III Programmation dynamique

29 août 2025

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 : distance de Levenshtein

Dans ce chapitre, nous allons prendre comme exemple le problème suivant : on part d'un mot de départ s (ici, `INFO !`), que l'on veut transformer en un mot d'arrivée t (ici, `BINGO`). Les opérations autorisées sont les suivantes :

— suppression d'un caractère (exemple : `INFO !` $\to$ `INFO`) ;
— ajout d'un caractère (exemple : `INFO !` $\to$ `BINFO !`) ;
— modification d'un caractère (exemple : `INFO !` $\to$ `INGO !`) ;

On appelle *distance de Levenshtein* (ou distance d'édition) entre ces deux mots le nombre minimal d'opérations pour transformer s en t, et est notée $\mathscr{L}(s,t)$.

Notons $s_0,\dots,s_{n-1}$ les lettres de s, ou encore $s = s_0\cdots s_{n-1}$, et $t_0,\dots,t_{p-1}$ les lettres de t, ou encore $t = t_0\cdots t_{p-1}$. Par exemple, ici, $n = p = 5$, $s_0 =$ `I`, $s_4 =$ `!`.

Notons $L_{i,j}$ la distance de Levenshtein entre les i premiers caractères de s et les j premiers caractères de t :

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

où $s_0\cdots s_{i-1}$ est le mot vide (noté ε) si $i = 0$ (*idem* pour $j = 0$).

Supposons que l'on sache calculer pour $i \geqslant 1$ et $j \geqslant 1$ les valeurs de $L_{i-1,j-1}$, de $L_{i,j-1}$ et de $L_{i-1,j}$. Il y a deux cas possibles.

— Soit $s_{i-1} = t_{j-1}$: les deux dernières lettres de $s_0\cdots s_{i-1}$ et de $t_0\cdots t_{j-1}$ coïncidant, il suffit de ne pas les modifier, on a donc

$$L_{i,j} = \mathscr{L}(s_0\cdots s_{i-1}, t_0\cdots t_{j-1}) = \mathscr{L}(s_0\cdots s_{i-2}, t_0\cdots t_{j-2}) = L_{i-1,j-1}.$$

— Soit $s_{i-1} \neq t_{j-1}$: les deux dernières lettres de $s_0 \cdots s_{i-1}$ et de $t_0 \cdots t_{j-1}$ différant, il faudra les modifier, ce qui peut se faire de trois manières :
  - soit en supprimant s_{i-1}, on étudie ensuite comment passer du mot $s_0 \cdots s_{i-2}$ au mot $t_0 \cdots t_{j-1}$ (exemple : si l'on supprime `!`, on étudie comment passer de `INFO` à `BINGO`) ;
  - soit en ajoutant t_{j-1}, on étudie ensuite comment passer du mot $s_0 \cdots s_{i-1}$ au mot $t_0 \cdots t_{j-2}$ (exemple : si l'on ajoute `O`, on étudie comment passer de `INFO !` à `BING`) ;
  - soit en remplaçant s_{i-1} par t_{j-1}, on étudie ensuite comment passer du mot $s_0 \cdots s_{i-2}$ au mot $t_0 \cdots t_{j-2}$ (exemple : si l'on change `!` en `O`, on étudie comment passer de `INFO` à `BING`).

  Dans ces trois cas, on effectue une nouvelle modification. Le choix optimal est celui qui correspond à la plus petite distance :

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

On aboutit donc à une relation de récurrence. Le cas d'initialisation est assez aisé.
— Si s est vide, la solution optimale consiste à insérer toutes les lettres de t : $L_{0,j} = j$.
— Si t est vide, la solution optimale consiste à supprimer toutes les lettres de s : $L_{i,0} = i$.

On peut donc écrire un algorithme naïf réalisant cela.

```
Fonction L(s,t) :
    n,p ← dimensions de s,t
    Si n = 0 Alors
        Renvoyer p
    Sinon si p=0 Alors
        Renvoyer n
    Sinon si s[n-1] = t[p-1] Alors
        Renvoyer L(s[ :n-1],t[ :p-1])
    Sinon
        Renvoyer
1+min(L(s[ :n-1],t),L(s,t[ :p-1]),L(s[ :n-1],t[ :p-1]))
Fin Fonction
```

Exercice 1.2.1.
Transformer ce pseudo-code en code `Python` et vérifier que la distance entre les mots `INFO !` et `BINGO` est de 3.

Essayer ensuite de faire tourner cela sur deux mots de longueur 50.

Cet algorithme a une complexité temporelle catastrophique. Pour le voir, plaçons-nous dans le pire des cas, celui où toutes les lettres des mots sont différentes (on pourra par exemple transformer `ABCD` en `EFGH`). Considérons les premiers appels récursifs, dans le cas de notre exemple de la figure 1.
— On commence par calculer $L_{4,4}$. Cela appelle les calculs de $L_{3,4}$, $L_{4,3}$ et $L_{3,3}$.
— On calcule ensuite $L_{3,4}$. Cela appelle les calculs de $L_{2,4}$, $L_{3,3}$ et $L_{2,3}$.
— *etc.*
— On calcule ensuite $L_{4,3}$. Cela appelle les calculs de $L_{3,3}$, $L_{4,2}$ et $L_{3,2}$.
— *etc.*
Bref, on a ici calculé trois fois la même chose ($L_{3,3}$, appelé par les calculs de $L_{4,4}$, de $L_{3,4}$ et de $L_{3,4}$).

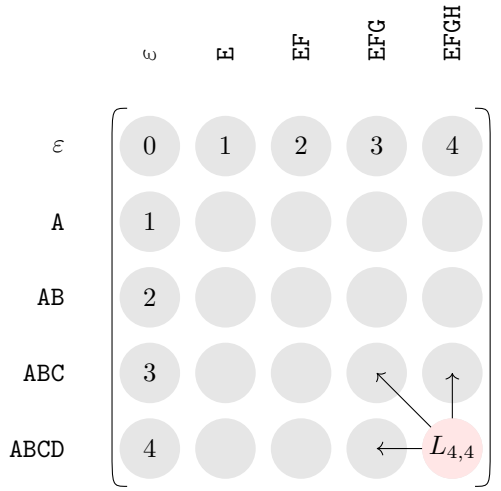


Figure 1 – Premiers appels récursifs

Voici par exemple le nombre de calculs de chaque case pour la matrice M précédente.

$$\begin{pmatrix} 321 & 129 & 41 & 9 & 1 \\ 129 & 63 & 25 & 7 & 1 \\ 41 & 25 & 13 & 5 & 1 \\ 9 & 7 & 5 & 3 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

On a recalculé un grand nombre de fois la même information. Par exemple, on a réalisé 25 fois le calcul de $L_{2,1}$, et 321 fois le calcul de $L_{0,0}$! Notamment, la complexité de cet algorithme appliqué sur une matrice $n \times n$ est exponentielle en n (voir figure 2) !

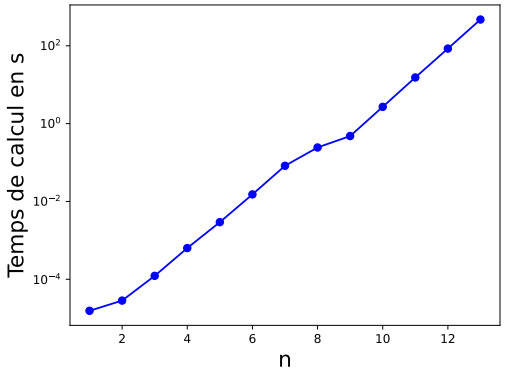


Figure 2 – Complexité expérimentale du calcul naïf par récurrence

1.3 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 1.3.1.
Un tel objet o^* n'est pas nécessairement unique.

Exemple 1.3.2.
Vous avez étudié en première année le problème du rendu de monnaie : il rentre dans notre cadre.

Exemple 1.3.3 (notre exemple)**.**
Le problème du calcul de la distance de Levenshtein rentre aussi dans notre cadre.

Définition 1.3.4 (sous-structure)**.**
Une *sous-structure* est une restriction de notre problème à un ensemble plus petit.

Définition 1.3.5 (sous-structure optimale)**.**
Un problème vérifie la propriété de *sous-structure optimale* si la solution optimale de tout sous-problème est une partie de la solution optimale de notre problème de départ.

Exemple 1.3.6 (sous-structure optimale pour le calcul de la distance de Levenshtein)**.**
Considérons une succession de modifications pour passer de `INFO !` à `BINGO`. Admettons que cette modification puisse se faire au mieux en trois coups (voir la partie de gauche de la figure 3). En effectuant ces modifications (cases carrées et vertes), on transforme `INF` en `BING`.

Si l'on considère le problème du calcul de la distance de Levenshtein entre `INF` et `BING`, alors on obtiendra exactement la même distance que celle obtenue en lorsque l'on a effectué le calcul pour passer de `INFO !` à `BINGO`. Le « chemin » obtenu sera aussi un « sous-chemin » d'un chemin optimal pour la transformation de `INFO !` en `BINGO`.

En effet, si l'on pouvait transformer `INF` et `BING` en strictement moins de 2 opérations, on pourrait en complétant le chemin de la même manière (conservation du `O`, ajout du `!`) obtenir la transformation de `INFO !` en `BINGO` en strictement moins de 3 coups, ce qui est absurde.

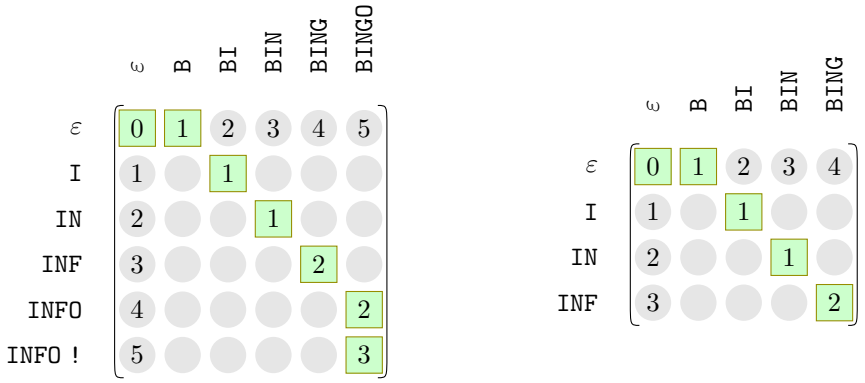


Figure 3 – Propriété de sous-structure optimale pour le calcul de la distance de Levenshtein.

Définition 1.3.7 (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 1.3.8 (chevauchement de sous-problèmes pour le calcul de la distance de Levenshtein)**.**
Dans le calcul de la distance de Levenshtein, le calcul de $L_{4,4}$ amène trois fois au calcul de $L_{3,3}$:
- une fois dans la définition récursive ;
- une fois dans le calcul de $L_{3,4}$;
- une fois dans le calcul de $L_{4,4}$.

Il y a donc bien chevauchement de sous-problèmes.

Définition 1.3.9 (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 1.3.10 (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-problème est appelée *équation de Bellman*.

Exemple 1.3.11 (équation de Bellman pour le calcul de la distance de Levenshtein)**.**
On a montré précédemment que si $i \geqslant 1$ et si $j \geqslant 1$:

$$L_{i,j} = \begin{cases} L_{i-1,j-1} & \text{si } s_{i-1} = t_{j-1}, \\ 1 + \min(L_{i-1,j}, L_{i,j-1}, L_{i-1,j-1}) & \text{si } s_{i-1} \neq t_{j-1}. \end{cases}$$

Remarque 1.3.12.
Bien entendu, il est important d'identifier un cas d'initialisation.

Exemple 1.3.13 (initialisation pour le calcul de la distance de Levenshtein)**.**
Le cas d'initialisation est ici : pour tout i, j,

$$L_{0,j} = j \quad \text{et} \quad L_{i,0} = i.$$

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.

1.4 Calcul de haut en bas, avec memoïsation

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 : *mémoïse*) 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 mémoïser 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.

Exemple 1.4.1 (calcul de haut en bas avec mémoïsation de la distance de Levenshtein)**.**
Cela se fait assez simplement. La première étape consiste à créer une matrice (presque) vide, contenant uniquement les cas de base (voir figure 4). Il suffit ensuite d'appliquer la formule de récurrence, en utilisant le principe de mémoïsation.

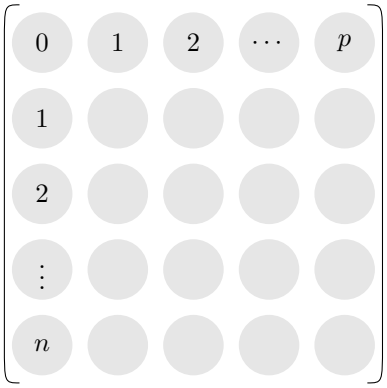


Figure 4 – Matrice à initialiser pour le calcul de la distance de Levenshtein

```
Fonction L_memo(s,t) :
    n,p ← longueurs de s,t
    memo ← matrice vide de taille (n+1)×(p+1)
    Pour i variant de 0 à n, Faire
        memo[i,0] ← i
    Pour j variant de 1 à p, Faire
        memo[0,j] ← j
    Fonction aux(i,j) :
        ##  aux(i,j) renvoie L_{i,j}
        Si memo[i,j] est vide Alors
            Si s[i-1] = t[j-1] Alors
                memo[i,j] ← aux(i-1,j-1)
            Sinon
                memo[i,j]
← 1+min(aux(i-1,j),aux(i,j-1),aux(i-1,j-1))
        Renvoyer memo[i,j]
    Fin Fonction
    Renvoyer aux(n,p)
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 $\mathcal{O}(1)$:

- on peut supposer que la création de `memo` se fait en $\mathcal{O}(np)$ (il y a $(n+1)(p+1)$ cases à créer) ;
- il y a un calcul à réaliser par case vide de `memo` ;
- pour chaque calcul, il y a au plus 2 opérations élémentaires ;
- il y a $\mathcal{O}(np)$ cases vides de `memo`.

Ainsi, la complexité temporelle de l'exécution de `L_memo(s,t)` est en $\mathcal{O}(np)$, quand `s` de longueur n et `t` de longueur p. On remarquera qu'il en est de même pour la complexité spatiale.

Remarque 1.4.2 (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 souvent 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`.

Exercice 1.4.3.
Traduire ceci en code `Python`. Vérifier le résultat sur `INFO !` et `BINGO`.

1.5 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.

Exemple 1.5.1 (calcul naïf de bas en haut de la distance de Levenshtein)**.**
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 $L_{i,j}$ (voir figure 1). On peut donc calculer les $L_{i,j}$ ligne par ligne, de la première à la dernière, et chaque ligne de gauche à droite.

```
Fonction L_basenhaut(s,t) :,
    n,p ← longueurs de s,t
    L ← matrice vide de taille (n+1)×(p+1)
    Pour j variant de 0 à p, Faire
        ##  Création de la première ligne
        L[0,j] ← j
    Pour i variant de 1 à n, Faire
        ##  parcours de la ligne i
        L[i,0] ← i
        Pour j variant de 1 à p, Faire
            Si s[i-1] = t[j-1] Alors
                L[i,j] ← L[i-1,j-1]
            Sinon
                L[i,j] ← 1+min(L[i-1,j],L[i,j-1],L[i-1,j-1])
    Renvoyer L[n,p]
Fin Fonction
```

Exercice 1.5.2.
Calculer $\mathscr{L}(\texttt{GABU}, \texttt{TALUS})$ en remplissant la matrice suivante.

	ε	T	A	L	U	S
ε	0	1	2	3	4	5
G	1					
A	2					
B	3					
U	4					

Exercice 1.5.3.
Traduire ceci en code `Python`. Vérifier le résultat sur `INFO !` et `BINGO`.

Exercice 1.5.4 (difficile)**.**
Réécrire le (pseudo-)code précédent de manière à ne garder en mémoire que deux lignes de la matrice, et non la matrice entière. Analyser la complexité spatiale de ce nouveau code.

1.6 Complexité expérimentale

On trouvera dans la figure 5 deux courbes expérimentales de temps de calcul pour les fonctions données dans les deux parties précédentes.

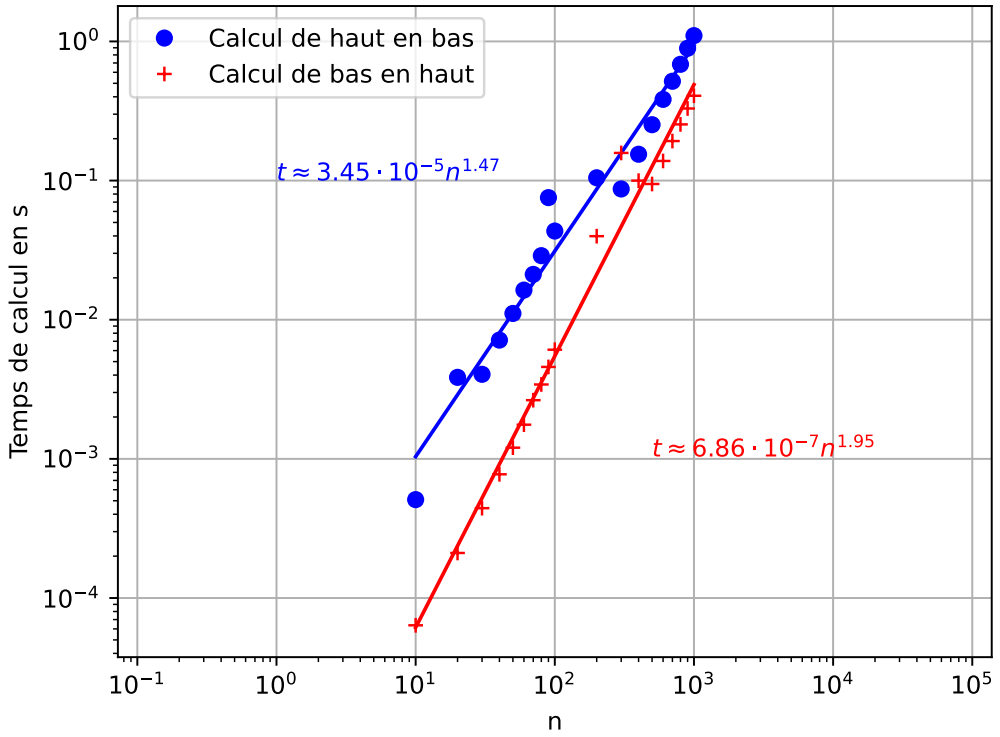


Figure 5 – Complexité expérimentale du calcul de la distance de Levenshtein par programmation dynamique.

On remarquera au moins trois choses.
- Le temps de calcul est spectaculairement moindre que par la méthode naïve : on calcule la distance entre deux mots de 1000 lettres en moins d'une seconde.
- La pente des courbes (si l'on ajustait une droite) est d'environ 2, en échelle logarithmique, ce qui conforte la complexité quadratique 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`.

1.7 Rendu d'un chemin optimal

Dans les exemples précédents, nous avons calculé la distance entre deux mots, mais nous n'avons pas obtenu la suite d'opérations à effectuer pour transformer un mot en l'autre. 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 memoïsation. C'est cette seconde manière de faire qui semble être privilégiée par le programme, nous la mettrons en place dès que possible.

Ici, pour le calcul de la distance de Levenshtein, il y a au moins deux manières de faire :
— à partir de la matrice des distances entre tous les sous-mots (ici, `memo`), reconstruire la suite d'opérations en la remontant petit à petit ;
— sauvegarder dans chaque case la suite de modifications aboutissant à cette case, ou au moins la dernière de ces modifications, au fur et à mesure de l'avancée de l'algorithme.

Exemple 1.7.1 (Calcul de la distance de Levenshtein)**.**
Lors du calcul de $L_{i,j}$, il y a quatre possibilités (voir figure 6).

Figure 6a : soit $s_{i-1} = t_{j-1}$ (voir figure 6a) : le « chemin » est diagonal.

Figure 6b : soit on substitue s_{i-1} par t_{j-1} dans s : le « chemin » est diagonal.

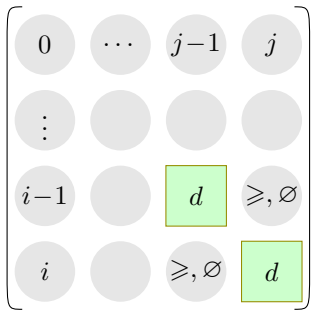
Figure 6c : soit on supprime s_{i-1} dans s : le « chemin » se dirige vers le haut.

Figure 6d : soit on insère t_{j-1} dans s : le « chemin » se dirige vers la gauche.

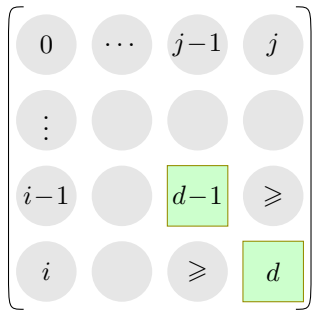
Remarquons que les deux premiers cas ne font en fait qu'un : on substitue s_{i-1} par t_{j-1} dans s, dans les deux cas.

Dans tous les cas, le « chemin » emprunté se « dirige » vers la case de la matrice dont le coefficient est le moindre, par construction. Il suffit de le détecter, pour savoir quelle est la modification à apporter .

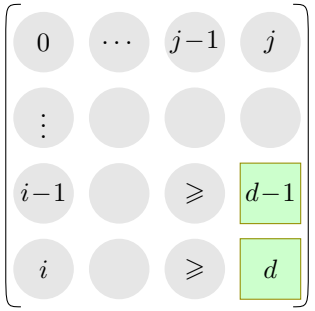
Exemple 1.7.2.
Voici la matrice obtenue sur notre exemple, ainsi que le « chemin » des modifications.



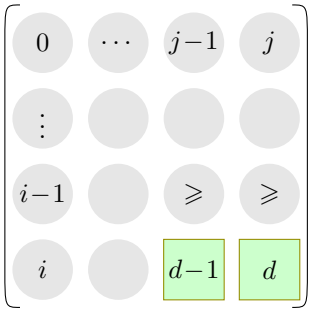
(a) Pas de modification ($s_{i-1} = t_{j-1}$) : $L_{i,j} = L_{i-1,j-1}$



(b) Substitution de s_{i-1} par t_{j-1} : $L_{i,j} = 1 + L_{i-1,j-1}$



(c) Suppression de s_{i-1} : $L_{i,j} = 1 + L_{i-1,j}$



(d) Insertion de t_{j-1} : $L_{i,j} = 1 + L_{i,j-1}$

Figure 6 – Les quatre opérations possibles sur s_{i-1} et t_{j-1}.

	ε	B	I	N	G	O
ε	0	1	2	3	4	5
I	1	1	1	2	3	4
N	2	2	2	1	2	3
F	3	3	3	2	2	3
O	4	4	4	3	3	2
!	5	5	5	4	4	3

On en déduit qu'une suite optimale d'opérations à réaliser est :

— insérer `B` au début de `INFO !` ;
— substituer `F` par `G`
— supprimer `!`.

Exercice 1.7.3.
On donne la matrice des $L_{i,j}$ pour le calcul de $\mathscr{L}$(`VALISE`, `MALHEUR`).

	ε	M	A	L	H	E	U	R
ε	0	1	2	3	4	5	6	7
V	1	1	2	3	4	5	6	7
A	2	2	1	2	3	4	5	6
L	3	3	2	1	2	3	4	5
I	4	4	3	2	2	3	4	5
S	5	5	4	3	3	3	4	5
E	6	6	5	4	4	3	4	5

Déterminer plusieurs suites de 5 opérations transformant `VALISE` en `MALHEUR`.

Exemple 1.7.4 (Reconstruction du chemin de poids minimal)**.**
Cela se fait assez simplement, à partir de la matrice `memo`. En effet, il suffit de remonter le chemin à l'envers, en allant toujours vers la case minimale parmi les cases situées en haut et à gauche de la case considérée, et en interprétant les déplacements comme vu dans la remarque 1.7.1.

On remarquera tout de même que toutes les cases de la matrice `memo` n'ont pas forcément été calculées dans le calcul de haut en bas. Dans ce cas, on interprétera une case vide comme valant $+\infty$.

```
Fonction reconstruction(memo,i,j) :
    Si i = j = 0 Alors
        Renvoyer []
    Sinon si i=0 Alors
        Renvoyer reconstruction(memo,0,j-1) + [Insertion de t_{j-1}]
    Sinon si j=0 Alors
        Renvoyer reconstruction(memo,i-1,0) + [Suppression de s_{i-1}]
    Sinon si memo[i,j-1] = min(memo[i,j-1],memo[i-1,j-1],memo[i-1,j]) Alors
        Renvoyer reconstruction(memo,i,j-1) + [Insertion de t_{j-1}]
    Sinon si memo[i-1,j-1] = min(memo[i,j-1],memo[i-1,j-1],memo[i-1,j]) Alors
        Renvoyer reconstruction(memo,i-1,j-1) + [Substitution de s_{i-1} par t_{j-1}]
    Sinon
        Renvoyer reconstruction(memo,i-1,j) + [Suppression de s_{i-1}]
Fin Fonction
```

Exemple 1.7.5 (calcul de bas en haut de la suite des modifications)**.**
On reprend le principe de la fonction précédente, en remplaçant chaque case de la matrice par un coupe $(L_{i,j}, I)$, où I est la liste des instructions permettant de transformer $s[:i]$ et $t[:j]$.

```
Fonction L_basenhaut2(s,t) :,
    n,p ← longueurs de s,t
    L ← matrice vide de taille (n+1)×(p+1)
    L[0,0] ← (0,[])
    Pour j variant de 1 à p, Faire
        ##  Création de la première ligne
        L[0,j] ← (j,L[0,j-1][1]+["Insertion de t[j-1]"])
    Pour i variant de 1 à n, Faire
        ##  parcours de la ligne i
        L[i,0] ← (i,L[i-1,0][1]+["Suppression de s_{i-1}"])
        Pour j variant de 1 à p, Faire
            Si s[i-1] = t[j-1] Alors
                L[i,j] ← (L[i-1,j-1][0],L[i-1,j-1][1] +
["Conservation de s_{i-1}"])
            Sinon si L[i-1,j][0] =
min(L[i-1,j][0],L[i,j-1][0],L[i-1,j-1][0]) Alors
                L[i,j] ← (1+L[i-1,j][0],L[i-1,j][1] + ["Suppression
de s_{i-1}"])
            Sinon si L[i,j-1][0]
min(L[i-1,j][0],L[i,j-1][0],L[i-1,j-1][0]) Alors
                L[i,j] ← (1+L[i,j-1][0],L[i,j-1][1] + ["Insertion de
t_{j-1}"]
            Sinon
                L[i,j] ← (1+L[i-1,j-1][0],L[i-1,j-1][1] +
["Substitution de s_{i-1} par t_{j-1}"]
    Renvoyer L[n,p]
Fin Fonction
```

Exercice 1.7.6 (difficile).
Mettre en œuvre un des algorithmes précédents. Vérifiez que l'on obtient la liste de modifications suivantes sur l'exemple `INFO !` et `BINGO`.

État de s	Action	Effet sur s
INFO !	Insertion de B	BINFO !
BINFO !	Substitution de I par I	BINFO !
BINFO !	Substitution de N par N	BINFO !
BINFO !	Substitution de F par G	BINGO !
BINGO !	Substitution de O par O	BINGO !
BINGO !	Suppression de !	BINGO

2 Exemples du programme

Voici les exemples cités par le programme (en plus de la distance de Levenshtein). 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 $[\![1, n]\!]$ (*i.e.* deux parties I, J vérifiant $I \cap J = \varnothing$ et $I \cup J = [\![1, n]\!]$) 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 $[\![1, n]\!]$, 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 $I \subset [\![1, n]\!]$ vérifiant

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

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

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

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

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 $[\![1,n]\!]$ 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 ← ∑_{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⩾ 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 !) : $f_0 \leqslant \ldots \leqslant f_{n-1}$.

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 $f_j \leqslant d_{i+1}$), 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 :

$$S_{i+1} = \max\left(S_j + p_{i+1}, S_i\right) \text{ où } j = \max\{\, k \in [\![0,i]\!] \mid f_k \leqslant d_{i+1} \,\}.$$

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 $0 \leqslant j < p$, 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 $0 \leqslant i_0 < \dots < i_{p-1} \leqslant n$.

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 $0 \leqslant i \leqslant n$ et $0 \leqslant j \leqslant p$.

Exercice 2.3.3.
En s'inspirant de l'exemple du calcul de la distance de Levenshtein dans une matrice, 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 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 $1 \leqslant i, j \leqslant n$,
- $M_{i,j}$ 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 2.4.1 (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 $1 \leqslant i < p$, (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 2.4.2.
Voici un exemple de graphe à 5 sommets $s_1, \dots, s_6$ (voir figure 7). 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.

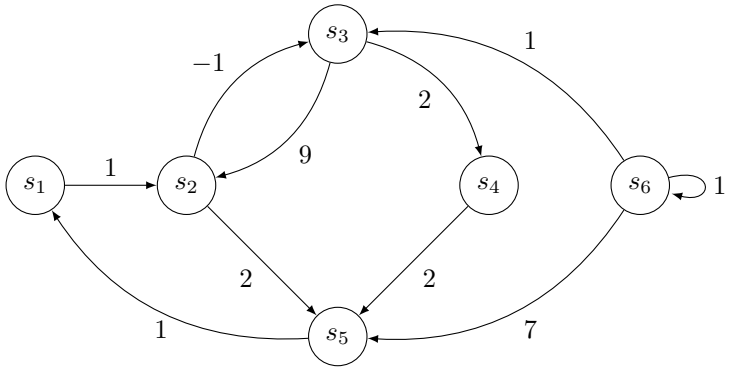


Figure 7 – Un exemple de graphe orienté et pondéré

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.

Pour $1 \leqslant i, j \leqslant n$ et $0 \leqslant k \leqslant n$, 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 $\{s_1, \ldots, s_k\}$ (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 $1 \leqslant i, j, k \leqslant n$:

— 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 $s_1, \ldots, s_{k-1}$, 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 2.4.3 (Algorithme de Floyd-Warshall, calcul de bas en haut)**.**
Cela se fait aisément

```
Fonction floyd_warshall( G ) :
    n ← nombre de sommets de G
    M^0 ← matrice d'adjacence de G
    Pour k variant de 1 à n, Faire
        M^k ← matrice vide nxn
        Pour i variant de 1 à n, Faire
            Pour j variant de 1 à n, Faire
                M^k_{i,j} ← min(M^{k-1}_{i,j}, M^{k-1}_{i,k} + M^{k-1}_{k,j})
    Renvoyer M^n
```

Exercice 2.4.4.
Calculer la complexité temporelle de cet algorithme, en fonction de n.

Exercice 2.4.5.
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 Fibonnaci est définie par :

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

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 $T = [t_0, \ldots, t_{n-1}]$ est un tableau d'entiers de longueur n, alors un sous-tableau de T est un tableau extrait de T, *i.e.* un tableau $T' = [t_{i_0}, \ldots, t_{i_{p-1}}]$, où $0 \leqslant i_0 < \cdots < t_{i_{p-1}} < n$.

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

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