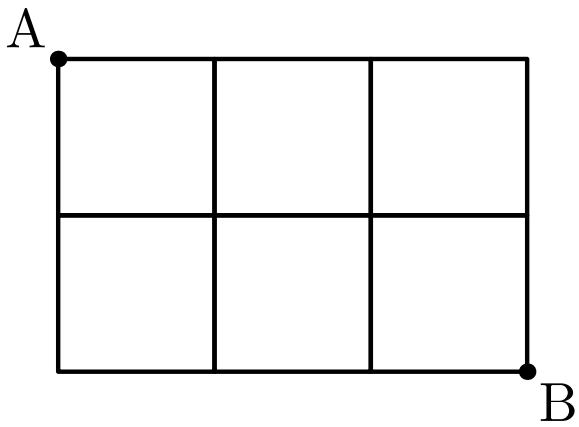
Programmation dynamique

Sommaire

Introduction

- Exemple : soit la grille 3×4 ci-dessous [1]. On veut dénombrer tous les chemins qui, partant de A, arrivent en B et qui n'utilisent que des déplacements sur des traits :
  - horizontaux de gauche à droite ;
  - verticaux de haut en bas.



- Problème : comment réaliser ce dénombrement [2] ?

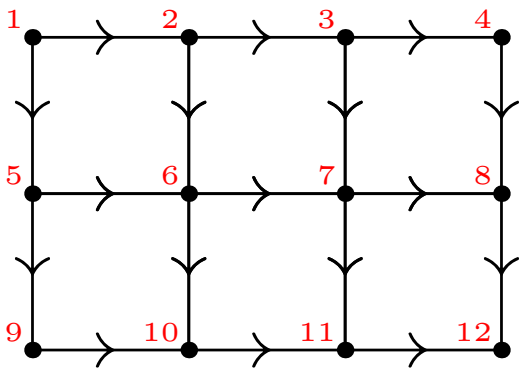
1. Les lignes et colonnes sont les segments horizontaux et verticaux tracés en noir.
2. On peut remarquer qu'il s'agit d'un problème de dénombrement de chemins sur un graphe orienté (sans circuit).

- Principe de la programmation dynamique :
  1. plonger le problème à résoudre dans une famille plus large de problèmes du même type. Cette famille doit contenir des problèmes plus simples, de «taille» décroissante ;
  2. trouver une (ou des) relation(s) (de récurrence) entre ces problèmes, permettant de les résoudre de proche en proche. Trouver également la solution des problèmes de plus petite taille ;
  3. organiser les calculs de façon à ne pas calculer plusieurs fois la même chose.

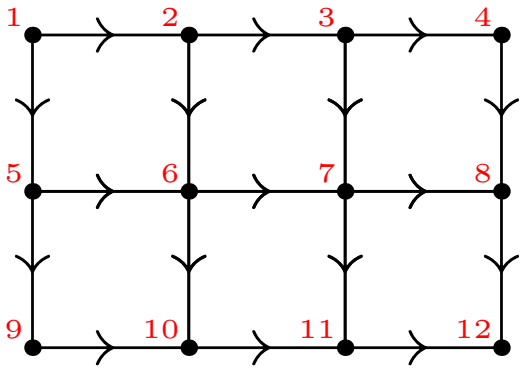
Application à l'exemple précédent sur la grille 3×4 :

- *Problème* : compter le nombre de chemins du sommet A (qu'on peut numéroter 1) au sommet B (numéroté 12) sur le graphe orienté.
- *Famille de problèmes* : dénombrer, sur le graphe orienté, le nombre de chemins du sommet i au sommet 12, pour $i \in \{1; 2; 3; \ldots; 12\}$.
- On note $v(i)$ le nombre de chemins du sommet i au sommet 12.
- *Relation entre ces problèmes* : si i est un sommet et si $\mathrm{Succ}(i)$ est l'ensemble de ses successeurs (immédiats) sur le graphe, alors

$$v(i) = \sum_{j \in \mathrm{Succ}(i)} v(j).$$



- Exemple : $v(6) = v(7) + v(10)$ (car $\text{Succ}(6) = \{7; 10\}$).
- On veut calculer $v(1)$. Par ailleurs, pour $i = 12$, ou 11 ou 8 le problème a une solution évidente ($v(12) = v(11) = v(8) = 1$).
- Si le sommet i est un prédécesseur du sommet j alors la formule précédente montre que le calcul de $v(i)$ suppose celui celui de $v(j)$.
- On a donc intérêt à calculer les $v(i)$ en «remontant» le graphe, en partant de $v(12)$ dont on a la valeur.

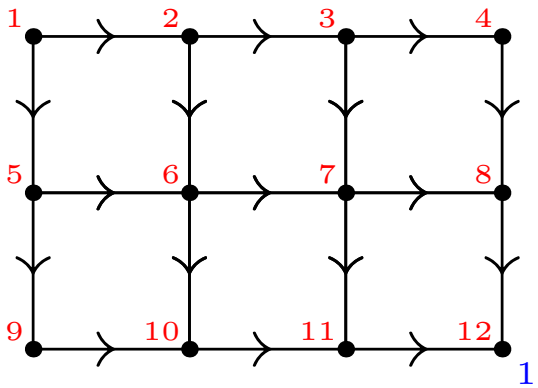


On note :

P : l'ensemble des derniers sommets traités ;

Pred : l'ensemble des prédécesseurs des sommets de P.

Initialisation : $P = \{12\}$ et $v(12) = 1$.

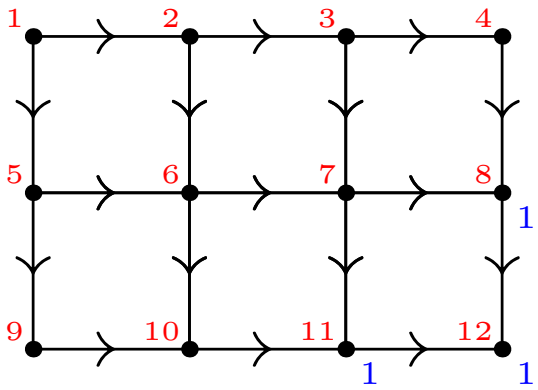


On note :

P : l'ensemble des derniers sommets traités ;

Pred : l'ensemble des prédécesseurs des sommets de P.

Initialisation : $P = \{12\}$ et $v(12) = 1$.



Pred $\leftarrow \{11; 8\}$

Succ$(11) = \{12\}$ $\quad v(11) = v(12) = 1$

Succ$(8) = \{12\}$ $\quad v(8) = v(12) = 1$

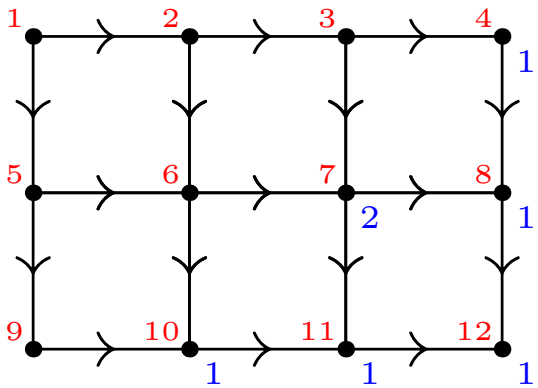
$P \leftarrow \{8; 11\}$

On note :

P : l'ensemble des derniers sommets traités ;

Pred : l'ensemble des prédécesseurs des sommets de P.

Initialisation : $P = \{12\}$ et $v(12) = 1$.



$\text{Pred} \leftarrow \{10; 7; 4\}$

$\text{Succ}(10) = \{11\}$ $\quad v(10) = v(11) = 1$

$\text{Succ}(7) = \{11; 8\}$ $\quad v(7) = v(11) + v(8) = 2$

$\text{Succ}(4) = \{8\}$ $\quad v(4) = v(8) = 1$

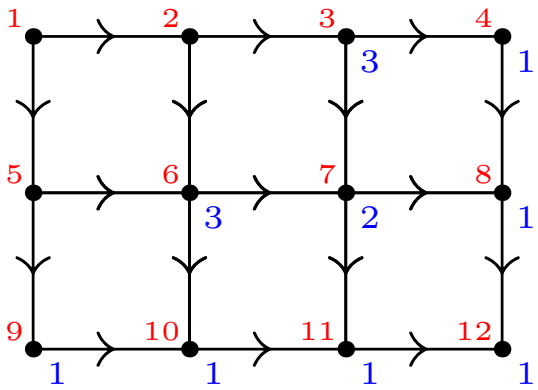
$P \leftarrow \{10; 7; 4\}$

On note :

P : l'ensemble des derniers sommets traités ;

Pred : l'ensemble des prédécesseurs des sommets de P.

Initialisation : $P = \{12\}$ et $v(12) = 1$.



$\text{Pred} \leftarrow \{9; 6; 3\}$

$\text{Succ}(9) = \{10\}$ $\quad v(9) = v(10) = 1$

$\text{Succ}(6) = \{10; 7\}$ $\quad v(6) = v(10) + v(7) = 3$

$\text{Succ}(3) = \{7; 4\}$ $\quad v(3) = v(7) + v(4) = 3$

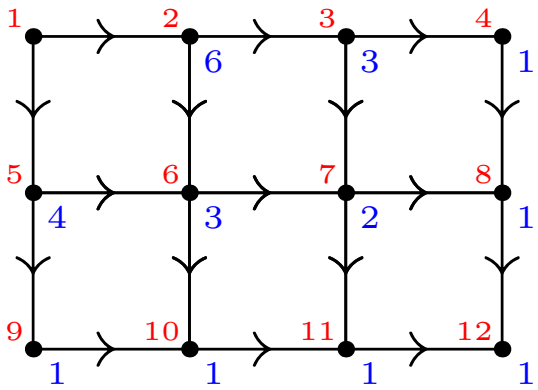
$P \leftarrow \{9; 6; 3\}$

On note :

P : l'ensemble des derniers sommets traités ;

Pred : l'ensemble des prédécesseurs des sommets de P.

Initialisation : $P = \{12\}$ et $v(12) = 1$.



$\text{Pred} \leftarrow \{5; 2\}$

$\text{Succ}(5) = \{9; 6\} \quad v(5) = v(9) + v(6) = 4$

$\text{Succ}(2) = \{6; 3\} \quad v(2) = v(6) + v(3) = 6$

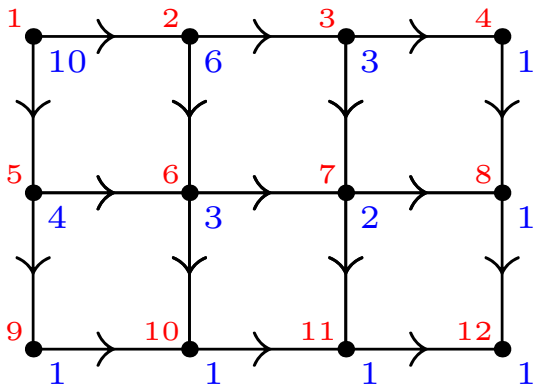
$P \leftarrow \{5; 2\}$

On note :

P : l'ensemble des derniers sommets traités ;

Pred : l'ensemble des prédécesseurs des sommets de P.

Initialisation : $P = \{12\}$ et $v(12) = 1$.



Pred $\leftarrow \{1\}$

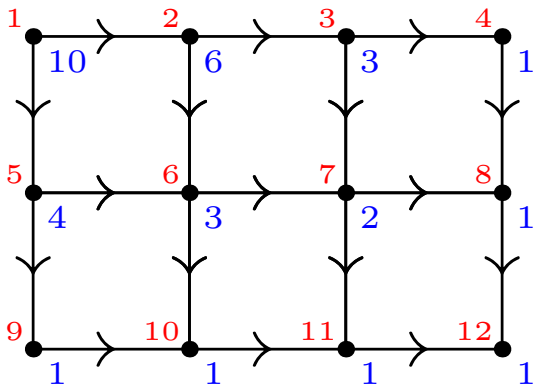
Succ$(1) = \{5; 2\}$ $\quad v(1) = v(5) + v(2) = 10$

On note :

P : l'ensemble des derniers sommets traités ;

Pred : l'ensemble des prédécesseurs des sommets de P.

Initialisation : $P = \{12\}$ et $v(12) = 1$.



Pred $\leftarrow \{1\}$

Succ$(1) = \{5; 2\}$ $\quad v(1) = v(5) + v(2) = 10$

Conclusion : il y a 10 chemins allant du sommet 1 au sommet 12.

On donne ci-dessous l'algorithme en pseudo-code qui met en œuvre ces idées pour résoudre le problème :

Fonction NB_CHEMINS(m, n)

▷ *Calcule et renvoie le nombre de chemins du sommet supérieur gauche au sommet inférieur droit d'une grille de m lignes et n colonnes qui n'empruntent que des traits horizontaux de gauche à droite et verticaux de haut en bas. Algorithme par programmation dynamique. On suppose que mn est le numéro du sommet à atteindre, et 1 celui d'où on part.* ◁

$P \leftarrow \{mn\}$ *# P : ensemble des derniers sommets traités.*
$v(mn) \leftarrow 1$
tant que $1 \notin P$ **faire**
    Déterminer l'ensemble Pred des sommets prédécesseurs des sommets de P
    pour tout $i \in$ Pred **faire**
        Déterminer l'ensemble Succ(i) des successeurs de i
        $v(i) \leftarrow \sum_{j \in \text{Succ}(i)} v(j)$
    $P \leftarrow$ Pred *# mise à jour des derniers sommets traités*
renvoyer $v(1)$

- Cet algorithme est proche d'un algorithme plus général de recherche de plus court chemin dans un graphe orienté sans circuit, qui est un des premiers à avoir illustré la technique de la programmation dynamique.
- L'inventeur de la programmation dynamique est Richard Bellman, mathématicien américain (1920-1984), qui a aussi contribué à l'essor du contrôle optimal et de la théorie de la décision.
- L'expression «programmation dynamique» ne fait pas référence à la programmation entendue au sens de programmation dans un langage de programmation, mais à une technique particulière de planification efficace des calculs.

- Pour les problèmes qu'elle permet de résoudre, on peut démontrer que la programmation dynamique est un algorithme qui est correct c'est à dire qu'elle renvoie effectivement une solution.
- Citons quelques problèmes fameux [3] résolus par programmation dynamique :
  - le problème du sac-à-dos ;
  - le problème du voyageur du commerce ;
  - la recherche d'un plus court chemin dans un graphe ;
  - le problème du rendu de monnaie ;
  - le problème d'alignement de séquences.

3. qui sont pour certains des problèmes NP-complets.

Sommaire

Le problème du rendu de monnaie

- Rappel : on dispose d'un système monétaire utilisant des pièces dont la valeur appartient à un ensemble P fini, par exemple $P = \{1; 2; 5; 10\}$.
- Une somme s étant donnée, on veut réaliser s avec un ensemble de pièces de P et on souhaite que le nombre de ces pièces soit le plus petit possible.
- Pour simplifier, on suppose que :
  - le système ne contient que des pièces à valeur entière ;
  - P contient toujours des pièces de valeur 1 ;
  - le nombre de pièces disponibles de chaque sorte est illimité ;
  - la somme s à réaliser est entière.

- On rappelle aussi qu'il existe un algorithme dit «glouton» pour résoudre ce problème (cf. le programme de Première).
- On se propose ici d'écrire deux algorithmes résolvant le problème du rendu de monnaie (RM en abrégé) :
  - un algorithme de «force brute» ;
  - un algorithme utilisant la programmation dynamique.

Algorithme par force brute

- Notons $\text{RM\_FB}(s, P)$ le nombre minimal de pièces avec lesquelles on peut réaliser la somme s en utilisant des pièces de la liste P. L'objectif est d'écrire l'algorithme d'une fonction calculant et renvoyant $\text{RM\_FB}(s, P)$.
- On remarque que :

$$\text{RM\_FB}(s, P) = \begin{cases} 0 & \text{si } s = 0 \\ 1 + \text{RM\_FB}(s - p^*, P) & \text{si } s > 0 \end{cases}$$

où p^* est la pièce de P (de valeur $\leqslant s$) qui minimise $\text{RM\_FB}(s - p, P)$.

- Avec une écriture plus compacte, pour $s \geqslant 1$:

$$\text{RM\_FB}(s, P) = 1 + \min_{\substack{p \in P \\ p \leqslant s}} \{\text{RM\_FB}(s - p, P)\} \qquad (1)$$

- La recherche du minimum de (1) est réalisable par une énumération exhaustive (complète), dans une boucle «Pour», des valeurs de $\text{RM\_FB}(s - p, P)$.
- On peut, dans cette recherche de minimum, initialiser $\text{RM}(s, P)$ à s

On déduit de ce qui précède l'algorithme suivant de la fonction RM_FB(s, P) écrit en pseudo-code :

```
1: Fonction RM_FB(s, P)
2:     si s = 0 alors
3:         renvoyer 0                                  # cas de base
4:     sinon
5:         nb ← s
6:         pour tout p ∈ P faire
7:             si p ⩽ s alors
8:                 nb ← min(nb, RM_FB(s − p, P))       # appel récursif
9:         renvoyer nb + 1
```

Ex. 1

1. Implémenter l'algorithme précédent en Python. Le tester sur quelques exemples.
2. On suppose que $P = \{1; 2; 5; 10\}$. Le temps de calcul de l'algorithme est-il raisonnable pour toute valeur de s ? Préciser, le cas échéant, à partir de quelle valeur de s ce n'est plus le cas.

- Pour mieux comprendre la complexité en temps de la fonction RM_FB(s, P) on se propose ici d'étudier l'arbre de ses appels récursifs pour $P = \{1; 2\}$ et $s = 100$.
- Pour $s = 0$: il n'y a que l'appel initial.
- Pour $s = 1$: l'appel initial est suivi de l'appel sur 0, soit au total 2 appels récursifs.
- Pour $s = 2$: l'appel initial est suivi de l'appel pour $s = 1$, qui engendre 2 appels au total, et de celui pour $s = 0$, qui en engendre 1. On a au total 4 appels.
- Plus généralement, pour $s \geqslant 2$, le nombre d'appels est la somme du nombre d'appels pour $s - 1$, du nombre d'appels pour $s - 2$, auxquels il faut ajouter l'appel initial.
- Si on note u_n ce nombre d'appels pour $n \geqslant 2$, on a donc

$$u_0 = 1 \quad u_1 = 2 \quad \text{et} \quad u_n = u_{n-2} + u_{n-1} \quad \text{pour tout } n \geqslant 2.$$

- Les premiers termes de (u_n) sont : $1, 2, 4, 7, 12, \ldots$
- On reconnaît, à l'unité près, les termes de la suite de Fibonacci. En fait, on a $u_n = F_{n+3} - 1$ pour tout entier $n \geqslant 0$, où F_n est la suite (déjà rencontrée) définie sur $\mathbb{N}$ par

$$\begin{cases} F_n = n \quad \text{pour } n \in \{0; 1\} \\ F_{n+2} = F_{n+1} + F_n \quad \text{pour tout } n \in \mathbb{N}. \end{cases}$$

- Il se trouve qu'on connaît une formule explicite pour F_n :

$$F_n = -\frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n + \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n$$

- On peut en déduire, pour $n \geqslant 1$, la minoration suivante de F_n

$$F_n > \frac{3}{5\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n > \frac{3}{5\sqrt{5}} \times 1,6^n > \frac{1}{4} \times 1,6^n$$

- Donc, pour $n = 100$,
  $u_{100} = F_{103} - 1 > F_{103} > 0,25 \times 1,6^{103} > 2 \times 10^{20}$ appels récursifs.

Ex. 2

1. Reprendre l'algorithme du RM par force brute. Dans le programme principal, à l'aide de la fonction `perf_counter` du module `time`, mesurer et afficher le temps d'exécution de l'algorithme (en s) pour $s = 10$, 15 et 20 avec $P = \{1; 2\}$.
2. D'après le cours, on a vu que le nombre d'appels récursifs u_n pour $s = n$ est dans ce cas le nombre $F_{n+3} - 1$. Or on sait que $F_{13} = 233$, $F_{18} = 2\,584$ et $F_{23} = 28\,657$. En déduire, à l'aide de la mesure du temps d'exécution réalisé à la question 1, le temps moyen d'exécution d'un appel récursif.
3. Puisque, pour $s = 100$, il y a plus de 2×10^{20} appels récursifs, en déduire une borne inférieure du temps d'attente pour le calcul du RM par force brute avec $s = 100$ et $P = \{1; 2\}$ sur votre ordinateur.

Programmation dynamique

- L'explosion du nombre d'appels récursifs de l'algorithme de RM par force brute vient de ce qu'on appelle de très nombreuses fois la fonction avec la même valeur de s. Par exemple, le calcul du RM avec cet algorithme pour $s = 35$ et $P = \{1; 2\}$ réalise $F_{31} = 1\ 346\ 269$ appels de la fonction avec $s = 5$ et $P = \{1; 2\}$.
- L'idée de l'algorithme de programmation dynamique est de stocker dans un *tableau nb*, indexé par les valeurs de la somme s, les calculs de RM_PD(s, P) (cette fonction réalisant le rendu de monnaie par l'algorithme par programmation dynamique) pour les valeurs croissantes de s.
- Ainsi les appels récursifs sont tous supprimés (relire la ligne 8 de l'algorithme), le calcul de RM_PD(s, P) étant construit sur des appels de la forme RM_PD$(s - p, P)$, avec $s - p < s$. Les valeurs pour $s - p$ seront maintenant lues dans le tableau nb, à savoir dans $nb(s - p)$.
- On initialise le tableau nb de taille $s + 1$ avec des 0 (qui est la valeur exacte pour $s = 0$) et on calcule itérativement la valeur de NB(i, P) pour i allant de 1 à s.

La mise en œuvre de ces idées conduit à l'algorithme écrit en pseudo-code ci-dessous.

```
1: Fonction RM_PD(s, P)
2:     Initialiser un tableau nb avec (s + 1) zéros
3:     pour i allant de 1 à s faire
4:         nb(i) ← i
5:         pour tout p ∈ P faire
6:             si p ⩽ i alors
7:                 nb(i) ← min(nb(i), nb(i − p))
8:         nb(i) ← nb(i) + 1
9:     renvoyer nb(s)
```

Ex. 3

1. Traduire l'algorithme précédent en Python. Le tester sur quelques exemples. Vérifier en particulier que le calcul de RM_PD(s, P) pour $s = 100$ et $P = \{1; 2\}$ est maintenant possible.
2. Dans le cas général, avec un système monétaire de taille l et une somme s, à combien d'opérations peut-on estimer le calcul de RM_PD(s, P) ?

Ex. 4 Modifier le programme précédent de RM par programmation dynamique pour qu'il affiche une liste de pièces réalisant la somme s avec un nombre minimal de pièces. La liste des pièces réalisant s avec un nombre minimal de pièces est-elle unique ? Justifier.

Sommaire

Alignement de séquences

- On dispose de 2 chaînes de caractères, mot1 et mot2, que l'on souhaite aligner.
- Les alignements :
  - doivent garder l'ordre des caractères de chaque mot ;
  - peuvent insérer un (ou plusieurs) «trou(s)» entre les lettres ;
  - ne doivent jamais superposer deux trous ;
  - doivent utiliser toutes les lettres de chaque mot.
- Exemple : voici plusieurs alignements des mots `INFORMATIQUE` et `FANTASTIQUE` (on utilise le tiret, `"-"`, pour matérialiser un trou) :

```
INFOR-MA-TIQU-E
--FANT-ASTIQUE-

---INFORMATIQUE
FANTASTIQUE----

-INFORMATIQUE
FANTAS--TIQUE

INFORM--A-TIQUE
--F--ANTASTIQUE
```

- Chaque alignement se voit attribuer un *score* calculé comme suit :
  - la superposition de deux caractères identiques ajoute 1 ;
  - la superposition de deux caractères distincts (dont le "-") enlève 1.
- Les quatre alignements précédents ont dans l'ordre (de gauche à droite et de haut en bas) comme score : -3, -1, -15 et -1. Quel est d'après vous le meilleur score possible ?
- Le but est de déterminer le meilleur score et un alignement qui lui correspond (il peut y en avoir plusieurs).
- Ce problème de correspondance peut apparaître par exemple :
  - dans les applications de reconnaissance de texte (ex. : correcteur orthographique) ;
  - en bio-informatique, pour comparer des séquences d'ADN (composées des lettres A,G,T,C).

Première solution récursive

- On note mot1 et mot2 les deux chaînes à aligner, n_1 et n_2 leur nombre respectif de caractères.
- Pour déterminer le meilleur alignement des 2 mots, on remarque que seulement 3 situations (mutuellement exclusives) correspondant à 3 types d'alignement peuvent se produire (en supposant que $n_1 \geqslant 1, n_2 \geqslant 1$) :
  1. le dernier caractère de mot1 est superposé au dernier caractère de mot2 ;
  2. le dernier caractère de mot1 est superposé à un tiret ;
  3. le dernier caractère de (l'alignement de) mot1 est un tiret superposé à un caractère de mot2.
- Pour chacun des trois cas, le meilleur score réalisable est calculable à condition de savoir calculer le meilleur score d'un alignement de mots plus courts.
- Par exemple, dans l'exemple des mots `INFORMATIQUE` et `FANTASTIQUE`, le meilleur score réalisable si les deux dernières lettres des deux mots sont superposées est le meilleur score réalisable par alignement des mots `INFORMATIQU` et `FANTASTIQU`, plus 1.

- L'idée précédente montre qu'il doit être possible d'écrire un algorithme récursif.
- Par ailleurs, les cas de base de cette récursivité correspondent au meilleur score réalisable lorsqu'un des 2 mots (ou les 2) sont vides, situation pour laquelle le calcul est facile (le meilleur score est simplement l'opposé du nombre de lettres du mot restant, puisque ses lettres sont alors obligatoirement alignées avec des tirets).
- Bien sûr, le score optimal réalisable s'obtient en prenant le maximum des 3 scores obtenus dans chacune des 3 situations précédemment décrites.
- On est maintenant en mesure d'écrire l'algorithme, donné en pseudo-code page suivante, d'une fonction ALIGNE_REC qui prend en paramètre mot1 et mot2 et renvoie le meilleur score.

Fonction ALIGNE_REC(mot1, mot2)

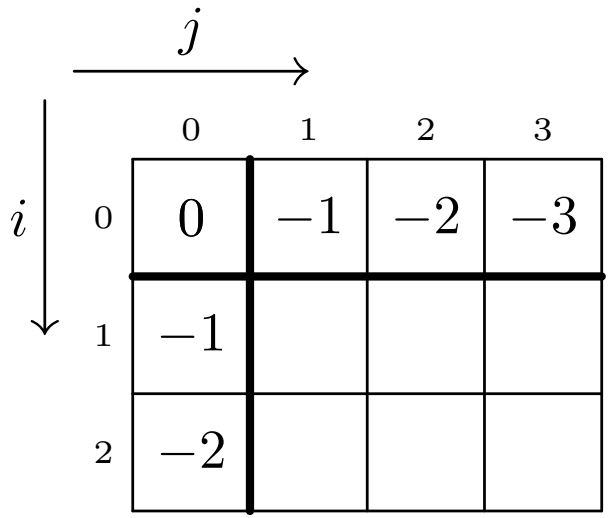
```
n1 ← longueur(mot1)
n2 ← longueur(mot2)
si n1 = 0 alors                                          # cas de base 1
    renvoyer −n2
si n2 = 0 alors                                          # cas de base 2
    renvoyer −n1
Construire mot1p contenant les (n1 − 1) premières lettres de mot1
Construire mot2p contenant les (n2 − 1) premières lettres de mot2
si les dernières lettres de mot1 et mot2 sont égales alors
    score1← 1 + ALIGNE_REC(mot1p, mot2p)      # appel récursif cas 1.1
sinon
    score1← −1 + ALIGNE_REC(mot1p, mot2p)     # appel récursif cas 1.2
score2← −1 + ALIGNE_REC(mot1p, mot2)          # appel récursif cas 2
score3← −1 + ALIGNE_REC(mot1, mot2p)          # appel récursif cas 3
renvoyer max(score1, score2, score3)
```

Ex. 5 Écrire l'algorithme précédent en Python. Le tester sur quelques mots dont le meilleur alignement est connu. Cet algorithme reste-t-il utilisable avec des mots ayant chacun une vingtaine de lettres ? Quelle vous paraît en être la raison ?

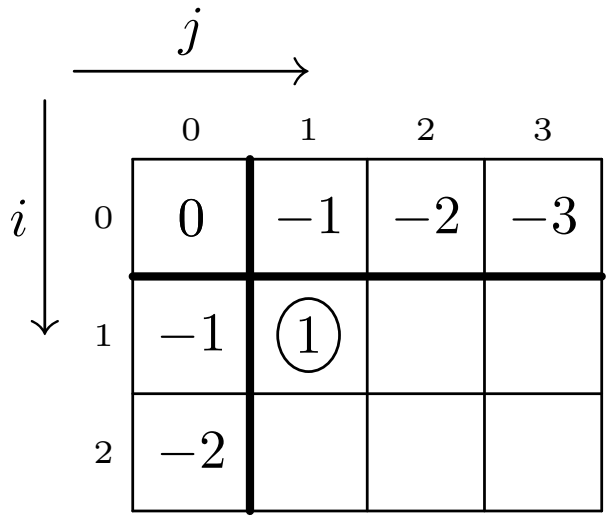
Solution par programmation dynamique

- L'écriture de la solution récursive nous a montré que le calcul du meilleur score pour aligner un mot1 de longueur i et un mot2 de longueur j dépend des solutions optimales trouvées pour :
  - l'alignement des $(i-1)$ premières lettres de mot1 avec les $(j-1)$ premières lettres de mot2 ;
  - l'alignement des $(i-1)$ premières lettres de mot1 avec les j lettres de mot2 ;
  - l'alignement des i lettres de mot1 avec les $(j-1)$ premières lettres de mot2.
- Autrement dit, si on note score(i,j) le meilleur score réalisable en alignant les i premières lettres de mot1 avec les j premières lettres de mot2, alors score(i,j) dépend directement et uniquement de score$(i-1,j-1)$, score$(i,j-1)$ et score$(i-1,j)$ (et des i^{e} et j^{e} lettre des 2 mots).
- Cette dépendance permet de calculer de proche en proche les nombres score(i,j), pour $1 \leqslant i \leqslant n_1$ et $1 \leqslant j \leqslant n_2$, en remplisant un tableau à 2 dimensions, ce qui évite d'appeler récursivement la fonction ALIGNE.

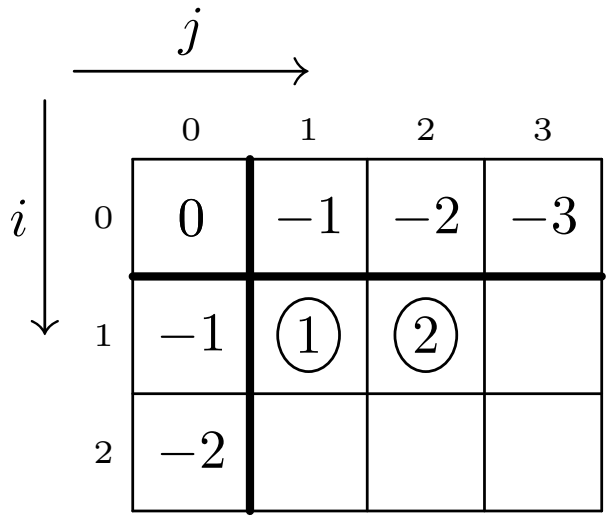
- En convenant que le premier indice du tableau score correspond à une ligne et le second à une colonne, score est un tableau à n_1 lignes et n_2 colonnes.
- Les valeurs de score$(i, 0)$ et de score$(0, j)$ doivent être respectivement initialisées avec $-i$ et $-j$.
- L'ordre de remplissage doit respecter la dépendance de la valeur située en (i, j) aux valeurs situées en $(i-1, j-1)$, $(i, j-1)$ et $(i-1, j)$ (pour les couples (i, j) tels que $i \geqslant 1$ et $j \geqslant 1$).
- Une possibilité est d'utiliser une double boucle imbriquée sur i et j comme illustré sur la figure suivante (où $n_1 = 2$ et $n_2 = 3$).



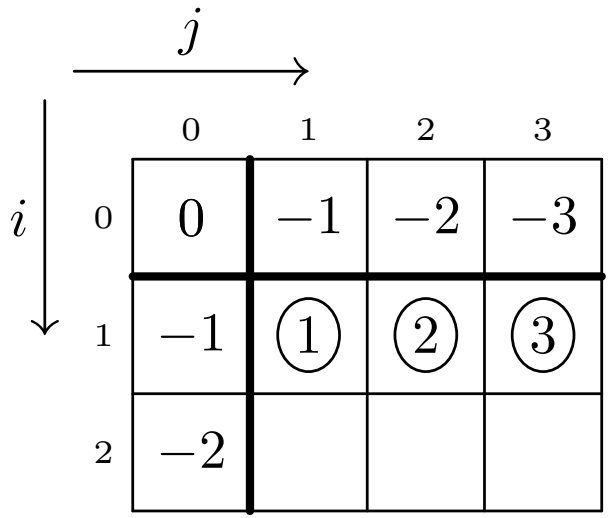
- En convenant que le premier indice du tableau score correspond à une ligne et le second à une colonne, score est un tableau à n_1 lignes et n_2 colonnes.
- Les valeurs de score$(i, 0)$ et de score$(0, j)$ doivent être respectivement initialisées avec $-i$ et $-j$.
- L'ordre de remplissage doit respecter la dépendance de la valeur située en (i, j) aux valeurs situées en $(i-1, j-1)$, $(i, j-1)$ et $(i-1, j)$ (pour les couples (i, j) tels que $i \geqslant 1$ et $j \geqslant 1$).
- Une possibilité est d'utiliser une double boucle imbriquée sur i et j comme illustré sur la figure suivante (où $n_1 = 2$ et $n_2 = 3$).



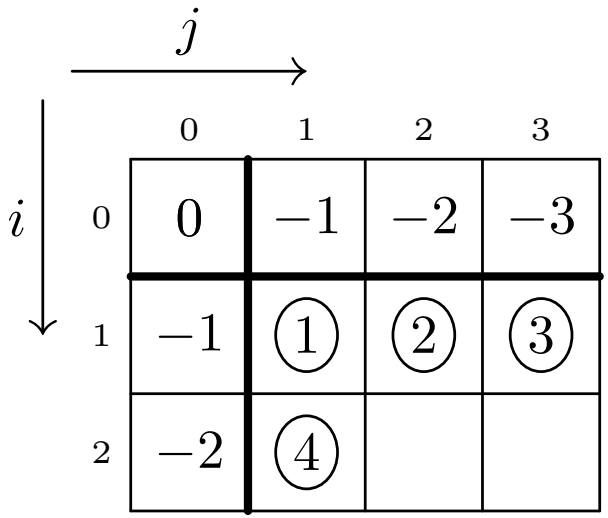
- En convenant que le premier indice du tableau score correspond à une ligne et le second à une colonne, score est un tableau à n_1 lignes et n_2 colonnes.
- Les valeurs de score$(i, 0)$ et de score$(0, j)$ doivent être respectivement initialisées avec $-i$ et $-j$.
- L'ordre de remplissage doit respecter la dépendance de la valeur située en (i, j) aux valeurs situées en $(i-1, j-1)$, $(i, j-1)$ et $(i-1, j)$ (pour les couples (i, j) tels que $i \geqslant 1$ et $j \geqslant 1$).
- Une possibilité est d'utiliser une double boucle imbriquée sur i et j comme illustré sur la figure suivante (où $n_1 = 2$ et $n_2 = 3$).



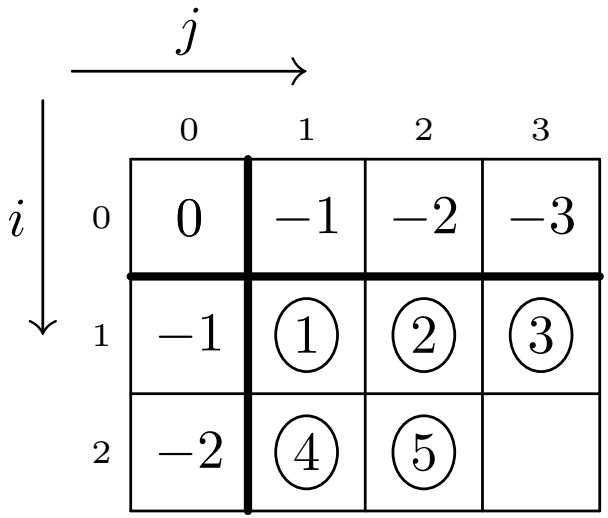
- En convenant que le premier indice du tableau score correspond à une ligne et le second à une colonne, score est un tableau à n_1 lignes et n_2 colonnes.
- Les valeurs de score$(i, 0)$ et de score$(0, j)$ doivent être respectivement initialisées avec $-i$ et $-j$.
- L'ordre de remplissage doit respecter la dépendance de la valeur située en (i, j) aux valeurs situées en $(i-1, j-1)$, $(i, j-1)$ et $(i-1, j)$ (pour les couples (i, j) tels que $i \geqslant 1$ et $j \geqslant 1$).
- Une possibilité est d'utiliser une double boucle imbriquée sur i et j comme illustré sur la figure suivante (où $n_1 = 2$ et $n_2 = 3$).



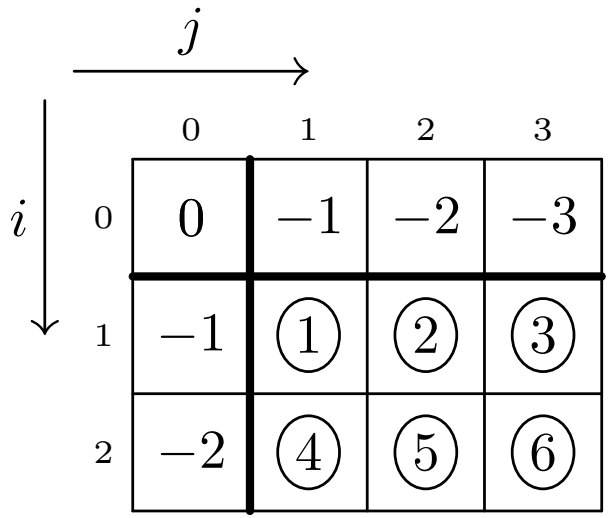
- En convenant que le premier indice du tableau score correspond à une ligne et le second à une colonne, score est un tableau à n_1 lignes et n_2 colonnes.
- Les valeurs de score$(i, 0)$ et de score$(0, j)$ doivent être respectivement initialisées avec $-i$ et $-j$.
- L'ordre de remplissage doit respecter la dépendance de la valeur située en (i, j) aux valeurs situées en $(i-1, j-1)$, $(i, j-1)$ et $(i-1, j)$ (pour les couples (i, j) tels que $i \geqslant 1$ et $j \geqslant 1$).
- Une possibilité est d'utiliser une double boucle imbriquée sur i et j comme illustré sur la figure suivante (où $n_1 = 2$ et $n_2 = 3$).



- En convenant que le premier indice du tableau score correspond à une ligne et le second à une colonne, score est un tableau à n_1 lignes et n_2 colonnes.
- Les valeurs de score$(i, 0)$ et de score$(0, j)$ doivent être respectivement initialisées avec $-i$ et $-j$.
- L'ordre de remplissage doit respecter la dépendance de la valeur située en (i, j) aux valeurs situées en $(i-1, j-1)$, $(i, j-1)$ et $(i-1, j)$ (pour les couples (i, j) tels que $i \geqslant 1$ et $j \geqslant 1$).
- Une possibilité est d'utiliser une double boucle imbriquée sur i et j comme illustré sur la figure suivante (où $n_1 = 2$ et $n_2 = 3$).



- En convenant que le premier indice du tableau score correspond à une ligne et le second à une colonne, score est un tableau à n_1 lignes et n_2 colonnes.
- Les valeurs de score$(i, 0)$ et de score$(0, j)$ doivent être respectivement initialisées avec $-i$ et $-j$.
- L'ordre de remplissage doit respecter la dépendance de la valeur située en (i, j) aux valeurs situées en $(i-1, j-1)$, $(i, j-1)$ et $(i-1, j)$ (pour les couples (i, j) tels que $i \geqslant 1$ et $j \geqslant 1$).
- Une possibilité est d'utiliser une double boucle imbriquée sur i et j comme illustré sur la figure suivante (où $n_1 = 2$ et $n_2 = 3$).



On est maintenant en mesure d'écrire l'algorithme complet qui rassemble les idées précédentes.

Fonction ALIGNE_PD(mot1, mot2)
- $n_1 \leftarrow$ longueur(mot1)
- $n_2 \leftarrow$ longueur(mot2)
- Initialiser un tableau score avec $(n_1 + 1) \times (n_2 + 1)$ zéros
- Initialiser score$(i, 0)$ avec $-i$ $(1 \leqslant i \leqslant n_1)$
- Initialiser score$(0, j)$ avec $-j$ $(1 \leqslant j \leqslant n_2)$
- **pour** i **allant de** 1 **à** n_1 **faire**
  - **pour** j **allant de** 1 **à** n_2 **faire**
    - **si** mot1(i) = mot2(j) **alors**
      - score1$\leftarrow 1 +$ score1$(i-1, j-1)$ *# cas 1.1*
    - **sinon**
      - score1$\leftarrow -1 +$ score1$(i-1, j-1)$ *# cas 1.2*
    - score2$\leftarrow -1 +$ score$(i-1, j)$ *# cas 2*
    - score3$\leftarrow -1 +$ score$(i, j-1)$ *# cas 3*
    - score$(i, j) \leftarrow$ max(score1, score2, score3)
- **renvoyer** score(n_1, n_2)

Sommaire

Exercices

Ex. 6 Traduire en Python l'algorithme précédent de calcul du score du meilleur alignement par programmation dynamique. Le tester sur des mots dont on connaît le meilleur alignement. Le programme est-il encore utilisable pour des mots ayant plus d'une vingtaine de caractères ?

On rappelle qu'un tableau à deux dimensions (ayant ici 2 lignes et 3 colonnes) initialisé avec des 0 peut être créé en Python par une instruction de la forme

```
tab = [[0]*3 for i in range(2)]
```

La valeur contenue dans la ligne 1 et la colonne 2 de `tab` est alors désignée par `tab[1][2]`.

Attention : lors de la traduction du pseudo-code vers Python, bien tenir du compte du fait qu'en Python les listes sont indexées à partir de 0 (en particulier la i^e lettre de mot1 est `mot1[i-1]`) et du fait que pour parcourir les indices i de 1 à n_1 il faut utiliser en Python `for i in range(1, n1 + 1)`.

Ex. 7 Modifier le programme précédent pour qu'il renvoie le meilleur alignement de mot1 et mot2.

Indication : le tableau des scores étant disponible par l'algorithme précédent qui calcule le meilleur score, réfléchir à la possibilité de «remonter» ce tableau à partir de case d'indice (n_1, n_2). Par exemple, si mot1 est de longueur 3 et mot2 de longueur 2, le meilleur score est lu en score$(3, 2)$.

- Il s'agit de savoir si ce meilleur score est obtenu :
  1. en superposant la dernière de mot1 et mot2 ;
  2. en superposant la dernière lettre de mot2 avec un tiret ;
  3. en superposant la dernière lettre de mot1 avec un tiret.
- Le cas 1 se produit si :
  - Soit score$(3, 2) = 1 +$ score$(2, 1)$ et mot1 et mot2 ont une dernière lettre identique ;
  - Soit score$(3, 2) = -1 +$ score$(2, 1)$ et mot1 et mot2 ont une dernière lettre différente.
- Le cas 2 se produit si score$(3, 2) = -1 +$ score$(3, 1)$.
- Le cas 3 se produit si score$(3, 2) = -1 +$ score$(2, 2)$.
- On remarque que, parmi les 3 cas précédents, plusieurs peuvent être réalisés. Ce dont on est sûr, c'est que nécessairement un des trois s'est réalisé (et qu'il correspond au meilleur score possible).

- On peut donc décider de remonter de score$(3, 2)$ à l'une des cases d'indice $(2, 1)$, $(3, 1)$ ou $(2, 2)$, par exemple la première qui vérifie l'égalité qui lui est associée.
- En remontant d'une ligne (cas 3), d'une colonne (cas 2) ou d'une ligne et d'une colonne (cas 1) on ajoute respectivement à l'alignement une lettre de mot1 superposée à un tiret, une lettre de mot2 superposée à un tiret ou les deux dernières lettres de mot1 et mot2 superposées. De façon concomitante on diminue le nombre de lettres restant à aligner.
- On arrive ainsi nécessairement à un des mots de longueur nulle. Il ne reste plus alors qu'à superposer les lettres du mot restant à des tirets.

Ex. 8 La suite de Fibonacci est la suite d'entiers qui a été rappelée dans le cours. Écrire une fonction `fibonacci(n)` en Python qui calcule et renvoie le terme de rang n de cette suite en utilisant le principe de la programmation dynamique.

Ex. 9 Calculer et remplir à la main le tableau des scores d'alignement pour les 2 mots `CHAT` et `CAT`.

Ex. 10 Quel est le score maximal de l'alignement des mots (de 10 lettres) `AAAAAAAAAA` et `BBBBBBBBBB` ? Quelle est la forme du tableau calculé par l'algorithme de programmation dynamique sur ces 2 mots ?

Ex. 11 Écrire une fonction `chemins(m, n)` qui calcule et renvoie le nombre de chemins sur une grille $m \times n$ menant du coin supérieur gauche au coin inférieur droit en se déplaçant uniquement sur des traits horizontaux vers la droite et sur des traits verticaux vers le bas. On doit vérifier que `chemins(10, 10)` est égal à 184 756.

Ex. 12 Écrire en Python une fonction `binomial(k, n)` qui calcule et renvoie, à l'aide d'un algorithme de programmation dynamique, le coefficient binomial k parmi n du triangle de Pascal (défini pour $n \in \mathbb{N}$ et $0 \leqslant k \leqslant n$).