Correction
DS 6
Sujet d'informatique

Exercice : dichotomie

Soit $n \in \mathbb{N}^*$. Soit la fonction $f_n : x \mapsto x^n + \dfrac{x}{n} - 1$ sur $[0, +\infty[$.

1. Tracer le tableau de variations de f_n.

   Correction. La fonction f_n est dérivable et on calcule $f_n'(x) = nx^{n-1} + \frac{1}{n}$. Mais si $x \geqslant 0$ alors $x^{n-1} \geqslant 0$, de plus $\frac{1}{n} > 0$ (toujours), donc $f_n'(x) > 0$. La fonction est donc strictement croissante. De plus $f_n(0) = -1$ et la limite de f_n en $+\infty$ est $+\infty$, car chacun des deux termes x^n et $\frac{x}{n}$ tend vers $+\infty$ (attention c'est bien x qui tend vers $+\infty$ et pas n) d'où le tableau de variations :

x	0		$+\infty$
$f_n(x)$	-1	$\nearrow$	$+\infty$

   □

2. Justifier que l'équation $f_n(x) = 0$ admet une unique solution dans $]0, 1[$, qu'on appelle x_n.

   Correction. On calcule $f_n(1) = 1 + \frac{1}{n} - 1 = \frac{1}{n} > 0$. On peut s'aider en plaçant 1 sur le tableau de variations :

x	0		x_n		1		$+\infty$
$f_n(x)$	-1	$\nearrow$	0	$\nearrow$	$\frac{1}{n}$	$\nearrow$	$+\infty$

   Par le théorème des valeurs intermédiaires, et comme f_n est strictement croissante, il existe bien un unique nombre $x_n \in [0, 1]$ tel que $f_n(x_n) = 0$ et comme $f_n(0) \neq 0$ et $f_n(1) \neq 0$ le nombre x_n est en fait bien dans $]0, 1[$.

   □

3. Écrire une fonction Python `f(n, x)` qui prend en argument l'entier n et un nombre réel $x \geqslant 0$ et renvoie la valeur de $f_n(x)$.

   Correction. Vraiment peu de surprise pour cette question.

   ```
   def f(n, x):
       return x**n + x/n - 1
   ```

   □

4. Le but est de rechercher par dichotomie une approximation de la solution x_n. Recopier et compléter la fonction suivante pour qu'à tout moment on ait $a \leqslant x_n \leqslant b$ et qu'à la fin de la boucle l'écart $b - a$ soit strictement inférieur à la valeur ε passée en argument :

   ```
   def solution(n, epsilon):
       a = ...
       b = ...
       while ... :
           m = (a+b) / 2
           ...
       return (a, b)
   ```

Correction. Il s'agit d'une recherche par dichotomie classique. Comme annoncé on veut qu'à tout moment les variables a et b vérifient $a \leqslant x_n \leqslant b$ donc on initialise à $a = 0$ et $b = 1$. La condition dans la boucle est $b - a \geqslant \varepsilon$, qui est le contraire de $|b - a| < \varepsilon$ (mais les valeurs absolues ne sont pas nécessaires car on sait que $b > a$ donc $b - a > 0$). Enfin, il faut tester la valeur de $f_n(m)$ et s'aider du tableau de variations :

```
def solution(n, epsilon):
    a = 0
    b = 1
    while b-a >= epsilon:
        m = (a+b) / 2
        if f(n, m) > 0:
            # tableau de variations : la solution est entre a et m
            # donc il faut changer b
            b = m
        else:
            # sinon, elle est entre m et b : changer a
            a = m
    return (a, b)
```

□

Problème : chemins dans les matrices

Soit $(n, p) \in (\mathbb{N}^*)^2$. Soit $A = (a_{i,j})_{(i,j)\in[\![0,n-1]\!]\times[\![0,p-1]\!]}$ une matrice à n lignes et p colonnes, où les lignes et les colonnes sont numérotées à partir de 0. On appelle **chemin** dans A une suite de coefficients partant du coin en haut à gauche $a_{0,0}$, allant jusqu'au coin en bas à droite $a_{n-1,p-1}$, et tel que chaque coefficient est soit immédiatement à droite soit immédiatement en dessous du précédent (on dit que les coefficients sont **adjacents**). Ainsi nos chemins se déplacent toujours vers le bas droit. On s'intéresse à la somme des coefficients sur un chemin.
Par exemple pour la matrice $A = \begin{pmatrix} 9 & 9 & 5 \\ 1 & 2 & 5 \\ 4 & 6 & 9 \end{pmatrix}$ on peut tracer les chemins suivants, avec indiqué en dessous la somme des coefficients sur le chemin :

$$\underset{9+9+2+6+9=35}{\begin{pmatrix} \mathbf{9} & \mathbf{\to} & \mathbf{9} & & 5 \\ & & \mathbf{\downarrow} & & \\ 1 & & \mathbf{2} & & 5 \\ & & \mathbf{\downarrow} & & \\ 4 & & \mathbf{6} & \mathbf{\to} & \mathbf{9} \end{pmatrix}} \qquad \underset{9+1+4+6+9=29}{\begin{pmatrix} \mathbf{9} & & 9 & & 5 \\ \mathbf{\downarrow} & & & & \\ \mathbf{1} & & 2 & & 5 \\ \mathbf{\downarrow} & & & & \\ \mathbf{4} & \mathbf{\to} & \mathbf{6} & \mathbf{\to} & \mathbf{9} \end{pmatrix}} \qquad \underset{9+1+2+5+9=26}{\begin{pmatrix} \mathbf{9} & & 9 & & 5 \\ \mathbf{\downarrow} & & & & \\ \mathbf{1} & \mathbf{\to} & \mathbf{2} & \mathbf{\to} & \mathbf{5} \\ & & & & \mathbf{\downarrow} \\ 4 & & 6 & & \mathbf{9} \end{pmatrix}} \qquad (*)$$

Le but du problème est de déterminer la somme minimale que l'on peut former, parmi tous les chemins dans la matrice A. Dans cet exemple, c'est le chemin de droite qui a une somme égale à 26, et on peut montrer que c'est bien la somme minimale parmi tous les chemins possibles pour cette matrice-là.

1. Préliminaires

On représente les matrices de taille $(n, p) \in \mathbb{N}^* \times \mathbb{N}^*$ par des listes de listes ; plus précisément une matrice A de taille (n, p) est donnée par la **liste** de ses **lignes**. Ainsi le coefficient $a_{i,j}$ est bien donné par `A[i][j]`.

1. (a) Quelle matrice la liste `[[9, 3, 4], [2, 6, 7]]` représente-t-elle ?

Correction. C'est la matrice

$$\begin{pmatrix} 9 & 3 & 4 \\ 2 & 6 & 7 \end{pmatrix}$$

□

(b) Quelle liste de listes représente la matrice $\begin{pmatrix} 9 & 2 \\ 3 & 6 \\ 4 & 7 \end{pmatrix}$?

Correction. C'est la liste

`[[9, 2], [3, 6], [4, 7]]`

□

2. Si `A` représente une matrice à n lignes et p colonnes, qu'est-ce que `len(A)` ? Et `len(A[0])` ? **Justifier**.

Correction. `len(A)` est la longueur de `A` vue comme une liste, or `A` est une liste de lignes, donc c'est le nombre de lignes donc c'est n. `len(A[0])` est la longueur de la liste qui est la première ligne de `A`, donc c'est le nombre de colonnes, donc c'est p. □

3. Laquelle de ces deux syntaxes permet de créer une matrice nulle à n lignes et p colonnes ? **Justifier**.

(i) `[[0 for i in range(n)] for j in range(p)]`

(ii) `[[0 for j in range(p)] for i in range(n)]`

Correction. Il s'agit de la deuxième car c'est une liste de longueur n dont les éléments sont des listes nulles de longueur p, alors que pour la première c'est l'inverse.
... Sauf si $n = p$ auquel cas les deux syntaxes donnent bien une matrice carrée ! Les indices i, j n'ont pas d'importance ici et d'ailleurs on peut aussi écrire `for _ in range(n)` pour itérer sans donner de nom à la variable. □

4. Écrire une fonction `miroir(L)` qui prend en argument une liste `L` (de taille quelconque) et qui renvoie une nouvelle liste qui est le miroir de `L`, c'est à dire la liste rangée en ordre inverse.
Par exemple, `miroir([4, 2, 5, 3, 3])` doit renvoyer `[3, 3, 5, 2, 4]`.

Correction. Il faut d'abord créer une nouvelle liste nulle, qu'on appelle `M`, puis l'élément `M[i]` est alors égal à `L[n-1-i]` (où `n` est la longueur de `L`). Les fonctions suivantes sont valables :

```
def miroir(L):
    n = len(L)
    M = [0 for i in range(n)]
    for i in range(n):
        M[i] = L[n-1-i]
    return M
```

ainsi que

```
def miroir(L):
    n = len(L)
    M = []
    for i in range(n):
        M.append(L[n-1-i])
    return M
```

Remarque 1. Plus dans l'esprit d'un programmeur Python (mais moins d'un sujet d'algorithmique !), on a les syntaxes `[L[n-1-i] for i in range(len(L))]` (liste en compréhension) ainsi que tout simplement `L[::-1]` (sélection de tranche, du début, à la fin, avec un pas de `-1`).

□

2. Calcul du minimum

Pour résoudre notre problème, il serait très inefficace de lister *tous* les chemins puis de chercher lesquels ont une longueur minimale. Partant d'une matrice A, on forme alors une matrice $S = (s_{i,j})$ de même taille que A appelée **matrice des sommes minimales** où $s_{i,j}$ est la somme minimale obtenus sur les chemins reliant le coefficient $a_{0,0}$ (coin haut gauche de A) au coefficient $a_{i,j}$. Dans l'exemple $(*)$ avec $A = \begin{pmatrix} 9 & 9 & 5 \\ 1 & 2 & 5 \\ 4 & 6 & 9 \end{pmatrix}$, on trouve $S = \begin{pmatrix} 9 & 18 & 23 \\ 10 & 12 & 17 \\ 14 & 18 & 26 \end{pmatrix}$, ce qui démontre que 26 est bien la somme minimale qu'on peut réaliser par des chemins allant du coin haut gauche au coin bas droit.
La matrice S se calcule pas à pas en partant du coin haut gauche.

5. Montrer que *(on pourra admettre ces formules pour passer à la suite)* les coefficients de S se calculent avec la relation de récurrence suivante :

(i) $s_{0,0} = a_{0,0}$

(ii) Pour $i \in [\![0, n-2]\!]$, $s_{i+1,0} = s_{i,0} + a_{i+1,0}$

(iii) Pour $j \in [\![0, p-2]\!]$, $s_{0,j+1} = s_{0,j} + a_{0,j+1}$

(iv) Pour $i \in [\![0, n-2]\!]$ et $j \in [\![0, p-2]\!]$: $s_{i+1,j+1} = \begin{cases} s_{i+1,j} + a_{i+1,j+1} & \text{si } s_{i+1,j} < s_{i,j+1} \\ s_{i,j+1} + a_{i+1,j+1} & \text{sinon} \end{cases}$

Correction. D'abord le chemin commence au haut à gauche, donc la somme sur le chemin joignant $a_{0,0}$ à lui-même est simplement $a_{0,0}$. Ceci est la relation (i).
Pour un coefficient d'indice (i, j) se trouvant sur la colonne la plus à gauche, alors $j = 0$. Mais comme le chemin ne peut se déplacer que vers le bas ou la droite, alors pour aller à la ligne suivante (le coefficient $(i+1, 0)$ il faut un chemin vertical, la somme correspondante $s_{i+1,0}$ est donc obtenue en sommant avec $a_{i+1,0}$. Ceci explique la relation (ii). De même, la relation (iii) traduit que pour arriver en $(0, j+1)$ (première ligne) on a pour seule possibilité un chemin horizontal, donc passant par $(0, j)$ juste avant.
Enfin dans le dernier cas, par rapport au coefficient d'indice (i, j) alors supposons que l'on connaisse les valeurs $s_{i,j}$, $s_{i+1,j}$ et $s_{i,j+1}$, représentons cela :

$$\begin{pmatrix} a_{i,j} & & a_{i,j+1} \\ & & \downarrow \\ a_{i+1,j} & \to & a_{i+1,j+1} \end{pmatrix} \qquad \begin{pmatrix} s_{i,j} & & s_{i,j+1} \\ & & \downarrow \\ s_{i+1,j} & \to & s_{i+1,j+1} \end{pmatrix}$$

Il y a deux façons d'arriver au coefficient $(i+1, j+1)$: soit par la gauche, soit par au-dessus. Mais on choisira le chemin qui donne une plus petite somme, ainsi la somme minimale $s_{i+1,j+1}$ est égale à la somme de $a_{i+1,j+1}$ avec soit la somme du chemin arrivant par la gauche (c'est $s_{i+1,j}$) soit de celle du chemin arrivant par au-dessus (c'est $s_{i,j+1}$) en fonction tout simplement duquel donne la plus petite somme. □

6. (a) En appliquant ce procédé, déterminer la matrice des sommes minimales pour $B = \begin{pmatrix} 5 & 9 & 2 \\ 8 & 7 & 1 \\ 2 & 5 & 3 \end{pmatrix}$

Correction. On applique pas à pas, en commençant par un 5 dans le coin haut gauche. Puis on remplit facilement la première ligne et la première colonne. On trouve d'abord

$$S = \begin{pmatrix} 5 & 14 & 16 \\ 13 & * & * \\ 15 & * & * \end{pmatrix}$$

Ensuite on voit par exemple que pour remplir $s_{1,1}$ le chemin aura une somme plus petite en passant par la gauche que par au-dessus (ce qui correspond dans B au fait que le chemin $5 \to 8$ a une somme plus petite que $5 \to 9$) qu'il faut encore sommer avec 7 donc on complète par $13 + 7 = 20$:

$$S = \begin{pmatrix} 5 & 14 & 16 \\ 13 & 20 & * \\ 15 & * & * \end{pmatrix}$$

Poursuivant le procédé, on complète chaque $*$ en prenant le coefficient soit à gauche soit au-dessus et en sommant avec le coefficient correspondant de B. On arrive à

$$S = \begin{pmatrix} 5 & 14 & 16 \\ 13 & 20 & 17 \\ 15 & 20 & * \end{pmatrix}$$

puis

$$S = \begin{pmatrix} 5 & 14 & 16 \\ 13 & 20 & 17 \\ 15 & 20 & 20 \end{pmatrix}$$

□

(b) En déduire que la somme minimale de B, parmi tous les chemins reliant le coin haut gauche au coin bas droit, est égale à 20.

Correction. C'est par définition le coefficient qui apparait tout en bas à droite de S, et si les calculs sont corrects c'est bien 20. □

7. (a) Compléter le programme suivant pour écrire une fonction `matrice_sommes_minimales(A)` qui renvoie la matrice S des sommes minimales d'une matrice A passée en argument en utilisant les relations de récurrence de la question 5.

```
def matrice_sommes_minimales(A):
    n = ...
    p = ...
    S = ...
    for i in range(...):
        S[i+1][0] = ...
    for j in range(...):
        S[0][j+1] = ...
    for i in range(...):
        for j in range(...):
            ...
    return S
```

Correction. Le programme pré-écrit traduit presque directement les relations de récurrence. On n'hésite pas à utiliser les résultats de la partie préliminaire : `n` et `p` représentent la taille de `A`, `S` est au départ une matrice nulle. Les indices de listes et de matrices coïncident car les matrices sont indicées depuis le début à partir de 0 : pas de piège.

```
def matrice_sommes_minimales(A):
    # nombre de lignes de A
    n = len(A)
    # nombre de colonnes
    p = len(A[0])
    # matrice nulle de même taille que A
    S = [[0 for j in range(p)] for i in range(n)]
    # relation (i)
    S[0][0] = A[0][0]
    # relation (ii)
    for i in range(n-1):
        S[i+1][0] = S[i][0] + A[i+1][0]
    # relation (iii)
    for j in range(p-1):
        S[0][j+1] = S[0][j] + A[0][j+1]
    # relation (iv)
    for i in range(n-1):
        for j in range(p-1):
            if S[i+1][j] < S[i][j+1]:
                S[i+1][j+1] = S[i+1][j] + A[i+1][j+1]
            else:
                S[i+1][j+1] = S[i][j+1] + A[i+1][j+1]
    return S
```

Admirez d'ailleurs comme, même si on ne comprenait pas ces manipulations d'indices i, j, le programme traduit directement les relations de récurrence... Tout le jeu dans ce type d'algorithme est d'écrire proprement les relations de récurrence ! □

(b) En déduire une fonction `somme_minimale(A)` qui renvoie la somme minimale des chemins de A.

Correction. Il suffit d'appeler la fonction précédente, et de renvoyer son coefficient en bas à droite !

```
def somme_minimale(A):
    n = len(A)
    p = len(A[0])
    S = matrice_sommes_minimales(A)
    return S[n-1][p-1]
```

□

3. Calcul sur les chemins

Un chemin dans une matrice sera représenté tout simplement par une **liste** de **tuples** `(i, j)` représentant la suite de coefficients par lesquels passe le chemin. Dans notre exemple initial (∗), les chemins sont représentés respectivement par

- `[(0, 0), (0, 1), (1, 1), (2, 1), (2, 2)]`
- `[(0, 0), (1, 0), (2, 0), (2, 1), (2, 2)]`
- `[(0, 0), (1, 0), (1, 1), (1, 2), (2, 2)]`

8. La fonction suivante contient une ou plusieurs erreurs. Le but est qu'elle renvoie `True` si la liste de tuples (supposée non-vide) passée en argument représente bien un chemin partant du coin haut gauche et constitué de coefficients adjacents, et `False` sinon. Recopier et corriger la fonction :

```
def est_chemin(L):
    # si L[0] n'est pas le coin haut gauche, c'est faux
    (i, j) = L[0]
    if i != 0 and j != 0:
        return False
    # puis il faut tester si les coefficients sont adjacents
    for k in range(len(L)):
        (i, j) = L[k]
        (i2, j2) = L[k+1]
        # teste si le coeffcient suivant est à droite ou en dessous
        if (i2 == i and j2 == j+1) or (i2 == i+1 and j2 == j):
            return True
        else:
            return False
```

Correction. La fonction contient les erreurs suivantes :

- Au début on veut tester si le coefficient (i, j) est ou pas celui en haut à gauche, c'est à dire $(0, 0)$. Mais cela signifie $i = 0$ **et** $j = 0$, dont la négation est $i \neq 0$ **ou** $j \neq 0$. En Python il faut donc écrire `if i != 0 or j != 0`. Le reste de la logique est correct pour cette partie.
- On veut comparer les indices `L[k]` et `L[k+1]`. La borne du `range` n'est alors pas correcte car quand `k` sera le dernier indice possible alors d'indice `k+1` sera hors de la liste, il faut écrire `for k in range(len(L)-1)`
- Enfin sans cesse la même erreur : le test permet bien de savoir si le coefficient suivant est bien à droite ou en dessous, mais si cela est vrai, alors à cause du `return` la fonction s'arrête. Or on veut poursuivre la boucle, pour tester tous les coefficients du chemin. Il faut donc « penser à l'envers » et arrêter la boucle quand cette condition **n'est pas** réalisée, sinon renvoyer `True` **à la fin** et **hors de la boucle**. Le plus simple pour tester la condition contraire est d'utiliser `not`.

```
def est_chemin(L):
    (i, j) = L[0]
    if i != 0 or j != 0:
        return False
    for k in range(len(L)-1):
        (i, j) = L[k]
        (i2, j2) = L[k+1]
        if not((i2 == i and j2 == j+1) or (i2 == i+1 and j2 == j)):
            return False
    return True
```

□

9. Écrire une fonction `somme_chemin(A, L)` qui prend en argument une matrice `A` et un chemin `L` et qui renvoie la somme des coefficients de A le long du chemin donné.

Correction. On s'inspire de la boucle de la question précédente pour obtenir les coefficients `(i, j)` sur le chemin. Puis il s'agit d'un programme qui calcule tout simplement la somme (comme d'habitude !) des `A[i][j]`, il faut donc introduire une variable `s` qui accumule les sommes.

```
def somme_chemin(A, L):
    s = 0
    for k in range(len(L)):
        (i, j) = L[k]
        s = s + A[i][j]
    return s
```

Remarque 2. Plus dans l'esprit du programmeur Python, on pourra itérer directement sur `L` avec la syntaxe `for (i, j) in L`.

□

4. Recherche du chemin

La méthode présentée permet de calculer la somme minimale des chemins joignant le coin haut gauche au coin bas droit, mais elle ne dit pas *quel* chemin donne ce minimum. Il pourrait d'ailleurs y en avoir plusieurs.
Pour trouver un chemin donnant une somme minimale, dans une matrice A donnée, on s'y prend de la façon suivante :

- On calcule la matrice S des sommes minimales,
- On démarre au coin bas droit de A. On note i et j les numéros de ligne et de colonnes actuels. Tant qu'on n'est pas remonté jusqu'au coin haut gauche alors :
  - Si on se trouve sur la première colonne, on remonte d'une ligne.
  - Si $s_{i-1,j} < s_{i,j-1}$, c'est que la somme minimale $s_{i-1,j}$ pour relier le coin haut gauche au coefficient $a_{i-1,j}$ est inférieure à celle pour relier le coin haut gauche au coefficient $a_{i,j-1}$; autrement dit, pour arriver à $a_{i,j}$ le chemin de somme minimal provient du coefficient d'au-dessus. On remonte alors d'une ligne.
  - Dans tous les autres cas, on remonte d'une colonne.

10. (a) Implémenter cet algorithme en écrivant une fonction `remonter_chemin(A)` qui prend en argument une matrice A et qui renvoie la liste des tuples qui donnent les coefficients parcourus par cet algorithme, partant du coin bas droit et remontant jusqu'au coin haut gauche.

Correction. On utilise une boucle `while` pour implémenter ceci : on a besoin de deux variables i, j et tant que $(i, j) \neq (0, 0)$ on continue la boucle. On a aussi besoin de déclarer une liste `L` qui accumule les coefficients visités : dans la boucle on fera `L.append((i, j))` (il y a deux parenthèses car c'est la méthode `append`, qui prend en argument le tuples `(i, j)`).

```
def remonter_chemin(A):
    n = len(A)
    p = len(A[0])
    S = matrice_sommes_minimales(A)
    L = []
    i = n - 1
    j = p - 1
    while i != 0 or j != 0:
        L.append((i, j))
        # première colonne : remonter
        if j == 0:
            i = i - 1
        # suivre l'algorithme
        elif S[i-1][j] < S[i][j-1]:
            i = i - 1
        else:
            j = j - 1
    return L
```

□

(b) En déduire une fonction `chemin(A)` qui renvoie un chemin de somme minimale dans A, rangé dans l'ordre depuis le coin haut gauche jusqu'au coin bas droit.

Correction. Il suffit d'utiliser la fonction `miroir` du tout début du sujet ! Et un détail : éventuellement dans la fonction précédente il manque le coefficient $(0,0)$, qu'on peut rajouter manuellement en fin de boucle.

```
def chemin(A):
    L = remonter_chemin(A)
    L.append((0, 0))
    return miroir(L)
```

□

11. En appliquant ce procédé sur l'exemple 6 avec $B = \left(\begin{smallmatrix} 5 & 9 & 2 \\ 8 & 7 & 1 \\ 2 & 5 & 3 \end{smallmatrix}\right)$, donner un chemin de somme minimal pour B.

Correction. Écrivons côte à côte

$$B = \begin{pmatrix} 5 & 9 & 2 \\ 8 & 7 & 1 \\ 2 & 5 & 3 \end{pmatrix} \qquad S = \begin{pmatrix} 5 & 14 & 16 \\ 13 & 20 & 17 \\ 15 & 20 & 20 \end{pmatrix}$$

On part du coefficient $(2,2)$ (bas droite). L'algorithme nous dit qu'il faut remonter d'une ligne, car il faut une somme de seulement 17 pour arriver par au-dessus alors qu'il faut déjà 20 pour arriver à droite... donc le chemin termine ainsi :

$$\begin{pmatrix} 5 & 9 & 2 \\ & & \\ 8 & 7 & \mathbf{1} \\ & & \downarrow \\ 2 & 5 & \mathbf{3} \end{pmatrix}$$

Là encore, partant de ce coefficient $(1,2)$, il faut une somme de 20 pour arriver par la gauche, et de seulement 16 pour arriver par au-dessus, c'est donc que le chemin provient d'au-dessus :

$$\begin{pmatrix} 5 & 9 & \mathbf{2} \\ & & \downarrow \\ 8 & 7 & \mathbf{1} \\ & & \downarrow \\ 2 & 5 & \mathbf{3} \end{pmatrix}$$

Arrivé ici on n'a plus le choix, le chemin provient de la gauche en étant horizontal.

$$\begin{pmatrix} \mathbf{5} & \rightarrow & \mathbf{9} & \rightarrow & \mathbf{2} \\ & & & & \downarrow \\ 8 & & 7 & & \mathbf{1} \\ & & & & \downarrow \\ 2 & & 5 & & \mathbf{3} \end{pmatrix}$$

□