

RECHERCHE UNIDIMENSIONNELLE

Les codes de ce corrigé sont exclusivement rédigés en python. Par conséquent, la convention pour l'indexation des tableaux (ou listes) est celle de ce langage, à savoir que la première case est d'indice 0. Il en est de même dans le cas de tableau/liste bidimensionné.

- 1** Une simple boucle suffit, avec une condition d'arrêt qui consiste soit à sortir du tableau, soit à rencontrer une case contenant un 1.

```
def nombreZerosDroite(i,tab,n):
    p=i
    while p<n and tab[p]==0:
        p+=1
    return (p-i)
```

- 2** Il s'agit simplement de rédiger l'algorithme décrit par l'énoncé. Un compteur est utilisé pour retenir la longueur de la plus grande séquence de zéros rencontrée.

```
def nombreZerosMaximum(tab,n):
    p=0; max=0;
    while p<n:
        a=nombreZerosDroite(p,tab,n)
        if a>max:
            max=a
        p+=a+1
    return max
```

La façon dont s'incrémente le compteur p assure que chaque case du tableau est consultée au plus une fois par les différentes appels à `nombreZerosDroite`. Ainsi, la complexité est bien linéaire.

DE LA 1D VERS LA 2D

- 3** Le programme proposé utilise le principe suivant :

- Dans un premier temps, on lance un appel à `nombreZerosDroite` sur la i -ième ligne du tableau à partir de l'indice j . Cela détermine un premier rectangle rempli de 0 et de coin inférieur gauche la case (i, j) .
- Plus généralement, étant donné $x \in \llbracket 0 ; i \rrbracket$, un appel à `nombreZerosDroite` sur la ligne d'indice x à partir de j donne la longueur de plus grande séquence de 0 sur cette ligne à partir de i . Pour constituer un rectangle de 0 de taille maximale entre les lignes d'indices i et x , il suffit alors de considérer le minimum de ces valeurs.

On effectue donc une boucle avec un indice x décroissante de $(i - 1)$ à 0. Pour chaque valeur, on lance un nouveau calcul de `nombreZerosDroite` puis on retient la plus petite longueur courante. On calcule alors l'aire du rectangle ainsi trouvé, et on met à jour le maximum courant en fonction de la valeur de l'aire calculée .

```
def rectangleHautDroit(tab2,n,i,j):
    l=nombreZerosDroite(j,tab2[i],n)
    max=l
    for x in range(i):
        d=nombreZerosDroite(j,tab2[i-x-1],n)
        l=min(l,d)
        if l*(x+2) > max:
            max=l*(x+2)
    return max
```

On pourrait légèrement améliorer le temps de calcul en interrompant les boucles dans certaines conditions :

- Lors du calcul de d à chaque boucle, l'appel `nombreZerosDroite` utilise une boucle `while` jusqu'à déborder du tableau ou atteindre un 1. On pourrait également arrêter cette boucle dès lors que l'on dépasse la valeur courante de 1.
- La boucle sur x peut être arrêtée dès lors que l'on tombe sur une valeur `tab2[x][j]` égale à 1.

Toutefois, ces modifications alourdissent le code (ne permettant pas notamment d'utiliser le programme de la question 1) sans pour autant changer quoi que ce soit à la complexité dans le pire cas. C'est pourquoi nous ne les avons pas prises en compte.

4 Une solution naïve consisterait maintenant à appliquer `rectangleHautDroit` à tous les couples $(i, j) \in \llbracket 0; n-1 \rrbracket$ puis à prendre le maximum de ces valeurs.

Ce travail n'est pas demandé car il n'est pas optimal. En effet, un appel à `rectangleHautDroit` est de complexité $O(n^2)$ dans le cas le pire (si par exemple, le tableau ne contient que des 0, pour le coin en bas à gauche). Puisque l'on doit l'appliquer pour n^2 couples, on obtient une complexité excessivement mauvaise en $O(n^4)$.

L'approche naïve donne une solution de complexité $O(n^4)$ dans le pire cas.

5 Une solution à éviter consisterait à calculer naïvement pour chaque case le nombre de zéros au dessus d'elle (au sens large) à l'aide d'une boucle. Celle-ci étant dans le cas le pire de complexité linéaire, on aurait un coût en $O(n^3)$. Une solution plus intelligente consiste à remarquer que :

- Pour la première ligne, on a `col[i][j]=1` si et seulement si `tab2[i][j]=0` ;
- Pour tout $i \geq 1$, on a

$$\text{col}[i][j] = \begin{cases} 0 & \text{si } \text{tab2}[i][j]=0 \\ 1+\text{col}[i-1][j] & \text{sinon.} \end{cases}$$

Ces constatations permettent de remplir directement le tableau `col` ligne par ligne avec un coût constant pour le calcul de chacune des valeurs. Il suffit d'écrire le code suivant.

```
def colonneZeros(tab2,n):
    t=[]
    for k in range(n):
        t.append([])
    t[0]=[1-tab2[0][j] for j in range(n)]
    for i in range(1,n):
        for j in range(n):
            if tab2[i][j]==0:
                t[i].append(1+t[i-1][j])
            else:
                t[i].append(0)
    return t
```

6 Avec la méthode ci-dessus, le calcul de chaque valeur de `col` prend un temps constant. Comme il y a n^2 valeurs à calculer,

La fonction `colonneZeros` a un coût en $O(n^2)$ dans le pire cas.

ALGORITHME OPTIMISÉ

7 Notons que compte tenu de la convention qui fait démarrer les indices des tableaux/listes à 0, il faut remplacer la première condition d'arrêt par: « si $j = 0$, affecter $L[i] = 0$ ». Justifions par récurrence sur $i \in \llbracket 0; n-1 \rrbracket$ que le calcul de $L[i]$ est correct à l'aide du prédicat :

« $\mathcal{P}_i$: le calcul de $L[i]$ termine et est correct. »

- Le calcul de $L[0]$ est correct, car j est initialisé à 0, et le calcul s'arrête aussitôt en affectant 0 à $L[0]$.
- Supposons les calculs de $L[0], \dots, L[i-1]$ terminant et corrects. Le calcul de $L[i]$ s'effectue par une boucle pour laquelle on va exhiber l'invariant de boucle suivant :

« $\mathcal{I}$: Pour tout $k \in \llbracket j; i \rrbracket$, on a $\text{histo}[k] \geq \text{histo}[i]$ »

- A l'initialisation, $\mathcal{I}$ est vérifiée car $j = i$ et donc $\llbracket j; i \rrbracket = \{i\}$.
- Supposons la propriété $\mathcal{I}$ réalisée au début de l'étape d'une boucle, et vérifions qu'elle l'est toujours à la fin de cette étape. Dans les deux premiers cas, j n'est pas modifié, donc $\mathcal{I}$ reste valable à la fin de l'étape de boucle. Dans le dernier cas, on a $\text{histo}[j-1] \geq \text{histo}[i]$. Alors, par hypothèse de récurrence et définition de L ,

$$\forall k \in \llbracket L[j-1]; j-1 \rrbracket, \quad \text{histo}[k] \geq \text{histo}[j-1] \geq \text{histo}[i]$$

Puisque j est affecté à $L[j-1]$, l'invariant de boucle $\mathcal{I}$ est bien conservé à la fin de la boucle.

On en déduit que $\mathcal{I}$ est bien conservé tout au long du calcul de $L[i]$. Justifions maintenant ce dernier termine et est correct.

- A chaque étape de la boucle, soit le calcul termine, soit j est affecté à $L[j-1]$ qui est inférieur à $j-1$. Ainsi, j diminue au moins d'une unité. Comme le calcul s'arrête lorsque $j = 0$, cette propriété prouve que le calcul ne peut pas boucler indéfiniment. Il termine donc nécessairement.

- Lorsque le calcul termine, on a alors deux possibilités: soit $j = 0$, soit $\text{histo}[j-1] < \text{histo}[i]$. L'invariant assure que $\text{histo}[k] \geq \text{histo}[i]$ pour tout $k \in \llbracket j; i \rrbracket$. Dans les deux cas, j est alors clairement le plus petit entier pour lequel cette propriété est vérifiée.

Par conséquent, on a bien démontré que le calcul de $L[i]$ termine et donne le bon résultat.

- Par récurrence, on en déduit que pour tout $i \in \llbracket 0; n-1 \rrbracket$, le calcul de $L[i]$ est correct.

L'algorithme calcule correctement les valeurs de L .

Etudions maintenant le déroulement de cet algorithme sur l'exemple de l'énoncé. L'utilisation d'un dessin précis est grandement recommandée. Notons déjà que par définition

$$\text{histo}[i] = \begin{cases} i+1 & \text{si } i \in \llbracket 0; n-1 \rrbracket \\ 2n-i & \text{si } i \in \llbracket n; 2n-1 \rrbracket \end{cases}$$

et

$$L[i] = \begin{cases} i & \text{si } i \in \llbracket 0; n-1 \rrbracket \\ 2n-1-i & \text{si } i \in \llbracket n; 2n-1 \rrbracket \end{cases}$$

- Par définition, le calcul de $L[0]$ se termine immédiatement en affectant la valeur 0 (et ce indépendamment du tableau `histo` au passage).
- Pour tout $i \in \llbracket 1; n-1 \rrbracket$, on a $\text{histo}[i-1] < \text{histo}[i]$. Le calcul de $L[i]$ initialise donc j à i et termine tout de suite pour un coût constant.
- Pour tout $i \in \llbracket n; 2n-1 \rrbracket$, l'algorithme commence par affecter j à i . Ensuite,
 - Puisque $\text{histo}[i-1] \geq \text{histo}[i]$ (avec égalité si $i = n$), j est remplacé par $L[i-1]$ (qui a déjà été correctement calculé) soit $2n-i$.
 - A l'étape de boucle suivante, on a $\text{histo}[j-1] = \text{histo}[2n-1-i]$ soit $2n-i-1$. Cette fois, $\text{histo}[2n-i-1]$ est égal à $2n-i$, donc à $\text{histo}[i]$. Ainsi, j est remplacé par $L[2n-i-1]$, c'est-à-dire $2n-i-1$ lui-même.
 - Pour finir, on a à la boucle suivante $\text{histo}[j-1] = \text{histo}[2n-i-2]$ soit $2n-1-i$ qui est strictement inférieur à $\text{histo}[i]$, sauf cas particulier où $i = 2n-1$ auquel cas $j = 0$. Dans les deux cas, la boucle s'arrête.

Dans tous les cas, le calcul de $T[i]$ ne nécessite qu'un nombre constant (indépendant de n) d'opérations élémentaires.

On déduit de cette analyse que

L'algorithme s'exécute en $O(n)$ opérations élémentaires sur l'histogramme $[1, 2, 3, \dots, n-1, n, n, n-1, \dots, 2, 1]$.

L'implémentation de la fonction `calculeL` n'est pas demandée, mais elle suit tout simplement la description de l'énoncé. En décomposant l'histogramme comme une succession de montées/descentes, on peut démontrer que la complexité reste en $O(n)$ dans le pire cas quel que soit l'histogramme.

```
def calculeL(histo,n):
    L=[0]*n
```

```

def maj(i,j):
    if j==0:
        L[i]=0
    elif histo[j-1]<histo[i]:
        L[i]=j
    else:
        maj(i,L[j-1])
for i in range(n):
    maj(i,i)
return L

```

La fonction calculeR se définit de manière symétrique.

```

def calculeR(histo,n):
    L=[0]*n
    def maj(i,j):
        if j==n-1:
            L[i]=n-1
        elif histo[j+1]<histo[i]:
            L[i]=j
        else:
            maj(i,L[j+1])
    for i in range(n):
        maj(n-1-i,n-1-i)
    return L

```

8 C'est une conséquence immédiate de la définition de L. En effet,

$$\forall k \in \llbracket L[i]; R[i] \rrbracket, \quad \text{histo}[k] \geq \text{histo}[i]$$

En particulier,

Pour tout entier i , le rectangle commençant à l'indice $L[i]$ et terminant en $R[i]$ et de hauteur $\text{histo}[i]$ est inclus dans l'histogramme.

9 Il suffit de faire les constatations suivantes, qui découlent de la maximalité de l'aire du rectangle considéré :

- Le rectangle est inclus dans l'histogramme, donc $\text{histo}[i]$ est supérieur ou égal à h pour tout i dans $\llbracket l; k \rrbracket$.
- Le rectangle est d'aire maximale, donc il existe un entier i dans $\llbracket l; k \rrbracket$ tel que $\text{histo}[i]$ soit égal à h exactement. Dans le cas contraire, on pourrait augmenter la hauteur du rectangle d'une unité et contredire la maximalité de ce dernier.
- De la même manière, on a nécessairement $L[i] = 1$, sans quoi $\text{histo}[l-1]$ serait supérieur ou égal à $\text{histo}[i]$ soit h . On pourrait alors rajouter la colonne de hauteur h située à l'indice $l-1$ au rectangle et contredire à nouveau sa maximalité. De la même manière, on a nécessairement $R[i] = r$ exactement.

Ce qui précède assure l'existence d'au moins un entier i vérifiant les conditions de l'énoncé.

Il existe i dans $\llbracket l; r \rrbracket$ tel que $\text{histo}[i]=h$, $L[i]=1$ et $R[i]=r$.

Le dernier argument montre qu'en fait, les conditions $L[i]=l$ et $R[i]=r$ sont vérifiées par tout entier i dans $[[l; r]]$ tel que $histo[i]=h$.

10 D'après le résultat de la question précédente, le rectangle d'aire maximale inclut dans l'histogramme est à chercher parmi les rectangles débutant en $L[i]$, terminant en $R[i]$ et de hauteur $histo[i]$. Une simple boucle suffit à calculer toutes les aires correspondantes, et on se contente de renvoyer le maximum.

```
def plusGrandRectangleHistogramme(histo,n):
    L=calculeL(histo,n)
    R=calculeR(histo,n)
    A=[histo[i]*(R[i]-L[i]+1) for i in range(n)]
    return max(A)
```

La complexité de `calculeL` et `calculeR` étant linéaire d'après l'énoncé, cette fonction est également de complexité linéaire.

CONCLUSION

11 L'énoncé a déjà tout dit, il n'y plus qu'à appliquer la méthode. On calcule pour chaque ligne du résultat de `colonneZeros` le rectangle d'aire maximale inclus dans l'histogramme qu'elle représente, puis on prend le maximum de toutes ces valeurs.

```
def rectangleToutZero(tab2,n):
    h=colonneZeros(tab2,n)
    t=[plusGrandRectangleHistogramme(h[i],n) for i in range(n)]
    return max(t)
```

12 La fonction `colonneZeros` est de complexité quadratique tandis que la fonction `plusGrandRectangleHistogramme` est de complexité linéaire et est exécutée n fois pour un coût total quadratique. Enfin, la fonction `max` est de complexité linéaire. Ainsi,

La fonction `rectangleToutZero` est de complexité quadratique.