Module ITC313 - Informatique

Partie C / C++

TP1 + TP2 Exercices sur les tableaux et matrices, et Fonctions récursives

Benoît Darties - benoit.darties@u-bourgogne.fr
Université de Bourgogne

Année universitaire 2016-2017

1 Exercices sur des tableaux

Ces exercice de cette section permettent d'appliquer quelques algorithmes vues en cours sur le tri de tableaux. Pour réaliser ces TPs sans perdre de temps, il est fortement conseillé d'utiliser les fonctions d'affichage et de génération de tableaux aléatoires qui ont été réalisés dans le cadre du TD1. L'objectif ce cette premiere partie est de montrer la différence de complexité de certaines opérations selon que les tableaux sont triés ou non triés. Pour ce faire, nous travaillerons sur des tableaux de grande taille. Cependant, nous ne ferons pas un usage optimisé de la mémoire ici. Nous allons également survoler un aspect peu abordé en cours, qui est celui du découpage du code en plusieurs fichiers. Cette section est organisée en fil rouge : chaque réalisation dans un exercice vient compléter le travail réalisé dans l'exercice précédent, et reprend un peu plus vite les notions présentées dans l'exercice d'avant. Il est donc important de réaliser les exercices dans l'ordre et de bien comprendre les mécanismes présentés.

1.1 Fonctions sur les tableaux non triés

Exercice 1 : *algorithmes de parcours classiques sur tableau non triés*

Dans cet exercice, nous nous intéressons à des algorithmes de parcours classiques sur les tableaux non triés. Nous allons définir des fonctions de parcours, et organiser le code en plusieurs fichiers. Découper le code en plusieurs fichiers permet non seulement une meilleure lecture, mais améliore la vitesse de compilation.

1. Récupérez les fichiers `main.c`, `generationTableaux.h` et `generationTableaux.c` fournis avec l'exercice, placez les dans un même répertoire et ouvrez-les. Le code présenté dans ces trois fichiers correspond globalement au même code que celui présenté dans l'exercice 5 TD1, avec une répartition dans 3 fichiers. Observez la façon dont les fonctions propres à la génération de tableaux sont regroupées dans le fichier `generationTableaux.c`. Le prototype de chacune de ces fonctions est donné dans le fichier `generationTableaux.h` (qui est un fichier d'en-tête, il ne contient pas de code mais seulement des déclarations. Ce fichier est cependant nécessaire dans des situations plus compliquées, aussi pour respecter les bonnes pratiques et éviter des erreurs de compilation, les fichiers en-tête seront systématiquement donnés). Le code de la fonction principale `main()` est contenu dans le fichier `main.c`. Editez ensuite *uniquement* le fichier `generationTableaux.c`, et modifiez les fonctions de la même façon que vous l'avez fait dans le TD1 (quelques copier-coller suffiront)
2. Comme le code se situe désormais dans deux fichiers (les fichiers de type en-tête ne sont pas comptabilisés), utilisez la commande suivante pour compiler votre code et créer un exécutable nommé `exercice1` :

```
gcc main.c generationTableaux.c -o exercice1
```

Le code devrait normalement compiler correctement, et vous devriez vous retrouver avec un résultat similaire à celui obtenu en fin de l'exercice 5 du TD1.

3. Nous allons ajouter un ensemble de fonctions supplémentaires propres aux tableaux non triés. Afin de respecter le découpage du code en fichiers, nous allons ajouter ces fonctions dans un troisième fichier nommé `manipTableauxNonTries.c`. Pour vous aider, le squelette de ce fichier ainsi que son fichier en-tête `manipTableauxNonTries.h` vous sont également donnés. Récupérez ces fichiers et placez les dans le meme répertoire que les autres fichiers. Editez ensuite le fichier `main.c` et décommentez la ligne `#include "manipTableauxNonTries.h"`. Une fois que les nouvelles fonctions seront ajoutées, pour compiler le programme en incluant le fichier contenant ces dernières, nous utiliserons la commande :

   ```
   gcc main.c generationTableaux.c manipTableauxNonTries.c -o exercice1
   ```

   Avant de définir les fonctions, il est important de garder en tête que le type de parcours à faire sur le tableau (déterministe ou non déterministe) doit être optimisé. Par exemple, si l'on cherche à savoir si un tableau contient une valeur donnée, à partir du moment où cette valeur a été trouvée, on ne doit pas continuer à regarder les autres cellules du tableau et on peut de suite s'arrêter. Les boucles de type `for(;;;)` ne doivent être utilisées que pour des parcours déterministes. Les boucles `while()` peuvent être utilisées pour des parcours déterministes et non déterministes. La règle est la suivante : si une boucle `for` est annoncée pour `i` allant de 1 à 100, il faudra faire 100 tours de boucle et pas un de moins. Si la boucle est arrêtée avant, c'est que le programmeur a mal fait son travail et aurait du utiliser une boucle `while`.

   Les fonctions à définir dans le fichier `manipTableauxNonTries.c` sont les suivantes (note : à partir du moment où une fonction a été définie, elle peut être utilisée dans une autre si besoin) :

   (a) une fonction `sommeElementsTab()` qui prend en paramètres un tableau d'entiers `tab` et sa taille `taille`, et renvoie la somme des éléments contenus dans le tableau.

   (b) une fonction `sommeElementsTab()` qui prend en paramètres un tableau d'entiers `tab` et sa taille `taille`, et renvoie la valeur moyenne des éléments contenus dans le tableau.

   (c) une fonction `valeurContenueDansTabNonTrie()` qui prend en paramètres un tableau d'entiers `tab` et sa taille `taille`, ainsi qu'une valeur `val`, et renvoie 1 si le tableau contient au moins une occurence de la valeur `val`, et 0 sinon.

   (d) une fonction `nbOccurencesValeurDansTabNonTrie()` qui prend en paramètres un tableau d'entiers`tab` et sa taille `taille`, ainsi qu'une valeur `val`, et renvoie le nombre de fois que la valeur `val` est contenue dans le tableau.

4. Modifiez ensuite la fonction `main.c` pour tester vos nouvelles fonctions et vérifiez leur résultat.

Exercice 2 : *Ajout et suppression d'éléments sur tableaux non triés*

Dans cet exercice, nous nous intéressons aux algorithmes modifiant le nombre d'éléments d'un tableau non triés, mais dont l'espace mémoire ne peut varier.

Dans l'exercice précédent, nous avons travaillé sur des tableaux dont la taille est fixe, et toutes les cellules étaient affectées à une valeur. Lorsqu'on l'utilise dans un cas concret, un tableau possède en réalité 2 tailles :

— sa *taille maximale* qui est le nombre maximum de cellules du tableau. Cette taille n'est pas censée varier pour le tableau

— sa *taille effective* (que l'on appellera sa taille) qui est le nombre de cellules du tableau affectées à un tableau. Cette taille varie à chaque fois que des éléments sont ajoutés ou enlevés du tableau.

A titre d'exemple, supposons le code présenté dans le le bout de code `taille réelle et taille d'un tableau`, dans lequel on définit un tableau de 5 entiers. Nous stockons dans ce tableau 3 valeurs entières seulement :

Remarques / questions : benoit.darties@u-bourgogne.fr

Listing 1 – taille réelle et taille d'un tableau

```
1 int main() {
2   int Tab[5];     //  taille maximale = 5, taille effective = 0
3   Tab[0] = 4;     //  taille maximale = 5, taille effective = 1
4   Tab[1] = 89;    //  taille maximale = 5, taille effective = 2
5   Tab[2] = -34;   //  taille maximale = 5, taille effective = 3
6 }
```

Nous allons définir dans cet exercice des fonctions qui permettent d'ajouter ou de supprimer des éléments à un tableau. Ces fonctions viennent compléter les fonctions de l'exercice 1. Pour cela il sera souvent nécessaire de connaitre non seulement la taille maximale du tableau, mais aussi le nombre d'élements réels du tableau. Pour vous aider, les fonctions à définir sont à insérer dans le fichier squelette `modifTableauxNonTries.c` qu'il vous faut récupérer ainsi que le fichier d'en-tete associé.

1. Récupérez les fichiers `modifTableauxNonTries.c` et `modifTableauxNonTries.h` et placez-les dans un même répertoire. Ajoutez dans le même répertoire les fichiers que vous avez obtenus à la fin de l'exercice 1. Modifiez ensuite le fichier `main.c` pour ajouter la directive `#include "modifTableauxNonTries.h"` en dessous de la ligne `#include "manipTableauxNonTries.h"`.
2. En étendant le raisonnement vu dans l'exercice 1, quelle serait alors la commande à écrire pour pouvoir compiler un programme en y incluant les fonctions qui seront définies dans le fichier textttmodifTableauxNonTries.c ?
3. Editez le fichier `modifTableauxNonTries.c` et ajoutez-y les fonctions suivantes :
   (a) une fonction `ajoutValeurTabNonTrie()` qui prend en paramètres un tableau d'entier `tab`, sa taille maximum `tailleMax`, le nombre réel d'éléments qu'il contient `taille`, et une valeur à ajouter `val`. Cette fonction essaye d'ajouter `val` au tableau tab (par exemple à la fin du tableau), et retourne 1 si l'ajout a pu être correctement réalisé, et 0 sinon.
   (b) une fonction `supprimeValeurTabNonTrie()` qui prend en paramètres un tableau d'entier `tab`, le nombre réel d'éléments qu'il contient `taille`, et une valeur à supprimer `val`. Cette fonction essaye de supprimer la première occurence de `val` rencontrée dans le tableau tab, et retourne 1 si la suppression a pu être correctement réalisée (le tableau contenait cette valeur), et 0 sinon. Pour supprimer une telle valeur dans un tableau non trié, une astuce consiste à chercher l'indice de la première cellule contenant la valeur à supprimer, à inverser cette valeur avec la valeur de la dernière cellule, et considérer que la taille du tableau sera diminuée de 1. L'inversion des valeurs de deux cellules n'a pas d'incidence ici puisque le tableau n'est pas censé être trié.
   (c) une fonction `supprimeToutesLesValeursTabNonTrie()` qui prend en paramètres un tableau d'entier `tab`, le nombre réel d'éléments qu'il contient `taille`, et une valeur à supprimer `val`. Cette fonction essaye de supprimer toutes les occurences de `val` rencontrée dans le tableau tab, et retourne le nombre d'éléments qui ont été supprimés et 0 sinon.
4. Modifiez ensuite la fonction `main.c` pour tester vos nouvelles fonctions en ajoutant et supprimant des valeurs, et vérifiez leur résultat. N'oubliez pas que la taille du tableau va varier et qu'elle doit être toujours mise à jour. Pour cela, vous pourrez vous inspirer de l'exemple présenté ci-après `"variation de la taille effective d'un tableau"`, dans lequel la taille du tableau est mise à jour en fonction de la valeur retournée par chacune des fonctions appelées.

Remarques / questions : benoit.darties@u-bourgogne.fr

Listing 2 – variation de la taille effective d'un tableau

```
#define TAILLEMAX = 5;
int main() {

  int Tab[TAILLEMAX];    //  taille maximale = 5, taille effective = 0
  int taille = 0;
  int resultat;

  // tentative d'ajout de la valeur 4
  resultat = ajoutValeurTabNonTrie(Tab, taille, TAILLEMAX, 4) ;
  if (resultat ==1) {
    taille = taille+1; // ajout correctement effectue
  }
  // ...
  // suppression de la valeur 8
  resultat = supprimeValeurTabNonTrie(Tab, taille, 8) ;
  if (resultat ==1) {
    taille = taille-1; // suppression correctement effectuee
  }
  // ...
  // suppression de toutes les valeurs 12
  resultat = supprimeToutesLesValeursTabNonTrie(Tab, taille, 12) ;
  if (resultat >0) {
    taille = taille-resultat; // suppressions correctement effectuees
  }

}
```

1.2 Algorithmes de tri de tableaux

Nous avons à disposition les éléments nécessaires pour créer des tableaux remplis de valeurs aléatoires, et pour ajouter et supprimer des valeurs dans des tableaux non triés. Certains algorithmes, notamment de recherche et de suppression de valeur, pourraient cependant être bien améliorés si l'on utilisait des tableaux triés. L'objet de cette section est tout d'abord de définir des méthodes permettant de trier des tableaux. Les fonctions d'ajout et de suppression de valeurs dans un tableau en le maintenant trié seront vus plus tard.

Exercice 3 : *Trier des tableaux aléatoires*

Dans cet exercice, nous allons écrire en C deux fonctions de tri de tableau, à partir des algorithmes de ces méthodes de tri. Dans cet exercice, nous allons nous baser sur les travaux réalisés dans les exercices précédents, en ajoutant de nouvelles fonctions dans un nouveau fichier, de la même façon que nous l'avons fait dans l'exercice 2

1. Récupérez les fichiers squelette `algosTri.c` et `algosTri.h` et placez-les dans le même répertoire que les fichiers que vous avez obtenus à la fin de l'exercice 2. Modifiez ensuite le fichier `main.c` pour ajouter la directive `#include "algosTri.h"`, et adaptez le ligne de commande pour pouvoir compiler et utiliser correctement les fonctions de tri qui seront définies dans le fichier `algosTri.c`.
2. Définissez dans le fichier `algosTri.c` les deux fonctions suivantes :
   (a) La fonction `triAbulle()` qui prend en paramètre un tableau d'entiers et sa taille (effective), et trie les éléments du tableau selon l'algorithme de tri *tri à bulle*. Le principe de cet algorithme est rappelé ci-après. Il vous revient de définir les variables intermédiaires au bon endroit, ou d'adapter les valeurs en fonction du cas de figure.

Algorithm 1: Algorithme Tri A bulles

```
Données: tab : tableau d'entiers ; taille : entier
début
    entier i;
    entier j;
    entier tmp;
    pour i ∈ [0, taille − 1] faire
        pour j ∈ [0, taille − 2] faire
            si tab[j] < tab[j + 1] alors
                tmp ← tab[j] ;
                tab[j] ← tab[j + 1] ;
                tab[j + 1] ← tmp ;
```

(b) La fonction `triSelection()` qui prend en paramètre un tableau d'entiers et sa taille (effective), et trie les éléments du tableau selon l'algorithme de tri *tri par sélection*. Le principe de cet algorithme est rappelé ci-après. Il vous revient de définir les variables intermédiaires au bon endroit, ou d'adapter les valeurs en fonction du cas de figure.

Algorithm 2: Algorithme Tri Par Selection

```
Données: Tab : tableau d'entiers ; taille : entier
début
    entier i;
    entier j;
    entier tmp;
    pour i ∈ [0, taille − 2] faire
        pour j ∈ [i, taille − 1] faire
            si Tab[j] < Tab[i] alors
                temp ← Tab[i] ;
                Tab[i] ← Tab[j] ;
                Tab[j] ← tmp ;
```

3. Modifiez ensuite la fonction `main.c` pour tester vos nouvelles fonctions en créant quelques tableaux aléatoires, en les triant, et vérifiez leur résultat avec les fonctions d'affichage du tableau.

1.3 Fonctions sur les tableaux triés

Dans la première section de cette partie, nous avons défini des algorithmes de manipulation et modification de tableaux non triés. Puis nous avons défini des fonctions de tri sur ces tableaux. Désormais, nous revenons sur les algorithmes de manipulation et modification, mais sur des tableaux triés. En effet, la particularité de manipuler des tableaux triés fait que certaines fonction (notamment la recherche d'éléments) vont être simplifées. Mais d'autres (notamment l'ajout ou suppression de valeurs) vont être plus complexes, car il faudra veiller à ce que le tableau reste ordonné.

Exercice 4 : *algorithmes de parcours classiques sur tableau non triés*

Dans cet exercice, nous nous intéressons à des algorithmes de parcours classiques sur les tableaux triés. Il s'agit d'une reprise des questions de l'exercice 1, adaptées aux tableaux triés.

1. Nous allons ajouter un ensemble de fonctions supplémentaires propres à la recherche de valeurs sur les tableaux triés. Récupérez les fichiers squelette `manipTableauxTries.c` et

Remarques / questions : benoit.darties@u-bourgogne.fr

`manipTableauxTries.c` et placez-les dans le même répertoire que les fichiers que vous avez obtenus à la fin de l'exercice 3. Modifiez ensuite le fichier `main.c` pour ajouter la directive `#include "manipTableauxTries.h"`, et adaptez le ligne de commande pour pouvoir compiler et utiliser correctement les fonctions qui seront définies dans le fichier `manipTableauxTries.c`.

2. Notez que nous ne redéfinirons pas les fonctions `sommeElementsTab()` et `moyenneValeursTab()` puisque ces dernières sont inchangées que le tableau soit trié ou non. Evidemment, toutes les fonctions de recherche d'élément qui marchent sur des tableaux non triés marchent également sur les tableaux triés. Mais dans le cas des tableaux triés, on peut améliorer grandement la complexité de ces fonctions, comme vu en cours. Les fonctions à définir dans le fichier `manipTableauxTries.c` sont les suivantes :
   (a) une fonction `valeurContenueDansTabTrie()` qui prend en paramètres un tableau d'entiers `tab` et sa taille `taille`, ainsi qu'une valeur `val`, et renvoie 1 si le tableau contient au moins une occurence de la valeur `val`, et 0 sinon. L'algorithme modifié fonctionne par recherche dichotomique, comme vu en cours : on définit les indices min et max de recherche de la valeur dans le tableau (respectivement initialisés à 0 et `taille`), et on regarde la valeur contenue dans la case d'indice médian ((min + max) /2). Selon que la valeur lu est plus grande, plus petite ou égale, on fait varier la valeur de l'indice min, ou max, ou on conclut.
   (b) une fonction `nbOccurencesValeurDansTabTrie()` qui prend en paramètres un tableau d'entiers`tab` et sa taille `taille`, ainsi qu'une valeur `val`, et renvoie le nombre de fois que la valeur `val` est contenue dans le tableau. La version optimisée consiste à rechercher une valeur, comme vu précédemment. Si cette valeur existe, on regarde simplement le nombre de cellules à gauche et à droite contenant cette valeur.
3. Modifiez ensuite la fonction `main.c` pour tester vos nouvelles fonctions et vérifiez leur résultat.

Exercice 5 : *Ajout et suppression d'éléments sur tableaux triés*

Dans cet ultime exercice, nous nous intéressons aux algorithmes d'ajout et suppression de valeur sur les tableaux triés. Il s'agit d'une reprise des questions de l'exercice 2, adaptées aux tableaux triés.

1. Nous allons ajouter un ensemble de fonctions supplémentaires propres à l'ajout et la suppression de valeur sur les tableaux triés. Récupérez les fichiers squelette `modifTableauxTries.c` et `modifTableauxTries.c` et placez-les dans le même répertoire que les fichiers que vous avez obtenus à la fin de l'exercice 3. Modifiez ensuite le fichier `main.c` pour ajouter la directive `#include "modifTableauxTries.h"`, et adaptez le ligne de commande pour pouvoir compiler et utiliser correctement les fonctions qui seront définies dans le fichier `modfTableauxTries.c`.
2. Afin de respecter l'invariant selon lequel un tableau trié doit resté trié après ajout et suppression de valeurs, les fonctions déjà définies sur les tableaux non triés ne marchent plus ici et doivent être redéfines. Dans le fichier `modifTableauxTries.c`, définissez les fonctions suivantes :
   (a) une fonction `ajoutValeurTabTrie()` qui prend en paramètres un tableau d'entier `tab` trié, sa taille maximum `tailleMax`, le nombre réel d'éléments qu'il contient `taille`, et une valeur à ajouter `val`. Cette fonction essaye d'ajouter `val` au tableau tab en respectant le tri (par exemple à la fin du tableau), et retourne 1 si l'ajout a pu être correctement réalisé, et 0 sinon. L'algorithme consiste à décaler les valeurs de 1 à droite en partant de la fin, jusqu'à trouver l'endroit où insérer la valeur.
   (b) une fonction `supprimeValeurTabTrie()` qui prend en paramètres un tableau d'entier `tab` trié, le nombre réel d'éléments qu'il contient `taille`, et une valeur à supprimer `val`. Cette fonction essaye de supprimer la première occurence de `val` rencontrée dans le tableau tab, et retourne 1 si la suppression a pu être correctement réalisée (le tableau contenait cette valeur), et 0 sinon. Pour supprimer une telle valeur dans un tableau trié, une astuce consiste à chercher l'indice de la première cellule contenant la valeur

à supprimer (par exemple en utilisant la recherche dichotomique), puis à décaler les valeurs successives de un vers la gauche jusqu'à la fin du tableau.

(c) une fonction `supprimeToutesLesValeursTabTrie()` qui prend en paramètres un tableau d'entier `tab` trié, le nombre réel d'éléments qu'il contient `taille`, et une valeur à supprimer `val`. Cette fonction essaye de supprimer toutes les occurences de `val` rencontrée dans le tableau tab, et retourne le nombre d'éléments qui ont été supprimés et 0 sinon. Cet algorithme est le moins facile : il consiste à trouver une occurence au moyen d'une recherche dichotomique, puis se positionner sur celle la plus à gauche, compter le nombre d'occurences de cette valeur (contenues dans les cellules immédiatement suivantes), et décaler les valeurs suivantes du nombre d'occurences calculé.

3. Modifiez ensuite la fonction `main.c` pour tester vos nouvelles fonctions en ajoutant et supprimant des valeurs, et vérifiez leur résultat. N'oubliez pas que la taille du tableau va varier et qu'elle doit être toujours mise à jour.

2 Fonctions récursives

une fonction récursive est une fonction qui s'appelle elle-même. Dans leur utilisation, les fonctions récursives utilisent plus de mémoire, mais se montrent très puissantes pour effectuer certaines opérations.

Exercice 6 : *Definition de fonction recursive*

Dans cet exercice, il vous est demandé de définir un algorithme récursif permettant de calculer deux fonctions mathématiques pour lesquelles vous avez déjà définis des algorithmes itératifs précédemment. Il vous est conseillé d'écrire dans un premier temps la fonction sous forme récursive réduite, en exprimant une valeur de départ, et d'exprimer la fonction à l'indice n selon sa valeur à un indice inférieur, par exemple $n-1$, puis d'écrire le pseudo-code associé

Dans cet exercice, nous présentons tout d'abord un exemple de calcul de la fonction `factorielle()`. La fonction `factorielle()` d'un nombre n se définit ainsi :

$$factorielle(n) \quad = \quad \prod_{k=1}^{n} k \quad = \quad 1 * 2 * 3 * \cdots * n$$

Cette fonction peut également s'exprimer mathématiquement de la façon récursive suivante :

$$\begin{aligned} factorielle(n) &= 1 \qquad \text{si n est égal à 1} \\ &= factorielle(n-1) * n \qquad \text{si n est strictement supérieur à 1} \end{aligned}$$

Dans le fichier `mainFactorielle.c`, les fonctions `factorielle()` et `factorielleRecursif()` sont déduites à partir des équations précédentes.

1. Examinez le fichier `mainFactorielle.c`, compilez-le et vérifiez que le programme fonctionne correctement.
2. Proposez un programme itératif nommé `harmonique` et un programme récursif nommé nommé `harmoniqueRecursif` prenant chacun un paramètre n de type entier, et permettant de calculer la valeur d'un nombre harmonique de rang n. Le nombre harmonique H_n pour un entier $n \geq 1$ donné est égal à :

$$H_n \quad = \quad \sum_{k=1}^{n} \frac{1}{k} \quad = \quad 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \tag{1}$$

Pour vous aider, le squelette de ces fonctions et de la fonction `main()` est proposé dans le fichier `mainHarmonique.c` qu'il vous suffit juste de remplir, compiler et exécuter. Il vous est également conseillé d'écrire la fonction sous forme récursive mathématique avant d'écrire le code.

Remarques / questions : benoit.darties@u-bourgogne.fr

3. Proposez un programme itératif nommé `piLeibniz` et un programme récursif nommé nommé `piLeibnizRecursif` prenant chacun un paramètre n de type entier, et permettant de calculer*une approximation du nombre* π, obtenue au moyen de la formulation de Leibniz. Pour rappel, la fonction de Leibniz est la suivante :

$$4.\sum_{k=0}^{n}\frac{(-1)^k}{2k+1} \quad = \quad \frac{4}{1}-\frac{4}{3}+\frac{4}{5}-\frac{4}{7}+\frac{4}{11}-\cdots+\frac{4}{2n+1} \tag{2}$$

   Pour vous aider, le squelette de ces fonctions et de la fonction `main()` est proposé dans le fichier `mainPiLeibniz.c` qu'il vous suffit juste de remplir, compiler et exécuter. Il vous est également conseillé d'écrire la fonction sous forme récursive mathématique avant d'écrire le code. La fonction `pow()` permet de calculer x^y pour x et y passés en paramètres.

Exercice 7 : *Algorithme recursif sur matrice*

Cet exercice présente un algorithme récursif sur matrice. Il illustre un cas de figure dans lequel l'utilisation de la récursivité permet de fournir une solution plus simple que l'utilisation d'un algorithme séquentiel.

Dans cet exercice (inspiré du jeu *bubble breaker*), on considère une matrice `M` de dimensions `m x n` dont les valeurs des cellules sont soit 0, soit 1. Un exemple de matrice utilisée dans cet exercice (de dimensions 9 par 15) est donné ci-après :

```
   | 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
-----------------------------------------------
 0 | 1  1  1  0  1  0  0  1  0  1  1  0  0  0  1
 1 | 1  0  0  1  1  1  0  0  0  1  1  1  1  1  0
 2 | 1  0  0  0  1  0  1  1  0  0  1  1  0  1  1
 3 | 1  1  1  0  1  0  0  1  0  1  1  0  0  0  1
 4 | 1  1  0  0  0  0  1  0  1  0  0  0  1  0  0
 5 | 1  0  0  0  0  0  0  0  0  1  1  0  1  0  0
 6 | 1  0  0  1  0  1  0  0  1  0  0  1  0  1  1
 7 | 0  1  1  1  0  0  0  1  1  1  0  1  0  1  0
 8 | 1  1  0  1  1  0  0  0  1  0  0  1  1  1  0
```

Dans cette matrice, deux cellules sont dites "adjacentes" si elles sont côte à côte et situées sur une même ligne ou même colonne. Par exemple, les cellules de coordonnées [6][5] et [7][5] sont adjacentes. On appellera *zone* un ensemble connexe maximal de cellules adjacentes. Par exemple, les cellules ayant les coordonnées suivantes forment une zone : `[6][14], [6][13], [7][13], [8][13], [8][12], [8][11], [7][11] et [6][11]`. Si on considère la cellule d'indice `[7][3]`, cette dernière est contenue dans une zone de 8 cellules contenant également la cellule `[8][0]`. Chaque cellule appartient à une zone exactement. On peut dont désigner une zone simplement par les coordonnées d'une cellule contenue dans cette zone. Afin de gagner du temps, le programme `bubbleBreaker.c` vous est donné. Il contient une matrice de test 9 x 15 remplie comme sur l'exemple ci-dessus, une fonction d'affichage, et le squelette de la fonction à écrire. Seul le code de la fonction est à modifier sur ce programme.

1. Proposez un algorithme récursif `remplir` qui prend en entrée une une matrice `M globale` [1] de dimensions `m x n`, ainsi que les coordonnées `[x][y]` d'une cellule quelconque. L'algorithme doit affecter la valeur 2 à chacune des cellules de la même zone que la cellule `[x][y]`. Les cellules des autres zones ne doivent pas avoir leur valeur modifiée. L'algorithme ne retournera rien, car la matrice d'entrée sera modifiée en direct. Pour vous aider, on vous donne l'idée générale de l'algorithme :

1. Par soucis de simplicité, on considère ici que tous les algorithmes récursifs vont travailler sur la même matrice, et pas sur des copies créées à chaque appel de l'algorithme

- Si la cellule `[x][y]` contient déjà la valeur 2, il n'y a rien a faire.
- Sinon, on enregistre la valeur dans une variable `temp`, on change la valeur de la cellule à 2, et on regarde chacune des cellules autours de coordonnées `[x+1][y]`, `[x-1][y]`, `[x][y-1]` et `[x][y+1]` . Quand une de ces coordonnées correspond à une cellule existante (dans les bornes de la matrice) et que sa valeur est égale à celle contenue dans `temp`, alors on appelle l'algorithme récursif en modifiant seulement la coordonnée en entrée.

2. Pourquoi est-il nécessaire de *d'abord* stocker la valeur de la cellule dans `temp` et modifier la valeur en 2, *avant* d'appeler la fonction récursive ?

Remarques / questions : benoit.darties@u-bourgogne.fr