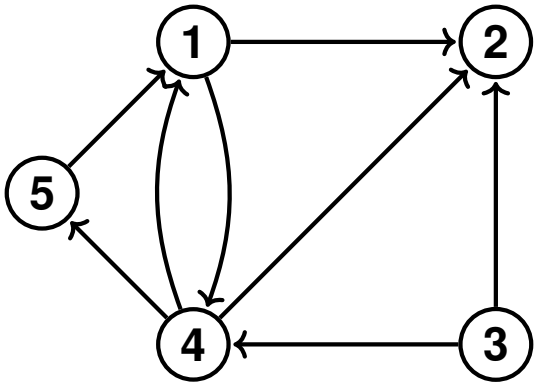
**Oral** **Pass'Ingénieur 2020–2021**

**Travaux dirigés 04**

## I Exercice : *Graphes*

On considère le graphe orienté $G = (S, A)$ suivant.




(a) Donnez le codage de $G$ par liste des successeurs, puis par matrice d'adjacence.

(b) Ecrire une fonction `degre_entrant`, prenant en paramètres un graphe codé par liste des successeurs et le numéro d'un sommet et calculant le degré entrant de ce sommet.

## II Exercice : *Plus longue sous-suite croissante*

Dans cet exercice, on veut algorithme recherchant une plus longue sous-suite strictement croissante (PLSSC) dans un tableau.

Une sous-suite est une suite de cases du tableau d'indices croissants, pas forcément successives. Par exemple dans le tableau `t` ci-dessous, les cases en gras constituent une sous-suite croissante.

| 6 | **1** | **4** | 9 | **5** | **11** | 3 |
|---|---|---|---|---|---|---|

**Longueur d'une plus longue sous-suite**

Pour résoudre le problème, on cherche à construire un tableau `long` tel que `long`$[i]$ est la plus grande longueur d'une sous-suite croissante finissant en `t`$[i]$.

(a) Que vaudra le tableau `long` pour le `t` donné en exemple au-dessus ?

(b) Supposons que $s_0, \ldots, s_p$ soit une PLSSC dans le sous-tableau terminant en `t`$[i]$.
Que peut-on dire de $s_0, \ldots, s_{p-1}$ ?

(c) Expliquez pourquoi

$$\texttt{long}[i] = \begin{cases} 1 + \max_{0 \leq j < i, t[j] < t[i]} \texttt{long[j]} \\ 1 \qquad\qquad\qquad\qquad\qquad \text{sinon} \end{cases}$$

(d) Écrire une fonction `longueurs` prenant en paramètres deux tableaux `t` et `long` et remplissant le tableau `long` grâce à la relation ci-dessus.

**Récupération de la plus longue sous-suite**

La fonction précédente calcule la longueur de la plus longue sous-suite, mais ne permet pas de récupérer cette sous-suite. Pour le faire, une technique consiste à enregistrer dans un tableau `pred`, pour chaque indice $i$, le prédécesseur de `t[i]` dans la sous-suite se terminant par `t[i]`.

Si ce prédécesseur n'existe pas (car la sous-suite la plus longue finissant en `t[i]` ne contient que `t[i]`), on posera `pred[i] = -1`.

(e) En modifiant `longueurs`, écrire une fonction `sous_suites` prenant en paramètres les tableaux `t, long` et `pred` et remplissant `long` et `pred` (on supposera `pred` déjà rempli avec des -1).

(f) Écrire une fonction `plssc` prenant en paramètres les tableaux `t`, `long`, et `pred` (supposé préalablement rempli) ainsi qu'un indice $i$ et retournant un tableau contenant la plus longue sous suite croissante finissant en $i$.

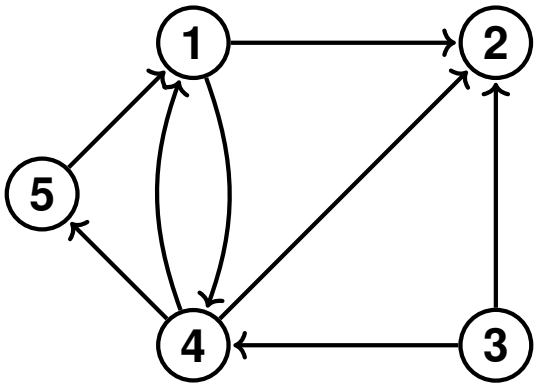
(g) Proposer un invariant, et prouver la correction de la fonction `longueur`.
Déterminer sa complexité. Peut-on l'améliorer ?

**Oral** **Pass'Ingénieur 2020–2021**

**Travaux dirigés 04**

## I Exercice : *Graphes*

On considère le graphe orienté $G = (S, A)$ suivant.




(a) Donnez le codage de $G$ par liste des successeurs, puis par matrice d'adjacence.

**Solution.**

On commence par donner pour chaque sommet la liste de ses successeurs :

| $x$ | $\Gamma^+(x)$ |
|---|---|
| 1 | 2, 4 |
| 2 | |
| 3 | 2, 4 |
| 4 | 1, 2, 5 |
| 5 | 1 |

La matrice d'adjacence $A$ est définie par :

$$A[i][j] = \begin{cases} 1 & \text{si } j \in \Gamma^+(i) \\ 0 & \text{sinon} \end{cases}$$

Donc :

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(b) Ecrire une fonction `degre_entrant`, prenant en paramètres un graphe codé par liste des successeurs et le numéro d'un sommet et calculant le degré entrant de ce sommet.

**Solution.**

On écrit le programme en C. On a besoin de définir 2 structures :

**une liste de sommets :** pour représenter $\Gamma^+(x)$

```
struct liste_sommet_s {
  int sommet ;
  // le suivant du dernier élément de la liste est NULL
  struct liste_sommet_s * suivant ;
} ;
```

**le graphe :** comme une liste de couples $(x, \Gamma^+(x))$ :

```
struct graphe_s {
  int sommet ;
  struct liste_sommet_s * gamma_plus ;
  // le suivant du dernier élément de la liste est NULL
  struct graphe_s * suivant ;
} ;
```

La fonction demandée prend en entrée un graphe (type **struct** graphe_s $*$) et un sommet $x$ (type **int**), et renvoie le degré entrant (type **int**). Il faut calculer combien de fois $x$ apparaît comme successeur d'un sommet.

```
// renvoie 1 si x est dans la liste, 0 sinon
int appartient ( int x , struct liste_sommet_s * gamma_plus ) {
  if ( gamma_plus == NULL ) return 0 ;
  if ( x == gamma_plus->sommet )
    return 1 + appartient ( x , gamma_plus->suivant ) ;
  return appartient ( x , gamma_plus->suivant ) ;
}

int degre_entrant ( int x , struct graphe_s * g ) {
  if ( g == NULL ) return 0 ;
  if ( appartient( x , g->gamma_plus ) )
    return 1 + degre_entrant ( x , g->suivant ) ;
  return degre_entrant ( x , g->suivant ) ;
}
```

---

## II Exercice : *Plus longue sous-suite croissante*

Dans cet exercice, on veut algorithme recherchant une plus longue sous-suite strictement croissante (PLSSC) dans un tableau.

Une sous-suite est une suite de cases du tableau d'indices croissants, pas forcément successives. Par exemple dans le tableau `t` ci-dessous, les cases en gras constituent une sous-suite croissante.

| 6 | **1** | **4** | 9 | **5** | **11** | 3 |
|---|---|---|---|---|---|---|

### Longueur d'une plus longue sous-suite

Pour résoudre le problème, on cherche à construire un tableau `long` tel que `long`$[i]$ est la plus grande longueur d'une sous-suite croissante finissant en `t`$[i]$.

(a) Que vaudra le tableau `long` pour le `t` donné en exemple au-dessus ?

**Solution.**

On fait le calcul à la main, on suppose $\texttt{t}[i]$ inclus dans la liste :

| 1 | 1 | 2 | 3 | 3 | 4 | 2 |
|---|---|---|---|---|---|---|

(b) Supposons que $s_0, \ldots, s_p$ soit une PLSSC dans le sous-tableau terminant en $\texttt{t}[i]$. Que peut-on dire de $s_0, \ldots, s_{p-1}$ ?

**Solution.**

1. la suite se termine en $i$, donc $s_p = \texttt{t}[i]$ ;
2. elle est croissante donc si $p \geq 1$ on a :
   (a) $\forall j \in \{0, \ldots, p-1\}, s_j \leq \texttt{t}[i]$ ;
   (b) C'est la PLSSC se terminant en $j$ tel que $s_{p-1} = \texttt{t}[j]$.

(c) Expliquez pourquoi

$$\texttt{long}[i] = \begin{cases} 1 + \max_{0 \leq j < i, t[j] < t[i]} \texttt{long[j]} \\ 1 \qquad\qquad\qquad\qquad\qquad \text{sinon} \end{cases}$$

**Solution.**

Soit $s_0, \ldots, s_p$ la PLSSC qui se termine en $\texttt{t}[i]$. D'après la question précédente, on a 2 cas :

- Si $p = 0$, alors $\texttt{t}[i]$ est le plus petit élément de `t` qu'on a vu jusque là, et $\texttt{long}[i] = 1$ ;
- Si $p > 0$, alors il existe $j < i$ tel que $\texttt{t}[j] < \texttt{t}[j]$ et $\texttt{long}[j] + 1 = \texttt{long}[i]$

(d) Écrire une fonction `longueurs` prenant en paramètres deux tableaux `t` et `long` et remplissant le tableau `long` grâce à la relation ci-dessus.

**Solution.**

Programme en C, les deux tableaux ont le même nombre $n$ d'éléments. On calcule le maximum comme à la question précédente.

```
void longueurs ( int * t , int * long , int n ) {
  int i , j ;
  for ( i = 0 ; i < n ; i++ ) {
    long[i] = 1 ;
    for ( j = 0 ; j < i ; j++ ) {
      if ( ( t[j] < t[i] ) && ( ( long[j]+1 ) > long[i] ) ) {
        long[i] = long[j] + 1 ;
      }
    }
  }
}
```

---

**Récupération de la plus longue sous-suite**

La fonction précédente calcule la longueur de la plus longue sous-suite, mais ne permet pas de récupérer cette sous-suite. Pour le faire, une technique consiste à enregistrer dans un tableau `pred`, pour chaque indice $i$, le prédécesseur de `t[i]` dans la sous-suite se terminant par `t[i]`.

Si ce prédécesseur n'existe pas (car la sous-suite la plus longue finissant en `t[i]` ne contient que `t[i]`), on posera `pred[i] = -1`.

(e) En modifiant `longueurs`, écrire une fonction `sous_suites` prenant en paramètres les tableaux `t, long` et `pred` et remplissant `long` et `pred` (on supposera `pred` déjà rempli avec des -1).

---

**Solution.**

À l'oral, il suffit d'expliquer et faire les changements sur la fonction précédente.

```
void sous_suites ( int * t , int * long , int * pred , int n ) {
  int i , j ;
  for ( i = 0 ; i < n ; i++ ) {
    long[i] = 1 ;
    for ( j = 0 ; j < i ; j++ ) {
      if ( ( t[j] < t[i] ) && ( ( long[j]+1 ) > long[i] ) ) {
        long[i] = long[j] + 1 ;
        pred[i]=j;
      }
    }
  }
}
```

---

(f) Écrire une fonction `plssc` prenant en paramètres les tableaux `t`, `long`, et `pred` (supposé préalablement rempli) ainsi qu'un indice $i$ et retournant un tableau contenant la plus longue sous suite croissante finissant en $i$.

---

**Solution.**

On parcourt le tableau `pred` à partir de $j = i$ tant que `pred`$[j] \neq -1$.

```
int * plssc ( int * t , int * long , int * pred , int i ) {
  int j , k ;
  int * res ;
  res = malloc ( long[i] * sizeof ( int ) ) ;
  j = i ;
  k = long[i] - 1 ;
  do {
    res[k] = j ;
    k = k-1 ;
    j = pred[i] ;
  } while ( j =-1 );
```

---

(g) Proposer un invariant, et prouver la correction de la fonction `longueur`. Déterminer sa complexité. Peut-on l'améliorer ?

---

**Solution.**

```
void longueurs ( int * t , int * long , int n ) {
  int i , j ;
  for ( i = 0 ; i < n ; i++ ) {
    long[i] = 1 ;
    for ( j = 0 ; j < i ; j++ ) {
      if ( ( t[j] < t[i] ) && ( ( long[j]+1 ) > long[i] ) ) {
        long[i] = long[j] + 1 ;
      }
    }
  }
}
```

L'invariant pour la boucle sur $i$ est la formule qu'on a utilisée : **au début de chaque boucle sur $i$ (après l'initialisation long[i]=1), et pour tout** $0 \leq k < i$, `long`$[k]$ **est la longueur de la PLSSC se terminant en** `t`$[k]$.
Pour chaque $i$, l'invariant pour la boucle sur $j$ est la formule qu'on a utilisée, mais limitée aux $j$ qu'on a déjà visité :

$$\texttt{long}[i] = \begin{cases} 1 + \max_{0 \leq \texttt{k} < \texttt{j} < \texttt{i}, t[\texttt{k}] < t[i]} \texttt{long[k]} \\ 1 \qquad\qquad\qquad\qquad\qquad\qquad \text{sinon} \end{cases}$$

- Le cas $k < j$ n'est jamais satisfait que $j = 0$, donc l'initialisation **long**[i]=1 est bonne ;
- Quand on passe de $j$ à $j + 1$, on a pris en compte le cas $k = j$ de la formule ;
- Quand on termine la seconde boucle, on a pris en compte tous les $j < i$, et donc on a bien calculé la formule pour le $i$ ;
- On fait la première boucle pour tous les $i$, donc on a bien rempli le tableau.

La complexité est :

$$\Sigma_{i=0}^{n-1} \Sigma_{j=0}^{i-1} 1 = \Sigma_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$$

Le calcul peut être fait en temps $\Theta(n)$.

# Oral d'informatique

Les programmes demandés pourront être rédigés dans le langage de votre choix : C, java, python...

## Exercice 1 : Automate pour un digicode

1. Proposez un automate (non déterministe) acceptant toutes les séquences de chiffres ($\{0, 1, \ldots, 9\}$) finissant par 3132.
2. Déterminisez l'automate trouvé question 1.

## Exercice 2 : Multiplication d'une suite de matrices

On considère un produit de $n$ matrices $M_0 M_1 \ldots M_{n-1}$.

On peut calculer ce produit en effectuant les produits dans l'ordre que l'on veut. Cependant l'ordre aura une influence sur la complexité du calcul.

1. Écrire une fonction `multiplication` prenant en entrée deux matrices $M$ et $N$ et retournant leur produit $M \cdot N$.

   Si les dimensions des deux matrices ne sont pas compatibles, on retournera un pointeur nul.
2. Justifier pourquoi il est raisonnable de considérer que la complexité du calcul de deux matrices de dimensions $(l, m)$ et $(m, n)$ est de $l * m * n$.
3. Donner un exemple de trois (dimensions de) matrices $M_0 M_1 M_2$ tel que les calculs $(M_0 M_1) M_2$ et $M_0 (M_1 M_2)$ n'aient pas la même complexité.

   Dans la suite,
   — on travaille avec un tableau $d$ de longueur $n + 1$ tel que chaque matrice $M_i$ est de dimensions $(d[i], d[i + 1])$.
   — on notera $p_{ij}$ le nombre minimal de multiplications d'entiers pour le calcul du produit $M_i M_{i+1} \ldots M_j$.
4. Expliquer pourquoi pour $i < j$,

$$p_{ij} = \min_{i \leq k < j} \Big[ p_{ik} + p_{k+1j} + d[i] d[k + 1] d[j + 1] \Big].$$

   (par convention $p_{ii} = 0$).
5. En déduire une fonction *récursive* `min_op_prod` prenant le tableau $d$ en paramètres ainsi que des indices $i$ et $j$ et calculant le nombre minimal de muliplications d'entiers pour le calcul du produit $M_i \ldots M_j$.

   La fonction précédente n'est pas efficace, car on est amené à recalculer plusieurs fois chaque terme $p_{ij}$.

   L'approche par *programmation dynamique* consiste à calculer successivement les valeurs des $p_{ij}$ dans l'ordre croissant des longeurs $j - i$, et à enregistrer ces valeurs dans un tableau $p$ à deux dimensions.

6. Calculer le tableau à la main pour un produit de quatre matrices de dimensions $(2 \times 4), (4 \times 3), (3 \times 1), (1 \times 5)$.
7. Écrivez une fonction `min_op_prod_dyn` prenant le tableau $d$ en paramètres et calculant le nombre minimum d'opérations pour calculer le produit $M_0 \dots M_{n-1}$ par programmation dynamique, en remplissant tout le tableau $p$.

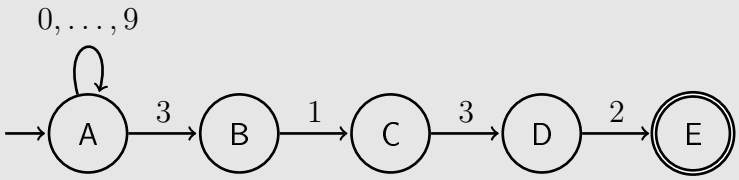
# Oral d'informatique

Les programmes demandés pourront être rédigés dans le langage de votre choix : C, java, python...

**Exercice 1 : Automate pour un digicode**

1. Proposez un automate (non déterministe) acceptant toutes les séquences de chiffres ($\{0, 1, \ldots, 9\}$) finissant par 3132.

On peut boucler sur l'état initial en lisant n'importe quel chiffre, puis (de manière non déterministe) décider de partir dans une suite d'états pour reconnaître la séquence finale.

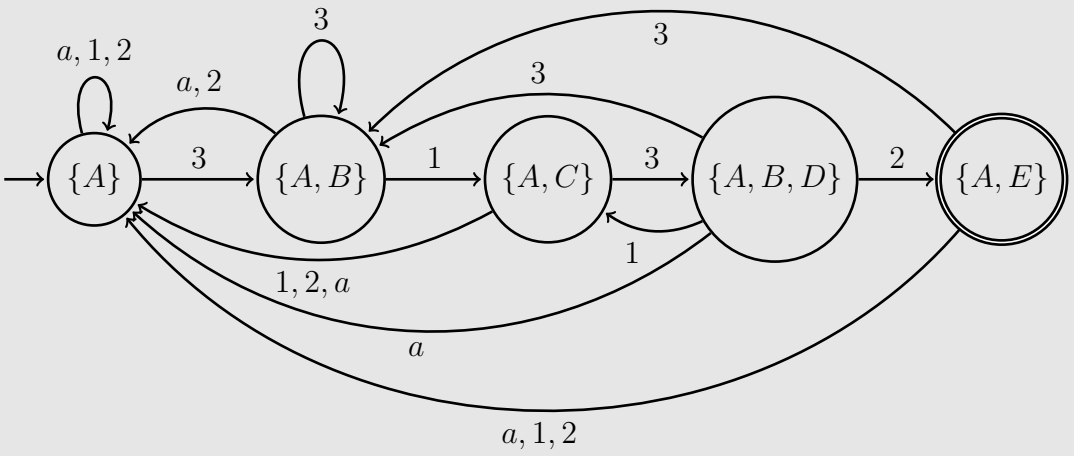



2. Déterminisez l'automate trouvé question 1.

On part du groupe d'états contenant le seul état initial, $\{A\}$. On calcule, pour chaque groupe d'états que l'on peut atteindre, le groupe d'états dans lequel on arrive après avoir lu chaque chiffre :

| | 1 | 2 | 3 | autre chiffre |
|---|---|---|---|---|
| $\{A\}$ | $\{A\}$ | $\{A\}$ | $\{A, B\}$ | $\{A\}$ |
| $\{A, B\}$ | $\{A, C\}$ | $\{A\}$ | $\{A, B\}$ | $\{A\}$ |
| $\{A, C\}$ | $\{A\}$ | $\{A\}$ | $\{A, B, D\}$ | $\{A\}$ |
| $\{A, B, D\}$ | $\{A, C\}$ | $\{A, E\}$ | $\{A, B\}$ | $\{A, \}$ |
| $\{A, E\}$ | $\{A\}$ | $\{A\}$ | $\{A, B\}$ | $\{A\}$ |

On obtient l'automate suivant (on notera $a$ pour "tout autre chiffre que 1,2 ou 3").



On attendra du candidat qu'il sache en outre discuter de l'intérêt de cette démarche : l'automate non déterministe est facile à concevoir ; il permet ensuite d'obtenir l'automate déterministe qui lui donne une solution efficace pour répondre au problème.

## Exercice 2 : Multiplication d'une suite de matrices

On considère un produit de $n$ matrices $M_0 M_1 \dots M_{n-1}$.

On peut calculer ce produit en effectuant les produits dans l'ordre que l'on veut. Cependant l'ordre aura une influence sur la complexité du calcul.

1. Écrire une fonction `multiplication` prenant en entrée deux matrices $M$ et $N$ et retournant leur produit $M \cdot N$.

   Si les dimensions des deux matrices ne sont pas compatibles, on retournera un pointeur nul.

La solution sera proposée en python. Les matrices seront des listes à deux dimensions de nombres.

Pour que les dimensions soit compatibles, il faut que le nombre de lignes de $N$ soit égal au nombre de colonnes de $M$ : on peut noter $l * m$ les dimensions de $M$, il faut que $N$ soit de dimensions $m * n$ pour un certain $n$.

```
def multiplication(M,N) :
    """
    M et N sont des matrices : tableaux a deux dimensions.
    La fonction retourne la matrice produit de M et N
    ou le pointeur nul si ces dimensions sont incompatibles.
    """

    #calcul des dimensions :
    l = len(M)
    m = len(M[0])
    n = len(N[0])

    #verification de la compatibilite
    if (len(N) != m :
        return None

    #construction de la matrice produit remplie de 0 :
    P = []
    for i in range(l) :
        P.append([])
        for j in range(n):
            P[i].append([])

    #calcul du produit par l'algorithme usuel :
    for i in range(l) :
        for j in range(n) :
            for k in range(m) :
```

```
                P[i][j] += M[i][k]*N[k][j]
    return P
```

2. Justifier pourquoi il est raisonnable de considérer que la complexité du calcul de deux matrices de dimensions $(l, m)$ et $(m, n)$ est de $l * m * n$.

La complexité de la fonction `multiplication` est dominée par les trois boucles imbriquées à la fin : elle est exécutée $l * m * n$ fois, avec les dimensions précédentes.

3. Donner un exemple de trois (dimensions de) matrices $M_0M_1M_2$ tel que les calculs $(M_0M_1)M_2$ et $M_0(M_1M_2)$ n'aient pas la même complexité.

Prenons par exemple $M_0$ de dimensions $n*1$, $M_1$ de dimensions $1*n$, $M_2$ de dimensions $n*1$ : alors le calcul de $(M_0M_1)$ est de coût $O(n*1*n) = O(n^2)$, le produit de $(M_0M_1)$ et $M_2$ est de coût $O(n*n*1 = O(n^2)$, soit un coût total en $O(n^2)$.

Le produit $(M_1M_2)$ est de coût $O(1*n*1) = O(n)$, le produit final de $M_0$ et de $(M_1M_2)$ est de coût $O(n*1*1) = O(n)$. Le coût total est linéaire en $n$.

On peut aussi très bien donner un exemple concret avec des matrices de taille petite, par exemple $1*2$, $2*1$ et $1*2$.

Dans la suite,

— on travaille avec un tableau $d$ de longueur $n+1$ tel que chaque matrice $M_i$ est de dimensions $(d[i], d[i+1])$.

— on notera $p_{ij}$ le nombre minimal de multiplications d'entiers pour le calcul du produit $M_iM_{i+1}\dots M_j$.

4. Expliquer pourquoi pour $i < j$,

$$p_{ij} = \min_{i \leq k < j} \Big[ p_{ik} + p_{k+1\,j} + d[i]d[k+1]d[j+1] \Big].$$

(par convention $p_{ii} = 0$).

$p_{ij}$ est la complexité du calcul de $M_iM_{i+1}\dots M_j$. Ce produit est calculé dans un certain ordre : on considère la dernière multiplication calculée : c'est-à-dire que le produit sera obtenu comme : $(M_i\dots M_k)(M_{k+1}\dots M_j)$ pour un certain $k \in \{i, \dots, j-1\}$, où les deux parenthèses auront déjà été calculées.

La complexité minimale $p_{ij}$ consiste donc à :

— calculer $(M_i\dots M_k)$ de manière optimale, pour un coût $p_{ik}$

— calculer $(M_{k+1}\dots M_j)$ de manière optimale, pour un coût $p_{k+1\,j}$

— calculer le produit des deux parenthèses, pour un coût $d[i]d[k+1]d[j+1]$ : en effet, les deux matrices $(M_i\dots M_k)$ et $(M_{k+1}\dots M_j)$ seront de dimensions $d[i]*d[k+1]$ et $d[k+1]*d[j]$ respectivement.

— enfin on choisit le $k$ pour lequel toute cette complexité est minimale.

5. En déduire une fonction *récursive* `min_op_prod` prenant le tableau $d$ en paramètres ainsi que des indices $i$ et $j$ et calculant le nombre minimal de muliplications d'entiers pour le calcul du produit $M_i \dots M_j$.

On peut en déduire une fonction calculant $p_{ij}$ à partir de cette relation de récurrence.

```
def min_op_prod(d,i,j) :
      #cas d'arret
      if (i==j) :
          return 0
      else :
          #calcul du minimum :
          #on l'initialise avec la valeur de l'expression pour k=
 i
          min = min_op_prod(d,i,i)+d[i]*d[i+1]*d[j+1]+min_op_prod
(d,i+1,j)
          for k in range(i+1,j) :
              cout = min_op_prod(d,i,k)+d[i]*d[k+1]*d[j+1]+
min_op_prod(d,k+1,j)
              if (cout<min ):
                  min = cout
          return min
```

La fonction précédente n'est pas efficace, car on est amené à recalculer plusieurs fois chaque terme $p_{ij}$.

L'approche par *programmation dynamique* consiste à calculer successivement les valeurs des $p_{ij}$ dans l'ordre croissant des longeurs $j - i$, et à enregistrer ces valeurs dans un tableau $p$ à deux dimensions.

6. Calculer le tableau à la main pour un produit de quatre matrices de dimensions $(2 \times 4), (4 \times 3), (3 \times 1), (1 \times 5)$.

On peut calculer le tableau suivant. Seule la diagonale du haut est calculée : seules les valeurs pour $i <= j$ ont un sens. On calcule successivement les cases rouges, bleues, vertes, violettes.

| $p_{ij}$ | $j=0$ | 1 | 2 | 3 |
|---|---|---|---|---|
| $i=0$ | 0 | $2*4*3=24$ | $\min(24+3*3*1, 2*4*1+12)=20$ | $\min(20+0+2*1*5, 24+15+2*3*5, 0+32+2*4*5)=30$ |
| 1 | | 0 | $4*3*1=12$ | $\min(12+4*1+5, 4*3*3+15)=32$ |
| 2 | | | 0 | $3*1*5=15$ |
| 3 | | | | 0 |

7. Écrivez une fonction `min_op_prod_dyn` prenant le tableau $d$ en paramètres et calculant le nombre minimum d'opérations pour calculer le produit $M_0 \dots M_{n-1}$ par programmation dynamique, en remplissant tout le tableau $p$.

La fonction va créer le tableau des $pij$, et calculer les cases avec la relation de récurrence, pour tous les intervalles (i,j) de taille croissante : pour t=0...n on calculera donc les $pi(i+t)$

```
def min_op_prod_dyn(d) :
      #nombre de matrices :
      n = len(d)-1

      #initialisation du tableau p :
      p = [[0 for i in range(n)] for j in range(n)]

      #on calcule les valeurs de p par taille croissante des
 intervalles

      for t in range(1,n) :
          for i in range(0,n-t) :
              #calcul de p{ij} pour j = i+t
              j = i+t
              #calcul du minimum :
              #on l'initialise avec la valeur de l'expression
 pour k=i
              p[i][j] = p[i][i]+d[i]*d[i+1]*d[j+1]+p[i+1][j]
              for k in range(i+1,j) :
                  cout = p[i][k]+d[i]*d[k+1]*d[j+1]+p[k+1][j]
                  if (cout<p[i][j] ):
                      p[i][j] = cout
      #au final, le cout total est la case p[0][n-1] du tableau

      return p[0][n-1]
```

On peut remarquer que notre programme ne calcule que la complexité minimale, sans retourner le découpage permettant d'obtenir cette complexité minimale : on pourrait pour cela enregistrer dans un autre tableau l'indice $k$ donnant la complexité minimale. Cela permet alors de retrouver toute la décomposition du produit optimal. Cela n'est pas demandé dans le sujet.