TP #8 — Parcours de tableaux à deux dimensions

L'objectif de ce TP est de manipuler des « tableaux » à deux dimensions en C, et d'en parcourir les cases suivant différentes contraintes.

Tableaux à deux dimensions en C

En C, les tableaux à deux dimensions sont des généralisations (assez immédiates) des tableaux à une dimension : un tel tableau de dimensions n et m sur un type t permet de stocker $n \times m$ objets du type t, un à chaque emplacement (i, j), avec $0 \leq i < n$, $0 \leq j < m$. Le premier indice i est celui des *lignes* du tableau, et le second j celui des *colonnes*.
Ces tableaux se déclarent et se manipulent avec une syntaxe généralisant (de façon assez immédiate) celle des tableaux à une dimension :

```c
int a[10][5];
a[0][0] = 3;
a[9][4] = 2;
a[10][0] = 0; // accès invalide
a[0][5] = 0;  // pareil
```

En mémoire, les éléments d'un tel tableau à deux dimensions sont stockés de façon consécutive, ligne par ligne (on parle de stockage *en ligne*, ou *row major*) : l'élément `a[i][j]` d'un tableau de dimensions $n \times m$ est simplement stocké à la case $i \times m + j$ d'un tableau unidimensionnel de longueur $n \times m$.
Le passage de paramètres tableaux à deux dimensions peut se faire suivant de multiples syntaxes, dont l'une généralise (toujours assez immédiatement) l'une des syntaxes possibles pour les tableaux à une dimension :

```c
void fun(int a[10][5]); // dimensions constantes
void fun2(size_t n, size_t m, int a[n][m]); // - variables
```

Il faut cependant considérer que ces signatures (surtout la seconde) sont **hors programme** (les types y apparaissant n'étant pas clairement au programme, et leur compréhension par les correcteurs de concours n'étant pas garantie). En pratique on les utilisera donc peu (voire pas du tout). C'est à dire qu'en règle générale l'on **s'abstiendra d'écrire des fonctions prenant des tableaux multidimensionnels en paramètres**. Pour les mêmes raisons, on ne cherchera pas à créer des « tableaux » multidimensionnels de durée de stockage *allouée* (par exemple en utilisant `malloc`) similaires à ceux ci-dessus.

Faux tableaux à deux dimensions

Pour répondre aux deux besoins ci-dessus (dans le cadre des CPGEs), on utilisera des faux tableaux construits comme des pointeurs de pointeurs (vers le type des éléments) : un « tableau » à deux dimensions vers un type `t` est un objet de type `t**` ; soit `a` une variable d'un tel type, on a alors que `a` pointe vers une zone mémoire contenant des pointeurs qui eux-même pointent vers des objets de type `t`. Ainsi, si l'on souhaite stocker $n \times m$ objets logiquement organisés en n lignes de m colonnes, on fera pointer `a` vers n pointeurs vers m objets. De cette façon,`a[0]` (de type `t*`) pointe vers les éléments de la première « ligne » du « tableau », `a[1]` vers ceux de la seconde etc., et `a[0][0]` (de type `t`) sera le premier élément de la première ligne, etc..
Cette façon de construire des « tableaux » a plusieurs inconvénients : leur création (et libération) nécessitera plusieurs appels à `malloc` (et à `free`), les éléments ne seront pas forcément stockés de façon consécutive (pouvant ainsi gâcher de la place et nuire aux performances des accès mémoires), et leur accès nécessitera plusieurs *indirections* (le déréférencement de plusieurs pointeurs). C'est pourtant la méthode que nous utiliserons par défaut…
Pour passer en paramètre un tel « tableau » à deux dimensions à une fonction, on propose d'utiliser la signature suivante :

```
void fun(size_t n, size_t m, int **pa)
```

Création & destruction

1. Écrivez une fonction C de signature :
   ```
   int **create(size_t n, size_t m)
   ```
   qui crée et renvoie un « tableau » (comme ci-dessus) de dimensions $n \times m$ de durée de stockage allouée.
2. Écrivez une fonction C de signature :
   ```
   void destroy(size_t n, int **pa)
   ```
   qui libère un tableau précédemment créé par `create`.
3. Écrivez une fonction C de signature :
   ```
   void test_cd(size_t u)
   ```
   qui crée puis libère immédiatement des « tableaux » de dimensions $n \times m \in [\![1, u]\!] \times [\![1, u]\!]$.
4. Testez, et validez notamment (du mieux que vous le pouvez) l'absence de fuites mémoire dans votre programme.

Initialisation & affichage

5. Écrivez une fonction C de signature :
   `void mult_pa(size_t n, size_t m, int **pa)`
   qui initialise son argument « tableau » de sorte que son élément d'indices (i, j) soit égal à $i \times j$.
6. Écrivez une fonction C de signature :
   `void print_pa(size_t n, size_t m, int **pa)`
   qui affiche les éléments de son argument « tableau », ligne par ligne, en affichant en premier la ligne `pa[0]`. Afin d'obtenir un affichage « aligné » *ad hoc* pour les questions suivantes, on affichera chaque élément sur au moins deux chiffres en utilisant le spécifieur de conversion « `"%02d"` ».
7. De même pour une fonction C de signature :
   `void print_pa_rev(size_t n, size_t m, int **pa)`
   qui affiche les éléments ligne par ligne dans l'ordre inverse des lignes.
8. Testez.

Parcours serpent

On souhaite pouvoir parcourir les cases d'un tableau en ordre « serpent », c'est à dire ligne par ligne (ou colonne par colonne) avec la contrainte que lorsque l'on passe d'une ligne (disons) à une autre, la colonne du prochain élément parcouru ne change pas.
Par exemple, un ordre de parcours satisfaisant ces contraintes pour un tableau de dimensions 5×10 est :

```
00 01 02 03 04 05 06 07 08 09
19 18 17 16 15 14 13 12 11 10
20 21 22 23 24 25 26 27 28 29
39 38 37 36 35 34 33 32 31 30
40 41 42 43 44 45 46 47 48 49
```

9. Écrivez une fonction C de signature :
   `void fill_snake_row(size_t n, size_t m, int **pa)`
   qui initialise son argument « tableau » en écrivant dans chaque case son ordre de visite (en partant de zéro) dans un parcours serpent en ligne.
10. Testez *via* une fonction de test dédiée (avec au moins trois cas).
11. Écrivez une fonction C de signature :
    `void fill_snake_col(size_t n, size_t m, int **pa)`
    qui fait de même pour un parcours serpent en colonne.
12. Testez *via* une fonction de test dédiée (avec au moins trois cas).

Courbe « de Hilbert »

On souhaite maintenant pouvoir parcourir les cases d'un tableau dans l'ordre donné par la courbe dite « de Hilbert ». Pour cela, on se placera dans le cas particulier d'un tableau carré dont les dimensions sont des puissances de deux. Cet ordre de parcours a une propriété de « localité » intéressante (parfois également utile en pratique pour des questions de perforance) : si l'on découpe le tableau en quatre sous-tableaux « sud-ouest (SO », « NO », « NE », « SE », l'on parcourera d'abord tous les éléments d'un premier sous-tableau avant de parcourir ceux d'un second sous-tableau, avant de parcourir ceux d'un troisième, avant de parcourir ceux du dernier sous-tableau, et ce récursivement à l'intérieur de chacun des sous-tableaux ! De plus, les éléments sont parcourus consécutivement (suivant une courbe, comme pour les parcours serpent).
Par exemple, pour des tableaux 2×2, 4×4 et 8×8 l'on obtient (en affichant la première ligne *en bas*) :

```
01 02
00 03

05 06 09 10
04 07 08 11
03 02 13 12
00 01 14 15

21 22 25 26 37 38 41 42
20 23 24 27 36 39 40 43
19 18 29 28 35 34 45 44
16 17 30 31 32 33 46 47
15 12 11 10 53 52 51 48
14 13 08 09 54 55 50 49
01 02 07 06 57 56 61 62
00 03 04 05 58 59 60 63
```

Pour implémenter ce parcours, on propose d'utiliser une fonction `pos_hc` qui calcule les indices (ou coordonnées) (x, y) du $k$ième point à être parcouru, en utilisant un automate proposé par Henry S. Warren Jr. :
Soit $n = 2^\ell$ les dimensions du tableau et $k \in [\![0, n^2 - 1]\!]$ un numéro d'ordre de parcours, on va calculer les ℓ bits des écritures en base deux (sur ℓ bits) de x et y bit par bit, dans l'ordre des bits de poids fort.
On initialise un *état courant* à A, et deux variables x et y à 0. On considère ensuite les 2ℓ bits de l'écriture en base deux de k, que l'on va lire deux par deux depuis les bits de poids fort. À chaque lecture on détermine deux choses :

— quel doit être l'état (A, B, C ou D) pour la prochaine étape
— par quels bits de poids faible « prolonger » les écritures en base deux de x et y

Ces informations sont données pour les 16 cas possibles (4 états fois 4 valeurs pour les deux bits lus) de la table suivante :

État courant	Bits lus	Bits à ajouter à (x, y)	Prochain état
A	00	$(0, 0)$	B
A	01	$(0, 1)$	A
A	10	$(1, 1)$	A
A	11	$(1, 0)$	D
B	00	$(0, 0)$	A
B	01	$(1, 0)$	B
B	10	$(1, 1)$	B
B	11	$(0, 1)$	C
C	00	$(1, 1)$	D
C	01	$(1, 0)$	C
C	10	$(0, 0)$	C
C	11	$(0, 1)$	B
D	00	$(1, 1)$	C
D	01	$(0, 1)$	D
D	10	$(0, 0)$	D
D	11	$(1, 0)$	A

Par exemple, si l'état courant est B, les bits de k lus sont 11 et que les valeurs courantes de x et y (en base deux) sont 100 et 10, alors le prochain état est C et l'on modifie x et y en leur « ajoutant » un bit de poids faible valant respectivement 0 et 1, donnant $x = 1000$ et $y = 101$.

On définit le type :

```
struct point
{
    size_t x;
    size_t y;
};
```

13. Écrivez une fonction C de signature :

```
struct point pos_hc(size_t n, int ln, size_t k)
```

qui utilise l'approche décrite ci-dessus pour calculer et renvoyer les coordonnées du $k$ième point à être visité, où `ln` désigne le logarithme (en base deux) de n (soit ℓ dans les notations ci-dessus).
On pourra (si l'on veut) utiliser certaines *opérateurs bits à bit* pour écrire cette fonction (*cf.* poly, § 10.3), mais ce n'est pas obligatoire.

14. Déduisez-en une fonction :

```
void fill_hc(size_t n, int ln, int **pa)
```

qui initialise son argument « tableau » avec l'ordre de parcours donné par le courbe « de Hilbert ».

15. Testez *via* une fonction de test dédiée.

Spirale rectangulaire

On souhaite maintenant pouvoir parcourir les cases d'un tableau rectangulaire dans l'ordre d'une spirale commençant à la case d'indices $(0, 0)$ et commençant par parcourir la première ligne. Par exemple, pour des tableaux de dimensions 10×10, 10×5 et 5×10 l'on obtient (en affichant les lignes de bas en haut) :

```
27 26 25 24 23 22 21 20 19 18
28 57 56 55 54 53 52 51 50 17
29 58 79 78 77 76 75 74 49 16
30 59 80 93 92 91 90 73 48 15
31 60 81 94 99 98 89 72 47 14
32 61 82 95 96 97 88 71 46 13
33 62 83 84 85 86 87 70 45 12
34 63 64 65 66 67 68 69 44 11
35 36 37 38 39 40 41 42 43 10
00 01 02 03 04 05 06 07 08 09
```

```
17 16 15 14 13
18 37 36 35 12
19 38 49 34 11
20 39 48 33 10
21 40 47 32 09
22 41 46 31 08
23 42 45 30 07
24 43 44 29 06
25 26 27 28 05
00 01 02 03 04
```

```
22 21 20 19 18 17 16 15 14 13
23 42 41 40 39 38 37 36 35 12
24 43 44 45 46 47 48 49 34 11
25 26 27 28 29 30 31 32 33 10
00 01 02 03 04 05 06 07 08 09
```

16. Écrivez une fonction C de signature :

```c
struct point pos_spiral(size_t n, size_t m, size_t k)
```

qui calcule et renvoie les coordonnées du $k$ième point à être visité lors d'un parcours en spirale comme ci-dessus d'un tableau de dimensions $n \times m$.

17. Déduisez-en une fonction C de signature :

```
void
fill_square_spiral_from_pos
(size_t n, size_t m, int **pa)
```

qui initialise son argument « tableau » avec l'ordre de parcours donné par une telle spirale.

18. Testez *via* une fonction de test dédiée.

19. Écrivez une fonction C de signature :

```
void
fill_square_spiral_direct(size_t n, size_t m, int **pa)
```

de même spécifications que `fill_square_spiral_from_pos`, mais qui n'utilise pas la fonction `pos_spiral`. Cette fonction devra directement déduire la prochaine case à parcourir étant donné l'ensemble des cases ayant déjà été parcourues.

Tracé de courbes chevronnées

Tous les parcours implémentés ci-dessus ont la particularité d'être des courbes : le $k+1$ème élément du tableau à être visité est nécessairement un « voisin » du $k$ème ; ce sont même des courbes « à angle droit », dans le sens où ce $k+1$ème élément partage exactement un indice avec le $k$ème. Le but de cette (avant-?) dernière partie est de visualiser ces courbes en affichant pour chaque case d'un tableau un symbole indiquant la prochaine case à être visitée lors d'un parcours (s'il y en a une). Par exemple, on veut pouvoir produire les sorties suivantes pour certaines des courbes déjà implémentées :

```
* < < < < < < < < <
> > > > > > > > > ^
^ < < < < < < < < <
> > > > > > > > > ^
^ < < < < < < < < <
> > > > > > > > > ^
^ < < < < < < < < <
> > > > > > > > > ^
^ < < < < < < < < <
> > > > > > > > > ^
```

```
> v > v > v > v
^ > ^ v ^ > ^ v
^ < v < ^ < v <
> ^ > > > ^ > v
^ v < < v < < v
^ < > ^ > v ^ <
> v ^ < v < > v
^ > > ^ > > ^ *
```

```
v < < < < < < < < <
v v < < < < < < < ^
v v v < < < < < ^ ^
v v v v < < < ^ ^ ^
v v v v * < ^ ^ ^ ^
v v v > > ^ ^ ^ ^ ^
v v > > > > ^ ^ ^ ^
v > > > > > > ^ ^ ^
> > > > > > > > ^ ^
> > > > > > > > > ^
```

20. Écrivez une ou plusieurs fonctions permettant de produire les sorties ci-dessus *sans réimplémenter les parcours*.
On suggère pour cela d'écrire entre autres une fonction :

```
char **draw_path(size_t n, size_t m, int **pa)
```

qui crée et renvoie un « tableau » de *caractères* à deux dimensions (un caractère pour chaque case du tableau `pa`) qui pourront ensuite être affichés par une fonction d'affichage *ad hoc*. Les éléments d'un tel tableau sont donc de type `char` ; leurs littéraux s'écrivent entre « quotes » simples, comme par exemple : `'^'`, et ils peuvent être affichés individuellement *via* le spécifieur de conversion `"%c"`.

21. Dessinez !

Parcours du cavalier (si vous vous ennuyez…)

Un parcours de tableau à deux dimensions est dit *du cavalier* si chaque case à être visitée est accessible depuis la précédente en utilisant la règle de déplacement de la pièce du cavalier aux échecs. Un tel parcours est entre autres garanti d'exister si la plus petite dimension du tableau est au moins 5.

22. Comment pourriez vous implémenter la recherche d'un tel parcours ?

Nous verrons plus tard dans l'année une technique générale permettant de résoudre ce type de problème (qui fonctionne très bien pour celui-ci, mais n'est pas forcément efficace en général).