots b_q$. Pour résoudre ce problème avec la programmation dynamique, on note $d(i,j)$ la distance d'édition entre les mots $a_1 \cdots a_i$ et $b_1 \cdots b_j$. En particulier, la distance d'édition entre les mots a et b est $d(p,q)$.

Étant donné deux entiers $i \in [\![1, p]\!]$ et $j \in [\![1, q]\!]$ et en considérant une suite de longueur minimale de transformations permettant de passer du mot $a_1 \cdots a_i$ au mot $b_1 \cdots b_j$, on a plusieurs possibilités.

- Cas 1 : si la lettre a_i est supprimée, alors $d(i,j) = d(i-1,j) + 1$.
- Cas 2 : si la lettre b_j est ajoutée, alors $d(i,j) = d(i,j-1) + 1$.
- Cas 3 : si la lettre a_i est remplacée par la lettre b_j, alors $d(i,j) = d(i-1,j-1) + 1$.
- Cas 4 : si on a $a_i = b_j$, alors $d(i,j) = d(i-1,j-1)$.

On en déduit la formule de récurrence

$$d(i,j) = \begin{cases} \min\left(d(i-1,j)+1, d(i,j-1)+1, d(i-1,j-1)+1\right) & \text{si} \quad a_i \neq b_j \\ \min\left(d(i-1,j)+1, d(i,j-1)+1, d(i-1,j-1)\right) & \text{si} \quad a_i = b_j. \end{cases}$$

D'autre part, on a directement les valeurs initiales $d(i,0) = i$ et $d(0,j) = j$.

III.C.2- Calcul de la valeur d'une solution optimale

Les formules précédentes nous permettent de calculer les nombres $d(i,j)$ pour tout $(i,j) \in [\![0,p]\!] \times [\![0,q]\!]$.

Exemple 6

En appliquant les formules précédentes avec les mots `chat` et `chien`, on obtient que la distance d'édition entre ces deux mots est 3.

	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
$i=0$	0	1	2	3	4	5
$i=1$	1	0	1	2	3	4
$i=2$	2	1	0	1	2	3
$i=3$	3	2	1	1	2	3
$i=4$	4	3	2	2	2	3

En Python, on peut utiliser la fonction ci-dessous se basant sur la méthode ascendante.

```python
import numpy as np

def dist_edition(a:str, b:str) -> int:
    p, q = len(a), len(b)
    d = np.zeros((p+1,q+1), dtype=int)

    for i in range(0,p+1):
        d[i,0] = i
    for j in range(0,q+1):
        d[0,j] = j

    for i in range(1,p+1):
        for j in range(1,q+1):
            if a[i-1] == b[j-1]:
                d[i,j] = min(d[i-1,j]+1, d[i,j-1]+1, d[i-1,j-1])
            else:
                d[i,j] = min(d[i-1,j]+1, d[i,j-1]+1, d[i-1,j-1]+1)

    return(d[p,q])
```

Cette fonction a une complexité temporelle et spatiale en $O(pq)$.

III.C.3- Construction d'une solution optimale

En partant de la valeur optimale pour notre problème et en remontant dans le tableau en suivant notre relation de dépendance, on peut reconstruire une suite de longueur minimale des transformations à utiliser pour passer du mot a au mot b.

Exemple 7

En utilisant les informations calculées dans le tableau précédent, on peut retrouver une suite de 3 transformations pour passer du mot `chat` au mot `chien`.