TP Chapitre 10 : Programmation dynamique

Objectifs du TP

Objectifs du TP. À l'issue de cette séance, tu dois être capable de :

— décomposer un problème en sous-problèmes et exprimer sa solution par une formule de récurrence ;

— expliquer le principe de la **mémoïsation** et le mettre en œuvre avec une liste ou un dictionnaire ;

— écrire un algorithme de **programmation dynamique** (version itérative, remplissage de bas en haut) ;

— comparer les coûts en temps d'une version récursive naïve (exponentielle) et de sa version mémoïsée ;

— connaître le coût en temps $O(s \times |P|)$ du rendu de monnaie dynamique et son coût en mémoire $O(s)$;

— appliquer ces techniques au comptage de chemins dans une grille et au rendu de monnaie avec des pièces réelles.

Consignes générales. Tous les exercices se traitent en Python (console ou fichier). Enregistre régulièrement ton travail. Attention : certaines fonctions récursives naïves (Fibonacci, rendu de monnaie) deviennent *extrêmement* lentes dès que les valeurs sont grandes — sauvegarde ton travail avant de les tester, puis utilise la version mémoïsée. Quand un programme ne fonctionne pas, lis le message d'erreur : il indique souvent la ligne et le type d'erreur. La partie B se fait *en autonomie* : tu écris des programmes entiers à partir d'une spécification, aucune ligne de code n'est fournie.

Partie A — Application guidée

Les exercices suivants se programment en Python. Ils sont classés du plus simple au plus difficile. Chaque exercice se termine par un *point de validation* (encadré vert) : le résultat exact que doit produire ton programme pour être juste. Complète les squelettes de code en respectant l'indentation.

Exercice A1 — Compter les chemins d'un livreur (tableau de sous-problèmes)

Un coursier à vélo part du point A (coin inférieur droit d'un quartier) pour rejoindre le point B (coin supérieur gauche). Il ne se déplace que *vers la gauche* ou *vers le haut*, le long des traits du quadrillage. Combien de chemins différents peut-il emprunter ?

On inscrit près de chaque intersection le nombre de chemins qui y mènent : pour atteindre une intersection, le dernier déplacement vient soit de l'intersection **à droite**, soit de l'intersection **en dessous** ; le nombre de chemins est donc la **somme** des deux. Les intersections du bord droit et du bord inférieur ne peuvent être atteintes que d'une seule façon : elles reçoivent 1. Chaque case ne dépend que de cases déjà calculées : chaque sous-problème est résolu une seule fois et son résultat est réutilisé.

```
def nb_chemins(nbligne, nbcol):
    tab = [[0] * (nbcol + 1) for _ in range(nbligne + 1)]
    for i in range(nbligne + 1):
        tab[i][nbcol] = 1
    for j in range(nbcol + 1):
        tab[nbligne][j] = 1
    for i in range(nbligne - 1, -1, -1):
        for j in range(nbcol - 1, -1, -1):
            # A COMPLETER : nombre de chemins vers cette intersection
```

```
    return tab
```

Point de validation

Vérifie ton programme : `nb_chemins(2, 3)[0][0]` doit renvoyer `10` (grille 2×3) ; `nb_chemins(5, 5)[0][0]` doit renvoyer `252` ; `nb_chemins(10, 10)[0][0]` doit renvoyer 184 756.

Exercice A2 — La suite de Fibonacci, version mémoïsée (liste)

La suite de Fibonacci est définie par $F(0) = 1$, $F(1) = 1$ et, pour tout $n \geq 2$, $F(n) = F(n-1)+F(n-2)$. La traduction récursive directe recalcule sans cesse les mêmes valeurs : pour $n = 30$, elle effectue des millions d'appels. La **mémoïsation** consiste à mémoriser les résultats déjà calculés pour ne pas les recalculer.

Complète la fonction : si le terme demandé est déjà dans la liste `memoisation`, on le renvoie directement ; sinon on le calcule par récurrence, on l'ajoute à la liste, puis on le renvoie.

```
memoisation = [1, 1]   # F(0) = F(1) = 1

def fibonacci_dynamique(n, memoisation):
    # A COMPLETER : si F(n) est deja memorise, le renvoyer directement
    # A COMPLETER : sinon le calculer, le memoriser, puis le renvoyer
```

Point de validation

Vérifie ton programme : `fibonacci_dynamique(5, memoisation)` doit renvoyer `8`, et la liste `memoisation` doit alors contenir `[1, 1, 2, 3, 5, 8]` ; `fibonacci_dynamique(30, memoisation)` doit renvoyer 1 349 269, et le calcul doit être quasi instantané.

Exercice A3 — Rendre la monnaie avec l'algorithme glouton (pièces réelles)

Un caissier doit rendre une somme avec *le moins de pièces possible*. L'**algorithme glouton** consiste à rendre à chaque étape la pièce de plus grande valeur inférieure ou égale à la somme restante. Les valeurs sont exprimées en **centimes** (l'euro vaut 100 centimes).

Complète la fonction : pour chaque pièce (de la plus grande à la plus petite), on la rend *tant que* la somme restante le permet.

```
def rendu_glouton(liste_pieces, somme):
    rendu = []
    reste = somme
    for piece in liste_pieces:
        # A COMPLETER : tant que la piece est utilisable, la rendre
    return rendu
```

Point de validation

Vérifie ton programme : avec `P = [50000, 20000, 10000, 5000, 2000, 1000, 500, 200, 100, 50, 20, 10, 5, 2, 1]` (le système de l'euro en centimes), `rendu_glouton(P, 29356)` doit renvoyer une liste de 9 pièces (pour 293,56 €) ; avec `[100, 50, 20, 10, 5, 2, 1]` et 42, elle doit renvoyer `[20, 20, 2]`.

Exercice A4 — Le nombre minimal de pièces garanti (programmation dynamique)

L'algorithme glouton ne donne *pas toujours* la solution optimale : avec les pièces $[10, 6, 1]$ et la somme 12, il rend $[10, 1, 1]$ (3 pièces), alors que $12 = 6 + 6$: 2 pièces suffisent. Pour garantir l'optimum, on utilise la **programmation dynamique** : on remplit une liste L de taille $s+1$, où $L[k]$ sera le nombre minimal de pièces pour la somme k. On initialise $L[k] = k$ (k pièces de 1), puis on met à jour *de bas en haut* : chaque valeur ne dépend que de valeurs déjà calculées.

Complète la fonction : pour chaque somme k, pour chaque pièce $p \leq k$, le nombre $1 + L[k-p]$ (une pièce p, puis le reste $k-p$) est une candidate pour $L[k]$; on garde le minimum.

```
def rendu_dyn(P, s):
    L = [k for k in range(s + 1)]  # initialisation : k pieces de 1
    for k in range(1, s + 1):
        for p in P:
            if p <= k:
                # A COMPLETER : mise a jour de L[k] (garder le minimum)
    return L[s]
```

Point de validation

Vérifie ton programme : `rendu_dyn([10, 6, 1], 12)` doit renvoyer `2` ; `rendu_dyn([6, 4, 1], 8)` doit renvoyer `2` (alors que le glouton en utilise 3) ; `rendu_dyn([278, 100, 6, 4, 1], 100)` doit renvoyer `1`.

Partie B — En autonomie

Tu écris maintenant des programmes *entiers*, sans squelette. Pour chaque exercice : lis la spécification (entrées, traitement, sorties, contraintes), réponds d'abord aux questions de conception de l'encadré orange (sur papier ou à voix haute), puis écris ton programme dans un fichier et teste-le avec les valeurs de l'encadré vert. L'objectif est de concevoir toi-même la solution : ne demande pas de lignes de code toutes faites.

Exercice B1 — La machine à café (récursivité et formule de récurrence)

Un distributeur de boissons installé dans un collège ne peut rendre que des pièces de **10, 6 et 1** centimes. Un élève paie une pièce de 20 centimes un gobelet à 8 centimes : la machine doit rendre 12 centimes. Pour rendre une somme s, on choisit une première pièce p ($p \leq s$), puis on rend $s - p$: le nombre total de pièces vaut $1 + n(s-p)$. Le nombre minimal s'obtient en prenant le *minimum* sur toutes les pièces possibles.

Énoncé. Écrire une fonction récursive `rendu_rec(P, s)` qui renvoie le nombre minimal de pièces de P dont la somme est s, en traduisant directement la formule de récurrence : $n(0) = 0$ et, pour $s > 0$, $n(s) = 1 + \min(n(s-p))$ pour $p \in P$, $p \leq s$.

Spécification.

— *Entrées* : une liste P d'entiers (contenant 1), un entier $s \geq 0$.

— *Traitement* : cas de base $s = 0$; sinon $1 + \min($`rendu_rec`$(P, s-p)$ pour chaque pièce $p \leq s)$.

— *Sorties* : un entier : le nombre minimal de pièces dont la somme est s.

— *Contraintes* : récursivité obligatoire ; on suppose $1 \in P$; ni boucle, ni mémoïsation.

Pour t'aider — questions de conception (sans donner le code)

- Pourquoi la fonction est-elle récursive ? Quel est le cas de base, et pourquoi $s = 0$ donne-t-il 0 ?
- Pourquoi faut-il supposer que $1 \in P$? (que se passerait-il pour une somme qui ne peut pas être rendue ?)
- Pourquoi le `min` est-il calculé sur les pièces $p \leq s$ uniquement ?
- La fonction est-elle efficace pour $s = 50$? Pourquoi ?

Point de validation

Vérifie ton programme : `rendu_rec([10, 6, 1], 12)` doit renvoyer `2` ; `rendu_rec([10, 6, 1], 22)` doit renvoyer `3` ($10 + 6 + 6$) ; `rendu_rec([10, 6, 1], 0)` doit renvoyer `0`.

Exercice B2 — La caisse du petit commerce (mémoïsation avec un dictionnaire)

Un petit commerce rend la monnaie avec des pièces de **6, 4 et 1** centimes — un système volontairement non canonique : pour 8 centimes, le glouton rend 3 pièces alors que 2 suffisent ($4 + 4$). La version récursive naïve de l'exercice B1 recalcule sans cesse les mêmes valeurs. On mémoïse les résultats dans un **dictionnaire** qui associe chaque somme k au nombre minimal $n(k)$.

Énoncé. Écrire une fonction récursive `rendu_memo(liste_pieces, somme, dico_memoisation)` qui renvoie le nombre minimal de pièces dont la somme est `somme`. Le dictionnaire, initialisé à `{0: 0}`, est rempli au fil des appels : si `somme` est déjà une clé, on renvoie la valeur mémorisée ; sinon on calcule $1 + \min($`rendu_memo`$(\ldots, somme - p, \ldots))$, on mémorise, puis on renvoie.

Spécification.

- *Entrées* : une liste d'entiers (contenant 1), un entier `somme` ≥ 0, un dictionnaire initialisé à `{0: 0}`.
- *Traitement* : cas de base : `somme` $\in$ `dico_memoisation` ; sinon calcul, mémorisation, renvoi.
- *Sorties* : un entier : le nombre minimal de pièces dont la somme est `somme`.
- *Contraintes* : dictionnaire de mémoïsation obligatoire ; récursivité ; pas de boucle.

Pour t'aider — questions de conception (sans donner le code)

- Pourquoi un dictionnaire plutôt qu'une liste ici ? (que faudrait-il connaître à l'avance pour utiliser une liste ?)
- Que se passerait-il si on oubliait le test « `somme` $\in$ `dico_memoisation` » ?
- Compare le nombre d'appels de `rendu_memo` avec celui de `rendu_rec` pour une même somme : que gagne-t-on ?
- Quelles sommes le dictionnaire contient-il à la fin du calcul de `rendu_memo([6, 4, 1], 8, {0: 0})` ? Pourquoi ?

Point de validation

Vérifie ton programme : en appelant `rendu_memo([6, 4, 1], k, {0: 0})` pour k allant de 0 à 10, la liste des résultats doit être `[0, 1, 2, 3, 1, 2, 1, 2, 2, 3, 2]`.

Exercice B3 — L'escalier de l'immeuble (programmation dynamique itérative)

L'ascenseur d'un immeuble est en panne : un coursier doit monter un escalier de n marches, en montant à chaque pas *soit 1 marche, soit 2 marches*. Combien de façons différentes a-t-il de monter ? On note

$E(n)$ ce nombre. Pour le dernier pas, il vient soit de la marche $n-1$ (1 marche), soit de la marche $n-2$ (2 marches) : $E(n) = E(n-1) + E(n-2)$, avec $E(0) = 1$ et $E(1) = 1$.

Énoncé. Écrire une fonction itérative `escaliers(n)` qui renvoie $E(n)$, en remplissant une liste E de taille $n+1$ *de bas en haut* (programmation dynamique).

Spécification.

— *Entrées* : un entier $n \geq 0$.

— *Traitement* : $E[0] = 1$; $E[1] = 1$; pour i de 2 à n : $E[i] = E[i-1] + E[i-2]$.

— *Sorties* : l'entier $E(n)$.

— *Contraintes* : version itérative obligatoire (pas de récursivité) ; la liste est remplie de bas en haut, chaque valeur ne dépend que des deux précédentes.

Pour t'aider — questions de conception (sans donner le code)

— Pourquoi $E(n) = E(n-1) + E(n-2)$? (à quoi correspond le dernier pas ?)

— Quelle est la taille de la liste E en mémoire ? Deux variables suffiraient-elles ?

— Cette suite est la suite de Fibonacci : laquelle exactement ?

— Comment vérifier le résultat à la main pour $n = 4$?

Point de validation

Vérifie ton programme : `escaliers(4)` doit renvoyer `5` ; `escaliers(10)` doit renvoyer `89` ; `escaliers(20)` doit renvoyer `10 946`.

Exercice B4 — Quel système de pièces choisir ? (moyenne du nombre de pièces)

Une association veut choisir son système de pièces pour minimiser le nombre moyen de pièces manipulées dans les transactions. On compare deux systèmes exprimés en **centimes** : le système « euro » $P_e = [100, 50, 20, 10, 5, 2, 1]$ et un système alternatif $P_a = [100, 40, 25, 10, 5, 2, 1]$ (on remplace la pièce de 20 par une pièce de 40 et on ajoute une pièce de 25). L'algorithme glouton ne suffit pas pour cette comparaison : il faut le *nombre minimal garanti*, donné par la programmation dynamique.

Énoncé. Écrire un programme complet qui : (1) définit la fonction `rendu_dyn(P, s)` (version itérative de l'exercice A4) ; (2) définit une fonction `nb_moyen(P, s_max)` qui renvoie la moyenne des nombres minimaux de pièces pour toutes les sommes de 1 à `s_max` ; (3) affiche la moyenne pour les deux systèmes et indique lequel est le plus économe.

Spécification.

— *Entrées* : les deux listes de pièces définies dans le programme (en centimes).

— *Traitement* : pour chaque système, pour chaque somme s de 1 à 100 : calculer `rendu_dyn(P, s)` et totaliser ; diviser le total par 100.

— *Sorties* : la moyenne pour P_e, la moyenne pour P_a, et le nom du système le plus économe.

— *Contraintes* : réutiliser la fonction `rendu_dyn` (programmation dynamique) ; on suppose $1 \in P$; les pièces sont exprimées en centimes.

Pour t'aider — questions de conception (sans donner le code)

— Pourquoi l'algorithme glouton ne suffit-il pas pour comparer les deux systèmes ?

— À quoi sert `nb_moyen` ? Pourquoi tester toutes les sommes de 1 à 100 et pas seulement quelques-unes ?

— Quel est le coût total du programme, en fonction de `s_max` et de $|P|$? (rappelle le coût de

`rendu_dyn`)

— Comment interpréter la moyenne ? (nombre moyen de pièces par transaction pour un montant entre 1 et 100 centimes)

— Le système alternatif est-il vraiment plus économe ? Le résultat te surprend-il ?

Point de validation

Vérifie ton programme : `nb_moyen(Pe, 100)` doit valoir `3.41` et `nb_moyen(Pa, 100)` doit valoir `3.31` : sur les 100 sommes de 1 à 100, le système alternatif totalise 331 pièces contre 341 pour l'euro.

Partie C — Exercice de réflexion (à finir à la maison)

À finir à la maison

La reconstruction : la liste exacte des pièces à rendre.

Contexte. Un distributeur de billets ne peut pas se contenter du *nombre* minimal de pièces : il doit afficher la *liste exacte* des pièces à rendre. On reprend le système non canonique $P = [10, 6, 4, 1]$. À partir de la liste L remplie par la version itérative ($L[k]$ = nombre minimal de pièces pour la somme k), on peut reconstruire une solution optimale : on part de la somme s ; tant qu'il reste de l'argent, on cherche une pièce p telle que $L[s-p] = L[s] - 1$ (cette pièce fait bien partie d'une solution optimale), on la retient, et on continue avec $s - p$.

Objectif. Écrire deux fonctions :

1. `rendu_dyn_pour_reconstruction(P, s)` qui renvoie la liste L *complète* (modifier une seule ligne de `rendu_dyn`) ;
2. `rendu_reconstruction(P, s)` qui renvoie une liste de pièces de P, minimale, dont la somme est s, en appliquant l'algorithme de reconstruction ci-dessus (une boucle `while` et une boucle `for`).

Questions à se poser avant de coder.

— Pourquoi la condition $L[s-p] = L[s] - 1$ garantit-elle que la pièce p est un bon choix ?

— Pourquoi l'algorithme se termine-t-il ? (que vaut $L[0]$?)

— Comment vérifier qu'une liste de pièces est valide et optimale sans la comparer à toutes les autres ?

— L'ordre des pièces dans P change-t-il la liste obtenue ? Reste-t-elle minimale ?

Étapes suggérées.

1. Vérifier que `rendu_dyn_pour_reconstruction([6, 4, 1], 8)` renvoie `[0, 1, 2, 3, 1, 2, 1, 2, 2]`.
2. Écrire `rendu_reconstruction` et vérifier que `rendu_reconstruction([6, 4, 1], 8)` renvoie `[4, 4]`.
3. Tester avec $P = [10, 6, 4, 1]$ et $s = 12$, puis $s = 22$: vérifier que la somme des pièces vaut bien s et que le nombre de pièces est minimal (comparer avec `rendu_dyn`).
4. Écrire une petite fonction de vérification automatique : une liste de pièces est valide si sa somme vaut s, et optimale si sa longueur vaut `rendu_dyn(P, s)`. L'utiliser sur plusieurs sommes.
5. Bonus : adapter le programme pour afficher les pièces et billets à rendre pour un montant réel en euros (système complet en centimes, ex. 29356).

Contraintes. On suppose $1 \in P$; les fonctions doivent être testées ; les messages peuvent être écrits sans accents (ex. `pieces`, `reste`) si ton éditeur pose problème.

Fin du TP — la partie C est à finir à la maison.