éduSCOL
Informer et accompagner les professionnels de l'éducation

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE DE L'ÉDUCATION NATIONALE ET DE LA JEUNESSE

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et Sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

PROGRAMMATION DYNAMIQUE

- Histoire de l'informatique
- Structures de données
- Bases de données
- Architectures matérielles et systèmes d'exploitation
- Langages et programmation
- Algorithmique

Ce document a pour objectif d'accompagner le professeur dans sa préparation de cours. Il n'est pas une proposition de cours et présente quelques notions qui dépassent le niveau attendu des élèves. En outre, les différents exemples traités ici n'ont pas vocation à être tous utilisés en classe, le professeur reste libre d'en choisir de tout autres s'il le souhaite.

1. Le problème du rendu de monnaie

1.1. Retour sur l'algorithme glouton

On a déjà vu en classe de première qu'un algorithme glouton pouvait donner une réponse au problème.
Voir https://cache.media.eduscol.education.fr/file/NSI/76/4/RA_Lycee_G_NSI_algo-gloutons_1170764.pdf
Fixons les notations. On suppose donné un système monétaire où les valeurs faciales des pièces (ou des billets) sont rangées en ordre décroissant. Par exemple, le système Euro pourra être décrit par la liste `euros = [50, 20, 10, 5, 2, 1]`. Pour payer une somme de 48 unités on pourrait bien sûr payer 48 pièces de 1, ou encore 3 pièces de 10, 3 pièces de 5, 1 pièce de 2 et 1 pièce de 1.
On cherche à payer la somme indiquée, en supposant qu'on a autant de pièces de chaque valeur que de besoin, en utilisant un nombre minimal de pièces.

L'algorithme glouton consiste à payer d'abord avec la plus grosse pièce possible : ici, il s'agit de 20, puisque 50 > 48. Ayant donné 20, il reste 28 à payer, et on poursuit avec la même méthode. Finalement, on va payer 48 sous la forme $48 = 20 + 20 + 5 + 2 + 1$. On a eu besoin de 5 pièces.

Considérons un autre système monétaire (en fait c'est l'ancien système impérial britannique) représenté par la liste suivante de valeurs faciales : `imperial = [30, 24, 12, 6, 3, 1]`. Pour payer 48 l'algorithme glouton va répondre : $48 = 30 + 12 + 6$.
À la différence du système euro, pour lequel on peut démontrer que l'algorithme glouton donne toujours la réponse optimale, on constate que ce n'est pas le cas avec le système impérial, puisque on aurait pu se contenter de 2 pièces : $48 = 24 + 24$.
Rappelons comment peut se programmer l'algorithme glouton.

```
euros = [50, 20, 10, 5, 2, 1]
imperial = [30, 24, 12, 6, 3, 1]

def glouton(valeursFaciales, somme):
    i = 0 # index de la pièce qu'on va essayer
    p = len(valeursFaciales) # nombre de valeurs de pièces disponibles
```

```
    monnaie = [] # liste des pièces rendues
    while i<p and somme>0:
        if valeursFaciales[i]>somme:
            i += 1
        else:
            monnaie.append(valeursFaciales[i])
            somme -= valeursFaciales[i]
    if somme==0:
        return monnaie
    else:
        return None
```

L'appel `glouton(euros, 48)` renvoie la liste `[20, 20, 5, 2, 1]`; l'appel `glouton(imperial, 48)` renvoie la liste `[30, 12, 6]`.

1.2. Recherche de la réponse optimale

La réponse optimale est celle qui utilise le nombre minimal de pièces. On a vu que l'algorithme glouton échoue à la trouver en général (l'exemple de rendre 48 avec le sysètme impérial suffit à le prouver).
Une approche récursive permet de résoudre le problème : soit x une valeur faciale de l'une des pièces du système monétaire. Pour rendre une somme s de façon optimale, si l'on veut utiliser au moins une fois la pièce x, il suffit de rendre x et la somme $s-x$ de façon optimale. Il n'y a plus qu'à choisir, parmi tous les choix possibles de x, celui qui permet d'utiliser le minimum de pièces.
Autrement dit, si j'appelle $f(s)$ le nombre minimal de pièces qu'il faut utiliser pour payer la somme x, on a simplement $f(0)=0$ et

$$f(s) = \min_{x \leqslant s}(1 + f(s-x)),$$

le minimum étant calculé sur toutes les valeurs x d'une pièce.
Cette remarque permet d'écrire le programme récursif suivant :

```
from math import inf

def dynRecursif(valeursFaciales, somme):
    if somme < 0:
        return inf
    elif somme == 0:
        return 0
    mini = inf
    for x in valeursFaciales:
        if somme >= x:
            mini = min(mini, 1 + dynRecursif(valeursFaciales, somme - x))
    return mini
```

On a importé du module `math` la valeur spéciale `inf` qui représente l'infini : pour tout entier a, l'expression `a < inf` vaut `True`.
Hélas, l'exécution de l'appel `dynRecursif(euros, 48)` est **extrêmement** lente. les mêmes calculs étant effectués de façon répétée. Une analyse du problème montrerait que la complexité est exponentielle.
Une idée classique consiste à mémoriser les résultats des appels pour être sûr qu'on n'aura pas besoin de les calculer plusieurs fois. Ici, le calcul de $f(s)$ utilise le calcul de $f(s-x)$ pour chaque valeur de x. Autrement dit, $f(s)$ n'utilise au plus que les valeurs de $f(s-1), f(s-2), \ldots, f(3), f(2), f(1)$.
On va donc créer un tableau conservant ces données, et calculer de façon systématique pour des valeurs croissantes de l'index.

```
def dynMemoise(valeursFaciales, somme):
    f = [0] * (somme + 1)
    for s in range(1, somme + 1):
        f[s] = inf
        for x in valeursFaciales:
```

```
            if s >= x:
                f[s] = min(f[s], 1 + f[s - x])
    return f[somme]
```

Le calcul de $f(48)$ va peut-être utiliser plusieurs fois la valeur de $f(8)$, mais celle-ci n'aura cette fois été calculée qu'une seule fois, et aussitôt rangée en mémoire. Cela ne fonctionne que parce que pour calculer $f(48)$, par exemple, on a recours seulement aux valeurs de $f(s)$ pour $s < 48$.
Cette technique est habituellement appelée *mémoïsation*, fort laide tentative de traduction de l'anglais *memoization* mais qui est passée dans l'usage.
Cette fois l'appel `dynMemoise(euros, 48)` répond immédiatement 5 (l'algorithme glouton avait bien trouvé la réponse optimale) et `dynMemoise(imperial, 48)` renvoie 2, ce qui correspond à la solution optimale $48 = 24 + 24$.
L'algorithme est de complexité tout à fait raisonnable : les deux boucles emboîtées correspondent à un temps d'exécution de l'ordre de `somme * len(valeursFaciales)`.
En revanche, il a un coût en espace (ou en mémoire) : il faut réserver la place nécessaire pour ranger toutes les valeurs de $f(n)$.

1.3. Reconstitution des détails de la réponse optimale

Si on remplace la dernière ligne de la fonction `dynMemoise` par `return f`, l'appel `dynMemoise(imperial, 17)` renvoie le tableau `[0, 1, 2, 1, 2, 3, 1, 2, 3, 2, 3, 4, 1, 2, 3, 2, 3, 4]`.
Il y a une réponse optimale avec 4 pièces, comment la reconstituer ?
On a $f(17) = 4$, on cherche une pièce x telle que $f(17 - x) = 3$, on a le choix entre $x = 1$, $x = 3$ et $x = 12$. Choisissons $x = 12$.
On a $f(17) = 1 + f(5)$. On cherche maintenant une pièce x' telle que $f(5 - x') = 2$, on peut choisir $x' = 3$ ($x' = 1$ aurait également convenu). On a $f(17) = 1 + f(5) = 1 + 1 + f(2)$ et enfin $f(2) = 1 + f(1)$. Une solution optimale est donc de payer $17 = 1 + 1 + 3 + 12$, avec 4 pièces.
On peut modifier la fonction précédente pour reconstituer une décomposition optimale : il suffit de détailler le calcul du minimum et en profiter pour mémoriser les valeurs notées x, x' etc. ci-dessus.

```
def dynMemoiseReconstitue(valeursFaciales, somme):
    f = [0] * (somme + 1)
    g = [0] * (somme + 1)
    for s in range(1, somme + 1):
        f[s] = inf
        for x in valeursFaciales:
            if s >= x:
                if 1 + f[s - x] < f[s]:
                    f[s] = 1 + f[s - x] # mise à jour du minimum
                    g[s] = s - x        # on retient d'où l'on vient
    monnaie = []
    s = somme
    while s>0 :
        monnaie.append(s - g[s])
        s = g[s]
    return monnaie
```

2. Qu'est-ce que la programmation dynamique ?

On peut dire que la résolution algorithmique d'un problème relève de la programmation dynamique si

- ▷ le problème peut être résolu à partir de sous-problèmes similaires mais plus petits ;
- ▷ l'ensemble de ces sous-problèmes est discret, c'est-à-dire qu'on peut les indexer et ranger les résultats dans un tableau ;
- ▷ la solution optimale au problème posé s'obtient à partir des solutions optimales des sous-problèmes ;
- ▷ les sous-problèmes ne sont pas indépendants et un traitement récursif fait apparaître les mêmes sous-problèmes un grand nombre de fois.

En général, c'est trouver une équation récursive, comme on l'a fait en, 1.2 pour le problème du rendu de monnaie, qui constitue l'étape essentielle de la résolution du problème.
Les algorithmes gloutons, en revanche, ne fournissent pas toujours une solution optimale, il décompose le problème initial à l'aide d'un seul sous-problème.
Les algorithmes nombreux sont souvent moins coûteux en exécution, mais sans pouvoir garantir une solution optimale.
Dans les deux cas, une propriété importante à vérifier est que l'optimalité d'une solution au problème soit garantie par l'optimalité des solutions aux sous-problèmes.

3. Un autre exemple : le problème du sac à dos

3.1. Description du problème, une équation récursive

On dispose de n objets de poids entiers strictement positifs $p_0, p_1, \ldots, p_{n-1}$, et auxquels on attache une valeur, qui est également un entier strictement positif. On note $v_0, v_1, \ldots, v_{n-1}$ les valeurs de ces objets.
On dispose d'un sac qu'on ne doit pas surcharger : le poids total des objets qu'il contient ne doit pas dépasser un certain poids P.
On cherche à maximiser le total des valeurs des objets du sac.
Autrement dit, on cherche des nombres x_i valant 0 ou 1 tels que $\sum_{i=1}^{n} x_i p_i \leqslant P$ tout en maximisant $\sum_{i=1}^{n} x_i v_i$.
Par exemple, on peut considérer des objets de poids respectifs 1, 2, 5, 6 et 7 et de valeurs 1, 6, 12, 18, 22, 28, et un sac dont le poids ne peut dépasser 11.
On peut choisir les objets de poids 1, 2 et 7, obtenant une valeur égale à $1 + 6 + 28 = 35$.
On peut aussi choisir les objets de poids 5 et 6, obtenant une valeur égale à $12 + 18 = 40$.
Mais comment s'assurer qu'on ne peut pas mieux faire ?
Notons, pour $0 \leqslant i \leqslant n-1$ et $0 \leqslant p \leqslant P$, $V(i,p)$ la valeur maximale des objets qu'on peut ranger parmi ceux de numéros 0 à i sans dépasser le poids p. La valeur maximale que nous cherchons est alors $V(n-1, P)$.
Bien sûr, pour tout i, $V(i,0) = 0$ et $V(0,p) = \begin{cases} 0 & \text{si } p < p_0; \\ v_0 & \text{si } p \geqslant p_0. \end{cases}$
La clé de la résolution tient à l'équation récursive :

$$V(i,p) = \begin{cases} V(i-1,p) & \text{si } p < p_i; \\ \max(V(i-1,p), v_i + V(i-1,p-p_i)) & \text{si } p \geqslant p_i. \end{cases}$$

On a distingué le cas où on renonce à utiliser l'objet numéro i : la valeur maximale est alors $V(i,p)$. Si on utilise l'objet numéro i, on doit limiter à $p - p_i$ le poids consacré aux objets de numéros 0 à $i-1$ et ajouter sa valeur v_i à la valeur optimale du sous-problème.

3.2. Programmation

On en déduit facilement le programme suivant, où on initialise le tableau V à zéro.

```
def sacADos1(poids, valeurs, poidsMaximum):
    assert(len(poids) == len(valeurs))
    n = len(poids)
    V = [[0]*(poidsMaximum + 1) for i in range(n)]
    for p in range(poids[0], poidsMaximum + 1):
        V[0][p] = valeurs[0]
    for i in range(1, n):
        for p in range(poids[i]):
            V[i][p] = V[i - 1][p]
        for p in range(poids[i], poidsMaximum + 1):
            V[i][p] = max(V[i - 1][p], valeurs[i] + V[i - 1][p - poids[i]])
    return V[n - 1][poidsMaximum]
```

Avec `poids = [1, 2, 5, 6, 7]` et `valeurs = [1, 6, 18, 22, 28]`, l'appel `sacADos1(poids, valeurs, 11)` renvoie 40.

L'imbrication des deux boucles permet d'assurer un temps d'exécution de l'ordre de $n \times P$. Le coût en mémoire est également de l'ordre de $n \times P$, puisqu'il faut réserver la place du tableau V.
En fait, on pourrait consommer moins de mémoire. En effet on s'aperçoit que le calcul de la ligne `V[i]` du tableau utilise seulement les valeurs de la ligne précédent `V[i - 1]` et que la mise à jour de `V[i][p]` n'utilise que les valeurs de `V[i - 1][k]` pour k⩽p.
Ainsi peut-on réécrire le programme de la façon suivante, en n'utilisant qu'une ligne de mémoire pour le tableau V.

```
1 def sacADos2(poids, valeurs, poidsMaximum):
2     assert(len(poids) == len(valeurs))
3     n = len(poids)
4     L = [0] * poids[0] + [valeurs[0]] * (poidsMaximum + 1 - poids[0])
5     for i in range(1, n):
6         for p in range(poids[i], poidsMaximum + 1):
7             L[p] = max(L[p], valeurs[i] + L[p - poids[i]])
8     return L[poidsMaximum]
```

En ligne 4, on initialise la ligne correspondant à l'objet 0 : $V[0][p]$ est nul si $p < p_0$, et égal à v_0 si $p \geqslant p_0$.

4. Alignement de séquences : la plus longue sous-chaîne commune

4.1. Présentation du problème

Le séquençage du génome a conduit à une collaboration fructueuse entre généticiens et informaticiens et au développement de nouveaux algorithmes.
En effet, on peut coder le génome avec les quatre lettres ACTG qui sont les initiales de quatre bases nucléiques : adénine, cytosine, thymine et guanine.
Du point de vue algorithmique, on considère donc des mots (très très longs) sur cet alphabet.
Un problème utile aux généticiens est celui de l'alignement de séquences, qui se décline en de nombreux sous-problèmes, dont plusieurs peuvent être abordés à l'aide de la programmation dynamique.
Nous nous intéressons ici à la recherche de la plus longue sous-chaîne commune. Étant donné deux textes x et y sur l'alphabet $\{A, C, T, G\}$, on cherche le texte z le plus longueur possible qui soit à la fois extrait de x et de y. Dire que z est extrait de x signifie que l'on peut obtenir z en effaçant des lettres de x. En particulier, les lettres qui figurent dans z figurent, dans le même ordre dans x.
Prenons l'exemple de $x = ATATACAGGTCA$ et $y = GACTACACGACT$, pour lesquels une plus longue sous-chaîne commune est $z = ATACACA$.

```
A T A   T A C A G G T C   A
    A   T A C A       C   A
  G A C T A C A       C G A C T
```

Notons bien qu'il n'y a pas unicité ! $ATAACAT$ est également une plus longue sous-chaîne commune, de même longueur 7, comme le schéma suivant le montre :

```
  A   T A T A C   A G G T C A
  A   T A   A C   A     T
G A C T A C A C G A C   T
```

Dans la suite, nous noterons plscc(x, y) une plus longue sous-chaîne commuen de x et y, et $\lambda(x, y)$ sa longueur. [1]
Autrement dit : plscc$(ATATACAGGTCA, GACTACACGACT) = ATAACAT$
et $\lambda(ATATACAGGTCA, GACTACACGACT) = 7$.

1. comme il n'y a pas unicité, il s'agit d'un abus de langage. Il serait plus correct de noter plscc(x, y) pour l'ensemble des plus longues sous-chaînes communes de x et y et donc, si z est l'une d'elle, $z \in$ plscc(x, y).

4.2. Résolution du problème

Considérons deux mots x et y, dont on cherche la longueur $\lambda(x, y)$ des plus longues sous-chaînes communes.
On peut distinguer deux cas :

Premier cas Si les deux mots ont la même lettre finale, on écrit $x = uA$ et $y = vA$ où A est la lettre finale en commun et où u et v sont les préfixes des deux mots. Alors bien sûr, une plus longue sous-chaîne commune de x et y peut se construire en cherchant une plus longue sous-chaîne commune de u et v et d'y ajouter au bout la lettre A. Autrement dit : $\lambda(x, y) = \lambda(u, v) + 1$.

Deuxième cas Si les deux mots n'ont pas la même lettre finale, par exemple si $x = uA$ et $y = vT$, on ne peut pas aligner le A et le T. Mais on pourrait trouver un A dans v ou un T dans u, donc on doit chercher la plus longue sous-chaîne commune entre plscc(uA, v) et plscc(u, vT). Autrement dit : $\lambda(x, y) = \max(\lambda(x, v), \lambda(u, y))$.

4.3. Programmation

Calcul de la longueur d'une plus longue sous-chaîne commune

On applique la discussion menée ci-dessus.

Pour cela on va construire une matrice `longueur` telle que `longueur[i][j]` représente la longueur d'une plus longue sous-chaîne commune des préfixes `x[:i]` et `y[:j]` de x et y.

Il faut faire attention à l'initialisation de cette matrice, c'est-à-dire aux valeurs à donner à `longueur[i][0]` (ou bien, de la même façon, à `longueur[0][j]`). En effet, on ne peut pas, quand i (ou j) vaut 0, utiliser l'indice $i - 1$ (ou $j - 1$), ce qui oblige à gérer *à la main* ces cas particuliers.

C'est ce qui est fait dans le programme suivant, en lignes 5–16. C'est finalement le cas général, traité en lignes 17–24, qui se révèle le plus simple.

```
1  def lplscc(x, y):
2      n = len(x)
3      p = len(y)
4      longueur = [[0]*p for i in range(n)]
5      if x[0]==y[0]:
6          longueur[0][0] = 1
7      for i in range(1, n):
8          if x[i]==y[0]:
9              longueur[i][0] = 1
10         else:
11             longueur[i][0] = longueur[i - 1][0]
12     for j in range(1, p):
13         if x[0]==y[j]:
14             longueur[0][j] = 1
15         else:
16             longueur[0][j] = longueur[0][j - 1]
17     for i in range(1, n):
18         for j in range(1, p):
19             if x[i] == y[j]:
20                 longueur[i][j] = longueur[i - 1][j - 1] + 1
21             elif longueur[i - 1][j]>longueur[i][j - 1]:
22                 longueur[i][j] = longueur[i - 1][j]
23             else:
24                 longueur[i][j] = longueur[i][j - 1]
25     return longueur[n - 1][p - 1]
```

Calcul d'une plus longue sous-chaîne commune

On s'inspire du programme précédent pour calculer une plus longue sous-chaîne commune.

Il suffit de créer une nouvelle matrice, appelée ici `scc`, telle que `scc[i][j]` contienne une plus longue sous-chaîne commune de `x[:i]` et `y[:j]`. Il s'agit donc d'une matrice de chaînes de caractères.

Cela signifie qu'on associe à chaque affectation de `longueur` une affectation de `scc`.
Quand on recopie une valeur précédente dans `longueur`, on fait tout simplement de même pour la matrice `scc`.
En revanche, à l'affectation `longueur[i][j] = longueur[i - 1][j - 1] + 1`, on associe l'instruction suivante : `scc[i][j] = scc[i - 1][j - 1] + x[i]`, c'est-à-dire qu'on prolonge la sous-chaîne commune avec le caractère en commun qu'on vient de repérer.

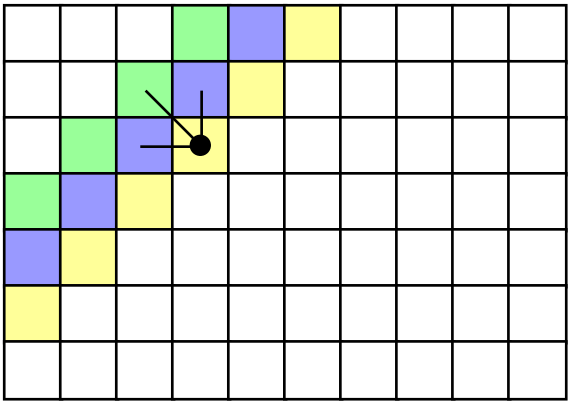
```
def plscc(x, y):
    n = len(x)
    p = len(y)
    longueur = [[0]*p for i in range(n)]
    scc = [[""]*p for i in range(n)]
    if x[0]==y[0]:
        longueur[0][0] = 1
        scc[0][0] = x[0]
    for i in range(1, n):
        if x[i]==y[0]:
            longueur[i][0] = 1
            scc[i][0] = y[0]
        else:
            longueur[i][0] = longueur[i - 1][0]
            scc[i][0] = scc[i - 1][0]
    for j in range(1, p):
        if x[0]==y[j]:
            longueur[0][j] = 1
            scc[0][0] = x[0]
        else:
            longueur[0][j] = longueur[0][j - 1]
            scc[0][j] = scc[0][j - 1]
    for i in range(1, n):
        for j in range(1, p):
            if x[i] == y[j]:
                longueur[i][j] = longueur[i - 1][j - 1] + 1
                scc[i][j] = scc[i - 1][j - 1] + x[i]
            elif longueur[i - 1][j]>longueur[i][j - 1]:
                longueur[i][j] = longueur[i - 1][j]
                scc[i][j] = scc[i - 1][j]
            else:
                longueur[i][j] = longueur[i][j - 1]
                scc[i][j] = scc[i][j - 1]
    return scc[n - 1][p - 1]
```

4.4. Une amélioration

Le coût d'exécution des programmes précédents est de l'ordre de $n \times p$ dans la mesure où on a deux boucles emboîtées à l'intérieur desquelles se déroulent un nombre fini d'instructions.
Cependant, leur coût en mémoire est important, puisqu'il faut réserver de la place pour une (ou deux) matrice(s) de taille $n \times p$.

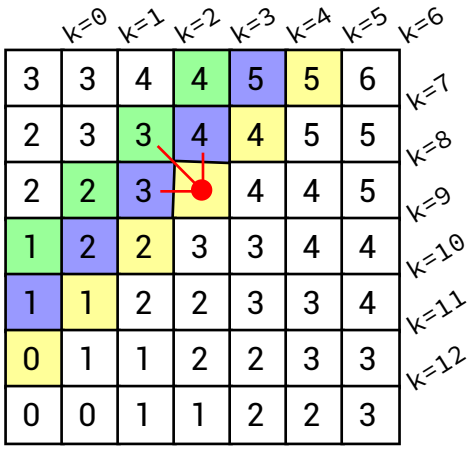
En examinant le code précédent, on observe que le calcul de l'élément d'indices (i, j) n'utilise que les valeurs d'indices $(i-1, j-1)$, $(i-1, j)$ ou $(i, j-1)$.
Comme le montre la figure ci-dessous, il suffit donc de conserver en mémoire deux "diagonales" [2] verte et bleue pour en déduire la suivante, colorée en jaune.
Si on est attentif, on s'aperçoit même que l'on peut calculer les nouvelles valeurs (en jaune) en écrasant les plus anciennes en vert. En effet, une case verte n'est utilisée qu'une seule fois, pour le calcul d'une seule case jaune.

Cela permet de passer d'un coût mémoire de l'ordre de $n \times p$ à un coût de l'ordre de $2\max(n, p)$. Cependant, cela rend techniquement beaucoup plus difficile la programmation de l'algorithme, le travail sur les indices est bien plus compliqué.



Dans une première lecture, on peut s'arrêter ici : le programme ci-dessous est vraiment peu lisible, très technique, et largement au-dessus de ce qui peut être attendu des lycéens.
Si $n \neq p$, on ajoute en nombre suffisant des espaces au mot x ou au mot y de telle sorte qu'ils soient finalement de la même longueur. Cela ne modifiera pas les sous-chaînes communes, puisque l'espace ne figurait au départ ni dans x ni dans y. Cela permet d'assurer que $n = p$, ce qui simplifie un petit peu le travail.

La figure ci-dessous montre comment on indexe les diagonales, de $k = 0$ à $k = 2n - 2$: la case jaune modifiée doit avoir le même index que la case verte qui est utilisée par le calcul.



On propose le programme suivant pour le calcul de la longueur de la plus longue sous-chaîne commune. Il va de soi que sa difficulté empêche son utilisation directe en classe. Il démontre simplement la possibilité de ne recourir qu'à deux tableaux de taille n, sans utiliser de matrice $n \times p$.
Quelques explications :

- k désigne le numéro de la diagonale en cours de traitement ;
- r désigne l'index dans les tableaux `longueurV` et `longueurB` de l'élément qui serait dans la première colonne de la matrice ;

2. dans la suite on continuera à les appeler des diagonales, ce qui est bien sûr un abus de langage

- `r` et `r2` sont les index dans la diagonale bleue des éléments qui sont utiles au calcul de l'élément d'index `r` dans la diagonale jaune (tantôt au-dessus, tantôt à sa gauche dans la matrice) ;
- `i` et `j` désignent les index des caractères observés dans x et y.

```
def lplscc2(x, y):
    n = len(x)
    p = len(y)
    if n < p:
        x = x + (" " * (p - n))
        n = p
    elif n > p:
        y = y + (" " * (n - p))
        p = n
    assert(n == p)
    longueurV = [0]*n
    longueurB = [0]*n
    # initialisation diagonale k=0
    k, r = 0, (n - 1)//2
    if x[0] == y[0]:
        longueurV[r] = 1
    # initialisation diagonale k=1
    k, r = 1, (n - 2)//2
    if x[0] == y[1]:
        longueurB[r + 1] = 1
    if x[1] == y[0]:
        longueurB[r] = 1
    #-------------- phase 1 ----------------
    for k in range(2, n): # les diagonales qui rallongent
        for r in range((n - k - 1)//2, (n - k - 1)//2 + k + 1):
            j = r - (n - k - 1)//2
            i = k - j
            r2 = r + 1 if (n + k)%2 == 0 else r - 1
            if x[i] == y[j]:
                longueurV[r] = longueurV[r] + 1
            else:
                longueurV[r] = max(longueurB[r], longueurB[r2])
        longueurV, longueurB = longueurB, longueurV
    #-------------- phase 2 ----------------
    for k in range(n, 2*n - 1): # les diagonales qui rétrécissent
        for r in range((k - n + 1)//2, (k - n + 1)//2 + 2*n - k - 1):
            i = n - 1 - r + (k - n + 1)//2
            j = k - i
            r2 = r + 1 if (n + k)%2 == 0 else r - 1
            if x[i] == y[j]:
                longueurV[r] = longueurV[r] + 1
            else:
                longueurV[r] = max(longueurB[r], longueurB[r2])
        longueurV, longueurB = longueurB, longueurV
    return longueurB[(n - 1)//2]
```

5. Références

Le lecteur qui voudrait en savoir davantage peut consulter les ouvrages suivants, d'un niveau général plus élevé bien sûr que celui attendu en NSI, mais qui peuvent contribuer utilement au professeur.

Benjamin Werner et François Pottier ont mis en ligne leur cours de l'école Polytechnique, dont le chapitre 8 est consacré à la programmation dynamique https://www.enseignement.polytechnique.fr/informatique/INF431/X11-2012-2013/inf431-poly.pdf

Pour les lecteurs anglophones, on peut aussi conseiller le chapitre 6 du livre *Algorithm Design*, de Jon Kleinberg et Éva Tardos qui enseignent à la Cornell University, ouvrage qui est également disponible en ligne http://www.cs.sjtu.edu.cn/~jiangli/teaching/CS222/files/materials/Algorithm%20Design.pdf