

III Programmation dynamique

6 octobre 2026

1 Principes de la programmation dynamique

1.1 Diviser pour régner (rappels)

Les stratégies du type « diviser pour régner » sont mises en œuvre lorsque l'on arrive à

- ▷ décomposer un problème en plusieurs sous-problèmes ;
- ▷ résoudre plus simplement chaque sous-problème ;
- ▷ résoudre notre problème initial à partir de ces sous-problèmes résolus.

Exemple 1.1.1 (tri rapide).

L'algorithme de tri rapide d'une liste de flottants fait partie de cette famille de stratégies. On en rappelle le principe, via ce pseudo-code (pour une version non en-place, et en choisissant comme pivot le premier élément).

```
Fonction TRI_RAPIDE(L) :  
  Si L a au plus 1 élément Alors  
    | Renvoyer L  
  Sinon  
    | pivot ← L[0]  
    | L1 ← [x ∈ L[1 :] tq. x ≤ pivot]  
    | L2 ← [x ∈ L[1 :] tq. x > pivot]  
    | L1 ← TRI_RAPIDE(L1)  
    | L2 ← TRI_RAPIDE(L2)  
    | Renvoyer Concaténation de L1, [pivot], L2  
Fin Fonction
```

Ici, on choisit un pivot (par exemple le premier élément de L), et l'on décompose le problème du tri de L en celui du tri des deux listes L1 et L2 constituées :

- ▷ des éléments de L (en dehors du premier) inférieurs strictement au pivot, pour L1 ;
- ▷ des autres éléments de L (en dehors du premier), pour L2.

Il est clair que si l'on sait trier L1 et L2, on sait trier L. Il est aussi clair que l'on sait trier L si L est vide ou constituée d'un élément.

1.2 Notre exemple : le rendu de monnaie

Dans ce chapitre, nous allons prendre comme exemple le problème le problème du rendu de monnaie. On se donne une liste d'entiers strictement positifs **pieces** = $[p_0, \dots, p_{n-1}]$, qui représente les différentes valeurs de pièces d'un système monétaire.

Exemple 1.2.1.

Supposons qu'il existe des pièces de 1 €, 3 € et 4 €. On cherche à décomposer la quantité $k = 39$ €. On peut procéder de la manière suivante :

$$39 \text{ €} = 39 \times 1 \text{ €} = 9 \times 4 \text{ €} + 3 \text{ €}.$$

On pourra supposer que les valeurs des pièces sont triées par ordre croissant et que la valeur minimale vaut 1, soit

$$1 = p_0 < p_1 < \dots < p_{n-1}.$$

Étant donné un nombre entier k , on cherche à déterminer le nombre minimal de pièces permettant de décomposer (ou rendre) la valeur k . Plus formellement, on cherche à minimiser la quantité

$$\sum_{i=0}^{n-1} a_i,$$

où $a_0, \dots, a_{n-1}$ sont des entiers naturels vérifiant

$$k = \sum_{i=0}^{n-1} a_i p_i.$$

On note alors $\mathcal{M}(k)$ ce nombre minimal.

1.3 Rappels : algorithme glouton

Une solution gloutonne à ce problème serait de rendre le maximum de pièces de plus haute valeur, et ainsi de suite. On peut donc l'écrire ainsi, en utilisant le quotient et le reste de la division euclidienne de la quantité à rendre par la pièce de plus haute valeur.

```

Fonction monnaieGlouton(k, pieces) :
    aRendre ← k
    nbPieces ← 0
    Pour chaque piece De pieces[ : -1], Faire
        nbPieces ← nbPieces + (quotient de aRendre ÷ piece)
        aRendre ← reste de aRendre ÷ piece
    Renvoyer nbPieces
Fin Fonction

```

Exercice 1.3.1.

Traduire cet algorithme en Python. Le tester sur quelques exemples.

Malheureusement, cet algorithme ne donne pas une solution exacte à notre problème. Ici, avec des valeurs de pièces de 1, 3 et 4 et $k = 6$, il donne une décomposition en trois pièces $6 = 4 + 1 + 1$ alors que la décomposition $6 = 3 + 3$ est optimale.

1.4 Relation de récurrence et solution naïve

On remarque que, pour rendre la monnaie, il convient de rendre une première pièce, puis de rendre la somme restante. Si l'on choisit de rendre la pièce p , il reste alors la valeur $k - p$ à rendre, en supposant bien entendu que cette quantité est positive. Le nombre optimal de pièces rendu pour cette quantité est alors $\mathcal{M}(k - p)$, et il ne faut pas oublier de compter la pièce que l'on vient d'enlever.

On aboutit donc à la relation suivante.

$$\mathcal{M}(k) = 1 + \min \{ \mathcal{M}(k - p) \mid p \in \text{pieces et } p \leq k \}.$$

Le cas d'initialisation de la récurrence est ici assez simple : on rend la valeur 0 avec 0 pièces, donc $\mathcal{M}(0) = 0$. On pourrait aussi considérer que $\mathcal{M}(1) = 1$.

On peut donc écrire un algorithme naïf réalisant cela.

```

Fonction M(k, pieces) :
    Si k=0 Alors
        Renvoyer 0
    Sinon
        L ← [M(k-p, pieces) pour p ∈ pieces tq. p ≤ k]
        Renvoyer 1 + min(L)
Fin Fonction

```

Exercice 1.4.1.

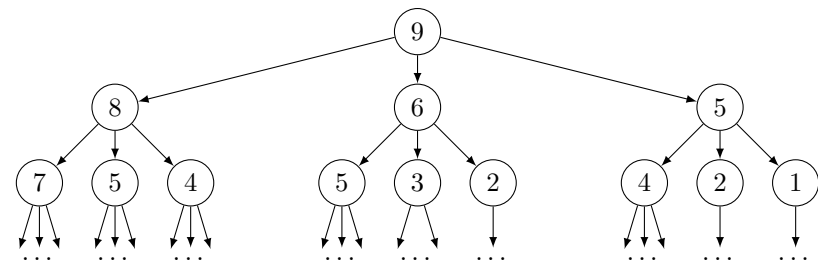
Transformer ce pseudo-code en code Python et vérifier que $\mathcal{M}(6, [1, 3, 4]) = 2$.

Essayer ensuite de faire tourner cela avec $k = 50$.

Cet algorithme a une complexité temporelle catastrophique. Considérons les premiers appels récursifs dans le cas $k = 9$.

- ▷ On commence par calculer $\mathcal{M}(9)$. Cela appelle les calculs de $\mathcal{M}(8)$, de $\mathcal{M}(6)$ et de $\mathcal{M}(5)$.
- ▷ Pour le calcul de $\mathcal{M}(8)$.
 - Cela appelle les calculs de $\mathcal{M}(7)$, de $\mathcal{M}(5)$ et de $\mathcal{M}(4)$.
 - Cela appelle ensuite les calculs de $\mathcal{M}(6)$, de $\mathcal{M}(4)$ et de $\mathcal{M}(3)$.
 - etc.
- ▷ On calcule ensuite $\mathcal{M}(6)$.
 - Cela appelle les calculs de $\mathcal{M}(5)$, de $\mathcal{M}(3)$ et de $\mathcal{M}(2)$.
 - etc.
- ▷ etc.

Bref, on a ici calculé au moins deux fois la même valeur ($\mathcal{M}(5)$), appelé par les calculs de $\mathcal{M}(8)$ et de $\mathcal{M}(6)$. On peut par exemple représenter par un graphe les premiers appels récursifs.



On a recalculé un grand nombre de fois la même information. Par exemple, avec le système de pièces $[1, 3, 4]$, pour calculer $\mathcal{M}(20)$, on appelle 4895 fois le calcul de $\mathcal{M}(1)$, et 3025 fois celui de $\mathcal{M}(2)$! Notamment, la complexité de cet algorithme est exponentielle en n (voir figure 1) !

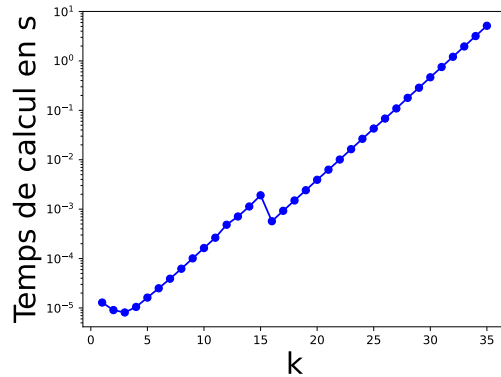


FIGURE 1 – Complexité expérimentale du calcul naïf par récurrence

1.5 La programmation dynamique

Nous nous intéresserons à des problèmes consistant à chercher une meilleure solution parmi un (potentiellement grand) ensemble d'objets informatiques. Typiquement, si l'on définit une fonction réelle c (parfois appelée *fonction de coût* sur un ensemble d'objets informatiques O , on peut vouloir déterminer le minimum de c sur O , ou bien aussi un objet o^* tel que

$$c(o^*) = \min \{ c(o) \mid o \in O \}.$$

Remarque 1.5.1.

Un tel objet o^* n'est pas nécessairement unique.

Exemple 1.5.2.

Le problème du rendu de monnaie rentre dans notre cadre.

Définition 1.5.3 (sous-structure).

Une *sous-structure* (ou *sous-problème*) est une restriction de notre problème à un ensemble plus petit.

Exemple 1.5.4.

Pour calculer $\mathcal{M}(n)$, un sous problème est de calculer $\mathcal{M}(k)$ pour $0 \leq k < n$.

Définition 1.5.5 (sous-structure optimale).

Un problème vérifie la propriété de *sous-structure optimale* si toute solution optimale de tout sous-problème est une partie d'une solution optimale de notre problème de départ.

Exemple 1.5.6 (sous-structure optimale pour le rendu de monnaie).

Considérons une solution optimale pour décomposer la quantité k . Supposons que, suivant cette décomposition, on ait rendu un certain nombre de pièces, totalisant une quantité ℓ . On pourra alors noter $\mathcal{M}'(\ell)$ le nombre de pièces ainsi utilisées.

Il reste alors à décomposer la quantité $\ell' = k - \ell$, ce qui est un sous-problème de notre problème. Notons $\mathcal{M}'(\ell')$ le nombre de pièces utilisées pour décomposer ℓ' : on a

$$\mathcal{M}(k) = \mathcal{M}'(\ell) + \mathcal{M}'(\ell').$$

Supposons que l'on puisse trouver une solution strictement meilleure à celle qui a été effectuée pour rendre cette quantité ℓ' , ce qui s'écrit :

$$\mathcal{M}(\ell') < \mathcal{M}'(\ell').$$

Alors, en décomposant de la même manière la quantité ℓ , on arriverait à décomposer k en strictement moins de $\mathcal{M}(k)$ pièces, ce qui est absurde.

On a donc $\mathcal{M}(\ell') = \mathcal{M}'(\ell')$, et de même $\mathcal{M}(\ell) = \mathcal{M}'(\ell)$, ce qui donne ici :

$$\mathcal{M}(k) = \mathcal{M}(\ell) + \mathcal{M}(k - \ell).$$

Le problème du rendu de monnaie vérifie donc bien la propriété de sous-structure optimale.

Définition 1.5.7 (chevauchement de sous-problèmes).

Des sous-problèmes se *chevauchent* lorsque la résolution du problème appliqué à des sous-structures différentes amène à résoudre ce problème sur des mêmes sous-structures.

Cette situation amène donc à résoudre plusieurs fois ce problème sur une même sous-structure.

Exemple 1.5.8 (chevauchement de sous-problèmes pour le rendu de monnaie).

Avec le système de pièces $[1, 3, 4]$, si l'on cherche à décomposer $k \geq 4$, on va notamment

- ▷ décomposer $k - 1$, puis $k - 1 - 3 = k - 4$;
- ▷ décomposer $k - 3$, puis $k - 3 - 1 = k - 4$;

▷ décomposer $k - 4$.

Il y a donc bien chevauchement de sous-problèmes.

Définition 1.5.9 (programmation dynamique).

Les méthodes de programmation dynamique sont mises en œuvre lorsque le problème étudié possède une propriété de sous-structure optimale, et lorsque les sous-problèmes utilisés pour le résoudre se chevauchent.

Définition 1.5.10 (équation de Bellman).

Lorsque l'on arrive à décomposer notre problème en plusieurs sous-problèmes plus simples, la relation liant la solution optimale de notre problème à celles des sous-problèmes est appelée *équation de Bellman*.

Exemple 1.5.11 (équation de Bellman pour le rendu de monnaie).

On a montré précédemment que si $n \in \mathbb{N}^*$:

$$\mathcal{M}(n) = 1 + \min \{ \mathcal{M}(n - k) \mid k \in \text{pieces et } k \leq n \}.$$

Remarque 1.5.12.

Bien entendu, il est important d'identifier un cas d'initialisation.

Exemple 1.5.13 (initialisation pour le rendu de monnaie).

Le cas d'initialisation est ici :

$$\mathcal{M}(0) = 0 \text{ et } \mathcal{M}(1) = 1.$$

La programmation dynamique intervient lorsque

- ▷ notre problème possède une propriété de sous-structure optimale ;
- ▷ notre problème se résout avec une stratégie « diviser pour régner » ;
- ▷ les sous-problèmes se chevauchent, *i.e.* qu'une résolution récursive naïve fait calculer plusieurs fois les mêmes sous-problèmes, et conduit donc à une complexité temporelle dégradée.

Pour mettre en œuvre une méthode de programmation dynamique, il est fondamental de suivre les étapes suivantes

1. de définir ce qu'est un sous-problème, voire de comprendre que le problème possède une propriété de sous-structure optimale (pas forcément de le formaliser) ;
2. de définir une stratégie récursive en
 - ▷ définissant les cas de base (initialisation) ;
 - ▷ comprenant comment relier les sous-problèmes au problème final (équation de Bellman).

On peut alors

1. analyser les performances, par un calcul de complexité ;
2. mettre en place une des deux stratégies de programmation listées ci-après.

1.6 Calcul de haut en bas, avec mémoïsation

On effectue le calcul de haut en bas lorsque l'on procède par appels récursifs. C'est souvent la manière la plus simple d'écrire l'algorithme, mais une implémentation naïve peut donner une complexité temporelle catastrophique, comme nous l'avons vu dans notre cas sur notre exemple.

Il est alors primordial de prévoir une structure de données dans laquelle on sauvegarde (on dit aussi : *mémoire*) toutes les solutions de sous-problèmes déjà rencontrés. Lorsque l'on veut obtenir un résultat pour un sous-problème :

1. on vérifie d'abord si on l'a déjà calculé, et si c'est le cas on n'a rien à faire ;
2. sinon, on lance un calcul récursif.

Pour faire cela, il faut souvent :

- ▷ prévoir une structure de données *ad-hoc* pour mémoriser les résultats des sous-problèmes calculés ;
- ▷ s'organiser pour ne pas recopier les données du problèmes, sans quoi la complexité spatiale peut exploser.

On pourra souvent utiliser une fonction auxiliaire pour le deuxième point.

Remarque 1.6.1.

On utilisera un dictionnaire comme structure universelle de mémoïsation. On pourrait à chaque fois utiliser une structure adaptée au problème, ici une liste par exemple.

Exemple 1.6.2 (calcul de haut en bas avec mémoïsation du rendu de monnaie).

Cela se fait assez simplement. La première étape consiste à créer un dictionnaire (presque) vide, contenant uniquement les cas de base. Il suffit ensuite d'appliquer la formule de récurrence, en utilisant le principe de mémoïsation.

```

Fonction monnaieHautEnBas(k, pieces) :
    memo ← dictionnaire {0:0}
    Fonction M(i) :
        ## M(i) renvoie  $\mathcal{M}(i)$ 
        Si  $i \notin \text{memo}$  Alors
            memo[i] ← 1 + min { M(i - p) | p ∈ pieces tq.  $p \leq i$  }
        Renvoyer memo[i]
    Fin Fonction
    Renvoyer M(k)
Fin Fonction

```

On peut en étudier sommairement la complexité, en négligeant le coût d'un appel récursif et en supposant que toutes les opérations arithmétiques (addition, minimum de deux flottants) se font en temps $\mathcal{O}(1)$:

- ▷ la création de `memo` se fait en $\mathcal{O}(1)$;
- ▷ il y a un calcul à réaliser par entier entre 2 et k ;
- ▷ pour chaque calcul, il y a au plus $\mathcal{O}(n)$ opérations à faire (calcul du minimum).

Ainsi, la complexité temporelle de l'exécution de `monnaie_hautenbas(k, pieces)` est en $\mathcal{O}(kn)$, quand avec un système de n pièces. On remarquera que la complexité spatiale sera aussi en $\mathcal{O}(kn)$ (cela demande une étude approfondie de la pile d'appels).

Exercice 1.6.3.

Traduire ceci en code `Python`. Calculer $\mathcal{M}(1000)$ sur notre exemple.

1.7 Calcul de bas en haut

On effectue un calcul de bas en haut lorsque l'on utilise un style de programmation *impératif*, en partant des cas de base et en construisant petit à petit les solutions des sous-problèmes de plus en plus grand, jusqu'à arriver au problème que l'on souhaite résoudre.

La mise en œuvre d'un calcul de bas en haut n'est pas toujours aisée, et demande une compréhension fine du problème. Pour un problème unidimensionnel, comme ici, cela sera assez simple.

Exercice 1.7.1.

Avec `pieces` = [1, 3, 4], calculer $\mathcal{M}(10)$ en remplissant la grille suivante.

i	0	1	2	3	4	5	6	7	8	9	10
$\mathcal{M}(i)$	0										

Exemple 1.7.2 (calcul de bas en haut du rendu de monnaie).

```

Fonction monnaieBasEnHaut(k, pieces) :
    M ← [0]
    Pour i variant de 1 à k, Faire
        L ← min { M[i - p] | p ∈ pieces tq.  $p \leq i$  }
        ajout de 1 + min(L) à M
    Renvoyer M[k]
Fin Fonction

```

Exercice 1.7.3.

Traduire ceci en code `Python`. Vérifier le résultat sur $\mathcal{M}(10)$ et $\mathcal{M}(1000)$.

1.8 Complexité expérimentale

On trouvera dans la figure 2 deux courbes expérimentales de temps de calcul pour les fonctions données dans les deux parties précédentes.

On remarquera au moins trois choses.

- ▷ Le temps de calcul est spectaculairement moindre que par la méthode naïve : on calcule la décomposition de nombres de l'ordre d'un million en environ une seconde.
- ▷ La pente des courbes (si l'on ajustait une droite) est d'environ 1, en échelle logarithmique, ce qui conforte la complexité linéaire (par rapport à k) annoncée précédemment.
- ▷ La méthode de bas en haut est plus efficace que la méthode de haut en bas. C'était attendu : les solutions impératives sont souvent plus efficaces que les méthodes récursives, au moins en `Python`. De plus, on a utilisé un dictionnaire pour la méthode de haut en bas, et une liste dans la méthode de bas en haut. Cette dernière solution est probablement un peu plus efficace que la première.

1.9 Rendu d'une solution optimale

Dans les exemples précédents, nous avons calculé le nombre minimal de pièces nécessaires pour décomposer un entier k , mais nous n'avons pas obtenu la suite de pièces permettant de réaliser cette décomposition. De manière générale, il n'est jamais très compliqué d'obtenir un objet réalisant le minimum de la fonction coût, une fois que l'on sait calculer ce minimum. Il suffit souvent d'adapter la structure de données utilisée afin de garder en mémoire l'action effectuée.

On peut aussi parfois recalculer la solution optimale en fonction de la structure de données calculée, lors du calcul de haut en bas avec memoisation. C'est cette seconde manière de faire qui semble être privilégiée par le programme, nous la mettrons en place dès que possible.

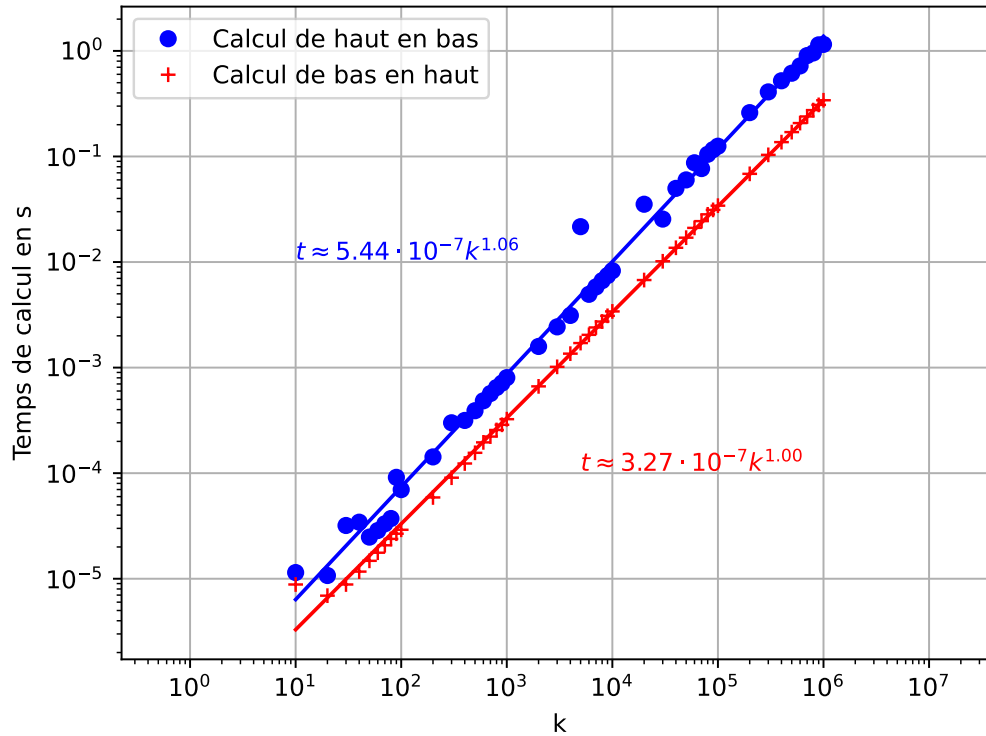


FIGURE 2 – Complexité expérimentale du calcul du rendu de monnaie par programmation dynamique.

Ici, pour le rendu de monnaie, il y a au moins deux manières de faire :

- ▷ de haut en bas : à partir de la liste des $\mathcal{M}(i)$ calculée (ici, **memo**), reconstruire la suite de décompositions en la remontant petit à petit ;
- ▷ de bas en haut : sauvegarder dans chaque « case » la suite de pièces rendues aboutissant à cette case, ou au moins la dernière pièce rendue, au fur et à mesure de l'avancée de l'algorithme.

Remarque 1.9.1 (Non unicité).

Il peut y avoir plusieurs solutions optimales. Par exemple, avec des pièces de valeur 1, 3 et 4, on décompose optimalement $9 = 3 + 3 + 3 = 4 + 4 + 1$.

Remarque 1.9.2 (Rendu de monnaie).

La dernière pièce p rendue permet de minimiser la quantité $\mathcal{M}(k-p')$, pour $p' \in \text{pieces}$

vérifiant $p' \leq k$. On a alors la relation

$$1 + \mathcal{M}(k - p) = \mathcal{M}(k),$$

qui permet de détecter une pièce p optimale à rendre.

Exercice 1.9.3.

On donne la liste des $\mathcal{M}(i)$, pour le système de pièces $[1, 3, 4]$.

i	0	1	2	3	4	5	6	7	8	9	10
$\mathcal{M}(i)$	0	1	2	1	1	2	2	2	2	3	3

Déterminer une manière optimale de décomposer 10 sur ce système.

Exemple 1.9.4 (Reconstruction du chemin de poids minimal).

Cela se fait assez simplement, à partir du dictionnaire **memo**. En effet, il suffit de remonter les décompositions « à l'envers », en respectant la règle vue dans la remarque 1.9.2.

```

Fonction reconstruction(k, memo, pieces) :
    Si k=0 Alors
        Renvoyer []
    Sinon
        Pour chaque p De pieces, Faire
            Si p ≤ k et memo[k-p]+1 = memo[p] Alors
                L ← reconstruction(k-p, memo, pieces)
                ajouter p à L
            Renvoyer L
    Fin Fonction
    
```

On peut aussi procéder de bas en haut, en calculant pas à pas la décomposition optimale pour chaque sous-problème rencontré.

Exercice 1.9.5.

Avec **pieces** = $[1, 3, 4]$, calculer $\mathcal{M}(10)$ et une décomposition optimale de 10 en remplissant la grille suivante.

i	0	1	2	3	4	5	6	7	8	9	10
$\mathcal{M}(i)$	0										
Décomposition optimale	[]										

Exemple 1.9.6 (calcul de bas en haut du rendu de monnaie).

On reprend le principe de la fonction précédente, en remplaçant chaque valeur de la

liste des $\mathcal{M}(i)$ par un couple $(\mathcal{M}(i), I)$, où I est une liste optimale de pièces pour décomposer la valeur i .

On commence toutefois par écrire une fonction qui calcule la pièce optimale à rendre, et la décomposition associée. Ici, M est une liste dont les éléments sont de la forme $(\mathcal{M}(i), I)$ décrite précédemment.

```

Fonction meilleurePiece(i, M, pieces) :
    L ← [M[i-p][0] pour p ∈ pieces tq. p ≤ i]
    mini ← min(L)
    Pour chaque p De pieces, Faire
        Si p ≤ i et M[i-p][0] = mini Alors
            dec ← copie de M[i-p][1]
            ajouter p à dec
            Renvoyer (mini+1, dec)
Fin Fonction
    
```

```

Fonction monnaieBasEnHaut2(k, pieces) :
    M ← [(0, [])]
    Pour i variant de 1 à k, Faire
        ajouter meilleurePiece(i, M, pieces) à M
    Renvoyer M[k]
Fin Fonction
    
```

Exercice 1.9.7.

Mettre en œuvre un des algorithmes précédents. Vérifier qu'une manière optimale de décomposer 10 avec $[1, 3, 4]$ est $10 = 4 + 3 + 3$.

2 Exemples du programme

Voici les exemples cités par le programme. Même s'il est précisé que la liste n'est ni impérative, ni limitative, il peut être intéressant de réviser cette partie avant les concours.

2.1 Partition équilibrée d'un tableau d'entiers positifs.

On se donne une liste d'entiers $p = [p_1, \dots, p_n]$, et l'on cherche à déterminer une partition I, J de $\llbracket 1, n \rrbracket$ (*i.e.* deux parties I, J vérifiant $I \cap J = \emptyset$ et $I \cup J = \llbracket 1, n \rrbracket$)

minimisant la quantité

$$\left| \sum_{i \in I} p_i - \sum_{j \in J} p_j \right|.$$

Exemple 2.1.1 (situation presque réelle).

Les valeurs $p_1, \dots, p_n$ peuvent correspondre aux poids des passagers d'un avion, et l'on cherche à répartir les passagers à gauche et à droite de l'avion de la manière la plus équilibrée possible.

On peut remarquer que pour une partition I, J de $\llbracket 1, n \rrbracket$, alors

$$\sum_{i \in I} p_i + \sum_{j \in J} p_j = \sum_{k=1}^n p_k.$$

Notons

$$S = \sum_{k=1}^n p_k.$$

Pour une partition I, J optimale, on peut bien entendu échanger I et J . Il suffit donc de déterminer une partie $I \subset \llbracket 1, n \rrbracket$ vérifiant

$$\frac{S}{2} \leq \sum_{i \in I} p_i$$

et minimisant la somme $\sum_{i \in I} p_i$.

Pour cela, on va calculer efficacement les sommes de toutes les parties de $\llbracket 1, n \rrbracket$: il y en a au plus autant que d'éléments de $\llbracket 0, S \rrbracket$. Pour $0 \leq i \leq n$, notons

$$S_i = \left\{ \sum_{j \in I} p_j \mid I \subset \{1, \dots, i\} \right\},$$

on cherche à calculer S_n . Or,

$$S_{i+1} = S_i \cup \{s + p_{i+1} \mid s \in S_i\}.$$

On peut poser $S_0 = \{0\}$. On peut donc calculer $S_0, \dots, S_n$ de proche en proche, par programmation dynamique (on a en effet une relation de récurrence, un cas d'initialisation, et un recouvrement de sous-problèmes : plusieurs sous-parties de $\llbracket 1, n \rrbracket$ peuvent avoir la même somme).

Exemple 2.1.2 (calcul de bas en haut de la somme d'un sous-tableau équilibré). On peut écrire l'algorithme suivant.

```

Fonction sstab_equilibre(T) :
  n ← nombre d'éléments de T
  S ←  $\sum_{i=0}^{n-1} t_i$ 
  L ← tableau vide de taille S+1
  L[0] ← vrai
  Pour chaque x De T, Faire
    Pour chaque s De L, Faire
      Si L[s] Alors
        L[s+x] ← vrai
  Renvoyer plus petit i tel que  $2i \geq S$  et L[i]
Fin Fonction

```

Exercice 2.1.3.

Déterminer la complexité temporelle de cet algorithme, en fonction de S et de n .

Exercice 2.1.4.

Transformer l'algorithme précédent en code **Python**. On fera attention à parcourir la liste L à partir de la fin, ou bien à recopier L .

Adapter cela pour renvoyer un sous-tableau équilibré.

2.2 Ordonnancement de tâches pondérées

On considère un certain nombre de tâches $T_0, \dots, T_{n-1}$. Chaque tâche T_i est décrite par une date de début d_i , une date de fin f_i et un poids p_i . Ces trois objets sont supposés être des flottants, et l'on suppose de plus que $d_i < f_i$.

L'objectif est de déterminer la somme maximale des poids des tâches que l'on peut enchaîner sans qu'elles ne se superposent (on parle alors de tâches *compatibles*).

On va supposer que les tâches sont triées par date de fin croissante (cela tombe bien, vous savez écrire un algorithme de tri !) : $f_0 \leq \dots \leq f_{n-1}$.

Si l'on note S_i la somme maximale des poids des tâches compatibles parmi $T_0, \dots, T_i$, et si l'on note j l'indice maximal des tâches se terminant avant que T_{i+1} ne commence (*i.e.* j est le plus grand entier vérifiant $f_j \leq d_{i+1}$), alors :

▷ soit on peut rajouter la tâche T_{i+1} aux tâches précédentes déjà terminées, et alors $S_{i+1} = S_j + p_{i+1}$ (on convient que si aucune tâche précédente n'est terminée, on a $S_j = 0$),

▷ sinon il n'est pas intéressant de rajouter la tâche T_{i+1} , et alors $S_{i+1} = S_i$.

On a donc une définition récursive de la suite des S_i :

$$S_{i+1} = \max(S_j + p_{i+1}, S_i) \text{ où } j = \max \{ k \in \llbracket 0, i \rrbracket \mid f_k \leq d_{i+1} \}.$$

Le cas d'initialisation est des plus évident : $S_0 = p_0$.

La propriété de sous-structure optimale est ici la suivante : si $T_0, \dots, T_{p-1}$ sont les tâches optimales, et si $0 \leq j < p$, alors les tâches optimales se terminant toutes avant le temps f_j sont $T_0, \dots, T_j$.

Exemple 2.2.1 (calcul de bas en haut d'un ordonnancement optimal de tâches). On peut écrire l'algorithme suivant, en supposant que d , f et p sont les tableaux contenant respectivement les dates de début, de fin, et les poids.

```

Fonction ordonnancement(d,f,p) :
  n ← nombre de tâches
  S ← tableau de longueur n contenant des p[0]
  Pour i variant de 1 à n-1, Faire
    j ← plus grand k < i tel que f[k] ≤ d[i] (-1 s'il n'y en a pas)
    S[i] ← max(S[i-1], S[j] + p[i]) avec la convention S[-1] = 0
  Renvoyer S[n-1]
Fin Fonction

```

Exercice 2.2.2.

Déterminer la complexité temporelle de cet algorithme, en fonction de n .

Exercice 2.2.3.

Adapter le pseudo-code précédent en code **Python**.

Modifier ce code pour renvoyer aussi les tâches optimales.

2.3 Plus longue sous-suite commune

Dans cet exemple, on peut s'intéresser autant à des chaînes qu'à des tableaux. Nous choisirons de travailler sur des chaînes.

Définition 2.3.1.

Une sous-chaîne d'une chaîne $t = t_0 \dots t_{n-1}$ est une chaîne $s = t_{i_0} \dots t_{i_{p-1}}$, où $0 \leq i_0 < \dots < i_{p-1} \leq n$.

Moins formellement, on obtient une sous-chaîne de t en enlevant de t un certains nombres de caractères.

Exemple 2.3.2.

Les chaînes *abra* et *aaaaa* sont des sous-chaînes de la chaîne *abracadabra*, mais *barbar* ne l'est pas.

Nous allons étudier le problème suivant : étant données deux chaînes $s = s_0 \dots s_{n-1}$ et $t = t_0 \dots t_p$, déterminer une plus longue sous-chaîne commune à s et t (ou au moins sa longueur).

Notons $L(s, t)$ cette longueur optimale. On peut remarquer que :

- ▷ si $s_{n-1} = t_{p-1}$ (i.e. s et t terminent par le même caractère), alors la plus longue sous-chaîne commune à s et t termine par ce caractère, et $L(s, t) = L(s_0 \dots s_{n-2}, t_0 \dots t_{p-2}) + 1$;
- ▷ sinon, la plus longue sous-chaîne commune à s est t est soit la plus longue sous-chaîne commune à s et $t_0 \dots t_{p-2}$, soit celle commune à $s_0 \dots s_{n-2}$ et t . On a alors

$$L(s, t) = \max(L(s, t_0 \dots t_{p-2}), L(s_0 \dots s_{n-2}, t)).$$

Les cas d'initialisation est ici évident : si s ou t sont vides, on a $L(s, t) = 0$.

On peut donc prévoir une matrice à n lignes et p colonnes, dans lequel on calcule les $L(s_0 \dots s_{i-1}, t_0 \dots t_{j-1})$ pour $0 \leq i \leq n$ et $0 \leq j \leq p$.

Exercice 2.3.3.

Écrire un algorithme (en pseudo-code ou en langage Python) qui calcule la longueur de la plus longue sous-chaîne commune à deux chaînes.

Justifier/Vérifier que la complexité temporelle est en $\mathcal{O}(np)$, où n et p sont les longueurs des chaînes s et t , respectivement.

L'adapter pour calculer une plus longue sous-chaîne.

2.4 Distance d'édition (ou de Levenshtein)

On considère dans cet exemple deux chaînes de caractères $s = s_0 \dots s_{n-1}$ et $t = t_0 \dots t_{p-1}$. On se pose la question de calculer le nombre minimal d'opérations permettant de transformer s en t , en s'autorisant les opérations suivantes :

- ▷ substitution d'un caractère par un autre ;
- ▷ suppression d'un caractère ;
- ▷ insertion d'un caractère.

On notera $L(s, t)$ ce nombre minimal d'opérations, appelé *distance de Levenshtein* entre s et t .

Soit $0 \leq i \leq n$ et $0 \leq j \leq p$, et notons

$$L_{i,j} = L(s_0 \dots s_{i-1}, t_0 \dots t_{j-1}).$$

Supposons que l'on sache calculer $L_{i-1,j-1}$, $L_{i-1,j}$ et $L_{i,j-1}$.

Pour calculer $L_{i,j}$, il y a plusieurs cas particuliers :

- ▷ si $s_i = t_j$, il n'y a rien à faire : $L_{i,j} = L_{i-1,j-1}$.

▷ sinon, il y a trois situations :

- on supprime la dernière lettre s_i dans s : $L_{i,j} = L_{i-1,j} + 1$;
- on insère t_j à la fin de s : $L_{i,j} = L_{i,j-1} + 1$;
- on substitue s_i par t_j : $L_{i,j} = L_{i-1,j-1} + 1$.

Dans ce cas, on a donc

$$L_{i,j} = \min(L_{i-1,j}, L_{i,j-1}, L_{i-1,j-1}) + 1.$$

Le cas d'initialisation est assez évident :

- ▷ si s est vide, on insère t entièrement : $L_{0,j} = j$;
- ▷ si t est vide, on supprime s entièrement : $L_{i,0} = i$.

On peut donc écrire un algorithme résolvant ce problème par programmation dynamique. On utilisera alors une structure de données matricielle.

Remarque 2.4.1 (Mise en œuvre efficace d'une matrice).

Nous avons besoin de manipuler une matrice (ici **memo**) initialement vide, dont les cases seront (éventuellement) remplies au fur et à mesure de l'exécution du programme.

Pour cela, nous utiliserons un dictionnaire, dont les clefs seront les couples de coefficients d'une case. Pour créer une telle matrice vide, on écrira donc la ligne suivante.

```
1 memo = {}
```

Pour remplir la première colonne, on écrira par exemple la chose suivante.

```
2 for i in range(n+1) :
3     memo[(i, 0)] = i
```

Pour tester si **memo**[i,j] est vide, on utilisera alors le test **(i,j) not in memo**.

Remarque 2.4.2 (Calcul de bas en haut).

On remarque que le calcul de $L_{i,j}$ ne fait intervenir que des cases situées en haut et/ou à gauche de la case contenant $L_{i,j}$. On peut donc calculer les $L_{i,j}$ ligne par ligne, de la première à la dernière, et chaque ligne de gauche à droite.

2.5 Algorithme de Floyd-Warshall

Dans cet exercice, on considère un graphe orienté pondéré $G = (S, A, P)$, possédant n sommets numérotés de 1 à n donnés par l'ensemble de sommets $S = \{s_1, \dots, s_n\}$. On suppose le graphe décrit par sa matrice d'adjacence M : pour tout $1 \leq i, j \leq n$,

- ▷ $M_{i,j}$ vaut le poids de l'arête (i, j) si elle existe ;
- ▷ si l'arête (i, j) n'existe pas, alors $M_{i,j} = +\infty$.

On rappelle quelques définitions.

Définition 2.5.1 (chemin, poids et circuit).

Un chemin $c = (c_1, \dots, c_p)$ est une suite finie de sommets du graphe (appartenant donc à S) tels que pour tout $1 \leq i < p$, (c_i, c_{i+1}) est une arête (soit $(c_i, c_{i+1}) \in A$).

L'entier p est alors la *longueur* du chemin c , le sommet c_1 est le *sommet de départ* de c , le sommet c_p est son *sommet d'arrivée*, et les sommets $c_2, \dots, c_{p-1}$ sont les *sommets intermédiaires* du chemin. On dit aussi que le chemin relie c_1 à c_p en passant par les sommets $c_2, \dots, c_{p-1}$.

Si l'on note $p(a)$ le poids d'une arête (ce sont les flottants contenus dans la matrice d'adjacence), on note

$$p(c) = \sum_{i=1}^{p-1} p(c_i, c_{i+1})$$

le poids du chemin c : c'est la somme des poids des arêtes qui le composent.

Enfin, on appelle *circuit* un chemin qui a même sommet de départ et d'arrivée.

Exemple 2.5.2.

Voici un exemple de graphe à 6 sommets $s_1, \dots, s_6$ (voir figure 3). La matrice d'adjacence de ce graphe est donc

$$M = \begin{pmatrix} +\infty & 1 & +\infty & +\infty & +\infty & +\infty \\ +\infty & +\infty & -1 & +\infty & 2 & +\infty \\ +\infty & 9 & +\infty & 2 & +\infty & +\infty \\ +\infty & +\infty & +\infty & +\infty & 2 & +\infty \\ 1 & +\infty & +\infty & +\infty & +\infty & +\infty \\ +\infty & +\infty & 1 & +\infty & 7 & 1 \end{pmatrix}.$$

Il y a par exemple deux chemins pour aller de s_3 à s_2 :

- ▷ (s_3, s_2) , de poids 9 et sans sommet intermédiaire ;
- ▷ $(s_3, s_4, s_5, s_1, s_2)$, de poids 6 et dont les sommets intermédiaires sont s_4, s_5 et s_1 .

On suppose maintenant que le graphe G ne possède aucun circuit de poids négatif. Ainsi, on peut définir pour chaque couple de sommets (s_i, s_j) le poids minimal d'un chemin reliant s_i à s_j .

Nous allons nous intéresser au problème du calcul de tous les chemins de poids minimal reliant toutes les paires de sommets du graphe, ou au moins au poids minimal d'un tel chemin.

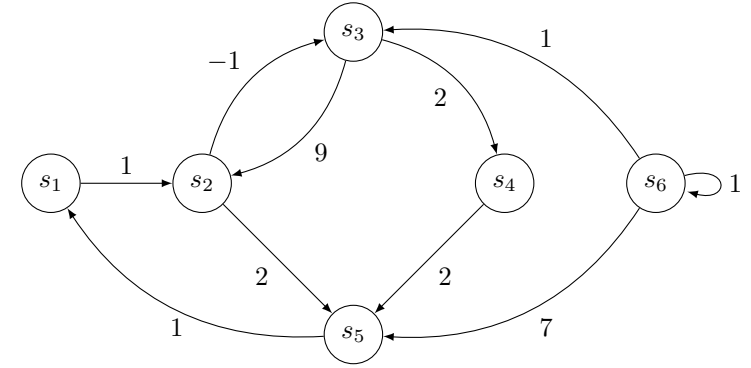


FIGURE 3 – Un exemple de graphe orienté et pondéré

Pour $1 \leq i, j \leq n$ et $0 \leq k \leq n$, on note $M_{i,j}^{(k)}$ le poids minimal d'un chemin reliant i à j et dont tous les sommets intermédiaires sont dans l'ensemble $\{s_1, \dots, s_k\}$ (on définit $M_{i,j}^{(k)} = +\infty$ s'il n'existe aucun chemin reliant i à j dont les sommets intermédiaires sont dans cet ensemble). On pourra noter $M^{(k)}$ la matrice des $M_{i,j}^{(k)}$, et l'on a donc $M^{(0)} = M$.

L'objectif est donc de calculer la matrice $M_{i,j}^{(n)}$. On pourra remarquer que si $1 \leq i, j, k \leq n$:

- ▷ soit le chemin optimal de poids $M_{i,j}^{(k)}$ ne passe pas par le sommet s_k , et alors $M_{i,j}^{(k)} = M_{i,j}^{(k-1)}$,
- ▷ soit le chemin optimal de poids $M_{i,j}^{(k)}$ passe par s_k , et se décompose donc en un chemin allant de s_i à s_k puis en un chemin allant de s_k à s_j . Par définition, ces deux chemins ne passent que par $s_1, \dots, s_{k-1}$, et donc $M_{i,j}^{(k)} = M_{i,k}^{(k-1)} + M_{k,j}^{(k-1)}$

On a donc la relation de récurrence

$$M_{i,j}^{(k)} = \min \left(M_{i,j}^{(k-1)}, M_{i,k}^{(k-1)} + M_{k,j}^{(k-1)} \right).$$

On peut donc exploiter cette relation de récurrence afin de résoudre ce problème par programmation dynamique.

Exemple 2.5.3 (Algorithme de Floyd-Warshall, calcul de bas en haut). Cela se fait aisément

```

Fonction floyd_warshall(G) :
  n ← nombre de sommets de G
  M0 ← matrice d'adjacence de G
  Pour k variant de 1 à n, Faire
    Mk ← matrice vide nxn
    Pour i variant de 1 à n, Faire
      Pour j variant de 1 à n, Faire
        Mi,jk ← min(Mi,jk-1, Mi,kk-1 + Mk,jk-1)
  Renvoyer Mn

```

Exercice 2.5.4.

Calculer la complexité temporelle de cet algorithme, en fonction de n .

Exercice 2.5.5.

Transformer le pseudo-code précédent en code Python. Vérifier que les plus courts poids de chemins entre deux points sont donnés par la matrice suivante :

$$\begin{pmatrix} 4 & 1 & 0 & 2 & 3 & +\infty \\ 3 & 4 & -1 & 1 & 2 & +\infty \\ 5 & 6 & 5 & 2 & 4 & +\infty \\ 3 & 4 & 3 & 5 & 2 & +\infty \\ 1 & 2 & 1 & 3 & 4 & +\infty \\ 6 & 7 & 1 & 3 & 5 & 1 \end{pmatrix}.$$

Pour obtenir le chemin de poids minimal entre deux points, et non plus seulement son poids, il suffit de modifier légèrement l'algorithme précédent, en créant une seconde matrice, de même dimension que M , et conservant pour chaque couple (s_i, s_j) le dernier sommet avant s_j du chemin optimal reliant s_i à s_j :

- ▷ on l'initialise à s_i s'il y a une arête entre s_i et s_j , et c'est vide sinon ;
- ▷ à chaque fois que l'on a $M_{i,j}^{(k-1)} > M_{i,k}^{(k-1)} + M_{k,j}^{(k-1)}$ et que l'on met donc à jour ce chemin, ce dernier sommet est mis à jour comme celui du chemin optimal reliant s_k à s_j .

Il suffit ensuite de reconstruire récursivement le chemin en remontant ces antécédents.

3 Exercices**3.1 Suite de Fibonacci**

On rappelle que la suite de Fibonacci est définie par :

$$F_0 = 0, \quad F_1 = 1, \quad \forall n \geq 2, \quad F_n = F_{n-1} + F_{n-2}.$$

Écrire un algorithme de calcul de cette suite de Fibonacci par programmation dynamique, de haut en bas.

3.2 Ordonnancement de tâches

Lire la partie 2.2. On considère le problème d'ordonnancement des tâches avec le tableau suivant (voire table 1). Par exemple, la tâche T_4 commence au temps 2, termine au temps 5 et a pour poids 1.

Recopier et compléter le tableau de calcul de bas en haut de la table 2, et donner l'ordonnancement optimal de tâches du problème, ainsi que son poids.

Coder en Python l'algorithme de calcul de ce poids optimal, de bas en haut, et retrouver le poids optimal obtenu sur papier.

n° de tâche i	début (d_i)	fin (f_i)	poids (p_i)
0	0	2	3
1	3	4	1
2	2	4	2
3	1	5	6
4	2	5	1
5	6	7	1
6	4	8	2
7	4	9	2
8	7	10	3
9	3	10	1
10	6	10	3

TABLE 1 – Exemple d'ordonnancement de tâches

3.3 Plus longue sous-suite strictement croissante.

On rappelle que si $T = [t_0, \dots, t_{n-1}]$ est un tableau d'entiers de longueur n , alors un sous-tableau de T est un tableau extrait de T , *i.e.* un tableau $T' = [t_{i_0}, \dots, t_{i_{p-1}}]$, où $0 \leq i_0 < \dots < i_{p-1} < n$.

On s'intéresse à la question suivante : étant donné un tel tableau T , quelle est la longueur maximal d'un tel sous-tableau T' qui est strictement croissant ? On aimerait aussi donner un tel sous-tableau T' .

1. Combien y-a-t-il de sous-tableaux T' de T , en fonction de n ? Quelle serait la complexité temporelle d'une recherche exhaustive de la longueur du plus grand sous-tableau croissant de T ?

n° de tâche i	S_i	Tâches optimales
0	3	T_0
1	4	T_0, T_1
2		
3		
4		
5		
6		
7		
8		
9		
10		

TABLE 2 – Exemple d'ordonnancement de tâches

2. Si $0 \leq j < n$, on considère le tableau $T' = [t_{i_0}, \dots, t_{i_{p-2}}, t_j]$ ayant la propriété suivante : « T' est le plus grand sous-tableau de T strictement croissant et dont le dernier élément est t_j ». Que peut-on dire de $T'' = [t_{i_0}, \dots, t_{i_{p-2}}]$?
3. On note ℓ_j la longueur de ce sous-tableau T' . Écrire une relation liant ℓ_j à $\ell_0, \dots, \ell_{j-1}$. On remarquera que cette relation devra faire apparaître les conditions « $t_i < t_j$ » pour $0 \leq i < j$.
4. Que vaut ℓ_0 ?
5. En déduire un pseudo-code permettant de calculer $L = [\ell_0, \dots, \ell_{n-1}]$.
6. À partir de ce tableau L , comment peut-on répondre au problème initial ?
7. Évaluer la complexité temporelle de l'algorithme ainsi construit.