Chapitre II
Programmation dynamique

1. Dictionnaires

Cette première partie a pour objet l'étude d'une structure de données que vous avez déjà rencontré en première année : les dictionnaires. Cette structure de données répond à la problématique suivante : comment rechercher efficacement de l'information dans un ensemble géré de façon dynamique (c'est à dire dont le contenu est susceptible d'évoluer au cours du temps).

Les dictionnaires sont couramment utilisées en informatique : c'est par exemple la structure de données utilisée pour gérer les systèmes de noms de domaine (DNS, pour *Domain Name System*). Les ordinateurs connectés à un réseau comme Internet possèdent une adresse numérique (en IPv4 par exemple, celles-ci sont représentées sous la forme xxx.xxx.xxx.xxx, où xxx est un nombre hexadécimal variant entre 0 et 255). Pour faciliter l'accès aux systèmes qui disposent de ces adresses, un mécanisme a été mis en place pour associer un nom (plus facile à retenir) à une adresse IP. Ce mécanisme utilise un dictionnaire dans laquelle les clefs sont les noms de domaine et les valeurs les adresses IP.

1.1 Description

Un *dictionnaire* (ou mieux une *table d'association*) est un type de données associant un ensemble de clefs à un ensemble de valeurs. Plus formellement, si C désigne l'ensemble des clefs et V l'ensemble des valeurs, un dictionnaire est un sous-ensemble T de $C \times V$ tel que pour toute clef $c \in C$ il existe *au plus* un élément $v \in V$ tel que $(c, v) \in T$. Les éléments de T sont appelés des *associations*.

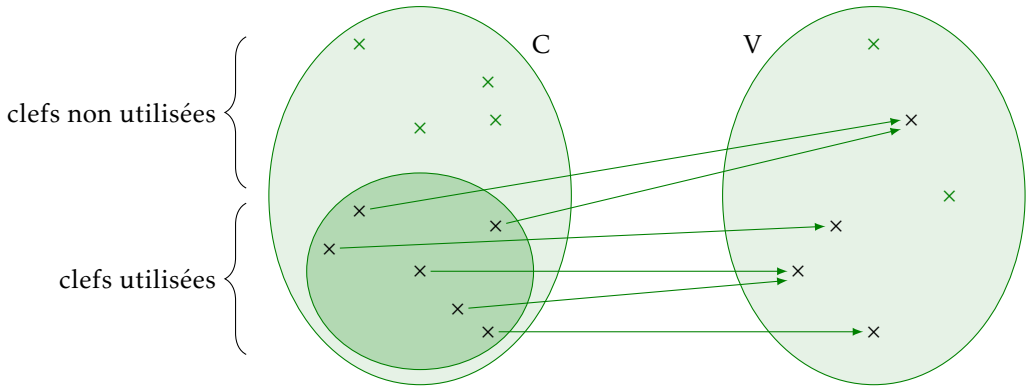


Figure 1 – Représentation informelle d'un dictionnaire.

Un dictionnaire supporte en général les opérations suivantes :

- *ajout* d'une nouvelle association $(c, v) \in C \times V$ dans T ;
- *suppression* d'une association (c, v) de T ;
- *existence* d'une association (c, v) dans T pour une clef $c \in C$ donnée ;
- *lecture* de la valeur v associée à une clef c présente dans T.

En Python, la création d'un dictionnaire se réalise en suivant la syntaxe `{c1: v1, ... , cn: vn}` où $c_1, \ldots, c_n$ sont des clefs (nécessairement deux-à-deux distinctes) et $v_1, \ldots, v_n$ les valeurs qui leur sont associées. Ainsi, `{ }` crée un dictionnaire vide.

Si `D` est un dictionnaire et `c` une clef,

- `D[c] = v` ajoute une nouvelle association si la clef n'est pas présente dans le dictionnaire, et modifie l'association précédente sinon ;

- `del D[c]` supprime une association si la clef est présente dans le dictionnaire, et déclenche l'exception `KeyError` sinon ;
- l'expression `c in D` renvoie un booléen indiquant si la clef est présente ou non dans le dictionnaire ;
- `D[c]` renvoie la valeur associée à la clef si celle-ci est présente dans le dictionnaire, et déclenche l'exception `KeyError` sinon.

Notons que, pour des raisons qui apparaîtrons plus loin, les clefs d'un même dictionnaire doivent impérativement appartenir à un type immuable.

1.2 Mise en œuvre pratique d'un dictionnaire

Il est communément admis (même si ce n'est pas tout-à-fait correct) que les quatre opérations de base d'un dictionnaire se réalisent avec une complexité temporelle constante ; pour comprendre comment une telle performance est possible, nous allons décrire quelques méthodes d'implémentation de cette structure.

Si les clefs étaient des entiers compris entre 0 et $m-1$, le problème serait simple : il suffirait d'utiliser un tableau de taille m. Comme ce n'est en général pas le cas, on se ramène à cette situation en utilisant une fonction $f : \mathrm{C} \to [\![0, m-1]\!]$ associant aux différentes clefs $c \in \mathrm{C}$ possibles un entier dans $[\![0, m-1]\!]$. Cependant, le nombre de clefs possibles étant très important, le cardinal de C est beaucoup plus grand que m et une telle fonction ne sera pas injective. Il existera donc des couples (c_1, c_2) dans C tels que $c_1 \neq c_2$ et $f(c_1) = f(c_2)$. Un tel couple (c_1, c_2) est appelé une *collision*, et il faudra trouver une solution pour gérer ces collisions lorsqu'elles se produiront.
Une solution possible pour résoudre les collisions consiste à stocker les valeurs dont les clefs sont rentrées en collision dans un même « paquet ». Si la répartition entre les différents paquets est équilibrée, chaque paquet ne contiendra qu'un petit nombre de valeurs, et on retrouvera rapidement la valeur associée à une clef en la cherchant dans son paquet.
La figure 2 illustre cette solution avec $f(c_1) = f(c_4)$ et $f(c_3) = f(c_5) = f(c_6)$.

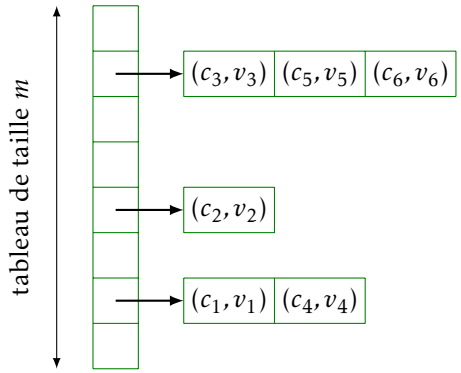


Figure 2 – Les clefs qui entrent en collision sont stockées dans le même paquet.

Si on considère que le calcul de $f(c)$ se réalise en temps constant et que chaque paquet est de petite taille, les quatre opérations se réalisent effectivement avec une complexité temporelle constante.

■ Fonction de hachage

Pour définir la fonction f, on utilise une fonction h, appelée *fonction de hachage*, associant un entier à une clef de C, et on définit f en posant :

$$f(c) = h(c) \bmod m$$

Pour conduire à une implémentation efficiente, une fonction de hachage doit :

- être facile à calculer ;
- avoir une distribution la plus uniforme possible.

Cette dernière condition est motivée par le souhait de minimiser le nombre de collisions. Pour que cette fonction assure une bonne répartition des clefs dans les différents emplacements du tableau, il faut, *de manière informelle*, qu'étant donné un entier $i \in [\![0, m-1]\!]$, la probabilité que $h(c) \bmod m$ soit égal à i soit de l'ordre de $1/m$.

Dans ces conditions, si k désigne le nombre de clefs distinctes de T, la probabilité pour qu'il y ait au moins une collision est égale à :

$$1 - \frac{m!}{m^k(m-k)!}$$

Par exemple, pour $m = 1\,000\,000$, la probabilité pour qu'il y ait collision dépasse 50% lorsque le nombre de clefs dépasse 1 200 ; pour $k = 2\,500$ la probabilité qu'il y ait collision dépasse 95%.
Ces chiffres montrent que les collisions sont, quoi qu'on fasse, rapidement inévitables.

Choix de la fonction de hachage
C'est bien évidemment un problème très complexe, que nous n'aborderons pas. Sachez seulement que Python propose une fonction de hachage : si `x` est un objet immuable (`int`, `str`, `float`, ...) alors `hash(x)` renvoie un entier répondant peu ou prou aux exigences d'une fonction de hachage :

```
In [1]: hash(12345)
Out[1]: 12345

In [2]: hash("12345")
Out[2]: -3486454717630806608

In [3]: hash((1, 2, 3, 4, 5))  # un tuple est immuable
Out[3]: -5659871693760987716

In [4]: hash([1, 2, 3, 4, 5])  # une liste est mutable
TypeError: unhashable type: 'list'
```

Remarque. Une fonction de hachage a d'autres usages. Par exemple, lorsque vous choisissez un mot de passe sur un site sécurisé, ce dernier ne va pas stocker votre mot de passe en clair, mais va stocker le résultat de l'application d'une fonction de hachage h à votre mot de passe. Sachant qu'il est très difficile de trouver les antécédents d'un entier par la fonction h, pirater le site ne mettra pas en cause la sécurité de votre mot de passe.
Une autre application des fonctions de hachage est le contrôle d'intégrité d'un fichier informatique : à chaque fichier est associé une valeur par une fonction de hachage (vous connaissez peut-être le MD5) qui permet de vérifier si un fichier téléchargé a été transmis correctement.

1.3 Résolution des collisions

■ Résolution des collisions par chaînage

On l'a déjà dit, une première solution consiste, pour deux clefs c_1 et c_2 entrant en collision, à stocker les associations (c_1, v_1) et (c_2, v_2) dans un même paquet (une liste par exemple) (voir figure 2).
Pour trouver la valeur associée à une clef c présente dans le dictionnaire, on procède alors en deux temps :

- on calcule $f(c) = h(c) \bmod m$ pour déterminer l'emplacement du tableau où se trouve le paquet contenant l'association recherchée ;
- on procède à une recherche naïve dans le paquet pour trouver cette association.

Cette méthode présente l'avantage de pouvoir contenir un nombre k de clefs plus grand que le nombre m de cases du tableau.
Si on note $\alpha = \frac{k}{m}$ le taux de remplissage du dictionnaire, les paquets auront une taille moyenne de α (sous une hypothèse de hachage uniforme) et la recherche d'une association dans le dictionnaire utilisera un nombre de comparaison en moyenne de l'ordre de α.

Exercice 1

On dispose d'une table de hachage de taille m dont les collisions sont résolues par chaînage.
Exécuter manuellement l'algorithme d'insertion dans la table pour $m = 9$ et $f : c \mapsto c \bmod m$ des clefs suivantes : 5, 28, 19, 15, 20, 33, 12, 17, 10.

■ Adressage ouvert

Une autre solution consiste, en cas de collision, à chercher un emplacement libre dans lequel déposer la nouvelle association. La recherche d'un emplacement libre porte le nom de *sondage*.

- le sondage *linéaire* consiste à partir de $i = h(c)$ et à chercher une place libre en testant successivement les cases d'indices $(i+1) \bmod m$, $(i+2) \bmod m$, $(i+3) \bmod m$,… jusqu'à trouver un emplacement libre.
- le sondage *quadratique* procède de même mais en sondant les cases d'indices $(i+1) \bmod m$, $(i+1+2) \bmod m$, $(i+1+2+3) \bmod m$,…

L'inconvénient d'un sondage linéaire est qu'il y a un risque de former des « agrégats », autrement dit de longues successions de cases contigües occupées, qui nuisent à la répartition uniforme recherchée.

0	1	2	3	4	5	6	7	8
(c_5, v_5)		(c_1, v_1)	(c_4, v_4)	(c_3, v_3)			(c_2, v_2)	

FIGURE 3 – Un exemple de sondage linéaire : $f(c_6) = 2$ mais l'association (c_6, v_6) sera stockée dans la case 5.

Remarque. D'autres solutions plus complexes existent, comme par exemple sonder les cases $i + h'(c) \bmod m$, $i + 2h'(c) \bmod m$, $i + 3h'(c) \bmod m$,… où h' est une autre fonction de hachage, mais aucune ne résout complètement le problème de la création d'agrégats.

Évidemment, un adressage ouvert exige que le nombre k de clefs soit inférieur à la taille de la table m. En outre, on imagine aisément que lorsque le rapport $\alpha = \dfrac{k}{m}$ se rapproche de 1, il devient de plus en plus difficile de trouver un emplacement vide : si $\alpha = 0{,}5$ un ajout nécessite en moyenne deux sondages, contre dix lorsque $\alpha = 0{,}9$. Aussi, lorsque α devient trop grand il est nécessaire de créer une table plus grande (la taille est en général doublée) pour préserver les performances.

Exercice 2

Dans cet exercice, on s'intéresse à la méthode de résolution des collisions par adressage ouvert à l'aide d'un sondage linéaire.

a. Exécuter manuellement l'algorithme d'insertion dans une table de taille $m = 9$ des clefs 5, 28, 19, 15, 20, 33, 12, 17, 10 avec la fonction de hachage $f(c) = c \bmod 9$.

b. On représente le dictionnaire par un tableau `D=[None, None, ..., None]` de m cases et on prend toujours pour fonction de hachage $f(c) = c \bmod m$. les clefs sont des entiers et les valeurs des chaînes de caractères. Rédiger les fonctions de lecture et d'ajout associées : `add(D, c, v)` ajoute au dictionnaire `D` l'association (c, v), `find(D, c)` renvoie la valeur associée à la clef c. On supposera que le dictionnaire n'est pas complètement rempli.

c. Quel problème se pose lors de la suppression de certaines associations de la table ? Comment peut-on le résoudre ? Rédiger la fonction `remove(D, c)` correspondante.

1.4 Ensembles

Les dictionnaires permettent aussi de représenter des ensembles : il suffit de supprimer les valeurs associées aux clefs pour faire d'un dictionnaire un ensemble de clefs.
En Python, le type correspondant s'appelle `set`. Un ensemble peut être défini par l'énumération de ses éléments initiaux : `s = {2, 3, 5, 7, 11, 13}` définit un ensemble. En revanche, pour qu'il n'y ait pas de confusion avec le dictionnaire vide il faut, pour définir l'ensemble vide, écrire `set()`.

Exemple. La conversion d'une liste en ensemble est un moyen simple de supprimer les doublons. Dans le script suivant, je tire au hasard 1 000 nombres compris entre 0 et 999 puis je supprime les doublons.

```
In [1]: import numpy.random as rd
In [2]: s = set(rd.randint(1000, size=1000))

In [3]: len(s)
Out[3]: 630
```

Après suppression des doublons il n'en reste que 630.

Remarque. L'espérance du cardinal de s est d'environ 632. Sauriez-vous le démontrer ?

2. Programmation dynamique

2.1 Un problème posé par la programmation récursive

Nous allons nous intéresser au calcul du coefficient binomial $\binom{n}{p}$. Une solution consiste à utiliser la programmation récursive et la formule de Pascal, ce qui nous amène à écrire :

```
def binom(n, p):
    if p == 0 or n == p:
        return 1
    return binom(n - 1, p - 1) + binom(n - 1, p)
```

Malheureusement, cette fonction s'avère très peu efficace, même pour de relativement faibles valeurs de n et p : à titre d'illustration, il faut 64 secondes à mon ordinateur pour calculer $\binom{30}{15}$.

La raison en est facile à comprendre : lorsqu'on observe par exemple l'arbre de calcul de $\binom{5}{2}$ on constate que des appels récursifs sont identiques et donc superflus (figure 4).

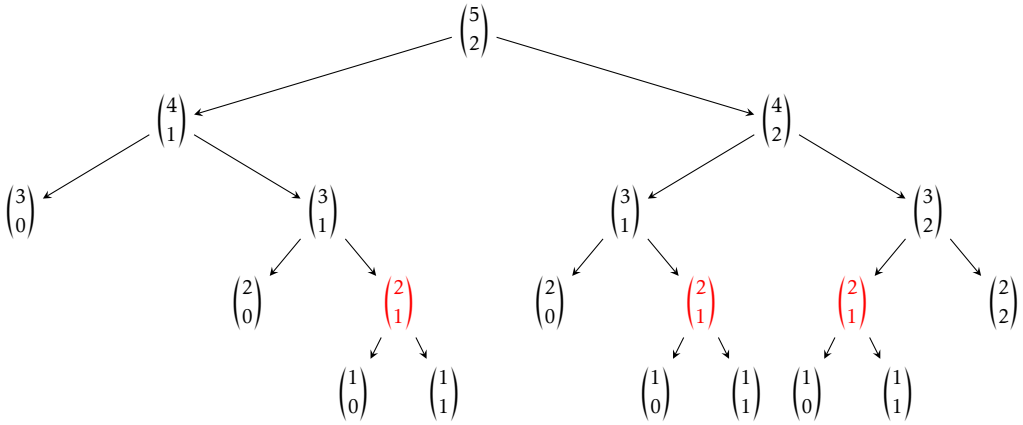


FIGURE 4 – Le calcul de $\binom{5}{2}$ fait appel trois fois au calcul de $\binom{2}{1}$.

Nous pouvons par exemple constater que le calcul de $\binom{5}{2}$ nécessite de calculer trois fois $\binom{2}{1}$. Et l'expérience montre que le calcul de $\binom{30}{15}$ fait appel 40 116 600 fois (!) au calcul de $\binom{2}{1}$.

Complexité temporelle
Pour évaluer la complexité de cette fonction, on note $C(n,p)$ le nombre d'additions réalisées par cette fonction, on dispose des relations :

$$C(n,0) = C(n,p) = 0 \quad \text{et} \quad \forall p \in [\![1, n-1]\!], \quad C(n,p) = C(n-1,p-1) + C(n-1,p) + 1$$

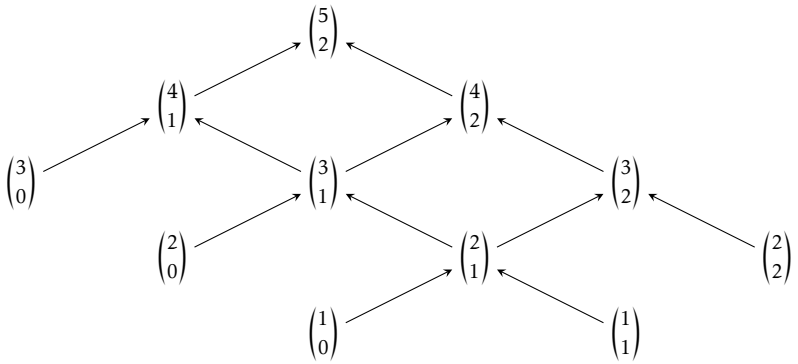
On démontre alors par récurrence sur $n \in \mathbb{N}$ que pour tout $p \in [\![0,n]\!]$, $C(n,p) = \binom{n}{p} - 1$. Or la formule de Stirling permet d'établir l'équivalent : $\binom{2n}{n} \sim \frac{4^n}{\sqrt{\pi n}}$; le calcul de $\binom{2n}{n}$ par cette fonction est donc de complexité exponentielle.

Où est le problème ?

Le problème à résoudre, ici le calcul de $\binom{n}{p}$, se ramène à la résolution de deux sous-problèmes : le calcul de $\binom{n-1}{p-1}$ et de $\binom{n-1}{p}$, *sous-problèmes qui sont en interaction*.
Par exemple, on constate sur la figure 4 que le calcul de $\binom{4}{1}$ et le calcul de $\binom{4}{2}$ font tous deux appel au même sous-problème : le calcul de $\binom{3}{1}$.
Ainsi, la présence de sous-problèmes en interaction peut faire croitre très rapidement la complexité d'une fonction, au point d'en rendre son usage rédhibitoire.

■ La solution proposée par la programmation dynamique

La solution proposée par la programmation dynamique consiste à commencer par résoudre les plus petits des sous-problèmes, puis de combiner leurs solutions pour résoudre des sous-problèmes de plus en plus grands.
Concrètement, le calcul de $\binom{5}{2}$ se réalise en suivant le schéma :



Pour réaliser ce type de solution on utilise souvent un tableau, ici un tableau bi-dimensionnel $(n+1)\times(p+1)$ (dont seule la partie pour laquelle $i \geqslant j$ sera utilisée). Ce tableau sera progressivement rempli par les valeurs des coefficients binomiaux, en commençant par les plus petits (figure 5).

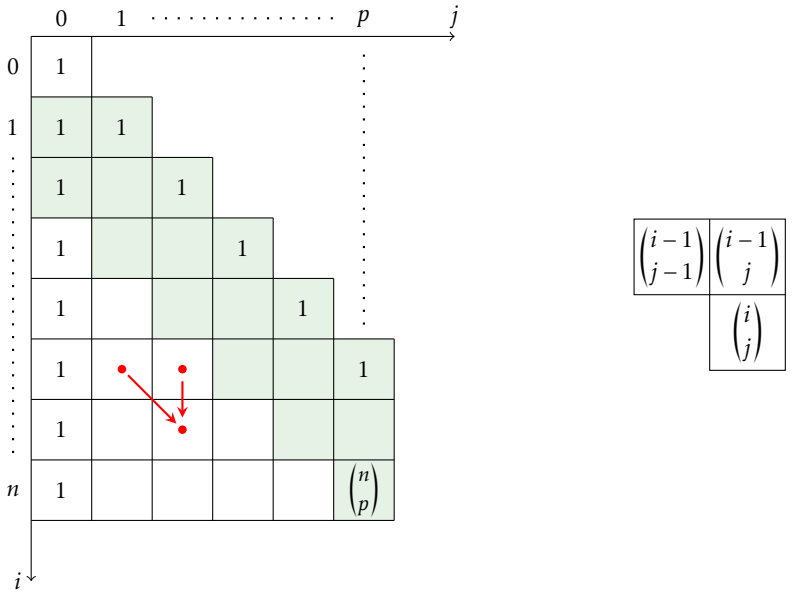


FIGURE 5 – Le schéma de dépendance du calcul de $\binom{n}{p}$.

Il faut faire attention à bien respecter la relation de dépendance (modélisée par les flèches sur le schéma ci-dessus) pour remplir les cases de ce tableau : la case destinée à recevoir la valeur de $\binom{i}{j}$ ne peut être remplie qu'après les cases destinées à recevoir $\binom{i-1}{j-1}$ et $\binom{i-1}{j}$.

```
1  def binom(n, p):
2      t = np.zeros((n + 1, p + 1), dtype=np.int64)
3      for i in range(0, n + 1):
4          t[i, 0] = 1
5      for i in range(1, p + 1):
6          t[i, i] = 1
7      for i in range(2, n + 1):
8          for j in range(1, min(p, i) + 1):
9              t[i, j] = t[i - 1, j - 1] + t[i - 1, j]
10     return t[n, p]
```

Au prix d'un coût spatial (la création du tableau) cet algorithme est bien plus efficace que l'algorithme récursif initial puisque sa complexité temporelle et spatiale est maintenant en $O(np)$.

Remarque. Notons que cette solution n'est pas encore optimale : il est facile de constater sur le schéma de dépendance que l'algorithme ci-dessus remplit des cases inutiles pour le calcul de $\binom{n}{p}$: seules celles qui sont colorées sont nécessaires.

Un autre inconvénient, plus important celui-là réside dans la perte de lisibilité de l'algorithme, comparativement à l'algorithme récursif. L'idéal serait donc de combiner l'élégance de la programmation récursive avec l'efficacité de la programmation dynamique.

La solution existe, elle porte le nom de *mémoïsation*. Elle consiste à associer à la fonction un dictionnaire qui va mémoriser le résultat du calcul réalisé. Ainsi, à chaque fois que le programme aura besoin de calculer une valeur, il ira voir dans le dictionnaire si la valeur dont il a besoin a déjà été calculée, et ne réalisera le calcul que dans le cas contraire, en ajoutant ensuite la nouvelle valeur calculée au dictionnaire.
Le calcul du coefficient binomial va alors prendre la forme qui suit :

```
1  binom_dict = {}
2
3  def binom(n, p):
4      if (n, p) not in binom_dict:
5          if p == 0 or n == p:
6              b = 1
7          else:
8              b = binom(n - 1, p - 1) + binom(n - 1, p)
9          binom_dict[(n, p)] = b
10     return binom_dict[(n, p)]
```

On peut observer que le programme récursif se retrouve presque mot pour mot lignes 5 à 8.
Calculons $\binom{5}{2}$ avec cette fonction, puis observons le contenu du dictionnaire :

```
In [1]: binom(5, 2)
Out[1]: 10

In [2]: binom_dict
Out[2]: {(3, 0): 1, (2, 0): 1, (1, 0): 1, (1, 1): 1, (2, 1): 2, (3, 1): 3, (4, 1): 4,
         (2, 2): 1, (3, 2): 3, (4, 2): 6, (5, 2): 10}
```

On peut constater qu'on y retrouve les 10 valeurs nécessaires pour réaliser ce calcul. J'ai représenté figure 6 l'ordre dans lequel ces valeurs ont été introduites dans le dictionnaire.

Remarque. La mémoïsation est tellement utile pour résoudre les problèmes de programmation dynamique que cette fonctionnalité est présente dans la librairie standard de Python : `lru_cache` est un décorateur de la

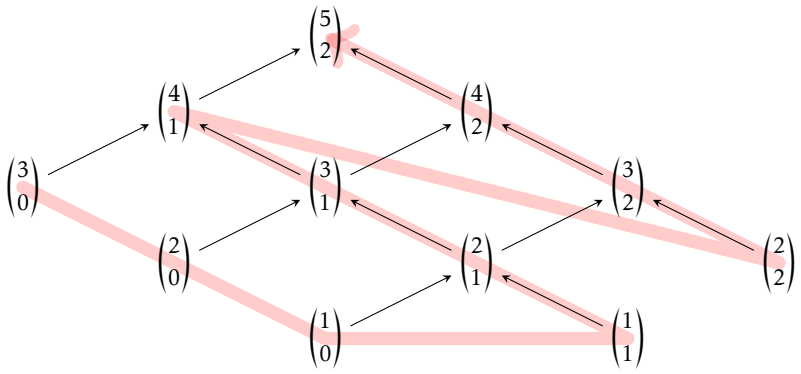


FIGURE 6 – Ordre d'entrée dans le dictionnaire.

bibliothèque `functools` qui associe automatiquement un dictionnaire à la définition d'une fonction récursive. Ainsi, pour ajouter à la fonction `binom` initiale un dictionnaire, il suffit d'écrire :

```
from functools import lru_cache

@lru_cache
def binom(n, p):
    if p == 0 or n == p:
        return 1
    return binom(n - 1, p - 1) + binom(n - 1, p)
```

et vous avez une fonction récursive efficiente !
Malheureusement, on peut légitimement supposer que cette solution simple ne sera pas acceptée aux concours, et il vous faudra sans doute définir explicitement un dictionnaire.

2.2 Programmation dynamique et gloutonne

Tout comme les problèmes que l'on résout par une méthode « diviser pour régner », les problèmes que l'on résout par la programmation dynamique se ramènent à la résolution de sous-problèmes de tailles inférieures. Mais à la différence de la méthode « diviser pour régner », ces sous-problèmes *ne sont pas indépendants*, ce qui impose d'accompagner la programmation récursive par une analyse fine des relations de dépendance, ou beaucoup plus simplement par l'utilisation de la mémoïsation qui gère les relations de dépendance à notre place.

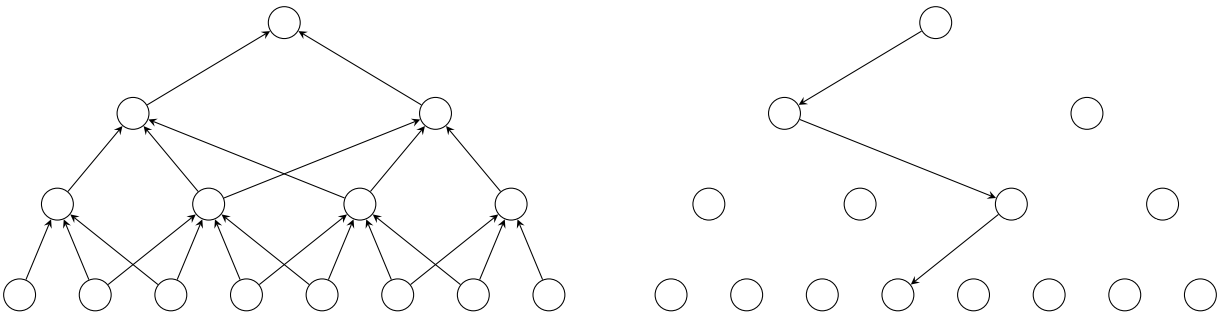


FIGURE 7 – Une illustration des programmations dynamique et gloutonne.

■ Problèmes d'optimisation

La programmation dynamique est fréquemment employée pour résoudre des problèmes d'optimisation : elle s'applique dès lors que la solution optimale peut être déduite des solutions optimales des sous-problèmes (c'est le *principe d'optimalité* de Bellman, du nom de son concepteur). Cette méthode garantit d'obtenir la meilleure solution au problème étudié, mais dans un certain nombre de cas sa complexité temporelle reste trop importante pour pouvoir être utilisée dans la pratique.
Dans ce type de situation, on se résout à utiliser un autre paradigme de programmation, la *programmation gloutonne*. Alors que la programmation dynamique se caractérise par la résolution par taille croissante de *tous* les problèmes locaux, la stratégie gloutonne consiste à choisir à partir du problème global un problème local *et un seul* en suivant une heuristique (c'est à dire une stratégie permettant de faire un choix rapide mais pas nécessairement optimal). On ne peut en général garantir que la stratégie gloutonne détermine la solution optimale, mais lorsque l'heuristique est bien choisie on peut espérer obtenir une solution proche de celle-ci.

Pour apprécier la différence entre programmation dynamique et programmation gloutonne, nous allons maintenant étudier le problème suivant :

■ Le problème du sac à dos

Il se pose dans les termes suivants :

> *étant donnés n objets de valeurs $c_1,\dots,c_n$ et de poids respectifs $w_1,\dots,w_n$, comment remplir un sac à dos maximisant la valeur emportée $\sum_{i\in I} c_i$ tout en respectant la contrainte $\sum_{i\in I} w_i \leqslant W_{max}$?*

Pour élaborer un algorithme glouton résolvant le problème, il faut définir une heuristique, ici un critère de priorité pour le choix des objets à prendre. Nous pouvons par exemple choisir en priorité les objets dont le rapport $\dfrac{\text{valeur}}{\text{poids}} = \dfrac{c_i}{w_i}$ est maximal, et remplir le sac tant que c'est possible :

```
def glouton(c, w, Wmax):
    r = sorted(
        [(c[k], w[k]) for k in range(len(c))], key=lambda t: t[0] / t[1], reverse=True
    )
    s, p = 0, Wmax
    for k in range(len(c)):
        if r[k][1] <= p:
            s += r[k][0]
            p -= r[k][1]
    return s
```

Pour évaluer la qualité de cette heuristique, j'ai réalisé 1 000 expériences pour chacune desquelles j'ai :

- pris au hasard 100 objets de valeurs comprises entre 1 et 20 et de poids compris entre 1 et 30 ;
- calculé la valeur optimale obtenue pour un poids maximal égal à 300 à la fois par l'algorithme glouton ci-dessus et par l'algorithme dynamique que nous étudierons plus loin.

Sur les 1 000 expériences, 421 ont donné le même résultat pour chacun des deux algorithmes, et dans le cas des 579 autres, l'algorithme glouton a toujours rendu un résultat au moins égal à 98% du résultat de l'algorithme dynamique.
On peut donc considérer ici qu'à défaut de donner un résultat toujours exact, l'algorithme glouton donne un résultat acceptable tout en ayant une complexité moindre que l'algorithme dynamique.

La solution dynamique
Pour résoudre ce problème, nous allons noter $f(k,W)$ la valeur maximale qu'il est possible d'atteindre avec les k premiers objets pour un poids total égal à W.
Si l'objet d'indice k est dans la solution optimale, alors $w_k \leqslant W$ et $f(k,W) = c_k + f(k-1, W-w_k)$; s'il n'y est pas alors $f(k,W) = f(k-1,W)$. On en déduit :

$$f(k,W) = \begin{cases} \max\big(c_k + f(k-1,W-w_k), f(k-1,W)\big) & \text{si } w_k \leqslant W \\ f(k-1,W) & \text{sinon} \end{cases}$$

Pour calculer cette valeur, nous allons utiliser un tableau bi-dimensionnel de taille $(n+1)\times(W_{max}+1)$ destiné à contenir les valeurs de $f(k,w)$ pour $k \in [\![0,n]\!]$ et $W \in [\![0,W_{max}]\!]$.
Nous prendrons comme valeurs initiales $f(0,W) = f(k,0) = 0$, et notre but est de calculer $f(n,W_{max})$.

Pour remplir ce tableau, il est primordial de respecter l'ordre de dépendance des cases de ce tableau : la case $f(k,W)$ ne peut être calculée que lorsque les cases $f(k-1,W)$ et $f(k-1,W-w_k)$ auront été remplies.

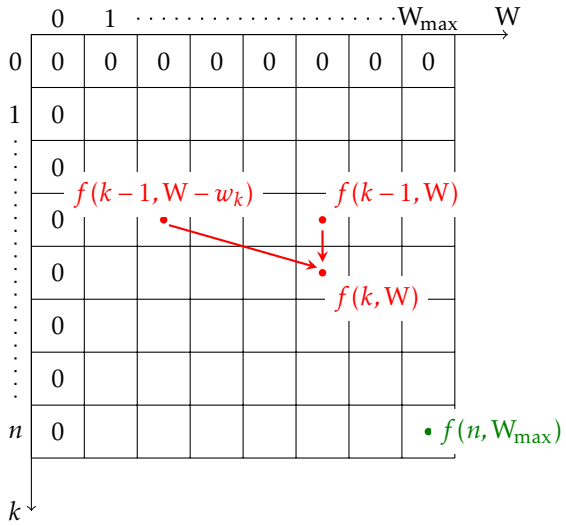


FIGURE 8 – Ordre de dépendance du sac à dos.

En considérant que les valeurs c_k et w_k sont données sous forme de tableaux, on en déduit l'algorithme :

```
def sacAdos(c, w, Wmax):
    n = len(c)
    f = np.zeros((n + 1, Wmax + 1), dtype=int)
    for k in range(n):
        for W in range(0, Wmax + 1):
            if w[k] <= W:
                f[k + 1, W] = max(c[k] + f[k, W - w[k]], f[k, W])
            else:
                f[k + 1, W] = f[k, W]
    return f[n, Wmax]
```

Il apparaît clairement que la complexité temporelle de cet algorithme est proportionnel au produit nW_{max}, soit en $O(nW_{max})$. L'algorithme glouton quant à lui est en $O(n\log n)$ (le coût du tri).

Remarque. Nous n'avons pas utilisé ici la technique de mémoïsation pour résoudre le problème. Cette dernière, lorsqu'elle est utilisée, nous permet de moins nous préoccuper de l'ordre de dépendance qui est géré par la récursivité :

```
def sacAdos(c, w, Wmax):
    dico = {}
    def f(k, W):
        if (k, W) not in dico:
            if k == 0 or W == 0:
                x = 0
            elif w[k - 1] <= W:
                x = max(c[k - 1] + f(k - 1, W - w[k - 1]), f(k - 1, W))
            else:
                x = f(k - 1, W)
            dico[(k, W)] = x
        return dico[(k, W)]
    return f(len(c), Wmax)
```

Remarque. Cet algorithme calcule la valeur maximale qui peut être emportée dans le sac, mais pas la façon d'y parvenir. Pour la connaître il faut utiliser le tableau (ou le dictionnaire) calculé par la fonction précédente, et retrouver le chemin qui mène de la case initiale à la case finale.
Par exemple, si on modifie la fonction non récursive (la première) pour qu'elle renvoie le tableau `f` qui a été calculé au lieu de la valeur `f[len(c), Wmax]`, le fonction qui détermine les objets à choisir s'écrira :

```
def objetsAchoisir(c, w, Wmax):
    f = sacAdos(c, w, Wmax)
    sac = []
    k, W = len(c), Wmax
    while k > 0:
        if f[k, W] > f[k - 1, W]:
            sac.append((c[k - 1], w[k - 1]))
            W -= w[k - 1]
        k -= 1
    return sac
```

2.3 Distance d'édition

La distance d'édition, ou distance de Levenshtein, est une mesure de la similarité de deux chaînes de caractères : elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne de caractères à une autre.
Par exemple, on peut passer du mot `polynomial` au mot `polygonal` en suivant les étapes suivantes :

- suppression de la lettre `'i'` : `polynomial` $\to$ `polynomal` ;
- remplacement du `'n'` par un `'g'` : `polynomal` $\to$ `polygomal` ;
- remplacement du `'m'` par un `'n'` : `polygomal` $\to$ `polygonal` ;

donc la distance d'édition entre ces deux mots est égale au plus à 3, et on se convaincra aisément qu'il n'est pas possible de faire mieux.
Nous allons calculer la distance d'édition entre deux mots $a = a_1 a_2 \cdots a_m$ et $b = b_1 b_2 \cdots b_n$ en généralisant le problème, c'est à dire en définissant la distance d'édition $\mathrm{d}(i,j)$ entre les mots $a_1 a_2 \cdots a_i$ et $b_1 b_2 \cdots b_j$.
Dans le chemin reliant de manière optimale $a_1 a_2 \cdots a_i$ et $b_1 b_2 \cdots b_j$, plusieurs cas de figure peuvent se rencontrer :

- a_i a été supprimé, auquel cas $\mathrm{d}(i,j) = \mathrm{d}(i-1,j)+1$;
- b_j a été ajouté, auquel cas $\mathrm{d}(i,j) = \mathrm{d}(i,j-1)+1$;
- a_i a été remplacé par b_j, auquel cas $\mathrm{d}(i,j) = \mathrm{d}(i-1,j-1)+1$;
- $a_i = b_j$, auquel cas $\mathrm{d}(i,j) = \mathrm{d}(i-1,j-1)$.

On en déduit que $\mathrm{d}(i,j) = \begin{cases} \min\big(\mathrm{d}(i-1,j), \mathrm{d}(i,j-1), \mathrm{d}(1-1,j-1)\big)+1 & \text{si } a_i \neq b_j \\ \min\big(\mathrm{d}(i-1,j)+1, \mathrm{d}(i,j-1)+1, \mathrm{d}(i-1,j-1)\big) & \text{si } a_i = b_j \end{cases}$.
Les conditions initiales sont clairement : $\mathrm{d}(i,0) = i$ et $\mathrm{d}(0,j) = j$, ce qui conduit au schema de dépendance représenté figure 9. puis à l'algorithme :

```
def dist(a, b):
    m, n = len(a), len(b)
    d = np.zeros((m + 1, n + 1), dtype=int)
    for i in range(1, m + 1):
        d[i, 0] = i
    for j in range(1, n + 1):
        d[0, j] = j
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if a[i - 1] == b[j - 1]:
                d[i, j] = min(d[i - 1, j] + 1, d[i, j - 1] + 1, d[i - 1, j - 1])
            else:
                d[i, j] = min(d[i - 1, j] + 1, d[i, j - 1] + 1, d[i - 1, j - 1] + 1)
    return d[m, n]
```

Cette fonction a une complexité temporelle et spatiale en $\mathrm{O}(mn)$.

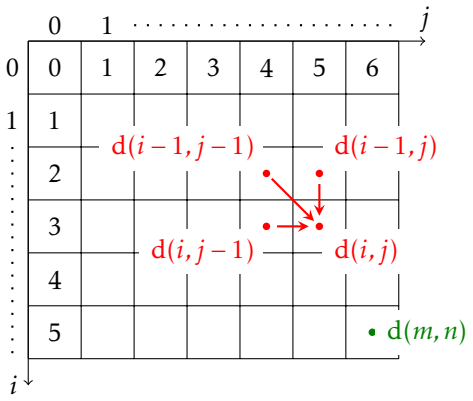
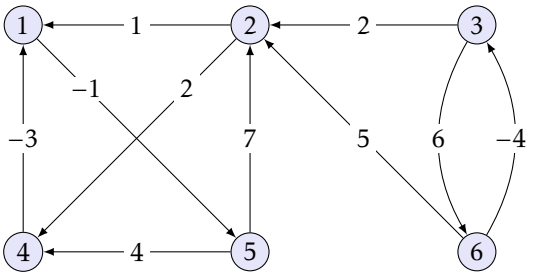


FIGURE 9 – Schéma de dépendance pour la distance de Levenshtein.

2.4 Algorithme de Floyd-Warshall

Le dernier algorithme de programmation dynamique que nous allons étudier concerne les graphes.
Nous allons considérer ici un graphe orienté et pondéré : autrement dit, les arêtes sont orientées, et chaque arête possède un poids, ici un entier relatif.
Une façon de représenter en machine un tel graphe consiste à numéroter les sommets et à considérer une matrice $M \in \mathcal{M}_n(\mathbb{Z})$, pour laquelle le coefficient $m_{i,j}$ sera égal au poids de l'arête reliant les sommets v_i à v_j, avec la convention que ce poids est égal à $+\infty$ s'il n'y a pas d'arête reliant v_i à v_j (illustration figure 10.)



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

FIGURE 10 – Un exemple de graphe pondéré et de sa matrice d'adjacence.

L'algorithme de FLOYD-WARSHALL a pour objet de calculer, pour chacun des couples de points (v_i, v_j), le chemin de poids minimal reliant v_i à v_j, s'il existe. Pour ce faire, cet algorithme calcule la suite finie de matrices $M^{(k)}$, $0 \leqslant k \leqslant n$ avec $M^{(0)} = M$ et :

$$\forall k < n, \quad \forall (i,j) \in [\![1,n]\!]^2, \qquad m_{ij}^{(k+1)} = \min\left(m_{ij}^{(k)}, m_{i,k+1}^{(k)} + m_{k+1,j}^{(k)}\right).$$

THÉORÈME 2.1 — *Si G ne contient pas de cycle de poids strictement négatif, alors $m_{ij}^{(k)}$ est égal au poids du chemin minimal reliant v_i à v_j et ne s'autorisant de passer que par des sommets parmi $v_1, v_2, \ldots, v_k$.*
De ceci il découle immédiatement que $m_{ij}^{(n)}$ est le poids minimal d'un chemin reliant v_i à v_j.

Raisonnons par récurrence sur k.

– Si $k = 0$, $m_{ij}^{(0)} = m_{ij}$ est bien entendu le poids du chemin minimal reliant v_i à v_j sans passer par aucun autre sommet.

– Si $k < n$, supposons le résultat acquis au rang k et considérons un chemin $v_i \rightsquigarrow v_j$ ne passant que par les sommets $v_1, \ldots, v_{k+1}$ et de poids minimal.

Si ce chemin ne passe pas par v_{k+1}, son poids total est par hypothèse de récurrence égal à $m_{ij}^{(k)}$.

Si ce chemin passe par v_{k+1}, alors par principe de sous-optimalité les chemins $v_i \rightsquigarrow v_{k+1}$ et $v_{k+1} \rightsquigarrow v_j$ sont minimaux et ne passent que par des sommets de la liste $v_1,\dots,v_k$ donc par hypothèse de récurrence son poids total est égal à $m^{(k)}_{i,k+1} + m^{(k)}_{k+1,j}$ (car il n'existe pas de cycle de poids négatif reliant v_{k+1} à lui-même).

De ceci il résulte que $\min\left(m^{(k)}_{ij}, m^{(k)}_{i,k+1} + m^{(k)}_{k+1,j}\right) = m^{(k+1)}_{ij}$ est le poids minimal d'un plus court chemin reliant v_i et v_j et ne passant que par des sommets de la liste $v_1,\dots,v_{k+1}$.

■ Mise en œuvre pratique

Il est possible de calculer les différentes valeurs de la suite $(\mathrm{M}^{(k)})$ en utilisant une seule matrice car la formule :

$$m^{(k+1)}_{ij} = \min\left(m^{(k)}_{ij}, m^{(k)}_{i,k+1} + m^{(k)}_{k+1,j}\right)$$

reste valable si on remplace $m^{(k)}_{i,k+1}$ par $m^{(k+1)}_{i,k+1}$ ou $m^{(k)}_{k+1,j}$ par $m^{(k+1)}_{k+1,j}$. En effet, en l'absence de cycles de poids négatif on a $m^{(k)}_{i,k+1} = m^{(k+1)}_{i,k+1}$ et $m^{(k)}_{k+1,j} = m^{(k+1)}_{k+1,j}$ donc l'ordre dans lequel sont modifiées les cases d'indices (i,j), $(i,k+1)$ et $(k+1,j)$ n'a pas d'importance.

```
def floydwarshall(M):
    n = M.shape[0]
    N = M.copy()
    for k in range(n):
        for i in range(n):
            for j in range(n):
                N[i, j] = min(N[i, j], N[i, k] + N[k, j])
    return N
```

Appliqué au graphe pondéré donné figure 10, cet algorithme retourne la matrice :

$$\mathrm{M}^{(6)} = \begin{pmatrix} 0 & 6 & +\infty & 3 & -1 & +\infty \\ -1 & 0 & +\infty & 2 & -2 & +\infty \\ 1 & 2 & 0 & 4 & 0 & 6 \\ -3 & 3 & +\infty & 0 & -4 & +\infty \\ 1 & 7 & +\infty & 4 & 0 & +\infty \\ -3 & -2 & -4 & 0 & -4 & 0 \end{pmatrix}$$

On observe que les sommets 3 et 6 ne sont pas accessibles à partir des sommets 1, 2, 4 et 5. En revanche, on peut aller du sommet 6 au sommet 1 pour un coût total égal à -3 (il n'est pas difficile de deviner qu'il faut passer par les sommets 3, 2 et 4) ou du sommet 3 au sommet 5 pour un coût total nul (en passant par les sommets 2, 4 et 1).

Remarque. De manière évidente la complexité temporelle de l'algorithme de Floyd-Warshall est en $\mathrm{O}(n^3)$, où n est l'ordre du graphe et la complexité spatiale en $\mathrm{O}(n^2)$.

■ Application au calcul de la fermeture transitive d'un graphe

Considérons de nouveau un graphe non pondéré, orienté ou non. Le problème de la *fermeture transitive* consiste à déterminer si deux sommets a et b peuvent être reliés par un chemin allant de a à b. Pour le résoudre, nous allons utiliser la matrice d'adjacence associée à ce graphe, mais cette fois en utilisant les valeurs booléennes `True` pour dénoter l'existence d'une arête et `False` pour en marquer l'absence. Remplaçons maintenant dans l'algorithme de Floyd-Warshall la relation de récurrence sur les coefficients des matrices $\mathrm{M}^{(k)}$ par :

$$m^{(k+1)}_{ij} = m^{(k)}_{ij} \text{ ou } \left(m^{(k)}_{i,k+1} \text{ et } m^{(k)}_{k+1,j}\right).$$

Il n'est pas difficile de prouver que le booléen $m^{(k)}_{ij}$ dénote l'existence d'un chemin reliant les sommets v_i et v_j en ne passant que par les sommets $v_1, v_2, \dots, v_k$ et que par voie de conséquence la matrice $\mathrm{M}^{(n)}$ résout le problème de la fermeture transitive.

L'algorithme ainsi modifié est connu sous le nom d'algorithme de Warshall :

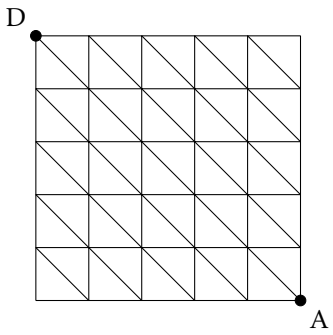
```python
def warshall(M):
    n = M.shape[0]
    N = M.copy()
    for k in range(n):
        for i in range(n):
            for j in range(n):
                N[i, j] = N[i, j] or N[i, k] and N[k, j]
    return N
```

3. Exercices

Exercice 3

Partant du coin supérieur gauche d'une grille $n \times n$, on souhaite calculer le nombre de chemins menant au coin inférieur droit en ne suivant que les trois directions suivantes :

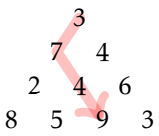
les trois directions possibles



Rédiger une fonction `chemins(n)` qui répond à la question.

Exercice 4

En partant du sommet du triangle ci-dessous et en se déplaçant vers les nombres adjacents de la ligne inférieure, le total maximum que l'on peut obtenir pour relier le sommet à la base est égal à 23 :



$$3 + 7 + 4 + 9 = 23$$

Rédiger une fonction calculant le total maximum d'un chemin reliant le sommet à la base d'un tel triangle de hauteur n. On pourra considérer que les valeurs de ce triangle sont stockées dans un tableau bi-dimensionnel $n \times n$ (autrement dit, `t[i, j]` contient la $(j+1)^{\text{e}}$ valeur de la $(i+1)^{\text{e}}$ ligne).

Exercice 5

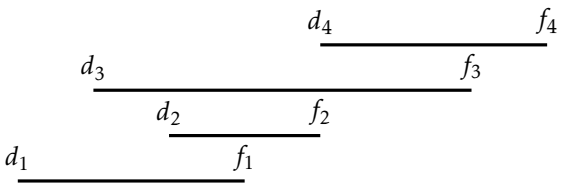
Étant donné une chaîne de caractères $a = a_1 a_2 \cdots a_m$, on appelle *sous-chaîne de longueur k* toute chaîne de caractère $a_{i_1} a_{i_2} \cdots a_{i_k}$ avec $1 \leqslant i_1 \leqslant i_2 \leqslant \cdots \leqslant i_k \leqslant m$.

a. Rédiger une fonction `pgscc(a, b)` qui, étant donné deux chaînes de caractères $a = a_1 \cdots a_m$ et $b = b_1 \cdots b_n$, renvoie la longueur de la plus grande sous-chaîne commune à a et b.

b. Modifier votre algorithme pour qu'il renvoie cette fois une sous-chaîne commune de longueur maximale.

Exercice 6 [Ordonnancement d'intervalles pondérés]

On considère une succession d'intervalles de temps $[d_i, f_i]$, $1 \leqslant i \leqslant n$, chacun d'eux étant associé à une valeur v_i. On dit que deux intervalles $[d_i, f_i]$ et $[d_j, f_j]$ *ne se chevauchent pas* lorsque $f_i \leqslant d_j$ ou $f_j \leqslant d_i$. Rédiger une fonction `ordonnancement(d, f, v)` qui détermine la somme maximale des valeurs d'une sous-famille d'intervalles ne se chevauchant pas. On supposera $f_1 \leqslant f_2 \leqslant \cdots \leqslant f_n$.



Exercice 7

Si A et B sont deux matrices de tailles respectives $a \times b$ et $c \times d$, le produit AB n'est possible que si $b = c$, et dans ce cas la matrice produit AB est de taille $a \times d$, et peut être calculée à l'aide de acd multiplications.

Sachant que le produit matriciel est associatif mais pas commutatif, le produit ABC peut être calculé de deux manières : (AB)C ou A(BC), qui ne nécessitent pas *a priori* le même nombre de multiplications. Par exemple, si A est de taille 10×100, B de taille 100×5, et C de taille 5×50, le produit (AB)C nécessite $5000 + 2500 = 7500$ multiplications et le produit A(BC), $25000 + 50000 = 75000$ multiplications.

On considère une chaîne de matrices $\mathrm{M}_0\mathrm{M}_1\mathrm{M}_2\cdots\mathrm{M}_{n-1}$ de tailles respectives $m_0 \times m_1$, $m_1 \times m_2$, …, $m_{n-1} \times m_n$. Rédiger une fonction calculant le nombre minimal de multiplications nécessaires pour effectuer ce produit matriciel. On pourra considérer que les valeurs de $m_0, m_1, \ldots, m_n$ sont rangées dans un tableau de taille $n+1$.