Tableaux (2/2)

D'après un TP de Stéphane Gonnord

Buts du TP

- ☞ Faire de nouveaux exemples de parcours de tableaux.
- ☞ Découvrir les tableaux à deux dimensions... et leurs pièges.
- ☞ Se confronter à un extrait de sujet de concours.
- ☞ Découvrir des algorithmes de programmation dynamique.

Créer (au bon endroit) un dossier associé à ce TP.
Lancer Spyder ou Pyzo, sauvegarder immédiatement au bon endroit le fichier `TP10.py`.

Après 40 minutes, on passera impérativement à la deuxième partie du TP.

1 Encore des parcours de tableaux

EXERCICE 1 *Écrire une fonction prenant en entrée un tableau t non vide, et renvoyant le nombre de couples (i,j) tels que $i<j$ et $t_i > t_j$. Complexité ?*

```
>>> inversions([5,1,2,4,3])
5
```

EXERCICE 2 *Savez-vous encore écrire une fonction prenant en entrée un tableau (disons de réels) et déterminant l'indice d'un élément maximal ?*
Chiche ! Programmez moi ça rapidement ! Complexité ?

```
>>> indice_maxi([0, 1, 2, 3, 42, 5, 6])
4
```

EXERCICE 3 *Écrire une fonction prenant en entrée un tableau t non vide, et renvoyant un couple (i,j) tel que $i \geqslant j$ et $|t_i - t_j|$ est maximal. Complexité ?*

EXERCICE 4
1. *Écrire une fonction prenant en entrée un tableau t non vide, et renvoyant* `True` *si le tableau possède un élément répété au moins deux fois, et* `False` *sinon. Complexité ?*

```
def undoublon(t):
    ...

>>> undoublon([10, 2, 5, 2, 42])
True
>>> undoublon([10, 2, 5, 3, 42])
False
```

2. *Adapter la fonction précédente pour que la complexité soit linéaire dans le cas où l'on connaît a priori la plage des valeurs de* `t` *: un intervalle d'entiers $[\![n;m]\!]$.*

EXERCICE 5 *Écrire une fonction prenant en entrée un tableau t non vide, et renvoyant la liste (éventuellement vide) des couples (i,j) tels que $i<j$ et $t_i = t_j$. Complexité ?*

```
def doublons(t):
    ...

>>> doublons([10, 2, 5, 2, 42])
[(1, 3)]
>>> doublons([10, 2, 5, 3, 42])
[]
```

EXERCICE 6 *Le polynôme $P = 4 - 3X + X^2 + 5X^3$ peut être stocké dans un tableau de la forme $[4,-3,1,5]$. D'une manière générale, on stocke le polynôme $p_0 + p_1X + ... + p_nX^n$ dans un tableau de longueur $n+1$: $[p_0, p_1, ..., p_n]$. Pour évaluer un polynôme en un réel (c'est-à-dire calculer $P(t) = 4 - 3t + t^2 + 5t^3$ dans l'exemple précédent), on peut additionner les différents monômes :*

Entrées : P, t
$s \leftarrow 0$ *# la somme provisoire*
pour i *allant de* 0 *à* $|P|$ **faire**
  $s \leftarrow s + P_i\, t^i$
Résultat : s

1. *Programmer cet algorithme dans une fonction* `evaluation`.

   ```
   def evaluation(P, t):
       ...

   >>> evaluation([4,-3,1,5], 2)
   42
   ```

2. *En imaginant que le calcul de t^i réclame $i-1$ multiplications, combien de multiplications demande l'évaluation d'un polynôme de degré d ?*

   On va améliorer un peu les choses, en calculant les t^i « à la volée » : une variable p est chargée de stocker les puissances successives de t : elle vaut 1 au départ, et est multipliée par t *à chaque étape.*

   Entrées : P, t
   $s \leftarrow 0$ *# la somme provisoire*
   $p \leftarrow 1$ *# la puissance de t en cours. Ici, $p = t^0$.*
   pour i *allant de* 0 *à* $|P|$ **faire**
     $s \leftarrow s + P_i \times p$
     $p \leftarrow p \times t$ *# calcul de la puissance suivante, c'est-à-dire t^1 puis t^2, etc.*
   Résultat : s

3. *Programmer une nouvelle fonction d'évaluation utilisant cette idée.*

4. *Avec ce nouvel algorithme, combien de multiplications demande l'évaluation d'un polynôme de degré d ?*

   On propose un nouvel algorithme, basé sur l'idée suivante :

   $$4 - 3t + t^2 + 5t^3 = 4 + t(\underbrace{-3 + t(\underbrace{1 + t \times \underbrace{5}_{s_0}}_{s_1})}_{s_2}),$$

   ce qui permet de calculer $P(t)$ grâce aux opérations suivantes (on effectue le calcul de l'intérieur vers l'extérieur, pour calculer s_0 puis s_1...) :

   $$s \leftarrow 5; \qquad s \leftarrow 1 + t \times s; \qquad s \leftarrow -3 + t \times s; \qquad s \leftarrow 4 + t \times s.$$

5. *Écrire l'algorithme permettant de calculer $P(t)$ suivant cette idée, puis un dernier programme d'évaluation suivant cet algorithme.* Il s'agit de l'algorithme de *Hörner*.

6. *Évaluer le nombre de multiplications nécessaires pour évaluer $P(t)$ avec cet algorithme.*

```
>>> P0 = [4, -3, 1, 5]
>>> [f(P0, 2) for f in [evaluation, evaluation_meilleure, horner]]
[42, 42, 42]
```

2 Manipulation de tableaux bidimensionnels

EXERCICE 7 *Expliquer ce qui suit !*

```
>>> t1 = [[0] * 3] * 2
>>> t2 = [[0] * 3 for _ in range(2)]
>>> t1,t2
([[0, 0, 0], [0, 0, 0]], [[0, 0, 0], [0, 0, 0]])
>>> t1[1][1] = 42
>>> t2[1][1] = 42
>>> t1,t2
([[0, 42, 0], [0, 42, 0]], [[0, 0, 0], [0, 42, 0]])
```

EXERCICE 8 *Les fonctions suivantes seront nécessaires à la suite du tp :*

- `matrice_nulle` *prend en entrée deux entiers (n, p) et retourne la matrice nulle de $\mathcal{M}_{n,p}(\mathbb{R})$.*
- `dimensions` *prend en entrée une matrice (un tableau à deux dimensions) et retourne le nombre de lignes et le nombre de colonnes de cette matrice.*
- `copie` *réalise la copie d'une matrice.*

1. *Réfléchir à la façon d'écrire de telles fonctions puis les coder (sans regarder le cours !).*
2. *Récupérer de telles fonctions qui marchent dans le fichier* `cadeau.py` *généreusement fourni.*

```
>>> m1 = matrice_nulle(3, 2)
>>> m1
[[0, 0], [0, 0], [0, 0]]
>>> dimensions(m1)
(3, 2)
>>> m2 = copie(m1)
>>> m3 = m1
>>> m1[1][1] = 3
>>> m1
[[0, 0], [0, 3], [0, 0]]
>>> m2
[[0, 0], [0, 0], [0, 0]]
>>> m3
[[0, 0], [0, 3], [0, 0]]
```

EXERCICE 9 *Écrire une fonction réalisant l'addition de deux matrices (de mêmes dimensions).*

EXERCICE 10 *Écrire une fonction réalisant la multiplication d'une matrice par un scalaire.*

EXERCICE 11 *Écrire une fonction renvoyant la transposée d'une matrice donnée en paramètre.*

EXERCICE 12 *Écrire une fonction réalisant la multiplication de deux matrices (de dimensions compatibles, c'est-à-dire de la forme (n, p) et (p, q)).*

EXERCICE 13 *Écrire une fonction réalisant le calcul de A^q, avec $A \in \mathcal{M}_n(\mathbb{R})$ et $q \in \mathbb{N}$ donnés. Combien de multiplications matricielles sont réalisées pour calculer A^q ?*

EXERCICE 14 *Combien d'additions et de multiplications de réels sont nécessaires pour effectuer les calculs des 5 derniers exercices ?*

	Addition	*Multiplication scalaire*	*Transposition*	*Multiplication matrices*	*Calcul de A^q*
Additions	$n \times p$	*0*	*0*		
Multiplications	*0*		*0*		

EXERCICE 15 *Écrire une fonction testant le caractère symétrique d'une matrice (et renvoyant donc un booléen). Elle doit utiliser strictement moins de n^2 comparaisons.*

```
>>> est_symetrique([[1,3],[3,2]])
True
>>> est_symetrique([[1,3],[-3,2]])
False
```

3 Extrait du sujet posé au concours Centrale 2016

3.1 Présentation du problème

Lors du dépôt d'un plan de vol, la compagnie aérienne doit préciser à quel niveau de vol elle souhaite faire évoluer son avion lors de la phase de croisière. Ce niveau de vol souhaité, le RFL pour *requested flight level*, correspond le plus souvent à l'altitude à laquelle la consommation de carburant sera minimale. Cette altitude dépend du type d'avion, de sa charge, de la distance à parcourir, des conditions météorologiques, ...

Cependant, du fait des similitudes entre les différents avions qui équipent les compagnies aériennes, certains niveaux de vols sont très demandés ce qui engendre des conflits potentiels, deux avions risquant de se croiser à des altitudes proches. Les contrôleurs aériens de la région concernée par un conflit doivent alors gérer le croisement de ces deux avions.

Pour alléger le travail des contrôleurs et diminuer les risques, le système de régulation s'autorise à faire voler un avion à un niveau différent de son RFL. Cependant, cela engendre généralement une augmentation de la consommation de carburant. C'est pourquoi on limite le choix aux niveaux immédiatement supérieur et inférieur au RFL.

Ce problème de régulation est modélisé par un graphe dans lequel chaque vol est représenté par trois sommets. Le sommet 0 correspond à l'attribution du RFL, le sommet + au niveau supérieur et le sommet − au niveau inférieur.

Chaque conflit potentiel entre deux vols sera représenté par une arête reliant les deux sommets concernés. Le coût d'un conflit potentiel (plus ou moins important en fonction de sa durée, de la distance minimale entre les avions, ...) sera représenté par une valuation sur l'arête correspondante.

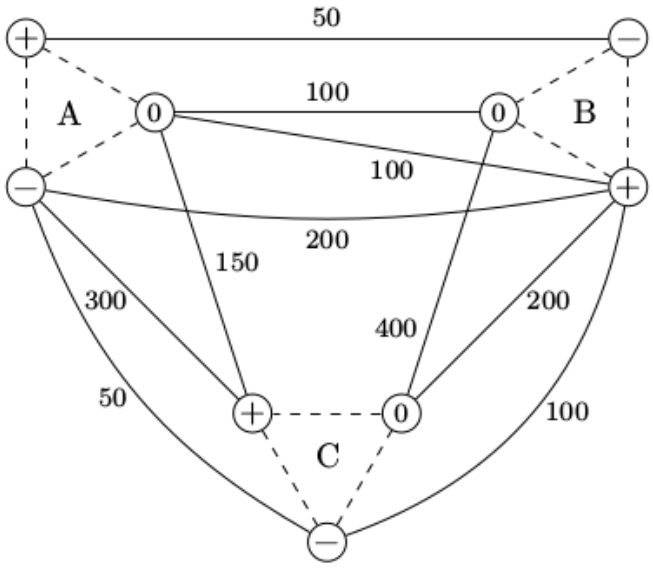


Figure 1

Dans l'exemple de la figure 1, faire voler les trois avions à leur RFL engendre un coût de régulation entre A et B de 100 et un cout de régulation entre B et C de 400, soit un cout total de la régulation de 500 (il n'y a pas de conflit entre A et C). Faire voler l'avion A à son RFL et les avions B et C au-dessus de leur RFL engendre un conflit potentiel de cout 100 entre A et B et 150 entre A et C, soit un cout total de 250 (il n'y a plus de conflit entre B et C).

On peut observer que cet exemple possède des solutions de coût nul, par exemple faire voler A et C à leur RFL et B au-dessous de son RFL. Mais en général le nombre d'avions en vol est tel que des conflits potentiels sont inévitables. Le but de la régulation est d'imposer des plans de vol qui réduisent le plus possible le coût total de la résolution des conflits.

3.2 Implémentation du problème

Chaque vol étant représenté par trois sommets, le graphe des conflits associé à n vols v_0, v_1, …, v_{n-1} possède $3n$ sommets que nous numéroterons de 0 à $3n-1$. Nous conviendrons que pour $0 \leqslant k < n$:

- le sommet $3k$ représente le vol v_k à son RFL ;
- le sommet $3k+1$ représente le vol v_k au-dessus de son RFL ;
- le sommet $3k+2$ représente le vol v_k au-dessous de son RFL.

Le coût de chaque conflit potentiel est stocké dans une liste de $3n$ listes de $3n$ entiers (tableau $3n \times 3n$) accessible grâce à la **variable globale** `conflit` : si i et j désignent deux sommets du graphe, alors `conflit[i][j]` est égal au coût du conflit potentiel (s'il existe) entre les plans de vol représentés par les sommets i et j. S'il n'y a pas de conflit entre ces deux sommets, `conflit[i][j]` vaut 0. On convient que `conflit[i][j]` vaut 0 si les sommets i et j correspondent au même vol (figure 1).

On notera que pour tout couple de sommets (i, j), `conflit[i][j]` et `conflit[j][i]`, représentent un seul et même conflit et donc `conflit[i][j] == conflit[j][i]`.

Dans le fichier `cadeau.py`, on trouvera la variable conflit implémentant le graphe de la figure 1.

```
conflit = [ [  0,  0,  0,100,100,  0,  0,150,  0],
            [  0,  0,  0,  0,  0, 50,  0,  0,  0],
            [  0,  0,  0,  0,200,  0,  0,300, 50],
            [100,  0,  0,  0,  0,  0,400,  0,  0],
            [100,  0,200,  0,  0,  0,200,  0,100],
            [  0, 50,  0,  0,  0,  0,  0,  0,  0],
            [  0,  0,  0,400,200,  0,  0,  0,  0],
            [150,  0,300,  0,  0,  0,  0,  0,  0],
            [  0,  0, 50,  0,100,  0,  0,  0,  0] ]
```

EXERCICE 16 *1. Écrire en Python une fonction* `nb_conflits()` *sans paramètre qui renvoie le nombre de conflits potentiels, c'est-à-dire le nombre d'arêtes de valuation non nulle du graphe.*

```
>>> nb_conflits()
8
```

2. Exprimer en fonction de n la complexité de cette fonction.

3.3 Régulation

Pour un vol v_k on appelle *niveau relatif* l'entier r_k valant 0, 1 ou 2 tel que :

- $r_k = 0$ représente le vol v_k à son RFL ;
- $r_k = 1$ représente le vol v_k au-dessus de son RFL ;
- $r_k = 2$ représente le vol v_k au-dessous de son RFL.

On appelle régulation la liste $(r_0, r_1, \ldots, , ..., r_{n-1})$. Par exemple, la régulation $(0, 0, \ldots, 0)$ représente la situation dans laquelle chaque avion se voit attribuer son RFL. Une régulation sera implantée en Python par une liste d'entiers.

Il pourra être utile d'observer que les sommets du graphe des conflits choisis par la régulation r portent les numéros $3k + r_k$ pour $0 \leqslant k < n$. On remarque également qu'au sommet s du graphe correspond le niveau relatif $r_k \equiv s \bmod 3$ et le vol v_k tel que $k = \lfloor s/3 \rfloor$.

EXERCICE 17 *1. Écrire en Python une fonction* `nb_vol_par_niveau_relatif(regulation)` *qui prend en paramètre une régulation (liste de n entiers) et qui renvoie une liste de 3 entiers* `[a, b, c]` *dans laquelle* `a` *est le nombre de vols à leurs niveaux RFL,* `b` *le nombre de vols au-dessus de leurs niveaux RFL et* `c` *le nombre de vols au-dessous de leurs niveaux RFL.*

```
>>> nb_vol_par_niveau_relatif([0, 1, 1])
[1, 2, 0]
```

2. Coût d'une régulation

On appelle coût d'une régulation la somme des coûts des conflits potentiels que cette régulation engendre.

(a) Écrire en Python une fonction `cout_regulation(regulation)` *qui prend en paramètre une liste représentant une régulation et qui renvoie le coût de celle-ci.*

```
>>> cout_regulation([0, 1, 1])
250
```

(b) Évaluer en fonction de n, la complexité de cette fonction.

(c) Déduire de la question (a) une fonction `cout_RFL()` *qui renvoie le coût de la régulation pour laquelle chaque avion vole à son RFL.*

```
>>> cout_RFL()
500
```

3. Combien existe-t-il de régulations possibles pour n vols ?

Est-il envisageable de calculer les coûts de toutes les régulations possibles pour trouver celle de coût minimal ?

3.4 L'algorithme minimal

On définit le coût d'un sommet comme la somme des coûts des conflits potentiels dans lesquels ce sommet intervient. Par exemple, le coût du sommet correspondant au niveau RFL de l'avion A dans le graphe de la figure 1 est égal à $100 + 100 + 150 = 350$.

L'algorithme *Minimal* consiste à sélectionner le sommet du graphe de coût minimal ; une fois ce dernier trouvé, les deux autres niveaux possibles de ce vol sont supprimés du graphe et on recommence avec ce nouveau graphe jusqu'à avoir attribué un niveau à chaque vol. Dans la pratique, plutôt que de supprimer effectivement des sommets du graphe, on utilise une liste `etat_sommet` de $3n$ entiers tels que :

- `etat_sommet[s]` vaut 0 lorsque le sommet `s` a été supprimé du graphe ;
- `etat_sommet[s]` vaut 1 lorsque le sommet `s` a été choisi dans la régulation ;
- `etat_sommet[s]` vaut 2 dans les autres cas.

EXERCICE 18 *1. Écrire en Python une fonction* `cout_du_sommet(s, etat_sommet)` *qui prend en paramètres un numéro de sommet* `s` *(n'ayant pas été supprimé) ainsi que la liste* `etat_sommet` *et qui renvoie le coût du sommet* `s` *dans le graphe défini par la variable globale* `conflit` *et le paramètre* `etat_sommet`.

```
>>> cout_du_sommet(0, [1, 0, 0, 0, 1, 0, 2, 2, 2])
250
```

2. Exprimer en fonction de n la complexité de la fonction `cout_du_sommet`.

3. *Écrire en Python une fonction* `sommet_de_cout_min(etat_sommet)` *qui, parmi les sommets qui n'ont pas encore été choisis ou supprimés, renvoie le numéro du sommet de coût minimal.*

```
>>> sommet_de_cout_min([1, 0, 0, 0, 1, 0, 2, 2, 2])
8
```

4. *Exprimer en fonction de n la complexité de la fonction* `sommet_de_cout_min`.

5. *En déduire une fonction* `minimal()` *qui renvoie la régulation résultant de l'application de l'algorithme Minimal.*

```
>>> minimal()
[1, 0, 1]
>>> cout_regulation([1, 0, 1])
0
```

6. *Quelle est sa complexité ? Commenter.*

3.5 Recuit simulé

L'algorithme de *recuit simulé* part d'une régulation initiale quelconque (par exemple la régulation pour laquelle chacun des avions vole à son RFL) et d'une valeur positive T choisie empiriquement. Il réalise un nombre fini d'étapes se déroulant ainsi :

- un vol v_k est tiré au hasard ;
- on modifie r_k en tirant parmi les deux autres valeurs possibles ;
  - si cette modification diminue le coût de la régulation, cette modification est conservée ;
  - sinon cette modification n'est conservée qu'avec une probabilité $p = \exp(-\Delta c/T)$ où Δc est l'augmentation de coût liée à la modification de la régulation ;
- le paramètre T est diminué d'une certaine quantité.

EXERCICE 19 *Écrire en Python une fonction* `recuit(regulation)` *qui modifie la liste* `regulation` *passée en paramètre en appliquant l'algorithme du recuit simulé. On pourra utiliser les fonctions* `randint` *et* `random` *du module* `random`.
On fera débuter l'algorithme avec la valeur $T = 1000$ et à chaque étape la valeur de T sera diminuée de 1%.
L'algorithme se terminera lorsque $T < 1$.

```
>>> recuit([0,0,0])
[1, 0, 2]
>>> cout_regulation([1, 0, 2])
0
```

Remarque. Dans la pratique, l'algorithme de recuit simulé est appliqué plusieurs fois de suite en partant à chaque fois de la régulation obtenue à l'étape précédente, jusqu'à ne plus trouver d'amélioration notable.

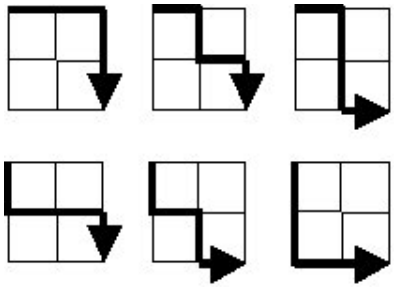
4 Pour ceux qui s'ennuient : un peu de programmation dynamique

Vous connaissez la programmation dynamique ? Il s'agit grosso-modo de calculer une quantité en exploitant une relation de récurrence/induction assez forte. La programmation dynamique est en général associée à la *memoïzation* : on stocke les valeurs dans un tableau dès qu'elles ont été calculées... et on réfléchit à un ordre pertinent pour calculer les différentes quantités.

Dans le premier exemple, on peut «compliquer» le problème en cherchant le nombre de chemins de $(0,0)$ vers (p,q) : ce nombre $\varphi(p,q)$ vérifie alors une relation assez simple ...

EXERCICE 20 Project Euler ; problem 15

Starting in the top left corner of a 2×2 *grid, and only being able to move to the right and down, there are exactly 6 routes to the bottom right corner.*
How many such routes are there through a 20×20 *grid ?*

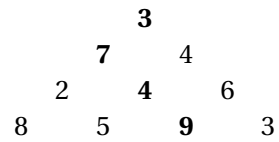


EXERCICE 21 *Créer le tableau à deux dimensions constitué des entiers présents dans le fichier* `triangle.txt`

Il s'agira d'une liste constituée de 100 listes constituées de i entiers pour i variant entre 1 et 100. On commencera par ouvrir le fichier avec l'éditeur, pour voir la tête du contenu.

EXERCICE 22 Project Euler, problem 67 (« Maximum path sum II »).

By starting at the top of the triangle below and moving to adjacent numbers on the row below, the maximum total from top to bottom is 23.
That is, $3+7+4+9=23$.
Find the maximum total from top to bottom in `triangle.txt`*, a 15K text file containing a triangle with one-hundred rows.*



EXERCICE 23 *Créer la matrice* $(80,80)$ *constituée des entiers présents dans le fichier* `matrix.txt`

Il s'agira d'une liste constituée de 80 listes constituées de 80 entiers. On commencera par ouvrir le fichier avec l'éditeur, pour voir la tête du contenu.

EXERCICE 24 Project Euler, problem 81 (« path sum : two ways »).
In the 5×5 *matrix below, the minimal path sum from the top left to the bottom right,* **by only moving to the right and down**, *is indicated in bold and is equal to* 2427.

$$\begin{pmatrix} \mathbf{131} & 673 & 234 & 103 & 18 \\ \mathbf{201} & \mathbf{96} & \mathbf{342} & 965 & 150 \\ 630 & 803 & \mathbf{746} & \mathbf{422} & 111 \\ 537 & 699 & 497 & \mathbf{121} & 956 \\ 805 & 732 & 524 & \mathbf{37} & \mathbf{331} \end{pmatrix}$$

Find the minimal path sum, in `matrix.txt`*, a 31K text file containing a* 80×80 *matrix, from the top left to the bottom right by only moving right and down.*

5 Besoin d'indications

- Exercice 20. Considérer le nombre de chemins allant de $(0,0)$ à (i,j) : pour $i,j>0$, on a $\varphi(i,j)=\varphi(i-1,j)+\varphi(i,j-1)$, tout simplement (mais un calcul massivement récursif ne passera pas : il faut « *memoïzer* »...). En fait, on montre sans trop de mal que $\varphi(i,j)=\binom{i+j}{i}$. Soit ici : 137846528820.
- Exercice 22. Si on note $\varphi(i,j)$ le poids maximal des chemins pour arriver en position (i,j) (indexation à la Python), alors l'objectif est $\varphi(99,99)$. Soit tab le tableau contenant le triangle de nombres, on a par ailleurs :

$$\forall (i,j) \mid \big(0<i, 0<j\leqslant i-1\big), \qquad \varphi(i,j)=tab_{i,j}+\max\big(\varphi(i-1,j),\varphi(i-1,j-1)\big).$$

- Exercice 24. Si on note $\varphi(i,j)$ le poids minimal des chemins pour arriver en position (i,j) (indexation à la Python), alors l'objectif est $\varphi(79,79)$. On a par ailleurs :

$$\forall i,j>0, \qquad \varphi(i,j)=M_{i,j}+\min\big(\varphi(i-1,j),\varphi(i,j-1)\big).$$