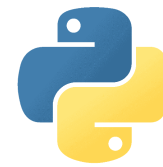


7 Programmation dynamique

La *programmation dynamique* est une méthode algorithmique qui, pour résoudre un problème, le décompose en sous-problèmes qui sont résolus récursivement, et dont les solutions sont combinées pour donner la solution du problème initial.

Cette méthode peut s'appliquer même lorsque les sous-problèmes se recoupent, c'est-à-dire lorsque les sous-problèmes présentent des cas (ou sous-sous-problèmes) communs. Afin d'éviter de résoudre plusieurs fois le même cas, un algorithme de programmation dynamique utilisera alors un tableau ou un dictionnaire (on en a d'ailleurs déjà vu quelques exemples dans le **Cours n°3**) pour enregistrer les solutions des sous-problèmes déjà trouvées (on parle de *mémoïsation*).

La programmation dynamique s'applique le plus souvent à des problèmes d'optimisation. Elle est à rapprocher des méthodes *gloutonnes*, mais ces dernières sont plus rudimentaires car elles ne permettant pas de gérer les recoupements de sous-cas.

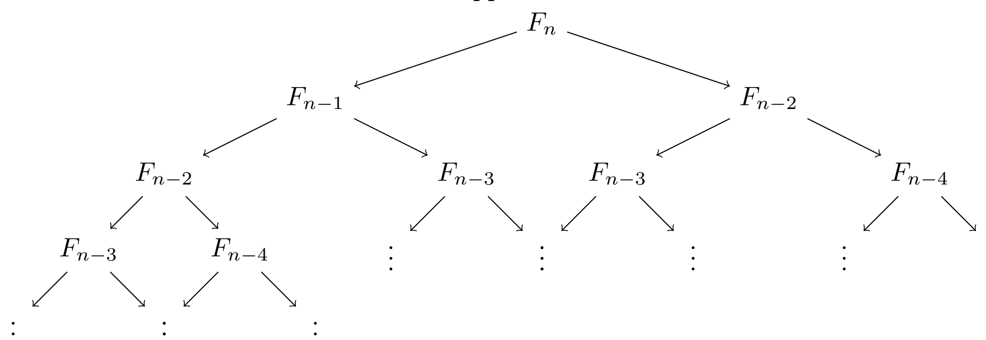
1 Les concepts de la programmation dynamique à travers l'exemple de la suite de Fibonacci

Algorithme naïf et problème de redondance des appels récursifs

On rappelle que *suite de Fibonacci* est la suite $(F_n)_{n\in\mathbb{N}}$ définie par $F_0 = 0$, $F_1 = 1$ et pour tout n entier : $F_{n+2} = F_{n+1} + F_n$. Le code récursif suivant permet de calculer F_n .

```
def Fibonacci(n):
    if n==0:
        return 0
    if n==1:
        return 1
    else:
        return Fibonacci(n-1)+Fibonacci(n-2)
```

Figure 1.1. L'arbre des appels récursifs de `Fibonacci(n)`

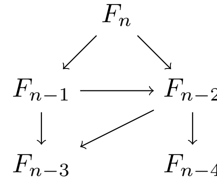


Mais cette fonction offre de piètres performances. En effet, l'arbre des appels récursifs de cette fonction montre que, pour évaluer F_n, elle calcule F_{n-1} une fois, F_{n-2} deux fois, F_{n-3} trois fois, F_{n-4} cinq fois. . . On obtient ainsi une complexité

exponentielle (en fait géométrique) ce qui est rédhibitoire pour ce type de problème.

On peut également tracer le graphe des sous-problèmes associé à cette fonction, qui illustre d'une autre façon que le recoupement des sous-problèmes engendre des répétitions dans les calculs.

FIGURE 1.2. Le graphe des sous-problèmes de `Fibonacci(n)`



Le fait que le graphe ci-dessus possède un cycle (ce n'est donc pas un arbre) exprime le fait que les sous-problèmes se recoupent.

Algorithme ascendant

Une première façon d'optimiser ce code est d'utiliser une *approche ascendante*. Il s'agit de commencer par calculer les solutions des sous-problèmes les plus petits (ici calculer F_0 et F_1), puis de remonter étape par étape jusqu'à arriver au problème initial (ici F_n), en **mémorisant** à chaque étape les valeurs intermédiaires dans une liste, afin de ne pas recalculer celles-ci. On obtient alors un code itératif.

```
def Fibonacci_ASC(n):
    F=[]
    F.append(0)
    F.append(1)
    for k in range(2,n+1):
        F.append(F[k-2]+F[k-1])
    return F[n]
```

☞ Avec ce code il n'y a aucun avantage à utiliser un dictionnaire plutôt qu'une liste.

Algorithme descendant et mémoïsation

Une seconde façon d'optimiser ce code est d'utiliser une *approche descendante* (en anglais : *top-down*).
On reprend alors le cheminement du code initial, mais en mémorisant à chaque étape les valeurs intermédiaires dans un dictionnaire, afin de ne pas recalculer celles-ci : on parle de **mémoïsation**.
On obtient alors un code récursif, davantage dans l'esprit de la programmation dynamique.

```
def Fibonacci_DSC(n):
    global F  #Le dictionnaire doit être défini comme une variable globale
    F={}      #et avant d'appeler Fibo
    return Fibo(n)

def Fibo(n):
    if not n in F :
        if n==0 or n==1:
            F[n]=n
        else :
            F[n]=Fibo(n-1)+Fibo(n-2)
    return F[n]
```

2 Un autre exemple : chemin de poids maximal dans une matrice

On considère une matrice d'entiers naturels de n lignes et p colonnes dont les lignes sont numérotées de 0 à $n-1$ et les colonnes sont numérotées de 0 à $p-1$.
La case en haut à gauche est indexée par $(0,0)$ et la case en bas à droite est indexée par $(n-1,p-1)$.

- ✗ Un chemin est une succession de cases allant de la case $(0,0)$ à la case $(n-1,p-1)$, en n'autorisant que des déplacements case par case : soit vers la droite, soit vers le bas.

- ✗ Le poids d'un chemin est la somme des entiers situés sur ce chemin. Par exemple, pour la matrice M suivante,

$$M = \begin{pmatrix} \mathbf{4} & \mathbf{1} & \mathbf{1} & 3 \\ 2 & 0 & \mathbf{2} & 1 \\ 3 & 1 & \mathbf{5} & \mathbf{1} \end{pmatrix}$$

  Un chemin dans M est : $(0,0),(0,1),(0,2),(1,2),(2,2),(2,3)$ (en vert dans M) et son poids est 14.

Problème. Déterminer un chemin de poids maximal ainsi que son poids, qu'on appellera un chemin optimal, parmi tous les chemins allant de la case $(0,0)$ à la case $(n-1,p-1)$.

Recherche exhaustive d'une solution

On pourrait essayer d'examiner tous les chemins possibles, car ils sont en nombre fini. Dénombrons ces chemins.
Pour construire un tel chemin, il suffit de savoir où placer les $n-1$ déplacements $\downarrow$ parmi les $n-1+m-1$ déplacements totaux. Ainsi, il y a $N_{n,m} = \binom{n+m-2}{n-1}$ chemins possibles.
Utilisons la formule de Stirling pour calculer un équivalent asymptotique de cette quantité lorsque $n=m$.

$$\begin{aligned} N_{n,n} = \binom{2n-2}{n-1} &= \frac{(2n-2)!}{(n-1)!^2} \\ &\underset{n\to\infty}{\sim} \frac{(2n-2)^{2n-2}e^{-2(n-1)}\sqrt{2\pi(2n-2)}}{2(n-1)^{2(n-1)}e^{-2(n-1)}\pi(n-1)} \\ &= \frac{2^{2n-2}}{\sqrt{\pi(n-1)}} \underset{n\to\infty}{\sim} \frac{2^{2n-2}}{\sqrt{\pi n}} \end{aligned}$$

Le nombre de chemins possibles est donc exponentiel en n. Pour $n \geq 30$, un algorithme qui explore tous les chemins est inutilisable en pratique.

Sous-problèmes

Considérons une solution à notre problème, c'est-à-dire une chemin c dans M de la case $(0,0)$ à la case $(n-1,m-1)$ dont le poids est maximal.

Supposons que ce chemin passe par la case (i,j). Le chemin se décompose en deux morceaux

$$(0,0) \overset{c_1}{\rightsquigarrow} (i,j) \overset{c_2}{\rightsquigarrow} (n-1,m-1)$$

où c_1 et c_2 sont des chemins de poids maximal de sous-matrices de M (comme on se restreint aux déplacements $\rightarrow$ et $\downarrow$, c_1 et c_2 ne peuvent pas sortir de la sous-matrice en question).

En effet, raisonnons par l'absurde ; supposons que c_1 n'est pas optimal. Il existe donc un chemin c_1' de poids strictement supérieur de la case $(0,0)$ à la case (i,j). Mais alors, le chemin c' suivant

$$(0,0) \overset{c_1'}{\rightsquigarrow} (i,j) \overset{c_2}{\rightsquigarrow} (n-1,m-1)$$

est un chemin de poids strictement supérieur à c : impossible. De même pour c_2.

Le problème d'optimisation d'un chemin de la case $(0,0)$ à une case (i,j) peut être qualifié de **sous-problème** au problème initial, car la matrice à considérer est plus petite.

Une solution au problème initial (sur la matrice $n \times m$) donne donc une solution à de multiples sous-problèmes : **c'est une caractéristique qui indique que la programmation dynamique peut être utilisée pour résoudre ce problème.**

Remarque 1. **Principe d'optimalité de Bellman**

De façon plus générale, la programmation dynamique est utilisée pour résoudre des problèmes d'optimisation dès lors qu'une solution optimale peut être déduite de solutions optimales de sous-problèmes (c'est le *principe d'optimalité de Bellman*, sur lequel s'appuie donc la programmation dynamique) et que ces sous-problèmes se chevauchent.

Une approche récursive pour construire une solution.

Une approche récursive possible, pour trouver **le poids d'un chemin maximal**, est fondée sur les observations suivantes :

- ✗ Si $n = p = 1$, alors il y a un chemin, il est maximal et son poids est $M_{0,0}$;
- ✗ Si $n = 1$ et $p \geq 2$, alors il suffit de calculer le poids du chemin allant de $M_{0,0}$ à $M_{0,p-2}$ puis lui ajouter $M_{0,p-1}$ (même principe si $n \geq 2$ et $p = 1$);
- ✗ sinon, il suffit de calculer le poids d'un chemin de poids maximal allant de
  - de $M_{0,0}$ à $M_{n-1,p-2}$ (chemin arrivant par la gauche) ;
  - de $M_{0,0}$ à $M_{n-2,p-1}$ (chemin arrivant par le haut) ;

  puis de prendre le plus grand des deux et lui ajouter $M_{n-1,p-1}$.

Cet algorithme a une complexité temporelle dégradée; par exemple pour $n = 4$ et $p = 3$, on calcule récursivement le poids des chemins optimaux arrivant en $M_{4,2}$ et en $M_{3,3}$.

i. Pour calculer le poids d'un chemin optimal arrivant en $M_{4,2}$, on calcule récursivement les poids des chemins optimaux arrivant en $M_{4,1}$ et en $M_{3,2}$;

ii. Pour calculer le poids d'un chemin optimal arrivant en $M_{3,3}$, on calcule récursivement les poids des chemins optimaux arrivant en $M_{3,2}$ et en $M_{2,3}$.

On voit dès lors que l'on fait plusieurs fois un même calcul et on peut montrer que cette approche conduit à un algorithme de complexité exponentielle (lorsque $n = p$).

Une solution utilisant la programmation dynamique avec calcul de bas en haut.

L'approche précédente suggère l'utilisation d'un tableau W de poids, de même taille que la matrice M et dont l'élément Wi, j est calculé de façon itérative en utilisant la relation de récurrence suivante :

$$W_{0,0} = M_{0,0} \qquad W_{i,j} = M_{i,j} + \begin{cases} W_{i,j-1} & \text{si } i = 0 \text{ et } j \geq 1 \\ W_{i-1,j} & \text{si } i \geq 1 \text{ et } j = 0 \\ \max\left(\{W_{i,j-1}, W_{i-1,j}\}\right) & \text{sinon} \end{cases}$$

La matrice W mémorise les chemins optimaux successifs.
La valeur du poids d'un chemin optimal est alors $W_{n-1,p-1}$.

```
def poids_optimal(M):
        n = len(M)
        p = len(M[0])
        W = [[0]*p]*n # on pré-remplit une matrice de 0
        W[0][0] = M[0][0]
        for i in range(1,n):
                W[i][0] = W[i-1][0]+M[i][0]
        for j in range(1,p):
                W[0][j] = W[0][j-1]+M[0][j]
        for i in range(1, n):
                for j in range(1, p):
                        W[i][j] = M[i][j] + max(W[i-1][j],W[i][j-1])
        return W[n-1][p-1]
```

La complexité de notre fonction `W` est en $O(np)$.

Construction d'un chemin optimal.

Pour construire un chemin optimal, il suffit de remonter de la case $(n-1,p-1)$ la case $(0,0)$ en observant la règle suivante: en remontant depuis une case (i,j), on choisit parmi les cases $(i-1,j)$ et $(i,j-1)$ celle ayant un poids maximal.

```
def max_chemin(M):
        n = len(M)
        p = len(M[0])
        W = poids_optimal(M)
        L = [ ]
        i, j = n-1, p-1
        while i > 0 and j > 0:
                if W[i-1][j] > W[i][j-1]:
                        L.append('Sud')
                        i -= 1
                else:
                        L.append('Est')
                        j -= 1
        while i > 0:
                L.append('Sud')
                i -= 1
        while j > 0:
                L.append('Est')
                j -= 1
        return L[::-1], W[n-1][p-1] # inversion de l'ordre des termes de la liste
```

Programmation dynamique : ce qu'il faut retenir.

La programmation dynamique intervient dans un problème d'optimisation lorsqu'il est possible de décomposer celui-ci en sous-problèmes (on dit que ce problème possède la propriété de sous-structure optimale) : on peut déduire une solution optimale du dit problème en combinant des solutions optimales d'une famille de sous-problèmes qui, en général se chevauchent.

Problème. On se donne donc une fonction $f:\mathcal{U}\to\mathbb{Z}$, et on cherche à déterminer x tel que $f(x)=\max_{y\in\mathcal{U}}\{f(y)\}$ (la démarche est la même si on cherche à minimiser f sur $\mathcal{U}$).
Dans la suite, on notera $M_{f,\mathcal{U}}=\max_{y\in\mathcal{U}}\{f(y)\}$.
Démarche. Il y a 4 étapes dans la résolution d'un tel problème par programmation dynamique.
On les présente ci-dessous en les illustrant par un retour sur l'exemple traité ci-avant.

Étape 1. **Identifier une sous-structure optimale**

C'est un indice que la programmation dynamique peut être appliquée.

Il s'agit de voir que si l'ont connaît $x\in\mathcal{U}$ tel que $f(x)=M_{f,\mathcal{U}}$, alors de x on déduit des solutions $(x_i)_{i\in I}$ à des sous-problèmes de la forme "trouver $x_i\in\mathcal{U}_i$ tel que $f_i(x_i)=M_{f_i,\mathcal{U}_i}$ ".

Les sous-problèmes doivent être plus simples à résoudre que le problème initial.

Exemple.
Dans l'exemple précédent, les sous-problèmes concernaient des matrices plus petites.

Étape 2. **Déduire une relation de récurrence**

Déduire de la sous-structure optimale une relation récursive permettant de calculer $M_{f,\mathcal{U}}$ à partir de certains $M_{f_i,\mathcal{U}_i}$.

Exemple.
Dans l'exemple précédent, on a exhibé une relation entre $W_{n-1,p-1}, W_{n-2,p-1}$ et $W_{n-1,p-2}$.

Étape 3. **Mémoïsation**

On calcule les $M_{f_i,\mathcal{U}_i}$ utiles itérativement, en faisant usage d'un tableau pour stocker tous ces éléments.
On peut parfois se contenter de n'en stocker que certains.

Exemple.

Dans l'exemple précédent, on a écrit un code pour calculer et stocker les coefficients de la matrice W.
On *pourrait* ne stocker qu'une ligne de la matrice W, ce qui mènerait à une complexité spatiale en $O(n)$ au lieu de $O(n \times p)$.

Étape 4. **Conclusion : solution au problème initial**

Enfin, on modifie légèrement le calcul des $M_{f_i,\mathcal{U}_i}$ pour obtenir en même temps $\forall i \in I$ un x_i satisfaisant $f_i(x_i) = M_{f_i,\mathcal{U}_i}$.
En général, cette étape n'est pas difficile.

Exemple.

Dans l'exemple précédent, on a choisi de ne calculer un chemin optimal qu'après le calcul des $W_{i,j}$, mais on aurait pu par exemple stocker en parallèle des $W_{i,j}$ (dans une autre matrice) le dernier déplacement à effectuer pour arriver en (i,j) en suivant un chemin optimal.

Parmi les problèmes classiques qui peuvent être traités par programmation dynamique, on peut citer :

- ✗ le problème du sac à dos ;
- ✗ le calcul des plus longues sous-chaînes communes à deux chaînes de caractères ;
- ✗ l'algorithme de Floyd-Warshall de recherche des plus courts chemins dans un graphe pondéré ;

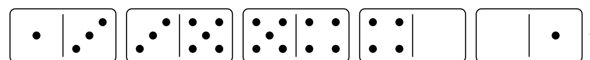
3 Exercice 1 : sous-séquence de dominos

Un domino D est une pièce rectangulaire contenant deux chiffres, de 0 à N, matérialisés par des points.
Par exemple,

sont deux dominos, représentant les paires $(3,5)$ et $(4,0)$. Un domino est donc représenté par une paire d'entiers $(i,j), i,j \in [\![0,N]\!]$.
On dispose d'un sac $\mathcal{S}$ contenant n dominos $D_1, D_2, ..., D_n$. Pour tout $k \in [\![1,n]\!]$, $D_k = (i_k, j_k)$ où $i_k, j_k \in [\![0,N]\!]$.
Un sac de dominos est donc implémenté en `Python` par une liste de tuples.
Une **chaîne** de dominos est une séquence de pièces telle que les chiffres voisins sur chaque paire de dominos consécutifs coïncident.
Par exemple, pour le sac

$$\mathcal{S} = \left(\quad , \quad , \quad , \quad , \quad \right),$$

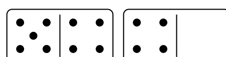
la séquence suivante est une chaîne de 5 dominos :



On s'intéresse alors à la recherche de la plus longue sous-séquence de n dominos d'un sac $\mathcal{S}$ qui forme une chaîne.
Les dominos sont supposés ordonnés et ne peuvent être permutés. De plus, $\mathcal{S}$ ne contient pas nécessairement tous les dominos possibles.
Par exemple, pour le sac

$$\mathcal{S} = \left(\quad , \quad , \quad , \quad , \quad \right),$$

la sous-séquence



est de taille maximale, égale à 2.

On aborde ce problème sous l'angle de la programmation dynamique.
Soit $\ell : \mathbb{N}^* \to \mathbb{N}^*$ la fonction qui associe à $k \in \mathbb{N}^*$ la longueur de la plus longue chaîne sous-séquence de $D_1 \cdots D_k$, se terminant par le domino D_k.

1. Donner la valeur de $\ell(1)$ et une formule de récurrence sur k pour la fonction ℓ en détaillant la réponse.

Solution. On observe qu'ici l'énoncé, dès sa première question, fusionne les étapes 1 et 2 présentées ci-avant.
Il est clair que si $k = 1$, la plus longue sous-séquence commençant et se terminant par D_1 est D_1 de longueur 1 donc $\ell(1) = 1$.
Si $k > 1$, la sous-séquence qui nous intéresse est la plus longue parmi les sous-séquence suivantes

- ✗ la sous-séquence réduite au dé D_k ;
- ✗ pour $m < k$, la plus longue chaîne sous-séquence de $D_1 ... D_m$ se terminant par D_m à laquelle on rajoute D_k si D_m est *compatible* avec D_k.

Ceci donne la formule

$$\ell(k) = \begin{cases} 1, & \text{si } k = 1 \\ \max\left(\{1\} \cup \{1 + \ell(m) : m \in [\![1, k-1]\!],\ D_m \text{ et } D_k \text{ compatibles}\}\right), & \text{si } k > 1 \end{cases}$$

□

2. En l'écriture, en `Python`, d'une fonction `longueur_max` qui renvoie la liste de toutes les valeurs de la fonction ℓ sur $[\![1, n]\!]$ en prenant en argument une liste `D`, représentant le sac de dominos sous forme d'une une liste `D` de tuples à deux éléments
On pourra commencer par écrire une fonction `compatible(D1, D2)` qui renvoie `True` ou `False` selon que les dominos D_1 et D_2 forment une chaîne (dans cet ordre) ou non.

Solution. Cette question correspond à l'étape 3 présentée ci-avant.
On commence par écrire une fonction de compatibilité entre des dominos.

```
def compatible(D1, D2) :
     return D1[1]==D2[0]
```

Ensuite, on utilise la formule de la question ci-dessus pour calculer successivement $\ell(1), \ell(2), \ell(3), ..., \ell(n)$, stockés comme `L[0], L[1], ..., L[n-1]`.

```
def longueur_max(D) :
    n=len(D)
    L=[0]*n # on pré-remplit avec des 0
    L[0]=1 # valeur de l(1)
    for i in range(1, n) :
        meilleur = 1
        for j in range(i):
            if compatible(D[j], D[i]) and L[j]+1 > meilleur :
                meilleur=L[j]+1
        L[i]=meilleur
    return L
```

□

3. Écrire une fonction permettant d'obtenir une chaîne sous-séquence la plus longue parmi $D_1, D_2, \ldots, D_n$.

Solution. Cette question correspond à l'étape 4 décrite plus haut. Une plus longue chaîne sous-séquence se termine par le domino D_k tel que $\ell(k)$ est maximale. Il faut donc récupérer la valeur de cet indice (et la valeur de ce maximum pour savoir quelle sera la longueur de la chaîne). Pour compléter la chaîne, on répète le processus dans la sous-liste des dominos qui précèdent le dernier ajouté : on récupère l'indice j du domino compatible tel que $\ell(j)$ est maximal, etc...

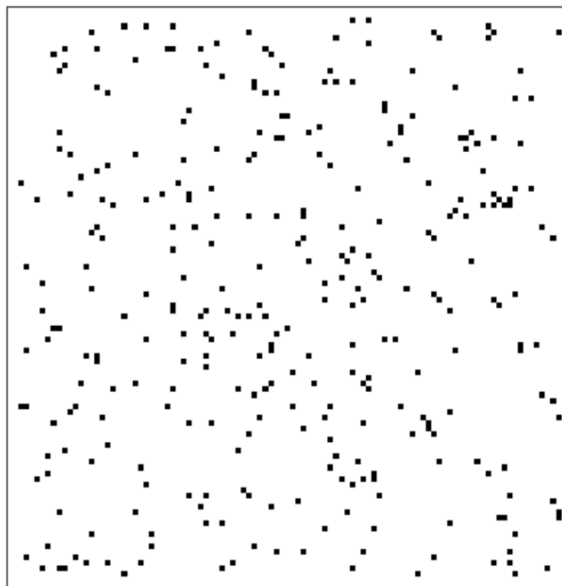
```
def valeurs_max(D) :
    n, L =len(D), longueur_max(D)
    k, m =0, L[0]
    for i in range(1, n):
        if L[i] > m :
            m,k = L[i], i
    return m,k
```

```
def max_chaine_dominos(D) :
    n=len(D)
    M,k=valeurs_max(D)
    domino=D[k]
    chaine=[domino]
    while len(chaine) != M :
        D=D[:k]
        m,k=valeurs_max(D)
        while not compatible(D[k], domino) :
            D=D[:k]
            m,k=valeurs_max(D)
        domino=D[k]
        chaine.append(domino)
    return chaine[::-1]
```

□

4 Exercice 2 : plus grand carré blanc

Problème. On considère le problème suivant: étant donné une image monochrome $n \times n$, déterminer le plus grand carré blanc, *i.e.* qui ne contient aucun point noir.



Chaque pixel est représenté le coefficient d'une matrice $V = (V_{i,j})$; $V_{i,j} = 0$ si le pixel ayant pour coordonnées (i, j) est blanc, et $V_{i,j} = 1$ s'il est noir.

Pour $(i, j) \in [\![1, n]\!]^2$, on note $\text{pgcb}(i, j)$ la *taille* (c'est à dire la longueur d'un côté) du plus grand carré blanc dont le pixel en bas à droite a pour coordonnées (i, j) et qui vaut 0 si le pixel de coordonnées (i, j) est noir.

1. Que vaut $\text{pgcb}(i, j)$ lorsque $i = 1$ ou lorsque $j = 1$?

   Solution. Lorsqu'un pixel de coordonnées (i, j) est noir, on a toujours $\text{pgcb}(i, j) = 0 = 1 - V_{i,j}$.
   Si un pixel de la première ligne ou de la première colonne est blanc, alors le plus grand carré donc ce pixel est le coin en bas à droite est lui-même et a donc pour longueur de côté $1 = 1 - V_{i,j}$. Ainsi, on a

   $$\forall (i, j) \in \{1\} \times [\![1, n]\!] \cup [\![1, n]\!] \times \{1\}, \quad \text{pgcb}(i, j) = 1 - V_{i,j}.$$

   □

2. Pour un pixel blanc de coordonnées (i, j) avec $i \neq 1$ et $j \neq 1$, donner une relation entre $\text{pgcb}(i, j)$, $\text{pgcb}(i - 1, j - 1)$, $\text{pgcb}(i - 1, j)$ et $\text{pgcb}(i, j - 1)$.

Solution. Observons la chose suivante, cruciale pour la suite.
Un carré de pixels C, de taille $m \times m$ est blanc si et seulement si :

- ✗ le pixel en bas à droite de C est blanc ;
- ✗ Les trois carrés $(m-1) \times (m-1)$ en haut à gauche, en haut à droite et en bas à gauche sont tous blancs.

Ainsi, si $V_{i,j}$ est blanc, on a la formule

$$\mathrm{pgcb}(i,j) = 1 + \min\left(\{\mathrm{pgcb}(i-1,j-1), \mathrm{pgcb}(i,j-1), \mathrm{pgcb}(i-1,j)\}\right).$$

□

3. En déduire l'écriture d'une fonction `pgcb`, en `Python`, qui prend en argument une image représentée sous forme de matrice $\mathtt{V} \in \mathcal{M}_n(\{0,1\})$ et qui renvoie le tableau des $\mathrm{pgcb}(i,j)$ pour $(i,j) \in [\![1,n]\!]^2$.

Solution. Il suffit d'implémenter les formules de récurrence ci-dessus :

```
import numpy as np

def pgcb(V) :
    n=len(V)
    P=[[0]*n]*n
    for i in range(n) :
        for j in range(n) :
            if V[i][j] == 0:
                if i==0 or j==0 :
                    P[i][j]=1
                else :
                    P[i][j] = 1+np.min(P[i-1][j-1], P[i-1][j], P[i][j-1])
    return P
```

□

4. Comment résoudre, à partir de la fonction précédente, le problème initial ?

Solution. Il suffit de récupérer la position, dans la matrice P renvoyée par la fonction précédente, du pixel qui est positionné en bas à droite du plus grand carré blanc.

```
def plus_grand_carre_blanc(V) :
    n=len(V)
    P=pgcb(V)
    i_max,j_max=0,0
    cote=1
    for i in range(n) :
        for j in range(n) :
            if P[i][j] > cote :
                i_max, j_max, cote = i, j, P[i][j]
    return (i_max, j_max), cote
```

□